What's new in ASP.NET Core 8.0

Article • 11/21/2023

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#

```csharp
[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.

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

```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

```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

```javascript
var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chatHub")
  .withServerTimeoutInMilliseconds(60000)
  .withKeepAliveIntervalInMilliseconds(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

```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.StatefulRecon-
nectBufferSize = 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, de-
faults 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.State-
fulReconnectBufferSize = 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 anti-forgery token. The anti-forgery 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#

```csharp
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.

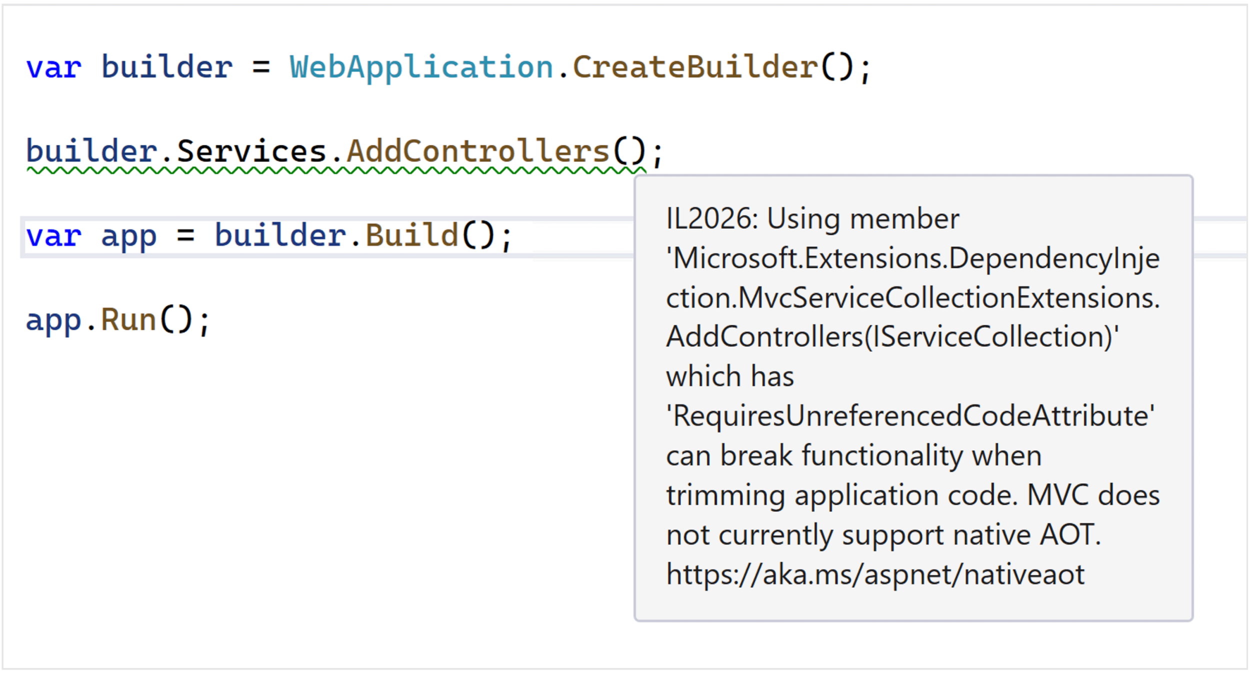
One of the new features is support for JSON serialization of IAsyncEnumerable<T> implementations implemented by the C# compiler. This support opens up their use in ASP.NET Core projects configured to publish Native AOT.

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#

```csharp
// 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 se-
rializer 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<IHttpSysRe-
questTimingFeature>();

    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 autho-
rization requirement
            // succeeded.
            if (age >= requirement.Age)
            {
                _logger.LogInformation("Minimum age authorization re-
quirement {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

```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

```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#

```csharp
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)
=>
                                                                small-
Cache.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.

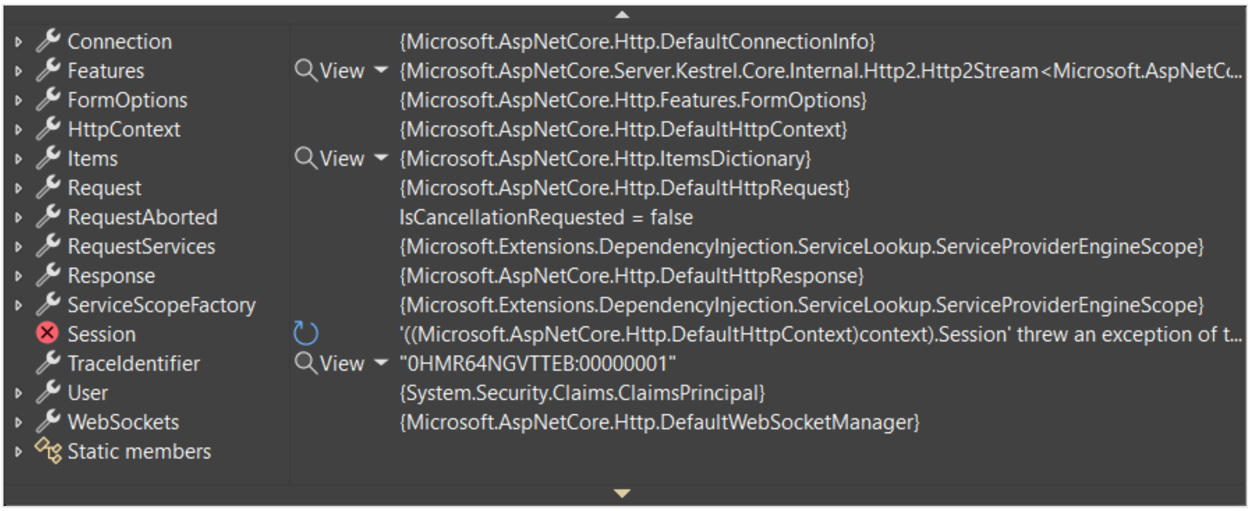
`IExceptionHandler` implementations are registered by calling IServiceCollection.AddExceptionHandler<T>. Multiple implementations can be added, and they're called in the order registered. If an exception handler handles a request, it can return `true` to stop processing. If an exception isn't handled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

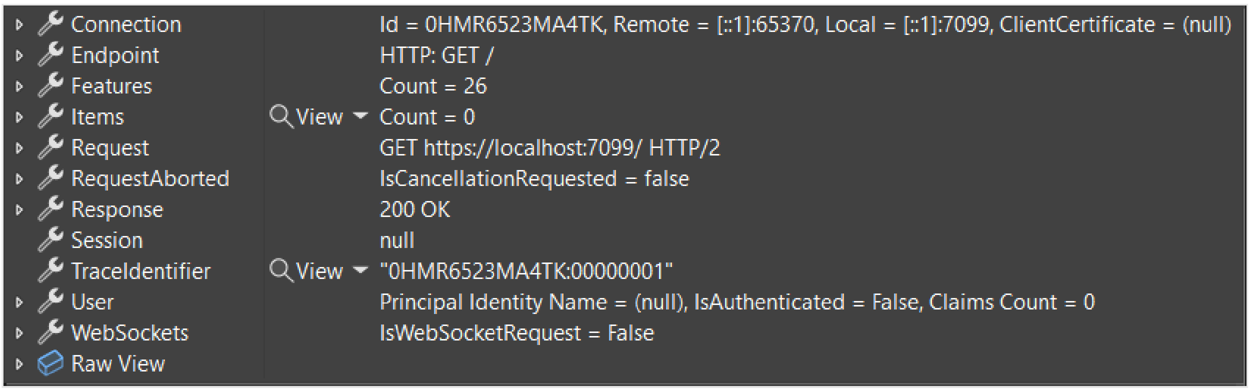
Debug customization attributes have been added to types like `HttpContext`, `HttpRequest`, `HttpResponse`, `ClaimsPrincipal`, and `WebApplication`. The enhanced debugger displays for these types make finding important information easier in an IDE's debugger. The following screenshots show the difference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

.NET 7:



```
Connection                           {Microsoft.AspNetCore.Http.DefaultConnectionInfo}
Features               View          {Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http2.Http2Stream<Microsoft.AspNetC...
FormOptions                          {Microsoft.AspNetCore.Http.Features.FormOptions}
HttpContext                          {Microsoft.AspNetCore.Http.DefaultHttpContext}
Items                  View          {Microsoft.AspNetCore.Http.ItemsDictionary}
Request                              {Microsoft.AspNetCore.Http.DefaultHttpRequest}
RequestAborted                       IsCancellationRequested = false
RequestServices                      {Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceProviderEngineScope}
Response                             {Microsoft.AspNetCore.Http.DefaultHttpResponse}
ServiceScopeFactory                  {Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceProviderEngineScope}
Session                              '((Microsoft.AspNetCore.Http.DefaultHttpContext)context).Session' threw an exception of t...
TraceIdentifier        View          "0HMR64NGVTTEB:00000001"
User                                 {System.Security.Claims.ClaimsPrincipal}
WebSockets                           {Microsoft.AspNetCore.Http.DefaultWebSocketManager}
Static members
```

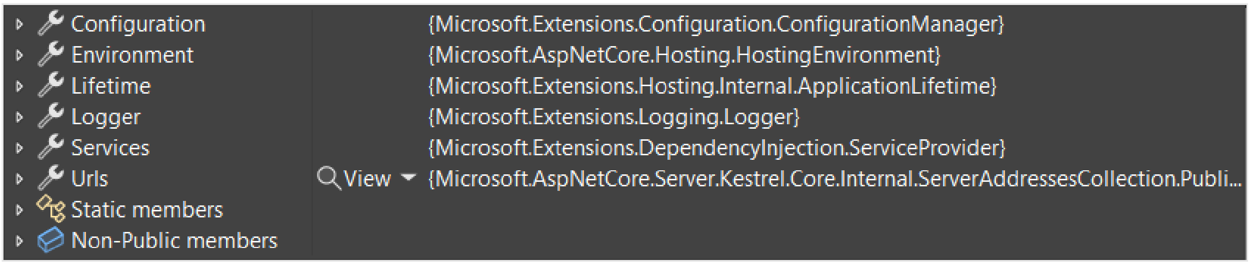
.NET 8:



```
Connection                      Id = 0HMR6523MA4TK, Remote = [::1]:65370, Local = [::1]:7099, ClientCertificate = (null)
Endpoint                        HTTP: GET /
Features                        Count = 26
Items              View         Count = 0
Request                         GET https://localhost:7099/ HTTP/2
RequestAborted                  IsCancellationRequested = false
Response                        200 OK
Session                         null
TraceIdentifier    View         "0HMR6523MA4TK:00000001"
User                            Principal Identity Name = (null), IsAuthenticated = False, Claims Count = 0
WebSockets                      IsWebSocketRequest = False
Raw View
```

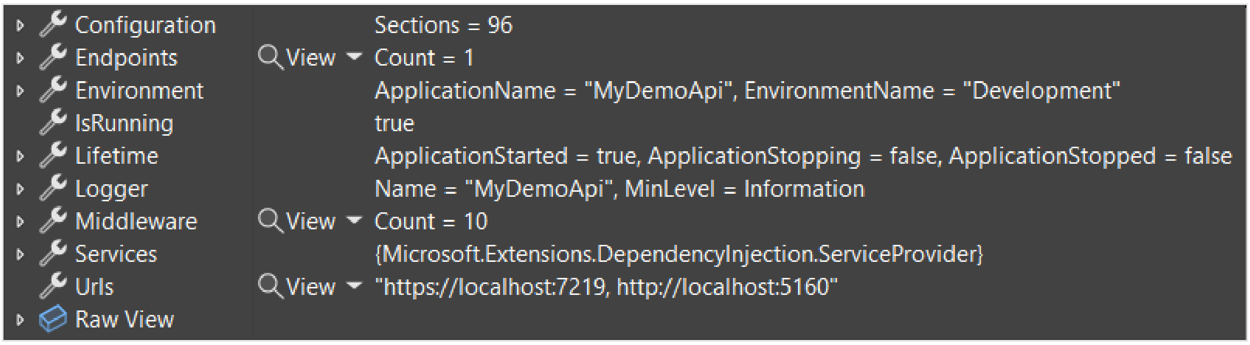
The debugger display for `WebApplication` highlights important information such as configured endpoints, middleware, and `IConfiguration` values.

.NET 7:



```
Configuration          {Microsoft.Extensions.Configuration.ConfigurationManager}
Environment            {Microsoft.AspNetCore.Hosting.HostingEnvironment}
Lifetime               {Microsoft.Extensions.Hosting.Internal.ApplicationLifetime}
Logger                 {Microsoft.Extensions.Logging.Logger}
Services               {Microsoft.Extensions.DependencyInjection.ServiceProvider}
Urls            View   {Microsoft.AspNetCore.Server.Kestrel.Core.Internal.ServerAddressesCollection.Publi...
Static members
Non-Public members
```

.NET 8:



```
Configuration          Sections = 96
Endpoints       View   Count = 1
Environment            ApplicationName = "MyDemoApi", EnvironmentName = "Development"
IsRunning              true
Lifetime               ApplicationStarted = true, ApplicationStopping = false, ApplicationStopped = false
Logger                 Name = "MyDemoApi", MinLevel = Information
Middleware      View   Count = 10
Services               {Microsoft.Extensions.DependencyInjection.ServiceProvider}
Urls            View   "https://localhost:7219, http://localhost:5160"
Raw View
```

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#

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

C#

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

C#

```csharp
// 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<IProblemDe-
tailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = http-
Context }))
        {
            // 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)

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

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