ASP.NET documentation

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

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

GET STARTED
Create your first web UI

OVERVIEW
ASP.NET Core overview

DOWNLOAD
Download .NET

GET STARTED
Create your first web API

GET STARTED
Create your first real-time web app

Develop ASP.NET Core apps

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

HTTP API apps

Develop HTTP services with ASP.NET Core

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

Interactive client-side Blazor apps

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

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

Page-focused web UI with Razor Pages

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

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

Data-driven web apps

Page-focused web UI with MVC

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

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

Create data-driven web apps in ASP.NET Core

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

Real-time web apps with SignalR

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

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

Remote Procedure Call (RPC) apps - gRPC services

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

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

Previous ASP.NET framework versions

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

- ASP.NET 4.x

ASP.NET Core video tutorials

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

Concepts and features

API reference for ASP.NET Core

.NET API browser

Servers

Overview

Kestrel

IIS

HTTP.sys

Host and deploy

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

Security and identity

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

Globalization and localization

- Overview
- Portable object localization
- Localization extensibility
- Troubleshoot

Test, debug and troubleshoot

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

Azure and ASP.NET Core

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

Performance

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

Advanced features

- Model binding

Migration

- ASP.NET Core 5.0 to 6.0

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

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

Architecture

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

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

Overview of ASP.NET Core

Article • 06/18/2024

By Daniel Roth, Rick Anderson, and Shaun Luttin

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

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

With ASP.NET Core, you can:

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

Why choose ASP.NET Core?

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

ASP.NET Core provides the following benefits:

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

- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

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

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

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

Client-side development

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

ASP.NET Core target frameworks

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

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

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

Recommended learning path

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

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

Expand table

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

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

Expand table

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

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

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

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

Migrate from .NET Framework

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

How to download a sample

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

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

Preprocessor directives in sample code

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

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

```
C#
```

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

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

C#

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

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

Breaking changes and security advisories

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

Next steps

For more information, see the following resources:

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

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

Article • 04/10/2024

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

ASP.NET Core

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

ASP.NET Core provides the following benefits:

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

ASP.NET 4.x

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

Framework selection

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

Expand table

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

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

ASP.NET Core scenarios

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

ASP.NET 4.x scenarios

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

Additional resources

- Introduction to ASP.NET

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

.NET vs. .NET Framework for server apps

Article • 11/22/2024

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

Expand table

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

Choose .NET

.NET has the following advantages for server apps:

- **Works cross-platform.**

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

- **Lets you target microservices.**

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

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

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

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

- **Supports Docker containers.**

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

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

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

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

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

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

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

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

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

- **Is more secure.**

When to choose .NET Framework

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

- Your app currently uses .NET Framework.

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

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

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

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

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

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

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

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

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

See also

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

Tutorial: Get started with ASP.NET Core

Article • 09/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

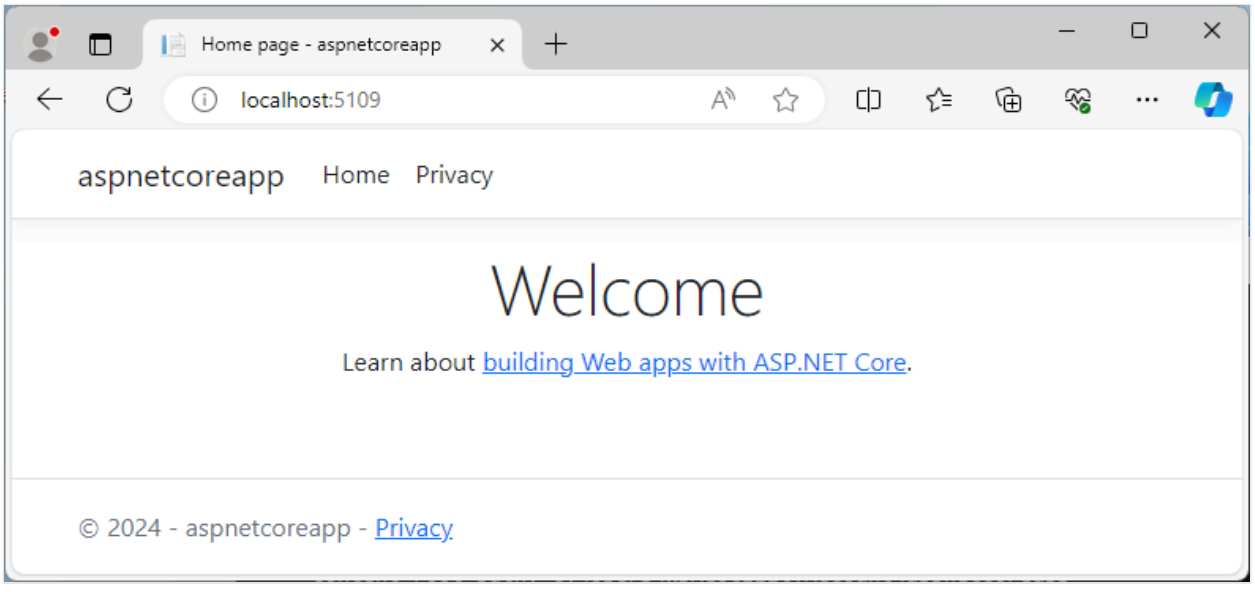
This tutorial shows how to create and run an ASP.NET Core web app using the .NET CLI.

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

You'll learn how to:

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

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



Prerequisites

.NET 8.0 SDK

Create a web app project

Open a command shell, and enter the following command:

.NET CLI

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

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

Run the app

Run the following commands:

.NET CLI

```
cd aspnetcoreapp
dotnet run
```

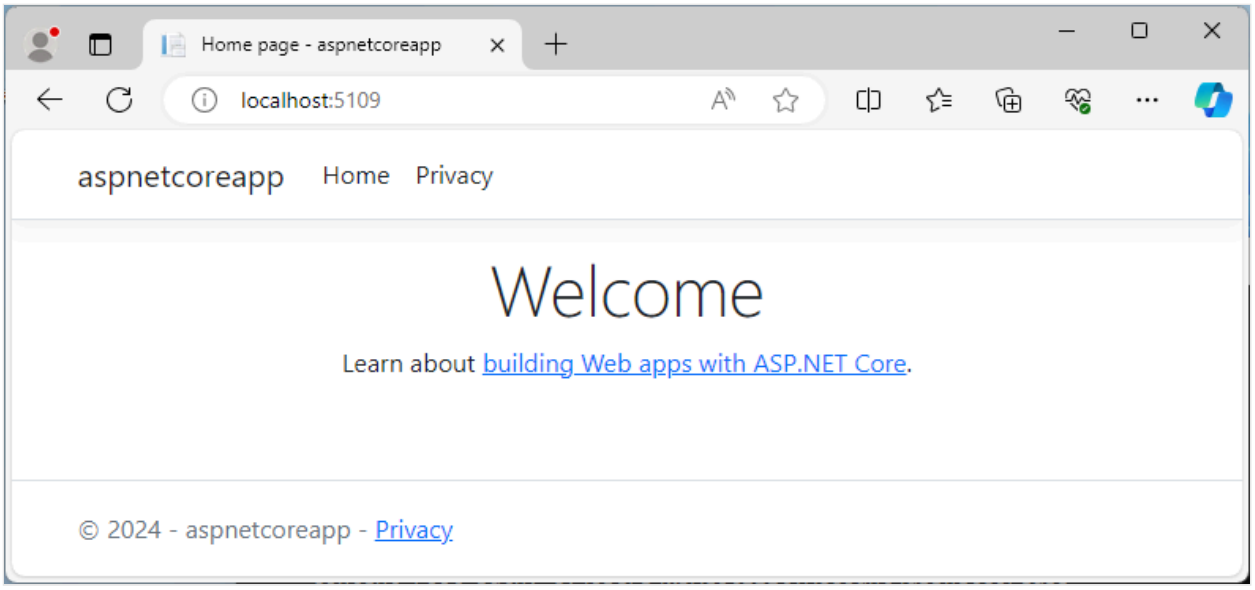
The `run` command produces output like the following example:

Output

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

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

The browser shows the home page.



Edit a Razor page

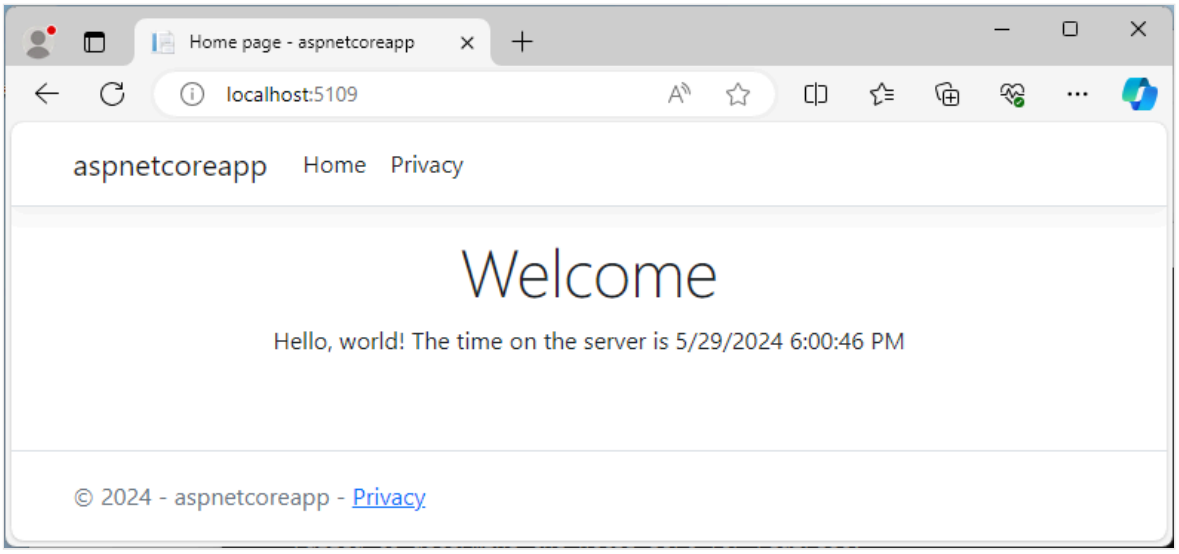
Change the home page:

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

CSHTML

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

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

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



Next steps

In this tutorial, you learned how to:

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

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

Overview of ASP.NET Core

What's new in ASP.NET Core 9.0

Article • 11/19/2024

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

Static asset delivery optimization

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

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

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

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

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

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

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

```
diff
```

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

builder.Services.AddRazorPages();

var app = builder.Build();

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

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

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

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

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

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

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

Expand table

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

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

Expand table

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

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

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

Expand table

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

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

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

- Static files in ASP.NET Core

- ASP.NET Core Blazor static files

Enabling dynamic compression on the server vs using `MapStaticAssets`

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

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

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

Expand table

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

Blazor

This section describes new features for Blazor.

.NET MAUI Blazor Hybrid and Web App solution template

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

Key features of this template include:

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

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

.NET CLI

```
dotnet workload install maui
```

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

.NET CLI

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

The template is also available in Visual Studio.

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

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

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

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

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

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

Improved server-side reconnection experience:

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

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

  JavaScript

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

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

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

Simplified authentication state serialization for Blazor Web Apps

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

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

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

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

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

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

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

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

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

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

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

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

razor

```
@attribute [ExcludeFromInteractiveRouting]
```

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

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

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

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

razor

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

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

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

```
null;
}
```

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

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

Constructor injection

Razor components support constructor injection.

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

C#

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

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

Websocket compression for Interactive Server components

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

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

C#

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

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

C#

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

For more information, see the following resources:

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

Handle keyboard composition events in Blazor

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

Added `OverscanCount` parameter to `QuickGrid`

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

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

razor

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

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

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

razor

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

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

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

    private void Submit() {}

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

New enhanced navigation events

Trigger JavaScript callbacks either before or after enhanced navigation with new event listeners:

- `blazor.addEventListener("enhancednavigationstart", {CALLBACK})`
- `blazor.addEventListener("enhancednavigationend", {CALLBACK})`

For more information, see ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR).

SignalR

This section describes new features for SignalR.

Polymorphic type support in SignalR Hubs

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

C#

```
public class MyHub : Hub
{
    public void Method(JsonPerson person)
    {
        if (person is JsonPersonExtended)
        {
        }
        else if (person is JsonPersonExtended2)
        {
        }
        else
        {
        }
    }
}

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

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

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

Improved Activities for SignalR

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

.NET SignalR server ActivitySource

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

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

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

XML

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

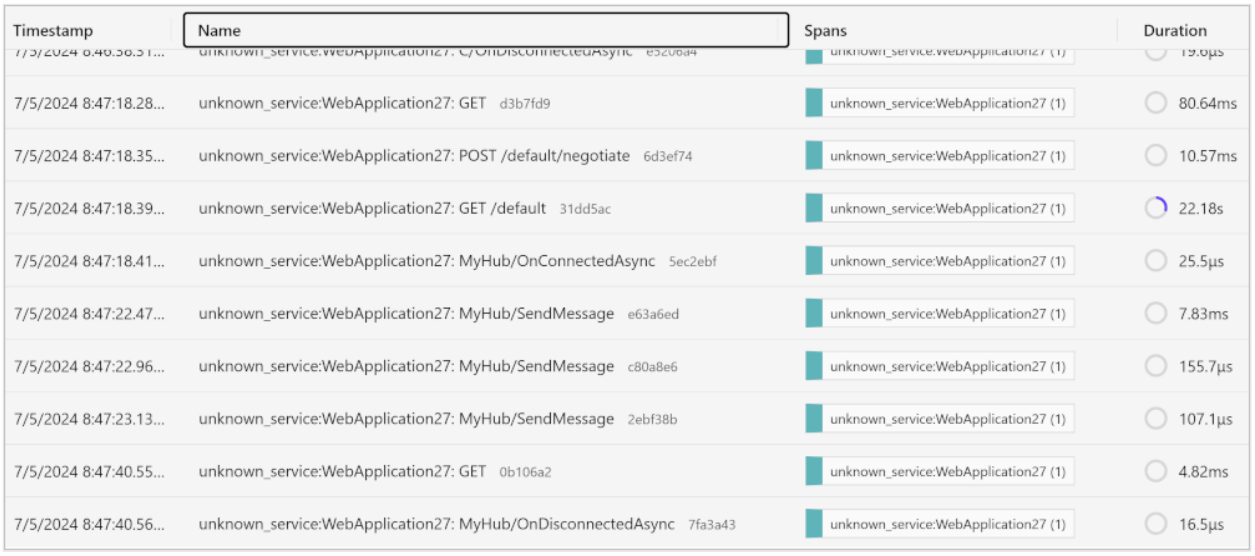
Add the following startup code to the `Program.cs` file:

C#

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

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

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

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

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

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

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

The following is example output from the Aspire Dashboard:

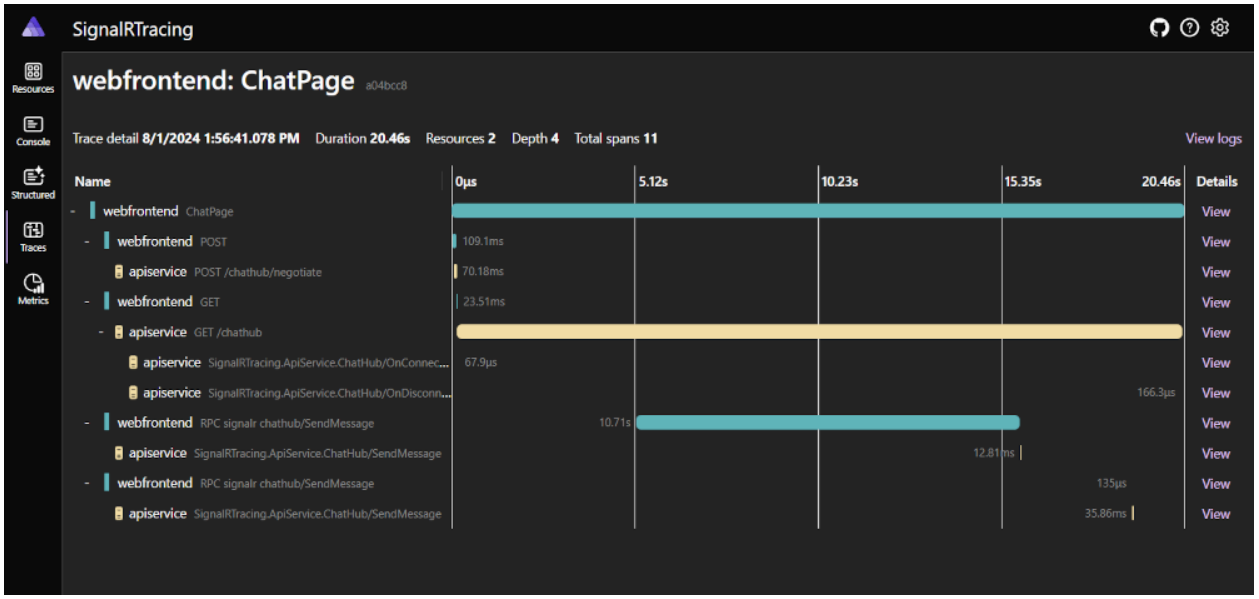


.NET SignalR client ActivitySource

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

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

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



SignalR supports trimming and Native AOT

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

Getting started

Install the latest .NET 9 SDK.

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

.NET CLI

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

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

C#

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

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.AddSignalR();
builder.Services.Configure<JsonHubProtocolOptions>(o =>
```

```
{
    o.PayloadSerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

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

    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/8.0.7/signalr.min.js"></script>
    <script>
        const connection = new signalR.HubConnectionBuilder()
            .withUrl("/chatHub")
            .build();

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

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

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

app.Run();

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

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

```
    }
}
```

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

Limitations

- Only the JSON protocol is currently supported:
  - As shown in the preceding code, apps that use JSON serialization and Native AOT must use the `System.Text.Json` Source Generator.
  - This follows the same approach as minimal APIs.
- On the SignalR server, Hub method parameters of type `IAsyncEnumerable<T>` and `ChannelReader<T>` where `T` is a ValueType (`struct`) aren't supported. Using these types results in a runtime exception at startup in development and in the published app. For more information, see SignalR: Using IAsyncEnumerable<T> and ChannelReader<T> with ValueTypes in native AOT (dotnet/aspnetcore #56179).
- Strongly typed hubs aren't supported with Native AOT (`PublishAot`). Using strongly typed hubs with Native AOT will result in warnings during build and publish, and a runtime exception. Using strongly typed hubs with trimming (`PublishedTrimmed`) is supported.
- Only `Task`, `Task<T>`, `ValueTask`, or `ValueTask<T>` are supported for async return types.

Minimal APIs

This section describes new features for minimal APIs.

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

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

```
C#
```

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

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

app.Run();
```

Call `ProducesProblem` and `ProducesValidationProblem` on route groups

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

C#

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

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

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

app.Run();

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

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

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

C#

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

app.MapGet("/", () =>
{
```

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

Thanks to GitHub user joegoldman2 for this contribution!

OpenAPI

This section describes new features for OpenAPI

Built-in support for OpenAPI document generation

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

The following highlighted code calls:

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

C#

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

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();

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

app.Run();
```

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

.NET CLI

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

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

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

.NET CLI

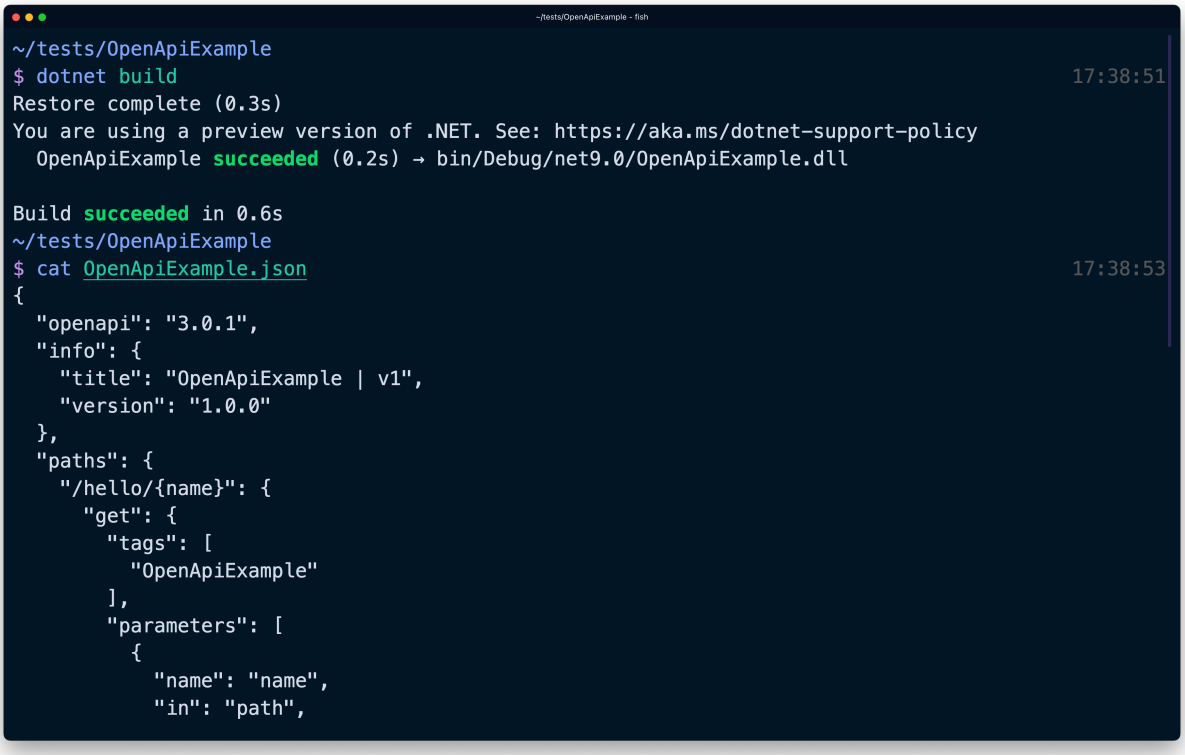
```
dotnet add package Microsoft.Extensions.ApiDescription.Server --prerelease
```

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

XML

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

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



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

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

Microsoft.AspNetCore.OpenApi supports trimming and Native AOT

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

Get started

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

```
Console

dotnet new webapiaot
```

Add the Microsoft.AspNetCore.OpenAPI package.

```
Console

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

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

```
Console

dotnet add package Microsoft.OpenApi
```

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

```
diff

+ builder.Services.AddOpenApi();

var app = builder.Build();

+ app.MapOpenApi();
```

Publish the app.

```
Console

dotnet publish
```

The app publishes using Native AOT without warnings.

Authentication and authorization

This section describes new features for authentication and authorization.

OpenIdConnectHandler adds support for Pushed Authorization Requests (PAR)

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

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

For .NET 9, we have decided to enable PAR by default if the identity provider's discovery document advertises support for PAR, since it should provide enhanced security for providers that support it. The identity provider's discovery document is usually found at

`.well-known/openid-configuration`. If this causes problems, you can disable PAR via OpenIdConnectOptions.PushedAuthorizationBehavior as follows:

C#

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

        // The default value is PushedAuthorizationBehavior.UseIfAvailable.

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

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

OIDC and OAuth Parameter Customization

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

C#

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

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

C#

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

Configure HTTP.sys extended authentication flags

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

C#

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

Miscellaneous

The following sections describe miscellaneous new features.

New `HybridCache` library

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

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

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

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

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

C#

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

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

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

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

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

XML

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

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

C#

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

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

C#

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

We provide a concrete implementation of the `HybridCache` abstract class via dependency injection, but it's intended that developers can provide custom implementations of the API. The `HybridCache` implementation deals with everything related to caching, including concurrent operation handling. The `cancel` token here represents the combined cancellation of *all* concurrent callers—not just the cancellation of the caller we can see (that is, `token`).

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

C#

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

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

A note on object reuse

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

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

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

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

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

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

Other `HybridCache` features

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

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

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

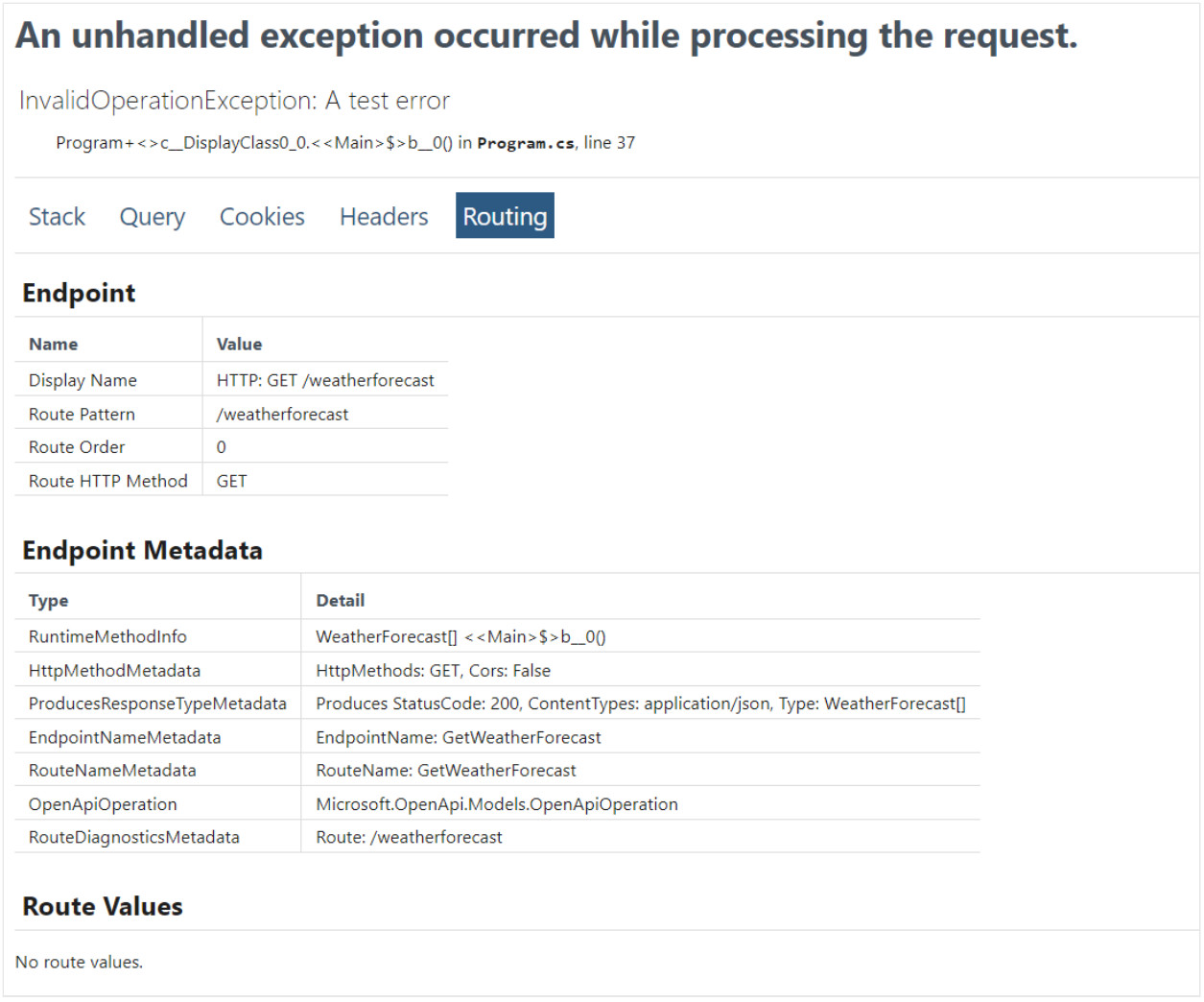
`HybridCache` supports older .NET runtimes, down to .NET Framework 4.7.2 and .NET Standard 2.0.

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

Developer exception page improvements

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

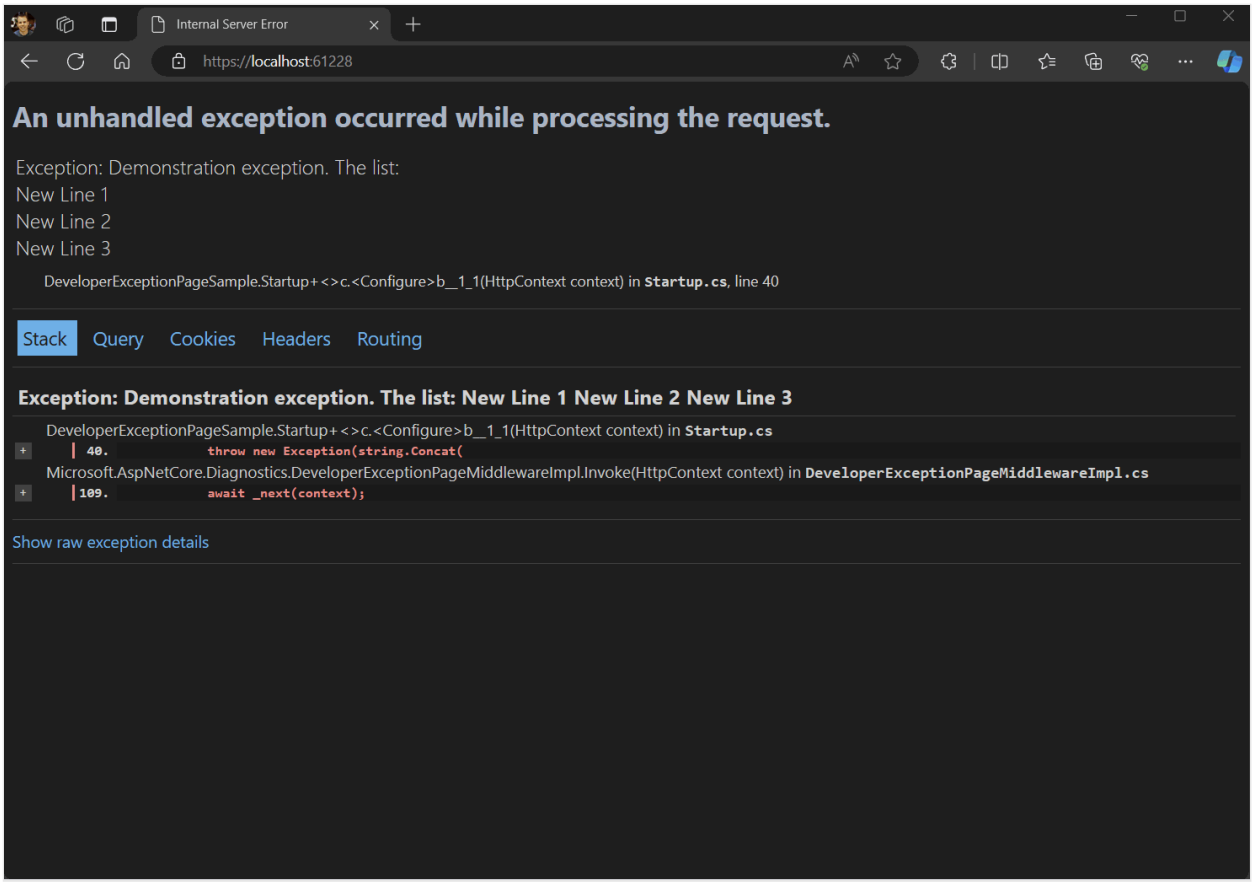
Preview 3 added endpoint metadata to the developer exception page. ASP.NET Core uses endpoint metadata to control endpoint behavior, such as routing, response caching, rate limiting, OpenAPI generation, and more. The following image shows the new metadata information in the `Routing` section of the developer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



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

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

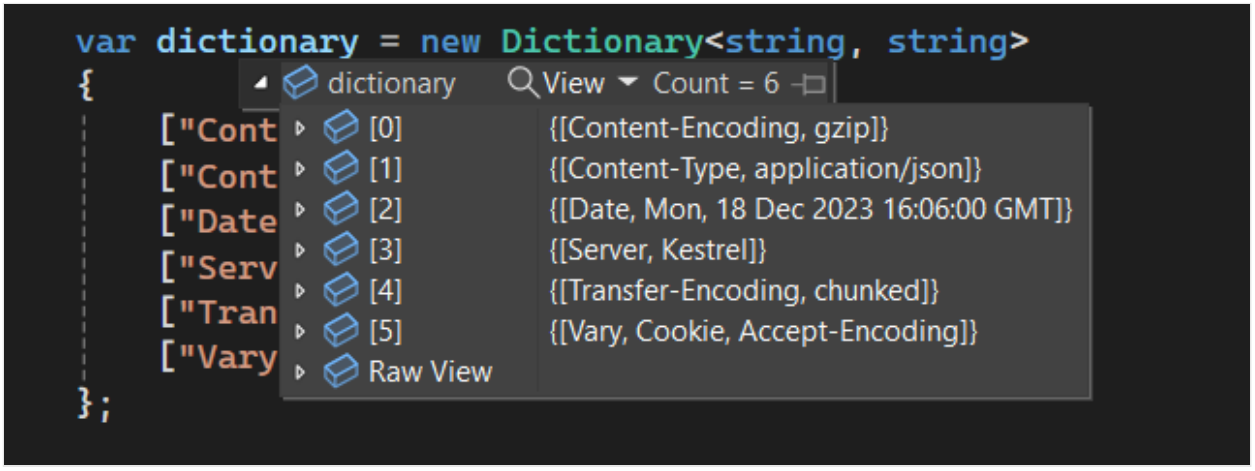
The following animated image shows the new developer exception page:



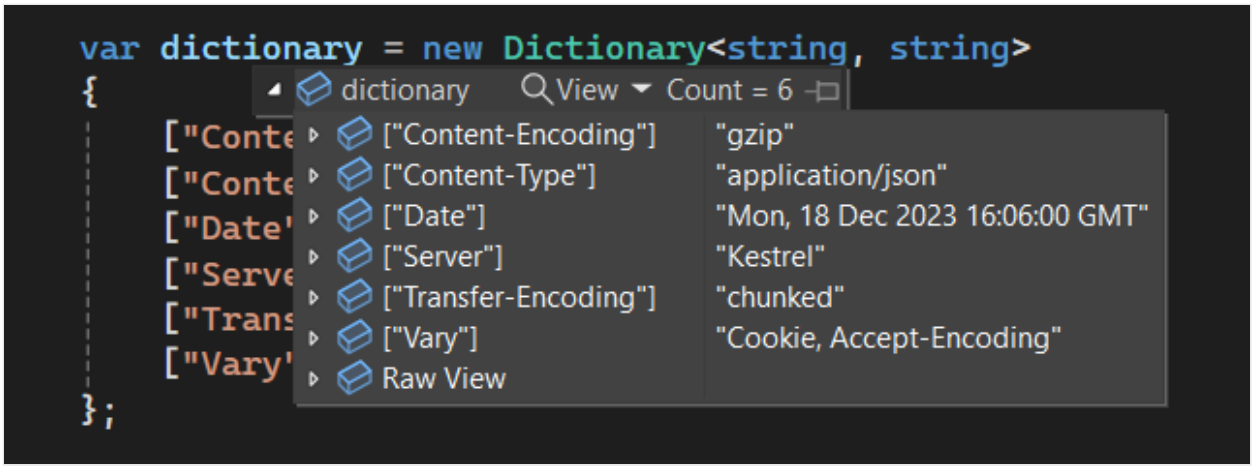
Dictionary debugging improvements

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

Before:



After:



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

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

Fix for 503's during app recycle in IIS

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

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

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

Example of setting `shutdownDelay`:

XML

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

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

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

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

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

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

C#

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

C#

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

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

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

C#

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

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

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

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

C#

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

Improved Kestrel connection metrics

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

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

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

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

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

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

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

For more information, see ASP.NET Core metrics.

Customize Kestrel named pipe endpoints

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

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

```
C#
```

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

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

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

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

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

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

C#

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

Opt-out of HTTP metrics on certain endpoints and requests

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

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

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

```
C#

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

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

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

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

```
C#

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

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

Data Protection support for deleting keys

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

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

C#

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

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

var serviceProvider = services.BuildServiceProvider();

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

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

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

Middleware supports Keyed DI

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

C#

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

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

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

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

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

Trust the ASP.NET Core HTTPS development certificate on Linux

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

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

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

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

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

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

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

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

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

- Bootstrap 5.3.3
- jQuery 3.7.1
- jQuery Validation 1.21.0

What's new in ASP.NET Core 8.0

Article • 11/06/2024

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

Blazor

Full-stack web UI

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

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

Interactive render modes also prerender content by default.

For more information, see the following articles:

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

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

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

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

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

New article on HTTP caching issues

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

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

New Blazor Web App template

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

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

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

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

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

New JS initializers for Blazor Web Apps

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

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

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

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

Split of prerendering and integration guidance

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

- Prerender ASP.NET Core Razor components
- Integrate ASP.NET Core Razor components into ASP.NET Core apps

Persist component state in a Blazor Web App

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

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

Form handling and model binding

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

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

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

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

Enhanced navigation and form handling

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

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

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

- Enhanced navigation and form handling
- Location changes

New article on static rendering with enhanced navigation for JS interop

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

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

Streaming rendering

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

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

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

Inject keyed services into components

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

C#

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

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

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

Access `HttpContext` as a cascading parameter

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

C#

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

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

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

Render Razor components outside of ASP.NET Core

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

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

Sections support

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

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

Error page support

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

Error.razor in 8.0 reference source

QuickGrid

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

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

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

Route to named elements

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

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

Root-level cascading values

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

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

Virtualize empty content

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

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

Close circuits when there are no remaining interactive server components

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

Monitor SignalR circuit activity

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

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

Faster runtime performance with the Jiterpreter

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

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

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

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

For more information, see the following articles:

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

Web-friendly Webcil packaging

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

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

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

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

Blazor WebAssembly debugging improvements

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

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

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

Content Security Policy (CSP) compatibility

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

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

Handle caught exceptions outside of a Razor component's lifecycle

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

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

Configure the .NET WebAssembly runtime

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

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

Configuration of connection timeouts in `HubConnectionBuilder`

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

For more information, see the following:

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

Project templates shed Open Iconic

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

Support for dialog cancel and close events

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

In the following example:

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

razor

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

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

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

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

@code {
    private string? message;

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

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

Blazor Identity UI

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

For more information, see the following resources:

- Secure ASP.NET Core server-side Blazor apps
- What's new with identity in .NET 8 (blog post)

Secure Blazor WebAssembly with ASP.NET Core Identity

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

For more information, see the following resources:

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

Blazor Server with Yarp routing

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

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

Multiple Blazor Web Apps per server project

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

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

Blazor Hybrid

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

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

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

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

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

SignalR

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

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

Prior approach for JavaScript clients

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

JavaScript

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

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

New approach for JavaScript clients

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

JavaScript

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

Prior approach for the JavaScript client of a Blazor Server app

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

JavaScript

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

```
  }
});
```

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

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

Blazor Web App:

JavaScript

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

Blazor Server:

JavaScript

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

Prior approach for .NET clients

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

C#

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

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

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

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

New approach for .NET clients

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

```
C#

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

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

await builder.StartAsync();
```

SignalR stateful reconnect

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

Stateful reconnect achieves this by:

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

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

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

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

  ```
  C#

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

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

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

The `StatefulReconnectBufferSize` option set globally:

C#

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

The `StatefulReconnectBufferSize` option set for a specific hub:

C#

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

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

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

JavaScript

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

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

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

C#

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

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

For more information, see Configure stateful reconnect.

Minimal APIs

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

User override culture

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

C#

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

Binding to forms

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

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

For more information, see:

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

Binding from forms is now supported for:

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

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

Antiforgery with minimal APIs

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

```
C#

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

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

app.Run();
```

The antiforgery middleware:

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

The antiforgery token is only validated if:

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

For more information, see Antiforgery with Minimal APIs.

New `IResettable` interface in `ObjectPool`

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

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

For more information, see the ObjectPool sample.

Native AOT

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

Libraries and Native AOT

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

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

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

Library authors hoping to support Native AOT are encouraged to:

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

New project template

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

New `CreateSlimBuilder` method

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

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

For more information, see The CreateSlimBuilder method.

New `CreateEmptyBuilder` method

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

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

C#

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

var app = builder.Build();

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

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

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

Reduced app size with configurable HTTPS support

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

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

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

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

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

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

Top-level APIs annotated for trim warnings

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

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

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

app.Run();
```

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

Request delegate generator

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

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

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

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

Improved performance using Interceptors

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

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

Logging and exception handling in compile-time generated minimal APIs

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

AOT and System.Text.Json

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

C#

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

...

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

}
```

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

- JsonSerializerOptions.TypeInfoResolverChain
- Chain source generators

- Changes to support source generation

Libraries and Native AOT

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

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

Kestrel and HTTP.sys servers

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

Support for named pipes in Kestrel

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

C#

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

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

Performance improvements to named pipes transport

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

Time to create 100,000 connections:

- Before : 5.916 seconds
- After : 2.374 seconds

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

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

Certificate file watching in Kestrel

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

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

Warning when specified HTTP protocols won't be used

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

`HTTP_PORTS` and `HTTPS_PORTS` config keys

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

cli

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

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

```
cli
```

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

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

SNI host name in ITlsHandshakeFeature

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

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

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

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

For more information, see ITlsHandshakeFeature.HostName.

IHttpSysRequestTimingFeature

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

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

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

C#

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

builder.WebHost.UseHttpSys();

var app = builder.Build();

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

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

    var timingType = HttpSysRequestTimingType.RequestRoutingEnd;

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

    return next(context);
});

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

app.Run();
```

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

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

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

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

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

Authentication and authorization

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

Identity API endpoints

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

IAuthorizationRequirementData

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

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

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

```
```

C#

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

var builder = WebApplication.CreateBuilder();

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

var app = builder.Build();

app.MapControllers();

app.Run();
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace AuthRequirementsData.Controllers;

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

C#

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

namespace AuthRequirementsData.Authorization;

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

    public
MinimumAgeAuthorizationHandler(ILogger<MinimumAgeAuthorizationHandler>
```

```
logger)
    {
        _logger = logger;
    }

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

requirement.Age);

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

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

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

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

        return Task.CompletedTask;
    }
}
```

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

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

diff

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

  var builder = WebApplication.CreateBuilder();

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

  var app = builder.Build();

  app.MapControllers();

  app.Run();
```

The updated `MinimumAgeAuthorizationHandler`:

diff

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

namespace AuthRequirementsData.Authorization;
```

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

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

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

The complete updated sample can be found here.

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

Securing Swagger UI endpoints

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

Miscellaneous

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

Keyed services support in Dependency Injection

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

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

C#

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

var builder = WebApplication.CreateBuilder(args);

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

var app = builder.Build();

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

smallCache.Get("date"));

app.MapControllers();

app.Run();

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

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

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

public class MyHub : Hub
{
```

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

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

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

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

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

Support for generic attributes

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

diff

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

Generic variants are supported for the following attributes:

- `[ProducesResponseType<T>]`

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

Code analysis in ASP.NET Core apps

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

Expand table

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

Route tooling

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

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

- Route syntax highlighting
- Autocomplete of parameter and route names

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

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

ASP.NET Core metrics

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

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

Metrics offers many improvements compared to existing event counters:

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

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

IExceptionHandler

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

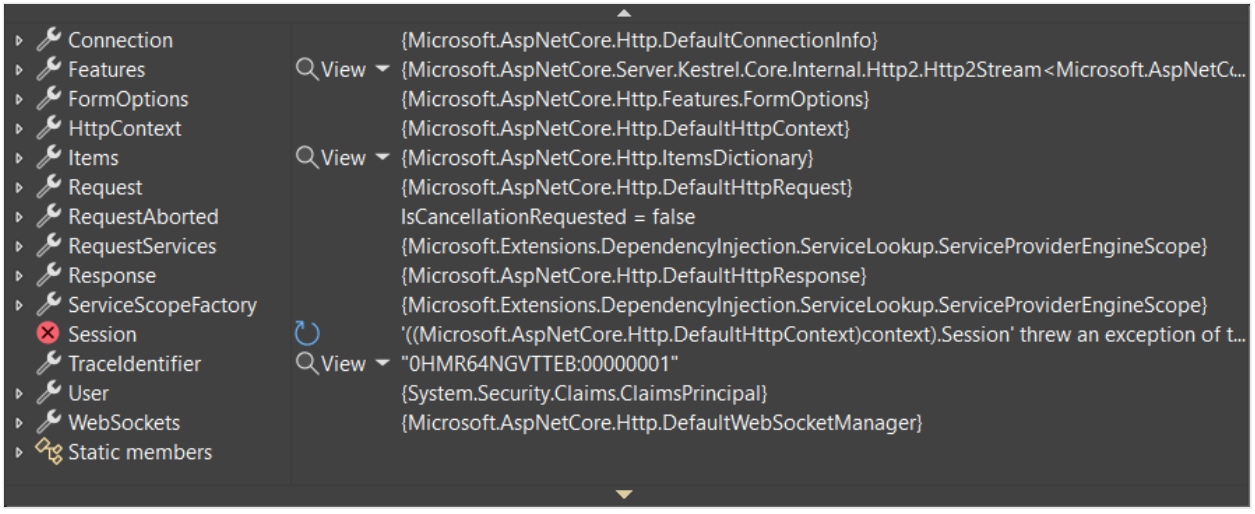
`IExceptionHandler` implementations are registered by calling IServiceCollection.AddExceptionHandler<T>. Multiple implementations can be added, and they're called in the order registered. If an exception handler handles a request, it can return `true` to stop processing. If an exception isn't handled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

For more information, see IExceptionHandler.

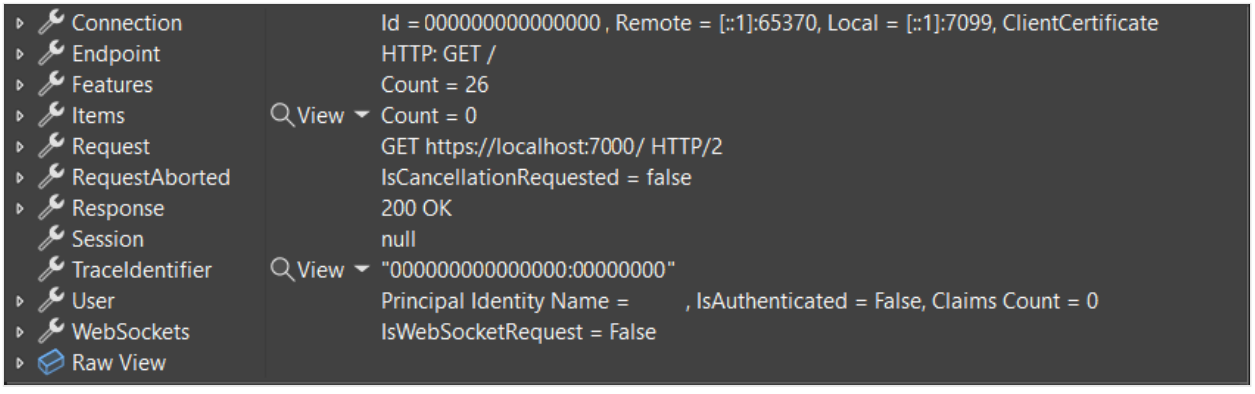
Improved debugging experience

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

.NET 7:

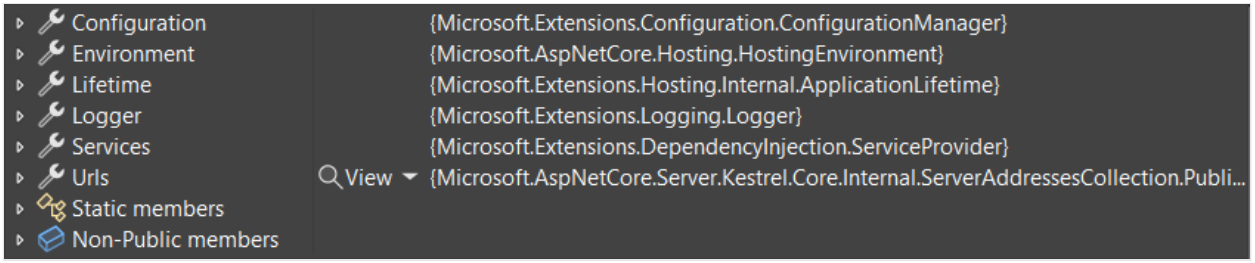


.NET 8:

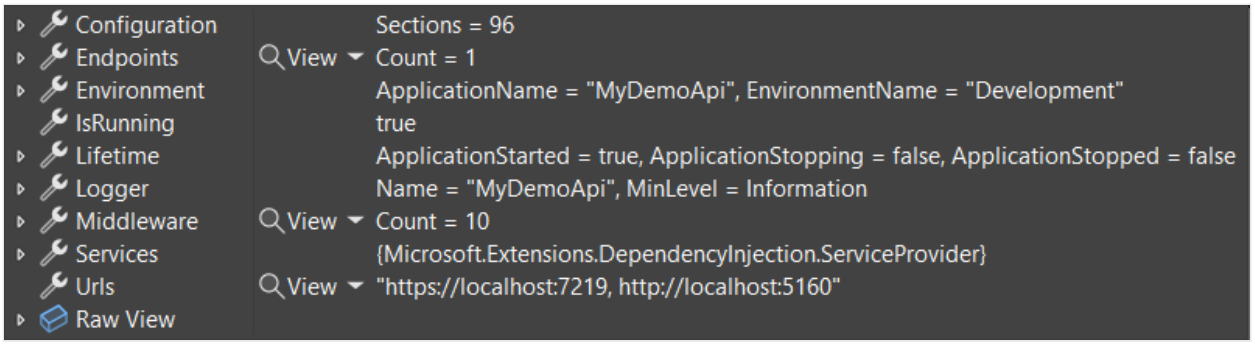


The debugger display for `WebApplication` highlights important information such as configured endpoints, middleware, and `IConfiguration` values.

.NET 7:



.NET 8:



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

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

`IPNetwork.Parse` and `TryParse`

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

Here are IPv4 examples:

C#

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

C#

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

C#

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

And here are examples for IPv6:

C#

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

C#

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

C#

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

Redis-based output caching

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

Short-circuit middleware after routing

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

C#

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

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

C#

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

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

HTTP logging middleware extensibility

The HTTP logging middleware has several new capabilities:

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

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

New APIs in ProblemDetails to support more resilient integrations

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

C#

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

var app = builder.Build();

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

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

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

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

app.Run();
```

For more information, see IProblemDetailsService fallback

Additional resources

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

What's new in ASP.NET Core 7.0

Article • 09/27/2024

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

Rate limiting middleware in ASP.NET Core

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

Authentication uses single scheme as DefaultScheme

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

MVC and Razor pages

Support for nullable models in MVC views and Razor Pages

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

```
C#

@model Product?
```

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

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

Customize the cookie consent value

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

C#

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

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

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

For more information, see Customize the cookie consent value.

API controllers

Parameter binding with DI in API controllers

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

C#

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

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

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

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

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

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

var app = builder.Build();

app.MapControllers();

app.Run();
```

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

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

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

JSON property names in validation errors

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

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

C#

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

var builder = WebApplication.CreateBuilder(args);

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

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

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

C#

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

var builder = WebApplication.CreateBuilder(args);

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

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

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

Minimal APIs

Filters in Minimal API apps

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

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

Filters can be helpful in the following scenarios:

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

For more information, see Filters in Minimal API apps

Bind arrays and string values from headers and query strings

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

C#

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

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

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

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

values from headers and query strings.

For more information, see Add endpoint summary or description.

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

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

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

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

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

New Results.Stream overloads

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

C#

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

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

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

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

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

For more information, see Stream examples

Typed results for minimal APIs

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

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

C#

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

Improved unit testability for minimal route handlers

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

The following code uses the Ok<TValue> class:

C#

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

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

    await context.SaveChangesAsync();

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

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

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

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

For more information, see IResult implementation types.

New HttpResult interfaces

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

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

For more information, see IHttpResult interfaces.

OpenAPI improvements for minimal APIs

`Microsoft.AspNetCore.OpenApi` NuGet package

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

C#

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

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

Call `WithOpenApi` with parameters

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

C#

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

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

Provide endpoint descriptions and summaries

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

For more information, see OpenAPI in minimal API apps

File uploads using IFormFile and IFormFileCollection

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

C#

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

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

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

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

app.Run();
```

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

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

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

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

Minimal APIs and API controllers

New problem details service

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

For more information, see Problem details service.

Route groups

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

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

C#

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

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


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

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

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

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

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

C#

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

    return group;
}
```

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

C#

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

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

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

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

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

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

C#

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

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

C#

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

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

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

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

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

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

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

.NET CLI

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

gRPC

JSON transcoding

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

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

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

gRPC health checks in ASP.NET Core

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

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

reported to callers.

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

Improved call credentials support

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

- Support for call credentials with plaintext connections. Previously, a gRPC call only sent call credentials if the connection was secured with TLS. A new setting on `GrpcChannelOptions`, called `UnsafeUseInsecureChannelCallCredentials`, allows this behavior to be customized. There are security implications to not securing a connection with TLS.
- A new method called `AddCallCredentials` is available with the gRPC client factory. `AddCallCredentials` is a quick way to configure call credentials for a gRPC client and integrates well with dependency injection (DI).

The following code configures the gRPC client factory to send `Authorization` metadata:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
       o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
       if (!string.IsNullOrEmpty(_token))
       {
          metadata.Add("Authorization", $"Bearer {_token}");
       }
       return Task.CompletedTask;
    });
```

For more information, see Configure a bearer token with the gRPC client factory.

SignalR

Client results

The server now supports requesting a result from a client. This requires the server to use `ISingleClientProxy.InvokeAsync` and the client to return a result from its `.On` handler. Strongly-typed hubs can also return values from interface methods.

For more information, see Client results

Dependency injection for SignalR hub methods

SignalR hub methods now support injecting services through dependency injection (DI).

Hub constructors can accept services from DI as parameters, which can be stored in properties on the class for use in a hub method. For more information, see Inject services into a hub

Blazor

Handle location changing events and navigation state

In .NET 7, Blazor supports location changing events and maintaining navigation state. This allows you to warn users about unsaved work or to perform related actions when the user performs a page navigation.

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

- Navigation options
- Handle/prevent location changes

Empty Blazor project templates

Blazor has two new project templates for starting from a blank slate. The new **Blazor Server App Empty** and **Blazor WebAssembly App Empty** project templates are just like their non-empty counterparts but without example code. These empty templates only include a basic home page, and we've removed Bootstrap so that you can start with a different CSS framework.

For more information, see the following articles:

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

Blazor custom elements

The Microsoft.AspNetCore.Components.CustomElements package enables building standards based custom DOM elements using Blazor.

For more information, see ASP.NET Core Razor components.

Bind modifiers (`@bind:after`, `@bind:get`, `@bind:set`)

> **Important**
>
> The `@bind:after`/`@bind:get`/`@bind:set` features are receiving further updates at this time. To take advantage of the latest updates, confirm that you've installed the latest SDK.
>
> Using an event callback parameter (`[Parameter] public EventCallback<string> ValueChanged { get; set; }`) isn't supported. Instead, pass an Action-returning or Task-returning method to `@bind:set`/`@bind:after`.
>
> For more information, see the following resources:
>
> - Blazor @bind:after not working on .NET 7 RTM release (dotnet/aspnetcore #44957)
> - BindGetSetAfter701 sample app (javiercn/BindGetSetAfter701 GitHub repository)

In .NET 7, you can run asynchronous logic after a binding event has completed using the new `@bind:after` modifier. In the following example, the `PerformSearch` asynchronous method runs automatically after any changes to the search text are detected:

razor

```
<input @bind="searchText" @bind:after="PerformSearch" />

@code {
    private string searchText;

    private async Task PerformSearch()
    {
        ...
    }
}
```

In .NET 7, it's also easier to set up binding for component parameters. Components can support two-way data binding by defining a pair of parameters:

- `@bind:get`: Specifies the value to bind.
- `@bind:set`: Specifies a callback for when the value changes.

The `@bind:get` and `@bind:set` modifiers are always used together.

Examples:

razor

```
@* Elements *@

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

@* Components *@

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void After(){}
    private void Set() {}
    private Task AfterAsync() { return Task.CompletedTask; }
```

```
    private Task SetAsync(string value) { return Task.CompletedTask; }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Hot Reload improvements

In .NET 7, Hot Reload support includes the following:

- Components reset their parameters to their default values when a value is removed.
- Blazor WebAssembly:
  - Add new types.
  - Add nested classes.
  - Add static and instance methods to existing types.
  - Add static fields and methods to existing types.
  - Add static lambdas to existing methods.
  - Add lambdas that capture `this` to existing methods that already captured `this` previously.

Dynamic authentication requests with MSAL in Blazor WebAssembly

New in .NET 7, Blazor WebAssembly supports creating dynamic authentication requests at runtime with custom parameters to handle advanced authentication scenarios.

For more information, see the following articles:

- Secure ASP.NET Core Blazor WebAssembly
- ASP.NET Core Blazor WebAssembly additional security scenarios

Blazor WebAssembly debugging improvements

Blazor WebAssembly debugging has the following improvements:

- Support for the **Just My Code** setting to show or hide type members that aren't from user code.
- Support for inspecting multidimensional arrays.
- **Call Stack** now shows the correct name for asynchronous methods.
- Improved expression evaluation.
- Correct handling of the `new` keyword on derived members.

- Support for debugger-related attributes in `System.Diagnostics`.

`System.Security.Cryptography` support on WebAssembly

.NET 6 supported the SHA family of hashing algorithms when running on WebAssembly. .NET 7 enables more cryptographic algorithms by taking advantage of SubtleCrypto, when possible, and falling back to a .NET implementation when SubtleCrypto can't be used. The following algorithms are supported on WebAssembly in .NET 7:

- SHA1
- SHA256
- SHA384
- SHA512
- HMACSHA1
- HMACSHA256
- HMACSHA384
- HMACSHA512
- AES-CBC
- PBKDF2
- HKDF

For more information, see Developers targeting browser-wasm can use Web Crypto APIs (dotnet/runtime #40074).

Inject services into custom validation attributes

You can now inject services into custom validation attributes. Blazor sets up the `ValidationContext` so that it can be used as a service provider.

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

`Input*` components outside of an `EditContext`/`EditForm`

The built-in input components are now supported outside of a form in Razor component markup.

For more information, see ASP.NET Core Blazor input components.

Project template changes

When .NET 6 was released last year, the HTML markup of the `_Host` page (`Pages/_Host.chstml`) was split between the `_Host` page and a new `_Layout` page (`Pages/_Layout.chstml`) in the .NET 6 Blazor Server project template.

In .NET 7, the HTML markup has been recombined with the `_Host` page in project templates.

Several additional changes were made to the Blazor project templates. It isn't feasible to list every change to the templates in the documentation. To migrate an app to .NET 7 in order to adopt all of the changes, see Migrate from ASP.NET Core 6.0 to 7.0.

Experimental `QuickGrid` component

The new `QuickGrid` component provides a convenient data grid component for most common requirements and as a reference architecture and performance baseline for anyone building Blazor data grid components.

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

Live demo: QuickGrid for Blazor sample app

Virtualization enhancements

Virtualization enhancements in .NET 7:

- The `Virtualize` component supports using the document itself as the scroll root, as an alternative to having some other element with `overflow-y: scroll` applied.
- If the `Virtualize` component is placed inside an element that requires a specific child tag name, `SpacerElement` allows you to obtain or set the virtualization spacer tag name.

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

- Root-level virtualization
- Control the spacer element tag name

`MouseEventArgs` updates

`MovementX` and `MovementY` have been added to `MouseEventArgs`.

For more information, see ASP.NET Core Blazor event handling.

New Blazor loading page

The Blazor WebAssembly project template has a new loading UI that shows the progress of loading the app.

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

Improved diagnostics for authentication in Blazor WebAssembly

To help diagnose authentication issues in Blazor WebAssembly apps, detailed logging is available.

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

JavaScript interop on WebAssembly

JavaScript `[JSImport]`/`[JSExport]` interop API is a new low-level mechanism for using .NET in Blazor WebAssembly and JavaScript-based apps. With this new JavaScript interop capability, you can invoke .NET code from JavaScript using the .NET WebAssembly runtime and call into JavaScript functionality from .NET without any dependency on the Blazor UI component model.

For more information:

- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor: Pertains only to Blazor WebAssembly apps.
- JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly: Pertains only to JavaScript apps that don't depend on the Blazor UI component model.

Conditional registration of the authentication state provider

Prior to the release of .NET 7, `AuthenticationStateProvider` was registered in the service container with `AddScoped`. This made it difficult to debug apps, as it forced a specific order of service registrations when providing a custom implementation. Due to internal framework changes over time, it's no longer necessary to register `AuthenticationStateProvider` with `AddScoped`.

In developer code, make the following change to the authentication state provider service registration:

```diff
- builder.Services.AddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
+ builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

In the preceding example, `ExternalAuthStateProvider` is the developer's service implementation.

Improvements to the .NET WebAssembly build tools

New features in the `wasm-tools` workload for .NET 7 that help improve performance and handle exceptions:

- WebAssembly Single Instruction, Multiple Data (SIMD) support (only with AOT, not supported by Apple Safari)
- WebAssembly exception handling support

For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Blazor Hybrid

External URLs

An option has been added that permits opening external webpages in the browser.

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

Security

New guidance is available for Blazor Hybrid security scenarios. For more information, see the following articles:

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor Hybrid security considerations

Performance

Output caching middleware

Output caching is a new middleware that stores responses from a web app and serves them from a cache rather than computing them every time. Output caching differs from response caching in the following ways:

- The caching behavior is configurable on the server.
- Cache entries can be programmatically invalidated.
- Resource locking mitigates the risk of cache stampede and thundering herd.
- Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of a cached response body.
- The cache storage medium is extensible.

For more information, see Overview of caching and Output caching middleware.

HTTP/3 improvements

This release:

- Makes HTTP/3 fully supported by ASP.NET Core, it's no longer experimental.
- Improves Kestrel's support for HTTP/3. The two main areas of improvement are feature parity with HTTP/1.1 and HTTP/2, and performance.
- Provides full support for UseHttps(ListenOptions, X509Certificate2) with HTTP/3. Kestrel offers advanced options for configuring connection certificates, such as hooking into Server Name Indication (SNI).
- Adds support for HTTP/3 on HTTP.sys and IIS.

The following example shows how to use an SNI callback to resolve TLS options:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
using Microsoft.AspNetCore.Server.Kestrel.Https;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(8080, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
        listenOptions.UseHttps(new TlsHandshakeCallbackOptions
        {
            OnConnection = context =>
            {
                var options = new SslServerAuthenticationOptions
                {
                    ServerCertificate =
```

```
MyResolveCertForHost(context.ClientHelloInfo.ServerName)
                };
                return new ValueTask<SslServerAuthenticationOptions>
(options);
            },
        });
    });
});
```

Significant work was done in .NET 7 to reduce HTTP/3 allocations. You can see some of those improvements in the following GitHub PR's:

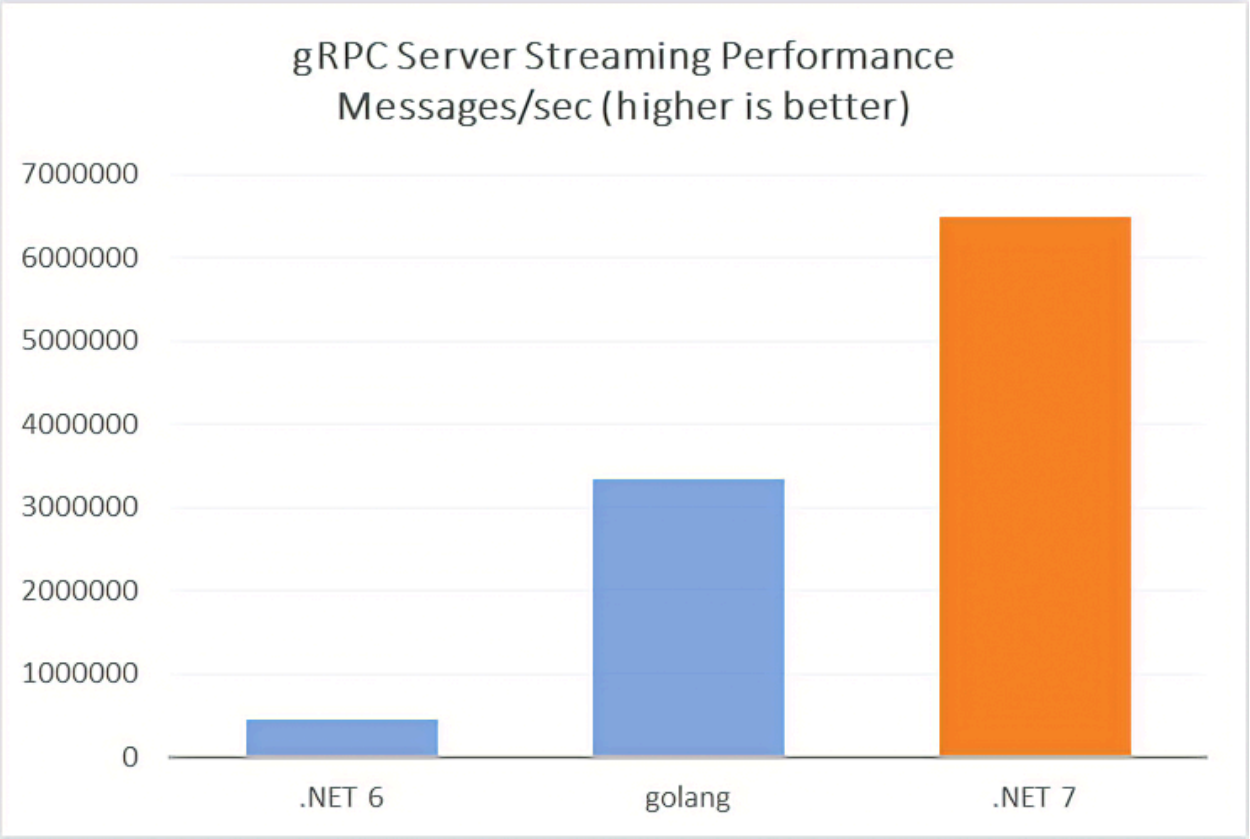
- HTTP/3: Avoid per-request cancellation token allocations
- HTTP/3: Avoid ConnectionAbortedException allocations
- HTTP/3: ValueTask pooling

HTTP/2 Performance improvements

.NET 7 introduces a significant re-architecture of how Kestrel processes HTTP/2 requests. ASP.NET Core apps with busy HTTP/2 connections will experience reduced CPU usage and higher throughput.

Previously, the HTTP/2 multiplexing implementation relied on a lock controlling which request can write to the underlying TCP connection. A thread-safe queue replaces the write lock. Now, rather than fighting over which thread gets to use the write lock, requests now queue up and a dedicated consumer processes them. Previously wasted CPU resources are available to the rest of the app.

One place where these improvements can be noticed is in gRPC, a popular RPC framework that uses HTTP/2. Kestrel + gRPC benchmarks show a dramatic improvement:



Changes were made in the HTTP/2 frame writing code that improves performance when there are multiple streams trying to write data on a single HTTP/2 connection. We now dispatch TLS work to the thread pool and more quickly release a write lock that other streams can acquire to write their data. The reduction in wait times can yield significant performance improvements in cases where there is contention for this write lock. A gRPC benchmark with 70 streams on a single connection (with TLS) showed a ~15% improvement in requests per second (RPS) with this change.

Http/2 WebSockets support

.NET 7 introduces WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly.

Using WebSockets over HTTP/2 takes advantage of new features such as:

- Header compression.
- Multiplexing, which reduces the time and resources needed when making multiple requests to the server.

These supported features are available in Kestrel on all HTTP/2 enabled platforms. The version negotiation is automatic in browsers and Kestrel, so no new APIs are needed.

For more information, see Http/2 WebSockets support.

Kestrel performance improvements on high core machines

Kestrel uses ConcurrentQueue<T> for many purposes. One purpose is scheduling I/O operations in Kestrel's default Socket transport. Partitioning the `ConcurrentQueue` based on the associated socket reduces contention and increases throughput on machines with many CPU cores.

Profiling on high core machines on .NET 6 showed significant contention in one of Kestrel's other `ConcurrentQueue` instances, the `PinnedMemoryPool` that Kestrel uses to cache byte buffers.

In .NET 7, Kestrel's memory pool is partitioned the same way as its I/O queue, which leads to much lower contention and higher throughput on high core machines. On the 80 core ARM64 VMs, we're seeing over 500% improvement in responses per second (RPS) in the TechEmpower plaintext benchmark. On 48 Core AMD VMs, the improvement is nearly 100% in our HTTPS JSON benchmark.

`ServerReady` event to measure startup time

Apps using EventSource can measure the startup time to understand and optimize startup performance. The new ServerReady event in Microsoft.AspNetCore.Hosting represents the point where the server is ready to respond to requests.

Server

New ServerReady event for measuring startup time

The ServerReady event has been added to measure startup time of ASP.NET Core apps.

IIS

Shadow copying in IIS

Shadow copying app assemblies to the ASP.NET Core Module (ANCM) for IIS can provide a better end user experience than stopping the app by deploying an app offline file.

For more information, see Shadow copying in IIS.

Miscellaneous

Kestrel full certificate chain improvements

HttpsConnectionAdapterOptions has a new ServerCertificateChain property of type X509Certificate2Collection, which makes it easier to validate certificate chains by allowing a full chain including intermediate certificates to be specified. See dotnet/aspnetcore#21513 for more details.

dotnet watch

Improved console output for dotnet watch

The console output from dotnet watch has been improved to better align with the logging of ASP.NET Core and to stand out with 😮emojis😍.

Here's an example of what the new output looks like:

```
dotnet watch 🔥 Hot reload enabled. For a list of supported edits, see https://aka.ms/dotnet/hot-reload.
  💡 Press "Ctrl + R" to restart.
dotnet watch 🔧 Building...
  Determining projects to restore...
  All projects are up-to-date for restore.
  WebApplication18 -> C:\tmp\WebApplication18\bin\Debug\net6.0\WebApplication18.dll
dotnet watch 🚀 Started
dotnet watch ⌚ Files changed: .\Pages\Index.cshtml, .\Pages\Index.cshtml.cs
dotnet watch 🔥 Hot reload of changes succeeded.
```

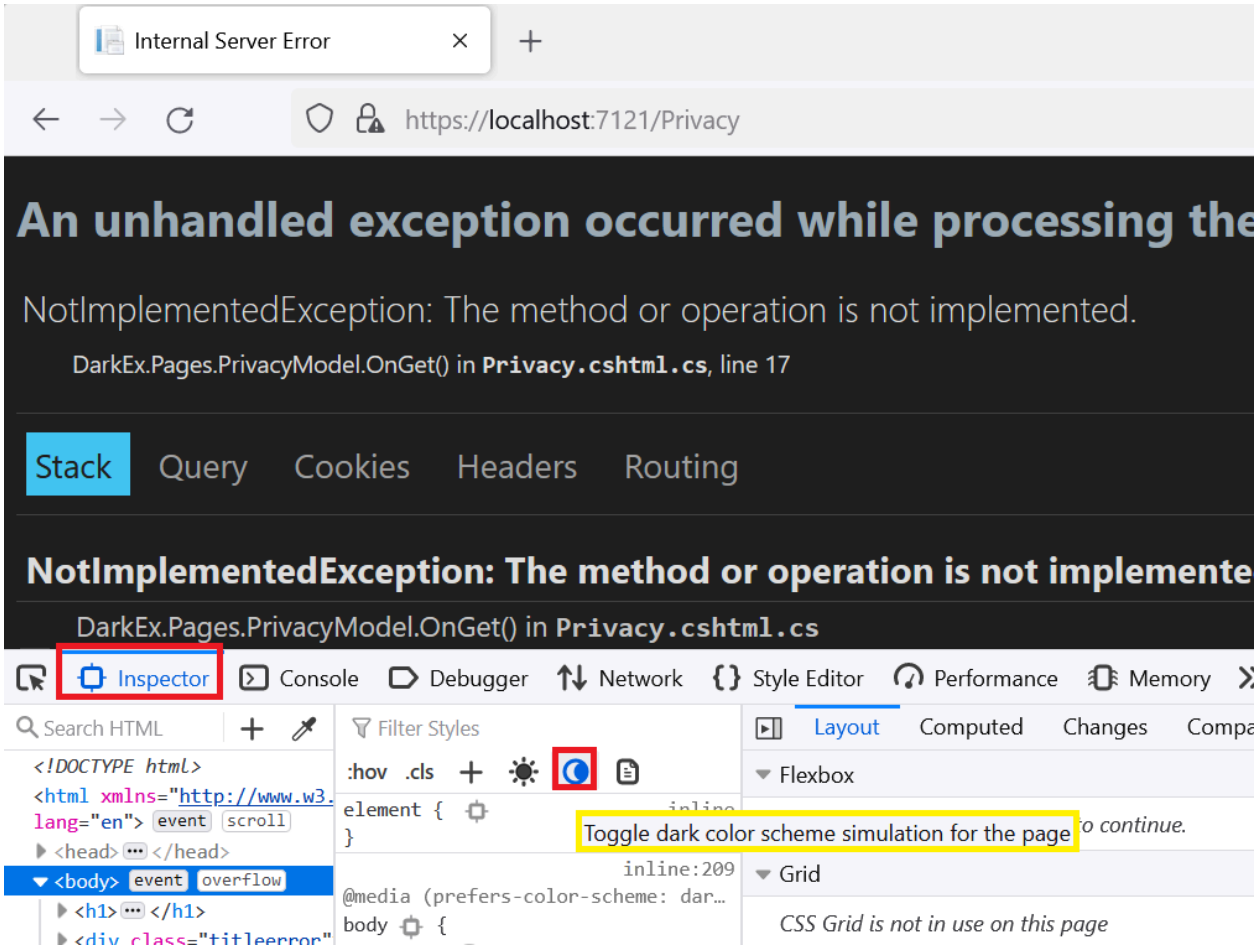
For more information, see this GitHub pull request.

Configure dotnet watch to always restart for rude edits

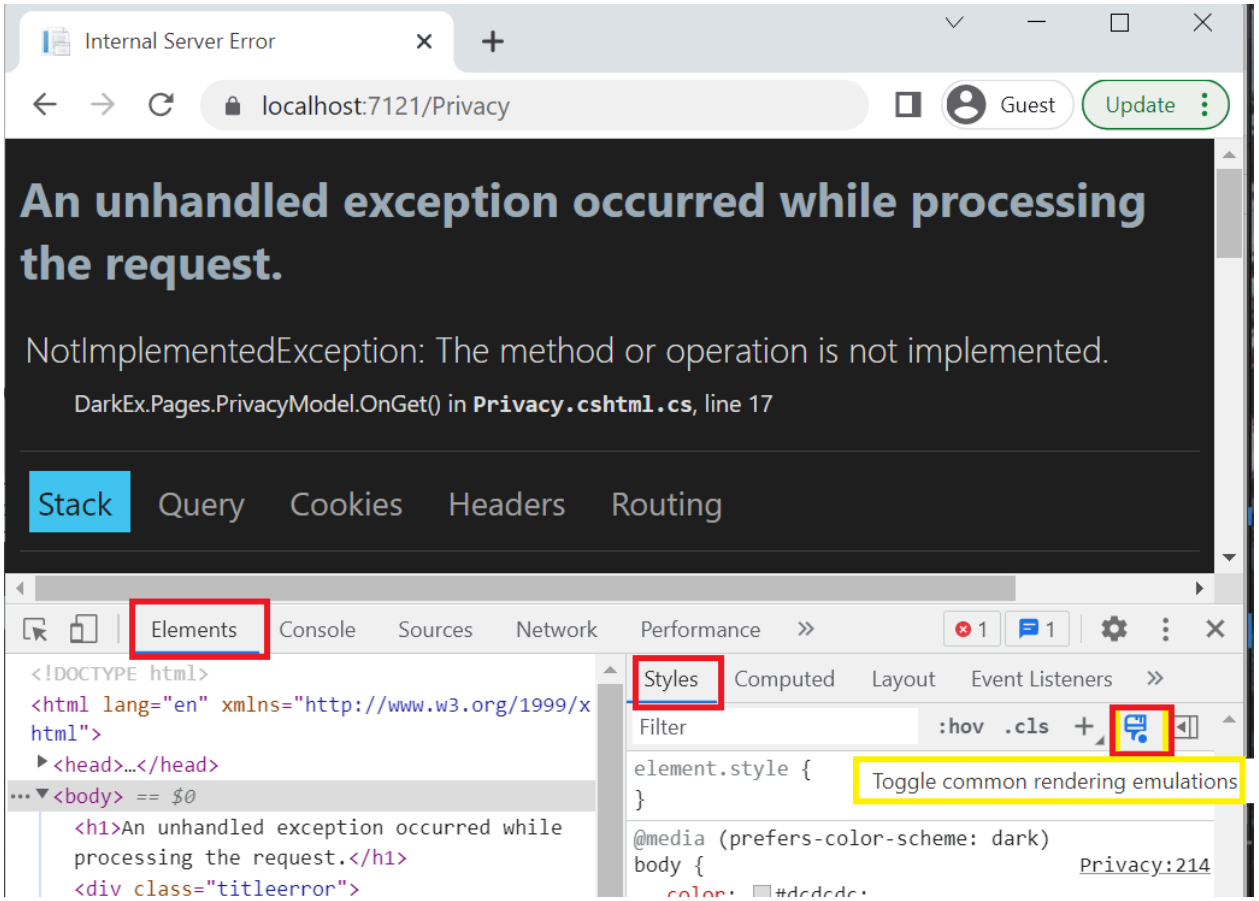
Rude edits are edits that can't be hot reloaded. To configure dotnet watch to always restart without a prompt for rude edits, set the `DOTNET_WATCH_RESTART_ON_RUDE_EDIT` environment variable to `true`.

Developer exception page dark mode

Dark mode support has been added to the developer exception page, thanks to a contribution by Patrick Westerhoff. To test dark mode in a browser, from the developer tools page, set the mode to dark. For example, in Firefox:


Internal Server Error
https://localhost:7121/Privacy
An unhandled exception occurred while processing the
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
NotImplementedException: The method or operation is not implemente
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Search HTML
Filter Styles
Layout
Computed
Changes
:hov .cls
Flexbox
Toggle dark color scheme simulation for the page
Grid
CSS Grid is not in use on this page

In Chrome:


Internal Server Error
localhost:7121/Privacy
Guest
Update
An unhandled exception occurred while processing the request.
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
Elements
Console
Sources
Network
Performance
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
element.style {
}
Toggle common rendering emulations
@media (prefers-color-scheme: dark)
body {
Privacy:214

Project template option to use Program.Main method instead of top-level statements

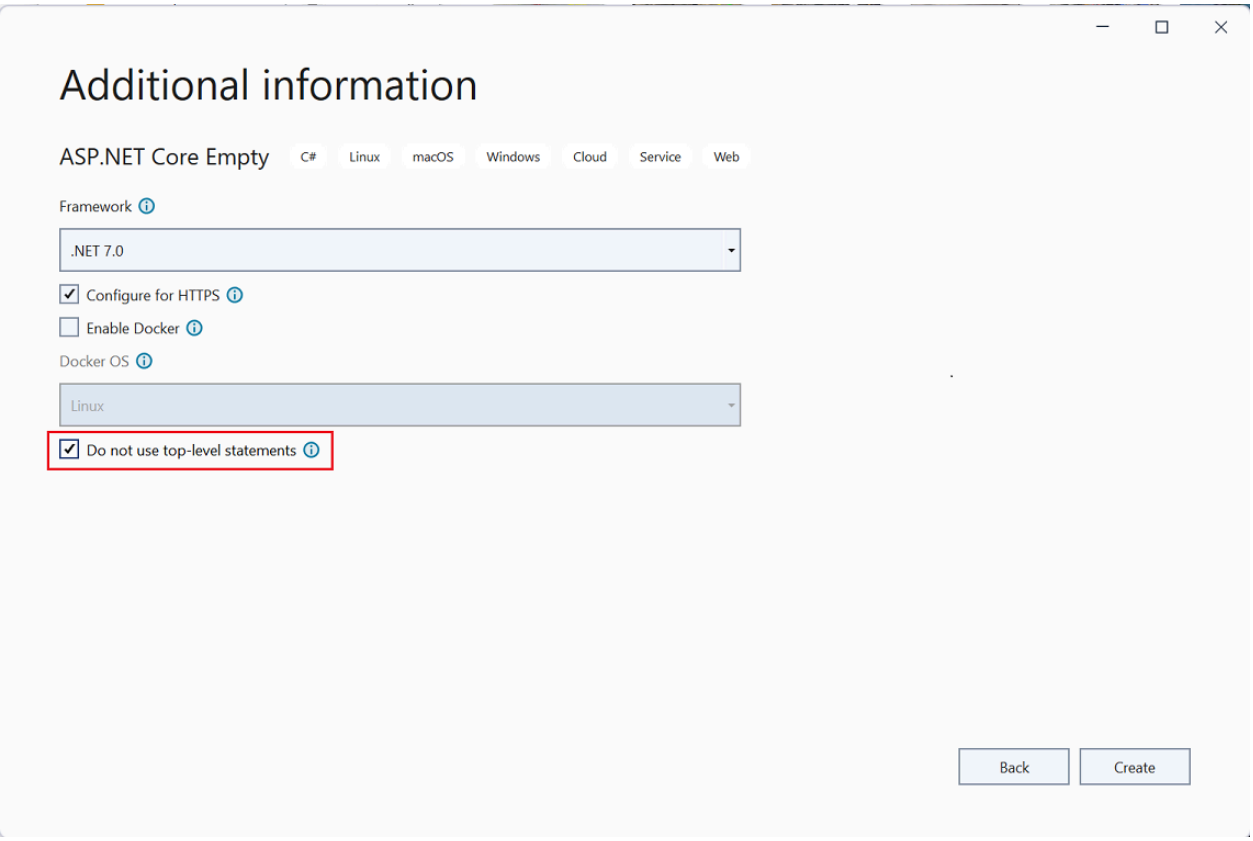
The .NET 7 templates include an option to not use top-level statements and generate a `namespace` and a `Main` method declared on a `Program` class.

Using the .NET CLI, use the `--use-program-main` option:

.NET CLI

```
dotnet new web --use-program-main
```

With Visual Studio, select the new **Do not use top-level statements** checkbox during project creation:



Updated Angular and React templates

The Angular project template has been updated to Angular 14. The React project template has been updated to React 18.2.

Manage JSON Web Tokens in development with dotnet user-jwts

The new `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs). For more information, see Manage JSON Web Tokens in development with dotnet user-jwts.

Support for additional request headers in W3CLogger

You can now specify additional request headers to log when using the W3C logger by calling `AdditionalRequestHeaders()` on W3CLoggerOptions:

C#

```
services.AddW3CLogging(logging =>
{
    logging.AdditionalRequestHeaders.Add("x-forwarded-for");
    logging.AdditionalRequestHeaders.Add("x-client-ssl-protocol");
});
```

For more information, see W3CLogger options.

Request decompression

The new Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

For more information, see Request decompression middleware.

What's new in ASP.NET Core 6.0

Article • 10/18/2024

This article highlights the most significant changes in ASP.NET Core 6.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Minimal APIs

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They are ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core. For more information, see:

- Tutorial: Create a minimal API with ASP.NET Core
- Differences between minimal APIs and APIs with controllers
- Minimal APIs quick reference
- Code samples migrated to the new minimal hosting model in 6.0

SignalR

Long running activity tag for SignalR connections

SignalR uses the new Microsoft.AspNetCore.Http.Features.IHttpActivityFeature.Activity to add an `http.long_running` tag to the request activity. `IHttpActivityFeature.Activity` is used by APM services like Azure Monitor Application Insights to filter SignalR requests from creating long running request alerts.

SignalR performance improvements

- Allocate HubCallerClients once per connection instead of every hub method call.
- Avoid closure allocation in SignalR `DefaultHubDispatcher.Invoke`. State is passed to a local static function via parameters to avoid a closure allocation. For more information, see this GitHub pull request.
- Allocate a single StreamItemMessage per stream instead of per stream item in server-to-client streaming. For more information, see this GitHub pull request.

Razor compiler

Razor compiler updated to use source generators

The Razor compiler is now based on C# source generators. Source generators run during compilation and inspect what is being compiled to produce additional files that are compiled along with the rest of the project. Using source generators simplifies the Razor compiler and significantly speeds up build times.

Razor compiler no longer produces a separate Views assembly

The Razor compiler previously utilized a two-step compilation process that produced a separate Views assembly that contained the generated views and pages (`.cshtml` files) defined in the app. The generated types were public and under the `AspNetCore` namespace.

The updated Razor compiler builds the views and pages types into the main project assembly. These types are now generated by default as internal sealed in the `AspNetCoreGeneratedDocument` namespace. This change improves build performance, enables single file deployment, and enables these types to participate in Hot Reload.

For more information about this change, see the related announcement issue on GitHub.

ASP.NET Core performance and API improvements

Many changes were made to reduce allocations and improve performance across the stack:

- Non-allocating app.Use extension method. The new overload of `app.Use` requires passing the context to `next` which saves two internal per-request allocations that are required when using the other overload.
- Reduced memory allocations when accessing HttpRequest.Cookies. For more information, see this GitHub issue.
- Use LoggerMessage.Define for the windows only HTTP.sys web server. The ILogger extension methods calls have been replaced with calls to `LoggerMessage.Define`.
- Reduce the per connection overhead in SocketConnection by ~30%. For more information, see this GitHub pull request.

- Reduce allocations by removing logging delegates in generic types. For more information, see this GitHub pull request.
- Faster GET access (about 50%) to commonly-used features such as IHttpRequestFeature, IHttpResponseFeature, IHttpResponseBodyFeature, IRouteValuesFeature, and IEndpointFeature. For more information, see this GitHub pull request.
- Use single instance strings for known header names, even if they aren't in the preserved header block. Using single instance string helps prevent multiple duplicates of the same string in long lived connections, for example, in Microsoft.AspNetCore.WebSockets. For more information, see this GitHub issue.
- Reuse HttpProtocol CancellationTokenSource in Kestrel. Use the new CancellationTokenSource.TryReset method on `CancellationTokenSource` to reuse tokens if they haven't been canceled. For more information, see this GitHub issue and this video.
- Implement and use an AdaptiveCapacityDictionary in Microsoft.AspNetCore.Http RequestCookieCollection for more efficient access to dictionaries. For more information, see this GitHub pull request.

Reduced memory footprint for idle TLS connections

For long running TLS connections where data is only occasionally sent back and forth, we've significantly reduced the memory footprint of ASP.NET Core apps in .NET 6. This should help improve the scalability of scenarios such as WebSocket servers. This was possible due to numerous improvements in System.IO.Pipelines, SslStream, and Kestrel. The following sections detail some of the improvements that have contributed to the reduced memory footprint:

Reduce the size of `System.IO.Pipelines.Pipe`

For every connection that is established, two pipes are allocated in Kestrel:

- The transport layer to the app for the request.
- The application layer to the transport for the response.

By shrinking the size of System.IO.Pipelines.Pipe from 368 bytes to 264 bytes (about a 28.2% reduction), 208 bytes per connection are saved (104 bytes per Pipe).

Pool SocketSender

`SocketSender` objects (that subclass SocketAsyncEventArgs) are around 350 bytes at runtime. Instead of allocating a new `SocketSender` object per connection, they can be

pooled. `SocketSender` objects can be pooled because sends are usually very fast. Pooling reduces the per connection overhead. Instead of allocating 350 bytes per connection, only pay 350 bytes per `IOQueue` are allocated. Allocation is done per queue to avoid contention. Our WebSocket server with 5000 idle connections went from allocating ~1.75 MB (350 bytes * 5000) to allocating ~2.8 kb (350 bytes * 8) for `SocketSender` objects.

Zero bytes reads with SslStream

Bufferless reads are a technique employed in ASP.NET Core to avoid renting memory from the memory pool if there's no data available on the socket. Prior to this change, our WebSocket server with 5000 idle connections required ~200 MB without TLS compared to ~800 MB with TLS. Some of these allocations (4k per connection) were from Kestrel having to hold on to an ArrayPool<T> buffer while waiting for the reads on SslStream to complete. Given that these connections were idle, none of reads completed and returned their buffers to the `ArrayPool`, forcing the `ArrayPool` to allocate more memory. The remaining allocations were in `SslStream` itself: 4k buffer for TLS handshakes and 32k buffer for normal reads. In .NET 6, when the user performs a zero byte read on `SslStream` and it has no data available, `SslStream` internally performs a zero-byte read on the underlying wrapped stream. In the best case (idle connection), these changes result in a savings of 40 Kb per connection while still allowing the consumer (Kestrel) to be notified when data is available without holding on to any unused buffers.

Zero byte reads with PipeReader

With bufferless reads supported on `SslStream`, an option was added to perform zero byte reads to `StreamPipeReader`, the internal type that adapts a `Stream` into a `PipeReader`. In Kestrel, a `StreamPipeReader` is used to adapt the underlying `SslStream` into a `PipeReader`. Therefore it was necessary to expose these zero byte read semantics on the `PipeReader`.

A `PipeReader` can now be created that supports zero bytes reads over any underlying `Stream` that supports zero byte read semantics (e.g,. `SslStream`, NetworkStream, etc) using the following API:

.NET CLI

```
var reader = PipeReader.Create(stream, new
StreamPipeReaderOptions(useZeroByteReads: true));
```

Remove slabs from the `SlabMemoryPool`

To reduce fragmentation of the heap, Kestrel employed a technique where it allocated slabs of memory of 128 KB as part of its memory pool. The slabs were then further divided into 4 KB blocks that were used by Kestrel internally. The slabs had to be larger than 85 KB to force allocation on the large object heap to try and prevent the GC from relocating this array. However, with the introduction of the new GC generation, Pinned Object Heap (POH), it no longer makes sense to allocate blocks on slab. Kestrel now directly allocates blocks on the POH, reducing the complexity involved in managing the memory pool. This change should make easier to perform future improvements such as making it easier to shrink the memory pool used by Kestrel.

IAsyncDisposable supported

IAsyncDisposable is now available for controllers, Razor Pages, and View Components. Asynchronous versions have been added to the relevant interfaces in factories and activators:

- The new methods offer a default interface implementation that delegates to the synchronous version and calls Dispose.
- The implementations override the default implementation and handle disposing `IAsyncDisposable` implementations.
- The implementations favor `IAsyncDisposable` over `IDisposable` when both interfaces are implemented.
- Extenders must override the new methods included to support `IAsyncDisposable` instances.

`IAsyncDisposable` is beneficial when working with:

- Asynchronous enumerators, for example, in asynchronous streams.
- Unmanaged resources that have resource-intensive I/O operations to release.

When implementing this interface, use the `DisposeAsync` method to release resources.

Consider a controller that creates and uses a Utf8JsonWriter. `Utf8JsonWriter` is an `IAsyncDisposable` resource:

```
C#

public class HomeController : Controller, IAsyncDisposable
{
    private Utf8JsonWriter? _jsonWriter;
    private readonly ILogger<HomeController> _logger;
```

```
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
        _jsonWriter = new Utf8JsonWriter(new MemoryStream());
    }
```

`IAsyncDisposable` must implement `DisposeAsync`:

C#

```
public async ValueTask DisposeAsync()
{
    if (_jsonWriter is not null)
    {
        await _jsonWriter.DisposeAsync();
    }

    _jsonWriter = null;
}
```

Vcpkg port for SignalR C++ client

Vcpkg is a cross-platform command-line package manager for C and C++ libraries. We've recently added a port to `vcpkg` to add `CMake` native support for the SignalR C++ client. `vcpkg` also works with MSBuild.

The SignalR client can be added to a CMake project with the following snippet when the vcpkg is included in the toolchain file:

.NET CLI

```
find_package(microsoft-signalr CONFIG REQUIRED)
link_libraries(microsoft-signalr::microsoft-signalr)
```

With the preceding snippet, the SignalR C++ client is ready to use #include and used in a project without any additional configuration. For a complete example of a C++ application that utilizes the SignalR C++ client, see the halter73/SignalR-Client-Cpp-Sample repository.

Blazor

Project template changes

Several project template changes were made for Blazor apps, including the use of the `Pages/_Layout.cshtml` file for layout content that appeared in the `_Host.cshtml` file for earlier Blazor Server apps. Study the changes by creating an app from a 6.0 project template or accessing the ASP.NET Core reference source for the project templates:

- Blazor Server
- Blazor WebAssembly

Blazor WebAssembly native dependencies support

Blazor WebAssembly apps can use native dependencies built to run on WebAssembly. For more information, see ASP.NET Core Blazor WebAssembly native dependencies.

WebAssembly Ahead-of-time (AOT) compilation and runtime relinking

Blazor WebAssembly supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size. Relinking the .NET WebAssembly runtime trims unused runtime code and thus improves download speed. For more information, see Ahead-of-time (AOT) compilation and Runtime relinking.

Persist prerendered state

Blazor supports persisting state in a prerendered page so that the state doesn't need to be recreated when the app is fully loaded. For more information, see Prerender and integrate ASP.NET Core Razor components.

Error boundaries

Error boundaries provide a convenient approach for handling exceptions on the UI level. For more information, see Handle errors in ASP.NET Core Blazor apps.

SVG support

The <foreignObject> element element is supported to display arbitrary HTML within an SVG. For more information, see ASP.NET Core Razor components.

Blazor Server support for byte array transfer in JS Interop

Blazor supports optimized byte array JS interop that avoids encoding and decoding byte arrays into Base64. For more information, see the following resources:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Query string enhancements

Support for working with query strings is improved. For more information, see ASP.NET Core Blazor routing and navigation.

Binding to select multiple

Binding supports multiple option selection with `<input>` elements. For more information, see the following resources:

- ASP.NET Core Blazor data binding
- ASP.NET Core Blazor input components

Head (`<head>`) content control

Razor components can modify the HTML `<head>` element content of a page, including setting the page's title (`<title>` element) and modifying metadata (`<meta>` elements). For more information, see Control <head> content in ASP.NET Core Blazor apps.

Generate Angular and React components

Generate framework-specific JavaScript components from Razor components for web frameworks, such as Angular or React. For more information, see ASP.NET Core Razor components.

Render components from JavaScript

Render Razor components dynamically from JavaScript for existing JavaScript apps. For more information, see ASP.NET Core Razor components.

Custom elements

Experimental support is available for building custom elements, which use standard HTML interfaces. For more information, see ASP.NET Core Razor components.

Infer component generic types from ancestor components

An ancestor component can cascade a type parameter by name to descendants using the new `[CascadingTypeParameter]` attribute. For more information, see ASP.NET Core Razor components.

Dynamically rendered components

Use the new built-in `DynamicComponent` component to render components by type. For more information, see Dynamically-rendered ASP.NET Core Razor components.

Improved Blazor accessibility

Use the new `FocusOnNavigate` component to set the UI focus to an element based on a CSS selector after navigating from one page to another. For more information, see ASP.NET Core Blazor routing and navigation.

Custom event argument support

Blazor supports custom event arguments, which enable you to pass arbitrary data to .NET event handlers with custom events. For more information, see ASP.NET Core Blazor event handling.

Required parameters

Apply the new `[EditorRequired]` attribute to specify a required component parameter. For more information, see ASP.NET Core Razor components.

Collocation of JavaScript files with pages, views, and components

Collocate JavaScript files for pages, views, and Razor components as a convenient way to organize scripts in an app. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

JavaScript initializers

JavaScript initializers execute logic before and after a Blazor app loads. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Streaming JavaScript interop

Blazor now supports streaming data directly between .NET and JavaScript. For more information, see the following resources:

- Stream from .NET to JavaScript
- Stream from JavaScript to .NET

Generic type constraints

Generic type parameters are now supported. For more information, see ASP.NET Core Razor components.

WebAssembly deployment layout

Use a deployment layout to enable Blazor WebAssembly app downloads in restricted security environments. For more information, see Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps.

New Blazor articles

In addition to the Blazor features described in the preceding sections, new Blazor articles are available on the following subjects:

- ASP.NET Core Blazor file downloads: Learn how to download a file using native `byte[]` streaming interop to ensure efficient transfer to the client.
- Display images and documents in ASP.NET Core Blazor: Discover how to work with images and documents in Blazor apps, including how to stream image and document data.

Build Blazor Hybrid apps with .NET MAUI, WPF, and Windows Forms

Use Blazor Hybrid to blend desktop and mobile native client frameworks with .NET and Blazor:

- .NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.
- Blazor Hybrid apps can be built with Windows Presentation Foundation (WPF) and Windows Forms frameworks.

> **ⓘ Important**
>
> Blazor Hybrid is in preview and shouldn't be used in production apps until final release.

For more information, see the following resources:

- Preview ASP.NET Core Blazor Hybrid documentation
- What is .NET MAUI?
- Microsoft .NET Blog (category: ".NET MAUI")

Kestrel

HTTP/3 is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

Kestrel now supports HTTP/3. For more information, see Use HTTP/3 with the ASP.NET Core Kestrel web server and the blog entry HTTP/3 support in .NET 6.

New Kestrel logging categories for selected logging

Prior to this change, enabling verbose logging for Kestrel was prohibitively expensive as all of Kestrel shared the `Microsoft.AspNetCore.Server.Kestrel` logging category name. `Microsoft.AspNetCore.Server.Kestrel` is still available, but the following new subcategories allow for more control of logging:

- `Microsoft.AspNetCore.Server.Kestrel` (current category): `ApplicationError`, `ConnectionHeadResponseBodyWrite`, `ApplicationNeverCompleted`, `RequestBodyStart`, `RequestBodyDone`, `RequestBodyNotEntirelyRead`, `RequestBodyDrainTimedOut`, `ResponseMinimumDataRateNotSatisfied`, `InvalidResponseHeaderRemoved`, `HeartbeatSlow`.
- `Microsoft.AspNetCore.Server.Kestrel.BadRequests`: `ConnectionBadRequest`, `RequestProcessingError`, `RequestBodyMinimumDataRateNotSatisfied`.
- `Microsoft.AspNetCore.Server.Kestrel.Connections`: `ConnectionAccepted`, `ConnectionStart`, `ConnectionStop`, `ConnectionPause`, `ConnectionResume`,

`ConnectionKeepAlive`, `ConnectionRejected`, `ConnectionDisconnect`, `NotAllConnectionsClosedGracefully`, `NotAllConnectionsAborted`, `ApplicationAbortedConnection`.

- `Microsoft.AspNetCore.Server.Kestrel.Http2`: `Http2ConnectionError`, `Http2ConnectionClosing`, `Http2ConnectionClosed`, `Http2StreamError`, `Http2StreamResetAbort`, `HPackDecodingError`, `HPackEncodingError`, `Http2FrameReceived`, `Http2FrameSending`, `Http2MaxConcurrentStreamsReached`.
- `Microsoft.AspNetCore.Server.Kestrel.Http3`: `Http3ConnectionError`, `Http3ConnectionClosing`, `Http3ConnectionClosed`, `Http3StreamAbort`, `Http3FrameReceived`, `Http3FrameSending`.

Existing rules continue to work, but you can now be more selective on which rules you enable. For example, the observability overhead of enabling `Debug` logging for just bad requests is greatly reduced and can be enabled with the following configuration:

XML

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.Kestrel.BadRequests": "Debug"
    }
  }
```

Log filtering applies rules with the longest matching category prefix. For more information, see How filtering rules are applied

Emit KestrelServerOptions via EventSource event

The KestrelEventSource emits a new event containing the JSON-serialized KestrelServerOptions when enabled with verbosity `EventLevel.LogAlways`. This event makes it easier to reason about the server behavior when analyzing collected traces. The following JSON is an example of the event payload:

JSON

```
{
  "AllowSynchronousIO": false,
  "AddServerHeader": true,
  "AllowAlternateSchemes": false,
  "AllowResponseHeaderCompression": true,
  "EnableAltSvc": false,
```

```
  "IsDevCertLoaded": true,
  "RequestHeaderEncodingSelector": "default",
  "ResponseHeaderEncodingSelector": "default",
  "Limits": {
    "KeepAliveTimeout": "00:02:10",
    "MaxConcurrentConnections": null,
    "MaxConcurrentUpgradedConnections": null,
    "MaxRequestBodySize": 30000000,
    "MaxRequestBufferSize": 1048576,
    "MaxRequestHeaderCount": 100,
    "MaxRequestHeadersTotalSize": 32768,
    "MaxRequestLineSize": 8192,
    "MaxResponseBufferSize": 65536,
    "MinRequestBodyDataRate": "Bytes per second: 240, Grace Period:
00:00:05",
    "MinResponseDataRate": "Bytes per second: 240, Grace Period: 00:00:05",
    "RequestHeadersTimeout": "00:00:30",
    "Http2": {
      "MaxStreamsPerConnection": 100,
      "HeaderTableSize": 4096,
      "MaxFrameSize": 16384,
      "MaxRequestHeaderFieldSize": 16384,
      "InitialConnectionWindowSize": 131072,
      "InitialStreamWindowSize": 98304,
      "KeepAlivePingDelay": "10675199.02:48:05.4775807",
      "KeepAlivePingTimeout": "00:00:20"
    },
    "Http3": {
      "HeaderTableSize": 0,
      "MaxRequestHeaderFieldSize": 16384
    }
  },
  "ListenOptions": [
    {
      "Address": "https://127.0.0.1:7030",
      "IsTls": true,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "https://[::1]:7030",
      "IsTls": true,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "http://127.0.0.1:5030",
      "IsTls": false,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "http://[::1]:5030",
      "IsTls": false,
      "Protocols": "Http1AndHttp2"
    }
  ]
}
```

New DiagnosticSource event for rejected HTTP requests

Kestrel now emits a new `DiagnosticSource` event for HTTP requests rejected at the server layer. Prior to this change, there was no way to observe these rejected requests. The new `DiagnosticSource` event `Microsoft.AspNetCore.Server.Kestrel.BadRequest` contains a IBadRequestExceptionFeature that can be used to introspect the reason for rejecting the request.

C#

```
using Microsoft.AspNetCore.Http.Features;
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var diagnosticSource = app.Services.GetRequiredService<DiagnosticListener>
();
using var badRequestListener = new BadRequestEventListener(diagnosticSource,
    (badRequestExceptionFeature) =>
{
    app.Logger.LogError(badRequestExceptionFeature.Error, "Bad request
received");
});
app.MapGet("/", () => "Hello world");

app.Run();

class BadRequestEventListener : IObserver<KeyValuePair<string, object>>,
IDisposable
{
    private readonly IDisposable _subscription;
    private readonly Action<IBadRequestExceptionFeature> _callback;

    public BadRequestEventListener(DiagnosticListener diagnosticListener,
                                   Action<IBadRequestExceptionFeature>
callback)
    {
        _subscription = diagnosticListener.Subscribe(this!, IsEnabled);
        _callback = callback;
    }
    private static readonly Predicate<string> IsEnabled = (provider) =>
provider switch
    {
        "Microsoft.AspNetCore.Server.Kestrel.BadRequest" => true,
        _ => false
    };
    public void OnNext(KeyValuePair<string, object> pair)
    {
        if (pair.Value is IFeatureCollection featureCollection)
```

```
        {
            var badRequestFeature =
featureCollection.Get<IBadRequestExceptionFeature>();

            if (badRequestFeature is not null)
            {
                _callback(badRequestFeature);
            }
        }
    }
    public void OnError(Exception error) { }
    public void OnCompleted() { }
    public virtual void Dispose() => _subscription.Dispose();
}
```

For more information, see Logging and diagnostics in Kestrel.

Create a ConnectionContext from an Accept Socket

The new SocketConnectionContextFactory makes it possible to create a ConnectionContext from an accepted socket. This makes it possible to build a custom socket-based IConnectionListenerFactory without losing out on all the performance work and pooling happening in SocketConnection.

See this example of a custom IConnectionListenerFactory which shows how to use this `SocketConnectionContextFactory`.

Kestrel is the default launch profile for Visual Studio

The default launch profile for all new dotnet web projects is Kestrel. Starting Kestrel is significantly faster and results in a more responsive experience while developing apps.

IIS Express is still available as a launch profile for scenarios such as Windows Authentication or port sharing.

Localhost ports for Kestrel are random

See Template generated ports for Kestrel in this document for more information.

Authentication and authorization

Authentication servers

.NET 3 to .NET 5 used IdentityServer4 as part of our template to support the issuing of JWT tokens for SPA and Blazor applications. The templates now use the Duende Identity Server.

If you are extending the identity models and are updating existing projects, update the namespaces in your code from `IdentityServer4.IdentityServer` to `Duende.IdentityServer` and follow their migration instructions.

The license model for Duende Identity Server has changed to a reciprocal license, which may require license fees when it's used commercially in production. See the Duende license page for more details.

Delayed client certificate negotiation

Developers can now opt-in to using delayed client certificate negotiation by specifying ClientCertificateMode.DelayCertificate on the HttpsConnectionAdapterOptions. This only works with HTTP/1.1 connections because HTTP/2 forbids delayed certificate renegotiation. The caller of this API must buffer the request body before requesting the client certificate:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Https;
using Microsoft.AspNetCore.WebUtilities;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseKestrel(options =>
{
    options.ConfigureHttpsDefaults(adapterOptions =>
    {
        adapterOptions.ClientCertificateMode =
ClientCertificateMode.DelayCertificate;
    });
});

var app = builder.Build();
app.Use(async (context, next) =>
{
    bool desiredState = GetDesiredState();
    // Check if your desired criteria is met
    if (desiredState)
    {
        // Buffer the request body
        context.Request.EnableBuffering();
        var body = context.Request.Body;
        await body.DrainAsync(context.RequestAborted);
        body.Position = 0;
```

```
        // Request client certificate
        var cert = await context.Connection.GetClientCertificateAsync();

        //  Disable buffering on future requests if the client doesn't
provide a cert
    }
    await next(context);
});


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

`OnCheckSlidingExpiration` event for controlling cookie renewal

Cookie authentication sliding expiration can now be customized or suppressed using the new OnCheckSlidingExpiration. For example, this event can be used by a single-page app that needs to periodically ping the server without affecting the authentication session.

Miscellaneous

Hot Reload

Quickly make UI and code updates to running apps without losing app state for faster and more productive developer experience using Hot Reload. For more information, see .NET Hot Reload support for ASP.NET Core and Update on .NET Hot Reload progress and Visual Studio 2022 Highlights.

Improved single-page app (SPA) templates

The ASP.NET Core project templates have been updated for Angular and React to use an improved pattern for single-page apps that is more flexible and more closely aligns with common patterns for modern front-end web development.

Previously, the ASP.NET Core template for Angular and React used specialized middleware during development to launch the development server for the front-end framework and then proxy requests from ASP.NET Core to the development server. The logic for launching the front-end development server was specific to the command-line interface for the corresponding front-end framework. Supporting additional front-end frameworks using this pattern meant adding additional logic to ASP.NET Core.

The updated ASP.NET Core templates for Angular and React in .NET 6 flips this arrangement around and take advantage of the built-in proxying support in the development servers of most modern front-end frameworks. When the ASP.NET Core app is launched, the front-end development server is launched just as before, but the development server is configured to proxy requests to the backend ASP.NET Core process. All of the front-end specific configuration to setup proxying is part of the app, not ASP.NET Core. Setting up ASP.NET Core projects to work with other front-end frameworks is now straight-forward: setup the front-end development server for the chosen framework to proxy to the ASP.NET Core backend using the pattern established in the Angular and React templates.

The startup code for the ASP.NET Core app no longer needs any single-page app-specific logic. The logic for starting the front-end development server during development is injecting into the app at runtime by the new Microsoft.AspNetCore.SpaProxy package. Fallback routing is handled using endpoint routing instead of SPA-specific middleware.

Templates that follow this pattern can still be run as a single project in Visual Studio or using `dotnet run` from the command-line. When the app is published, the front-end code in the *ClientApp* folder is built and collected as before into the web root of the host ASP.NET Core app and served as static files. Scripts included in the template configure the front-end development server to use HTTPS using the ASP.NET Core development certificate.

Draft HTTP/3 support in .NET 6

HTTP/3 is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

See the blog entry HTTP/3 support in .NET 6.

Nullable Reference Type Annotations

Portions of the ASP.NET Core 6.0 source code has had nullability annotations applied.

By utilizing the new Nullable feature in C# 8, ASP.NET Core can provide additional compile-time safety in the handling of reference types. For example, protecting against `null` reference exceptions. Projects that have opted in to using nullable annotations may see new build-time warnings from ASP.NET Core APIs.

To enable nullable reference types, add the following property to project files:

```
```

XML

```xml
<PropertyGroup>
    <Nullable>enable</Nullable>
</PropertyGroup>
```

For more information, see Nullable reference types.

Source Code Analysis

Several .NET compiler platform analyzers were added that inspect application code for problems such as incorrect middleware configuration or order, routing conflicts, etc. For more information, see Code analysis in ASP.NET Core apps.

Web app template improvements

The web app templates:

- Use the new minimal hosting model.
- Significantly reduces the number of files and lines of code required to create an app. For example, the ASP.NET Core empty web app creates one C# file with four lines of code and is a complete app.
- Unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.
- Uses top-level statements to minimize the code required for an app.
- Uses global using directives to eliminate or minimize the number of using statement lines required.

Template generated ports for Kestrel

Random ports are assigned during project creation for use by the Kestrel web server. Random ports help minimize a port conflict when multiple projects are run on the same machine.

When a project is created, a random HTTP port between 5000-5300 and a random HTTPS port between 7000-7300 is specified in the generated `Properties/launchSettings.json` file. The ports can be changed in the `Properties/launchSettings.json` file. If no port is specified, Kestrel defaults to the HTTP 5000 and HTTPS 5001 ports. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

New logging defaults

The following changes were made to both `appsettings.json` and `appsettings.Development.json`:

```diff
- "Microsoft": "Warning",
- "Microsoft.Hosting.Lifetime": "Information"
+ "Microsoft.AspNetCore": "Warning"
```

The change from `"Microsoft": "Warning"` to `"Microsoft.AspNetCore": "Warning"` results in logging all informational messages from the `Microsoft` namespace ***except*** `Microsoft.AspNetCore`. For example, `Microsoft.EntityFrameworkCore` is now logged at the informational level.

Developer exception page Middleware added automatically

In the development environment, the DeveloperExceptionPageMiddleware is added by default. It's no longer necessary to add the following code to web UI apps:

```C#
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

Support for Latin1 encoded request headers in HttpSysServer

`HttpSysServer` now supports decoding request headers that are `Latin1` encoded by setting the UseLatin1RequestHeaders property on HttpSysOptions to `true`:

```C#
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseHttpSys(o => o.UseLatin1RequestHeaders = true);

var app = builder.Build();

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

app.Run();
```

The ASP.NET Core Module logs include timestamps and PID

The ASP.NET Core Module (ANCM) for IIS (ANCM) enhanced diagnostic logs include timestamps and PID of the process emitting the logs. Logging timestamps and PID makes it easier to diagnose issues with overlapping process restarts in IIS when multiple IIS worker processes are running.

The resulting logs now resemble the sample output show below:

.NET CLI

```
[2021-07-28T19:23:44.076Z, PID: 11020] [aspnetcorev2.dll] Initializing logs
for 'C:\<path>\aspnetcorev2.dll'. Process Id: 11020. File Version:
16.0.21209.0. Description: IIS ASP.NET Core Module V2. Commit:
96475a2acdf50d7599ba8e96583fa73efbe27912.
[2021-07-28T19:23:44.079Z, PID: 11020] [aspnetcorev2.dll] Resolving hostfxr
parameters for application: '.\InProcessWebSite.exe' arguments: '' path:
'C:\Temp\e86ac4e9ced24bb6bacf1a9415e70753\'
[2021-07-28T19:23:44.080Z, PID: 11020] [aspnetcorev2.dll] Known dotnet.exe
location: ''
```

Configurable unconsumed incoming buffer size for IIS

The IIS server previously only buffered 64 KiB of unconsumed request bodies. The 64 KiB buffering resulted in reads being constrained to that maximum size, which impacts the performance with large incoming bodies such as uploads. In .NET 6 , the default buffer size changes from 64 KiB to 1 MiB which should improve throughput for large uploads. In our tests, a 700 MiB upload that used to take 9 seconds now only takes 2.5 seconds.

The downside of a larger buffer size is an increased per-request memory consumption when the app isn't quickly reading from the request body. So, in addition to changing the default buffer size, the buffer size configurable, allowing apps to configure the buffer size based on workload.

View Components Tag Helpers

Consider a view component with an optional parameter, as shown in the following code:

C#

```
class MyViewComponent
{
```

```
    IViewComponentResult Invoke(bool showSomething = false) { ... }
}
```

With ASP.NET Core 6, the tag helper can be invoked without having to specify a value for the `showSomething` parameter:

razor

```
<vc:my />
```

Angular template updated to Angular 12

The ASP.NET Core 6.0 template for Angular now uses Angular 12.

The React template has been updated to React 17.

Configurable buffer threshold before writing to disk in Json.NET output formatter

Note: We recommend using the System.Text.Json output formatter except when the `Newtonsoft.Json` serializer is required for compatibility reasons. The `System.Text.Json` serializer is fully `async` and works efficiently for larger payloads.

The `Newtonsoft.Json` output formatter by default buffers responses up to 32 KiB in memory before buffering to disk. This is to avoid performing synchronous IO, which can result in other side-effects such as thread starvation and application deadlocks. However, if the response is larger than 32 KiB, considerable disk I/O occurs. The memory threshold is now configurable via the MvcNewtonsoftJsonOptions.OutputFormatterMemoryBufferThreshold property before buffering to disk:

C#

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

builder.Services.AddRazorPages()
            .AddNewtonsoftJson(options =>
            {
                options.OutputFormatterMemoryBufferThreshold = 48 * 1024;
            });

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

For more information, see this GitHub pull request and the NewtonsoftJsonOutputFormatterTest.cs file.

Faster get and set for HTTP headers

New APIs were added to expose all common headers available on Microsoft.Net.Http.Headers.HeaderNames as properties on the IHeaderDictionary resulting in an easier to use API. For example, the in-line middleware in the following code gets and sets both request and response headers using the new APIs:

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

var app = builder.Build();

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

app.Use(async (context, next) =>
{
    var hostHeader = context.Request.Headers.Host;
    app.Logger.LogInformation("Host header: {host}", hostHeader);
    context.Response.Headers.XPoweredBy = "ASP.NET Core 6.0";
    await next.Invoke(context);
    var dateHeader = context.Response.Headers.Date;
    app.Logger.LogInformation("Response date: {date}", dateHeader);
});

app.Run();
```

For implemented headers the get and set accessors are implemented by going directly to the field and bypassing the lookup. For non-implemented headers, the accessors can bypass the initial lookup against implemented headers and directly perform the `Dictionary<string, StringValues>` lookup. Avoiding the lookup results in faster access for both scenarios.

Async streaming

ASP.NET Core now supports asynchronous streaming from controller actions and responses from the JSON formatter. Returning an `IAsyncEnumerable` from an action no longer buffers the response content in memory before it gets sent. Not buffering helps reduce memory usage when returning large datasets that can be asynchronously enumerated.

Note that Entity Framework Core provides implementations of `IAsyncEnumerable` for querying the database. The improved support for `IAsyncEnumerable` in ASP.NET Core in .NET 6 can make using EF Core with ASP.NET Core more efficient. For example, the following code no longer buffers the product data into memory before sending the response:

C#

```
public IActionResult GetMovies()
{
    return Ok(_context.Movie);
}
```

However, when using lazy loading in EF Core, this new behavior may result in errors due to concurrent query execution while the data is being enumerated. Apps can revert back to the previous behavior by buffering the data:

C#

```
public async Task<IActionResult> GetMovies2()
{
    return Ok(await _context.Movie.ToListAsync());
}
```

See the related announcement for additional details about this change in behavior.

HTTP logging middleware

HTTP logging is a new built-in middleware that logs information about HTTP requests and HTTP responses including the headers and entire body:

C#

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

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

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

app.Run();
```

Navigating to `/` with the previous code logs information similar to the following output:

.NET CLI

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /
      Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-US,en;q=0.9
      Cache-Control: max-age=0
      Connection: close
      Cookie: [Redacted]
      Host: localhost:44372
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.54 Safari/537.36
Edg/95.0.1020.30
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      upgrade-insecure-requests: [Redacted]
      sec-fetch-site: [Redacted]
      sec-fetch-mode: [Redacted]
      sec-fetch-user: [Redacted]
      sec-fetch-dest: [Redacted]
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8
```

The preceding output was enabled with the following `appsettings.Development.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware":
"Information"
    }
  }
}
```

HTTP logging provides logs of:

- HTTP Request information

- Common properties
- Headers
- Body
- HTTP Response information

To configure the HTTP logging middleware, specify HttpLoggingOptions:

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpLogging(logging =>
{
    // Customize HTTP logging.
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("My-Request-Header");
    logging.ResponseHeaders.Add("My-Response-Header");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
});

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

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

app.Run();
```

IConnectionSocketFeature

The IConnectionSocketFeature request feature provides access to the underlying accept socket associated with the current request. It can be accessed via the FeatureCollection on `HttpContext`.

For example, the following app sets the LingerState property on the accepted socket:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
listenOptions.Use((connection, next) =>
    {
        var socketFeature =
connection.Features.Get<IConnectionSocketFeature>();
        socketFeature.Socket.LingerState = new LingerOption(true, seconds:
```

```
10);
        return next();
    }));
});
var app = builder.Build();
app.MapGet("/", (Func<string>)(() => "Hello world"));
await app.RunAsync();
```

Generic type constraints in Razor

When defining generic type parameters in Razor using the `@typeparam` directive, generic type constraints can now be specified using the standard C# syntax:

Smaller SignalR, Blazor Server, and MessagePack scripts

The SignalR, MessagePack, and Blazor Server scripts are now significantly smaller, enabling smaller downloads, less JavaScript parsing and compiling by the browser, and faster start-up. The size reductions:

- `signalr.js`: 70%
- `blazor.server.js`: 45%

The smaller scripts are a result of a community contribution from Ben Adams. For more information on the details of the size reduction, see Ben's GitHub pull request.

Enable Redis profiling sessions

A community contribution from Gabriel Lucaci enables Redis profiling session with Microsoft.Extensions.Caching.StackExchangeRedis:

C#

```
using StackExchange.Redis.Profiling;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddStackExchangeRedisCache(options =>
{
    options.ProfilingSession = () => new ProfilingSession();
});
```

For more information, see StackExchange.Redis Profiling.

Shadow copying in IIS

An experimental feature has been added to the ASP.NET Core Module (ANCM) for IIS to add support for shadow copying application assemblies. Currently .NET locks application binaries when running on Windows making it impossible to replace binaries when the app is running. While our recommendation remains to use an app offline file, we recognize there are certain scenarios (for example FTP deployments) where it isn't possible to do so.

In such scenarios, enable shadow copying by customizing the ASP.NET Core module handler settings. In most cases, ASP.NET Core apps do not have a `web.config` checked into source control that you can modify. In ASP.NET Core, `web.config` is ordinarily generated by the SDK. The following sample `web.config` can be used to get started:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!-- To customize the asp.net core module uncomment and edit the following
section.
  For more info see https://go.microsoft.com/fwlink/?linkid=838655 -->

  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout">
      <handlerSettings>
        <handlerSetting name="experimentalEnableShadowCopy" value="true" />
        <handlerSetting name="shadowCopyDirectory"
value="../ShadowCopyDirectory/" />
        <!-- Only enable handler logging if you encounter issues-->
        <!--<handlerSetting name="debugFile" value=".\logs\aspnetcore-
debug.log" />-->
        <!--<handlerSetting name="debugLevel" value="FILE,TRACE" />-->
      </handlerSettings>
    </aspNetCore>
  </system.webServer>
</configuration>
```

Shadow copying in IIS is an experimental feature that is not guaranteed to be part of ASP.NET Core. Please leave feedback on IIS Shadow copying in this GitHub issue.

Additional resources

- Code samples migrated to the new minimal hosting model in 6.0

- What's new in .NET 6

What's new in ASP.NET Core 5.0

Article • 09/27/2024

This article highlights the most significant changes in ASP.NET Core 5.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Model binding DateTime as UTC

Model binding now supports binding UTC time strings to `DateTime`. If the request contains a UTC time string, model binding binds it to a UTC `DateTime`. For example, the following time string is bound the UTC `DateTime`:

`https://example.com/mycontroller/myaction?time=2019-06-14T02%3A30%3A04.0576719Z`

Model binding and validation with C# 9 record types

C# 9 record types can be used with model binding in an MVC controller or a Razor Page. Record types are a good way to model data being transmitted over the network.

For example, the following `PersonController` uses the `Person` record type with model binding and form validation:

C#

```
public record Person([Required] string Name, [Range(0, 150)] int Age);

public class PersonController
{
   public IActionResult Index() => View();

   [HttpPost]
   public IActionResult Index(Person person)
   {
          // ...
   }
}
```

The `Person/Index.cshtml` file:

CSHTML

```
@model Person

<label>Name: <input asp-for="Model.Name" /></label>
<span asp-validation-for="Model.Name" />

<label>Age: <input asp-for="Model.Age" /></label>
<span asp-validation-for="Model.Age" />
```

Improvements to DynamicRouteValueTransformer

ASP.NET Core 3.1 introduced DynamicRouteValueTransformer as a way to use custom endpoint to dynamically select an MVC controller action or a Razor page. ASP.NET Core 5.0 apps can pass state to a `DynamicRouteValueTransformer` and filter the set of endpoints chosen.

Miscellaneous

- The [Compare] attribute can be applied to properties on a Razor Page model.
- Parameters and properties bound from the body are considered required by default.

Web API

OpenAPI Specification on by default

OpenAPI Specification is an industry standard for describing HTTP APIs and integrating them into complex business processes or with third parties. OpenAPI is widely supported by all cloud providers and many API registries. Apps that emit OpenAPI documents from web APIs have a variety of new opportunities in which those APIs can be used. In partnership with the maintainers of the open-source project Swashbuckle.AspNetCore, the ASP.NET Core API template contains a NuGet dependency on Swashbuckle. Swashbuckle is a popular open-source NuGet package that emits OpenAPI documents dynamically. Swashbuckle does this by introspecting over the API controllers and generating the OpenAPI document at run-time, or at build time using the Swashbuckle CLI.

In ASP.NET Core 5.0, the web API templates enable the OpenAPI support by default. To disable OpenAPI:

- From the command line:

.NET CLI

```
dotnet new webapi --no-openapi true
```

- From Visual Studio: Uncheck **Enable OpenAPI support**.

All `.csproj` files created for web API projects contain the Swashbuckle.AspNetCore NuGet package reference.

XML

```
<ItemGroup>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.5.1" />
</ItemGroup>
```

The template generated code contains code in `Startup.ConfigureServices` that activates OpenAPI document generation:

C#

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "WebApp1", Version =
"v1" });
    });
}
```

The `Startup.Configure` method adds the Swashbuckle middleware, which enables the:

- Document generation process.
- Swagger UI page by default in development mode.

The template generated code won't accidentally expose the API's description when publishing to production.

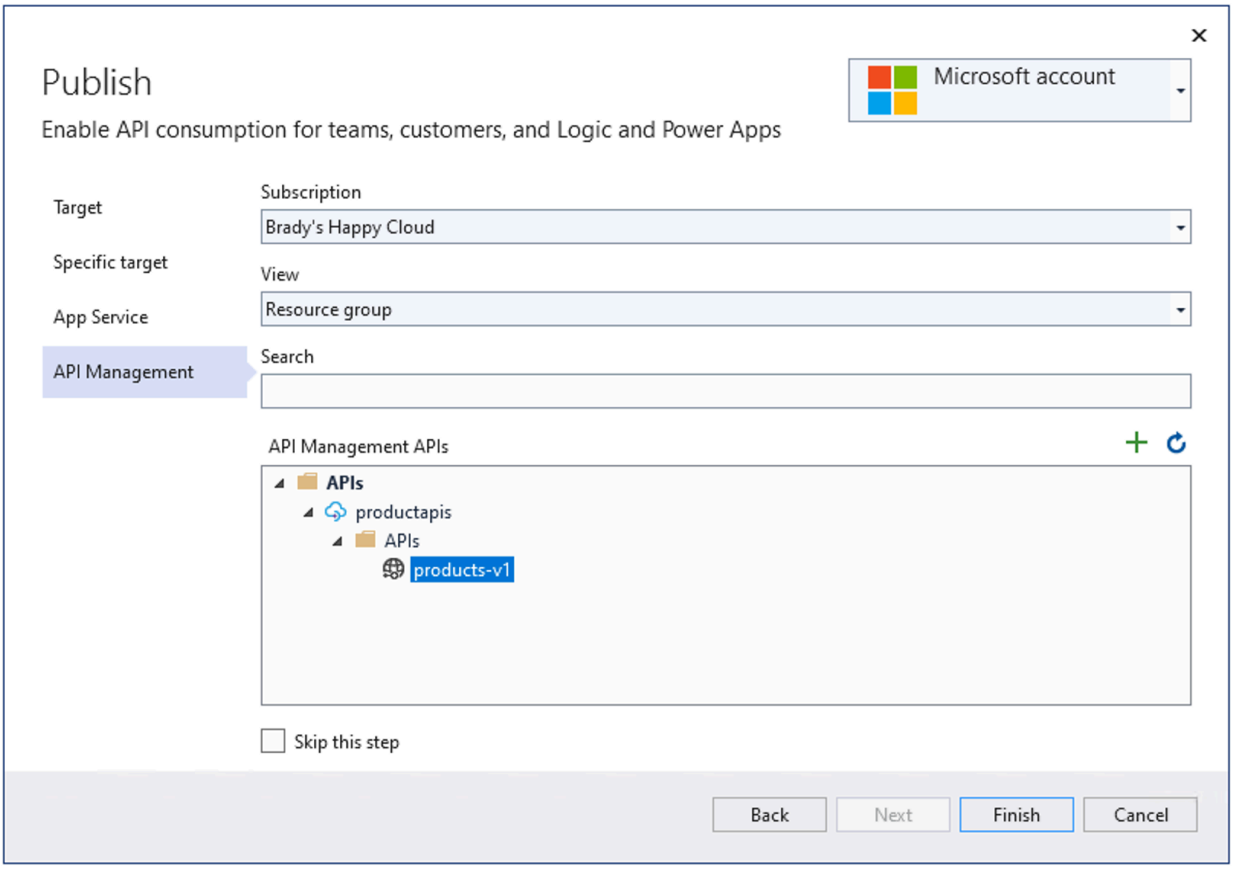
C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();  // UseSwaggerUI Protected by if
```

```
(env.IsDevelopment())
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
                            "WebApp1 v1"));
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

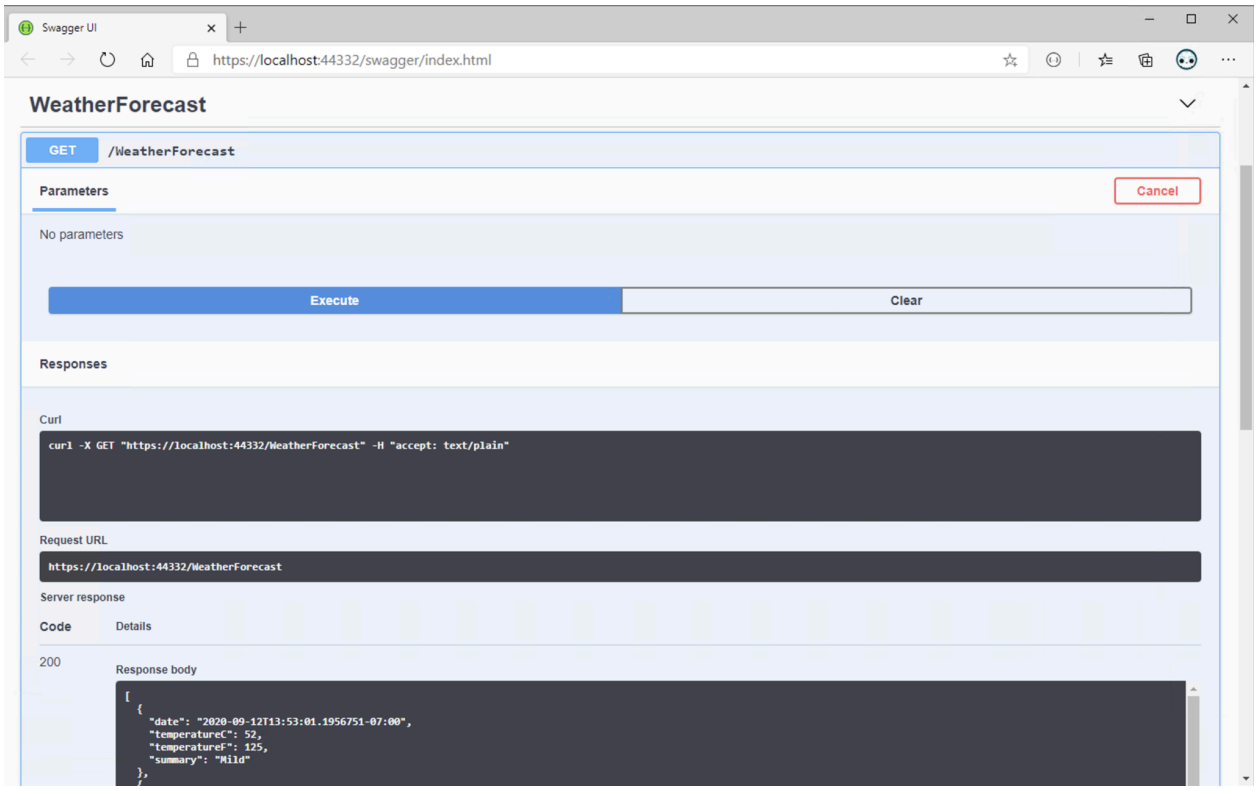
Azure API Management Import

When ASP.NET Core API projects enable OpenAPI, the Visual Studio 2019 version 16.8 and later publishing automatically offer an additional step in the publishing flow. Developers who use Azure API Management have an opportunity to automatically import the APIs into Azure API Management during the publish flow:



Better launch experience for web API projects

With OpenAPI enabled by default, the app launching experience (F5) for web API developers significantly improves. With ASP.NET Core 5.0, the web API template comes pre-configured to load up the Swagger UI page. The Swagger UI page provides both the documentation added for the published API, and enables testing the APIs with a single click.



Blazor

Performance improvements

For .NET 5, we made significant improvements to .NET WebAssembly runtime performance with a specific focus on complex UI rendering and JSON serialization. In our performance tests, Blazor WebAssembly in .NET 5 is two to three times faster for most scenarios. For more information, see ASP.NET Blog: ASP.NET Core updates in .NET 5 Release Candidate 1.

CSS isolation

Blazor now supports defining CSS styles that are scoped to a given component. Component-specific CSS styles make it easier to reason about the styles in an app and to avoid unintentional side effects of global styles. For more information, see ASP.NET Core Blazor CSS isolation.

New `InputFile` component

The `InputFile` component allows reading one or more files selected by a user for upload. For more information, see ASP.NET Core Blazor file uploads.

New `InputRadio` and `InputRadioGroup` components

Blazor has built-in `InputRadio` and `InputRadioGroup` components that simplify data binding to radio button groups with integrated validation. For more information, see ASP.NET Core Blazor input components.

Component virtualization

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support. For more information, see ASP.NET Core Razor component virtualization.

`ontoggle` event support

Blazor events now support the `ontoggle` DOM event. For more information, see ASP.NET Core Blazor event handling.

Set UI focus in Blazor apps

Use the `FocusAsync` convenience method on element references to set the UI focus to that element. For more information, see ASP.NET Core Blazor event handling.

Custom validation CSS class attributes

Custom validation CSS class attributes are useful when integrating with CSS frameworks, such as Bootstrap. For more information, see ASP.NET Core Blazor forms validation.

IAsyncDisposable support

Razor components now support the IAsyncDisposable interface for the asynchronous release of allocated resources.

JavaScript isolation and object references

Blazor enables JavaScript isolation in standard JavaScript modules. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Form components support display name

The following built-in components support display names with the `DisplayName` parameter:

- `InputDate`
- `InputNumber`
- `InputSelect`

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

Catch-all route parameters

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. For more information, see ASP.NET Core Blazor routing and navigation.

Debugging improvements

Debugging Blazor WebAssembly apps is improved in ASP.NET Core 5.0. Additionally, debugging is now supported on Visual Studio for Mac. For more information, see Debug ASP.NET Core Blazor apps.

Microsoft Identity v2.0 and MSAL v2.0

Blazor security now uses Microsoft Identity v2.0 (Microsoft.Identity.Web and Microsoft.Identity.Web.UI) and MSAL v2.0. For more information, see the topics in the Blazor Security and Identity node.

Protected Browser Storage for Blazor Server

Blazor Server apps can now use built-in support for storing app state in the browser that has been protected from tampering using ASP.NET Core data protection. Data can be stored in either local browser storage or session storage. For more information, see ASP.NET Core Blazor state management.

Blazor WebAssembly prerendering

Component integration is improved across hosting models, and Blazor WebAssembly apps can now prerender output on the server.

Trimming/linking improvements

Blazor WebAssembly performs Intermediate Language (IL) trimming/linking during a build to trim unnecessary IL from the app's output assemblies. With the release of ASP.NET Core 5.0, Blazor WebAssembly performs improved trimming with additional configuration options. For more information, see Configure the Trimmer for ASP.NET Core Blazor and Trimming options.

Browser compatibility analyzer

Blazor WebAssembly apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For more information, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Lazy load assemblies

Blazor WebAssembly app startup performance can be improved by deferring the loading of some application assemblies until they're required. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Updated globalization support

Globalization support is available for Blazor WebAssembly based on International Components for Unicode (ICU). For more information, see ASP.NET Core Blazor globalization and localization.

gRPC

Many preformance improvements have been made in gRPC. For more information, see gRPC performance improvements in .NET 5.

For more gRPC information, see Overview for gRPC on .NET.

SignalR

SignalR Hub filters

SignalR Hub filters, called Hub pipelines in ASP.NET SignalR, is a feature that allows code to run before and after Hub methods are called. Running code before and after Hub methods are called is similar to how middleware has the ability to run code before and after an HTTP request. Common uses include logging, error handling, and argument validation.

For more information, see Use hub filters in ASP.NET Core SignalR.

SignalR parallel hub invocations

ASP.NET Core SignalR is now capable of handling parallel hub invocations. The default behavior can be changed to allow clients to invoke more than one hub method at a time:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        options.MaximumParallelInvocationsPerClient = 5;
    });
}
```

Added Messagepack support in SignalR Java client

A new package, com.microsoft.signalr.messagepack, adds MessagePack support to the SignalR Java client. To use the MessagePack hub protocol, add `.withHubProtocol(new MessagePackHubProtocol())` to the connection builder:

Java

```
HubConnection hubConnection = HubConnectionBuilder.create(
                              "http://localhost:53353/MyHub")
                .withHubProtocol(new MessagePackHubProtocol())
                .build();
```

Kestrel

- Reloadable endpoints via configuration: Kestrel can detect changes to configuration passed to KestrelServerOptions.Configure and unbind from existing

endpoints and bind to new endpoints without requiring an application restart when the `reloadOnChange` parameter is `true`. By default when using ConfigureWebHostDefaults or CreateDefaultBuilder, Kestrel binds to the "Kestrel" configuration subsection with `reloadOnChange` enabled. Apps must pass `reloadOnChange: true` when calling `KestrelServerOptions.Configure` manually to get reloadable endpoints.

- HTTP/2 response headers improvements. For more information, see Performance improvements in the next section.

- Support for additional endpoints types in the sockets transport: Adding to the new API introduced in System.Net.Sockets, the sockets default transport in Kestrel allows binding to both existing file handles and Unix domain sockets. Support for binding to existing file handles enables using the existing `Systemd` integration without requiring the `libuv` transport.

- Custom header decoding in Kestrel: Apps can specify which Encoding to use to interpret incoming headers based on the header name instead of defaulting to UTF-8. Set the `Microsoft.AspNetCore.Server.Kestrel.KestrelServerOptions.RequestHeaderEncodingSelector` property to specify which encoding to use:

```
C#
```

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.RequestHeaderEncodingSelector = encoding =>
              {
                    return encoding switch
                    {
                        "Host" => System.Text.Encoding.Latin1,
                        _ => System.Text.Encoding.UTF8,
                    };
              };
          });
          webBuilder.UseStartup<Startup>();
      });
```

Kestrel endpoint-specific options via configuration

Support has been added for configuring Kestrel's endpoint-specific options via configuration. The endpoint-specific configurations includes the:

- HTTP protocols used
- TLS protocols used
- Certificate selected
- Client certificate mode

Configuration allows specifying which certificate is selected based on the specified server name. The server name is part of the Server Name Indication (SNI) extension to the TLS protocol as indicated by the client. Kestrel's configuration also supports a wildcard prefix in the host name.

The following example shows how to specify endpoint-specific using a configuration file:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "EndpointName": {
        "Url": "https://*",
        "Sni": {
          "a.example.org": {
            "Protocols": "Http1AndHttp2",
            "SslProtocols": [ "Tls11", "Tls12"],
            "Certificate": {
              "Path": "testCert.pfx",
              "Password": "testPassword"
            },
            "ClientCertificateMode" : "NoCertificate"
          },
          "*.example.org": {
            "Certificate": {
              "Path": "testCert2.pfx",
              "Password": "testPassword"
            }
          },
          "*": {
            // At least one sub-property needs to exist per
            // SNI section or it cannot be discovered via
            // IConfiguration
            "Protocols": "Http1",
          }
        }
      }
    }
  }
}
```

Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. What this effectively means is that the virtual domain name, or a hostname, can be used to identify the network end point.

Performance improvements

HTTP/2

- Significant reductions in allocations in the HTTP/2 code path.
- Support for HPack dynamic compression of HTTP/2 response headers in Kestrel. For more information, see Header table size and HPACK: the silent killer (feature) of HTTP/2.
- Sending HTTP/2 PING frames: HTTP/2 has a mechanism for sending PING frames to ensure an idle connection is still functional. Ensuring a viable connection is especially useful when working with long-lived streams that are often idle but only intermittently see activity, for example, gRPC streams. Apps can send periodic PING frames in Kestrel by setting limits on KestrelServerOptions:

```
C#
```

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.Limits.Http2.KeepAlivePingInterval =
                                              TimeSpan.FromSeconds(10);
              options.Limits.Http2.KeepAlivePingTimeout =
                                              TimeSpan.FromSeconds(1);
          });
          webBuilder.UseStartup<Startup>();
      });
```

Containers

Prior to .NET 5.0, building and publishing a *Dockerfile* for an ASP.NET Core app required pulling the entire .NET Core SDK and the ASP.NET Core image. With this release, pulling the SDK images bytes is reduced and the bytes pulled for the ASP.NET Core image is largely eliminated. For more information, see this GitHub issue comment.

Authentication and authorization

Microsoft Entra ID authentication with Microsoft.Identity.Web

The ASP.NET Core project templates now integrate with Microsoft.Identity.Web to handle authentication with Microsoft Entra ID. The Microsoft.Identity.Web package provides:

- A better experience for authentication through Microsoft Entra ID.
- An easier way to access Azure resources on behalf of your users, including Microsoft Graph. See the Microsoft.Identity.Web sample, which starts with a basic login and advances through multi-tenancy, using Azure APIs, using Microsoft Graph, and protecting your own APIs. `Microsoft.Identity.Web` is available alongside .NET 5.

Allow anonymous access to an endpoint

The `AllowAnonymous` extension method allows anonymous access to an endpoint:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        })
        .AllowAnonymous();
    });
}
```

Custom handling of authorization failures

Custom handling of authorization failures is now easier with the new IAuthorizationMiddlewareResultHandler interface that is invoked by the authorization Middleware. The default implementation remains the same, but a custom handler can

be registered in [Dependency injection, which allows custom HTTP responses based on why authorization failed. See this sample that demonstrates usage of the `IAuthorizationMiddlewareResultHandler`.

Authorization when using endpoint routing

Authorization when using endpoint routing now receives the `HttpContext` rather than the endpoint instance. This allows the authorization middleware to access the `RouteData` and other properties of the `HttpContext` that were not accessible though the Endpoint class. The endpoint can be fetched from the context using context.GetEndpoint.

Role-based access control with Kerberos authentication and LDAP on Linux

See Kerberos authentication and role-based access control (RBAC)

API improvements

JSON extension methods for HttpRequest and HttpResponse

JSON data can be read and written to from an `HttpRequest` and `HttpResponse` using the new ReadFromJsonAsync and `WriteAsJsonAsync` extension methods. These extension methods use the System.Text.Json serializer to handle the JSON data. The new `HasJsonContentType` extension method can also check if a request has a JSON content type.

The JSON extension methods can be combined with endpoint routing to create JSON APIs in a style of programming we call ***route to code***. It is a new option for developers who want to create basic JSON APIs in a lightweight way. For example, a web app that has only a handful of endpoints might choose to use route to code rather than the full functionality of ASP.NET Core MVC:

C#

```
endpoints.MapGet("/weather/{city:alpha}", async context =>
{
    var city = (string)context.Request.RouteValues["city"];
    var weather = GetFromDatabase(city);
```

```
    await context.Response.WriteAsJsonAsync(weather);
});
```

System.Diagnostics.Activity

The default format for System.Diagnostics.Activity now defaults to the W3C format. This makes distributed tracing support in ASP.NET Core interoperable with more frameworks by default.

FromBodyAttribute

FromBodyAttribute now supports configuring an option that allows these parameters or properties to be considered optional:

C#

```
public IActionResult Post([FromBody(EmptyBodyBehavior =
EmptyBodyBehavior.Allow)]
                          MyModel model)
{
    ...
}
```

Miscellaneous improvements

We've started applying nullable annotations to ASP.NET Core assemblies. We plan to annotate most of the common public API surface of the .NET 5 framework.

Control Startup class activation

An additional UseStartup overload has been added that lets an app provide a factory method for controlling `Startup` class activation. Controlling `Startup` class activation is useful to pass additional parameters to `Startup` that are initialized along with the host:

C#

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var logger = CreateLogger();
        var host = Host.CreateDefaultBuilder()
            .ConfigureWebHost(builder =>
```

```
            {
                builder.UseStartup(context => new Startup(logger));
            })
            .Build();

        await host.RunAsync();
    }
}
```

Auto refresh with dotnet watch

In .NET 5, running dotnet watch on an ASP.NET Core project both launches the default browser and auto refreshes the browser as changes are made to the code. This means you can:

- Open an ASP.NET Core project in a text editor.
- Run `dotnet watch`.
- Focus on the code changes while the tooling handles rebuilding, restarting, and reloading the app.

Console Logger Formatter

Improvements have been made to the console log provider in the `Microsoft.Extensions.Logging` library. Developers can now implement a custom `ConsoleFormatter` to exercise complete control over formatting and colorization of the console output. The formatter APIs allow for rich formatting by implementing a subset of the VT-100 escape sequences. VT-100 is supported by most modern terminals. The console logger can parse out escape sequences on unsupported terminals allowing developers to author a single formatter for all terminals.

JSON Console Logger

In addition to support for custom formatters, we've also added a built-in JSON formatter that emits structured JSON logs to the console. The following code shows how to switch from the default logger to JSON:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
           Host.CreateDefaultBuilder(args)
  .ConfigureLogging(logging =>
  {
     logging.AddJsonConsole(options =>
     {
```

```
            options.JsonWriterOptions = new JsonWriterOptions()
            { Indented = true };
        });
    })
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
    });
```

Log messages emitted to the console are JSON formatted:

JSON

```
{
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Now listening on: https://localhost:5001",
  "State": {
    "Message": "Now listening on: https://localhost:5001",
    "address": "https://localhost:5001",
    "{OriginalFormat}": "Now listening on: {address}"
  }
}
```

What's new in ASP.NET Core 3.1

Article • 09/25/2023

This article highlights the most significant changes in ASP.NET Core 3.1 with links to relevant documentation.

Partial class support for Razor components

Razor components are now generated as partial classes. Code for a Razor component can be written using a code-behind file defined as a partial class rather than defining all the code for the component in a single file. For more information, see Partial class support.

Component Tag Helper and pass parameters to top-level components

In Blazor with ASP.NET Core 3.0, components were rendered into pages and views using an HTML Helper (`Html.RenderComponentAsync`). In ASP.NET Core 3.1, render a component from a page or view with the new Component Tag Helper:

CSHTML

```
<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

The HTML Helper remains supported in ASP.NET Core 3.1, but the Component Tag Helper is recommended.

Blazor Server apps can now pass parameters to top-level components during the initial render. Previously you could only pass parameters to a top-level component with RenderMode.Static. With this release, both RenderMode.Server and RenderMode.ServerPrerendered are supported. Any specified parameter values are serialized as JSON and included in the initial response.

For example, prerender a `Counter` component with an increment amount (`IncrementAmount`):

CSHTML

```
<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-IncrementAmount="10" />
```

For more information, see Integrate components into Razor Pages and MVC apps.

Support for shared queues in HTTP.sys

HTTP.sys supports creating anonymous request queues. In ASP.NET Core 3.1, we've added the ability to create or attach to an existing named HTTP.sys request queue. Creating or attaching to an existing named HTTP.sys request queue enables scenarios where the HTTP.sys controller process that owns the queue is independent of the listener process. This independence makes it possible to preserve existing connections and enqueued requests between listener process restarts:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
     Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // ...
            webBuilder.UseHttpSys(options =>
            {
                options.RequestQueueName = "MyExistingQueue";
                options.RequestQueueMode = RequestQueueMode.CreateOrAttach;
            });
        });
```

Breaking changes for SameSite cookies

The behavior of SameSite cookies has changed to reflect upcoming browser changes. This may affect authentication scenarios like AzureAd, OpenIdConnect, or WsFederation. For more information, see Work with SameSite cookies in ASP.NET Core.

Prevent default actions for events in Blazor apps

Use the `@on{EVENT}:preventDefault` directive attribute to prevent the default action for an event. In the following example, the default action of displaying the key's character in the text box is prevented:

razor

```
<input value="@_count" @onkeypress="KeyHandler" @onkeypress:preventDefault
/>
```

For more information, see Prevent default actions.

Stop event propagation in Blazor apps

Use the `@on{EVENT}:stopPropagation` directive attribute to stop event propagation. In the following example, selecting the checkbox prevents click events from the child `<div>` from propagating to the parent `<div>`:

razor

```
<input @bind="_stopPropagation" type="checkbox" />

<div @onclick="OnSelectParentDiv">
    <div @onclick="OnSelectChildDiv"
@onclick:stopPropagation="_stopPropagation">
        ...
    </div>
</div>

@code {
    private bool _stopPropagation = false;
}
```

For more information, see Stop event propagation.

Detailed errors during Blazor app development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a gold bar at the bottom of the screen:

- During development, the gold bar directs you to the browser console, where you can see the exception.
- In production, the gold bar notifies the user that an error has occurred and recommends refreshing the browser.

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

What's new in ASP.NET Core 3.0

Article • 09/27/2023

This article highlights the most significant changes in ASP.NET Core 3.0 with links to relevant documentation.

Blazor

Blazor is a new framework in ASP.NET Core for building interactive client-side web UI with .NET:

- Create rich interactive UIs using C#.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.

Blazor framework supported scenarios:

- Reusable UI components (Razor components)
- Client-side routing
- Component layouts
- Support for dependency injection
- Forms and validation
- Supply Razor components in Razor class libraries
- JavaScript interop

For more information, see ASP.NET Core Blazor.

Blazor Server

Blazor decouples component rendering logic from how UI updates are applied. Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection. Blazor Server is supported in ASP.NET Core 3.0.

Blazor WebAssembly (Preview)

Blazor apps can also be run directly in the browser using a WebAssembly-based .NET runtime. Blazor WebAssembly is in preview and *not* supported in ASP.NET Core 3.0. Blazor WebAssembly will be supported in a future release of ASP.NET Core.

Razor components

Blazor apps are built from components. Components are self-contained chunks of user interface (UI), such as a page, dialog, or form. Components are normal .NET classes that define UI rendering logic and client-side event handlers. You can create rich interactive web apps without JavaScript.

Components in Blazor are typically authored using Razor syntax, a natural blend of HTML and C#. Razor components are similar to Razor Pages and MVC views in that they both use Razor. Unlike pages and views, which are based on a request-response model, components are used specifically for handling UI composition.

gRPC

gRPC:

- Is a popular, high-performance RPC (remote procedure call) framework.
- Offers an opinionated contract-first approach to API development.
- Uses modern technologies such as:
  - HTTP/2 for transport.
  - Protocol Buffers as the interface description language.
  - Binary serialization format.
- Provides features such as:
  - Authentication
  - Bidirectional streaming and flow control.
  - Cancellation and timeouts.

gRPC functionality in ASP.NET Core 3.0 includes:

- Grpc.AspNetCore: An ASP.NET Core framework for hosting gRPC services. gRPC on ASP.NET Core integrates with standard ASP.NET Core features like logging, dependency injection (DI), authentication, and authorization.
- Grpc.Net.Client: A gRPC client for .NET Core that builds upon the familiar `HttpClient`.
- Grpc.Net.ClientFactory: gRPC client integration with `HttpClientFactory`.

For more information, see Overview for gRPC on .NET.

SignalR

See Update SignalR code for migration instructions. SignalR now uses `System.Text.Json` to serialize/deserialize JSON messages. See Switch to Newtonsoft.Json for instructions to restore the `Newtonsoft.Json`-based serializer.

In the JavaScript and .NET Clients for SignalR, support was added for automatic reconnection. By default, the client tries to reconnect immediately and retry after 2, 10, and 30 seconds if necessary. If the client successfully reconnects, it receives a new connection ID. Automatic reconnect is opt-in:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

The reconnection intervals can be specified by passing an array of millisecond-based durations:

JavaScript

```
.withAutomaticReconnect([0, 3000, 5000, 10000, 15000, 30000])
//.withAutomaticReconnect([0, 2000, 10000, 30000]) The default intervals.
```

A custom implementation can be passed in for full control of the reconnection intervals.

If the reconnection fails after the last reconnect interval:

- The client considers the connection is offline.
- The client stops trying to reconnect.

During reconnection attempts, update the app UI to notify the user that the reconnection is being attempted.

To provide UI feedback when the connection is interrupted, the SignalR client API has been expanded to include the following event handlers:

- `onreconnecting`: Gives developers an opportunity to disable UI or to let users know the app is offline.
- `onreconnected`: Gives developers an opportunity to update the UI once the connection is reestablished.

The following code uses `onreconnecting` to update the UI while trying to connect:

JavaScript

```
connection.onreconnecting((error) => {
    const status = `Connection lost due to error "${error}". Reconnecting.`;
    document.getElementById("messageInput").disabled = true;
    document.getElementById("sendButton").disabled = true;
    document.getElementById("connectionStatus").innerText = status;
});
```

The following code uses `onreconnected` to update the UI on connection:

JavaScript

```
connection.onreconnected((connectionId) => {
    const status = `Connection reestablished. Connected.`;
    document.getElementById("messageInput").disabled = false;
    document.getElementById("sendButton").disabled = false;
    document.getElementById("connectionStatus").innerText = status;
});
```

SignalR 3.0 and later provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of `HubInvocationContext`. The `HubInvocationContext` includes the:

- `HubCallerContext`
- Name of the hub method being invoked.
- Arguments to the hub method.

Consider the following example of a chat room app allowing multiple organization sign-in via Azure Active Directory. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization can ban users or view users' chat histories. The app could restrict certain functionality from specific users.

C#

```
public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
    IAuthorizationRequirement
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User?.Identity?.Name == null)
        {
            return Task.CompletedTask;
        }
```

```
        if (IsUserAllowedToDoThis(resource.HubMethodName,
context.User.Identity.Name))
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName, string
currentUsername)
    {
        if (hubMethodName.Equals("banUser",
StringComparison.OrdinalIgnoreCase))
        {
            return currentUsername.Equals("bob42@jabbr.net",
StringComparison.OrdinalIgnoreCase);
        }

        return currentUsername.EndsWith("@jabbr.net",
StringComparison.OrdinalIgnoreCase));
    }
}
```

In the preceding code, `DomainRestrictedRequirement` serves as a custom `IAuthorizationRequirement`. Because the `HubInvocationContext` resource parameter is being passed in, the internal logic can:

- Inspect the context in which the Hub is being called.
- Make decisions on allowing the user to execute individual Hub methods.

Individual Hub methods can be marked with the name of the policy the code checks at run-time. As clients attempt to call individual Hub methods, the `DomainRestrictedRequirement` handler runs and controls access to the methods. Based on the way the `DomainRestrictedRequirement` controls access:

- All logged-in users can call the `SendMessage` method.
- Only users who have logged in with a `@jabbr.net` email address can view users' histories.
- Only `bob42@jabbr.net` can ban users from the chat room.

C#

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }
```

```
    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}
```

Creating the `DomainRestricted` policy might involve:

- In `Startup.cs`, adding the new policy.
- Provide the custom `DomainRestrictedRequirement` requirement as a parameter.
- Registering `DomainRestricted` with the authorization middleware.

C#

```
services
    .AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new DomainRestrictedRequirement());
        });
    });
```

SignalR hubs use Endpoint Routing. SignalR hub connection was previously done explicitly:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});
```

In the previous version, developers needed to wire up controllers, Razor pages, and hubs in a variety of places. Explicit connection results in a series of nearly-identical routing segments:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
```

```
});

app.UseRouting(routes =>
{
    routes.MapRazorPages();
});
```

SignalR 3.0 hubs can be routed via endpoint routing. With endpoint routing, typically all routing can be configured in `UseRouting`:

C#

```
app.UseRouting(routes =>
{
    routes.MapRazorPages();
    routes.MapHub<ChatHub>("hubs/chat");
});
```

ASP.NET Core 3.0 SignalR added:

Client-to-server streaming. With client-to-server streaming, server-side methods can take instances of either an `IAsyncEnumerable<T>` or `ChannelReader<T>`. In the following C# sample, the `UploadStream` method on the Hub will receive a stream of strings from the client:

C#

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        // process content
    }
}
```

.NET client apps can pass either an `IAsyncEnumerable<T>` or `ChannelReader<T>` instance as the `stream` argument of the `UploadStream` Hub method above.

After the `for` loop has completed and the local function exits, the stream completion is sent:

C#

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
```

```
        var data = await FetchSomeData();
        yield return data;
    }
}

await connection.SendAsync("UploadStream", clientStreamData());
```

JavaScript client apps use the SignalR `Subject` (or an RxJS Subject) for the `stream` argument of the `UploadStream` Hub method above.

JavaScript

```
let subject = new signalR.Subject();
await connection.send("StartStream", "MyAsciiArtStream", subject);
```

The JavaScript code could use the `subject.next` method to handle strings as they are captured and ready to be sent to the server.

JavaScript

```
subject.next("example");
subject.complete();
```

Using code like the two preceding snippets, real-time streaming experiences can be created.

New JSON serialization

ASP.NET Core 3.0 now uses System.Text.Json by default for JSON serialization:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support.

New Razor directives

The following list contains new Razor directives:

- @attribute: The `@attribute` directive applies the given attribute to the class of the generated page or view. For example, `@attribute [Authorize]`.

- @implements: The `@implements` directive implements an interface for the generated class. For example, `@implements IDisposable`.

IdentityServer4 supports authentication and authorization for web APIs and SPAs

ASP.NET Core 3.0 offers authentication in Single Page Apps (SPAs) using the support for web API authorization. ASP.NET Core Identity for authenticating and storing users is combined with IdentityServer4 for implementing OpenID Connect.

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 3.0. It enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see the IdentityServer4 documentation or Authentication and authorization for SPAs.

Certificate and Kerberos authentication

Certificate authentication requires:

- Configuring the server to accept certificates.
- Adding the authentication middleware in `Startup.Configure`.
- Adding the certificate authentication service in `Startup.ConfigureServices`.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
            .AddCertificate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Options for certificate authentication include the ability to:

- Accept self-signed certificates.
- Check for certificate revocation.
- Check that the proffered certificate has the right usage flags in it.

A default user principal is constructed from the certificate properties. The user principal contains an event that enables supplementing or replacing the principal. For more information, see Configure certificate authentication in ASP.NET Core.

Windows Authentication has been extended onto Linux and macOS. In previous versions, Windows Authentication was limited to IIS and HTTP.sys. In ASP.NET Core 3.0, Kestrel has the ability to use Negotiate, Kerberos, and NTLM on Windows, Linux, and macOS for Windows domain-joined hosts. Kestrel support of these authentication schemes is provided by the Microsoft.AspNetCore.Authentication.Negotiate NuGet package. As with the other authentication services, configure authentication app wide, then configure the service:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
        .AddNegotiate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Host requirements:

- Windows hosts must have Service Principal Names (SPNs) added to the user account hosting the app.
- Linux and macOS machines must be joined to the domain.
  - SPNs must be created for the web process.
  - Keytab files must be generated and configured on the host machine.

For more information, see Configure Windows Authentication in ASP.NET Core.

Template changes

The web UI templates (Razor Pages, MVC with controller and views) have the following removed:

- The cookie consent UI is no longer included. To enable the cookie consent feature in an ASP.NET Core 3.0 template-generated app, see General Data Protection Regulation (GDPR) support in ASP.NET Core.
- Scripts and related static assets are now referenced as local files instead of using CDNs. For more information, see Scripts and related static assets are now referenced as local files instead of using CDNs based on the current environment (dotnet/AspNetCore.Docs #14350).

The Angular template updated to use Angular 8.

The Razor class library (RCL) template defaults to Razor component development by default. A new template option in Visual Studio provides template support for pages and views. When creating an RCL from the template in a command shell, pass the `--support-pages-and-views` option (`dotnet new razorclasslib --support-pages-and-views`).

Generic Host

The ASP.NET Core 3.0 templates use .NET Generic Host in ASP.NET Core. Previous versions used WebHostBuilder. Using the .NET Core Generic Host (HostBuilder) provides better integration of ASP.NET Core apps with other server scenarios that aren't web-specific. For more information, see HostBuilder replaces WebHostBuilder.

Host configuration

Prior to the release of ASP.NET Core 3.0, environment variables prefixed with `ASPNETCORE_` were loaded for host configuration of the Web Host. In 3.0, `AddEnvironmentVariables` is used to load environment variables prefixed with `DOTNET_` for host configuration with `CreateDefaultBuilder`.

Changes to Startup constructor injection

The Generic Host only supports the following types for `Startup` constructor injection:

- IHostEnvironment
- `IWebHostEnvironment`
- IConfiguration

All services can still be injected directly as arguments to the `Startup.Configure` method. For more information, see Generic Host restricts Startup constructor injection (aspnet/Announcements #353).

Kestrel

- Kestrel configuration has been updated for the migration to the Generic Host. In 3.0, Kestrel is configured on the web host builder provided by `ConfigureWebHostDefaults`.
- Connection Adapters have been removed from Kestrel and replaced with Connection Middleware, which is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections.
- The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`.
- Ambiguity between headers and trailers has been resolved by moving trailing headers to a new collection.
- Synchronous I/O APIs, such as `HttpRequest.Body.Read`, are a common source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default.

For more information, see Migrate from ASP.NET Core 2.2 to 3.0.

HTTP/2 enabled by default

HTTP/2 is enabled by default in Kestrel for HTTPS endpoints. HTTP/2 support for IIS or HTTP.sys is enabled when supported by the operating system.

EventCounters on request

The Hosting EventSource, `Microsoft.AspNetCore.Hosting`, emits the following new EventCounter types related to incoming requests:

- `requests-per-second`
- `total-requests`
- `current-requests`
- `failed-requests`

Endpoint routing

Endpoint Routing, which allows frameworks (for example, MVC) to work well with middleware, is enhanced:

- The order of middleware and endpoints is configurable in the request processing pipeline of `Startup.Configure`.
- Endpoints and middleware compose well with other ASP.NET Core-based technologies, such as Health Checks.
- Endpoints can implement a policy, such as CORS or authorization, in both middleware and MVC.
- Filters and attributes can be placed on methods in controllers.

For more information, see Routing in ASP.NET Core.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
C#

app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see the following articles:

- Migrate from ASP.NET Core 2.2 to 3.0
- Health checks in ASP.NET Core

Pipes on HttpContext

It's now possible to read the request body and write the response body using the System.IO.Pipelines API. The `HttpRequest.BodyReader` property provides a PipeReader that can be used to read the request body. The `HttpResponse.BodyWriter` property provides a PipeWriter that can be used to write the response body.

`HttpRequest.BodyReader` is an analogue of the `HttpRequest.Body` stream.
`HttpResponse.BodyWriter` is an analogue of the `HttpResponse.Body` stream.

Improved error reporting in IIS

Startup errors when hosting ASP.NET Core apps in IIS now produce richer diagnostic data. These errors are reported to the Windows Event Log with stack traces wherever applicable. In addition, all warnings, errors, and unhandled exceptions are logged to the Windows Event Log.

Worker Service and Worker SDK

.NET Core 3.0 introduces the new Worker Service app template. This template provides a starting point for writing long running services in .NET Core.

For more information, see:

- .NET Core Workers as Windows Services
- Background tasks with hosted services in ASP.NET Core
- Host ASP.NET Core in a Windows Service

Forwarded Headers Middleware improvements

In previous versions of ASP.NET Core, calling UseHsts and UseHttpsRedirection were problematic when deployed to an Azure Linux or behind any reverse proxy other than IIS. The fix for previous versions is documented in Forward the scheme for Linux and non-IIS reverse proxies.

This scenario is fixed in ASP.NET Core 3.0. The host enables the Forwarded Headers Middleware when the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`. `ASPNETCORE_FORWARDEDHEADERS_ENABLED` is set to `true` in our container images.

Performance improvements

ASP.NET Core 3.0 includes many improvements that reduce memory usage and improve throughput:

- Reduction in memory usage when using the built-in dependency injection container for scoped services.

- Reduction in allocations across the framework, including middleware scenarios and routing.
- Reduction in memory usage for WebSocket connections.
- Memory reduction and throughput improvements for HTTPS connections.
- New optimized and fully asynchronous JSON serializer.
- Reduction in memory usage and throughput improvements in form parsing.

ASP.NET Core 3.0 only runs on .NET Core 3.0

As of ASP.NET Core 3.0, .NET Framework is no longer a supported target framework. Projects targeting .NET Framework can continue in a fully supported fashion using the .NET Core 2.1 LTS release. Most ASP.NET Core 2.1.x related packages will be supported indefinitely, beyond the three-year LTS period for .NET Core 2.1.

For migration information, see Port your code from .NET Framework to .NET Core.

Use the ASP.NET Core shared framework

The ASP.NET Core 3.0 shared framework, contained in the Microsoft.AspNetCore.App metapackage, no longer requires an explicit `<PackageReference />` element in the project file. The shared framework is automatically referenced when using the `Microsoft.NET.Sdk.Web` SDK in the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

Assemblies removed from the ASP.NET Core shared framework

The most notable assemblies removed from the ASP.NET Core 3.0 shared framework are:

- Newtonsoft.Json (Json.NET). To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support. ASP.NET Core 3.0 introduces `System.Text.Json` for reading and writing JSON. For more information, see New JSON serialization in this document.
- Entity Framework Core

For a complete list of assemblies removed from the shared framework, see Assemblies being removed from Microsoft.AspNetCore.App 3.0. For more information on the motivation for this change, see Breaking changes to Microsoft.AspNetCore.App in 3.0 and A first look at changes coming in ASP.NET Core 3.0.

What's new in ASP.NET Core 2.2

Article • 08/14/2024

This article highlights the most significant changes in ASP.NET Core 2.2, with links to relevant documentation.

OpenAPI Analyzers & Conventions

OpenAPI (formerly known as Swagger) is a language-agnostic specification for describing REST APIs. The OpenAPI ecosystem has tools that allow for discovering, testing, and producing client code using the specification. Support for generating and visualizing OpenAPI documents in ASP.NET Core MVC is provided via community driven projects such as NSwag and Swashbuckle.AspNetCore. ASP.NET Core 2.2 provides improved tooling and runtime experiences for creating OpenAPI documents.

For more information, see the following resources:

- Use web API analyzers
- Use web API conventions
- ASP.NET Core 2.2.0-preview1: OpenAPI Analyzers & Conventions

Problem details support

ASP.NET Core 2.1 introduced `ProblemDetails`, based on the RFC 7807 specification for carrying details of an error with an HTTP Response. In 2.2, `ProblemDetails` is the standard response for client error codes in controllers attributed with `ApiControllerAttribute`. An `IActionResult` returning a client error status code (4xx) now returns a `ProblemDetails` body. The result also includes a correlation ID that can be used to correlate the error using request logs. For client errors, `ProducesResponseType` defaults to using `ProblemDetails` as the response type. This is documented in OpenAPI / Swagger output generated using NSwag or Swashbuckle.AspNetCore.

Endpoint Routing

ASP.NET Core 2.2 uses a new *endpoint routing* system for improved dispatching of requests. The changes include new link generation API members and route parameter transformers.

For more information, see the following resources:

- Endpoint routing in 2.2
- Route parameter transformers (see **Routing** section)
- Differences between IRouter- and endpoint-based routing

Health checks

A new health checks service makes it easier to use ASP.NET Core in environments that require health checks, such as Kubernetes. Health checks includes middleware and a set of libraries that define an `IHealthCheck` abstraction and service.

Health checks are used by a container orchestrator or load balancer to quickly determine if a system is responding to requests normally. A container orchestrator might respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might respond to a health check by routing traffic away from the failing instance of the service.

Health checks are exposed by an application as an HTTP endpoint used by monitoring systems. Health checks can be configured for a variety of real-time monitoring scenarios and monitoring systems. The health checks service integrates with the BeatPulse project . which makes it easier to add checks for dozens of popular systems and dependencies.

For more information, see Health checks in ASP.NET Core.

HTTP/2 in Kestrel

ASP.NET Core 2.2 adds support for HTTP/2.

HTTP/2 is a major revision of the HTTP protocol. Notable features of HTTP/2 include:

- Support for header compression.
- Fully multiplexed streams over a single connection.

While HTTP/2 preserves HTTP's semantics (for example, HTTP headers and methods), it's a breaking change from HTTP/1.x on how data is framed and sent between the client and server.

As a consequence of this change in framing, servers and clients need to negotiate the protocol version used. Application-Layer Protocol Negotiation (ALPN) is a TLS extension that allows the server and client to negotiate the protocol version used as part of their TLS handshake. While it is possible to have prior knowledge between the server and the

client on the protocol, all major browsers support ALPN as the only way to establish an HTTP/2 connection.

For more information, see HTTP/2 support.

Kestrel configuration

In earlier versions of ASP.NET Core, Kestrel options are configured by calling `UseKestrel`. In 2.2, Kestrel options are configured by calling `ConfigureKestrel` on the host builder. This change resolves an issue with the order of `IServer` registrations for in-process hosting. For more information, see the following resources:

- Mitigate UseIIS conflict
- Configure Kestrel server options with ConfigureKestrel

IIS in-process hosting

In earlier versions of ASP.NET Core, IIS serves as a reverse proxy. In 2.2, the ASP.NET Core Module can boot the CoreCLR and host an app inside the IIS worker process (*w3wp.exe*). In-process hosting provides performance and diagnostic gains when running with IIS.

For more information, see in-process hosting for IIS.

SignalR Java client

ASP.NET Core 2.2 introduces a Java Client for SignalR. This client supports connecting to an ASP.NET Core SignalR Server from Java code, including Android apps.

For more information, see ASP.NET Core SignalR Java client.

CORS improvements

In earlier versions of ASP.NET Core, CORS Middleware allows `Accept`, `Accept-Language`, `Content-Language`, and `Origin` headers to be sent regardless of the values configured in `CorsPolicy.Headers`. In 2.2, a CORS Middleware policy match is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For more information, see CORS Middleware.