Server-side circuit handler to capture users for custom services

Use a CircuitHandler to capture a user from the AuthenticationStateProvider and set that user in a service. For more information and example code, see Server-side ASP.NET Core Blazor additional security scenarios.

Closure of 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 created 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.

`IHttpContextAccessor` / `HttpContext` in Razor components

IHttpContextAccessor must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

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

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see Server-side ASP.NET Core Blazor additional security scenarios.

Additional server-side resources

- Server-side host and deployment guidance: SignalR configuration
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR configuration
- Server-side security documentation
  - ASP.NET Core Blazor authentication and authorization
  - Secure ASP.NET Core server-side Blazor apps
  - Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering
  - Server-side ASP.NET Core Blazor additional security scenarios
- Server-side reconnection events and component lifecycle events
- What is Azure SignalR Service?
- Performance guide for Azure SignalR Service
- Publish an ASP.NET Core SignalR app to Azure App Service
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor static files

Article • 11/05/2024

> **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 article describes Blazor app configuration for serving static files.

Static asset delivery in server-side Blazor apps

Serving static assets is managed by either routing endpoint conventions or a middleware described in the following table.

Expand table

Feature	API	.NET Version	Description
Map Static Assets routing endpoint conventions	MapStaticAssets	.NET 9 or later	Optimizes the delivery of static assets to clients.
Static Files Middleware	UseStaticFiles	All .NET versions	Serves static assets to clients without the optimizations of Map Static Assets but useful for some tasks that Map Static Assets isn't capable of managing.

Configure Map Static Assets by calling MapStaticAssets in the app's request processing pipeline, which performs the following:

- Sets the ETag and Last-Modified headers.
- Sets caching headers.
- Uses Caching Middleware.
- When possible, serves compressed static assets.
- Works with a Content Delivery Network (CDN) (for example, Azure CDN) to serve the app's static assets closer to the user.
- Fingerprinting assets to prevent reusing old versions of files.

Map Static Assets operates by combining build and publish processes to collect information about the static assets in the app. This information is utilized by the runtime library to efficiently serve the static assets to browsers.

Map Static Assets can replace UseStaticFiles in most situations. However, Map Static Assets is optimized for serving the assets from known locations in the app at build and publish time. If the app serves assets from other locations, such as disk or embedded resources, UseStaticFiles should be used.

Map Static Assets (MapStaticAssets) replaces calling UseBlazorFrameworkFiles in apps that serve Blazor WebAssembly framework files, and explicitly calling UseBlazorFrameworkFiles in a Blazor Web App isn't necessary because the API is automatically called when invoking AddInteractiveWebAssemblyComponents.

Map Static Assets provides the following benefits that aren't available when calling UseStaticFiles:

- Build-time compression for all the assets in the app, including JavaScript (JS) and stylesheets but excluding image and font assets that are already compressed. Gzip (`Content-Encoding: gz`) compression is used during development. Gzip with Brotli (`Content-Encoding: br`) compression is used during publish.
- Fingerprinting for all assets at build time with a Base64-encoded string of the SHA-256 hash of each file's content. This prevents reusing an old version of a file, even if the old file is cached. Fingerprinted assets are cached using the immutable directive, which results in the browser never requesting the asset again until it changes. For browsers that don't support the `immutable` directive, a max-age directive is added.
  - Even if an asset isn't fingerprinted, content based `ETags` are generated for each static asset using the fingerprint hash of the file as the `ETag` value. This ensures that the browser only downloads a file if its content changes (or the file is being downloaded for the first time).
  - Internally, Blazor maps physical assets to their fingerprints, which allows the app to:
    - Find automatically-generated Blazor assets, such as Razor component scoped CSS for Blazor's CSS isolation feature, and JS assets described by JS import maps.
    - Generate link tags in the `<head>` content of the page to preload assets.
- During Visual Studio Hot Reload development testing:
  - Integrity information is removed from the assets to avoid issues when a file is changed while the app is running.
  - Static assets aren't cached to ensure that the browser always retrieves current content.

When Interactive WebAssembly or Interactive Auto render modes are enabled:

- Blazor creates an endpoint to expose the resource collection as a JS module.
- The URL is emitted to the body of the request as persisted component state when a WebAssembly component is rendered into the page.
- During WebAssembly boot, Blazor retrieves the URL, imports the module, and calls a function to retrieve the asset collection and reconstruct it in memory. The URL is specific to the content and cached forever, so this overhead cost is only paid once per user until the app is updated.
- The resource collection is also exposed at a human-readable URL (`_framework/resource-collection.js`), so JS has access to the resource collection for enhanced navigation or to implement features of other frameworks and third-party components.

Map Static Assets doesn't provide features for minification or other file transformations. Minification is usually handled by custom code or third-party tooling.

Static File Middleware (UseStaticFiles) is useful in the following situations that Map Static Assets (MapStaticAssets) can't handle:

- Applying a path prefix to Blazor WebAssembly static asset files, which is covered in the Prefix for Blazor WebAssembly assets section.
- Configuring file mappings of extensions to specific content types and setting static file options, which is covered in the File mappings and static file options section.

For more information, see Static files in ASP.NET Core.

Deliver assets with Map Static Assets routing endpoint conventions

This section applies to server-side Blazor apps.

Assets are delivered via the ComponentBase.Assets property, which resolves the fingerprinted URL for a given asset. In the following example, Bootstrap, the Blazor project template app stylesheet (`app.css`), and the CSS isolation stylesheet (based on an app's namespace of `BlazorSample`) are linked in a root component, typically the `App` component (`Components/App.razor`):

razor

```
<link rel="stylesheet" href="@Assets["bootstrap/bootstrap.min.css"]" />
<link rel="stylesheet" href="@Assets["app.css"]" />
```

```
<link rel="stylesheet" href="@Assets["BlazorSample.styles.css"]" />
```

Import maps

This section applies to server-side Blazor apps.

The Import Map component (ImportMap) represents an import map element (`<script type="importmap"></script>`) that defines the import map for module scripts. The Import Map component is placed in `<head>` content of the root component, typically the `App` component (`Components/App.razor`).

razor

```
<ImportMap />
```

If a custom ImportMapDefinition isn't assigned to an Import Map component, the import map is generated based on the app's assets.

The following examples demonstrate custom import map definitions and the import maps that they create.

Basic import map:

C#

```
new ImportMapDefinition(
    new Dictionary<string, string>
    {
        { "jquery", "https://cdn.example.com/jquery.js" },
    },
    null,
    null);
```

The preceding code results in the following import map:

JSON

```
{
  "imports": {
    "jquery": "https://cdn.example.com/jquery.js"
  }
}
```

Scoped import map:

C#

```
new ImportMapDefinition(
    null,
    new Dictionary<string, IReadOnlyDictionary<string, string>>
    {
        ["/scoped/"] = new Dictionary<string, string>
        {
            { "jquery", "https://cdn.example.com/jquery.js" },
        }
    },
    null);
```

The preceding code results in the following import map:

JSON

```
{
  "scopes": {
    "/scoped/": {
      "jquery": "https://cdn.example.com/jquery.js"
    }
  }
}
```

Import map with integrity:

C#

```
new ImportMapDefinition(
    new Dictionary<string, string>
    {
        { "jquery", "https://cdn.example.com/jquery.js" },
    },
    null,
    new Dictionary<string, string>
    {
        { "https://cdn.example.com/jquery.js", "sha384-abc123" },
    });
```

The preceding code results in the following import map:

JSON

```
{
  "imports": {
    "jquery": "https://cdn.example.com/jquery.js"
  },
  "integrity": {
    "https://cdn.example.com/jquery.js": "sha384-abc123"
```

```
  }
}
```

Combine import map definitions (ImportMapDefinition) with ImportMapDefinition.Combine.

Import map created from a ResourceAssetCollection that maps static assets to their corresponding unique URLs:

C#

```
ImportMapDefinition.FromResourceCollection(
    new ResourceAssetCollection(
    [
        new ResourceAsset(
            "jquery.fingerprint.js",
            [
                new ResourceAssetProperty("integrity", "sha384-abc123"),
                new ResourceAssetProperty("label", "jquery.js"),
            ])
    ]));
```

The preceding code results in the following import map:

JSON

```
{
  "imports": {
    "./jquery.js": "./jquery.fingerprint.js"
  },
  "integrity": {
    "jquery.fingerprint.js": "sha384-abc123"
  }
}
```

In releases prior to .NET 8, Blazor framework static files, such as the Blazor script, are served via Static File Middleware. In .NET 8 or later, Blazor framework static files are mapped using endpoint routing, and Static File Middleware is no longer used.

Summary of static file `<link>` `href` formats

This section applies to all .NET releases and Blazor apps.

The following tables summarize static file `<link>` `href` formats by .NET release.

For the location of `<head>` content where static file links are placed, see ASP.NET Core Blazor project structure. Static asset links can also be supplied using <HeadContent> components in individual Razor components.

.NET 9 or later

Expand table

App type	`href` value	Examples
Blazor Web App	`@Assets["{PATH}"]`	`<link rel="stylesheet" href="@Assets["app.css"]" />` `<link href="@Assets["_content/ComponentLib/styles.css"]" rel="stylesheet" />`
Blazor Server†	`@Assets["{PATH}"]`	`<link href="@Assets["css/site.css"]" rel="stylesheet" />` `<link href="@Assets["_content/ComponentLib/styles.css"]" rel="stylesheet" />`
Standalone Blazor WebAssembly	`{PATH}`	`<link rel="stylesheet" href="css/app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`

.NET 8.x

Expand table

App type	`href` value	Examples
Blazor Web App	`{PATH}`	`<link rel="stylesheet" href="app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Blazor Server†	`{PATH}`	`<link href="css/site.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Standalone Blazor WebAssembly	`{PATH}`	`<link rel="stylesheet" href="css/app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`

.NET 7.x or earlier

Expand table

App type	`href` value	Examples
Blazor Server†	`{PATH}`	`<link href="css/site.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Hosted Blazor WebAssembly‡	`{PATH}`	`<link href="css/app.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Blazor WebAssembly	`{PATH}`	`<link href="css/app.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`

†Blazor Server is supported in .NET 8 or later but is no longer a project template after .NET 7.
‡We recommend updating Hosted Blazor WebAssembly apps to Blazor Web Apps when adopting .NET 8 or later.

Static Web Asset Project Mode

This section applies to the `.Client` project of a Blazor Web App.

The required `<StaticWebAssetProjectMode>Default</StaticWebAssetProjectMode>` setting in the `.Client` project of a Blazor Web App reverts Blazor WebAssembly static asset behaviors back to the defaults, so that the project behaves as part of the hosted project. The Blazor WebAssembly SDK (`Microsoft.NET.Sdk.BlazorWebAssembly`) configures static web assets in a specific way to work in "standalone" mode with a server simply consuming the outputs from the library. This isn't appropriate for a Blazor Web App, where the WebAssembly portion of the app is a logical part of the host and must behave more like a library. For example, the project doesn't expose the styles bundle (for example, `BlazorSample.Client.styles.css`) and instead only provides the host with the project bundle, so that the host can include it in its own styles bundle.

Changing the value (`Default`) of `<StaticWebAssetProjectMode>` or removing the property from the `.Client` project isn't supported.

Static files in non-`Development` environments

This section applies to server-side static files.

When running an app locally, static web assets are only enabled in the Development environment. To enable static files for environments other than Development during local development and testing (for example, Staging), call UseStaticWebAssets on the WebApplicationBuilder in the `Program` file.

> ⚠ **Warning**
>
> Call UseStaticWebAssets for the ***exact environment*** to prevent activating the feature in production, as it serves files from separate locations on disk *other than from the project* if called in a production environment. The example in this section checks for the Staging environment by calling IsStaging.

C#

```
if (builder.Environment.IsStaging())
{
    builder.WebHost.UseStaticWebAssets();
}
```

Prefix for Blazor WebAssembly assets

This section applies to Blazor Web Apps.

Use the WebAssemblyComponentsEndpointOptions.PathPrefix endpoint option to set the path string that indicates the prefix for Blazor WebAssembly assets. The path must correspond to a referenced Blazor WebAssembly application project.

C#

```
endpoints.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode(options =>
        options.PathPrefix = "{PATH PREFIX}");
```

In the preceding example, the `{PATH PREFIX}` placeholder is the path prefix and must start with a forward slash (`/`).

In the following example, the path prefix is set to `/path-prefix`:

C#

```
endpoints.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode(options =>
```

```
        options.PathPrefix = "/path-prefix");
```

Static web asset base path

This section applies to standalone Blazor WebAssembly apps.

Publishing the app places the app's static assets, including Blazor framework files (`_framework` folder assets), at the root path (`/`) in published output. The `<StaticWebAssetBasePath>` property specified in the project file (`.csproj`) sets the base path to a non-root path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>{PATH}</StaticWebAssetBasePath>
</PropertyGroup>
```

In the preceding example, the `{PATH}` placeholder is the path.

Without setting the `<StaticWebAssetBasePath>` property, a standalone app is published at `/BlazorStandaloneSample/bin/Release/{TFM}/publish/wwwroot/`.

In the preceding example, the `{TFM}` placeholder is the Target Framework Moniker (TFM) (for example, `net6.0`).

If the `<StaticWebAssetBasePath>` property in a standalone Blazor WebAssembly app sets the published static asset path to `app1`, the root path to the app in published output is `/app1`.

In the standalone Blazor WebAssembly app's project file (`.csproj`):

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>app1</StaticWebAssetBasePath>
</PropertyGroup>
```

In published output, the path to the standalone Blazor WebAssembly app is `/BlazorStandaloneSample/bin/Release/{TFM}/publish/wwwroot/app1/`.

In the preceding example, the `{TFM}` placeholder is the Target Framework Moniker (TFM) (for example, `net6.0`).

File mappings and static file options

This section applies to server-side static files.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension, and the `{CONTENT TYPE}` placeholder is the content type. The namespace for the following API is Microsoft.AspNetCore.StaticFiles.

- Configure options through dependency injection (DI) in the `Program` file using StaticFileOptions:

  C#

  ```
  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  builder.Services.Configure<StaticFileOptions>(options =>
  {
      options.ContentTypeProvider = provider;
  });

  app.UseStaticFiles();
  ```

- Pass the StaticFileOptions directly to UseStaticFiles in the `Program` file:

  C#

  ```
  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider =
  provider });
  ```

Serve files from multiple locations

The guidance in this section only applies to Blazor Web Apps.

To serve files from multiple locations with a CompositeFileProvider:

- Add the namespace for Microsoft.Extensions.FileProviders to the top of the `Program` file of the server project.
- In the server project's `Program` file ***before*** the call to UseStaticFiles:
  - Create a PhysicalFileProvider with the path to the static assets.

- Create a CompositeFileProvider from the WebRootFileProvider and the PhysicalFileProvider. Assign the composite file provider back to the app's WebRootFileProvider.

Example:

Create a new folder in the server project named `AdditionalStaticAssets`. Place an image into the folder.

Add the following `using` statement to the top of the server project's `Program` file:

C#

```
using Microsoft.Extensions.FileProviders;
```

In the server project's `Program` file ***before*** the call to UseStaticFiles, add the following code:

C#

```
var secondaryProvider = new PhysicalFileProvider(
    Path.Combine(builder.Environment.ContentRootPath,
"AdditionalStaticAssets"));
app.Environment.WebRootFileProvider = new CompositeFileProvider(
    app.Environment.WebRootFileProvider, secondaryProvider);
```

In the app's `Home` component (`Home.razor`) markup, reference the image with an `<img>` tag:

razor

```
<img src="{IMAGE FILE NAME}" alt="{ALT TEXT}" />
```

In the preceding example:

- The `{IMAGE FILE NAME}` placeholder is the image file name. There's no need to provide a path segment if the image file is at the root of the `AdditionalStaticAssets` folder.
- The `{ALT TEXT}` placeholder is the image alternate text.

Run the app.

Additional resources

- App base path
- Avoid file capture in a route parameter

ASP.NET Core Razor components

Article • 10/18/2024

> **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 article explains how to create and use Razor components in Blazor apps, including guidance on Razor syntax, component naming, namespaces, and component parameters.

Razor components

Blazor apps are built using *Razor components*, informally known as *Blazor components* or only *components*. A component is a self-contained portion of user interface (UI) with processing logic to enable dynamic behavior. Components can be nested, reused, shared among projects, and used in MVC and Razor Pages apps.

Components render into an in-memory representation of the browser's Document Object Model (DOM) called a *render tree*, which is used to update the UI in a flexible and efficient way.

Although "Razor components" shares some naming with other ASP.NET Core content-rendering technologies, Razor components must be distinguished from the following different features in ASP.NET Core:

- Razor views, which are Razor-based markup pages for MVC apps.
- View components, which are for rendering chunks of content rather than whole responses in Razor Pages and MVC apps.

> **Important**
>
> When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component

adopts an interactive render mode. More information on this subject is provided by ASP.NET Core Blazor render modes, which is the next article in the table of contents after this article.

Component classes

Components are implemented using a combination of C# and HTML markup in Razor component files with the `.razor` file extension.

ComponentBase is the base class for components described by Razor component files. ComponentBase implements the lowest abstraction of components, the IComponent interface. ComponentBase defines component properties and methods for basic functionality, for example, to process a set of built-in component lifecycle events.

ComponentBase in dotnet/aspnetcore reference source : The reference source contains additional remarks on the built-in lifecycle events. However, keep in mind that the internal implementations of component features are subject to change at any time without notice.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205) .

Developers typically create Razor components from Razor component files (`.razor`) or base their components on ComponentBase, but components can also be built by implementing IComponent. Developer-built components that implement IComponent can take low-level control over rendering at the cost of having to manually trigger rendering with events and lifecycle methods that the developer must create and maintain.

Additional conventions adopted by Blazor documentation example code and sample apps is found in ASP.NET Core Blazor fundamentals.

Razor syntax

Components use Razor syntax. Two Razor features are extensively used by components, *directives* and *directive attributes*. These are reserved keywords prefixed with `@` that appear in Razor markup:

- Directives: Change the way component markup is compiled or functions. For example, the @page directive specifies a routable component with a route template that can be reached directly by a user's request in the browser at a specific URL.

  By convention, a component's directives at the top of a component definition (`.razor` file) are placed in a consistent order. For repeated directives, directives are placed alphabetically by namespace or type, except `@using` directives, which have special second-level ordering.

  The following order is adopted by Blazor sample apps and documentation. Components provided by a Blazor project template may differ from the following order and use a different format. For example, Blazor framework Identity components include blank lines between blocks of `@using` directives and blocks of `@inject` directives. You're free to use a custom ordering scheme and format in your own apps.

  Documentation and sample app Razor directive order:
  - `@page`
  - `@rendermode` (.NET 8 or later)
  - `@using`
    - `System` namespaces (alphabetical order)
    - `Microsoft` namespaces (alphabetical order)
    - Third-party API namespaces (alphabetical order)
    - App namespaces (alphabetical order)
  - Other directives (alphabetical order)

  No blank lines appear among the directives. One blank line appears between the directives and the first line of Razor markup.

  Example:

  razor

  ```
  @page "/doctor-who-episodes/{season:int}"
  @rendermode InteractiveWebAssembly
  @using System.Globalization
  @using System.Text.Json
  @using Microsoft.AspNetCore.Localization
  @using Mandrill
  ```

```
@using BlazorSample.Components.Layout
@attribute [Authorize]
@implements IAsyncDisposable
@inject IJSRuntime JS
@inject ILogger<DoctorWhoEpisodes> Logger

<PageTitle>Doctor Who Episode List</PageTitle>

...
```

- Directive attributes: Change the way a component element is compiled or functions.

  Example:

  ```
  razor

  <input @bind="episodeId" />
  ```

  You can prefix directive attribute values with the at symbol (`@`) for non-explicit Razor expressions (`@bind="@episodeId"`), but we don't recommend it, and the docs don't adopt the approach in examples.

Directives and directive attributes used in components are explained further in this article and other articles of the Blazor documentation set. For general information on Razor syntax, see Razor syntax reference for ASP.NET Core.

Component name, class name, and namespace

A component's name must start with an uppercase character:

✔ `ProductDetail.razor`

✘ `productDetail.razor`

Common Blazor naming conventions used throughout the Blazor documentation include:

- File paths and file names use Pascal case† and appear before showing code examples. If a path is present, it indicates the typical folder location. For example, `Components/Pages/ProductDetail.razor` indicates that the `ProductDetail` component has a file name of `ProductDetail.razor` and resides in the `Pages` folder of the `Components` folder of the app.
- Component file paths for routable components match their URLs in kebab case‡ with hyphens appearing between words in a component's route template. For

example, a `ProductDetail` component with a route template of `/product-detail` (`@page "/product-detail"`) is requested in a browser at the relative URL `/product-detail`.

†Pascal case (upper camel case) is a naming convention without spaces and punctuation and with the first letter of each word capitalized, including the first word.
‡Kebab case is a naming convention without spaces and punctuation that uses lowercase letters and dashes between words.

Components are ordinary C# classes and can be placed anywhere within a project. Components that produce webpages usually reside in the `Components/Pages` folder. Non-page components are frequently placed in the `Components` folder or a custom folder added to the project.

Typically, a component's namespace is derived from the app's root namespace and the component's location (folder) within the app. If the app's root namespace is `BlazorSample` and the `Counter` component resides in the `Components/Pages` folder:

- The `Counter` component's namespace is `BlazorSample.Components.Pages`.
- The fully qualified type name of the component is `BlazorSample.Components.Pages.Counter`.

For custom folders that hold components, add an @using directive to the parent component or to the app's `_Imports.razor` file. The following example makes components in the `AdminComponents` folder available:

razor

```
@using BlazorSample.AdminComponents
```

> **Note**
>
> @using directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`).

Aliased using statements are supported. In the following example, the public `WeatherForecast` class of the `GridRendering` component is made available as `WeatherForecast` in a component elsewhere in the app:

razor

```
@using WeatherForecast = Components.Pages.GridRendering.WeatherForecast
```

Components can also be referenced using their fully qualified names, which doesn't require an @using directive. The following example directly references the `ProductDetail` component in the `AdminComponents/Pages` folder of the app:

razor

```
<BlazorSample.AdminComponents.Pages.ProductDetail />
```

The namespace of a component authored with Razor is based on the following (in priority order):

- The @namespace directive in the Razor file's markup (for example, `@namespace BlazorSample.CustomNamespace`).
- The project's `RootNamespace` in the project file (for example, `<RootNamespace>BlazorSample</RootNamespace>`).
- The project namespace and the path from the project root to the component. For example, the framework resolves `{PROJECT NAMESPACE}/Components/Pages/Home.razor` with a project namespace of `BlazorSample` to the namespace `BlazorSample.Components.Pages` for the `Home` component. `{PROJECT NAMESPACE}` is the project namespace. Components follow C# name binding rules. For the `Home` component in this example, the components in scope are all of the components:
  - In the same folder, `Components/Pages`.
  - The components in the project's root that don't explicitly specify a different namespace.

The following are **not** supported:

- The global:: qualification.
- Partially-qualified names. For example, you can't add `@using BlazorSample.Components` to a component and then reference the `NavMenu` component in the app's `Components/Layout` folder (`Components/Layout/NavMenu.razor`) with `<Layout.NavMenu></Layout.NavMenu>`.

Partial class support

Components are generated as C# partial classes and are authored using either of the following approaches:

- A single file contains C# code defined in one or more @code blocks, HTML markup, and Razor markup. Blazor project templates define their components using this single-file approach.
- HTML and Razor markup are placed in a Razor file (`.razor`). C# code is placed in a code-behind file defined as a partial class (`.cs`).

> **Note**
>
> A component stylesheet that defines component-specific styles is a separate file (`.css`). Blazor CSS isolation is described later in ASP.NET Core Blazor CSS isolation.

The following example shows the default `Counter` component with an @code block in an app generated from a Blazor project template. Markup and C# code are in the same file. This is the most common approach taken in component authoring.

`Counter.razor`:

```razor
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

The following `Counter` component splits presentation HTML and Razor markup from the C# code using a code-behind file with a partial class. Splitting the markup from the C# code is favored by some organizations and developers to organize their component code to suit how they prefer to work. For example, the organization's UI expert can work on the presentation layer independently of another developer working on the component's C# logic. The approach is also useful when working with automatically-generated code or source generators. For more information, see Partial Classes and Methods (C# Programming Guide).

`CounterPartialClass.razor`:

razor

```razor
@page "/counter-partial-class"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

`CounterPartialClass.razor.cs`:

C#

```csharp
namespace BlazorSample.Components.Pages;

public partial class CounterPartialClass
{
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

@using directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`). Add namespaces to a partial class file as needed.

Typical namespaces used by components:

C#

```csharp
using System.Net.Http;
using System.Net.Http.Json;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Forms;
using Microsoft.AspNetCore.Components.Routing;
using Microsoft.AspNetCore.Components.Sections
using Microsoft.AspNetCore.Components.Web;
using static Microsoft.AspNetCore.Components.Web.RenderMode;
using Microsoft.AspNetCore.Components.Web.Virtualization;
using Microsoft.JSInterop;
```

Typical namespaces also include the namespace of the app and the namespace corresponding to the app's `Components` folder:

C#

```
using BlazorSample;
using BlazorSample.Components;
```

Additional folders can also be included, such as the `Layout` folder:

razor

```
using BlazorSample.Components.Layout;
```

Specify a base class

The @inherits directive is used to specify a base class for a component. Unlike using partial classes, which only split markup from C# logic, using a base class allows you to inherit C# code for use across a group of components that share the base class's properties and methods. Using base classes reduce code redundancy in apps and are useful when supplying base code from class libraries to multiple apps. For more information, see Inheritance in C# and .NET.

In the following example, the `BlazorRocksBase1` base class derives from ComponentBase.

`BlazorRocks1.razor`:

razor

```
@page "/blazor-rocks-1"
@inherits BlazorRocksBase1

<PageTitle>Blazor Rocks!</PageTitle>

<h1>Blazor Rocks! Example 1</h1>

<p>
    @BlazorRocksText
</p>
```

`BlazorRocksBase1.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample;

public class BlazorRocksBase1 : ComponentBase
```

```
{
    public string BlazorRocksText { get; set; } = "Blazor rocks the
browser!";
}
```

Routing

Routing in Blazor is achieved by providing a route template to each accessible component in the app with an @page directive. When a Razor file with an @page directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router searches for component classes with a RouteAttribute and renders whichever component has a route template that matches the requested URL.

The following `HelloWorld` component uses a route template of `/hello-world`, and the rendered webpage for the component is reached at the relative URL `/hello-world`.

`HelloWorld.razor`:

razor

```
@page "/hello-world"

<PageTitle>Hello World!</PageTitle>

<h1>Hello World!</h1>
```

The preceding component loads in the browser at `/hello-world` regardless of whether or not you add the component to the app's UI navigation. Optionally, components can be added to the `NavMenu` component so that a link to the component appears in the app's UI-based navigation.

For the preceding `HelloWorld` component, you can add a `NavLink` component to the `NavMenu` component. For more information, including descriptions of the `NavLink` and `NavMenu` components, see ASP.NET Core Blazor routing and navigation.

Markup

A component's UI is defined using Razor syntax, which consists of Razor markup, C#, and HTML. When an app is compiled, the HTML markup and C# rendering logic are converted into a component class. The name of the generated class matches the name of the file.

Members of the component class are defined in one or more @code blocks. In @code blocks, component state is specified and processed with C#:

- Property and field initializers.
- Parameter values from arguments passed by parent components and route parameters.
- Methods for user event handling, lifecycle events, and custom component logic.

Component members are used in rendering logic using C# expressions that start with the `@` symbol. For example, a C# field is rendered by prefixing `@` to the field name. The following `Markup` component evaluates and renders:

- `headingFontStyle` for the CSS property value `font-style` of the heading element.
- `headingText` for the content of the heading element.

`Markup.razor`:

razor

```
@page "/markup"

<PageTitle>Markup</PageTitle>

<h1>Markup Example</h1>

<h2 style="font-style:@headingFontStyle">@headingText</h2>

@code {
    private string headingFontStyle = "italic";
    private string headingText = "Put on your new Blazor!";
}
```

> **Note**
>
> Examples throughout the Blazor documentation specify the private access modifier for private members. Private members are scoped to a component's class. However, C# assumes the `private` access modifier when no access modifier is present, so explicitly marking members "`private`" in your own code is optional. For more information on access modifiers, see Access Modifiers (C# Programming Guide).

The Blazor framework processes a component internally as a *render tree*, which is the combination of a component's DOM and Cascading Style Sheet Object Model (CSSOM). After the component is initially rendered, the component's render tree is regenerated in response to events. Blazor compares the new render tree against the

previous render tree and applies any modifications to the browser's DOM for display. For more information, see ASP.NET Core Razor component rendering.

Razor syntax for C# control structures, directives, and directive attributes are lowercase (examples: @if, @code, @bind). Property names are uppercase (example: `@Body` for LayoutComponentBase.Body).

Asynchronous methods (`async`) don't support returning `void`

The Blazor framework doesn't track `void`-returning asynchronous methods (`async`). As a result, exceptions aren't caught if `void` is returned. Always return a Task from asynchronous methods.

Nested components

Components can include other components by declaring them using HTML syntax. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

Consider the following `Heading` component, which can be used by other components to display a heading.

`Heading.razor`:

razor

```razor
<h1 style="font-style:@headingFontStyle">Heading Example</h1>

@code {
    private string headingFontStyle = "italic";
}
```

The following markup in the `HeadingExample` component renders the preceding `Heading` component at the location where the `<Heading />` tag appears.

`HeadingExample.razor`:

razor

```razor
@page "/heading-example"

<PageTitle>Heading</PageTitle>
```

```
<h1>Heading Example</h1>

<Heading />
```

If a component contains an HTML element with an uppercase first letter that doesn't match a component name within the same namespace, a warning is emitted indicating that the element has an unexpected name. Adding an @using directive for the component's namespace makes the component available, which resolves the warning. For more information, see the Component name, class name, and namespace section.

The `Heading` component example shown in this section doesn't have an @page directive, so the `Heading` component isn't directly accessible to a user via a direct request in the browser. However, any component with an @page directive can be nested in another component. If the `Heading` component was directly accessible by including `@page "/heading"` at the top of its Razor file, then the component would be rendered for browser requests at both `/heading` and `/heading-example`.

Component parameters

Component parameters pass data to components and are defined using public C# properties on the component class with the [Parameter] attribute. In the following example, a built-in reference type (System.String) and a user-defined reference type (`PanelBody`) are passed as component parameters.

`PanelBody.cs`:

C#

```
namespace BlazorSample;

public class PanelBody
{
    public string? Text { get; set; }
    public string? Style { get; set; }
}
```

`ParameterChild.razor`:

razor

```
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">@Title</div>
    <div class="card-body" style="font-style:@Body.Style">
        @Body.Text
```

```
    </div>
</div>

@code {
    [Parameter]
    public string Title { get; set; } = "Set By Child";

    [Parameter]
    public PanelBody Body { get; set; } =
        new()
        {
            Text = "Card content set by child.",
            Style = "normal"
        };
}
```

> ⚠ **Warning**
>
> Providing initial values for component parameters is supported, but don't create a component that writes to its own parameters after the component is rendered for the first time. For more information, see Avoid overwriting parameters in ASP.NET Core Blazor.

The `Title` and `Body` component parameters of the `ParameterChild` component are set by arguments in the HTML tag that renders the instance of the component. The following `ParameterParent` component renders two `ParameterChild` components:

- The first `ParameterChild` component is rendered without supplying parameter arguments.
- The second `ParameterChild` component receives values for `Title` and `Body` from the `ParameterParent` component, which uses an explicit C# expression to set the values of the `PanelBody`'s properties.

`Parameter1.razor`:

razor

```
@page "/parameter-1"

<PageTitle>Parameter 1</PageTitle>

<h1>Parameter Example 1</h1>

<h1>Child component (without attribute values)</h1>

<ParameterChild />
```

```
<h1>Child component (with attribute values)</h1>

<ParameterChild Title="Set by Parent"
    Body="@(new PanelBody() { Text = "Set by parent.", Style = "italic" })"
/>
```

The following rendered HTML markup from the `ParameterParent` component shows `ParameterChild` component default values when the `ParameterParent` component doesn't supply component parameter values. When the `ParameterParent` component provides component parameter values, they replace the `ParameterChild` component's default values.

> **Note**
>
> For clarity, rendered CSS style classes aren't shown in the following rendered HTML markup.

HTML

```
<h1>Child component (without attribute values)</h1>

<div>
    <div>Set By Child</div>
    <div>Set by child.</div>
</div>

<h1>Child component (with attribute values)</h1>

<div>
    <div>Set by Parent</div>
    <div>Set by parent.</div>
</div>
```

Assign a C# field, property, or result of a method to a component parameter as an HTML attribute value. The value of the attribute can typically be any C# expression that matches the type of the parameter. The value of the attribute can optionally lead with a Razor reserved @ symbol, but it isn't required.

If the component parameter is of type string, then the attribute value is instead treated as a C# string literal. If you want to specify a C# expression instead, then use the `@` prefix.

The following `ParameterParent2` component displays four instances of the preceding `ParameterChild` component and sets their `Title` parameter values to:

- The value of the `title` field.
- The result of the `GetTitle` C# method.
- The current local date in long format with ToLongDateString, which uses an implicit C# expression.
- The `panelData` object's `Title` property.

Quotes around parameter attribute values are optional in most cases per the HTML5 specification. For example, `Value=this` is supported, instead of `Value="this"`. However, we recommend using quotes because it's easier to remember and widely adopted across web-based technologies.

Throughout the documentation, code examples:

- Always use quotes. Example: `Value="this"`.
- Don't use the `@` prefix with nonliterals unless required. Example: `Count="ct"`, where `ct` is a number-typed variable. `Count="@ct"` is a valid stylistic approach, but the documentation and examples don't adopt the convention.
- Always avoid `@` for literals, outside of Razor expressions. Example: `IsFixed="true"`. This includes keywords (for example, `this`) and `null`, but you can choose to use them if you wish. For example, `IsFixed="@true"` is uncommon but supported.

`Parameter2.razor`:

razor

```
@page "/parameter-2"

<PageTitle>Parameter 2</PageTitle>

<h1>Parameter Example 2</h1>

<ParameterChild Title="@title" />

<ParameterChild Title="@GetTitle()" />

<ParameterChild Title="@DateTime.Now.ToLongDateString()" />

<ParameterChild Title="@panelData.Title" />

@code {
    private string title = "From Parent field";
    private PanelData panelData = new();

    private string GetTitle() => "From Parent method";

    private class PanelData
    {
        public string Title { get; set; } = "From Parent object";
```

```
    }
}
```

> **Note**
>
> When assigning a C# member to a component parameter, don't prefix the parameter's HTML attribute with `@`.
>
> Correct (`Title` is a string parameter, `Count` is a number-typed parameter):
>
> ```razor
> <ParameterChild Title="@title" Count="ct" />
> ```
>
> ```razor
> <ParameterChild Title="@title" Count="@ct" />
> ```
>
> Incorrect:
>
> ```razor
> <ParameterChild @Title="@title" @Count="ct" />
> ```
>
> ```razor
> <ParameterChild @Title="@title" @Count="@ct" />
> ```

Unlike in Razor pages (`.cshtml`), Blazor can't perform asynchronous work in a Razor expression while rendering a component. This is because Blazor is designed for rendering interactive UIs. In an interactive UI, the screen must always display something, so it doesn't make sense to block the rendering flow. Instead, asynchronous work is performed during one of the asynchronous lifecycle events. After each asynchronous lifecycle event, the component may render again. The following Razor syntax is **not** supported:

```razor
<ParameterChild Title="await ..." />
<ParameterChild Title="@await ..." />
```

The code in the preceding example generates a *compiler error* when the app is built:

> The 'await' operator can only be used within an async method. Consider marking this method with the 'async' modifier and changing its return type to 'Task'.

To obtain a value for the `Title` parameter in the preceding example asynchronously, the component can use the OnInitializedAsync lifecycle event, as the following example demonstrates:

razor

```
<ParameterChild Title="@title" />

@code {
    private string? title;

    protected override async Task OnInitializedAsync()
    {
        title = await ...;
    }
}
```

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

Use of an explicit Razor expression to concatenate text with an expression result for assignment to a parameter is **not** supported. The following example seeks to concatenate the text "`Set by `" with an object's property value. Although this syntax is supported in a Razor page (`.cshtml`), it isn't valid for assignment to the child's `Title` parameter in a component. The following Razor syntax is **not** supported:

razor

```
<ParameterChild Title="Set by @(panelData.Title)" />
```

The code in the preceding example generates a *compiler error* when the app is built:

> Component attributes do not support complex content (mixed C# and markup).

To support the assignment of a composed value, use a method, field, or property. The following example performs the concatenation of "`Set by `" and an object's property value in the C# method `GetTitle`:

`Parameter3.razor`:

razor

```razor
@page "/parameter-3"

<PageTitle>Parameter 3</PageTitle>

<h1>Parameter Example 3</h1>

<ParameterChild Title="@GetTitle()" />

@code {
    private PanelData panelData = new();

    private string GetTitle() => $"Set by {panelData.Title}";

    private class PanelData
    {
        public string Title { get; set; } = "Parent";
    }
}
```

For more information, see Razor syntax reference for ASP.NET Core.

> ⚠ **Warning**
>
> Providing initial values for component parameters is supported, but don't create a component that writes to its own parameters after the component is rendered for the first time. For more information, see **Avoid overwriting parameters in ASP.NET Core Blazor**.

Component parameters should be declared as *auto-properties*, meaning that they shouldn't contain custom logic in their `get` or `set` accessors. For example, the following `StartData` property is an auto-property:

C#

```
[Parameter]
public DateTime StartData { get; set; }
```

Don't place custom logic in the `get` or `set` accessor because component parameters are purely intended for use as a channel for a parent component to flow information to a child component. If a `set` accessor of a child component property contains logic that causes rerendering of the parent component, an infinite rendering loop results.

To transform a received parameter value:

- Leave the parameter property as an auto-property to represent the supplied raw data.
- Create a different property or method to supply the transformed data based on the parameter property.

Override OnParametersSetAsync to transform a received parameter each time new data is received.

Writing an initial value to a component parameter is supported because initial value assignments don't interfere with the Blazor's automatic component rendering. The following assignment of the current local DateTime with DateTime.Now to `StartData` is valid syntax in a component:

C#

```
[Parameter]
public DateTime StartData { get; set; } = DateTime.Now;
```

After the initial assignment of DateTime.Now, do **not** assign a value to `StartData` in developer code. For more information, see Avoid overwriting parameters in ASP.NET Core Blazor.

Apply the [EditorRequired] attribute to specify a required component parameter. If a parameter value isn't provided, editors or build tools may display warnings to the user. This attribute is only valid on properties also marked with the [Parameter] attribute. The EditorRequiredAttribute is enforced at design-time and when the app is built. The attribute isn't enforced at runtime, and it doesn't guarantee a non-`null` parameter value.

C#

```
[Parameter]
[EditorRequired]
public string? Title { get; set; }
```

Single-line attribute lists are also supported:

C#

```
[Parameter, EditorRequired]
public string? Title { get; set; }
```

Don't use the required modifier or init accessor on component parameter properties. Components are usually instantiated and assigned parameter values using reflection,

which bypasses the guarantees that `init` and `required` are designed to make. Instead, use the [EditorRequired] attribute to specify a required component parameter.

Tuples (API documentation) are supported for component parameters and RenderFragment types. The following component parameter example passes three values in a `Tuple`:

`RenderTupleChild.razor`:

razor

```razor
<div class="card w-50" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Tuple Card</div>
    <div class="card-body">
        <ul>
            <li>Integer: @Data?.Item1</li>
            <li>String: @Data?.Item2</li>
            <li>Boolean: @Data?.Item3</li>
        </ul>
    </div>
</div>

@code {
    [Parameter]
    public (int, string, bool)? Data { get; set; }
}
```

`RenderTupleParent.razor`:

razor

```razor
@page "/render-tuple-parent"

<PageTitle>Render Tuple Parent</PageTitle>

<h1>Render Tuple Parent Example</h1>

<RenderTupleChild Data="data" />

@code {
    private (int, string, bool) data = new(999, "I aim to misbehave.",
true);
}
```

Named tuples are supported, as seen in the following example:

`NamedTupleChild.razor`:

razor

```
<div class="card w-50" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Tuple Card</div>
    <div class="card-body">
        <ul>
            <li>Integer: @Data?.TheInteger</li>
            <li>String: @Data?.TheString</li>
            <li>Boolean: @Data?.TheBoolean</li>
        </ul>
    </div>
</div>

@code {
    [Parameter]
    public (int TheInteger, string TheString, bool TheBoolean)? Data { get;
set; }
}
```

`NamedTuples.razor`:

razor

```
@page "/named-tuples"

<PageTitle>Named Tuples</PageTitle>

<h1>Named Tuples Example</h1>

<NamedTupleChild Data="data" />

@code {
    private (int TheInteger, string TheString, bool TheBoolean) data =
        new(999, "I aim to misbehave.", true);
}
```

Quote ©2005 Universal Pictures: Serenity (Nathan Fillion)

Route parameters

Components can specify route parameters in the route template of the @page directive. The Blazor router uses route parameters to populate corresponding component parameters.

`RouteParameter1.razor`:

razor

```
@page "/route-parameter-1/{text}"
```

```razor
<PageTitle>Route Parameter 1</PageTitle>

<h1>Route Parameter Example 1</h1>

<p>Blazor is @Text!</p>

@code {
    [Parameter]
    public string? Text { get; set; }
}
```

For more information, see the *Route parameters* section of ASP.NET Core Blazor routing and navigation. Optional route parameters are also supported and covered in the same section. For information on catch-all route parameters (`{*pageRoute}`), which capture paths across multiple folder boundaries, see the *Catch-all route parameters* section of ASP.NET Core Blazor routing and navigation.

> ⚠ **Warning**
>
> With compression, which is enabled by default, avoid creating secure (authenticated/authorized) interactive server-side components that render data from untrusted sources. Untrusted sources include route parameters, query strings, data from JS interop, and any other source of data that a third-party user can control (databases, external services). For more information, see **ASP.NET Core Blazor SignalR guidance** and **Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering**.

Child content render fragments

Components can set the content of another component. The assigning component provides the content between the child component's opening and closing tags.

In the following example, the `RenderFragmentChild` component has a `ChildContent` component parameter that represents a segment of the UI to render as a RenderFragment. The position of `ChildContent` in the component's Razor markup is where the content is rendered in the final HTML output.

`RenderFragmentChild.razor`:

razor

```razor
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Child content</div>
    <div class="card-body">@ChildContent</div>
```

```
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

> **Important**
>
> The property receiving the RenderFragment content must be named `ChildContent` by convention.
>
> Event callbacks aren't supported for RenderFragment.

The following component provides content for rendering the `RenderFragmentChild` by placing the content inside the child component's opening and closing tags.

`RenderFragments.razor`:

razor

```
@page "/render-fragments"

<PageTitle>Render Fragments</PageTitle>

<h1>Render Fragments Example</h1>

<RenderFragmentChild>
    Content of the child component is supplied
    by the parent component.
</RenderFragmentChild>
```

Render fragments are used to render child content throughout Blazor apps and are described with examples in the following articles and article sections:

- Blazor layouts
- Pass data across a component hierarchy
- Templated components
- Global exception handling

> **Note**
>
> Blazor framework's built-in Razor components use the same `ChildContent` component parameter convention to set their content. You can see the components that set child content by searching for the component parameter

property name `ChildContent` in the API documentation (filters API with the search term "ChildContent").

Render fragments for reusable rendering logic

You can factor out child components purely as a way of reusing rendering logic. In any component's `@code` block, define a RenderFragment and render the fragment from any location as many times as needed:

razor

```
@RenderWelcomeInfo

<p>Render the welcome info a second time:</p>

@RenderWelcomeInfo

@code {
    private RenderFragment RenderWelcomeInfo =  @<p>Welcome to your new app!
</p>;
}
```

For more information, see Reuse rendering logic.

Loop variables with component parameters and child content

Rendering components inside a for loop requires a local index variable if the incrementing loop variable is used by the component's parameters or RenderFragment child content.

Consider the following `RenderFragmentChild2` component that has both a component parameter (`Id`) and a render fragment to display child content (`ChildContent`).

`RenderFragmentChild2.razor`:

razor

```
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Child content (@Id)</div>
    <div class="card-body">@ChildContent</div>
</div>

@code {
```

```
    [Parameter]
    public string? Id { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

When rendering the `RenderFragmentChild2` component in a parent component, use a local index variable (`ct` in the following example) instead of the loop variable (`c`) when assigning the component parameter value and providing the child component's content:

razor

```
@for (int c = 1; c < 4; c++)
{
    var ct = c;

    <RenderFragmentChild2 Id="@($"Child{ct}")">
        Count: @ct
    </RenderFragmentChild2>
}
```

Alternatively, use a foreach loop with Enumerable.Range instead of a for loop:

razor

```
@foreach (var c in Enumerable.Range(1, 3))
{
    <RenderFragmentChild2 Id="@($"Child{c}")">
        Count: @c
    </RenderFragmentChild2>
}
```

Capture references to components

Component references provide a way to reference a component instance for issuing commands. To capture a component reference:

- Add an @ref attribute to the child component.
- Define a field with the same type as the child component.

When the component is rendered, the field is populated with the component instance. You can then invoke .NET methods on the instance.

Consider the following `ReferenceChild` component that logs a message when its `ChildMethod` is called.

`ReferenceChild.razor`:

razor

```razor
@inject ILogger<ReferenceChild> Logger

@if (value > 0)
{
    <p>
        <code>value</code>: @value
    </p>
}

@code {
    private int value;

    public void ChildMethod(int value)
    {
        Logger.LogInformation("Received {Value} in ChildMethod", value);

        this.value = value;
        StateHasChanged();
    }
}
```

A component reference is only populated after the component is rendered and its output includes `ReferenceChild`'s element. Until the component is rendered, there's nothing to reference. Don't attempt to call a referenced component method to an event handler directly (for example, `@onclick="childComponent!.ChildMethod(5)"`) because the reference variable may not be assigned at the time the click event is assigned.

To manipulate component references after the component has finished rendering, use the OnAfterRender or OnAfterRenderAsync methods.

The following example uses the preceding `ReferenceChild` component.

`ReferenceParent.razor`:

razor

```razor
@page "/reference-parent"

<div>
    <button @onclick="@(() => childComponent1!.ChildMethod(5))">
        Call <code>ReferenceChild.ChildMethod</code> (first instance)
        with an argument of 5
```

```
    </button>

    <ReferenceChild @ref="childComponent1" />
</div>

<div>
    <button @onclick="CallChildMethod">
        Call <code>ReferenceChild.ChildMethod</code> (second instance)
        with an argument of 5
    </button>

    <ReferenceChild @ref="childComponent2" />
</div>

@code {
    private ReferenceChild? childComponent1;
    private ReferenceChild? childComponent2;

    private void CallChildMethod() => childComponent2!.ChildMethod(5);
}
```

While capturing component references use a similar syntax to capturing element references, capturing component references isn't a JavaScript interop feature. Component references aren't passed to JavaScript code. Component references are only used in .NET code.

> **Important**
>
> Do **not** use component references to mutate the state of child components. Instead, use normal declarative component parameters to pass data to child components. Use of component parameters result in child components that rerender at the correct times automatically. For more information, see the component parameters section and the ASP.NET Core Blazor data binding article.

Apply an attribute

Attributes can be applied to components with the @attribute directive. The following example applies the [Authorize] attribute to the component's class:

razor

```
@page "/"
@attribute [Authorize]
```

Conditional HTML element attributes and DOM properties

Blazor adopts the following general behaviors:

- For HTML attributes, Blazor sets or removes the attribute conditionally based on the .NET value. If the .NET value is `false` or `null`, the attribute isn't set or is removed if it was previously set.
- For DOM properties, such as `checked` or `value`, Blazor sets the DOM property based on the .NET value. If the .NET value is `false` or `null`, the DOM property is reset to a default value.

Which Razor syntax attributes correspond to HTML attributes and which ones correspond to DOM properties remains undocumented because this is a framework implementation detail that might change without notice.

> ⚠ **Warning**
>
> Some HTML attributes, such as aria-pressed, must have a string value of either "true" or "false". Since they require a string value and not a boolean, you must use a .NET `string` and not a `bool` for their value. This is a requirement set by browser DOM APIs.

Raw HTML

Strings are normally rendered using DOM text nodes, which means that any markup they may contain is ignored and treated as literal text. To render raw HTML, wrap the HTML content in a MarkupString value. The value is parsed as HTML or SVG and inserted into the DOM.

> ⚠ **Warning**
>
> Rendering raw HTML constructed from any untrusted source is a **security risk** and should **always** be avoided.

The following example shows using the MarkupString type to add a block of static HTML content to the rendered output of a component.

`MarkupStrings.razor`:

razor

```razor
@page "/markup-strings"

<PageTitle>Markup Strings</PageTitle>

<h1>Markup Strings Example</h1>

@((MarkupString)myMarkup)

@code {
    private string myMarkup =
        "<p class=\"text-danger\">This is a dangerous <em>markup
string</em>.</p>";
}
```

Razor templates

Render fragments can be defined using Razor template syntax to define a UI snippet. Razor templates use the following format:

razor

```razor
@<{HTML tag}>...</{HTML tag}>
```

The following example illustrates how to specify RenderFragment and RenderFragment<TValue> values and render templates directly in a component. Render fragments can also be passed as arguments to templated components.

`RazorTemplate.razor`:

razor

```razor
@page "/razor-template"

<PageTitle>Razor Template</PageTitle>

<h1>Razor Template Example</h1>

@timeTemplate

@petTemplate(new Pet { Name = "Nutty Rex" })

@code {
    private RenderFragment timeTemplate = @<p>The time is @DateTime.Now.
</p>;
    private RenderFragment<Pet> petTemplate = (pet) => @<p>Pet:
@pet.Name</p>;
```

```
    private class Pet
    {
        public string? Name { get; set; }
    }
}
```

Rendered output of the preceding code:

HTML

```
<p>The time is 4/19/2021 8:54:46 AM.</p>
<p>Pet: Nutty Rex</p>
```

Static assets

Blazor follows the convention of ASP.NET Core apps for static assets. Static assets are located in the project's web root (wwwroot) folder or folders under the `wwwroot` folder.

Use a base-relative path (`/`) to refer to the web root for a static asset. In the following example, `logo.png` is physically located in the `{PROJECT ROOT}/wwwroot/images` folder. `{PROJECT ROOT}` is the app's project root.

razor

```
<img alt="Company logo" src="/images/logo.png" />
```

Components do **not** support tilde-slash notation (`~/`).

For information on setting an app's base path, see Host and deploy ASP.NET Core Blazor.

Tag Helpers aren't supported in components

Tag Helpers aren't supported in components. To provide Tag Helper-like functionality in Blazor, create a component with the same functionality as the Tag Helper and use the component instead.

Scalable Vector Graphics (SVG) images

Since Blazor renders HTML, browser-supported images, including Scalable Vector Graphics (SVG) images (.svg), are supported via the `<img>` tag:

HTML

```
<img alt="Example image" src="image.svg" />
```

Similarly, SVG images are supported in the CSS rules of a stylesheet file (`.css`):

css

```
.element-class {
    background-image: url("image.svg");
}
```

Blazor supports the <foreignObject> element to display arbitrary HTML within an SVG. The markup can represent arbitrary HTML, a RenderFragment, or a Razor component.

The following example demonstrates:

- Display of a `string` (`@message`).
- Two-way binding with an `<input>` element and a `value` field.
- A `Robot` component.

razor

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
    <rect x="0" y="0" rx="10" ry="10" width="200" height="200"
stroke="black"
        fill="none" />
    <foreignObject x="20" y="20" width="160" height="160">
        <p>@message</p>
    </foreignObject>
</svg>

<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject width="200" height="200">
        <label>
            Two-way binding:
            <input @bind="value" @bind:event="oninput" />
        </label>
    </foreignObject>
</svg>

<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject>
        <Robot />
    </foreignObject>
</svg>

@code {
```

```
    private string message = "Lorem ipsum dolor sit amet, consectetur
adipiscing " +
        "elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua.";

    private string? value;
}
```

Whitespace rendering behavior

Unless the @preservewhitespace directive is used with a value of `true`, extra whitespace is removed if:

- Leading or trailing within an element.
- Leading or trailing within a RenderFragment/RenderFragment<TValue> parameter (for example, child content passed to another component).
- It precedes or follows a C# code block, such as `@if` or `@foreach`.

Whitespace removal might affect the rendered output when using a CSS rule, such as `white-space: pre`. To disable this performance optimization and preserve the whitespace, take one of the following actions:

- Add the `@preservewhitespace true` directive at the top of the Razor file (`.razor`) to apply the preference to a specific component.
- Add the `@preservewhitespace true` directive inside an `_Imports.razor` file to apply the preference to a subdirectory or to the entire project.

In most cases, no action is required, as apps typically continue to behave normally (but faster). If stripping whitespace causes a rendering problem for a particular component, use `@preservewhitespace true` in that component to disable this optimization.

Root component

A *root Razor component* (*root component*) is the first component loaded of any component hierarchy created by the app.

In an app created from the Blazor Web App project template, the `App` component (`App.razor`) is specified as the default root component by the type parameter declared for the call to MapRazorComponents<TRootComponent> in the server-side `Program` file. The following example shows the use of the `App` component as the root component, which is the default for an app created from the Blazor project template:

```
```

C#

```
app.MapRazorComponents<App>();
```

> **Note**
>
> Making a root component interactive, such as the `App` component, isn't supported.

In an app created from the Blazor WebAssembly project template, the `App` component (`App.razor`) is specified as the default root component in the `Program` file:

C#

```
builder.RootComponents.Add<App>("#app");
```

In the preceding code, the CSS selector, `#app`, indicates that the `App` component is specified for the `<div>` in `wwwroot/index.html` with an `id` of `app`:

HTML

```
<div id="app">...</app>
```

MVC and Razor Pages apps can also use the Component Tag Helper to register statically-rendered Blazor WebAssembly root components:

CSHTML

```
<component type="typeof(App)" render-mode="WebAssemblyPrerendered" />
```

Statically-rendered components can only be added to the app. They can't be removed or updated afterwards.

For more information, see the following resources:

- Component Tag Helper in ASP.NET Core
- Integrate ASP.NET Core Razor components into ASP.NET Core apps

ASP.NET Core Blazor render modes

Article • 11/06/2024

This article explains control of Razor component rendering in Blazor Web Apps, either at compile time or runtime.

This guidance doesn't apply to standalone Blazor WebAssembly apps. Blazor WebAssembly apps only render on the client via a client-side WebAssembly-based runtime and have no concept of a *render mode*. If a render mode is applied to a component in a Blazor WebAssembly app, the render mode designation has no influence on rendering the component.

Render modes

Every component in a Blazor Web App adopts a *render mode* to determine the hosting model that it uses, where it's rendered, and whether or not it's interactive.

The following table shows the available render modes for rendering Razor components in a Blazor Web App. To apply a render mode to a component use the `@rendermode` directive on the component instance or on the component definition. Later in this article, examples are shown for each render mode scenario.

Expand table

Name	Description	Render location	Interactive
Static Server	Static server-side rendering (static SSR)	Server	✗
Interactive Server	Interactive server-side rendering (interactive SSR) using Blazor Server.	Server	✓
Interactive WebAssembly	Client-side rendering (CSR) using Blazor WebAssembly†.	Client	✓
Interactive Auto	Interactive SSR using Blazor Server initially and then CSR on subsequent visits after the Blazor bundle is downloaded.	Server, then client	✓

†Client-side rendering (CSR) is assumed to be interactive. "*Interactive* client-side rendering" and "*interactive* CSR" aren't used by the industry or in the Blazor documentation.

Prerendering is enabled by default for interactive components. Guidance on controlling prerendering is provided later in this article. For general industry terminology on client and server rendering concepts, see ASP.NET Core Blazor fundamentals.

The following examples demonstrate setting the component's render mode with a few basic Razor component features.

To test the render mode behaviors locally, you can place the following components in an app created from the *Blazor Web App* project template. When you create the app, select options from dropdown menus (Visual Studio) or apply the CLI options (.NET CLI) to enable both server-side and client-side interactivity. For guidance on how to create a Blazor Web App, see Tooling for ASP.NET Core Blazor.

Enable support for interactive render modes

A Blazor Web App must be configured to support interactive render modes. The following extensions are automatically applied to apps created from the *Blazor Web App* project template during app creation. Individual components are still required to declare their render mode per the *Render modes* section after the component services and endpoints are configured in the app's `Program` file.

Services for Razor components are added by calling AddRazorComponents.

Component builder extensions:

- AddInteractiveServerComponents adds services to support rendering Interactive Server components.
- AddInteractiveWebAssemblyComponents adds services to support rendering Interactive WebAssembly components.

MapRazorComponents discovers available components and specifies the root component for the app (the first component loaded), which by default is the `App` component (`App.razor`).

Endpoint convention builder extensions:

- AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) for the app.
- AddInteractiveWebAssemblyRenderMode configures the Interactive WebAssembly render mode for the app.

> ⓘ **Note**

For orientation on the placement of the API in the following examples, inspect the `Program` file of an app generated from the Blazor Web App project template. For guidance on how to create a Blazor Web App, see Tooling for ASP.NET Core Blazor.

Example 1: The following `Program` file API adds services and configuration for enabling interactive SSR:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Example 2: The following `Program` file API adds services and configuration for enabling the Interactive WebAssembly render mode:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode();
```

Example 3: The following `Program` file API adds services and configuration for enabling the Interactive Server, Interactive WebAssembly, and Interactive Auto render modes:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents()
    .AddInteractiveWebAssemblyComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()
```

```
    .AddInteractiveWebAssemblyRenderMode();
```

Blazor uses the Blazor WebAssembly hosting model to download and execute components that use the Interactive WebAssembly render mode. A separate client project is required to set up Blazor WebAssembly hosting for these components. The client project contains the startup code for the Blazor WebAssembly host and sets up the .NET runtime for running in a browser. The Blazor Web App template adds this client project for you when you select the option to enable WebAssembly interactivity. Any components using the Interactive WebAssembly render mode should be built from the client project, so they get included in the downloaded app bundle.

Apply a render mode to a component instance

To apply a render mode to a component instance use the @rendermode Razor directive attribute where the component is used.

In the following example, interactive server-side rendering (interactive SSR) is applied to the `Dialog` component instance:

```razor
<Dialog @rendermode="InteractiveServer" />
```

> **Note**
>
> Blazor templates include a static `using` directive for RenderMode in the app's `_Imports` file (`Components/_Imports.razor`) for shorter `@rendermode` syntax:
>
> ```razor
> @using static Microsoft.AspNetCore.Components.Web.RenderMode
> ```
>
> Without the preceding directive, components must specify the static RenderMode class in `@rendermode` syntax:
>
> ```razor
> <Dialog @rendermode="RenderMode.InteractiveServer" />
> ```

You can also reference custom render mode instances instantiated directly with custom configuration. For more information, see the Custom shorthand render modes section later in this article.

Apply a render mode to a component definition

To specify the render mode for a component as part of its definition, use the @rendermode Razor directive and the corresponding render mode attribute.

razor

```
@page "..."
@rendermode InteractiveServer
```

Applying a render mode to a component definition is commonly used when applying a render mode to a specific page. Routable pages use the same render mode as the Router component that rendered the page.

Technically, `@rendermode` is both a Razor *directive* and a Razor *directive attribute*. The semantics are similar, but there are differences. The `@rendermode` directive is on the component definition, so the referenced render mode instance must be static. The `@rendermode` directive attribute can take any render mode instance.

> **Note**
>
> Component authors should avoid coupling a component's implementation to a specific render mode. Instead, component authors should typically design components to support any render mode or hosting model. A component's implementation should avoid assumptions on where it's running (server or client) and should degrade gracefully when rendered statically. Specifying the render mode in the component definition may be needed if the component isn't instantiated directly (such as with a routable page component) or to specify a render mode for all component instances.

Apply a render mode to the entire app

To set the render mode for the entire app, indicate the render mode at the highest-level interactive component in the app's component hierarchy that isn't a root component.

> **Note**
>
> Making a root component interactive, such as the `App` component, isn't supported. Therefore, the render mode for the entire app can't be set directly by the `App` component.

For apps based on the Blazor Web App project template, a render mode assigned to the entire app is typically specified where the `Routes` component is used in the `App` component (`Components/App.razor`):

```razor
<Routes @rendermode="InteractiveServer" />
```

The Router component propagates its render mode to the pages it routes.

You also typically must set the same interactive render mode on the HeadOutlet component, which is also found in the `App` component of a Blazor Web App generated from the project template:

```
<HeadOutlet @rendermode="InteractiveServer" />
```

For apps that adopt an interactive client-side (WebAssembly or Auto) rendering mode and enable the render mode for the entire app via the `Routes` component:

- Place or move the layout and navigation files of the server app's `Components/Layout` folder into the `.Client` project's `Layout` folder. Create a `Layout` folder in the `.Client` project if it doesn't exist.
- Place or move the components of the server app's `Components/Pages` folder into the `.Client` project's `Pages` folder. Create a `Pages` folder in the `.Client` project if it doesn't exist.
- Place or move the `Routes` component of the server app's `Components` folder into the `.Client` project's root folder.

To enable global interactivity when creating a Blazor Web App:

- Visual Studio: Set the **Interactivity location** dropdown list to **Global**.
- .NET CLI: Use the `-ai|--all-interactive` option.

For more information, see Tooling for ASP.NET Core Blazor.

Apply a render mode programatically

Properties and fields can assign a render mode.

The second approach described in this section, setting the render mode by component instance, is especially useful when your app specification calls for one or more components to adopt static SSR in a globally-interactive app. This scenario is covered in the Static SSR pages in a globally-interactive app section later in this article.

Set the render mode by component definition

A component definition can define a render mode via a private field:

razor

```
@rendermode pageRenderMode

...

@code {
    private static IComponentRenderMode pageRenderMode = InteractiveServer;
}
```

Set the render mode by component instance

The following example applies interactive server-side rendering (interactive SSR) to any request.

razor

```
<Routes @rendermode="PageRenderMode" />

...

@code {
    private IComponentRenderMode? PageRenderMode => InteractiveServer;
}
```

Additional information on render mode propagation is provided in the Render mode propagation section later in this article. The Static SSR pages in a globally-interactive app section shows how to use the preceding approach to adopt static SSR in a globally-interactive app.

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

The ComponentBase.RendererInfo and ComponentBase.AssignedRenderMode properties permit the app to detect details about the location, interactivity, and assigned render mode of a component:

- RendererInfo.Name returns the location where the component is executing:
  - `Static`: On the server (SSR) and incapable of interactivity.
  - `Server`: On the server (SSR) and capable of interactivity after prerendering.
  - `WebAssembly`: On the client (CSR) and capable of interactivity after prerendering.
  - `WebView`: On the native device and capable of interactivity after prerendering.
- RendererInfo.IsInteractive indicates if the component supports interactivity at the time of rendering. The value is `true` when rendering interactively or `false` when prerendering or for static SSR (RendererInfo.Name of `Static`).
- ComponentBase.AssignedRenderMode exposes the component's assigned render mode:
  - `InteractiveServer` for Interactive Server.
  - `InteractiveAuto` for Interactive Auto.
  - `InteractiveWebAssembly` for Interactive WebAssembly.

Components use these properties to render content depending on their location or interactivity status. For example, a form can be disabled during prerendering and enabled when the component becomes interactive:

razor

```
<EditForm Model="Movie" ...>
    <fieldset disabled="@disabled">

        ...

        <button type="submit" >Save</button>
    </fieldset>
</EditForm>

@code {
    private bool disabled = true;

    [SupplyParameterFromForm]
    private Movie? Movie { get; set; }

    protected override async Task OnInitializedAsync()
    {
        Movie ??= await ...;
```

```
        if (RendererInfo.IsInteractive)
        {
            disabled = false;
        }
    }
}
```

The next example shows how to render markup to support performing a regular HTML action if the component is statically rendered:

razor

```razor
@if (AssignedRenderMode is null)
{
    // The render mode is Static Server
    <form action="/movies">
        <input type="text" name="titleFilter" />
        <input type="submit" value="Search" />
    </form>
}
else
{
    // The render mode is Interactive Server, WebAssembly, or Auto
    <input @bind="titleFilter" />
    <button @onclick="FilterMovies">Search</button>
}
```

In the preceding example:

- When the value of AssignedRenderMode is `null`, the component adopts static SSR. Blazor event handling isn't functional in a browser with static SSR, so the component submits a form (GET request) with a `titleFilter` query string set to the user's `<input>` value. The `Movie` component (`/movie`) can read the query string and process the value of `titleFilter` to render the component with the filtered results.
- Otherwise, the render mode is any of `InteractiveServer`, `InteractiveWebAssembly`, or `InteractiveAuto`. The component is capable of using an event handler delegate (`FilterMovies`) and the value bound to the `<input>` element (`titleFilter`) to filter movies interactively over the background SignalR connection.

Blazor documentation examples for Blazor Web Apps

When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component adopts an interactive render mode.

Prerendering

Prerendering is the process of initially rendering page content on the server without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users. Prerendering can also improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines use to calculate page rank.

Prerendering is enabled by default for interactive components.

Internal navigation for interactive routing doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests, including for enhanced navigation. For more information, see Static versus interactive routing, Interactive routing and prerendering, and Enhanced navigation and form handling.

Disabling prerendering using the following techniques only takes effect for top-level render modes. If a parent component specifies a render mode, the prerendering settings of its children are ignored. This behavior is under investigation for possible changes with the release of .NET 10 in November, 2025.

To disable prerendering for a *component instance*, pass the `prerender` flag with a value of `false` to the render mode:

- `<... @rendermode="new InteractiveServerRenderMode(prerender: false)" />`
- `<... @rendermode="new InteractiveWebAssemblyRenderMode(prerender: false)" />`
- `<... @rendermode="new InteractiveAutoRenderMode(prerender: false)" />`

To disable prerendering in a *component definition*:

- `@rendermode @(new InteractiveServerRenderMode(prerender: false))`
- `@rendermode @(new InteractiveWebAssemblyRenderMode(prerender: false))`
- `@rendermode @(new InteractiveAutoRenderMode(prerender: false))`

To disable prerendering for the entire app, indicate the render mode at the highest-level interactive component in the app's component hierarchy that isn't a root component.

For apps based on the Blazor Web App project template, a render mode assigned to the entire app is specified where the `Routes` component is used in the `App` component (`Components/App.razor`). The following example sets the app's render mode to Interactive Server with prerendering disabled:

```razor
<Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

Also, disable prerendering for the HeadOutlet component in the `App` component:

```razor
<HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender: false)"
/>
```

Making a root component, such as the `App` component, interactive with the `@rendermode` directive at the top of the root component's definition file (`.razor`) isn't supported. Therefore, prerendering can't be disabled directly by the `App` component.

Static server-side rendering (static SSR)

Components use static server-side rendering (static SSR). The component renders to the response stream and interactivity isn't enabled.

In the following example, there's no designation for the component's render mode, so the component inherits its render mode from its parent. Since no ancestor component specifies a render mode, the following component is *statically rendered* on the server. The button isn't interactive and doesn't call the `UpdateMessage` method when selected. The value of `message` doesn't change, and the component isn't rerendered in response to UI events.

`RenderMode1.razor`:

```razor
@page "/render-mode-1"

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
```

```
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the server project's `Components/Pages` folder. The server project is the solution's project with a name that doesn't end in `.Client`. When the app is running, navigate to `/render-mode-1` in the browser's address bar.

During static SSR, Razor component page requests are processed by server-side ASP.NET Core middleware pipeline request processing for routing and authorization. Dedicated Blazor features for routing and authorization aren't operational because Razor components aren't rendered during server-side request processing. Blazor router features in the `Routes` component that aren't available during static SSR include displaying:

- Not authorized content (<NotAuthorized>...</NotAuthorized>) (NotAuthorized): Blazor Web Apps typically process unauthorized requests on the server by customizing the behavior of Authorization Middleware.
- Not found content (<NotFound>...</NotFound>) (NotFound): Blazor Web Apps typically process bad URL requests on the server by either displaying the browser's built-in 404 UI or returning a custom 404 page (or other response) via ASP.NET Core middleware (for example, UseStatusCodePagesWithRedirects / API documentation).

If the app exhibits root-level interactivity, server-side ASP.NET Core request processing isn't involved after the initial static SSR, which means that the preceding Blazor features work as expected.

Enhanced navigation with static SSR requires special attention when loading JavaScript. For more information, see ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR).

Interactive server-side rendering (interactive SSR)

Interactive server-side rendering (interactive SSR) renders the component interactively from the server using Blazor Server. User interactions are handled over a real-time connection with the browser. The circuit connection is established when the Server component is rendered.

In the following example, the render mode is set interactive SSR by adding `@rendermode InteractiveServer` to the component definition. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI.

`RenderMode2.razor`:

razor

```razor
@page "/render-mode-2"
@rendermode InteractiveServer

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component in a Blazor Web App, place the component in the server project's `Components/Pages` folder. The server project is the solution's project with a name that doesn't end in `.Client`. When the app is running, navigate to `/render-mode-2` in the browser's address bar.

Client-side rendering (CSR)

Client-side rendering (CSR) renders the component interactively on the client using Blazor WebAssembly. The .NET runtime and app bundle are downloaded and cached when the WebAssembly component is initially rendered. Components using CSR must be built from a separate client project that sets up the Blazor WebAssembly host.

In the following example, the render mode is set to CSR with `@rendermode InteractiveWebAssembly`. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI.

`RenderMode3.razor`:

razor

```
@page "/render-mode-3"
@rendermode InteractiveWebAssembly

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the client project's `Pages` folder. The client project is the solution's project with a name that ends in `.Client`. When the app is running, navigate to `/render-mode-3` in the browser's address bar.

Automatic (Auto) rendering

Automatic (Auto) rendering determines how to render the component at runtime. The component is initially rendered with interactive server-side rendering (interactive SSR) using the Blazor Server hosting model. The .NET runtime and app bundle are downloaded to the client in the background and cached so that they can be used on future visits.

The Auto render mode never dynamically changes the render mode of a component already on the page. The Auto render mode makes an initial decision about which type of interactivity to use for a component, then the component keeps that type of interactivity for as long as it's on the page. One factor in this initial decision is considering whether components already exist on the page with WebAssembly/Server interactivity. Auto mode prefers to select a render mode that matches the render mode of existing interactive components. The reason that the Auto mode prefers to use an existing interactivity mode is to avoid introducing a new interactive runtime that doesn't share state with the existing runtime.

Components using the Auto render mode must be built from a separate client project that sets up the Blazor WebAssembly host.

In the following example, the component is interactive throughout the process. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI. Initially, the

component is rendered interactively from the server, but on subsequent visits it's rendered from the client after the .NET runtime and app bundle are downloaded and cached.

`RenderMode4.razor`:

razor

```razor
@page "/render-mode-4"
@rendermode InteractiveAuto

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the client project's `Pages` folder. The client project is the solution's project with a name that ends in `.Client`. When the app is running, navigate to `/render-mode-4` in the browser's address bar.

Render mode propagation

Render modes propagate down the component hierarchy.

Rules for applying render modes:

- The default render mode is Static.
- The Interactive Server (InteractiveServer), Interactive WebAssembly (InteractiveWebAssembly), and Interactive Auto (InteractiveAuto) render modes can be used from a component, including using different render modes for sibling components.
- You can't switch to a different interactive render mode in a child component. For example, a Server component can't be a child of a WebAssembly component.
- Parameters passed to an interactive child component from a Static parent must be JSON serializable. This means that you can't pass render fragments or child content from a Static parent component to an interactive child component.

The following examples use a non-routable, non-page `SharedMessage` component. The render mode agnostic `SharedMessage` component doesn't apply a render mode with an @attribute directive. If you're testing these scenarios with a Blazor Web App, place the following component in the app's `Components` folder.

`SharedMessage.razor`:

razor

```razor
<p>@Greeting</p>

<button @onclick="UpdateMessage">Click me</button> @message

<p>@ChildContent</p>

@code {
    private string message = "Not updated yet.";

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public string Greeting { get; set; } = "Hello!";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

Render mode inheritance

If the `SharedMessage` component is placed in a statically-rendered parent component, the `SharedMessage` component is also rendered statically and isn't interactive. The button doesn't call `UpdateMessage`, and the message isn't updated.

`RenderMode5.razor`:

razor

```razor
@page "/render-mode-5"

<SharedMessage />
```

If the `SharedMessage` component is placed in a component that defines the render mode, it inherits the applied render mode.

In the following example, the `SharedMessage` component is interactive over a SignalR connection to the client. The button calls `UpdateMessage`, and the message is updated.

`RenderMode6.razor`:

```razor
@page "/render-mode-6"
@rendermode InteractiveServer

<SharedMessage />
```

Child components with different render modes

In the following example, both `SharedMessage` components are prerendered and appear when the page is displayed in the browser.

- The first `SharedMessage` component with interactive server-side rendering (interactive SSR) is interactive after the SignalR circuit is established.
- The second `SharedMessage` component with client-side rendering (CSR) is interactive *after* the Blazor app bundle is downloaded and the .NET runtime is active on the client.

`RenderMode7.razor`:

```razor
@page "/render-mode-7"

<SharedMessage @rendermode="InteractiveServer" />
<SharedMessage @rendermode="InteractiveWebAssembly" />
```

Child component with a serializable parameter

The following example demonstrates an interactive child component that takes a parameter. Parameters must be serializable.

`RenderMode8.razor`:

```razor
@page "/render-mode-8"
```

```razor
<SharedMessage @rendermode="InteractiveServer" Greeting="Welcome!" />
```

Non-serializable component parameters, such as child content or a render fragment, aren't supported. In the following example, passing child content to the `SharedMessage` component results in a runtime error.

`RenderMode9.razor`:

```razor
@page "/render-mode-9"

<SharedMessage @rendermode="InteractiveServer">
    Child content
</SharedMessage>
```

❌ **Error**:

> System.InvalidOperationException: Cannot pass the parameter 'ChildContent' to component 'SharedMessage' with rendermode 'InteractiveServerRenderMode'. This is because the parameter is of the delegate type 'Microsoft.AspNetCore.Components.RenderFragment', which is arbitrary code and cannot be serialized.

To circumvent the preceding limitation, wrap the child component in another component that doesn't have the parameter. This is the approach taken in the Blazor Web App project template with the `Routes` component (`Components/Routes.razor`) to wrap the Router component.

`WrapperComponent.razor`:

```razor
<SharedMessage>
    Child content
</SharedMessage>
```

`RenderMode10.razor`:

```razor
@page "/render-mode-10"

<WrapperComponent @rendermode="InteractiveServer" />
```

In the preceding example:

- The child content is passed to the `SharedMessage` component without generating a runtime error.
- The `SharedMessage` component renders interactively on the server.

Child component with a different render mode than its parent

Don't try to apply a different interactive render mode to a child component than its parent's render mode.

The following component results in a runtime error when the component is rendered:

`RenderMode11.razor`:

razor

```razor
@page "/render-mode-11"
@rendermode InteractiveServer

<SharedMessage @rendermode="InteractiveWebAssembly" />
```

❌ **Error**:

> Cannot create a component of type 'BlazorSample.Components.SharedMessage' because its render mode 'Microsoft.AspNetCore.Components.Web.InteractiveWebAssemblyRenderMode' is not supported by Interactive Server rendering.

Static SSR pages in a globally-interactive app

There are cases where the app's specification calls for components to adopt static server-side rendering (static SSR) and only run on the server, while the rest of the app uses an interactive render mode.

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 [ExcludeFromInteractiveRouting] attribute assigned with the `@attribute` Razor directive:

razor

```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

```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`.

Client-side services fail to resolve during prerendering

Assuming that prerendering isn't disabled for a component or for the app, a component in the `.Client` project is prerendered on the server. Because the server doesn't have access to registered client-side Blazor services, it isn't possible to inject these services into a component without receiving an error that the service can't be found during prerendering.

For example, consider the following `Home` component in the `.Client` project in a Blazor Web App with global Interactive WebAssembly or Interactive Auto rendering. The component attempts to inject IWebAssemblyHostEnvironment to obtain the environment's name.

razor

```
@page "/"
@inject IWebAssemblyHostEnvironment Environment

<PageTitle>Home</PageTitle>

<h1>Home</h1>

<p>
    Environment: @Environment.Environment
</p>
```

No compile time error occurs, but a runtime error occurs during prerendering:

> Cannot provide a value for property 'Environment' on type 'BlazorSample.Client.Pages.Home'. There is no registered service of type 'Microsoft.AspNetCore.Components.WebAssembly.Hosting.IWebAssemblyHostEnvironment'.

This error occurs because the component must compile and execute on the server during prerendering, but IWebAssemblyHostEnvironment isn't a registered service on the server.

If the app doesn't require the value during prerendering, this problem can be solved by injecting IServiceProvider to obtain the service instead of the service type itself:

razor

```
@page "/"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IServiceProvider Services

<PageTitle>Home</PageTitle>

<h1>Home</h1>

<p>
    <b>Environment:</b> @environmentName
</p>

@code {
    private string? environmentName;

    protected override void OnInitialized()
    {
        if (Services.GetService<IWebAssemblyHostEnvironment>() is { } env)
        {
            environmentName = env.Environment;
        }
    }
}
```

However, the preceding approach isn't useful if your logic requires a value during prerendering.

You can also avoid the problem if you disable prerendering for the component, but that's an extreme measure to take in many cases that may not meet your component's specifications.

There are a three approaches that you can take to address this scenario. The following are listed from most recommended to least recommended:

- *Recommended* for shared framework services: For shared framework services that merely aren't registered server-side in the main project, register the services in the main project, which makes them available during prerendering. For an example of this scenario, see the guidance for HttpClient services in Call a web API from an ASP.NET Core Blazor app.

- *Recommended* for services outside of the shared framework: Create a custom service implementation for the service on the server. Use the service normally in interactive components of the `.Client` project. For a demonstration of this approach, see ASP.NET Core Blazor environments.
- Create a service abstraction and create implementations for the service in the `.Client` and server projects. Register the services in each project. Inject the custom service in the component.
- You might be able to add a `.Client` project package reference to a server-side package and fall back to using the server-side API when prerendering on the server.

Discover components from additional assemblies

Additional assemblies must be disclosed to the Blazor framework to discover routable Razor components in referenced projects. For more information, see ASP.NET Core Blazor routing and navigation.

Closure of 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 created 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.

Custom shorthand render modes

The `@rendermode` directive takes a single parameter that's a static instance of type IComponentRenderMode. The `@rendermode` directive attribute can take any render mode instance, static or not. The Blazor framework provides the RenderMode static class with some predefined render modes for convenience, but you can create your own.

Normally, a component uses the following `@rendermode` directive to disable prerendering:

```
razor
```

```
@rendermode @(new InteractiveServerRenderMode(prerender: false))
```

However, consider the following example that creates a shorthand interactive server-side render mode without prerendering via the app's `_Imports` file (`Components/_Imports.razor`):

C#

```
public static IComponentRenderMode InteractiveServerWithoutPrerendering {
get; } =
    new InteractiveServerRenderMode(prerender: false);
```

Use the shorthand render mode in components throughout the `Components` folder:

razor

```
@rendermode InteractiveServerWithoutPrerendering
```

Alternatively, a single component instance can define a custom render mode via a private field:

razor

```
@rendermode interactiveServerWithoutPrerendering

...

@code {
    private static IComponentRenderMode interactiveServerWithoutPrerendering
=
        new InteractiveServerRenderMode(prerender: false);
}
```

At the moment, the shorthand render mode approach is probably only useful for reducing the verbosity of specifying the `prerender` flag. The shorthand approach might be more useful in the future if additional flags become available for interactive rendering and you would like to create shorthand render modes with different combinations of flags.

Service injection via a top-level imports file (`_Imports.razor`)

This section only applies to Blazor Web Apps.

A top-level imports file in the `Components` folder (`Components/_Imports.razor`) injects its references into all of the components in the folder hierarchy, which includes the `App` component (`App.razor`). The `App` component is always rendered statically even if prerendering of a page component is disabled. Therefore, injecting services via the top-level imports file results in resolving *two instances* of the service in page components.

To address this scenario, inject the service in a new imports file placed in the `Pages` folder (`Components/Pages/_Imports.razor`). From that location, the service is only resolved once in page components.

Additional resources

- WebSocket compression
  - ASP.NET Core Blazor SignalR guidance
  - Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering
- ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)
- Cascading values/parameters and render mode boundaries: Also see the Root-level cascading parameters section earlier in the article.
- ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR)

Prerender ASP.NET Core Razor components

Article • 11/14/2024

> **ⓘ 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 article explains Razor component prerendering scenarios for server-rendered components in Blazor Web Apps.

Prerendering is the process of initially rendering page content on the server without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users. Prerendering can also improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines use to calculate page rank.

Persist prerendered state

Without persisting prerendered state, state used during prerendering is lost and must be recreated when the app is fully loaded. If any state is created asynchronously, the UI may flicker as the prerendered UI is replaced when the component is rerendered.

Consider the following `PrerenderedCounter1` counter component. The component sets an initial random counter value during prerendering in OnInitialized lifecycle method. After the SignalR connection to the client is established, the component rerenders, and the initial count value is replaced when `OnInitialized` executes a second time.

`PrerenderedCounter1.razor`:

razor

```razor
@page "/prerendered-counter-1"
@rendermode @(new InteractiveServerRenderMode(prerender: true))
@inject ILogger<PrerenderedCounter1> Logger
```

```razor
<PageTitle>Prerendered Counter 1</PageTitle>

<h1>Prerendered Counter 1</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount;

    protected override void OnInitialized()
    {
        currentCount = Random.Shared.Next(100);
        Logger.LogInformation("currentCount set to {Count}", currentCount);
    }

    private void IncrementCount() => currentCount++;
}
```

Run the app and inspect logging from the component. The following is example output.

> **Note**
>
> If the app adopts interactive (enhanced) routing and the page is reached via an internal navigation, prerendering doesn't occur. Therefore, you must perform a full page reload for the `PrerenderedCounter1` component to see the following output.

```
info: BlazorSample.Components.Pages.PrerenderedCounter1[0]
currentCount set to 41
info: BlazorSample.Components.Pages.PrerenderedCounter1[0]
currentCount set to 92
```

The first logged count occurs during prerendering. The count is set again after prerendering when the component is rerendered. There's also a flicker in the UI when the count updates from 41 to 92.

To retain the initial value of the counter during prerendering, Blazor supports persisting state in a prerendered page using the PersistentComponentState service (and for components embedded into pages or views of Razor Pages or MVC apps, the Persist Component State Tag Helper).

To preserve prerendered state, decide what state to persist using the PersistentComponentState service. PersistentComponentState.RegisterOnPersisting

registers a callback to persist the component state before the app is paused. The state is retrieved when the app resumes.

The following example demonstrates the general pattern:

- The `{TYPE}` placeholder represents the type of data to persist.
- The `{TOKEN}` placeholder is a state identifier string. Consider using `nameof({VARIABLE})`, where the `{VARIABLE}` placeholder is the name of the variable that holds the state. Using nameof() for the state identifier avoids the use of a quoted string.

razor

```
@implements IDisposable
@inject PersistentComponentState ApplicationState

...

@code {
    private {TYPE} data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<{TYPE}>(
            "{TOKEN}", out var restored))
        {
            data = await ...;
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("{TOKEN}", data);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

The following counter component example persists counter state during prerendering and retrieves the state to initialize the component.

`PrerenderedCounter2.razor`:

razor

```razor
@page "/prerendered-counter-2"
@implements IDisposable
@inject ILogger<PrerenderedCounter2> Logger
@inject PersistentComponentState ApplicationState

<PageTitle>Prerendered Counter 2</PageTitle>

<h1>Prerendered Counter 2</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override void OnInitialized()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistCount);

        if (!ApplicationState.TryTakeFromJson<int>(
            nameof(currentCount), out var restoredCount))
        {
            currentCount = Random.Shared.Next(100);
            Logger.LogInformation("currentCount set to {Count}",
currentCount);
        }
        else
        {
            currentCount = restoredCount!;
            Logger.LogInformation("currentCount restored to {Count}",
currentCount);
        }
    }

    private Task PersistCount()
    {
        ApplicationState.PersistAsJson(nameof(currentCount), currentCount);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose() => persistingSubscription.Dispose();
```

```
    private void IncrementCount() => currentCount++;
}
```

When the component executes, `currentCount` is only set once during prerendering. The value is restored when the component is rerendered. The following is example output.

> **Note**
>
> If the app adopts interactive routing and the page is reached via an internal navigation, prerendering doesn't occur. Therefore, you must perform a full page reload for the `PrerenderedCounter2` component to see the following output.

```
info: BlazorSample.Components.Pages.PrerenderedCounter2[0]
currentCount set to 96
info: BlazorSample.Components.Pages.PrerenderedCounter2[0]
currentCount restored to 96
```

By initializing components with the same state used during prerendering, any expensive initialization steps are only executed once. The rendered UI also matches the prerendered UI, so no flicker occurs in the browser.

The persisted prerendered state is transferred to the client, where it's used to restore the component state. During client-side rendering (CSR, `InteractiveWebAssembly`), the data is exposed to the browser and must not contain sensitive, private information. During interactive server-side rendering (interactive SSR, `InteractiveServer`), ASP.NET Core Data Protection ensures that the data is transferred securely. The `InteractiveAuto` render mode combines WebAssembly and Server interactivity, so it's necessary to consider data exposure to the browser, as in the CSR case.

Components embedded into pages and views (Razor Pages/MVC)

For components embedded into a page or view of a Razor Pages or MVC app, you must add the Persist Component State Tag Helper with the `<persist-component-state />` HTML tag inside the closing `</body>` tag of the app's layout. **This is only required for Razor Pages and MVC apps.** For more information, see Persist Component State Tag Helper in ASP.NET Core.

`Pages/Shared/_Layout.cshtml`:

CSHTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Interactive routing and prerendering

Internal navigation for interactive routing doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests.

The PersistentComponentState service only works on the initial page load and not across enhanced page navigation events. If the app performs a full (non-enhanced) navigation to a page utilizing persistent component state, the persisted state is made available for the app to use when it becomes interactive. But if an interactive circuit has already been established and an enhanced navigation is performed to a page that renders persisted component state, that state isn't made available in the existing circuit. The PersistentComponentState service isn't aware of enhanced navigation, and there's no mechanism to deliver state updates to components that are already running.

Prerendering guidance

Prerendering guidance is organized in the Blazor documentation by subject matter. The following links cover all of the prerendering guidance throughout the documentation set by subject:

- Fundamentals
  - OnNavigateAsync is executed *twice* when prerendering: Handle asynchronous navigation events with OnNavigateAsync
  - Startup: Control headers in C# code
  - Handle Errors: Prerendering
  - SignalR: Prerendered state size and SignalR message size limit
- Render modes: Prerendering
- Components
  - Control <head> content during prerendering
  - Razor component lifecycle subjects that pertain to prerendering
    - Component initialization (OnInitialized{Async})
    - After component render (OnAfterRender{Async})

    - Stateful reconnection after prerendering
    - Prerendering with JavaScript interop: This section also appears in the two JS interop articles on calling JavaScript from .NET and calling .NET from JavaScript.
  - QuickGrid component sample app: The **QuickGrid for Blazor** sample app is hosted on GitHub Pages. The site loads fast thanks to static prerendering using the community-maintained BlazorWasmPrerendering.Build GitHub project.
  - Prerendering when integrating components into Razor Pages and MVC apps

- Authentication and authorization
  - Server-side threat mitigation: Cross-site scripting (XSS)
  - Server-side unauthorized content display while prerendering with a custom AuthenticationStateProvider
  - Blazor WebAssembly rendered component authentication with prerendering

- State management: Handle prerendering: Besides the *Handle prerendering* section, several of the article's other sections include remarks on prerendering.

ASP.NET Core Razor component generic type support

Article • 10/18/2024

> **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 article describes generic type support in Razor components.

If you're new to generic types, see Generic classes and methods (C# Guide) for general guidance on the use of generics before reading this article.

The example code in this article is only available for the latest .NET release in the Blazor sample apps.

Generic type parameter support

The @typeparam directive declares a generic type parameter for the generated component class:

```razor
@typeparam TItem
```

C# syntax with where type constraints is supported:

```razor
@typeparam TEntity where TEntity : IEntity
```

A RenderFragment generic type is supported, but components aren't supported as the generic type. To render components by type, consider using a DynamicComponent. For more information, see Dynamically-rendered ASP.NET Core Razor components.

In the following example, the `ListItems1` component is generically typed as `TExample`, which represents the type of the `ExampleList` collection.

`ListItems1.razor`:

razor

```
@typeparam TExample

<h2>List Items 1</h2>

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }
}
```

The following component renders two `ListItems1` components:

- String or integer data is assigned to the `ExampleList` parameter of each component.
- Type `string` or `int` that matches the type of the assigned data is set for the type parameter (`TExample`) of each component.

`Generics1.razor`:

razor

```
@page "/generics-1"

<PageTitle>Generics 1</PageTitle>

<h1>Generic Type Example 1</h1>
```

```
<ListItems1 Color="blue"
            ExampleList="@(new List<string> { "Item 1", "Item 2" })"
            TExample="string" />

<ListItems1 Color="red"
            ExampleList="@(new List<int> { 1, 2 })"
            TExample="int" />
```

For more information, see Razor syntax reference for ASP.NET Core. For an example of generic typing with templated components, see ASP.NET Core Blazor templated components.

Cascaded generic type support

An ancestor component can cascade a type parameter by name to descendants using the [CascadingTypeParameter] attribute. This attribute allows a generic type inference to use the specified type parameter automatically with descendants that have a type parameter with the same name.

By adding `@attribute [CascadingTypeParameter(...)]` to a component, the specified generic type argument is automatically used by descendants that:

- Are nested as child content for the component in the same `.razor` document.
- Also declare a @typeparam with the exact same name.
- Don't have another value explicitly supplied or implicitly inferred for the type parameter. If another value is supplied or inferred, it takes precedence over the cascaded generic type.

When receiving a cascaded type parameter, components obtain the parameter value from the closest ancestor that has a [CascadingTypeParameter] attribute with a matching name. Cascaded generic type parameters are overridden within a particular subtree.

Matching is only performed by name. Therefore, we recommend avoiding a cascaded generic type parameter with a generic name, for example `T` or `TItem`. If a developer opts into cascading a type parameter, they're implicitly promising that its name is unique enough not to clash with other cascaded type parameters from unrelated components.

Generic types can be cascaded to child components with either of the following approaches for ancestor (parent) components, which are demonstrated in the following two sub-sections:

- Explicitly set the cascaded generic type.

- Infer the cascaded generic type.

The following subsections provide examples of the preceding approaches using the following `ListDisplay1` component. The component receives and renders list data generically typed as `TExample`. To make each instance of `ListDisplay1` stand out, an additional component parameter controls the color of the list.

`ListDisplay1.razor`:

razor

```razor
@typeparam TExample

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }
}
```

Explicit generic types based on ancestor components

The demonstration in this section cascades a type explicitly for `TExample`.

> ⓘ Note
>
> This section uses the preceding `ListDisplay1` component in the Cascaded generic type support section.

The following `ListItems2` component receives data and cascades a generic type parameter named `TExample` to its descendent components. In the upcoming parent component, the `ListItems2` component is used to display list data with the preceding `ListDisplay1` component.

`ListItems2.razor`:

```razor
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 2</h2>

@ChildContent

<p>
    Type of <code>TExample</code>: @typeof(TExample)
</p>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

The following parent component sets the child content (RenderFragment) of two `ListItems2` components specifying the `ListItems2` types (`TExample`), which are cascaded to child components. `ListDisplay1` components are rendered with the list item data shown in the example. String data is used with the first `ListItems2` component, and integer data is used with the second `ListItems2` component.

`Generics2.razor`:

```razor
@page "/generics-2"

<PageTitle>Generics 2</PageTitle>

<h1>Generic Type Example 2</h1>

<ListItems2 TExample="string">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<string> { "Item 1", "Item 2" })"
/>
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<string> { "Item 3", "Item 4" })"
/>
</ListItems2>

<ListItems2 TExample="int">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<int> { 1, 2 })" />
    <ListDisplay1 Color="red"
```

```
                    ExampleList="@(new List<int> { 3, 4 })" />
</ListItems2>
```

Specifying the type explicitly also allows the use of cascading values and parameters to provide data to child components, as the following demonstration shows.

`ListDisplay2.razor`:

razor

```razor
@typeparam TExample

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [CascadingParameter]
    protected IEnumerable<TExample>? ExampleList { get; set; }
}
```

`ListItems3.razor`:

razor

```razor
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 3</h2>

@ChildContent

@if (ExampleList is not null)
{
    <ul style="color:green">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
```

```
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}

@code {
    [CascadingParameter]
    protected IEnumerable<TExample>? ExampleList { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

When cascading the data in the following example, the type must be provided to the component.

`Generics3.razor`:

razor

```
@page "/generics-3"

<PageTitle>Generics 3</PageTitle>

<h1>Generic Type Example 3</h1>

<CascadingValue Value="stringData">
    <ListItems3 TExample="string">
        <ListDisplay2 Color="blue" />
        <ListDisplay2 Color="red" />
    </ListItems3>
</CascadingValue>

<CascadingValue Value="integerData">
    <ListItems3 TExample="int">
        <ListDisplay2 Color="blue" />
        <ListDisplay2 Color="red" />
    </ListItems3>
</CascadingValue>

@code {
    private List<string> stringData = new() { "Item 1", "Item 2" };
    private List<int> integerData = new() { 1, 2 };
}
```

When multiple generic types are cascaded, values for all generic types in the set must be passed. In the following example, `TItem`, `TValue`, and `TEdit` are `GridColumn` generic types, but the parent component that places `GridColumn` doesn't specify the `TItem` type:

razor

```
<GridColumn TValue="string" TEdit="TextEdit" />
```

The preceding example generates a compile-time error that the `GridColumn` component is missing the `TItem` type parameter. Valid code specifies all of the types:

razor

```
<GridColumn TValue="string" TEdit="TextEdit" TItem="User" />
```

Infer generic types based on ancestor components

The demonstration in this section cascades a type inferred for `TExample`.

> **Note**
>
> This section uses the `ListDisplay` component in the Cascaded generic type support section.

`ListItems4.razor`:

razor

```
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 4</h2>

@ChildContent

@if (ExampleList is not null)
{
    <ul style="color:green">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}

@code {
    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }
```

```
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

The following component with inferred cascaded types provides different data for display.

`Generics4.razor`:

razor

```
@page "/generics-4"

<PageTitle>Generics 4</PageTitle>

<h1>Generic Type Example 4</h1>

<ListItems4 ExampleList="@(new List<string> { "Item 5", "Item 6" })">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<string> { "Item 1", "Item 2" })"
/>
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<string> { "Item 3", "Item 4" })"
/>
</ListItems4>

<ListItems4 ExampleList="@(new List<int> { 5, 6 })">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<int> { 1, 2 })" />
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<int> { 3, 4 })" />
</ListItems4>
```

The following component with inferred cascaded types provides the same data for display. The following example directly assigns the data to the components.

`Generics5.razor`:

razor

```
@page "/generics-5"

<PageTitle>Generics 5</PageTitle>

<h1>Generic Type Example 5</h1>

<ListItems4 ExampleList="stringData">
    <ListDisplay1 Color="blue" ExampleList="stringData" />
    <ListDisplay1 Color="red" ExampleList="stringData" />
```

```
</ListItems4>

<ListItems4 ExampleList="integerData">
    <ListDisplay1 Color="blue" ExampleList="integerData" />
    <ListDisplay1 Color="red" ExampleList="integerData" />
</ListItems4>

@code {
    private List<string> stringData = new() { "Item 1", "Item 2" };
    private List<int> integerData = new() { 1, 2 };
}
```

ASP.NET Core Blazor synchronization context

Article • 10/18/2024

> **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.

Blazor uses a synchronization context (SynchronizationContext) to enforce a single logical thread of execution. A component's lifecycle methods and event callbacks raised by Blazor are executed on the synchronization context.

Blazor's server-side synchronization context attempts to emulate a single-threaded environment so that it closely matches the WebAssembly model in the browser, which is single threaded. This emulation is scoped only to an individual circuit, meaning two different circuits can run in parallel. At any given point in time within a circuit, work is performed on exactly one thread, which yields the impression of a single logical thread. No two operations execute concurrently within the same circuit.

Avoid thread-blocking calls

Generally, don't call the following methods in components. The following methods block the execution thread and thus block the app from resuming work until the underlying Task is complete:

- Result
- Wait
- WaitAny
- WaitAll
- Sleep
- GetResult

> **Note**

Blazor documentation examples that use the thread-blocking methods mentioned in this section are only using the methods for demonstration purposes, not as recommended coding guidance. For example, a few component code demonstrations simulate a long-running process by calling Thread.Sleep.

Invoke component methods externally to update state

In the event a component must be updated based on an external event, such as a timer or other notification, use the `InvokeAsync` method, which dispatches code execution to Blazor's synchronization context. For example, consider the following *notifier service* that can notify any listening component about updated state. The `Update` method can be called from anywhere in the app.

`TimerService.cs`:

C#

```
namespace BlazorSample;

public class TimerService(NotifierService notifier,
    ILogger<TimerService> logger) : IDisposable
{
    private int elapsedCount;
    private readonly static TimeSpan heartbeatTickRate =
TimeSpan.FromSeconds(5);
    private readonly ILogger<TimerService> logger = logger;
    private readonly NotifierService notifier = notifier;
    private PeriodicTimer? timer;

    public async Task Start()
    {
        if (timer is null)
        {
            timer = new(heartbeatTickRate);
            logger.LogInformation("Started");

            using (timer)
            {
                while (await timer.WaitForNextTickAsync())
                {
                    elapsedCount += 1;
                    await notifier.Update("elapsedCount", elapsedCount);
                    logger.LogInformation("ElapsedCount {Count}",
elapsedCount);
                }
            }
```

```
        }
    }

    public void Dispose()
    {
        timer?.Dispose();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }
}
```

`NotifierService.cs`:

C#

```
namespace BlazorSample;

public class NotifierService
{
    public async Task Update(string key, int value)
    {
        if (Notify != null)
        {
            await Notify.Invoke(key, value);
        }
    }

    public event Func<string, int, Task>? Notify;
}
```

Register the services:

- For client-side development, register the services as singletons in the client-side `Program` file:

  C#

  ```
  builder.Services.AddSingleton<NotifierService>();
  builder.Services.AddSingleton<TimerService>();
  ```

- For server-side development, register the services as scoped in the server `Program` file:

  C#

  ```
  builder.Services.AddScoped<NotifierService>();
  ```

```
builder.Services.AddScoped<TimerService>();
```

Use the `NotifierService` to update a component.

`Notifications.razor`:

razor

```razor
@page "/notifications"
@implements IDisposable
@inject NotifierService Notifier
@inject TimerService Timer

<PageTitle>Notifications</PageTitle>

<h1>Notifications Example</h1>

<h2>Timer Service</h2>

<button @onclick="StartTimer">Start Timer</button>

<h2>Notifications</h2>

<p>
    Status:
    @if (lastNotification.key is not null)
    {
        <span>@lastNotification.key = @lastNotification.value</span>
    }
    else
    {
        <span>Awaiting notification</span>
    }
</p>

@code {
    private (string key, int value) lastNotification;

    protected override void OnInitialized() => Notifier.Notify += OnNotify;

    public async Task OnNotify(string key, int value)
    {
        await InvokeAsync(() =>
        {
            lastNotification = (key, value);
            StateHasChanged();
        });
    }

    private void StartTimer() => _ = Task.Run(Timer.Start);
```

```
    public void Dispose() => Notifier.Notify -= OnNotify;
}
```

In the preceding example:

- The timer is initiated outside of Blazor's synchronization context with `_ = Task.Run(Timer.Start)`.
- `NotifierService` invokes the component's `OnNotify` method. `InvokeAsync` is used to switch to the correct context and enqueue a rerender. For more information, see ASP.NET Core Razor component rendering.
- The component implements IDisposable. The `OnNotify` delegate is unsubscribed in the `Dispose` method, which is called by the framework when the component is disposed. For more information, see ASP.NET Core Razor component lifecycle.

> **Important**
>
> If a Razor component defines an event that's triggered from a background thread, the component might be required to capture and restore the execution context (ExecutionContext) at the time the handler is registered. For more information, see Calling InvokeAsync(StateHasChanged) causes page to fallback to default culture (dotnet/aspnetcore #28521).

To dispatch caught exceptions from the background `TimerService` to the component to treat the exceptions like normal lifecycle event exceptions, see the Handle caught exceptions outside of a Razor component's lifecycle section.

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 the exceptions.

> **Note**

ComponentBase.DispatchExceptionAsync is used in Razor component files (`.razor`) that inherit from ComponentBase. When creating components that implement IComponent directly, use RenderHandle.DispatchExceptionAsync.

To handle caught exceptions outside of a Razor component's lifecycle, pass the exception to DispatchExceptionAsync and await the result:

C#

```
try
{
    ...
}
catch (Exception ex)
{
    await DispatchExceptionAsync(ex);
}
```

A common scenario for the preceding approach is when a component starts an asynchronous operation but doesn't await a Task, often called the *fire and forget* pattern because the method is *fired* (started) and the result of the method is *forgotten* (thrown away). If the operation fails, you may want the component to treat the failure as a component lifecycle exception for any of the following goals:

- Put the component into a faulted state, for example, to trigger an error boundary.
- Terminate the circuit if there's no error boundary.
- Trigger the same logging that occurs for lifecycle exceptions.

In the following example, the user selects the **Send report** button to trigger a background method, `ReportSender.SendAsync`, that sends a report. In most cases, a component awaits the Task of an asynchronous call and updates the UI to indicate the operation completed. In the following example, the `SendReport` method doesn't await a Task and doesn't report the result to the user. Because the component intentionally discards the Task in `SendReport`, any asynchronous failures occur off of the normal lifecycle call stack, hence aren't seen by Blazor:

razor

```
<button @onclick="SendReport">Send report</button>

@code {
    private void SendReport()
    {
        _ = ReportSender.SendAsync();
```

```
    }
}
```

To treat failures like lifecycle method exceptions, explicitly dispatch exceptions back to the component with DispatchExceptionAsync, as the following example demonstrates:

razor

```
<button @onclick="SendReport">Send report</button>

@code {
    private void SendReport()
    {
        _ = SendReportAsync();
    }

    private async Task SendReportAsync()
    {
        try
        {
            await ReportSender.SendAsync();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    }
}
```

An alternative approach leverages Task.Run:

C#

```
private void SendReport()
{
    _ = Task.Run(async () =>
    {
        try
        {
            await ReportSender.SendAsync();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    });
}
```

For a working demonstration, implement the timer notification example in Invoke component methods externally to update state. In a Blazor app, add the following files

from the timer notification example and register the services in the `Program` file as the section explains:

- `TimerService.cs`
- `NotifierService.cs`
- `Notifications.razor`

The example uses a timer outside of a Razor component's lifecycle, where an unhandled exception normally isn't processed by Blazor's error handling mechanisms, such as an error boundary.

First, change the code in `TimerService.cs` to create an artificial exception outside of the component's lifecycle. In the `while` loop of `TimerService.cs`, throw an exception when the `elapsedCount` reaches a value of two:

C#

```
if (elapsedCount == 2)
{
    throw new Exception("I threw an exception! Somebody help me!");
}
```

Place an error boundary in the app's main layout. Replace the `<article>...</article>` markup with the following markup.

In `MainLayout.razor`:

razor

```
<article class="content px-4">
    <ErrorBoundary>
        <ChildContent>
            @Body
        </ChildContent>
        <ErrorContent>
            <p class="alert alert-danger" role="alert">
                Oh, dear! Oh, my! - George Takei
            </p>
        </ErrorContent>
    </ErrorBoundary>
</article>
```

In Blazor Web Apps with the error boundary only applied to a static `MainLayout` component, the boundary is only active during the static server-side rendering (static SSR) phase. The boundary doesn't activate just because a component further down the component hierarchy is interactive. To enable interactivity broadly for the `MainLayout`

component and the rest of the components further down the component hierarchy, enable interactive rendering for the `HeadOutlet` and `Routes` component instances in the `App` component (`Components/App.razor`). The following example adopts the Interactive Server (`InteractiveServer`) render mode:

razor

```razor
<HeadOutlet @rendermode="InteractiveServer" />

...

<Routes @rendermode="InteractiveServer" />
```

If you run the app at this point, the exception is thrown when the elapsed count reaches a value of two. However, the UI doesn't change. The error boundary doesn't show the error content.

To dispatch exceptions from the timer service back to the `Notifications` component, the following changes are made to the component:

- Start the timer in a try-catch statement. In the `catch` clause of the `try-catch` block, exceptions are dispatched back to the component by passing the Exception to DispatchExceptionAsync and awaiting the result.
- In the `StartTimer` method, start the asynchronous timer service in the Action delegate of Task.Run and intentionally discard the returned Task.

The `StartTimer` method of the `Notifications` component (`Notifications.razor`):

C#

```
private void StartTimer()
{
    _ = Task.Run(async () =>
    {
        try
        {
            await Timer.Start();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    });
}
```

When the timer service executes and reaches a count of two, the exception is dispatched to the Razor component, which in turn triggers the error boundary to display the error content of the `<ErrorBoundary>` in the `MainLayout` component:

> Oh, dear! Oh, my! - George Takei

Retain element, component, and model relationships in ASP.NET Core Blazor

Article • 10/18/2024

> **ⓘ 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 article explains how to use the `@key` directive attribute to retain element, component, and model relationships when rendering and the elements or components subsequently change.

Use of the `@key` directive attribute

When rendering a list of elements or components and the elements or components subsequently change, Blazor must decide which of the previous elements or components are retained and how model objects should map to them. Normally, this process is automatic and sufficient for general rendering, but there are often cases where controlling the process using the @key directive attribute is required.

Consider the following example that demonstrates a collection mapping problem that's solved by using @key.

For the following components:

- The `Details` component receives data (`Data`) from the parent component, which is displayed in an `<input>` element. Any given displayed `<input>` element can receive the focus of the page from the user when they select one of the `<input>` elements.
- The parent component creates a list of person objects for display using the `Details` component. Every three seconds, a new person is added to the collection.

This demonstration allows you to:

- Select an `<input>` from among several rendered `Details` components.
- Study the behavior of the page's focus as the people collection automatically grows.

`Details.razor`:

```razor
<input value="@Data" />

@code {
    [Parameter]
    public string? Data { get; set; }
}
```

In the following parent component, each iteration of adding a person in `OnTimerCallback` results in Blazor rebuilding the entire collection. The page's focus remains on the *same index* position of `<input>` elements, so the focus shifts each time a person is added. *Shifting the focus away from what the user selected isn't desirable behavior.* After demonstrating the poor behavior with the following component, the @key directive attribute is used to improve the user's experience.

`People.razor`:

```razor
@page "/people"
@using System.Timers
@implements IDisposable

<PageTitle>People</PageTitle>

<h1>People Example</h1>

@foreach (var person in people)
{
    <Details Data="@person.Data" />
}

@code {
    private Timer timer = new Timer(3000);

    public List<Person> people =
        new()
        {
            { new Person { Data = "Person 1" } },
            { new Person { Data = "Person 2" } },
            { new Person { Data = "Person 3" } }
        };

    protected override void OnInitialized()
    {
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
```

```
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            people.Insert(0,
                new Person
                {
                    Data = $"INSERTED {DateTime.Now.ToString("hh:mm:ss
tt")}"
                });
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();

    public class Person
    {
        public string? Data { get; set; }
    }
}
```

The contents of the `people` collection changes with inserted, deleted, or re-ordered entries. Rerendering can lead to visible behavior differences. For example, each time a person is inserted into the `people` collection, the user's focus is lost.

The mapping process of elements or components to a collection can be controlled with the @key directive attribute. Use of @key guarantees the preservation of elements or components based on the key's value. If the `Details` component in the preceding example is keyed on the `person` item, Blazor ignores rerendering `Details` components that haven't changed.

To modify the parent component to use the @key directive attribute with the `people` collection, update the `<Details>` element to the following:

razor

```
<Details @key="person" Data="@person.Data" />
```

When the `people` collection changes, the association between `Details` instances and `person` instances is retained. When a `Person` is inserted at the beginning of the collection, one new `Details` instance is inserted at that corresponding position. Other instances are left unchanged. Therefore, the user's focus isn't lost as people are added to the collection.

Other collection updates exhibit the same behavior when the @key directive attribute is used:

- If an instance is deleted from the collection, only the corresponding component instance is removed from the UI. Other instances are left unchanged.
- If collection entries are re-ordered, the corresponding component instances are preserved and re-ordered in the UI.

> **ⓘ Important**
>
> Keys are local to each container element or component. Keys aren't compared globally across the document.

When to use `@key`

Typically, it makes sense to use @key whenever a list is rendered (for example, in a foreach block) and a suitable value exists to define the @key.

You can also use @key to preserve an element or component subtree when an object doesn't change, as the following examples show.

Example 1:

razor

```razor
<li @key="person">
    <input value="@person.Data" />
</li>
```

Example 2:

razor

```razor
<div @key="person">
    @* other HTML elements *@
</div>
```

If an `person` instance changes, the @key attribute directive forces Blazor to:

- Discard the entire `<li>` or `<div>` and their descendants.
- Rebuild the subtree within the UI with new elements and components.

This is useful to guarantee that no UI state is preserved when the collection changes within a subtree.

Scope of `@key`

The @key attribute directive is scoped to its own siblings within its parent.

Consider the following example. The `first` and `second` keys are compared against each other within the same scope of the outer `<div>` element:

razor

```
<div>
    <div @key="first">...</div>
    <div @key="second">...</div>
</div>
```

The following example demonstrates `first` and `second` keys in their own scopes, unrelated to each other and without influence on each other. Each @key scope only applies to its parent `<div>` element, not across the parent `<div>` elements:

razor

```
<div>
    <div @key="first">...</div>
</div>
<div>
    <div @key="second">...</div>
</div>
```

For the `Details` component shown earlier, the following examples render `person` data within the same @key scope and demonstrate typical use cases for @key:

razor

```
<div>
    @foreach (var person in people)
    {
        <Details @key="person" Data="@person.Data" />
    }
</div>
```

razor

```razor
@foreach (var person in people)
{
    <div @key="person">
        <Details Data="@person.Data" />
    </div>
}
```

razor

```razor
<ol>
    @foreach (var person in people)
    {
        <li @key="person">
            <Details Data="@person.Data" />
        </li>
    }
</ol>
```

The following examples only scope @key to the `<div>` or `<li>` element that surrounds each `Details` component instance. Therefore, `person` data for each member of the `people` collection is **not** keyed on each `person` instance across the rendered `Details` components. Avoid the following patterns when using @key:

razor

```razor
@foreach (var person in people)
{
    <div>
        <Details @key="person" Data="@person.Data" />
    </div>
}
```

razor

```razor
<ol>
    @foreach (var person in people)
    {
        <li>
            <Details @key="person" Data="@person.Data" />
        </li>
    }
</ol>
```

When not to use `@key`

There's a performance cost when rendering with @key. The performance cost isn't large, but only specify @key if preserving the element or component benefits the app.

Even if @key isn't used, Blazor preserves child element and component instances as much as possible. The only advantage to using @key is control over *how* model instances are mapped to the preserved component instances, instead of Blazor selecting the mapping.

Values to use for `@key`

Generally, it makes sense to supply one of the following values for @key:

- Model object instances. For example, the `Person` instance (`person`) was used in the earlier example. This ensures preservation based on object reference equality.
- Unique identifiers. For example, unique identifiers can be based on primary key values of type `int`, `string`, or `Guid`.

Ensure that values used for @key don't clash. If clashing values are detected within the same parent element, Blazor throws an exception because it can't deterministically map old elements or components to new elements or components. Only use distinct values, such as object instances or primary key values.

Avoid overwriting parameters in ASP.NET Core Blazor

Article • 12/13/2024

> **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.

By Robert Haken

This article explains how to avoid overwriting parameters in Blazor apps during rerendering.

Overwritten parameters

The Blazor framework generally imposes safe parent-to-child parameter assignment:

- Parameters aren't overwritten unexpectedly.
- Side effects are minimized. For example, additional renders are avoided because they may create infinite rendering loops.

A child component receives new parameter values that possibly overwrite existing values when the parent component rerenders. Accidentally overwriting parameter values in a child component often occurs when developing the component with one or more data-bound parameters and the developer writes directly to a parameter in the child:

- The child component is rendered with one or more parameter values from the parent component.
- The child writes directly to the value of a parameter.
- The parent component rerenders and overwrites the value of the child's parameter.

The potential for overwriting parameter values extends into the child component's property `set` accessors, too.

> **Important**

> Our general guidance is not to create components that directly write to their own parameters after the component is rendered for the first time.

Consider the following `ShowMoreExpander` component that:

- Renders the title.
- Shows the child content when selected.
- Allows you to set initial state with a component parameter (`InitiallyExpanded`).

After the following `ShowMoreExpander` component demonstrates an overwritten parameter, a modified `ShowMoreExpander` component is shown to demonstrate the correct approach for this scenario. The following examples can be placed in a local sample app to experience the behaviors described.

`ShowMoreExpander.razor`:

```
<div @onclick="ShowMore" class="card bg-light mb-3" style="width:30rem">
    <div class="card-header">
        <h2 class="card-title">Show more (<code>Expanded</code> =
@InitiallyExpanded)</h2>
    </div>
    @if (InitiallyExpanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    [Parameter]
    public bool InitiallyExpanded { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private void ShowMore() => InitiallyExpanded = true;
}
```

The `ShowMoreExpander` component is added to the following `Expanders` parent component that may call StateHasChanged:

- Calling StateHasChanged in developer code notifies a component that its state has changed and typically enqueues component rerendering to update the UI. StateHasChanged is covered in more detail later in ASP.NET Core Razor component lifecycle and ASP.NET Core Razor component rendering.

- The button's `@onclick` directive attribute attaches an event handler to the button's `onclick` event. Event handling is covered in more detail later in ASP.NET Core Blazor event handling.

`Expanders.razor`:

```
@page "/expanders"

<PageTitle>Expanders</PageTitle>

<h1>Expanders Example</h1>

<ShowMoreExpander InitiallyExpanded="false">
    Expander 1 content
</ShowMoreExpander>

<ShowMoreExpander InitiallyExpanded="false" />

<button @onclick="StateHasChanged">Call StateHasChanged</button>
```

Initially, the `ShowMoreExpander` components behave independently when their `InitiallyExpanded` properties are set. The child components maintain their states as expected.

If StateHasChanged is called in a parent component, the Blazor framework rerenders child components if their parameters might have changed:

- For a group of parameter types that Blazor explicitly checks, Blazor rerenders a child component if it detects that any of the parameters have changed.
- For unchecked parameter types, Blazor rerenders the child component *regardless of whether or not the parameters have changed*. Child content falls into this category of parameter types because child content is of type RenderFragment, which is a delegate that refers to other mutable objects.

For the `Expanders` component:

- The first `ShowMoreExpander` component sets child content in a potentially mutable RenderFragment, so a call to StateHasChanged in the parent component automatically rerenders the component and potentially overwrites the value of `InitiallyExpanded` to its initial value of `false`.
- The second `ShowMoreExpander` component doesn't set child content. Therefore, a potentially mutable RenderFragment doesn't exist. A call to StateHasChanged in the parent component doesn't automatically rerender the child component, so the component's `InitiallyExpanded` value isn't overwritten.

To maintain state in the preceding scenario, use a *private field* in the `ShowMoreExpander` component to maintain its state.

The following revised `ShowMoreExpander` component:

- Accepts the `InitiallyExpanded` component parameter value from the parent.
- Assigns the component parameter value to a *private field* (`expanded`) in the OnInitialized event.
- Uses the private field to maintain its internal toggle state, which demonstrates how to avoid writing directly to a parameter.

> **Note**
>
> The advice in this section extends to similar logic in component parameter `set` accessors, which can result in similar undesirable side effects.

`ShowMoreExpander.razor`:

```
<div @onclick="Expand" class="card bg-light mb-3" style="width:30rem">
    <div class="card-header">
        <h2 class="card-title">Show more (<code>Expanded</code> = @expanded)
</h2>
    </div>
    @if (expanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    private bool expanded;

    [Parameter]
    public bool InitiallyExpanded { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    protected override void OnInitialized() => expanded = InitiallyExpanded;

    private void Expand() => expanded = true;
}
```

> **Note**

The revised `ShowMoreExpander` doesn't reflect changes to the `InitiallyExpanded` parameter after initialization (`OnInitialized`). In certain scenarios, an already initialized component might receive new parameter values. This can happen, for example, in a primary-subordinate view where the same component is used to render different detail views or when the `/item/{id}` route parameter changes to display a different item.

Consider following `ToggleExpander` component that:

- Allows you to change the state both from inside and outside.
- Handles new parameter values even if the same component instance is reused.

`ToggleExpander.razor`:

```
<div class="card bg-light mb-3" style="width:30rem">
    <div @onclick="Toggle" class="card-header">
        <h2 class="card-title">Toggle (<code>Expanded</code> = @expanded)
</h2>
    </div>
    @if (expanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public EventCallback<bool> ExpandedChanged { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private bool expanded;

    protected override void OnParametersSet() => expanded = Expanded;

    private async Task Toggle()
    {
        expanded = !expanded;
        await ExpandedChanged.InvokeAsync(expanded);
    }
}
```

The `ToggleExpander` component should be used with the `@bind-Expanded="{field}"` binding syntax, allowing two-way synchronization of the parameter.

`ExpandersToggle.razor`:

```
@page "/expanders-toggle"

<PageTitle>Expanders Toggle</PageTitle>

<h1>Expanders Toggle</h1>

<ToggleExpander @bind-Expanded="expanded">
    Expander content
</ToggleExpander>

<button @onclick="Toggle">Toggle</button>

<button @onclick="StateHasChanged">Call StateHasChanged</button>

@code {
    private bool expanded;

    private void Toggle() => expanded = !expanded;
}
```

For more information on parent-child binding, see the following resources:

- Binding with component parameters
- Bind across more than two components
- Blazor Two Way Binding Error (dotnet/aspnetcore #24599)

For more information on change detection, including information on the exact types that Blazor checks, see ASP.NET Core Razor component rendering.

ASP.NET Core Blazor attribute splatting and arbitrary parameters

Article • 10/18/2024

> **ⓘ 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.

Components can capture and render additional attributes in addition to the component's declared parameters. Additional attributes can be captured in a dictionary and then applied to an element, called *splatting*, when the component is rendered using the @attributes Razor directive attribute. This scenario is useful for defining a component that produces a markup element that supports a variety of customizations. For example, it can be tedious to define attributes separately for an `<input>` that supports many parameters.

Attribute splatting

In the following `Splat` component:

- The first `<input>` element (`id="useIndividualParams"`) uses individual component parameters.
- The second `<input>` element (`id="useAttributesDict"`) uses attribute splatting.

`Splat.razor`:

razor

```
@page "/splat"

<PageTitle>SPLAT!</PageTitle>

<h1>Splat Parameters Example</h1>

<input id="useIndividualParams"
       maxlength="@maxlength"
       placeholder="@placeholder"
       required="@required"
```

```
        size="@size" />

<input id="useAttributesDict"
       @attributes="InputAttributes" />

@code {
    private string maxlength = "10";
    private string placeholder = "Input placeholder text";
    private string required = "required";
    private string size = "50";

    private Dictionary<string, object> InputAttributes { get; set; } =
        new()
        {
            { "maxlength", "10" },
            { "placeholder", "Input placeholder text" },
            { "required", "required" },
            { "size", "50" }
        };
}
```

The rendered `<input>` elements in the webpage are identical:

HTML

```
<input id="useIndividualParams"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">

<input id="useAttributesDict"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">
```

Arbitrary attributes

To accept arbitrary attributes, define a component parameter with the CaptureUnmatchedValues property set to `true`:

razor

```
@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object>? InputAttributes { get; set; }
}
```

The CaptureUnmatchedValues property on [Parameter] allows the parameter to match all attributes that don't match any other parameter. A component can only define a single parameter with CaptureUnmatchedValues. The property type used with CaptureUnmatchedValues must be assignable from Dictionary<string, object> with string keys. Use of IEnumerable<KeyValuePair<string, object>> or IReadOnlyDictionary<string, object> are also options in this scenario.

The position of @attributes relative to the position of element attributes is important. When @attributes are splatted on the element, the attributes are processed from right to left (last to first). Consider the following example of a parent component that consumes a child component:

`AttributeOrderChild1.razor`:

razor

```razor
<div @attributes="AdditionalAttributes" extra="5" />

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object>? AdditionalAttributes { get; set; }
}
```

`AttributeOrder1.razor`:

razor

```razor
@page "/attribute-order-1"

<PageTitle>Attribute Order 1</PageTitle>

<h1>Attribute Order Example 1</h1>

<AttributeOrderChild1 extra="10" />

<p>
    View the HTML markup in your browser to inspect the attributes on
    the AttributeOrderChild1 component.
</p>
```

The `AttributeOrderChild1` component's `extra` attribute is set to the right of @attributes. The `AttributeOrderParent1` component's rendered `<div>` contains `extra="5"` when passed through the additional attribute because the attributes are processed right to left (last to first):

HTML

```
<div extra="5" />
```

In the following example, the order of `extra` and @attributes is reversed in the child component's `<div>`:

`AttributeOrderChild2.razor`:

razor

```
<div extra="5" @attributes="AdditionalAttributes" />

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object>? AdditionalAttributes { get; set; }
}
```

`AttributeOrder2.razor`:

razor

```
@page "/attribute-order-2"

<PageTitle>Attribute Order 2</PageTitle>

<h1>Attribute Order Example 2</h1>

<AttributeOrderChild2 extra="10" />

<p>
    View the HTML markup in your browser to inspect the attributes on
    the AttributeOrderChild2 component.
</p>
```

The `<div>` in the parent component's rendered webpage contains `extra="10"` when passed through the additional attribute:

HTML

```
<div extra="10" />
```

ASP.NET Core Blazor layouts

Article • 11/06/2024

> **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 article explains how to create reusable layout components for Blazor apps.

Usefulness of Blazor layouts

Some app elements, such as menus, copyright messages, and company logos, are usually part of app's overall presentation. Placing a copy of the markup for these elements into all of the components of an app isn't efficient. Every time that one of these elements is updated, every component that uses the element must be updated. This approach is costly to maintain and can lead to inconsistent content if an update is missed. *Layouts* solve these problems.

A Blazor layout is a Razor component that shares markup with components that reference it. Layouts can use data binding, dependency injection, and other features of components.

Layout components

Create a layout component

To create a layout component:

- Create a Razor component defined by a Razor template or C# code. Layout components based on a Razor template use the `.razor` file extension just like ordinary Razor components. Because layout components are shared across an app's components, they're usually placed in the app's `Shared` or `Layout` folder. However, layouts can be placed in any location accessible to the components that use it. For example, a layout can be placed in the same folder as the components that use it.

- Inherit the component from LayoutComponentBase. The LayoutComponentBase defines a Body property (RenderFragment type) for the rendered content inside the layout.
- Use the Razor syntax `@Body` to specify the location in the layout markup where the content is rendered.

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The following `DoctorWhoLayout` component shows the Razor template of a layout component. The layout inherits LayoutComponentBase and sets the `@Body` between the navigation bar (`<nav>...</nav>`) and the footer (`<footer>...</footer>`).

`DoctorWhoLayout.razor`:

razor

```
@inherits LayoutComponentBase

<PageTitle>Doctor Who® Database</PageTitle>

<header>
    <h1>Doctor Who® Database</h1>
</header>

<nav>
    <a href="main-list">Main Episode List</a>
    <a href="search">Search</a>
    <a href="new">Add Episode</a>
</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/ https://www.bbc.com";
}
```

`MainLayout` component

In an app created from a Blazor project template, the `MainLayout` component is the app's default layout. Blazor's layout adopts the Flexbox layout model (MDN documentation) (W3C specification).

Blazor's CSS isolation feature applies isolated CSS styles to the `MainLayout` component. By convention, the styles are provided by the accompanying stylesheet of the same name, `MainLayout.razor.css`. The ASP.NET Core framework implementation of the stylesheet is available for inspection in the ASP.NET Core reference source (`dotnet/aspnetcore` GitHub repository):

- Blazor Web App MainLayout.razor.css
- Blazor WebAssembly MainLayout.razor.css

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Apply a layout

Make the layout namespace available

Layout file locations and namespaces changed over time for the Blazor framework. Depending on the version of Blazor and type of Blazor app that you're building, you may need to indicate the layout's namespace when using it. When referencing a layout implementation and the layout isn't found without indicating the layout's namespace, take any of the following approaches:

- Add an `@using` directive to the `_Imports.razor` file for the location of the layouts. In the following example, a folder of layouts with the name `Layout` is inside a `Components` folder, and the app's namespace is `BlazorSample`:

  ```razor
  @using BlazorSample.Components.Layout
  ```

- Add an `@using` directive at the top of the component definition where the layout is used:

  razor

  ```
  @using BlazorSample.Components.Layout
  @layout DoctorWhoLayout
  ```

- Fully qualify the namespace of the layout where it's used:

  razor

  ```
  @layout BlazorSample.Components.Layout.DoctorWhoLayout
  ```

Apply a layout to a component

Use the @layout Razor directive to apply a layout to a routable Razor component that has an @page directive. The compiler converts `@layout` into a LayoutAttribute and applies the attribute to the component class.

The content of the following `Episodes` component is inserted into the `DoctorWhoLayout` at the position of `@Body`.

`Episodes.razor`:

razor

```
@page "/episodes"
@layout DoctorWhoLayout

<h2>Doctor Who® Episodes</h2>

<ul>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfknq">
            <em>The Ribos Operation</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfdsb">
            <em>The Sunmakers</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vhc26">
            <em>Nightmare of Eden</em>
        </a>
```

```
    </li>
</ul>
```

The following rendered HTML markup is produced by the preceding `DoctorWhoLayout` and `Episodes` component. Extraneous markup doesn't appear in order to focus on the content provided by the two components involved:

- The H1 "database" heading (`<h1>...</h1>`) in the header (`<header>...</header>`), navigation bar (`<nav>...</nav>`), and trademark information in the footer (`<footer>...</footer>`) come from the `DoctorWhoLayout` component.
- The H2 "episodes" heading (`<h2>...</h2>`) and episode list (`<ul>...</ul>`) come from the `Episodes` component.

HTML

```
<header>
    <h1 ...>...</h1>
</header>

<nav>
    ...
</nav>

<h2>...</h2>

<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>

<footer>
    ...
</footer>
```

Specifying the layout directly in a component overrides a *default layout*:

- Set by an `@layout` directive imported from an `_Imports.razor` file, as described in the following Apply a layout to a folder of components section.
- Set as the app's default layout, as described in the Apply a default layout to an app section later in this article.

Apply a layout to a folder of components

Every folder of an app can optionally contain a template file named `_Imports.razor`. The compiler includes the directives specified in the imports file in all of the Razor templates

in the same folder and recursively in all of its subfolders. Therefore, an `_Imports.razor` file containing `@layout DoctorWhoLayout` ensures that all of the components in a folder use the `DoctorWhoLayout` component. There's no need to repeatedly add `@layout DoctorWhoLayout` to all of the Razor components (`.razor`) within the folder and subfolders.

`_Imports.razor`:

```razor
@layout DoctorWhoLayout
...
```

The `_Imports.razor` file is similar to the _ViewImports.cshtml file for Razor views and pages but applied specifically to Razor component files.

Specifying a layout in `_Imports.razor` overrides a layout specified as the router's default app layout, which is described in the following section.

> ⚠ **Warning**
>
> Do **not** add a Razor `@layout` directive to the root `_Imports.razor` file, which results in an infinite loop of layouts. To control the default app layout, specify the layout in the Router component. For more information, see the following Apply a default layout to an app section.

> ⓘ **Note**
>
> The @layout Razor directive only applies a layout to routable Razor components with an @page directive.

Apply a default layout to an app

Specify the default app layout in the Router component's RouteView component. Use the DefaultLayout parameter to set the layout type:

```razor
<RouteView RouteData="routeData" DefaultLayout="typeof({LAYOUT})" />
```

In the preceding example, the `{LAYOUT}` placeholder is the layout (for example, `DoctorWhoLayout` if the layout file name is `DoctorWhoLayout.razor`). You may need to idenfity the layout's namespace depending on the .NET version and type of Blazor app. For more information, see the Make the layout namespace available section.

Specifying the layout as a default layout in the Router component's RouteView is a useful practice because you can override the layout on a per-component or per-folder basis, as described in the preceding sections of this article. We recommend using the Router component to set the app's default layout because it's the most general and flexible approach for using layouts.

Apply a layout to arbitrary content (`LayoutView` component)

To set a layout for arbitrary Razor template content, specify the layout with a LayoutView component. You can use a LayoutView in any Razor component. The following example sets a layout component named `ErrorLayout` for the `MainLayout` component's NotFound template (`<NotFound>...</NotFound>`).

razor

```razor
<Router ...>
    <Found ...>
        ...
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(ErrorLayout)">
            <h1>Page not found</h1>
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

You may need to identity the layout's namespace depending on the .NET version and type of Blazor app. For more information, see the Make the layout namespace available section.

> **ⓘ Important**
>
> Blazor Web Apps don't use the **NotFound** parameter (`<NotFound>...</NotFound>` markup), but the parameter is supported for backward compatibility to avoid a breaking change in the framework. The server-side ASP.NET Core middleware

pipeline processes requests on the server. Use server-side techniques to handle bad requests. For more information, see ASP.NET Core Blazor render modes.

Nested layouts

A component can reference a layout that in turn references another layout. For example, nested layouts are used to create a multi-level menu structures.

The following example shows how to use nested layouts. The `Episodes` component shown in the Apply a layout to a component section is the component to display. The component references the `DoctorWhoLayout` component.

The following `DoctorWhoLayout` component is a modified version of the example shown earlier in this article. The header and footer elements are removed, and the layout references another layout, `ProductionsLayout`. The `Episodes` component is rendered where `@Body` appears in the `DoctorWhoLayout`.

`DoctorWhoLayout.razor`:

razor

```
@inherits LayoutComponentBase
@layout ProductionsLayout

<PageTitle>Doctor Who® Database</PageTitle>

<h1>Doctor Who® Database</h1>

<nav>
    <a href="main-episode-list">Main Episode List</a>
    <a href="episode-search">Search</a>
    <a href="new-episode">Add Episode</a>
</nav>

@Body

<div>
    @TrademarkMessage
</div>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/ https://www.bbc.com";
}
```

The `ProductionsLayout` component contains the top-level layout elements, where the header (`<header>...</header>`) and footer (`<footer>...</footer>`) elements now reside. The `DoctorWhoLayout` with the `Episodes` component is rendered where `@Body` appears.

`ProductionsLayout.razor`:

razor

```razor
@inherits LayoutComponentBase

<header>
    <h1>Productions</h1>
</header>

<nav>
    <a href="main-production-list">Main Production List</a>
    <a href="production-search">Search</a>
    <a href="new-production">Add Production</a>
</nav>

@Body

<footer>
    Footer of Productions Layout
</footer>
```

The following rendered HTML markup is produced by the preceding nested layout. Extraneous markup doesn't appear in order to focus on the nested content provided by the three components involved:

- The header (`<header>...</header>`), production navigation bar (`<nav>...</nav>`), and footer (`<footer>...</footer>`) elements and their content come from the `ProductionsLayout` component.
- The H1 "database" heading (`<h1>...</h1>`), episode navigation bar (`<nav>...</nav>`), and trademark information (`<div>...</div>`) come from the `DoctorWhoLayout` component.
- The H2 "episodes" heading (`<h2>...</h2>`) and episode list (`<ul>...</ul>`) come from the `Episodes` component.

HTML

```html
<header>
    ...
</header>

<nav>
    <a href="main-production-list">Main Production List</a>
```

```
    <a href="production-search">Search</a>
    <a href="new-production">Add Production</a>
</nav>

<h1>...</h1>

<nav>
    <a href="main-episode-list">Main Episode List</a>
    <a href="episode-search">Search</a>
    <a href="new-episode">Add Episode</a>
</nav>

<h2>...</h2>

<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>

<div>
    ...
</div>

<footer>
    ...
</footer>
```

Share a Razor Pages layout with integrated components

When routable components are integrated into a Razor Pages app, the app's shared layout can be used with the components. For more information, see Integrate ASP.NET Core Razor components into ASP.NET Core apps.

Sections

To control the content in a layout from a child Razor component, see ASP.NET Core Blazor sections.

Additional resources

- Layout in ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor sections

Article • 11/14/2024

> **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 article explains how to control the content in a Razor component from a child Razor component.

Blazor sections

To control the content in a Razor component from a child Razor component, Blazor supports *sections* using the following built-in components:

- SectionOutlet: Renders content provided by SectionContent components with matching SectionName or SectionId arguments. Two or more SectionOutlet components can't have the same SectionName or SectionId.
- SectionContent: Provides content as a RenderFragment to SectionOutlet components with a matching SectionName or SectionId. If several SectionContent components have the same SectionName or SectionId, the matching SectionOutlet component renders the content of the last rendered SectionContent.

Sections can be used in both layouts and across nested parent-child components.

Although the argument passed to SectionName can use any type of casing, the documentation adopts kebab casing (for example, `top-bar`), which is a common casing choice for HTML element IDs. SectionId receives a static `object` field, and we always recommend Pascal casing for C# field names (for example, `TopbarSection`).

In the following example, the app's main layout component implements an increment counter button for the app's `Counter` component.

If the namespace for sections isn't in the `_Imports.razor` file, add it:

```
razor
```

```
@using Microsoft.AspNetCore.Components.Sections
```

In the `MainLayout` component (`MainLayout.razor`), place a SectionOutlet component and pass a string to the SectionName parameter to indicate the section's name. The following example uses the section name `top-bar`:

razor

```
<SectionOutlet SectionName="top-bar" />
```

In the `Counter` component (`Counter.razor`), create a SectionContent component and pass the matching string (`top-bar`) to its SectionName parameter:

razor

```
<SectionContent SectionName="top-bar">
    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</SectionContent>
```

When the `Counter` component is accessed at `/counter`, the `MainLayout` component renders the increment count button from the `Counter` component where the SectionOutlet component is placed. When any other component is accessed, the increment count button isn't rendered.

Instead of using a named section, you can pass a static `object` with the SectionId parameter to identify the section. The following example also implements an increment counter button for the app's `Counter` component in the app's main layout.

If you don't want other SectionContent components to accidentally match the name of a SectionOutlet, pass an object SectionId parameter to identify the section. This can be useful when designing a Razor class library (RCL). When a SectionOutlet in the RCL uses an object reference with SectionId and the consumer places a SectionContent component with a matching SectionId object, an accidental match by name isn't possible when consumers of the RCL implement other SectionContent components.

The following example also implements an increment counter button for the app's `Counter` component in the app's main layout, using an object reference instead of a section name.

Add a `TopbarSection` static `object` to the `MainLayout` component in an `@code` block:

razor

```razor
@code {
    internal static object TopbarSection = new();
}
```

In the `MainLayout` component's Razor markup, place a SectionOutlet component and pass `TopbarSection` to the SectionId parameter to indicate the section:

razor

```razor
<SectionOutlet SectionId="TopbarSection" />
```

Add a SectionContent component to the app's `Counter` component that renders an increment count button. Use the `MainLayout` component's `TopbarSection` section static `object` as the SectionId (`MainLayout.TopbarSection`).

In `Counter.razor`:

razor

```razor
<SectionContent SectionId="MainLayout.TopbarSection">
    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</SectionContent>
```

When the `Counter` component is accessed, the `MainLayout` component renders the increment count button where the SectionOutlet component is placed.

> ⓘ Note
>
> SectionOutlet and SectionContent components can only set either SectionId or SectionName, not both.

Section interaction with other Blazor features

A section interacts with other Blazor features in the following ways:

- Cascading values flow into section content from where the content is defined by the SectionContent component.
- Unhandled exceptions are handled by error boundaries defined around a SectionContent component.

- A Razor component configured for streaming rendering also configures section content provided by a SectionContent component to use streaming rendering.

Control `<head>` content in ASP.NET Core Blazor apps

Article • 10/18/2024

> **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.

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).

Control `<head>` content in a Razor component

Specify the page's title with the PageTitle component, which enables rendering an HTML `<title>` element to a HeadOutlet component.

Specify `<head>` element content with the HeadContent component, which provides content to a HeadOutlet component.

The following example sets the page's title and description using Razor.

`ControlHeadContent.razor`:

razor

```
@page "/control-head-content"

<PageTitle>@title</PageTitle>

<h1>Control <head> Content Example</h1>

<p>
    Title: @title
</p>

<p>
    Description: @description
</p>
```

```razor
<HeadContent>
    <meta name="description" content="@description">
</HeadContent>

@code {
    private string description = "This description is set by the
component.";
    private string title = "Control <head> Content";
}
```

Set a page title for components via a layout

Set the page title in a layout component:

razor

```razor
@inherits LayoutComponentBase

<PageTitle>Page Title</PageTitle>

<div class="page">
    ...
</div>
```

`HeadOutlet` component

The HeadOutlet component renders content provided by PageTitle and HeadContent components.

In a Blazor Web App created from the project template, the HeadOutlet component in `App.razor` renders `<head>` content:

razor

```razor
<head>
    ...
    <HeadOutlet />
</head>
```

In an app created from the Blazor WebAssembly project template, the HeadOutlet component is added to the RootComponents collection of the WebAssemblyHostBuilder in the client-side `Program` file:

C#

```
builder.RootComponents.Add<HeadOutlet>("head::after");
```

When the ::after pseudo-selector is specified, the contents of the root component are appended to the existing head contents instead of replacing the content. This allows the app to retain static head content in `wwwroot/index.html` without having to repeat the content in the app's Razor components.

Set a default page title in a Blazor Web App

Set the page title in the `App` component (`App.razor`):

razor

```
<head>
    ...
    <HeadOutlet />
    <PageTitle>Page Title</PageTitle>
</head>
```

Not found page title in a Blazor WebAssembly app

In Blazor apps created from the Blazor WebAssembly Standalone App project template, the `NotFound` component template in the `App` component (`App.razor`) sets the page title to `Not found`.

`App.razor`:

razor

```
<NotFound>
    <PageTitle>Not found</PageTitle>
    ...
</NotFound>
```

Additional resources

- Control headers in C# code at startup
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Mozilla MDN Web Docs documentation:

- What's in the head? Metadata in HTML
- <head>: The Document Metadata (Header) element
- <title>: The Document Title element
- <meta>: The metadata element

ASP.NET Core Blazor cascading values and parameters

Article • 10/18/2024

> **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 article explains how to flow data from an ancestor Razor component to descendent components.

Cascading values and parameters provide a convenient way to flow data down a component hierarchy from an ancestor component to any number of descendent components. Unlike Component parameters, cascading values and parameters don't require an attribute assignment for each descendent component where the data is consumed. Cascading values and parameters also allow components to coordinate with each other across a component hierarchy.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the `CascadingType?`, `@ActiveTab?`, `RenderFragment?`, `ITab?`, `TabSet?`, and `string?` types in the article's examples.

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.

The following class is used in this section's examples.

`Dalek.cs`:

C#

```
// "Dalek" ©Terry Nation https://www.imdb.com/name/nm0622334/
// "Doctor Who" ©BBC https://www.bbc.co.uk/programmes/b006q2x0

namespace BlazorSample;

public class Dalek
{
    public int Units { get; set; }
}
```

The following registrations are made in the app's `Program` file with AddCascadingValue:

- `Dalek` with a property value for `Units` is registered as a fixed cascading value.
- A second `Dalek` registration with a different property value for `Units` is named "`AlphaGroup`".

C#

```
builder.Services.AddCascadingValue(sp => new Dalek { Units = 123 });
builder.Services.AddCascadingValue("AlphaGroup", sp => new Dalek { Units =
456 });
```

The following `Daleks` component displays the cascaded values.

`Daleks.razor`:

razor

```
@page "/daleks"

<PageTitle>Daleks</PageTitle>

<h1>Root-level Cascading Value Example</h1>

<ul>
    <li>Dalek Units: @Dalek?.Units</li>
    <li>Alpha Group Dalek Units: @AlphaGroupDalek?.Units</li>
</ul>

<p>
    Dalek© <a href="https://www.imdb.com/name/nm0622334/">Terry Nation</a>
<br>
    Doctor Who© <a href="https://www.bbc.co.uk/programmes/b006q2x0">BBC</a>
</p>

@code {
    [CascadingParameter]
    public Dalek? Dalek { get; set; }
```

```
    [CascadingParameter(Name = "AlphaGroup")]
    public Dalek? AlphaGroupDalek { get; set; }
}
```

In the following example, `Dalek` is registered as a cascading value using CascadingValueSource<T>, where `<T>` is the type. The `isFixed` flag indicates whether the value is fixed. If false, all recipients are subscribed for update notifications, which are issued by calling NotifyChangedAsync. Subscriptions create overhead and reduce performance, so set `isFixed` to `true` if the value doesn't change.

C#

```
builder.Services.AddCascadingValue(sp =>
{
    var dalek = new Dalek { Units = 789 };
    var source = new CascadingValueSource<Dalek>(dalek, isFixed: false);

    return source;
});
```

> ⚠ **Warning**
>
> Registering a component type as a root-level cascading value doesn't register additional services for the type or permit service activation in the component.
>
> Treat required services separately from cascading values, registering them separately from the cascaded type.
>
> Avoid using AddCascadingValue to register a component type as a cascading value. Instead, wrap the `<Router>...</Router>` in the `Routes` component (`Components/Routes.razor`) with the component and adopt global interactive server-side rendering (interactive SSR). For an example, see the CascadingValue component section.

`CascadingValue` component

An ancestor component provides a cascading value using the Blazor framework's CascadingValue component, which wraps a subtree of a component hierarchy and supplies a single value to all of the components within its subtree.

The following example demonstrates the flow of theme information down the component hierarchy to provide a CSS style class to buttons in child components.

The following `ThemeInfo` C# class specifies the theme information.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the app's namespace to your sample app's namespace.

`ThemeInfo.cs`:

```
C#

namespace BlazorSample;

public class ThemeInfo
{
    public string? ButtonClass { get; set; }
}
```

The following layout component specifies theme information (`ThemeInfo`) as a cascading value for all components that make up the layout body of the Body property. `ButtonClass` is assigned a value of btn-success, which is a Bootstrap button style. Any descendent component in the component hierarchy can use the `ButtonClass` property through the `ThemeInfo` cascading value.

`MainLayout.razor`:

```
razor

@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/core/"
target="_blank">About</a>
        </div>

        <CascadingValue Value="theme">
            <article class="content px-4">
                @Body
            </article>
        </CascadingValue>
```

```
    </main>
</div>

<div id="blazor-error-ui" data-nosnippet>
    An unhandled error has occurred.
    <a href="." class="reload">Reload</a>
    <span class="dismiss">✕</span>
</div>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };
}
```

Blazor Web Apps provide alternative approaches for cascading values that apply more broadly to the app than furnishing them via a single layout file:

- Wrap the markup of the `Routes` component in a CascadingValue component to specify the data as a cascading value for all of the app's components.

  The following example cascades `ThemeInfo` data from the `Routes` component.

  `Routes.razor`:

  razor

  ```
  <CascadingValue Value="theme">
      <Router ...>
          ...
      </Router>
  </CascadingValue>

  @code {
      private ThemeInfo theme = new() { ButtonClass = "btn-success" };
  }
  ```

  > **Note**
  >
  > Wrapping the `Routes` component instance in the `App` component (`Components/App.razor`) with a CascadingValue component isn't supported.

- Specify a *root-level cascading value* as a service by calling the AddCascadingValue extension method on the service collection builder.

  The following example cascades `ThemeInfo` data from the `Program` file.

  `Program.cs`

C#

```
builder.Services.AddCascadingValue(sp =>
    new ThemeInfo() { ButtonClass = "btn-primary" });
```

For more information, see the following sections of this article:

- Root-level cascading values
- Cascading values/parameters and render mode boundaries

`[CascadingParameter]` attribute

To make use of cascading values, descendent components declare cascading parameters using the [CascadingParameter] attribute. Cascading values are bound to cascading parameters **by type**. Cascading multiple values of the same type is covered in the Cascade multiple values section later in this article.

The following component binds the `ThemeInfo` cascading value to a cascading parameter, optionally using the same name of `ThemeInfo`. The parameter is used to set the CSS class for the **`Increment Counter (Themed)`** button.

`ThemedCounter.razor`:

razor

```
@page "/themed-counter"

<PageTitle>Themed Counter</PageTitle>

<h1>Themed Counter Example</h1>

<p>Current count: @currentCount</p>

<p>
    <button @onclick="IncrementCount">
        Increment Counter (Unthemed)
    </button>
</p>

<p>
    <button
        class="btn @(ThemeInfo is not null ? ThemeInfo.ButtonClass :
string.Empty)"
        @onclick="IncrementCount">
        Increment Counter (Themed)
    </button>
</p>
```

```
@code {
    private int currentCount = 0;

    [CascadingParameter]
    protected ThemeInfo? ThemeInfo { get; set; }

    private void IncrementCount() => currentCount++;
}
```

Similar to a regular component parameter, components accepting a cascading parameter are rerendered when the cascading value is changed. For instance, configuring a different theme instance causes the `ThemedCounter` component from the CascadingValue component section to rerender.

`MainLayout.razor`:

razor

```
<main>
    <div class="top-row px-4">
        <a href="https://docs.microsoft.com/aspnet/"
target="_blank">About</a>
    </div>

    <CascadingValue Value="theme">
        <article class="content px-4">
            @Body
        </article>
    </CascadingValue>
    <button @onclick="ChangeToDarkTheme">Dark mode</button>
</main>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };

    private void ChangeToDarkTheme()
    {
        theme = new() { ButtonClass = "btn-secondary" };
    }
}
```

CascadingValue<TValue>.IsFixed can be used to indicate that a cascading parameter doesn't change after initialization.

Cascading values/parameters and render mode boundaries

Cascading parameters don't pass data across render mode boundaries:

- Interactive sessions run in a different context than the pages that use static server-side rendering (static SSR). There's no requirement that the server producing the page is even the same machine that hosts some later Interactive Server session, including for WebAssembly components where the server is a different machine to the client. The benefit of static server-side rendering (static SSR) is to gain the full performance of pure stateless HTML rendering.
- State crossing the boundary between static and interactive rendering must be serializable. Components are arbitrary objects that reference a vast chain of other objects, including the renderer, the DI container, and every DI service instance. You must explicitly cause state to be serialized from static SSR to make it available in subsequent interactively-rendered components. Two approaches are adopted:
  - Via the Blazor framework, parameters passed across a static SSR to interactive rendering boundary are serialized automatically if they're JSON-serializable, or an error is thrown.
  - State stored in PersistentComponentState is serialized and recovered automatically if it's JSON-serializable, or an error is thrown.

Cascading parameters aren't JSON-serializable because the typical usage patterns for cascading parameters are somewhat like DI services. There are often platform-specific variants of cascading parameters, so it would be unhelpful to developers if the framework stopped developers from having server-interactive-specific versions or WebAssembly-specific versions. Also, many cascading parameter values in general aren't serializable, so it would be impractical to update existing apps if you had to stop using all nonserializable cascading parameter values.

Recommendations:

- If you need to make state available to all interactive components as a cascading parameter, we recommend using root-level cascading values. A factory pattern is available, and the app can emit updated values after app startup. Root-level cascading values are available to all components, including interactive components, since they're processed as DI services.
- For component library authors, you can create an extension method for library consumers similar to the following:

```
C#

builder.Services.AddLibraryCascadingParameters();
```

Instruct developers to call your extension method. This is a sound alternative to instructing them to add a `<RootComponent>` component in their `MainLayout` component.

Cascade multiple values

To cascade multiple values of the same type within the same subtree, provide a unique Name string to each CascadingValue component and their corresponding [CascadingParameter] attributes.

In the following example, two CascadingValue components cascade different instances of `CascadingType`:

razor

```
<CascadingValue Value="parentCascadeParameter1" Name="CascadeParam1">
    <CascadingValue Value="ParentCascadeParameter2" Name="CascadeParam2">
        ...
    </CascadingValue>
</CascadingValue>

@code {
    private CascadingType? parentCascadeParameter1;

    [Parameter]
    public CascadingType? ParentCascadeParameter2 { get; set; }
}
```

In a descendant component, the cascaded parameters receive their cascaded values from the ancestor component by Name:

razor

```
@code {
    [CascadingParameter(Name = "CascadeParam1")]
    protected CascadingType? ChildCascadeParameter1 { get; set; }

    [CascadingParameter(Name = "CascadeParam2")]
    protected CascadingType? ChildCascadeParameter2 { get; set; }
}
```

Pass data across a component hierarchy

Cascading parameters also enable components to pass data across a component hierarchy. Consider the following UI tab set example, where a tab set component

maintains a series of individual tabs.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the namespace to your sample app's namespace.

Create an `ITab` interface that tabs implement in a folder named `UIInterfaces`.

`UIInterfaces/ITab.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.UIInterfaces;

public interface ITab
{
    RenderFragment ChildContent { get; }
}
```

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The following `TabSet` component maintains a set of tabs. The tab set's `Tab` components, which are created later in this section, supply the list items (`<li>...</li>`) for the list (`<ul>...</ul>`).

Child `Tab` components aren't explicitly passed as parameters to the `TabSet`. Instead, the child `Tab` components are part of the child content of the `TabSet`. However, the `TabSet` still needs a reference each `Tab` component so that it can render the headers and the active tab. To enable this coordination without requiring additional code, the `TabSet` component *can provide itself as a cascading value* that is then picked up by the descendent `Tab` components.

`TabSet.razor`:

razor

```razor
@using BlazorSample.UIInterfaces

<!-- Display the tab headers -->

<CascadingValue Value="this">
    <ul class="nav nav-tabs">
        @ChildContent
    </ul>
</CascadingValue>

<!-- Display body for only the active tab -->

<div class="nav-tabs-body p-4">
    @ActiveTab?.ChildContent
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public ITab? ActiveTab { get; private set; }

    public void AddTab(ITab tab)
    {
        if (ActiveTab is null)
        {
            SetActiveTab(tab);
        }
    }

    public void SetActiveTab(ITab tab)
    {
        if (ActiveTab != tab)
        {
            ActiveTab = tab;
            StateHasChanged();
        }
    }
}
```

Descendent `Tab` components capture the containing `TabSet` as a cascading parameter. The `Tab` components add themselves to the `TabSet` and coordinate to set the active tab.

`Tab.razor`:

razor

```razor
@using BlazorSample.UIInterfaces
@implements ITab
```

```
<li>
    <a @onclick="ActivateTab" class="nav-link @TitleCssClass" role="button">
        @Title
    </a>
</li>

@code {
    [CascadingParameter]
    public TabSet? ContainerTabSet { get; set; }

    [Parameter]
    public string? Title { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private string? TitleCssClass =>
        ContainerTabSet?.ActiveTab == this ? "active" : null;

    protected override void OnInitialized()
    {
        ContainerTabSet?.AddTab(this);
    }

    private void ActivateTab()
    {
        ContainerTabSet?.SetActiveTab(this);
    }
}
```

The following `ExampleTabSet` component uses the `TabSet` component, which contains three `Tab` components.

`ExampleTabSet.razor`:

razor

```
@page "/example-tab-set"

<TabSet>
    <Tab Title="First tab">
        <h4>Greetings from the first tab!</h4>

        <label>
            <input type="checkbox" @bind="showThirdTab" />
            Toggle third tab
        </label>
    </Tab>

    <Tab Title="Second tab">
        <h4>Hello from the second tab!</h4>
    </Tab>
```

```
    @if (showThirdTab)
    {
        <Tab Title="Third tab">
            <h4>Welcome to the disappearing third tab!</h4>
            <p>Toggle this tab from the first tab.</p>
        </Tab>
    }
</TabSet>

@code {
    private bool showThirdTab;
}
```

Additional resources

- Generic type support: Explicit generic types based on ancestor components
- State management: Factor out the state preservation to a common location

ASP.NET Core Blazor event handling

Article • 10/18/2024

> **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 article explains Blazor's event handling features, including event argument types, event callbacks, and managing default browser events.

Delegate event handlers

Specify delegate event handlers in Razor component markup with `@on{DOM EVENT}="{DELEGATE}"` Razor syntax:

- The `{DOM EVENT}` placeholder is a DOM event (for example, `click`).
- The `{DELEGATE}` placeholder is the C# delegate event handler.

For event handling:

- Delegate event handlers in Blazor Web Apps are only called in components that adopt an interactive render mode. The examples throughout this article assume that the app adopts an interactive render mode globally in the app's root component, typically the `App` component. For more information, see ASP.NET Core Blazor render modes.
- Asynchronous delegate event handlers that return a Task are supported.
- Delegate event handlers automatically trigger a UI render, so there's no need to manually call StateHasChanged.
- Exceptions are logged.

The following code:

- Calls the `UpdateHeading` method when the button is selected in the UI.
- Calls the `CheckChanged` method when the checkbox is changed in the UI.

`EventHandler1.razor`:

razor

```razor
@page "/event-handler-1"

<PageTitle>Event Handler 1</PageTitle>

<h1>Event Handler Example 1</h1>

<h2>@headingValue</h2>

<p>
    <button @onclick="UpdateHeading">
        Update heading
    </button>
</p>

<p>
    <label>
        <input type="checkbox" @onchange="CheckChanged" />
        @checkedMessage
    </label>
</p>

@code {
    private string headingValue = "Initial heading";
    private string checkedMessage = "Not changed yet";

    private void UpdateHeading() => headingValue = $"New heading
({DateTime.Now})";

    private void CheckChanged() => checkedMessage = $"Last change
{DateTime.Now}";
}
```

In the following example, `UpdateHeading`:

- Is called asynchronously when the button is selected.
- Waits two seconds before updating the heading.

`EventHandler2.razor`:

razor

```razor
@page "/event-handler-2"

<PageTitle>Event Handler 2</PageTitle>

<h1>Event Handler Example 2</h1>

<h2>@headingValue</h2>

<p>
```

```
    <button @onclick="UpdateHeading">
        Update heading
    </button>
</p>

@code {
    private string headingValue = "Initial heading";

    private async Task UpdateHeading()
    {
        await Task.Delay(2000);

        headingValue = $"New heading ({DateTime.Now})";
    }
}
```

Built-in event arguments

For events that support an event argument type, specifying an event parameter in the event method definition is only necessary if the event type is used in the method. In the following example, MouseEventArgs is used in the `ReportPointerLocation` method to set message text that reports the mouse coordinates when the user selects a button in the UI.

`EventHandler3.razor`:

razor

```
@page "/event-handler-3"

<PageTitle>Event Handler 3</PageTitle>

<h1>Event Handler Example 3</h1>

@for (var i = 0; i < 4; i++)
{
    <p>
        <button @onclick="ReportPointerLocation">
            Where's my mouse pointer for this button?
        </button>
    </p>
}

<p>@mousePointerMessage</p>

@code {
    private string? mousePointerMessage;

    private void ReportPointerLocation(MouseEventArgs e) =>
```

```
        mousePointerMessage = $"Mouse coordinates: {e.ScreenX}:{e.ScreenY}";
}
```

Supported EventArgs are shown in the following table.

Expand table

Event	Class	DOM notes
Clipboard	ClipboardEventArgs	
Drag	DragEventArgs	DataTransfer and DataTransferItem hold dragged item data. Implement drag and drop in Blazor apps using JS interop with HTML Drag and Drop API.
Error	ErrorEventArgs	
Event	EventArgs	EventHandlers holds attributes to configure the mappings between event names and event argument types.
Focus	FocusEventArgs	Doesn't include support for `relatedTarget`.
Input	ChangeEventArgs	
Keyboard	KeyboardEventArgs	
Mouse	MouseEventArgs	
Mouse pointer	PointerEventArgs	
Mouse wheel	WheelEventArgs	
Progress	ProgressEventArgs	
Touch	TouchEventArgs	TouchPoint represents a single contact point on a touch-sensitive device.

For more information, see the following resources:

- EventArgs classes in the ASP.NET Core reference source (dotnet/aspnetcore main branch)

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release

of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- EventHandlers holds attributes to configure the mappings between event names and event argument types.

Custom event arguments

Blazor supports custom event arguments, which enable you to pass arbitrary data to .NET event handlers with custom events.

General configuration

Custom events with custom event arguments are generally enabled with the following steps.

In JavaScript, define a function for building the custom event argument object from the source event:

JavaScript

```
function eventArgsCreator(event) {
  return {
    customProperty1: 'any value for property 1',
    customProperty2: event.srcElement.id
  };
}
```

The `event` parameter is a DOM Event (MDN documentation).

Register the custom event with the preceding handler in a JavaScript initializer. Provide the appropriate browser event name to `browserEventName`, which for the example shown in this section is `click` for a button selection in the UI.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js` (the `{PACKAGE ID/ASSEMBLY NAME}` placeholder is the package ID or assembly name of the app):

For a Blazor Web App:

JavaScript

```
export function afterWebStarted(blazor) {
  blazor.registerCustomEventType('customevent', {
```

```
    browserEventName: 'click',
    createEventArgs: eventArgsCreator
  });
}
```

For a Blazor Server or Blazor WebAssembly app:

JavaScript

```
export function afterStarted(blazor) {
  blazor.registerCustomEventType('customevent', {
    browserEventName: 'click',
    createEventArgs: eventArgsCreator
  });
}
```

The call to `registerCustomEventType` is performed in a script only once per event.

For the call to `registerCustomEventType`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

The custom event name, `customevent` in the preceding example, must not match a reserved Blazor event name. The reserved names can be found in the Blazor framework reference source (see the calls to the registerBuiltInEventType function).

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Define a class for the event arguments:

C#

```
namespace BlazorSample.CustomEvents;

public class CustomEventArgs : EventArgs
{
    public string? CustomProperty1 {get; set;}
    public string? CustomProperty2 {get; set;}
}
```

Wire up the custom event with the event arguments by adding an [EventHandler] attribute annotation for the custom event:

- In order for the compiler to find the `[EventHandler]` class, it must be placed into a C# class file (`.cs`), making it a normal top-level class.
- Mark the class `public`.
- The class doesn't require members.
- The class *must* be called "`EventHandlers`" in order to be found by the Razor compiler.
- Place the class under a namespace specific to your app.
- Import the namespace into the Razor component (`.razor`) where the event is used.

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.CustomEvents;

[EventHandler("oncustomevent", typeof(CustomEventArgs),
    enableStopPropagation: true, enablePreventDefault: true)]
public static class EventHandlers
{
}
```

Register the event handler on one or more HTML elements. Access the data that was passed in from JavaScript in the delegate handler method:

razor

```
@using BlazorSample.CustomEvents

<button id="buttonId" @oncustomevent="HandleCustomEvent">Handle</button>

@if (!string.IsNullOrEmpty(propVal1) && !string.IsNullOrEmpty(propVal2))
{
    <ul>
        <li>propVal1: @propVal1</li>
        <li>propVal2: @propVal2</li>
    </ul>
}

@code
{
    private string? propVal1;
    private string? propVal2;

    private void HandleCustomEvent(CustomEventArgs eventArgs)
```

```
    {
        propVal1 = eventArgs.CustomProperty1;
        propVal2 = eventArgs.CustomProperty2;
    }
}
```

If the `@oncustomevent` attribute isn't recognized by IntelliSense, make sure that the component or the `_Imports.razor` file contains an `@using` statement for the namespace containing the `EventHandler` class.

Whenever the custom event is fired on the DOM, the event handler is called with the data passed from the JavaScript.

If you're attempting to fire a custom event, bubbles must be enabled by setting its value to `true`. Otherwise, the event doesn't reach the Blazor handler for processing into the C# custom [EventHandler] attribute class. For more information, see MDN Web Docs: Event bubbling.

Custom clipboard paste event example

The following example receives a custom clipboard paste event that includes the time of the paste and the user's pasted text.

Declare a custom name (`oncustompaste`) for the event and a .NET class (`CustomPasteEventArgs`) to hold the event arguments for this event:

`CustomEvents.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.CustomEvents;

[EventHandler("oncustompaste", typeof(CustomPasteEventArgs),
    enableStopPropagation: true, enablePreventDefault: true)]
public static class EventHandlers
{
}

public class CustomPasteEventArgs : EventArgs
{
    public DateTime EventTimestamp { get; set; }
    public string? PastedData { get; set; }
}
```

Add JavaScript code to supply data for the EventArgs subclass with the preceding handler in a JavaScript initializer. The following example only handles pasting text, but you could use arbitrary JavaScript APIs to deal with users pasting other types of data, such as images.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js`:

For a Blazor Web App:

JavaScript

```
export function afterWebStarted(blazor) {
  blazor.registerCustomEventType('custompaste', {
    browserEventName: 'paste',
    createEventArgs: event => {
      return {
        eventTimestamp: new Date(),
        pastedData: event.clipboardData.getData('text')
      };
    }
  });
}
```

For a Blazor Server or Blazor WebAssembly app:

JavaScript

```
export function afterStarted(blazor) {
  blazor.registerCustomEventType('custompaste', {
    browserEventName: 'paste',
    createEventArgs: event => {
      return {
        eventTimestamp: new Date(),
        pastedData: event.clipboardData.getData('text')
      };
    }
  });
}
```

In the preceding example, the `{PACKAGE ID/ASSEMBLY NAME}` placeholder of the file name represents the package ID or assembly name of the app.

> **Note**

For the call to `registerCustomEventType`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

The preceding code tells the browser that when a native paste event occurs:

- Raise a `custompaste` event.
- Supply the event arguments data using the custom logic stated:
  - For the `eventTimestamp`, create a new date.
  - For the `pastedData`, get the clipboard data as text. For more information, see MDN Web Docs: ClipboardEvent.clipboardData.

Event name conventions differ between .NET and JavaScript:

- In .NET, event names are prefixed with "`on`".
- In JavaScript, event names don't have a prefix.

In a Razor component, attach the custom handler to an element.

`CustomPasteArguments.razor`:

razor

```
@page "/custom-paste-arguments"
@using BlazorSample.CustomEvents

<label>
    Try pasting into the following text box:
    <input @oncustompaste="HandleCustomPaste" />
</label>

<p>
    @message
</p>

@code {
    private string? message;

    private void HandleCustomPaste(CustomPasteEventArgs eventArgs)
    {
        message = $"At {eventArgs.EventTimestamp.ToShortTimeString()}, " +
            $"you pasted: {eventArgs.PastedData}";
    }
}
```

Lambda expressions

Lambda expressions are supported as the delegate event handler.

`EventHandler4.razor`:

razor

```
@page "/event-handler-4"

<PageTitle>Event Handler 4</PageTitle>

<h1>Event Handler Example 4</h1>

<h2>@heading</h2>

<p>
    <button @onclick="@(e => heading = "New heading!!!")">
        Update heading
    </button>
</p>

@code {
    private string heading = "Initial heading";
}
```

It's often convenient to close over additional values using C# method parameters, such as when iterating over a set of elements. The following example creates three buttons, each of which calls `UpdateHeading` and passes the following data:

- An event argument (MouseEventArgs) in `e`.
- The button number in `buttonNumber`.

`EventHandler5.razor`:

razor

```
@page "/event-handler-5"

<PageTitle>Event Handler 5</PageTitle>

<h1>Event Handler Example 5</h1>

<h2>@heading</h2>

@for (var i = 1; i < 4; i++)
{
    var buttonNumber = i;

    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@i
```

```
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private void UpdateHeading(MouseEventArgs e, int buttonNumber) =>
        heading = $"Selected #{buttonNumber} at {e.ClientX}:{e.ClientY}";
}
```

Creating a large number of event delegates in a loop may cause poor rendering performance. For more information, see ASP.NET Core Blazor performance best practices.

Avoid using a loop variable directly in a lambda expression, such as `i` in the preceding `for` loop example. Otherwise, the same variable is used by all lambda expressions, which results in use of the same value in all lambdas. Capture the variable's value in a local variable. In the preceding example:

- The loop variable `i` is assigned to `buttonNumber`.
- `buttonNumber` is used in the lambda expression.

Alternatively, use a `foreach` loop with Enumerable.Range, which doesn't suffer from the preceding problem:

razor

```
@foreach (var buttonNumber in Enumerable.Range(1, 3))
{
    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@buttonNumber
        </button>
    </p>
}
```

EventCallback

A common scenario with nested components is executing a method in a parent component when a child component event occurs. An `onclick` event occurring in the child component is a common use case. To expose events across components, use an EventCallback. A parent component can assign a callback method to a child component's EventCallback.

The following `Child` component demonstrates how a button's `onclick` handler is set up to receive an EventCallback delegate from the sample's `ParentComponent`. The EventCallback is typed with MouseEventArgs, which is appropriate for an `onclick` event from a peripheral device.

`Child.razor`:

razor

```razor
<p>
    <button @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</p>

@code {
    [Parameter]
    public string? Title { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

The parent component sets the child's EventCallback<TValue> (`OnClickCallback`) to its `ShowMessage` method.

`ParentChild.razor`:

razor

```razor
@page "/parent-child"

<PageTitle>Parent Child</PageTitle>

<h1>Parent Child Example</h1>

<Child Title="Panel Title from Parent" OnClickCallback="ShowMessage">
    Content of the child component is supplied by the parent component.
</Child>

<p>@message</p>

@code {
    private string? message;

    private void ShowMessage(MouseEventArgs e) =>
        message = $"Blaze a new trail with Blazor! ({e.ScreenX}:
```

```
{e.ScreenY})";
}
```

When the button is selected in the `ChildComponent`:

- The `Parent` component's `ShowMessage` method is called. `message` is updated and displayed in the `Parent` component.
- A call to StateHasChanged isn't required in the callback's method (`ShowMessage`). StateHasChanged is called automatically to rerender the `Parent` component, just as child events trigger component rerendering in event handlers that execute within the child. For more information, see ASP.NET Core Razor component rendering.

Use EventCallback and EventCallback<TValue> for event handling and binding component parameters.

Prefer the strongly typed EventCallback<TValue> over EventCallback. EventCallback<TValue> provides enhanced error feedback when an inappropriate type is used, guiding users of the component towards correct implementation. Similar to other UI event handlers, specifying the event parameter is optional. Use EventCallback when there's no value passed to the callback.

EventCallback and EventCallback<TValue> permit asynchronous delegates. EventCallback is weakly typed and allows passing any type argument in `InvokeAsync(Object)`. EventCallback<TValue> is strongly typed and requires passing a `T` argument in `InvokeAsync(T)` that's assignable to `TValue`.

Invoke an EventCallback or EventCallback<TValue> with InvokeAsync and await the Task:

C#

```
await OnClickCallback.InvokeAsync({ARGUMENT});
```

In the preceding example, the `{ARGUMENT}` placeholder is an optional argument.

The following parent-child example demonstrates the technique.

`Child2.razor`:

razor

```
<h3>Child2 Component</h3>

<button @onclick="TriggerEvent">Click Me</button>
```

```
@code {
    [Parameter]
    public EventCallback<string> OnClickCallback { get; set; }

    private async Task TriggerEvent()
    {
        await OnClickCallback.InvokeAsync("Blaze It!");
    }
}
```

`ParentChild2.razor`:

razor

```
@page "/parent-child-2"

<PageTitle>Parent Child 2</PageTitle>

<h1>Parent Child 2 Example</h1>

<div>
    <Child2 OnClickCallback="(value) => { message1 = value; }" />
    @message1
</div>

<div>
    <Child2 OnClickCallback=
        "async (value) => { await Task.Delay(2000); message2 = value; }" />
    @message2
</div>

@code {
    private string message1 = string.Empty;
    private string message2 = string.Empty;
}
```

The second occurrence of the `Child2` component demonstrates an asynchronous callback, and the new `message2` value is assigned and rendered with a delay of two seconds.

Prevent default actions

Use the @on{DOM EVENT}:preventDefault directive attribute to prevent the default action for an event, where the `{DOM EVENT}` placeholder is a DOM event.

When a key is selected on an input device and the element focus is on a text box, a browser normally displays the key's character in the text box. In the following example, the default behavior is prevented by specifying the `@onkeydown:preventDefault` directive

attribute. When the focus is on the `<input>` element, the counter increments with the key sequence `Shift`+`+`. The `+` character isn't assigned to the `<input>` element's value. For more information on `keydown`, see MDN Web Docs: Document: keydown event.

`EventHandler6.razor`:

razor

```razor
@page "/event-handler-6"

<PageTitle>Event Handler 6</PageTitle>

<h1>Event Handler Example 6</h1>

<p>For this example, give the <code><input></code> focus.</p>

<p>
    <label>
        Count of '+' key presses:
        <input value="@count" @onkeydown="KeyHandler"
@onkeydown:preventDefault />
    </label>
</p>

@code {
    private int count = 0;

    private void KeyHandler(KeyboardEventArgs e)
    {
        if (e.Key == "+")
        {
            count++;
        }
    }
}
```

Specifying the `@on{DOM EVENT}:preventDefault` attribute without a value is equivalent to `@on{DOM EVENT}:preventDefault="true"`.

An expression is also a permitted value of the attribute. In the following example, `shouldPreventDefault` is a `bool` field set to either `true` or `false`:

razor

```razor
<input @onkeydown:preventDefault="shouldPreventDefault" />

...

@code {
```

```
    private bool shouldPreventDefault = true;
}
```

Stop event propagation

Use the @on{DOM EVENT}:stopPropagation directive attribute to stop event propagation within the Blazor scope. `{DOM EVENT}` is a placeholder for a DOM event.

The `stopPropagation` directive attribute's effect is limited to the Blazor scope and doesn't extend to the HTML DOM. Events must propagate to the HTML DOM root before Blazor can act upon them. For a mechanism to prevent HTML DOM event propagation, consider the following approach:

- Obtain the event's path by calling Event.composedPath().
- Filter events based on the composed event targets (EventTarget).

In the following example, selecting the checkbox prevents click events from the second child `<div>` from propagating to the parent `<div>`. Since propagated click events normally fire the `OnSelectParentDiv` method, selecting the second child `<div>` results in the parent `<div>` message appearing unless the checkbox is selected.

`EventHandler7.razor`:

razor

```
@page "/event-handler-7"

<PageTitle>Event Handler 7</PageTitle>

<h1>Event Handler Example 7</h1>

<div>
    <b>stopPropagation</b>: @stopPropagation
</div>

<div>
    <button @onclick="StopPropagation">
        Stop Propagation (stopPropagation = true)
    </button>
    <button @onclick="EnablePropagation">
        Enable Propagation (stopPropagation = false)
    </button>
</div>

<div class="m-1 p-1 border border-primary" @onclick="OnSelectParentDiv">
    <h3>Parent div</h3>
```

```
    <div class="m-1 p-1 border" @onclick="OnSelectChildDiv">
        Child div that never stops propagation to the parent div when
        selected.
    </div>

    <div class="m-1 p-1 border" @onclick="OnSelectChildDiv"
            @onclick:stopPropagation="stopPropagation">
        Child div that stops propagation when selected if
        <b>stopPropagation</b> is <b>true</b>.
    </div>
</div>

<p>
    @message
</p>

@code {
    private bool stopPropagation = false;
    private string? message;

    private void StopPropagation() => stopPropagation = true;

    private void EnablePropagation() => stopPropagation = false;

    private void OnSelectParentDiv() =>
        message = $"The parent div was selected. {DateTime.Now}";

    private void OnSelectChildDiv() =>
        message = $"The child div was selected. {DateTime.Now}";
}
```

Focus an element

Call FocusAsync on an element reference to focus an element in code. In the following example, select the button to focus the `<input>` element.

`EventHandler8.razor`:

razor

```
@page "/event-handler-8"

<PageTitle>Event Handler 8</PageTitle>

<h1>Event Handler Example 8</h1>

<p>Select the button to give the <code><input></code> focus.</p>

<p>
    <label>
        Input:
```

```
        <input @ref="exampleInput" />
    </label>

</p>

<button @onclick="ChangeFocus">
    Focus the Input Element
</button>

@code {
    private ElementReference exampleInput;

    private async Task ChangeFocus()
    {
        await exampleInput.FocusAsync();
    }
}
```

ASP.NET Core Blazor data binding

Article • 10/18/2024

> **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 article explains data binding features for Razor components and DOM elements in Blazor apps.

Binding features

Razor components provide data binding features with the @bind Razor directive attribute with a field, property, or Razor expression value.

The following example binds:

- An `<input>` element value to the C# `inputValue` field.
- A second `<input>` element value to the C# `InputValue` property.

When an `<input>` element loses focus, its bound field or property is updated.

`Bind.razor`:

razor

```razor
@page "/bind"

<PageTitle>Bind</PageTitle>

<h1>Bind Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>
</p>

<p>
    <label>
```

```
        InputValue:
        <input @bind="InputValue" />
    </label>
</p>

<ul>
    <li><code>inputValue</code>: @inputValue</li>
    <li><code>InputValue</code>: @InputValue</li>
</ul>

@code {
    private string? inputValue;

    private string? InputValue { get; set; }
}
```

The text box is updated in the UI only when the component is rendered, not in response to changing the field's or property's value. Since components render themselves after event handler code executes, field and property updates are usually reflected in the UI immediately after an event handler is triggered.

As a demonstration of how data binding composes in HTML, the following example binds the `InputValue` property to the second `<input>` element's `value` and `onchange` attributes (change). *The second `<input>` element in the following example is a concept demonstration and isn't meant to suggest how you should bind data in Razor components.*

`BindTheory.razor`:

razor

```
@page "/bind-theory"

<PageTitle>Bind Theory</PageTitle>

<h1>Bind Theory Example</h1>

<p>
    <label>
        Normal Blazor binding:
        <input @bind="InputValue" />
    </label>
</p>

<p>
    <label>
        Demonstration of equivalent HTML binding:
        <input value="@InputValue" @onchange="@((ChangeEventArgs __e) =>
            InputValue = __e?.Value?.ToString())" />
    </label>
</p>
```

```
<p>
    <code>InputValue</code>: @InputValue
</p>

@code {
    private string? InputValue { get; set; }
}
```

When the `BindTheory` component is rendered, the `value` of the HTML demonstration `<input>` element comes from the `InputValue` property. When the user enters a value in the text box and changes element focus, the `onchange` event is fired and the `InputValue` property is set to the changed value. In reality, code execution is more complex because @bind handles cases where type conversions are performed. In general, @bind associates the current value of an expression with the `value` attribute of the `<input>` and handles changes using the registered handler.

Bind a property or field on other DOM events by including an `@bind:event="{EVENT}"` attribute with a DOM event for the `{EVENT}` placeholder. The following example binds the `InputValue` property to the `<input>` element's value when the element's `oninput` event (input) is triggered. Unlike the `onchange` event (change), which fires when the element loses focus, `oninput` (input) fires when the value of the text box changes.

`Page/BindEvent.razor`:

razor

```
@page "/bind-event"

<PageTitle>Bind Event</PageTitle>

<h1>Bind Event Example</h1>

<p>
    <label>
        InputValue:
        <input @bind="InputValue" @bind:event="oninput" />
    </label>

</p>

<p>
    <code>InputValue</code>: @InputValue
</p>

@code {
```

```
    private string? InputValue { get; set; }
}
```

To execute asynchronous logic after binding, use `@bind:after="{EVENT}"` with a DOM event for the `{EVENT}` placeholder. An assigned C# method isn't executed until the bound value is assigned synchronously.

Using an event callback parameter (EventCallback/EventCallback<T>) with `@bind:after` isn't supported. Instead, pass a method that returns an Action or Task to `@bind:after`.

In the following example:

- Each `<input>` element's `value` is bound to the `searchText` field synchronously.
- The `PerformSearch` method executes asynchronously:
  - When the first box loses focus (`onchange` event) after the value is changed.
  - After each keystroke (`oninput` event) in the second box.
- `PerformSearch` calls a service with an asynchronous method (`FetchAsync`) to return search results.

razor

```
@inject ISearchService SearchService

<input @bind="searchText" @bind:after="PerformSearch" />
<input @bind="searchText" @bind:event="oninput" @bind:after="PerformSearch"
/>

@code {
    private string? searchText;
    private string[]? searchResult;

    private async Task PerformSearch() =>
        searchResult = await SearchService.FetchAsync(searchText);
}
```

Additional examples

`BindAfter.razor`:

razor

```
@page "/bind-after"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind After Examples</h1>

<h2>Elements</h2>
```

```
<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind="text" @bind:after="After" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<h2>Components</h2>

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value="text" @bind-Value:after="After" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

@code {
    private string text = "";

    private void After() {}
    private Task AfterAsync() { return Task.CompletedTask; }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Components 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

`BindGetSet.razor`:

razor

```
@page "/bind-get-set"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind Get Set Examples</h1>

<h2>Elements</h2>

<input type="text" @bind:get="text" @bind:set="(value) => { text = value; }"
/>
<input type="text" @bind:get="text" @bind:set="Set" />
<input type="text" @bind:get="text" @bind:set="SetAsync" />

<h2>Components</h2>
```

```razor
<InputText @bind-Value:get="text" @bind-Value:set="(value) => { text =
value; }" />
<InputText @bind-Value:get="text" @bind-Value:set="Set" />
<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void Set(string value)
    {
        text = value;
    }

    private Task SetAsync(string value)
    {
        text = value;
        return Task.CompletedTask;
    }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

For another example use of `@bind:get` and `@bind:set`, see the Bind across more than two components section later in this article.

Razor attribute binding is case-sensitive:

- `@bind`, `@bind:event`, and `@bind:after` are valid.
- `@Bind`/`@bind:Event`/`@bind:aftEr` (capital letters) or `@BIND`/`@BIND:EVENT`/`@BIND:AFTER` (all capital letters) **are invalid**.

Use `@bind:get`/`@bind:set` modifiers and avoid event handlers for two-way data binding

Two-way data binding isn't possible to implement with an event handler. Use `@bind:get`/`@bind:set` modifiers for two-way data binding.

❌ Consider the following ***dysfunctional approach*** for two-way data binding using an event handler:

razor

```razor
<p>
    <input value="@inputValue" @oninput="OnInput" />
</p>
```

```
<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private string? inputValue;

    private void OnInput(ChangeEventArgs args)
    {
        var newValue = args.Value?.ToString() ?? string.Empty;

        inputValue = newValue.Length > 4 ? "Long!" : newValue;
    }
}
```

The `OnInput` event handler updates the value of `inputValue` to `Long!` after a fourth character is provided. However, the user can continue adding characters to the element value in the UI. The value of `inputValue` isn't bound back to the element's value with each keystroke. The preceding example is only capable of one-way data binding.

The reason for this behavior is that Blazor isn't aware that your code intends to modify the value of `inputValue` in the event handler. Blazor doesn't try to force DOM element values and .NET variable values to match unless they're bound with `@bind` syntax. In earlier versions of Blazor, two-way data binding is implemented by binding the element to a property and controlling the property's value with its setter. In ASP.NET Core in .NET 7 or later, `@bind:get`/`@bind:set` modifier syntax is used to implement two-way data binding, as the next example demonstrates.

✔ Consider the following ***correct approach*** using `@bind:get`/`@bind:set` for two-way data binding:

razor

```
<p>
    <input @bind:event="oninput" @bind:get="inputValue" @bind:set="OnInput"
/>
</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private string? inputValue;

    private void OnInput(string value)
    {
        var newValue = value ?? string.Empty;
```

```
        inputValue = newValue.Length > 4 ? "Long!" : newValue;
    }
}
```

Using `@bind:get`/`@bind:set` modifiers both controls the underlying value of `inputValue` via `@bind:set` and binds the value of `inputValue` to the element's value via `@bind:get`. The preceding example demonstrates the correct approach for implementing two-way data binding.

Binding to a property with C# `get` and `set` accessors

C# get and set accessors can be used to create custom binding format behavior, as the following `DecimalBinding` component demonstrates. The component binds a positive or negative decimal with up to three decimal places to an `<input>` element by way of a `string` property (`DecimalValue`).

`DecimalBinding.razor`:

razor

```
@page "/decimal-binding"
@using System.Globalization

<PageTitle>Decimal Binding</PageTitle>

<h1>Decimal Binding Example</h1>

<p>
    <label>
        Decimal value (±0.000 format):
        <input @bind="DecimalValue" />
    </label>
</p>

<p>
    <code>decimalValue</code>: @decimalValue
</p>

@code {
    private decimal decimalValue = 1.1M;
    private NumberStyles style =
        NumberStyles.AllowDecimalPoint | NumberStyles.AllowLeadingSign;
    private CultureInfo culture = CultureInfo.CreateSpecificCulture("en-
US");
```

```
    private string DecimalValue
    {
        get => decimalValue.ToString("0.000", culture);
        set
        {
            if (Decimal.TryParse(value, style, culture, out var number))
            {
                decimalValue = Math.Round(number, 3);
            }
        }
    }
}
```

> **Note**
>
> Two-way binding to a property with `get`/`set` accessors requires discarding the Task returned by EventCallback.InvokeAsync. For two-way data binding, we recommend using `@bind:get`/`@bind:set` modifiers. For more information, see the `@bind:get`/`@bind:set` guidance in the earlier in this article.

Multiple option selection with `<select>` elements

Binding supports multiple option selection with `<select>` elements. The @onchange event provides an array of the selected elements via event arguments (ChangeEventArgs). The value must be bound to an array type.

`BindMultipleInput.razor`:

razor

```
@page "/bind-multiple-input"

<h1>Bind Multiple <code>input</code>Example</h1>

<p>
    <label>
        Select one or more cars:
        <select @onchange="SelectedCarsChanged" multiple>
            <option value="audi">Audi</option>
            <option value="jeep">Jeep</option>
            <option value="opel">Opel</option>
            <option value="saab">Saab</option>
            <option value="volvo">Volvo</option>
        </select>
    </label>
```

```
</p>

<p>
    Selected Cars: @string.Join(", ", SelectedCars)
</p>

<p>
    <label>
        Select one or more cities:
        <select @bind="SelectedCities" multiple>
            <option value="bal">Baltimore</option>
            <option value="la">Los Angeles</option>
            <option value="pdx">Portland</option>
            <option value="sf">San Francisco</option>
            <option value="sea">Seattle</option>
        </select>
    </label>
</p>

<span>
    Selected Cities: @string.Join(", ", SelectedCities)
</span>

@code {
    public string[] SelectedCars { get; set; } = new string[] { };
    public string[] SelectedCities { get; set; } = new[] { "bal", "sea" };

    private void SelectedCarsChanged(ChangeEventArgs e)
    {
        if (e.Value is not null)
        {
            SelectedCars = (string[])e.Value;
        }
    }
}
```

For information on how empty strings and `null` values are handled in data binding, see the Binding <select> element options to C# object null values section.

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.

- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

Unparsable values

When a user provides an unparsable value to a data-bound element, the unparsable value is automatically reverted to its previous value when the bind event is triggered.

Consider the following component, where an `<input>` element is bound to an `int` type with an initial value of `123`.

`UnparsableValues.razor`:

razor

```razor
@page "/unparsable-values"

<PageTitle>Unparsable Values</PageTitle>

<h1>Unparsable Values Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>

</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private int inputValue = 123;
}
```

Binding applies to the element's `onchange` event. If the user updates the value of the text box's entry to `123.45` and changes the focus, the element's value is reverted to `123` when `onchange` fires. When the value `123.45` is rejected in favor of the original value of `123`, the user understands that their value wasn't accepted.

For the `oninput` event (`@bind:event="oninput"`), a value reversion occurs after any keystroke that introduces an unparsable value. When targeting the `oninput` event with an `int`-bound type, a user is prevented from typing a dot (`.`) character. A dot (`.`) character is immediately removed, so the user receives immediate feedback that only whole numbers are permitted. There are scenarios where reverting the value on the `oninput` event isn't ideal, such as when the user should be allowed to clear an unparsable `<input>` value. Alternatives include:

- Don't use the `oninput` event. Use the default `onchange` event, where an invalid value isn't reverted until the element loses focus.
- Bind to a nullable type, such as `int?` or `string` and either use `@bind:get`/`@bind:set` modifiers (described earlier in this article) or bind to a property with custom get and set accessor logic to handle invalid entries.
- Use an input component, such as InputNumber<TValue> or InputDate<TValue>, with form validation. Input components together with form validation components provide built-in support to manage invalid inputs:
  - Permit the user to provide invalid input and receive validation errors on the associated EditContext.
  - Display validation errors in the UI without interfering with the user entering additional webform data.

Format strings

Data binding works with a single DateTime format string using `@bind:format="{FORMAT STRING}"`, where the `{FORMAT STRING}` placeholder is the format string. Other format expressions, such as currency or number formats, aren't available at this time but might be added in a future release.

`DateBinding.razor`:

razor

```
@page "/date-binding"

<PageTitle>Date Binding</PageTitle>

<h1>Date Binding Example</h1>
```

```razor
<p>
    <label>
        <code>yyyy-MM-dd</code> format:
        <input @bind="startDate" @bind:format="yyyy-MM-dd" />
    </label>
</p>

<p>
    <code>startDate</code>: @startDate
</p>

@code {
    private DateTime startDate = new(2020, 1, 1);
}
```

In the preceding code, the `<input>` element's field type (`type` attribute) defaults to `text`.

Nullable System.DateTime and System.DateTimeOffset are supported:

C#

```
private DateTime? date;
private DateTimeOffset? dateOffset;
```

Specifying a format for the `date` field type isn't recommended because Blazor has built-in support to format dates. In spite of the recommendation, only use the `yyyy-MM-dd` date format for binding to function correctly if a format is supplied with the `date` field type:

razor

```
<input type="date" @bind="startDate" @bind:format="yyyy-MM-dd">
```

Binding with component parameters

A common scenario is binding a property of a child component to a property in its parent component. This scenario is called a *chained bind* because multiple levels of binding occur simultaneously.

You can't implement chained binds with @bind syntax in a child component. An event handler and value must be specified separately to support updating the property in the parent from the child component. The parent component still leverages @bind syntax to set up data binding with the child component.

The following `ChildBind` component has a `Year` component parameter and an EventCallback<TValue>. By convention, the EventCallback<TValue> for the parameter must be named as the component parameter name with a "`Changed`" suffix. The naming syntax is `{PARAMETER NAME}Changed`, where the `{PARAMETER NAME}` placeholder is the parameter name. In the following example, the EventCallback<TValue> is named `YearChanged`.

EventCallback.InvokeAsync invokes the delegate associated with the binding with the provided argument and dispatches an event notification for the changed property.

`ChildBind.razor`:

razor

```
<div class="card bg-light mt-3" style="width:18rem ">
    <div class="card-body">
        <h3 class="card-title">ChildBind Component</h3>
        <p class="card-text">
            Child <code>Year</code>: @Year
        </p>
        <button @onclick="UpdateYearFromChild">Update Year from
Child</button>
    </div>
</div>

@code {
    [Parameter]
    public int Year { get; set; }

    [Parameter]
    public EventCallback<int> YearChanged { get; set; }

    private async Task UpdateYearFromChild() =>
        await YearChanged.InvokeAsync(Random.Shared.Next(1950, 2021));
}
```

For more information on events and EventCallback<TValue>, see the *EventCallback* section of the ASP.NET Core Blazor event handling article.

In the following `Parent1` component, the `year` field is bound to the `Year` parameter of the child component. The `Year` parameter is bindable because it has a companion `YearChanged` event that matches the type of the `Year` parameter.

`Parent1.razor`:

razor

```razor
@page "/parent-1"

<PageTitle>Parent 1</PageTitle>

<h1>Parent Example 1</h1>

<p>Parent <code>year</code>: @year</p>

<button @onclick="UpdateYear">Update Parent <code>year</code></button>

<ChildBind @bind-Year="year" />

@code {
    private int year = 1979;

    private void UpdateYear() => year = Random.Shared.Next(1950, 2021);
}
```

Component parameter binding can also trigger `@bind:after` events. In the following example, the `YearUpdated` method executes asynchronously after binding the `Year` component parameter.

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:after="YearUpdated" />

@code {
    ...

    private async Task YearUpdated()
    {
        ... = await ...;
    }
}
```

By convention, a property can be bound to a corresponding event handler by including an `@bind-{PROPERTY}:event` attribute assigned to the handler, where the `{PROPERTY}` placeholder is the property. `<ChildBind @bind-Year="year" />` is equivalent to writing:

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:event="YearChanged" />
```

In a more sophisticated and real-world example, the following `PasswordEntry` component:

- Sets an `<input>` element's value to a `password` field.

- Exposes changes of a `Password` property to a parent component with an EventCallback that passes in the current value of the child's `password` field as its argument.
- Uses the `onclick` event to trigger the `ToggleShowPassword` method. For more information, see ASP.NET Core Blazor event handling.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
</div>

@code {
    private bool showPassword;
    private string? password;

    [Parameter]
    public string? Password { get; set; }
```

```
    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private async Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        await PasswordChanged.InvokeAsync(password);
    }

    private void ToggleShowPassword() => showPassword = !showPassword;
}
```

The `PasswordEntry` component is used in another component, such as the following `PasswordBinding` component example.

`PasswordBinding.razor`:

razor

```
@page "/password-binding"

<PageTitle>Password Binding</PageTitle>

<h1>Password Binding Example</h1>

<PasswordEntry @bind-Password="password" />

<p>
    <code>password</code>: @password
</p>

@code {
    private string password = "Not set";
}
```

When the `PasswordBinding` component is initially rendered, the `password` value of `Not set` is displayed in the UI. After initial rendering, the value of `password` reflects changes made to the `Password` component parameter value in the `PasswordEntry` component.

> **Note**
>
> The preceding example binds the password one-way from the child `PasswordEntry` component to the parent `PasswordBinding` component. Two-way binding isn't a requirement in this scenario if the goal is for the app to have a shared password entry component for reuse around the app that merely passes the password to the

parent. For an approach that permits two-way binding without writing directly to the child component's parameter, see the `NestedChild` component example in the Bind across more than two components section of this article.

Perform checks or trap errors in the handler. The following revised `PasswordEntry` component provides immediate feedback to the user if a space is used in the password's value.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
            <span class="text-danger">@validationMessage</span>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
</div>

@code {
    private bool showPassword;
    private string? password;
    private string? validationMessage;

    [Parameter]
    public string? Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        if (password != null && password.Contains(' '))
        {
            validationMessage = "Spaces not allowed!";

            return Task.CompletedTask;
```

```
        }
        else
        {
            validationMessage = string.Empty;

            return PasswordChanged.InvokeAsync(password);
        }
    }

    private void ToggleShowPassword() => showPassword = !showPassword;
}
```

Bind across more than two components

You can bind parameters through any number of nested components, but you must respect the one-way flow of data:

- Change notifications *flow up the hierarchy*.
- New parameter values *flow down the hierarchy*.

A common and recommended approach is to only store the underlying data in the parent component to avoid any confusion about what state must be updated, as shown in the following example.

`Parent2.razor`:

razor

```
@page "/parent-2"

<PageTitle>Parent 2</PageTitle>

<h1>Parent Example 2</h1>

<p>Parent Message: <b>@parentMessage</b></p>

<p>
    <button @onclick="ChangeValue">Change from Parent</button>
</p>

<NestedChild @bind-ChildMessage="parentMessage" />

@code {
    private string parentMessage = "Initial value set in Parent";

    private void ChangeValue() => parentMessage = $"Set in Parent
{DateTime.Now}";
}
```

In the following `NestedChild` component, the `NestedGrandchild` component:

- Assigns the value of `ChildMessage` to `GrandchildMessage` with `@bind:get` syntax.
- Updates `GrandchildMessage` when `ChildMessageChanged` executes with `@bind:set` syntax.

`NestedChild.razor`:

razor

```
<div class="border rounded m-1 p-1">
    <h2>Child Component</h2>

    <p>Child Message: <b>@ChildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Child</button>
    </p>

    <NestedGrandchild @bind-GrandchildMessage:get="ChildMessage"
        @bind-GrandchildMessage:set="ChildMessageChanged" />
</div>

@code {
    [Parameter]
    public string? ChildMessage { get; set; }

    [Parameter]
    public EventCallback<string?> ChildMessageChanged { get; set; }

    private async Task ChangeValue() =>
        await ChildMessageChanged.InvokeAsync($"Set in Child
{DateTime.Now}");
}
```

`NestedGrandchild.razor`:

razor

```
<div class="border rounded m-1 p-1">
    <h3>Grandchild Component</h3>

    <p>Grandchild Message: <b>@GrandchildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Grandchild</button>
    </p>
</div>

@code {
    [Parameter]
```

```
    public string? GrandchildMessage { get; set; }

    [Parameter]
    public EventCallback<string> GrandchildMessageChanged { get; set; }

    private async Task ChangeValue() =>
        await GrandchildMessageChanged.InvokeAsync(
            $"Set in Grandchild {DateTime.Now}");
}
```

For an alternative approach suited to sharing data in memory and across components that aren't necessarily nested, see ASP.NET Core Blazor state management.

Bound field or property expression tree

To facilitate deeper interactions with a binding, Blazor allows you to capture of the expression tree of a bound field or property. This is achieved by defining a property with the field or property name suffixed with `Expression`. For any given field or property named `{FIELD OR PROPERTY NAME}`, the corresponding expression tree property is named `{FIELD OR PROPERTY NAME}Expression`.

The following `ChildParameterExpression` component identifies the `Year` expression's model and field name. A FieldIdentifier, which is used to obtain the model and field name, uniquely identifies a single field that can be edited. This may correspond to a property on a model object or can be any other named value. Use of a parameter's expression is useful when creating custom validation components, which isn't covered by the Microsoft Blazor documentation but is addressed by numerous third-party resources.

`ChildParameterExpression.razor`:

razor

```
@using System.Linq.Expressions

<ul>
    <li>Year model: @yearField.Model</li>
    <li>Year field name: @yearField.FieldName</li>
</ul>

@code {
    private FieldIdentifier yearField;

    [Parameter]
    public int Year { get; set; }

    [Parameter]
```

```
    public EventCallback<int> YearChanged { get; set; }

    [Parameter]
    public Expression<Func<int>> YearExpression { get; set; } = default!;

    protected override void OnInitialized() =>
        yearField = FieldIdentifier.Create(YearExpression);
}
```

`Parent3.razor`:

razor

```
@page "/parent-3"

<PageTitle>Parent 3</PageTitle>

<h1>Parent Example 3</h1>

<p>Parent <code>year</code>: @year</p>

<ChildParameterExpression @bind-Year="year" />

@code {
    private int year = 1979;
}
```

Additional resources

- Parameter change detection and additional guidance on Razor component rendering
- ASP.NET Core Blazor forms overview
- Binding to radio buttons in a form
- Binding InputSelect options to C# object null values
- ASP.NET Core Blazor event handling: EventCallback section
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Razor component lifecycle

Article • 12/06/2024

> **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 article explains the ASP.NET Core Razor component lifecycle and how to use lifecycle events.

Lifecycle events

The Razor component processes Razor component lifecycle events in a set of synchronous and asynchronous lifecycle methods. The lifecycle methods can be overridden to perform additional operations in components during component initialization and rendering.

This article simplifies component lifecycle event processing in order to clarify complex framework logic and doesn't cover every change that was made over the years. You may need to access the ComponentBase reference source to integrate custom event processing with Blazor's lifecycle event processing. Code comments in the reference source include additional remarks on lifecycle event processing that don't appear in this article or in the API documentation.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

The following simplified diagrams illustrate Razor component lifecycle event processing. The C# methods associated with the lifecycle events are defined with examples in the following sections of this article.

Component lifecycle events:

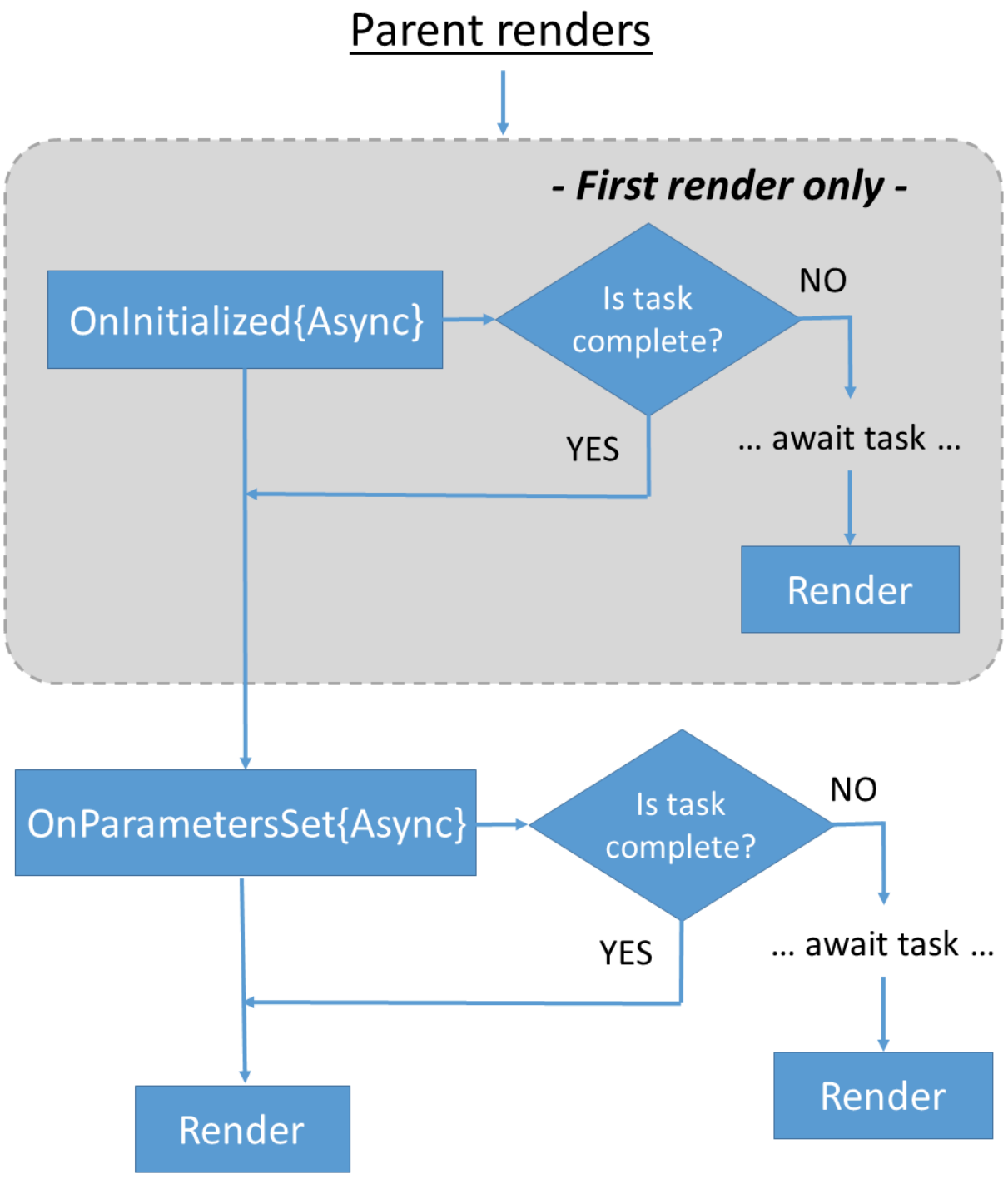
1. If the component is rendering for the first time on a request:

    - Create the component's instance.
    - Perform property injection.
    - Call OnInitialized{Async}. If an incomplete Task is returned, the Task is awaited and then the component is rerendered. The synchronous method is called prior to the asynchronous method.

2. Call OnParametersSet{Async}. If an incomplete Task is returned, the Task is awaited and then the component is rerendered. The synchronous method is called prior to the asynchronous method.
3. Render for all synchronous work and complete Tasks.

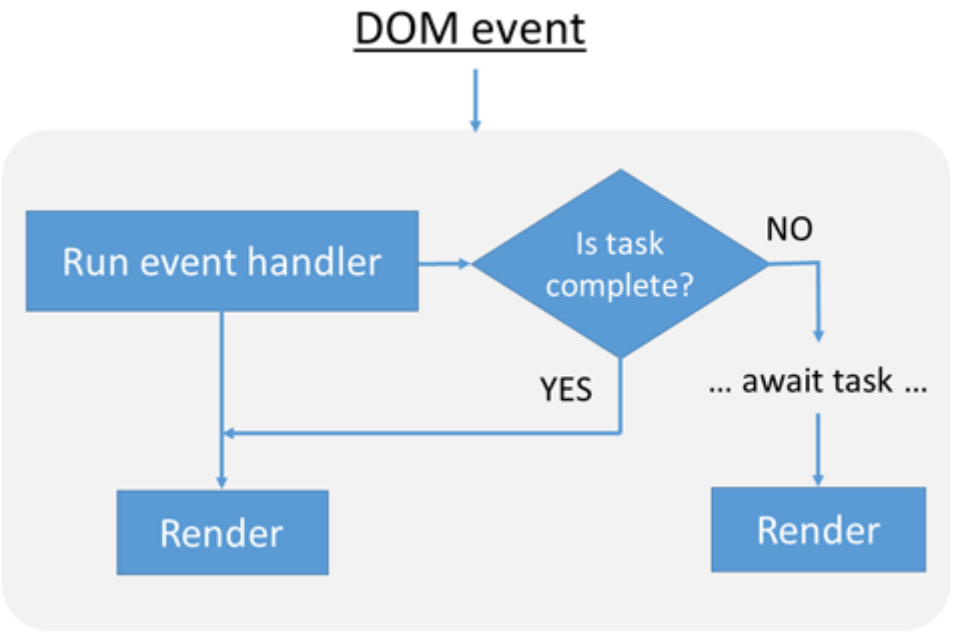
> ⓘ **Note**
>
> Asynchronous actions performed in lifecycle events might delay component rendering or displaying data. For more information, see the **Handle incomplete asynchronous actions at render** section later in this article.

A parent component renders before its children components because rendering is what determines which children are present. If synchronous parent component initialization is used, the parent initialization is guaranteed to complete first. If asynchronous parent component initialization is used, the completion order of parent and child component initialization can't be determined because it depends on the initialization code running.



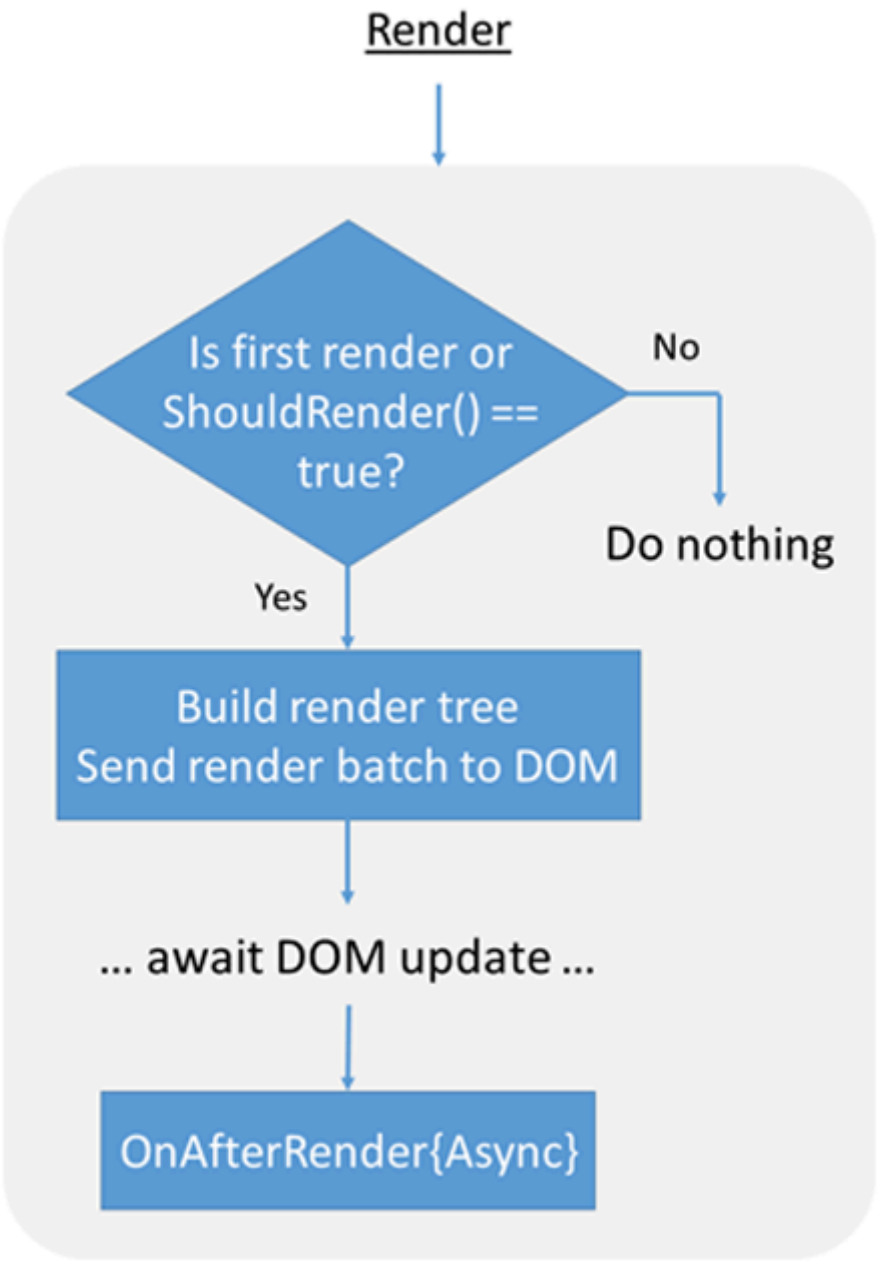
DOM event processing:

1. The event handler is run.
2. If an incomplete Task is returned, the Task is awaited and then the component is rerendered.
3. Render for all synchronous work and complete Tasks.



The `Render` lifecycle:

1. Avoid further rendering operations on the component when both of the following conditions are met:
    - It is not the first render.
    - ShouldRender returns `false`.
2. Build the render tree diff (difference) and render the component.
3. Await the DOM to update.
4. Call OnAfterRender{Async}. The synchronous method is called prior to the asynchronous method.



Developer calls to StateHasChanged result in a rerender. For more information, see ASP.NET Core Razor component rendering.

Quiescence during prerendering

In server-side Blazor apps, prerendering waits for *quiescence*, which means that a component doesn't render until all of the components in the render tree have finished rendering. Quiescence can lead to noticeable delays in rendering when a component performs long-running tasks during initialization and other lifecycle methods, leading to a poor user experience. For more information, see the Handle incomplete asynchronous actions at render section later in this article.

When parameters are set (`SetParametersAsync`)

SetParametersAsync sets parameters supplied by the component's parent in the render tree or from route parameters.

The method's ParameterView parameter contains the set of component parameter values for the component each time SetParametersAsync is called. By overriding the SetParametersAsync method, developer code can interact directly with ParameterView's parameters.

The default implementation of SetParametersAsync sets the value of each property with the [Parameter] or [CascadingParameter] attribute that has a corresponding value in the ParameterView. Parameters that don't have a corresponding value in ParameterView are left unchanged.

Generally, your code should call the base class method (`await base.SetParametersAsync(parameters);`) when overriding SetParametersAsync. In advanced scenarios, developer code can interpret the incoming parameters' values in any way required by not invoking the base class method. For example, there's no requirement to assign the incoming parameters to the properties of the class. However, you must refer to the ComponentBase reference source when structuring your code without calling the base class method because it calls other lifecycle methods and triggers rendering in a complex fashion.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

If you want to rely on the initialization and rendering logic of ComponentBase.SetParametersAsync but not process incoming parameters, you have the option of passing an empty ParameterView to the base class method:

```
C#

await base.SetParametersAsync(ParameterView.Empty);
```

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable and IAsyncDisposable section.

In the following example, ParameterView.TryGetValue assigns the `Param` parameter's value to `value` if parsing a route parameter for `Param` is successful. When `value` isn't `null`, the value is displayed by the component.

Although route parameter matching is case insensitive, TryGetValue only matches case-sensitive parameter names in the route template. The following example requires the use of `/{Param?}` in the route template in order to get the value with TryGetValue, not `/{param?}`. If `/{param?}` is used in this scenario, TryGetValue returns `false` and `message` isn't set to either `message` string.

`SetParamsAsync.razor`:

razor

```razor
@page "/set-params-async/{Param?}"

<PageTitle>Set Parameters Async</PageTitle>

<h1>Set Parameters Async Example</h1>

<p>@message</p>

@code {
    private string message = "Not set";

    [Parameter]
    public string? Param { get; set; }

    public override async Task SetParametersAsync(ParameterView parameters)
    {
        if (parameters.TryGetValue<string>(nameof(Param), out var value))
        {
            if (value is null)
            {
                message = "The value of 'Param' is null.";
            }
            else
            {
                message = $"The value of 'Param' is {value}.";
            }
        }

        await base.SetParametersAsync(parameters);
    }
}
```

Component initialization (`OnInitialized{Async}`)

OnInitialized and OnInitializedAsync are used exclusively to initialize a component for the entire lifetime of the component instance. Parameter values and parameter value changes shouldn't affect the initialization performed in these methods. For example, loading static options into a dropdown list that doesn't change for the lifetime of the component and that isn't dependent on parameter values is performed in one of these lifecycle methods. If parameter values or changes in parameter values affect component state, use OnParametersSet{Async} instead.

These methods are invoked when the component is initialized after having received its initial parameters in SetParametersAsync. The synchronous method is called prior to the asynchronous method.

If synchronous parent component initialization is used, the parent initialization is guaranteed to complete before child component initialization. If asynchronous parent component initialization is used, the completion order of parent and child component initialization can't be determined because it depends on the initialization code running.

For a synchronous operation, override OnInitialized:

`OnInit.razor`:

razor

```razor
@page "/on-init"

<PageTitle>On Initialized</PageTitle>

<h1>On Initialized Example</h1>

<p>@message</p>

@code {
    private string? message;

    protected override void OnInitialized() =>
        message = $"Initialized at {DateTime.Now}";
}
```

To perform an asynchronous operation, override OnInitializedAsync and use the await operator:

C#

```
protected override async Task OnInitializedAsync()
{
    await ...
}
```

If a custom base class is used with custom initialization logic, call OnInitializedAsync on the base class:

C#

```
protected override async Task OnInitializedAsync()
{
    await ...

    await base.OnInitializedAsync();
}
```

It isn't necessary to call ComponentBase.OnInitializedAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

Blazor apps that prerender their content on the server call OnInitializedAsync *twice*:

- Once when the component is initially rendered statically as part of the page.
- A second time when the browser renders the component.

To prevent developer code in OnInitializedAsync from running twice when prerendering, see the Stateful reconnection after prerendering section. The content in the section focuses on Blazor Web Apps and stateful SignalR *reconnection*. To preserve state during the execution of initialization code while prerendering, see Prerender ASP.NET Core Razor components.

While a Blazor app is prerendering, certain actions, such as calling into JavaScript (JS interop), aren't possible. Components may need to render differently when prerendered. For more information, see the Prerendering with JavaScript interop section.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

Use *streaming rendering* with static server-side rendering (static SSR) or prerendering to improve the user experience for components that perform long-running asynchronous tasks in OnInitializedAsync to fully render. For more information, see the following resources:

- Handle incomplete asynchronous actions at render (this article)

- ASP.NET Core Razor component rendering

After parameters are set (`OnParametersSet{Async}`)

OnParametersSet or OnParametersSetAsync are called:

- After the component is initialized in OnInitialized or OnInitializedAsync.
- When the parent component rerenders and supplies:
  - Known or primitive immutable types when at least one parameter has changed.
  - Complex-typed parameters. The framework can't know whether the values of a complex-typed parameter have mutated internally, so the framework always treats the parameter set as changed when one or more complex-typed parameters are present.

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

The synchronous method is called prior to the asynchronous method.

The methods can be invoked even if the parameter values haven't changed. This behavior underscores the need for developers to implement additional logic within the methods to check whether parameter values have indeed changed before re-initializing data or state dependent on those parameters.

For the following example component, navigate to the component's page at a URL:

- With a start date that's received by `StartDate`: `/on-parameters-set/2021-03-19`
- Without a start date, where `StartDate` is assigned a value of the current local time: `/on-parameters-set`

> **Note**
>
> In a component route, it isn't possible to both constrain a DateTime parameter with the route constraint datetime and make the parameter optional. Therefore, the following `OnParamsSet` component uses two @page directives to handle routing with and without a supplied date segment in the URL.

`OnParamsSet.razor`:

```
razor
```