```
        this.http = http;
    }

    public async Task<WeatherForecast[]?> GetForecastAsync(DateTime
startDate) =>
        await http.GetFromJsonAsync<WeatherForecast[]?>("WeatherForecast");
}
```

In the preceding example, the `{APP NAMESPACE}` placeholder is the app's namespace.

`Services/WeatherForecastService.cs` in the server-side Blazor app:

C#

```
using SharedLibrary.Data;
using SharedLibrary.Interfaces;

namespace {APP NAMESPACE}.Services;

public class WeatherForecastService : IWeatherForecastService
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot"
    };

    public async Task<WeatherForecast[]?> GetForecastAsync(DateTime
startDate) =>
        await Task.FromResult(Enumerable.Range(1, 5)
            .Select(index => new WeatherForecast
            {
                Date = startDate.AddDays(index),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = Summaries[Random.Shared.Next(Summaries.Length)]
            }).ToArray());
}
```

In the preceding example, the `{APP NAMESPACE}` placeholder is the app's namespace.

The Blazor Hybrid, Blazor WebAssembly, and server-side Blazor apps register their weather forecast service implementations (`Services.WeatherForecastService`) for `IWeatherForecastService`.

The Blazor WebAssembly project also registers an HttpClient. The HttpClient registered in an app created from the Blazor WebAssembly project template is sufficient for this purpose. For more information, see Call a web API from an ASP.NET Core Blazor app.

`Pages/FetchData.razor` in the RCL:

razor

```razor
@page "/fetchdata"
@inject IWeatherForecastService ForecastService

<PageTitle>Weather forecast</PageTitle>

<h1>Weather forecast</h1>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}
```

Additional resources

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core

- CSS isolation support with Razor class libraries

Pass root component parameters in ASP.NET Core Blazor Hybrid

Article • 02/09/2024

> **ⓘ 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 pass root component parameters in a Blazor Hybrid app.

The `RootComponent` class of a `BlazorWebView` defines a `Parameters` property of type `IDictionary<string, object?>?`, which represents an optional dictionary of parameters to pass to the root component:

- .NET MAUI: Microsoft.AspNetCore.Components.WebView.Maui.RootComponent
- WPF: Microsoft.AspNetCore.Components.WebView.Wpf.RootComponent
- Windows Forms: Microsoft.AspNetCore.Components.WebView.WindowsForms.RootComponent

The following example passes a view model to the root component, which further passes the view model as a cascading type to a Razor component in the Blazor portion of the app. The example is based on the keypad example in the .NET MAUI documentation:

- Data binding and MVVM: Commanding (.NET MAUI documentation): Explains data binding with MVVM using a keypad example.
- .NET MAUI Samples

Although the keypad example focuses on implementing the MVVM pattern in .NET MAUI Blazor Hybrid apps:

- The dictionary of objects passed to root components can include any type for any purpose where you need to pass one or more parameters to the root component for use by Razor components in the app.
- The concepts demonstrated by the following .NET MAUI Blazor example are the same for Windows Forms Blazor apps and WPF Blazor apps.

Place the following view model into your .NET MAUI Blazor Hybrid app.

`KeypadViewModel.cs`:

C#

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace MauiBlazor;

public class KeypadViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _inputString = "";
    private string _displayText = "";
    private char[] _specialChars = { '*', '#' };

    public ICommand AddCharCommand { get; private set; }
    public ICommand DeleteCharCommand { get; private set; }

    public string InputString
    {
        get => _inputString;
        private set
        {
            if (_inputString != value)
            {
                _inputString = value;
                OnPropertyChanged();
                DisplayText = FormatText(_inputString);

                // Perhaps the delete button must be enabled/disabled.
                ((Command)DeleteCharCommand).ChangeCanExecute();
            }
        }
    }

    public string DisplayText
    {
        get => _displayText;
        set
        {
            if (_displayText != value)
            {
                _displayText = value;
                OnPropertyChanged();
            }
        }
    }

    public KeypadViewModel()
```

```
    {
        // Command to add the key to the input string
        AddCharCommand = new Command<string>((key) => InputString += key);

        // Command to delete a character from the input string when allowed
        DeleteCharCommand =
            new Command(
                // Command strips a character from the input string
                () => InputString = InputString.Substring(0,
InputString.Length - 1),

                // CanExecute is processed here to return true when there's
something to delete
                () => InputString.Length > 0
            );
    }

    string FormatText(string str)
    {
        bool hasNonNumbers = str.IndexOfAny(_specialChars) != -1;
        string formatted = str;

        // Format the string based on the type of data and the length
        if (hasNonNumbers || str.Length < 4 || str.Length > 10)
        {
            // Special characters exist, or the string is too small or large
for special formatting
            // Do nothing
        }

        else if (str.Length < 8)
            formatted = string.Format("{0}-{1}", str.Substring(0, 3),
str.Substring(3));

        else
            formatted = string.Format("({0}) {1}-{2}", str.Substring(0, 3),
str.Substring(3, 3), str.Substring(6));

        return formatted;
    }

    public void OnPropertyChanged([CallerMemberName] string name = "") =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

In this article's example, the app's root namespace is `MauiBlazor`. Change the namespace of `KeypadViewModel` to match the app's root namespace:

C#

```
namespace MauiBlazor;
```

> **Note**
>
> At the time the `KeypadViewModel` view model was created for the .NET MAUI sample app and the .NET MAUI documentation, view models were placed in a folder named `ViewModels`, but the namespace was set to the root of the app and didn't include the folder name. If you wish to update the namespace to include the folder in the `KeypadViewModel.cs` file, modify the example code in this article to match. Add `using` (C#) and `@using` (Razor) statements to the following files or fully-qualify the references to the view model type as `{APP NAMESPACE}.ViewModels.KeypadViewModel`, where the `{APP NAMESPACE}` placeholder is the app's root namespace.

Although you can set `Parameters` directly in XAML, the following example names the root component (`rootComponent`) in the XAML file and sets the parameter dictionary in the code-behind file.

In `MainPage.xaml`:

XAML

```
<RootComponent x:Name="rootComponent"
               Selector="#app"
               ComponentType="{x:Type local:Main}" />
```

In the code-behind file (`MainPage.xaml.cs`), assign the view model in the constructor:

C#

```
public MainPage()
{
    InitializeComponent();

    rootComponent.Parameters =
        new Dictionary<string, object>
        {
            { "KeypadViewModel", new KeypadViewModel() }
        };
}
```

The following example cascades the object (`KeypadViewModel`) down component hierarchies in the Blazor portion of the app as a CascadingValue.

In the `Main` component (`Main.razor`):

- Add a parameter matching the type of the object passed to the root component:

  razor

  ```
  @code {
      [Parameter]
      public KeypadViewModel KeypadViewModel { get; set; }
  }
  ```

- Cascade the `KeypadViewModel` with the CascadingValue component. Update the `<Found>` XAML content to the following markup:

  XAML

  ```
  <Found Context="routeData">
      <CascadingValue Value="KeypadViewModel">
          <RouteView RouteData="routeData"
  DefaultLayout="typeof(MainLayout)" />
          <FocusOnNavigate RouteData="routeData" Selector="h1"/>
      </CascadingValue>
  </Found>
  ```

At this point, the cascaded type is available to Razor components throughout the app as a CascadingParameter.

The following `Keypad` component example:

- Displays the current value of `KeypadViewModel.DisplayText`.
- Permits character deletion by calling the `KeypadViewModel.DeleteCharCommand` command if the display string length is greater than 0 (zero), which is checked by the call to ICommand.CanExecute.
- Permits adding characters by calling `KeypadViewModel.AddCharCommand` with the key pressed in the UI.

`Pages/Keypad.razor`:

razor

```
@page "/keypad"

<h1>Keypad</h1>

<table id="keypad">
    <thead>
        <tr>
            <th colspan="2">@KeypadViewModel.DisplayText</th>
            <th><button @onclick="DeleteChar">&#x21E6;</button></th>
```

```
        </tr>
    </thead>
    <tbody>
        <tr>
            <td><button @onclick="@(e => AddChar("1"))">1</button></td>
            <td><button @onclick="@(e => AddChar("2"))">2</button></td>
            <td><button @onclick="@(e => AddChar("3"))">3</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("4"))">4</button></td>
            <td><button @onclick="@(e => AddChar("5"))">5</button></td>
            <td><button @onclick="@(e => AddChar("6"))">6</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("7"))">7</button></td>
            <td><button @onclick="@(e => AddChar("8"))">8</button></td>
            <td><button @onclick="@(e => AddChar("9"))">9</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("*"))">*</button></td>
            <td><button @onclick="@(e => AddChar("0"))">0</button></td>
            <td><button @onclick="@(e => AddChar("#"))">#</button></td>
        </tr>
    </tbody>
</table>

@code {
    [CascadingParameter]
    protected KeypadViewModel KeypadViewModel { get; set; }

    private void DeleteChar()
    {
        if (KeypadViewModel.DeleteCharCommand.CanExecute(null))
        {
            KeypadViewModel.DeleteCharCommand.Execute(null);
        }
    }

    private void AddChar(string key)
    {
        KeypadViewModel.AddCharCommand.Execute(key);
    }
}
```

Purely for demonstration purposes, style the buttons by placing the following CSS styles in the `wwwroot/index.html` file's `<head>` content:

HTML

```
<style>
    #keypad button {
        border: 1px solid black;
        border-radius:6px;
```

```
        height: 35px;
        width:80px;
    }
</style>
```

Create a sidebar navigation entry in the `NavMenu` component (`Shared/NavMenu.razor`) with the following markup:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="keypad">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Keypad
    </NavLink>
</div>
```

Additional resources

- Host a Blazor web app in a .NET MAUI app using BlazorWebView
- Data binding and MVVM: Commanding (.NET MAUI documentation)
- ASP.NET Core Blazor cascading values and parameters

ASP.NET Core Blazor Hybrid authentication and authorization

Article • 08/09/2024

> **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 ASP.NET Core's support for the configuration and management of security and ASP.NET Core Identity in Blazor Hybrid apps.

Authentication in Blazor Hybrid apps is handled by native platform libraries, as they offer enhanced security guarantees that the browser sandbox can't offer. Authentication of native apps uses an OS-specific mechanism or via a federated protocol, such as OpenID Connect (OIDC). Follow the guidance for the identity provider that you've selected for the app and then further integrate identity with Blazor using the guidance in this article.

Integrating authentication must achieve the following goals for Razor components and services:

- Use the abstractions in the Microsoft.AspNetCore.Components.Authorization package, such as AuthorizeView.
- React to changes in the authentication context.
- Access credentials provisioned by the app from the identity provider, such as access tokens to perform authorized API calls.

After authentication is added to a .NET MAUI, WPF, or Windows Forms app and users are able to log in and log out successfully, integrate authentication with Blazor to make the authenticated user available to Razor components and services. Perform the following steps:

- Reference the Microsoft.AspNetCore.Components.Authorization package.

> **Note**

> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

- Implement a custom AuthenticationStateProvider, which is the abstraction that Razor components use to access information about the authenticated user and to receive updates when the authentication state changes.
- Register the custom authentication state provider in the dependency injection container.

.NET MAUI apps use Xamarin.Essentials: Web Authenticator: The `WebAuthenticator` class allows the app to initiate browser-based authentication flows that listen for a callback to a specific URL registered with the app.

For additional guidance, see the following resources:

- Web authenticator (.NET MAUI documentation)
- Sample.Server.WebAuthenticator sample app

Create a custom `AuthenticationStateProvider` without user change updates

If the app authenticates the user immediately after the app launches and the authenticated user remains the same for the entirety of the app lifetime, user change notifications aren't required, and the app only provides information about the authenticated user. In this scenario, the user logs into the app when the app is opened, and the app displays the login screen again after the user logs out. The following `ExternalAuthStateProvider` is an example implementation of a custom AuthenticationStateProvider for this authentication scenario.

> **Note**
>
> The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

```
C#
```

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
    private readonly Task<AuthenticationState> authenticationState;

    public ExternalAuthStateProvider(AuthenticatedUser user) =>
        authenticationState = Task.FromResult(new
AuthenticationState(user.Principal));

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        authenticationState;
}

public class AuthenticatedUser
{
    public ClaimsPrincipal Principal { get; set; } = new();
}
```

The following steps describe how to:

- Add required namespaces.
- Add the authorization services and Blazor abstractions to the service collection.
- Build the service collection.
- Resolve the `AuthenticatedUser` service to set the authenticated user's claims principal. See your identity provider's documentation for details.
- Return the built host.

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add namespaces for Microsoft.AspNetCore.Components.Authorization and System.Security.Claims:

C#

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

Remove the following line of code that returns a built Microsoft.Maui.Hosting.MauiApp:

diff

```
- return builder.Build();
```

Replace the preceding line of code with the following code. Add OpenID/MSAL code to authenticate the user. See your identity provider's documentation for details.

C#

```
builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
builder.Services.AddSingleton<AuthenticatedUser>();
var host = builder.Build();

var authenticatedUser = host.Services.GetRequiredService<AuthenticatedUser>
();

/*
Provide OpenID/MSAL code to authenticate the user. See your identity
provider's
documentation for details.

The user is represented by a new ClaimsPrincipal based on a new
ClaimsIdentity.
*/
var user = new ClaimsPrincipal(new ClaimsIdentity());

authenticatedUser.Principal = user;

return host;
```

Create a custom `AuthenticationStateProvider` with user change updates

To update the user while the Blazor app is running, call NotifyAuthenticationStateChanged within the AuthenticationStateProvider implementation using ***either*** of the following approaches:

- Signal an authentication update from outside of the BlazorWebView)
- Handle authentication within the BlazorWebView

Signal an authentication update from outside of the `BlazorWebView` (Option 1)

A custom AuthenticationStateProvider can use a global service to signal an authentication update. We recommend that the service offer an event that the AuthenticationStateProvider can subscribe to, where the event invokes NotifyAuthenticationStateChanged.

> ⓘ Note

The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

C#

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
    private AuthenticationState currentUser;

    public ExternalAuthStateProvider(ExternalAuthService service)
    {
        currentUser = new AuthenticationState(service.CurrentUser);

        service.UserChanged += (newUser) =>
        {
            currentUser = new AuthenticationState(newUser);
            NotifyAuthenticationStateChanged(Task.FromResult(currentUser));
        };
    }

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(currentUser);
}

public class ExternalAuthService
{
    public event Action<ClaimsPrincipal>? UserChanged;
    private ClaimsPrincipal? currentUser;

    public ClaimsPrincipal CurrentUser
    {
        get { return currentUser ?? new(); }
        set
        {
            currentUser = value;

            if (UserChanged is not null)
            {
                UserChanged(currentUser);
            }
        }
```

```
    }
}
```

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add a namespace for Microsoft.AspNetCore.Components.Authorization:

C#

```
using Microsoft.AspNetCore.Components.Authorization;
```

Add the authorization services and Blazor abstractions to the service collection:

C#

```
builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
builder.Services.AddSingleton<ExternalAuthService>();
```

Wherever the app authenticates a user, resolve the `ExternalAuthService` service:

C#

```
var authService = host.Services.GetRequiredService<ExternalAuthService>();
```

Execute your custom OpenID/MSAL code to authenticate the user. See your identity provider's documentation for details. The authenticated user (`authenticatedUser` in the following example) is a new ClaimsPrincipal based on a new ClaimsIdentity.

Set the current user to the authenticated user:

C#

```
authService.CurrentUser = authenticatedUser;
```

An alternative to the preceding approach is to set the user's principal on System.Threading.Thread.CurrentPrincipal instead of setting it via a service, which avoids use of the dependency injection container:

C#

```
public class CurrentThreadUserAuthenticationStateProvider :
AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
```

```
=>
        Task.FromResult(
            new AuthenticationState(Thread.CurrentPrincipal as
ClaimsPrincipal ??
                new ClaimsPrincipal(new ClaimsIdentity())));
}
```

Using the alternative approach, only authorization services (AddAuthorizationCore) and `CurrentThreadUserAuthenticationStateProvider` (`.TryAddScoped<AuthenticationStateProvider, CurrentThreadUserAuthenticationStateProvider>()`) are added to the service collection.

Handle authentication within the `BlazorWebView` (Option 2)

A custom AuthenticationStateProvider can include additional methods to trigger log in and log out and update the user.

> **Note**
>
> The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

C#

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
    private ClaimsPrincipal currentUser = new ClaimsPrincipal(new
ClaimsIdentity());

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(new AuthenticationState(currentUser));

    public Task LogInAsync()
    {
        var loginTask = LogInAsyncCore();
        NotifyAuthenticationStateChanged(loginTask);
```

```
        return loginTask;

        async Task<AuthenticationState> LogInAsyncCore()
        {
            var user = await LoginWithExternalProviderAsync();
            currentUser = user;

            return new AuthenticationState(currentUser);
        }
    }

    private Task<ClaimsPrincipal> LoginWithExternalProviderAsync()
    {
        /*
            Provide OpenID/MSAL code to authenticate the user. See your
identity
            provider's documentation for details.

            Return a new ClaimsPrincipal based on a new ClaimsIdentity.
        */
        var authenticatedUser = new ClaimsPrincipal(new ClaimsIdentity());

        return Task.FromResult(authenticatedUser);
    }

    public void Logout()
    {
        currentUser = new ClaimsPrincipal(new ClaimsIdentity());
        NotifyAuthenticationStateChanged(
            Task.FromResult(new AuthenticationState(currentUser)));
    }
}
```

In the preceding example:

- The call to `LogInAsyncCore` triggers the login process.
- The call to NotifyAuthenticationStateChanged notifies that an update is in progress, which allows the app to provide a temporary UI during the login or logout process.
- Returning `loginTask` returns the task so that the component that triggered the login can await and react after the task is complete.
- The `LoginWithExternalProviderAsync` method is implemented by the developer to log in the user with the identity provider's SDK. For more information, see your identity provider's documentation. The authenticated user (`authenticatedUser`) is a new ClaimsPrincipal based on a new ClaimsIdentity.

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add the authorization services and the Blazor abstraction to the service collection:

C#

```
builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

The following `LoginComponent` component demonstrates how to log in a user. In a typical app, the `LoginComponent` component is only shown in a parent component if the user isn't logged into the app.

`Shared/LoginComponent.razor`:

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<button @onclick="Login">Log in</button>

@code
{
    public async Task Login()
    {
        await ((ExternalAuthStateProvider)AuthenticationStateProvider)
            .LogInAsync();
    }
}
```

The following `LogoutComponent` component demonstrates how to log out a user. In a typical app, the `LogoutComponent` component is only shown in a parent component if the user is logged into the app.

`Shared/LogoutComponent.razor`:

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<button @onclick="Logout">Log out</button>

@code
{
    public async Task Logout()
    {
        await ((ExternalAuthStateProvider)AuthenticationStateProvider)
            .Logout();
    }
}
```

Accessing other authentication information

Blazor doesn't define an abstraction to deal with other credentials, such as access tokens to use for HTTP requests to web APIs. We recommend following the identity provider's guidance to manage the user's credentials with the primitives that the identity provider's SDK provides.

It's common for identity provider SDKs to use a token store for user credentials stored in the device. If the SDK's token store primitive is added to the service container, consume the SDK's primitive within the app.

The Blazor framework isn't aware of a user's authentication credentials and doesn't interact with credentials in any way, so the app's code is free to follow whatever approach you deem most convenient. However, follow the general security guidance in the next section, Other authentication security considerations, when implementing authentication code in an app.

Other authentication security considerations

The authentication process is external to Blazor, and we recommend that developers access the identity provider's guidance for additional security guidance.

When implementing authentication:

- Avoid authentication in the context of the Web View. For example, avoid using a JavaScript OAuth library to perform the authentication flow. In a single-page app, authentication tokens aren't hidden in JavaScript and can be easily discovered by malicious users and used for nefarious purposes. Native apps don't suffer this risk because native apps are only able to obtain tokens outside of the browser context, which means that rogue third-party scripts can't steal the tokens and compromise the app.
- Avoid implementing the authentication workflow yourself. In most cases, platform libraries securely handle the authentication workflow, using the system's browser instead of using a custom Web View that can be hijacked.
- Avoid using the platform's Web View control to perform authentication. Instead, rely on the system's browser when possible.
- Avoid passing the tokens to the document context (JavaScript). In some situations, a JavaScript library within the document is required to perform an authorized call to an external service. Instead of making the token available to JavaScript via JS interop:
  - Provide a generated temporary token to the library and within the Web View.
  - Intercept the outgoing network request in code.

    - Replace the temporary token with the real token and confirm that the destination of the request is valid.

Additional resources

- ASP.NET Core Blazor authentication and authorization
- ASP.NET Core Blazor Hybrid security considerations
- Entra ID documentation for .NET MAUI
    - Tutorial: Register and configure .NET MAUI mobile app in an external tenant
    - Sign in users in a sample .NET MAUI Android application
    - Sign in users in a sample .NET MAUI desktop application
- Azure documentation for .NET MAUI
    - Add authentication to your .NET MAUI app
    - MAUI mobile or desktop application using Microsoft Entra ID for authentication

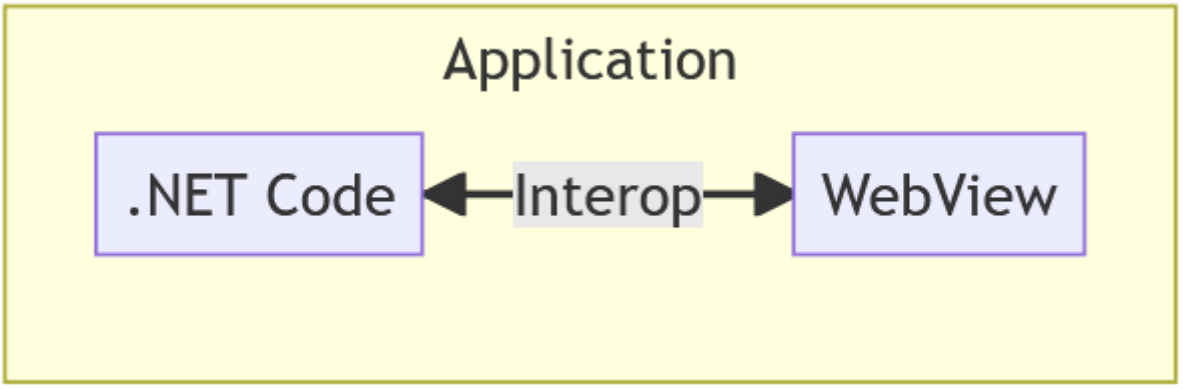
ASP.NET Core Blazor Hybrid security considerations

Article • 09/27/2024

> **ⓘ 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 security considerations for Blazor Hybrid apps.

Blazor Hybrid apps that render web content execute .NET code inside a platform Web View. The .NET code interacts with the web content via an interop channel between the .NET code and the Web View.



The web content rendered into the Web View can come from assets provided by the app from either of the following locations:

- The `wwwroot` folder in the app.
- A source external to the app. For example, a network source, such as the Internet.

A trust boundary exists between the .NET code and the code that runs inside the Web View. .NET code is provided by the app and any trusted third-party packages that you've installed. After the app is built, the .NET code Web View content sources can't change.

In contrast to the .NET code sources of content, content sources from the code that runs inside the Web View can come not only from the app but also from external sources. For example, static assets from an external Content Delivery Network (CDN) might be used or rendered by an app's Web View.

Consider the code inside the Web View as **untrusted** in the same way that code running inside the browser for a web app isn't trusted. The same threats and general security recommendations apply to untrusted resources in Blazor Hybrid apps as for other types of apps.

If possible, avoid loading content from a third-party origin. To mitigate risk, you might be able to serve content directly from the app by downloading the external assets, verifying that they're safe to serve to users, and placing them into the app's `wwwroot` folder for packaging with the rest of the app. When the external content is downloaded for inclusion in the app, we recommend scanning it for viruses and malware before placing it into the `wwwroot` folder of the app.

If your app must reference content from an external origin, we recommend that you use common web security approaches to provide the app with an opportunity to block the content from loading if the content is compromised:

- Serve content securely with TLS/HTTPS.
- Institute a Content Security Policy (CSP).
- Perform subresource integrity checks.

Even if all of the resources are packed into the app and don't load from any external origin, remain cautious about problems in the resources' code that run inside the Web View, as the resources might have vulnerabilities that could allow cross-site scripting (XSS) attacks.

In general, the Blazor framework protects against XSS by dealing with HTML in safe ways. However, some programming patterns allow Razor components to inject raw HTML into rendered output, such as rendering content from an untrusted source. For example, rendering HTML content directly from a database should be avoided. Additionally, JavaScript libraries used by the app might manipulate HTML in unsafe ways to inadvertently or deliberately render unsafe output.

For these reasons, it's best to apply the same protections against XSS that are normally applied to web apps. Prevent loading scripts from unknown sources and don't implement potentially unsafe JavaScript features, such as eval and other unsafe JavaScript primitives. Establishing a CSP is recommended to reduce these security risks.

If the code inside the Web View is compromised, the code gains access to all of the content inside the Web View and might interact with the host via the interop channel. For that reason, any content coming from the Web View (events, JS interop) must be treated as **untrusted** and validated in the same way as for other sensitive contexts, such as in a compromised Blazor Server app that can lead to malicious attacks on the host system.

Don't store sensitive information, such as credentials, security tokens, or sensitive user data, in the context of the Web View, as it makes the information available to a cyberattacker if the Web View is compromised. There are safer alternatives, such as handling the sensitive information directly within the native portion of the app.

External content rendered in an `iframe`

When using an iframe to display external content within a Blazor Hybrid page, we recommend that users leverage sandboxing features to ensure that the content is isolated from the parent page containing the app. In the following Razor component example, the sandbox attribute is present for the `<iframe>` tag to apply sandboxing features to the `admin.html` page:

razor

```razor
<iframe sandbox src="https://contoso.com/admin.html" />
```

> ⚠ **Warning**
>
> The sandbox attribute isn't supported in early browser versions. For more information, see Can I use: sandbox.

Links to external URLs

Links to URLs outside of the app are opened in an appropriate external app, not loaded within the Web View. We don't recommend overriding the default behavior.

Keep the Web View current in deployed apps

The BlazorWebView control uses the currently-installed, platform-specific native Web View. Since the native Web View is periodically updated with support for new APIs and fixes for security issues, it may be necessary to ensure that an app is using a Web View version that meets the app's requirements.

Use one of the following approaches to keep the Web View current in deployed apps:

- **On all platforms**: Check the Web View version and prompt the user to take any necessary steps to update it.
- **Only on Windows**: Package a fixed-version Web View within the app, using it in place of the system's shared Web View.

Android

The Android Web View is distributed and updated via the Google Play Store. Check the Web View version by reading the User-Agent string. Read the Web View's navigator.userAgent property using JavaScript interop and optionally cache the value using a singleton service if the user agent string is required outside of a Razor component context.

When using the Android Emulator:

- Use an emulated device with **Google Play Services** preinstalled. Emulated devices without Google Play Services preinstalled aren't supported.
- Install Google Chrome from the Google Play Store. If Google Chrome is already installed, update Chrome from the Google Play Store. If an emulated device doesn't have the latest version of Chrome installed, it might not have the latest version of the Android Web View installed.

iOS/Mac Catalyst

iOS and Mac Catalyst both use WKWebView, a Safari-based control, which is updated by the operating system. Similar to the Android case, determine the Web View version by reading the Web View's User-Agent string.

Windows (.NET MAUI, WPF, Windows Forms)

On Windows, the Chromium-based Microsoft Edge WebView2 is required to run Blazor web apps.

The newest installed version of `WebView2`, known as the *Evergreen distribution*, is used. If you wish to ship a specific version of `WebView2` with the app, use the *Fixed Version distribution*.

For more information on checking the currently-installed `WebView2` version and the distribution modes, see the WebView2 distribution documentation.

Additional resources

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor authentication and authorization

Publish ASP.NET Core Blazor Hybrid apps

Article • 02/09/2024

> **ⓘ 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 publish Blazor Hybrid apps.

Publish for a specific framework

Blazor Hybrid supports .NET MAUI, WPF, and Windows Forms. The publishing steps for apps using Blazor Hybrid are nearly identical to the publishing steps for the target platform.

- WPF and Windows Forms
  - .NET application publishing overview
- .NET MAUI
  - Windows
  - Android
  - iOS
  - macOS

Blazor-specific considerations

Blazor Hybrid apps require a Web View on the host platform. For more information, see Keep the Web View current in deployed Blazor Hybrid apps.

Troubleshoot ASP.NET Core Blazor Hybrid

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.

BlazorWebView has built-in logging that can help you diagnose problems in your Blazor Hybrid app.

This article explains the steps to use BlazorWebView logging:

- Enable BlazorWebView and related components to log diagnostic information.
- Configure logging providers.
- View logger output.

Enable `BlazorWebView` logging

Enable logging configuration during service registration. To enable maximum logging for BlazorWebView and related components under the Microsoft.AspNetCore.Components.WebView namespace, add the following code in the `Program` file:

C#

```
services.AddLogging(logging =>
{
    logging.AddFilter("Microsoft.AspNetCore.Components.WebView",
LogLevel.Trace);
});
```

Alternatively, use the following code to enable maximum logging for every component that uses Microsoft.Extensions.Logging:

C#

```
services.AddLogging(logging =>
{
    logging.SetMinimumLevel(LogLevel.Trace);
});
```

Configure logging providers

After configuring components to write log information, configure where the loggers should write log information.

The **Debug** logging providers write the output using Debug statements.

To configure the **Debug** logging provider, add a reference to the Microsoft.Extensions.Logging.Debug NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Register the provider inside the call to AddLogging added in the previous step by calling the AddDebug extension method:

C#

```
services.AddLogging(logging =>
{
    logging.AddFilter("Microsoft.AspNetCore.Components.WebView",
LogLevel.Trace);
    logging.AddDebug();
});
```

View logger output

When the app is run from Visual Studio with debugging enabled, the debug output appears in Visual Studio's **Output** window.

Additional resources

- Logging in C# and .NET

- Logging in .NET Core and ASP.NET Core

ASP.NET Core Blazor project structure

Article • 09/12/2024

> **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 the files and folders that make up a Blazor app generated from a Blazor project template.

Blazor Web App

Blazor Web App project template: `blazor`

The Blazor Web App project template provides a single starting point for using Razor components (`.razor`) to build any style of web UI, both server-side rendered and client-side rendered. It combines the strengths of the existing Blazor Server and Blazor WebAssembly hosting models with server-side rendering, streaming rendering, enhanced navigation and form handling, and the ability to add interactivity using either Blazor Server or Blazor WebAssembly on a per-component basis.

If both client-side rendering (CSR) and interactive server-side rendering (interactive SSR) are selected on app creation, the project template uses the Interactive Auto render mode. The automatic rendering mode initially uses interactive SSR while the .NET app bundle and runtime are downloaded to the browser. After the .NET WebAssembly runtime is activated, rendering switches to CSR.

The Blazor Web App template enables both static and interactive server-side rendering using a single project. If you also enable Interactive WebAssembly rendering, the project includes an additional client project (`.Client`) for your WebAssembly-based components. The built output from the client project is downloaded to the browser and executed on the client. Components using the Interactive WebAssembly or Interactive Auto render modes must be located in the `.Client` project.

The component folder structure of the `.Client` project differs from the Blazor Web App's main project folder structure because the main project is a standard ASP.NET Core

project. The main project must take into account other assets for ASP.NET Core projects that are unrelated to Blazor. You're welcome to use whatever component folder structure you wish in the `.Client` project. You're free to mirror the component folder layout of the main project in the `.Client` project if you wish. Note that namespaces might require adjustments for such assets as layout files if you move components into different folders than the project template uses.

More information on components and render modes is found in the ASP.NET Core Razor components and ASP.NET Core Blazor render modes articles.

Based on the interactive render mode selected at app creation, the `Layout` folder is either in the server project in the `Components` folder or at the root of the `.Client` project. The folder contains the following layout components and stylesheets:

- The `MainLayout` component (`MainLayout.razor`) is the app's layout component.
- The `MainLayout.razor.css` is the stylesheet for the app's main layout.
- The `NavMenu` component (`NavMenu.razor`) implements sidebar navigation. The component includes NavLink components (NavLink), which render navigation links to other Razor components. The NavLink component indicates to the user which component is currently displayed.
- The `NavMenu.razor.css` is the stylesheet for the app's navigation menu.

The `Routes` component (`Routes.razor`) is either in the server project or the `.Client` project and sets up routing using the Router component. For client-side interactive components, the Router component intercepts browser navigation and renders the page that matches the requested address.

The `Components` folder of the server project holds the app's server-side Razor components. Shared components are often placed at the root of the `Components` folder, while layout and page components are usually placed in folders within the `Components` folder.

The `Components/Pages` folder of the server project contains the app's routable server-side Razor components. The route for each page is specified using the @page directive.

The `App` component (`App.razor`) is the root component of the app with HTML `<head>` markup, the `Routes` component, and the Blazor `<script>` tag. The root component is the first component that the app loads.

An `_Imports.razor` file in each of the server and `.Client` projects includes common Razor directives for Razor components of either project, such as @using directives for namespaces.

The `Properties` folder of the server project holds development environment configuration in the `launchSettings.json` file.

> **Note**
>
> The `http` profile precedes the `https` profile in the `launchSettings.json` file. When an app is run with the .NET CLI, the app runs at an HTTP endpoint because the first profile found is `http`. The profile order eases the transition of adopting HTTPS for Linux and macOS users. If you prefer to start the app with the .NET CLI without having to pass the `-lp https` or `--launch-profile https` option to the `dotnet watch` (or `dotnet run`) command, simply place the `https` profile above the `http` profile in the file.

The `wwwroot` folder of the server project is the Web Root folder for the server project that holds the app's public static assets.

The `Program.cs` file of the server project is the project's entry point that sets up the ASP.NET Core web application host and contains the app's startup logic, including service registrations, configuration, logging, and request processing pipeline:

- Services for Razor components are added by calling AddRazorComponents. 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`). AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) for the app. AddInteractiveWebAssemblyRenderMode configures the Interactive WebAssembly render mode for the app.

The app settings files (`appsettings.Development.json`, `appsettings.json`) in either the server or `.Client` project provide configuration settings. In the server project, settings files are at the root of the project. In the `.Client` project, settings files are consumed from the Web Root folder, `wwwroot`.

In the `.Client` project:

- The `Pages` folder contains routable client-side Razor components. The route for each page is specified using the @page directive.

- The `wwwroot` folder is the Web Root folder for the `.Client` project that holds the app's public static assets.
- The `Program.cs` file is the project's entry point that sets up the WebAssembly host and contains the project's startup logic, including service registrations, configuration, logging, and request processing pipeline.

Additional files and folders may appear in an app produced from a Blazor Web App project template when additional options are configured. For example, generating an app with ASP.NET Core Identity includes additional assets for authentication and authorization features.

Standalone Blazor WebAssembly

Standalone Blazor WebAssembly project template: `blazorwasm`

The Blazor WebAssembly template creates the initial files and directory structure for a standalone Blazor WebAssembly app:

- If the `blazorwasm` template is used, the app is populated with the following:
  - Demonstration code for a `Weather` component that loads data from a static asset (`weather.json`) and user interaction with a `Counter` component.
  - Bootstrap frontend toolkit.
- The `blazorwasm` template can also be generated without sample pages and styling.

Project structure:

- `Layout` folder: Contains the following layout components and stylesheets:
  - `MainLayout` component (`MainLayout.razor`): The app's layout component.
  - `MainLayout.razor.css`: Stylesheet for the app's main layout.
  - `NavMenu` component (`NavMenu.razor`): Implements sidebar navigation. Includes the NavLink component (NavLink), which renders navigation links to other Razor components. The NavLink component automatically indicates a selected state when its component is loaded, which helps the user understand which component is currently displayed.
  - `NavMenu.razor.css`: Stylesheet for the app's navigation menu.
- `Pages` folder: Contains the Blazor app's routable Razor components (`.razor`). The route for each page is specified using the @page directive. The template includes the following components:
  - `Counter` component (`Counter.razor`): Implements the Counter page.
  - `Index` component (`Index.razor`): Implements the Home page.

    - `Weather` component (`Weather.razor`): Implements the Weather page.

- `_Imports.razor`: Includes common Razor directives to include in the app's components (`.razor`), such as @using directives for namespaces.

- `App.razor`: The root component of the app that sets up client-side routing using the Router component. The Router component intercepts browser navigation and renders the page that matches the requested address.

- `Properties` folder: Holds development environment configuration in the `launchSettings.json` file.

  > **Note**
  >
  > The `http` profile precedes the `https` profile in the `launchSettings.json` file. When an app is run with the .NET CLI, the app runs at an HTTP endpoint because the first profile found is `http`. The profile order eases the transition of adopting HTTPS for Linux and macOS users. If you prefer to start the app with the .NET CLI without having to pass the `-lp https` or `--launch-profile https` option to the `dotnet watch` (or `dotnet run`) command, simply place the `https` profile above the `http` profile in the file.

- `wwwroot` folder: The Web Root folder for the app containing the app's public static assets, including `appsettings.json` and environmental app settings files for configuration settings and sample weather data (`sample-data/weather.json`). The `index.html` webpage is the root page of the app implemented as an HTML page:
    - When any page of the app is initially requested, this page is rendered and returned in the response.
    - The page specifies where the root `App` component is rendered. The component is rendered at the location of the `div` DOM element with an `id` of `app` (`<div id="app">Loading...</div>`).

- `Program.cs`: The app's entry point that sets up the WebAssembly host:
    - The `App` component is the root component of the app. The `App` component is specified as the `div` DOM element with an `id` of `app` (`<div id="app">Loading...</div>` in `wwwroot/index.html`) to the root component collection (`builder.RootComponents.Add<App>("#app")`).
    - Services are added and configured (for example, `builder.Services.AddSingleton<IMyDependency, MyDependency>()`).

Additional files and folders may appear in an app produced from a Blazor WebAssembly project template when additional options are configured. For example, generating an app with ASP.NET Core Identity includes additional assets for authentication and authorization features.

Location of the Blazor script

The Blazor script is served from an embedded resource in the ASP.NET Core shared framework.

In a Blazor Web App, the Blazor script is located in the `Components/App.razor` file:

```
HTML
<script src="_framework/blazor.web.js"></script>
```

In a Blazor Server app, the Blazor script is located in the `Pages/_Host.cshtml` file:

```
HTML
<script src="_framework/blazor.server.js"></script>
```

In a Blazor WebAssembly app, the Blazor script content is located in the `wwwroot/index.html` file:

```
HTML
<script src="_framework/blazor.webassembly.js"></script>
```

Location of `<head>` and `<body>` content

In a Blazor Web App, `<head>` and `<body>` content is located in the `Components/App.razor` file.

In a Blazor WebAssembly app, `<head>` and `<body>` content is located in the `wwwroot/index.html` file.

Additional resources

- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models

- Minimal APIs overview
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor fundamentals

Article • 11/19/2024

> **ⓘ 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**.

Fundamentals articles provide guidance on foundational Blazor concepts. Some of the concepts are connected to a basic understanding of *Razor components*, which are described further in the next section of this article and covered in detail in the *Components* articles.

Static and interactive rendering concepts

Razor components are either *statically* rendered or *interactively* rendered.

Static or *static rendering* is a server-side scenario that means the component is rendered without the capacity for interplay between the user and .NET/C# code. JavaScript and HTML DOM events remain unaffected, but no user events on the client can be processed with .NET running on the server.

Interactive or *interactive rendering* means that the component has the capacity to process .NET events via C# code. The .NET events are either processed on the server by the ASP.NET Core runtime or in the browser on the client by the WebAssembly-based Blazor runtime.

> **ⓘ 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 these concepts and how to control static and interactive rendering is found in the ASP.NET Core Blazor render modes article later in the Blazor

documentation.

Client and server rendering concepts

Throughout the Blazor documentation, activity that takes place on the user's system is said to occur *on the client* or *client-side*. Activity that takes place on a server is said to occur *on the server* or *server-side*.

The term *rendering* means to produce the HTML markup that browsers display.

- **Client-side rendering (CSR)** means that the final HTML markup is generated by the .NET WebAssembly runtime on the client. No HTML for the app's client-generated UI is sent from a server to the client for this type of rendering. User interactivity with the page is assumed. There's no such concept as *static* client-side rendering. CSR is assumed to be interactive, so "*interactive* client-side rendering" and "*interactive* CSR" aren't used by the industry or in the Blazor documentation.

- **Server-side rendering (SSR)** means that the final HTML markup is generated by the ASP.NET Core runtime on the server. The HTML is sent to the client over a network for display by the client's browser. No HTML for the app's server-generated UI is created by the client for this type of rendering. SSR can be of two varieties:
  - **Static SSR**: The server produces static HTML that doesn't provide for user interactivity or maintaining Razor component state.
  - **Interactive SSR**: Blazor events permit user interactivity and Razor component state is maintained by the Blazor framework.

- **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 always followed by final rendering, either on the server or the client.

Razor components

Blazor apps are based on *Razor components*, often referred to as just *components*. A *component* is an element of UI, such as a page, dialog, or data entry form. Components are .NET C# classes built into .NET assemblies.

Razor refers to how components are usually written in the form of a Razor markup page for client-side UI logic and composition. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor files use the `.razor` file extension.

Although some Blazor developers and online resources use the term "Blazor components," the documentation avoids that term and universally uses "Razor components" or "components."

Blazor documentation adopts several conventions for showing and discussing components:

- Generally, examples adhere to ASP.NET Core/C# coding conventions and engineering guidelines. For more information see the following resources:
  - ASP.NET Core framework engineering guidelines (dotnet/aspnetcore GitHub repository)
  - C# Coding Conventions (C# guide)
- Project code, file paths and names, project template names, and other specialized terms are in United States English and usually code-fenced.
- Components are usually referred to by their C# class name (Pascal case) followed by the word "component." For example, a typical file upload component is referred to as the "`FileUpload` component."
- Usually, a component's C# class name is the same as its file name.
- Routable components usually set their relative URLs to the component's class name in kebab-case. For example, a `FileUpload` component includes routing configuration to reach the rendered component at the relative URL `/file-upload`. Routing and navigation is covered in ASP.NET Core Blazor routing and navigation.
- When multiple versions of a component are used, they're numbered sequentially. For example, the `FileUpload3` component is reached at `/file-upload-3`.
- Razor directives at the top of a component definition (`.razor file`) are placed in the following order: `@page`, `@rendermode` (.NET 8 or later), `@using` statements, other directives in alphabetical order.
- Although not required for `private` members, access modifiers are used in article examples and sample apps. For example, `private` is stated for declaring a field named `maxAllowedFiles` as `private int maxAllowedFiles = 3;`.
- Component parameter values lead with a Razor reserved @ symbol, but it isn't required. Literals (for example, boolean values), keywords (for example, `this`), and `null` as component parameter values aren't prefixed with `@`, but this is also merely a documentation convention. Your own code can prefix literals with `@` if you wish.
- C# classes use the this keyword and avoid prefixing fields with an underscore (`_`) that are assigned to in constructors, which differs from the ASP.NET Core

framework engineering guidelines.

- In examples that use primary constructors (C# 12 or later), primary constructor parameters are typically used directly by class members.

Additional information on Razor component syntax is provided in the *Razor syntax* section of ASP.NET Core Razor components.

The following is an example counter component and part of an app created from a Blazor project template. Detailed components coverage is found in the *Components* articles later in the documentation. The following example demonstrates component concepts seen in the *Fundamentals* articles before reaching the *Components* articles later in the documentation.

`Counter.razor`:

The component assumes that an interactive render mode is inherited from a parent component or applied globally to the app.

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 preceding `Counter` component:

- Sets its route with the `@page` directive in the first line.
- Sets its page title and heading.
- Renders the current count with `@currentCount`. `currentCount` is an integer variable defined in the C# code of the `@code` block.
- Displays a button to trigger the `IncrementCount` method, which is also found in the `@code` block and increases the value of the `currentCount` variable.

Render modes

Articles in the *Fundamentals* node make reference to the concept of *render modes*. This subject is covered in detail in the ASP.NET Core Blazor render modes article in the *Components* node, which appears after the *Fundamentals* node of articles.

For the early references in this node of articles to render mode concepts, merely note the following at this time:

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 rendered statically on the server, rendered with for user interactivity on the server, or rendered for user interactivity on the client (usually with prerendering on the server).

Blazor Server and Blazor WebAssembly apps for ASP.NET Core releases prior to .NET 8 remain fixated on *hosting model* concepts, not render modes. Render modes are conceptually applied to Blazor Web Apps in .NET 8 or later.

The following table shows the available render modes for rendering Razor components in a Blazor Web App. Render modes are applied to components with the `@rendermode` directive on the component instance or on the component definition. It's also possible to set a render mode for the entire app.

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.

The preceding information on render modes is all that you need to know to understand the *Fundamentals* node articles. If you're new to Blazor and reading Blazor articles in

order down the table of contents, you can delay consuming in-depth information on render modes until you reach the ASP.NET Core Blazor render modes article in the *Components* node.

Document Object Model (DOM)

References to the *Document Object Model* use the abbreviation *DOM*.

For more information, see the following resources:

- Introduction to the DOM (MDN documentation)
- Level 1 Document Object Model Specification (W3C)

Subset of .NET APIs for Blazor WebAssembly apps

A curated list of specific .NET APIs that are supported on the browser for Blazor WebAssembly isn't available. However, you can manually search for a list of .NET APIs annotated with [UnsupportedOSPlatform("browser")] to discover .NET APIs that aren't supported in WebAssembly.

> **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).

For more information, see the following resources:

- Class libraries: Client-side browser compatibility analyzer
- Annotating APIs as unsupported on specific platforms (dotnet/designs GitHub repository

Sample apps

Documentation sample apps are available for inspection and download:

Blazor samples GitHub repository (dotnet/blazor-samples)

Locate a sample app by first selecting the version folder that matches the version of .NET that you're working with.

Samples apps in the repository:

- Blazor Web App
- Blazor WebAssembly
- Blazor Web App with EF Core (ASP.NET Core Blazor with Entity Framework Core (EF Core))
- Blazor Web App with SignalR (Use ASP.NET Core SignalR with Blazor)
- Two Blazor Web Apps and a Blazor WebAssembly app for calling web (server) APIs (Call a web API from an ASP.NET Core Blazor app)
- Blazor Web App with OIDC (BFF and non-BFF patterns) (Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC))
- Blazor WebAssembly scopes-enabled logging (ASP.NET Core Blazor logging)
- Blazor WebAssembly with ASP.NET Core Identity (Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity)
- .NET MAUI Blazor Hybrid app with a Blazor Web App and a shared UI provided by a Razor class library (RCL) (Build a .NET MAUI Blazor Hybrid app with a Blazor Web App)

For more information and a list of the samples in the repository, see the Blazor samples GitHub repository README.md file.

The ASP.NET Core repository's Basic Test App is also a helpful set of samples for various Blazor scenarios:

BasicTestApp in ASP.NET Core reference source (dotnet/aspnetcore)

> **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).

To download the sample apps:

- Download the Blazor samples repository ZIP file.
- Unzip the file.

Byte multiples

.NET byte sizes use metric prefixes for non-decimal multiples of bytes based on powers of 1024.

Expand table

Name (abbreviation)	Size	Example
Kilobyte (KB)	1,024 bytes	1 KB = 1,024 bytes
Megabyte (MB)	$1,024^2$ bytes	1 MB = 1,048,576 bytes
Gigabyte (GB)	$1,024^3$ bytes	1 GB = 1,073,741,824 bytes

Support requests

Only documentation-related issues are appropriate for the `dotnet/AspNetCore.Docs` repository. ***For product support, don't open a documentation issue.*** Seek assistance through one or more of the following support channels:

- Stack Overflow (tagged: blazor)
- General ASP.NET Core Slack Team
- Blazor Gitter

For a potential bug in the framework or product feedback, open an issue for the ASP.NET Core product unit at dotnet/aspnetcore issues. Bug reports usually ***require*** the following:

- **Clear explanation of the problem**: Follow the instructions in the GitHub issue template provided by the product unit when opening the issue.
- **Minimal repro project**: Place a project on GitHub for the product unit engineers to download and run. Cross-link the project into the issue's opening comment.

For a potential problem with a Blazor article, open a documentation issue. To open a documentation issue, use the **Open a documentation issue** feedback link at the bottom of the article. Metadata added to your issue provides tracking data and automatically pings the author of the article. If the subject was discussed with the product unit prior to opening the documentation issue, place a cross-link to the engineering issue in the documentation issue's opening comment.

For problems or feedback on Visual Studio, use the **Report a Problem** or **Suggest a Feature** gestures from within Visual Studio, which open internal issues for Visual Studio.

For more information, see Visual Studio Feedback.

For problems with Visual Studio Code, ask for support on community support forums. For bug reports and product feedback, open an issue on the microsoft/vscode GitHub repo.

GitHub issues for Blazor documentation are automatically marked for triage on the Blazor.Docs project (dotnet/AspNetCore.Docs GitHub repository). Please wait a short while for a response, especially over weekends and holidays. Usually, documentation authors respond within 24 hours on weekdays.

Community links to Blazor resources

For a collection of links to Blazor resources maintained by the community, visit Awesome Blazor.

> **Note**
>
> Microsoft doesn't own, maintain, or support *Awesome Blazor* and most of the community products and services described and linked there.

ASP.NET Core Blazor routing and navigation

Article • 10/21/2024

> **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 manage Blazor app request routing and how to use the NavLink component to create navigation links.

> **Important**
>
> Code examples throughout this article show methods called on `Navigation`, which is an injected NavigationManager in classes and components.

Static versus interactive routing

This section applies to Blazor Web Apps.

If prerendering is enabled, the Blazor router (`Router` component, `<Router>` in `Routes.razor`) performs static routing to components during static server-side rendering (static SSR). This type of routing is called *static routing*.

When an interactive render mode is assigned to the `Routes` component, the Blazor router becomes interactive after static SSR with static routing on the server. This type of routing is called *interactive routing*.

Static routers use endpoint routing and the HTTP request path to determine which component to render. When the router becomes interactive, it uses the document's URL (the URL in the browser's address bar) to determine which component to render. This means that the interactive router can dynamically change which component is rendered if the document's URL dynamically changes to another valid internal URL, and it can do so without performing an HTTP request to fetch new page content.

Interactive routing also prevents prerendering because new page content isn't requested from the server with a normal page request. For more information, see Prerender ASP.NET Core Razor components.

Route templates

The Router component enables routing to Razor components and is located in the app's `Routes` component (`Components/Routes.razor`).

When a Razor component (`.razor`) with an @page directive is compiled, the generated component class is provided a RouteAttribute specifying the component's route template.

When the app starts, the assembly specified as the Router's `AppAssembly` is scanned to gather route information for the app's components that have a RouteAttribute.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any route parameters.

- Renders the specified component with its layout, including any further nested layouts.

Optionally specify a DefaultLayout parameter with a layout class for components that don't specify a layout with the @layout directive. The framework's Blazor project templates specify the `MainLayout` component (`MainLayout.razor`) as the app's default layout. For more information on layouts, see ASP.NET Core Blazor layouts.

Components support multiple route templates using multiple @page directives. The following example component loads on requests for `/blazor-route` and `/different-blazor-route`.

`BlazorRoute.razor`:

razor

```
@page "/blazor-route"
@page "/different-blazor-route"

<PageTitle>Routing</PageTitle>

<h1>Routing Example</h1>

<p>
    This page is reached at either <code>/blazor-route</code> or
    <code>/different-blazor-route</code>.
</p>
```

> **Important**
>
> For URLs to resolve correctly, the app must include a `<base>` tag (location of <head> content) with the app base path specified in the `href` attribute. For more information, see Host and deploy ASP.NET Core Blazor.

As an alternative to specifying the route template as a string literal with the `@page` directive, constant-based route templates can be specified with the @attribute directive.

In the following example, the `@page` directive in a component is replaced with the `@attribute` directive and the constant-based route template in `Constants.CounterRoute`, which is set elsewhere in the app to "`/counter`":

diff

```
- @page "/counter"
+ @attribute [Route(Constants.CounterRoute)]
```

Focus an element on navigation

The FocusOnNavigate component sets the UI focus to an element based on a CSS selector after navigating from one page to another.

razor

```
<FocusOnNavigate RouteData="routeData" Selector="h1" />
```

When the Router component navigates to a new page, the FocusOnNavigate component sets the focus to the page's top-level header (`<h1>`). This is a common strategy for ensuring that a page navigation is announced when using a screen reader.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

Set custom content for the Router component's NotFound parameter:

```razor
<Router ...>
    ...
    <NotFound>
        ...
    </NotFound>
</Router>
```

Arbitrary items are supported as content of the NotFound parameter, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

> **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**.

Route to components from multiple assemblies

This section applies to Blazor Web Apps.

Use the Router component's AdditionalAssemblies parameter and the endpoint convention builder AddAdditionalAssemblies to discover routable components in additional assemblies. The following subsections explain when and how to use each API.

Static routing

To discover routable components from additional assemblies for static server-side rendering (static SSR), even if the router later becomes interactive for interactive rendering, the assemblies must be disclosed to the Blazor framework. Call the AddAdditionalAssemblies method with the additional assemblies chained to MapRazorComponents in the server project's `Program` file.

The following example includes the routable components in the `BlazorSample.Client` project's assembly using the project's `_Imports.razor` file:

```C#
app.MapRazorComponents<App>()
    .AddAdditionalAssemblies(typeof(BlazorSample.Client._Imports).Assembly);
```

> **Note**
>
> The preceding guidance also applies in **component class library** scenarios. Additional important guidance for class libraries and static SSR is found in **ASP.NET Core Razor class libraries (RCLs) with static server-side**

rendering (static SSR).

Interactive routing

An interactive render mode can be assigned to the `Routes` component (`Routes.razor`) that makes the Blazor router become interactive after static SSR and static routing on the server. For example, `<Routes @rendermode="InteractiveServer" />` assigns interactive server-side rendering (interactive SSR) to the `Routes` component. The `Router` component inherits interactive server-side rendering (interactive SSR) from the `Routes` component. The router becomes interactive after static routing on the server.

Internal navigation for interactive routing doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. For more information, see Prerender ASP.NET Core Razor components.

If the `Routes` component is defined in the server project, the AdditionalAssemblies parameter of the `Router` component should include the `.Client` project's assembly. This allows the router to work correctly when rendered interactively.

In the following example, the `Routes` component is in the server project, and the `_Imports.razor` file of the `BlazorSample.Client` project indicates the assembly to search for routable components:

razor

```
<Router
    AppAssembly="..."
    AdditionalAssemblies="new[] { typeof(BlazorSample.Client._Imports).Assembly }">
    ...
</Router>
```

Additional assemblies are scanned in addition to the assembly specified to AppAssembly.

> **Note**
>
> The preceding guidance also applies in component class library scenarios.

Alternatively, routable components only exist in the `.Client` project with global Interactive WebAssembly or Auto rendering applied, and the `Routes` component is defined in the `.Client` project, not the server project. In this case, there aren't external assemblies with routable components, so it isn't necessary to specify a value for AdditionalAssemblies.

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name. Route parameter names are case insensitive. In the following example, the `text` parameter assigns the value of the route segment to the component's `Text` property. When a request is made for `/route-parameter-1/amazing`, the content is rendered as `Blazor is amazing!`.

`RouteParameter1.razor`:

razor

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

<PageTitle>Route Parameter 1</PageTitle>

<h1>Route Parameter Example 1</h1>

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

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

Optional parameters are supported. In the following example, the `text` optional parameter assigns the value of the route segment to the component's `Text` property. If the segment isn't present, the value of `Text` is set to `fantastic`.

`RouteParameter2.razor`:

razor

```razor
@page "/route-parameter-2/{text?}"

<PageTitle>Route Parameter 2</PageTitle>

<h1>Route Parameter Example 2</h1>

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

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

    protected override void OnParametersSet() => Text = Text ?? "fantastic";
}
```

When the OnInitialized{Async} lifecycle method is used instead of the OnParametersSet{Async} lifecycle method, the default assignment of the `Text` property to `fantastic` doesn't occur if the user navigates within the same component. For example, this situation arises when the user navigates from `/route-parameter-2/amazing` to `/route-parameter-2`. As the component instance persists and accepts new parameters, the `OnInitialized` method isn't invoked again.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `User` component only matches if:

- An `Id` route segment is present in the request URL.
- The `Id` segment is an integer (`int`) type.

`User.razor`:

razor

```razor
@page "/user/{Id:int}"

<PageTitle>User</PageTitle>
```

```razor
<h1>User Example</h1>

<p>User Id: @Id</p>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match the invariant culture, see the warning below the table for more information.

Expand table

Constraint	Example	Example Matches	Invariant culture matching
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes
`guid`	`{id:guid}`	`00001111-aaaa-2222-bbbb-3333cccc4444`, `{00001111-aaaa-2222-bbbb-3333cccc4444}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes
`nonfile`	`{parameter:nonfile}`	Not `BlazorSample.styles.css`, not `favicon.ico`	Yes

> ⚠ **Warning**
>
> Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Route constraints also work with optional parameters. In the following example, `Id` is required, but `Option` is an optional boolean route parameter.

`User.razor`:

```razor
@page "/user/{id:int}/{option:bool?}"

<p>
    Id: @Id
</p>

<p>
    Option: @Option
</p>

@code {
    [Parameter]
```

```
    public int Id { get; set; }

    [Parameter]
    public bool Option { get; set; }
}
```

Avoid file capture in a route parameter

The following route template inadvertently captures static asset paths in its optional route parameter (`Optional`). For example, the app's stylesheet (`.styles.css`) is captured, which breaks the app's styles:

razor

```
@page "/{optional?}"

...

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

To restrict a route parameter to capturing non-file paths, use the :nonfile constraint in the route template:

razor

```
@page "/{optional:nonfile?}"
```

Catch-all route parameters

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components.

Catch-all route parameters are:

- Named to match the route segment name. Naming isn't case-sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

`CatchAll.razor`:

razor

```
@page "/catch-all/{*pageRoute}"

<PageTitle>Catch All</PageTitle>

<h1>Catch All Parameters Example</h1>

<p>Add some URI segments to the route and request the page again.</p>

<p>
    PageRoute: @PageRoute
</p>

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

For the URL `/catch-all/this/is/a/test` with a route template of `/catch-all/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/catch-all/{*pageRoute}`, the URL `/catch-all/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

URI and navigation state helpers

Use NavigationManager to manage URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

Expand table

Member	Description
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element (location of <head> content).
NavigateTo	Navigates to the specified URI. If `forceLoad` is `false`: • And enhanced navigation is available at the current URL, Blazor's enhanced navigation is activated. • Otherwise, Blazor performs a full-page reload for the requested URL. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side interactive router. For more information, see the Enhanced navigation and form handling section. If `replace` is `true`, the current URI in the browser history is replaced instead of pushing a new URI onto the history stack.
LocationChanged	An event that fires when the navigation location has changed. For more information, see the Location changes section.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Based on the app's base URI, converts an absolute URI into a URI relative to the base URI prefix. For an example, see the Produce a URI relative to the base URI prefix section.
RegisterLocationChangingHandler	Registers a handler to process incoming navigation events. Calling NavigateTo always invokes the handler.
GetUriWithQueryParameter	Returns a URI constructed by updating NavigationManager.Uri with a single parameter added, updated, or removed. For more information, see the Query strings section.

Location changes

For the LocationChanged event, LocationChangedEventArgs provides the following information about navigation events:

- Location: The URL of the new location.

- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

The following component:

- Navigates to the app's `Counter` component (`Counter.razor`) when the button is selected using NavigateTo.
- Handles the location changed event by subscribing to NavigationManager.LocationChanged.
  - The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.
  - The logger implementation logs the following information when the button is selected:

    > BlazorSample.Pages.Navigate: Information: URL of new location: https://localhost:{PORT}/counter

`Navigate.razor`:

razor

```
@page "/navigate"
@implements IDisposable
@inject ILogger<Navigate> Logger
@inject NavigationManager Navigation

<PageTitle>Navigate</PageTitle>

<h1>Navigate Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent() => Navigation.NavigateTo("counter");

    protected override void OnInitialized() =>
        Navigation.LocationChanged += HandleLocationChanged;

    private void HandleLocationChanged(object? sender, LocationChangedEventArgs e) =>
        Logger.LogInformation("URL of new location: {Location}", e.Location);

    public void Dispose() => Navigation.LocationChanged -= HandleLocationChanged;
}
```

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

Enhanced navigation and form handling

This section applies to Blazor Web Apps.

Blazor Web Apps are capable of two types of routing for page navigation and form handling requests:

- Normal navigation (cross-document navigation): a full-page reload is triggered for the request URL.
- Enhanced navigation (same-document navigation): Blazor intercepts the request and performs a `fetch` request instead. Blazor then patches the response content into the page's DOM. Blazor's enhanced navigation and form handling avoid the need for a full-page reload and preserves more of the page state, so pages load faster, usually without losing the user's scroll position on the page.

Enhanced navigation is available when:

- The Blazor Web App script (`blazor.web.js`) is used, not the Blazor Server script (`blazor.server.js`) or Blazor WebAssembly script (`blazor.webassembly.js`).
- The feature isn't explicitly disabled.
- The destination URL is within the internal base URI space (the app's base path).

If server-side routing and enhanced navigation are enabled, location changing handlers are only invoked for programmatic navigation initiated from an interactive runtime. In future releases, additional types of navigation, such as following a link, may also invoke location changing handlers.

When an enhanced navigation occurs, LocationChanged event handlers registered with Interactive Server and WebAssembly runtimes are typically invoked. There are cases when location changing handlers might not intercept an enhanced navigation. For example, the user might switch to another page before an interactive runtime becomes available. Therefore, it's important that app logic not rely on invoking a location changing handler, as there's no guarantee of the handler executing.

When calling NavigateTo:

- If `forceLoad` is `false`, which is the default:
  - And enhanced navigation is available at the current URL, Blazor's enhanced navigation is activated.
  - Otherwise, Blazor performs a full-page reload for the requested URL.
- If `forceLoad` is `true`: Blazor performs a full-page reload for the requested URL, whether enhanced navigation is available or not.

You can refresh the current page by calling `NavigationManager.Refresh(bool forceLoad = false)`, which always performs an enhanced navigation, if available. If enhanced navigation isn't available, Blazor performs a full-page reload.

C#

```
Navigation.Refresh();
```

Pass `true` to the `forceLoad` parameter to ensure a full-page reload is always performed, even if enhanced navigation is available:

C#

```
Navigation.Refresh(true);
```

Enhanced navigation is enabled by default, but it can be controlled hierarchically and on a per-link basis using the `data-enhance-nav` HTML attribute.

The following examples disable enhanced navigation:

HTML

```
<a href="redirect" data-enhance-nav="false">
    GET without enhanced navigation
</a>
```

razor

```
<ul data-enhance-nav="false">
    <li>
        <a href="redirect">GET without enhanced navigation</a>
    </li>
    <li>
```

```
        <a href="redirect-2">GET without enhanced navigation</a>
    </li>
</ul>
```

If the destination is a non-Blazor endpoint, enhanced navigation doesn't apply, and the client-side JavaScript retries as a full page load. This ensures no confusion to the framework about external pages that shouldn't be patched into an existing page.

To enable enhanced form handling, add the Enhance parameter to EditForm forms or the `data-enhance` attribute to HTML forms (`<form>`):

razor

```
<EditForm ... Enhance ...>
    ...
</EditForm>
```

HTML

```
<form ... data-enhance ...>
    ...
</form>
```

Enhanced form handling isn't hierarchical and doesn't flow to child forms:

❌ You can't set enhanced navigation on a form's ancestor element to enable enhanced navigation for the form.

HTML

```
<div ... data-enhance ...>
    <form ...>
        <!-- NOT enhanced -->
    </form>
</div>
```

Enhanced form posts only work with Blazor endpoints. Posting an enhanced form to non-Blazor endpoint results in an error.

To disable enhanced navigation:

- For an EditForm, remove the Enhance parameter from the form element (or set it to `false`: `Enhance="false"`).
- For an HTML `<form>`, remove the `data-enhance` attribute from form element (or set it to `false`: `data-enhance="false"`).

Blazor's enhanced navigation and form handing may undo dynamic changes to the DOM if the updated content isn't part of the server rendering. To preserve the content of an element, use the `data-permanent` attribute.

In the following example, the content of the `<div>` element is updated dynamically by a script when the page loads:

HTML

```
<div data-permanent>
    ...
</div>
```

Once Blazor has started on the client, you can use the `enhancedload` event to listen for enhanced page updates. This allows for re-applying changes to the DOM that may have been undone by an enhanced page update.

JavaScript

```
Blazor.addEventListener('enhancedload', () => console.log('Enhanced update!'));
```

To disable enhanced navigation and form handling globally, see ASP.NET Core Blazor startup.

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

Produce a URI relative to the base URI prefix

Based on the app's base URI, ToBaseRelativePath converts an absolute URI into a URI relative to the base URI prefix.

Consider the following example:

C#

```
try
{
    baseRelativePath = Navigation.ToBaseRelativePath(inputURI);
}
catch (ArgumentException ex)
{
    ...
}
```

If the base URI of the app is `https://localhost:8000`, the following results are obtained:

- Passing `https://localhost:8000/segment` in `inputURI` results in a `baseRelativePath` of `segment`.
- Passing `https://localhost:8000/segment1/segment2` in `inputURI` results in a `baseRelativePath` of `segment1/segment2`.

If the base URI of the app doesn't match the base URI of `inputURI`, an ArgumentException is thrown.

Passing `https://localhost:8001/segment` in `inputURI` results in the following exception:

> System.ArgumentException: 'The URI 'https://localhost:8001/segment' is not contained by the base URI 'https://localhost:8000/'.'

Navigation history state

The NavigationManager uses the browser's History API to maintain navigation history state associated with each location change made by the app. Maintaining history state is particularly useful in external redirect scenarios, such as when authenticating users with external identity providers. For more information, see the Navigation options section.

Navigation options

Pass NavigationOptions to NavigateTo to control the following behaviors:

- ForceLoad: Bypass client-side routing and force the browser to load the new page from the server, whether or not the URI is handled by the client-side router. The default value is `false`.
- ReplaceHistoryEntry: Replace the current entry in the history stack. If `false`, append the new entry to the history stack. The default value is `false`.
- HistoryEntryState: Gets or sets the state to append to the history entry.

C#

```
Navigation.NavigateTo("/path", new NavigationOptions
{
    HistoryEntryState = "Navigation state"
});
```

For more information on obtaining the state associated with the target history entry while handling location changes, see the Handle/prevent location changes section.

Query strings

Use the [SupplyParameterFromQuery] attribute to specify that a component parameter comes from the query string.

Component parameters supplied from the query string support the following types:

- `bool`, `DateTime`, `decimal`, `double`, `float`, `Guid`, `int`, `long`, `string`.
- Nullable variants of the preceding types.
- Arrays of the preceding types, whether they're nullable or not nullable.

The correct culture-invariant formatting is applied for the given type (CultureInfo.InvariantCulture).

Specify the `[SupplyParameterFromQuery]` attribute's Name property to use a query parameter name different from the component parameter name. In the following example, the C# name of the component parameter is `{COMPONENT PARAMETER NAME}`. A different query parameter name is specified for the `{QUERY PARAMETER NAME}` placeholder:

Unlike component parameter properties (`[Parameter]`), `[SupplyParameterFromQuery]` properties can be marked `private` in addition to `public`.

C#

```
[SupplyParameterFromQuery(Name = "{QUERY PARAMETER NAME}")]
private string? {COMPONENT PARAMETER NAME} { get; set; }
```

In the following example with a URL of `/search?filter=scifi%20stars&page=3&star=LeVar%20Burton&star=Gary%20Oldman`:

- The `Filter` property resolves to `scifi stars`.
- The `Page` property resolves to `3`.
- The `Stars` array is filled from query parameters named `star` (`Name = "star"`) and resolves to `LeVar Burton` and `Gary Oldman`.

> **Note**
>
> The query string parameters in the following routable page component also work in a *non-routable* component without an `@page` directive (for example, `Search.razor` for a shared `Search` component used in

other components).

`Search.razor`:

razor

```razor
@page "/search"

<h1>Search Example</h1>

<p>Filter: @Filter</p>

<p>Page: @Page</p>

@if (Stars is not null)
{
    <p>Stars:</p>

    <ul>
        @foreach (var name in Stars)
        {
            <li>@name</li>
        }
    </ul>
}

@code {
    [SupplyParameterFromQuery]
    private string? Filter { get; set; }

    [SupplyParameterFromQuery]
    private int? Page { get; set; }

    [SupplyParameterFromQuery(Name = "star")]
    private string[]? Stars { get; set; }
}
```

Use GetUriWithQueryParameter to add, change, or remove one or more query parameters on the current URL:

razor

```razor
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameter("{NAME}", {VALUE})
```

For the preceding example:

- The `{NAME}` placeholder specifies the query parameter name. The `{VALUE}` placeholder specifies the value as a supported type. Supported types are listed later in this section.
- A string is returned equal to the current URL with a single parameter:
  - Added if the query parameter name doesn't exist in the current URL.
  - Updated to the value provided if the query parameter exists in the current URL.
  - Removed if the type of the provided value is nullable and the value is `null`.
- The correct culture-invariant formatting is applied for the given type (CultureInfo.InvariantCulture).
- The query parameter name and value are URL-encoded.
- All of the values with the matching query parameter name are replaced if there are multiple instances of the type.

Call GetUriWithQueryParameters to create a URI constructed from Uri with multiple parameters added, updated, or removed. For each value, the framework uses `value?.GetType()` to determine the runtime type for each query parameter and selects the correct culture-invariant formatting. The framework throws an error for unsupported types.

razor

```
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameters({PARAMETERS})
```

The `{PARAMETERS}` placeholder is an `IReadOnlyDictionary<string, object>`.

Pass a URI string to GetUriWithQueryParameters to generate a new URI from a provided URI with multiple parameters added, updated, or removed. For each value, the framework uses `value?.GetType()` to determine the runtime type for each query parameter and selects the correct culture-invariant formatting. The framework throws an error for unsupported types. Supported types are listed later in this section.

razor

```
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameters("{URI}", {PARAMETERS})
```

- The `{URI}` placeholder is the URI with or without a query string.
- The `{PARAMETERS}` placeholder is an `IReadOnlyDictionary<string, object>`.

Supported types are identical to supported types for route constraints:

- `bool`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Guid`
- `int`
- `long`
- `string`

Supported types include:

- Nullable variants of the preceding types.
- Arrays of the preceding types, whether they're nullable or not nullable.

> ⚠ **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.

Replace a query parameter value when the parameter exists

```
C#

Navigation.GetUriWithQueryParameter("full name", "Morena Baccarin")
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&age=42`	`scheme://host/?full%20name=Morena%20Baccarin&age=42`
`scheme://host/?fUlL%20nAmE=David%20Krumholtz&AgE=42`	`scheme://host/?full%20name=Morena%20Baccarin&AgE=42`
`scheme://host/?full%20name=Jewel%20Staite&age=42&full%20name=Summer%20Glau`	`scheme://host/?full%20name=Morena%20Baccarin&age=42&full%20name=Morena%20Baccarin`
`scheme://host/?full%20name=&age=42`	`scheme://host/?full%20name=Morena%20Baccarin&age=42`
`scheme://host/?full%20name=`	`scheme://host/?full%20name=Morena%20Baccarin`

Append a query parameter and value when the parameter doesn't exist

```
C#

Navigation.GetUriWithQueryParameter("name", "Morena Baccarin")
```

Expand table

Current URL	Generated URL
`scheme://host/?age=42`	`scheme://host/?age=42&name=Morena%20Baccarin`
`scheme://host/`	`scheme://host/?name=Morena%20Baccarin`
`scheme://host/?`	`scheme://host/?name=Morena%20Baccarin`

Remove a query parameter when the parameter value is `null`

```
C#

Navigation.GetUriWithQueryParameter("full name", (string)null)
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&age=42`	`scheme://host/?age=42`

Current URL	Generated URL
`scheme://host/?full%20name=Sally%20Smith&age=42&full%20name=Summer%20Glau`	`scheme://host/?age=42`
`scheme://host/?full%20name=Sally%20Smith&age=42&FuLl%20NaMe=Summer%20Glau`	`scheme://host/?age=42`
`scheme://host/?full%20name=&age=42`	`scheme://host/?age=42`
`scheme://host/?full%20name=`	`scheme://host/`

Add, update, and remove query parameters

In the following example:

- `name` is removed, if present.
- `age` is added with a value of `25` (`int`), if not present. If present, `age` is updated to a value of `25`.
- `eye color` is added or updated to a value of `green`.

C#

```
Navigation.GetUriWithQueryParameters(
    new Dictionary<string, object?>
    {
        ["name"] = null,
        ["age"] = (int?)25,
        ["eye color"] = "green"
    })
```

Expand table

Current URL	Generated URL
`scheme://host/?name=David%20Krumholtz&age=42`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?NaMe=David%20Krumholtz&AgE=42`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?name=David%20Krumholtz&age=42&keepme=true`	`scheme://host/?age=25&keepme=true&eye%20color=green`
`scheme://host/?age=42&eye%20color=87`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/`	`scheme://host/?age=25&eye%20color=green`

Support for enumerable values

In the following example:

- `full name` is added or updated to `Morena Baccarin`, a single value.
- `ping` parameters are added or replaced with `35`, `16`, `87` and `240`.

C#

```
Navigation.GetUriWithQueryParameters(
    new Dictionary<string, object?>
    {
        ["full name"] = "Morena Baccarin",
```

```
        ["ping"] = new int?[] { 35, 16, null, 87, 240 }
    })
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&ping=8&ping=300`	`scheme://host/?full%20name=Morena%20Baccarin&ping=35&ping=16&ping=87&ping=240`
`scheme://host/?ping=8&full%20name=David%20Krumholtz&ping=300`	`scheme://host/?ping=35&full%20name=Morena%20Baccarin&ping=16&ping=87&ping=240`
`scheme://host/?ping=8&ping=300&ping=50&ping=68&ping=42`	`scheme://host/?ping=35&ping=16&ping=87&ping=240&full%20name=Morena%20Baccarin`

Navigate with an added or modified query string

To navigate with an added or modified query string, pass a generated URL to NavigateTo.

The following example calls:

- GetUriWithQueryParameter to add or replace the `name` query parameter using a value of `Morena Baccarin`.
- Calls NavigateTo to trigger navigation to the new URL.

C#

```
Navigation.NavigateTo(
    Navigation.GetUriWithQueryParameter("name", "Morena Baccarin"));
```

Hashed routing to named elements

Navigate to a named element using the following approaches with a hashed (`#`) reference to the element. Routes to elements within the component and routes to elements in external components use root-relative paths. A leading forward slash (`/`) is optional.

Examples for each of the following approaches demonstrate navigation to an element with an `id` of `targetElement` in the `Counter` component:

- Anchor element (`<a>`) with an `href`:

  razor

  ```
  <a href="/counter#targetElement">
  ```

- NavLink component with an `href`:

  razor

  ```
  <NavLink href="/counter#targetElement">
  ```

- NavigationManager.NavigateTo passing the relative URL:

  C#

```
Navigation.NavigateTo("/counter#targetElement");
```

The following example demonstrates hashed routing to named H2 headings within a component and to external components.

In the `Home` (`Home.razor`) and `Counter` (`Counter.razor`) components, place the following markup at the bottoms of the existing component markup to serve as navigation targets. The `<div>` creates artificial vertical space to demonstrate browser scrolling behavior:

razor

```
<div class="border border-info rounded bg-info" style="height:500px"></div>

<h2 id="targetElement">Target H2 heading</h2>
<p>Content!</p>
```

Add the following `HashedRouting` component to the app.

`HashedRouting.razor`:

razor

```
@page "/hashed-routing"
@inject NavigationManager Navigation

<PageTitle>Hashed routing</PageTitle>

<h1>Hashed routing to named elements</h1>

<ul>
    <li>
        <a href="/hashed-routing#targetElement">
            Anchor in this component
        </a>
    </li>
    <li>
        <a href="/#targetElement">
            Anchor to the <code>Home</code> component
        </a>
    </li>
    <li>
        <a href="/counter#targetElement">
            Anchor to the <code>Counter</code> component
        </a>
    </li>
    <li>
        <NavLink href="/hashed-routing#targetElement">
            Use a `NavLink` component in this component
        </NavLink>
    </li>
    <li>
        <button @onclick="NavigateToElement">
            Navigate with <code>NavigationManager</code> to the
            <code>Counter</code> component
        </button>
    </li>
</ul>

<div class="border border-info rounded bg-info" style="height:500px"></div>

<h2 id="targetElement">Target H2 heading</h2>
<p>Content!</p>
```

```
@code {
    private void NavigateToElement()
    {
        Navigation.NavigateTo("/counter#targetElement");
    }
}
```

User interaction with `<Navigating>` content

If there's a significant delay during navigation, such as while lazy-loading assemblies in a Blazor WebAssembly app or for a slow network connection to a Blazor server-side app, the Router component can indicate to the user that a page transition is occurring.

At the top of the component that specifies the Router component, add an @using directive for the Microsoft.AspNetCore.Components.Routing namespace:

razor

```
@using Microsoft.AspNetCore.Components.Routing
```

Provide content to the Navigating parameter for display during page transition events.

In the router element (`<Router>...</Router>`) content:

razor

```
<Navigating>
    <p>Loading the requested page&hellip;</p>
</Navigating>
```

For an example that uses the Navigating property, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Handle asynchronous navigation events with `OnNavigateAsync`

The Router component supports an OnNavigateAsync feature. The OnNavigateAsync handler is invoked when the user:

- Visits a route for the first time by navigating to it directly in their browser.
- Navigates to a new route using a link or a NavigationManager.NavigateTo invocation.

razor

```
<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext args)
    {
        ...
    }
}
```

For an example that uses OnNavigateAsync, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

When prerendering on the server, OnNavigateAsync is executed *twice*:

- Once when the requested endpoint component is initially rendered statically.
- A second time when the browser renders the endpoint component.

To prevent developer code in OnNavigateAsync from executing twice, the `Routes` component can store the NavigationContext for use in the OnAfterRender{Async} lifecycle method, where `firstRender` can be checked. For more information, see Prerendering with JavaScript interop.

Handle cancellations in `OnNavigateAsync`

The NavigationContext object passed to the OnNavigateAsync callback contains a CancellationToken that's set when a new navigation event occurs. The OnNavigateAsync callback must throw when this cancellation token is set to avoid continuing to run the OnNavigateAsync callback on an outdated navigation.

If a user navigates to an endpoint but then immediately navigates to a new endpoint, the app shouldn't continue running the OnNavigateAsync callback for the first endpoint.

In the following example:

- The cancellation token is passed in the call to `PostAsJsonAsync`, which can cancel the POST if the user navigates away from the `/about` endpoint.
- The cancellation token is set during a product prefetch operation if the user navigates away from the `/store` endpoint.

razor

```
@inject HttpClient Http
@inject ProductCatalog Products

<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext context)
    {
        if (context.Path == "/about")
        {
            var stats = new Stats { Page = "/about" };
            await Http.PostAsJsonAsync("api/visited", stats,
                context.CancellationToken);
        }
        else if (context.Path == "/store")
        {
            var productIds = new[] { 345, 789, 135, 689 };

            foreach (var productId in productIds)
            {
                context.CancellationToken.ThrowIfCancellationRequested();
                Products.Prefetch(productId);
            }
        }
    }
}
```

> ⓘ Note

Not throwing if the cancellation token in NavigationContext is canceled can result in unintended behavior, such as rendering a component from a previous navigation.

Handle/prevent location changes

RegisterLocationChangingHandler registers a handler to process incoming navigation events. The handler's context provided by LocationChangingContext includes the following properties:

- TargetLocation: Gets the target location.
- HistoryEntryState: Gets the state associated with the target history entry.
- IsNavigationIntercepted: Gets whether the navigation was intercepted from a link.
- CancellationToken: Gets a CancellationToken to determine if the navigation was canceled, for example, to determine if the user triggered a different navigation.
- PreventNavigation: Called to prevent the navigation from continuing.

A component can register multiple location changing handlers in the OnAfterRender{Async} lifecycle method. Navigation invokes all of the location changing handlers registered across the entire app (across multiple components), and any internal navigation executes them all in parallel. In addition to NavigateTo handlers are invoked:

- When selecting internal links, which are links that point to URLs under the app's base path.
- When navigating using the forward and back buttons in a browser.

Handlers are only executed for internal navigation within the app. If the user selects a link that navigates to a different site or changes the address bar to a different site manually, location changing handlers aren't executed.

Implement IDisposable and dispose registered handlers to unregister them. For more information, see ASP.NET Core Razor component lifecycle.

> **Important**
>
> Don't attempt to execute DOM cleanup tasks via JavaScript (JS) interop when handling location changes. Use the MutationObserver pattern in JS on the client. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

In the following example, a location changing handler is registered for navigation events.

`NavHandler.razor`:

razor

```
@page "/nav-handler"
@implements IDisposable
@inject NavigationManager Navigation

<p>
    <button @onclick="@(() => Navigation.NavigateTo("/"))">
        Home (Allowed)
    </button>
    <button @onclick="@(() => Navigation.NavigateTo("/counter"))">
        Counter (Prevented)
    </button>
</p>

@code {
    private IDisposable? registration;
```

```
    protected override void OnAfterRender(bool firstRender)
    {
        if (firstRender)
        {
            registration =
                Navigation.RegisterLocationChangingHandler(OnLocationChanging);
        }
    }

    private ValueTask OnLocationChanging(LocationChangingContext context)
    {
        if (context.TargetLocation == "/counter")
        {
            context.PreventNavigation();
        }

        return ValueTask.CompletedTask;
    }

    public void Dispose() => registration?.Dispose();
}
```

Since internal navigation can be canceled asynchronously, multiple overlapping calls to registered handlers may occur. For example, multiple handler calls may occur when the user rapidly selects the back button on a page or selects multiple links before a navigation is executed. The following is a summary of the asynchronous navigation logic:

- If any location changing handlers are registered, all navigation is initially reverted, then replayed if the navigation isn't canceled.
- If overlapping navigation requests are made, the latest request always cancels earlier requests, which means the following:
  - The app may treat multiple back and forward button selections as a single selection.
  - If the user selects multiple links before the navigation completes, the last link selected determines the navigation.

For more information on passing NavigationOptions to NavigateTo to control entries and state of the navigation history stack, see the Navigation options section.

For additional example code, see the NavigationManagerComponent in the BasicTestApp (dotnet/aspnetcore reference source).

> **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 NavigationLock component intercepts navigation events as long as it is rendered, effectively "locking" any given navigation until a decision is made to either proceed or cancel. Use `NavigationLock` when navigation interception can be scoped to the lifetime of a component.

NavigationLock parameters:

- ConfirmExternalNavigation sets a browser dialog to prompt the user to either confirm or cancel external navigation. The default value is `false`. Displaying the confirmation dialog requires initial user interaction with

the page before triggering external navigation with the URL in the browser's address bar. For more information on the interaction requirement, see Window: beforeunload event (MDN documentation).
- OnBeforeInternalNavigation sets a callback for internal navigation events.

In the following `NavLock` component:

- An attempt to follow the link to Microsoft's website must be confirmed by the user before the navigation to `https://www.microsoft.com` succeeds.
- PreventNavigation is called to prevent navigation from occurring if the user declines to confirm the navigation via a JavaScript (JS) interop call that spawns the JS confirm dialog.

`NavLock.razor`:

razor

```razor
@page "/nav-lock"
@inject IJSRuntime JSRuntime
@inject NavigationManager Navigation

<NavigationLock ConfirmExternalNavigation="true"
    OnBeforeInternalNavigation="OnBeforeInternalNavigation" />

<p>
    <button @onclick="Navigate">Navigate</button>
</p>

<p>
    <a href="https://www.microsoft.com">Microsoft homepage</a>
</p>

@code {
    private void Navigate()
    {
        Navigation.NavigateTo("/");
    }

    private async Task OnBeforeInternalNavigation(LocationChangingContext context)
    {
        var isConfirmed = await JSRuntime.InvokeAsync<bool>("confirm",
            "Are you sure you want to navigate to the root page?");

        if (!isConfirmed)
        {
            context.PreventNavigation();
        }
    }
}
```

For additional example code, see the ConfigurableNavigationLock component in the BasicTestApp (dotnet/aspnetcore reference source).

`NavLink` component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path (`/`). The second NavLink receives the `active` class when the user visits any URL with a `component` prefix (for example, `/component` and `/component/another-segment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

razor

```
<NavLink href="example-page" target="_blank">Example page</NavLink>
```

The following HTML markup is rendered:

HTML

```
<a href="example-page" target="_blank">Example page</a>
```

> ⚠ **Warning**
>
> Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:
>
> razor
>
> ```
> @for (int c = 1; c < 4; c++)
> {
>     var ct = c;
>     <li ...>
>         <NavLink ...>
>             <span ...> Product #@ct
>         </NavLink>
>     </li>
> }
> ```
>
> Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.
>
> Alternatively, use a `foreach` loop with Enumerable.Range:
>
> razor
>
> ```
> @foreach (var c in Enumerable.Range(1, 3))
> {
>     <li ...>
>         <NavLink ...>
>             <span ...> Product #@c
>         </NavLink>
>     </li>
> }
> ```

NavLink component entries can be dynamically created from the app's components via reflection. The following example demonstrates the general approach for further customization.

For the following demonstration, a consistent, standard naming convention is used for the app's components:

- Routable component file names use Pascal case†, for example `Pages/ProductDetail.razor`.
- Routable component file paths 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.

In the Razor markup of the `NavMenu` component (`NavMenu.razor`) under the default `Home` page, NavLink components are added from a collection:

diff

```
<div class="nav-scrollable"
    onclick="document.querySelector('.navbar-toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu"
                    aria-hidden="true"></span> Home
            </NavLink>
        </div>

+       @foreach (var name in GetRoutableComponents())
+       {
+           <div class="nav-item px-3">
+               <NavLink class="nav-link"
+                       href="@Regex.Replace(name, @"(\B[A-Z]|\d+)", "-$1").ToLower()">
+                   @Regex.Replace(name, @"(\B[A-Z]|\d+)", " $1")
+               </NavLink>
+           </div>
+       }

    </nav>
</div>
```

The `GetRoutableComponents` method in the `@code` block:

C#

```
public IEnumerable<string> GetRoutableComponents() =>
    Assembly.GetExecutingAssembly()
        .ExportedTypes
        .Where(t => t.IsSubclassOf(typeof(ComponentBase)))
        .Where(c => c.GetCustomAttributes(inherit: true)
                     .OfType<RouteAttribute>()
                     .Any())
        .Where(c => c.Name != "Home" && c.Name != "Error")
        .OrderBy(o => o.Name)
        .Select(c => c.Name);
```

The preceding example doesn't include the following pages in the rendered list of components:

- `Home` page: The page is listed separately from the automatically generated links because it should appear at the top of the list and set the `Match` parameter.
- `Error` page: The error page is only navigated to by the framework and shouldn't be listed.

For an example of the preceding code in a sample app that you can run locally, obtain the **Blazor Web App** or **Blazor WebAssembly** sample app.

ASP.NET Core endpoint routing integration

This section applies to Blazor Web Apps operating over a circuit.

A Blazor Web App is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapRazorComponents in the `Program` file. The default root component (first component loaded) is the `App` component (`App.razor`):

```
C#

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

ASP.NET Core Blazor configuration

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 configure Blazor apps, including app settings, authentication, and logging configuration.

This guidance applies to client-side project configuration in a Blazor Web App or a standalone Blazor WebAssembly app.

Default behavior in Blazor Web Apps:

- For server-side configuration:
  - See Configuration in ASP.NET Core for guidance.
  - Only configuration in the project's root app settings files is loaded.
  - The remainder of this article only applies to client-side configuration in the `.Client` project.
- For client-side configuration (`.Client` project), configuration is loaded from the following app settings files:
  - `wwwroot/appsettings.json`.
  - `wwwroot/appsettings.{ENVIRONMENT}.json`, where the `{ENVIRONMENT}` placeholder is the app's runtime environment.

In standalone Blazor WebAssembly apps, configuration is loaded from the following app settings files:

- `wwwroot/appsettings.json`.
- `wwwroot/appsettings.{ENVIRONMENT}.json`, where the `{ENVIRONMENT}` placeholder is the app's runtime environment.

> **ⓘ Note**
>
> Logging configuration placed into an app settings file in `wwwroot` isn't loaded by default. For more information, see the **Logging configuration** section later in this

article.

In some scenarios, such as with Azure services, it's important to use an environment file name segment that exactly matches the environment name. For example, use the file name `appsettings.Staging.json` with a capital "S" for the `Staging` environment. For recommended conventions, see the opening remarks of ASP.NET Core Blazor environments.

Other configuration providers registered by the app can also provide configuration, but not all providers or provider features are appropriate:

- Azure Key Vault configuration provider: The provider isn't supported for managed identity and application ID (client ID) with client secret scenarios. Application ID with a client secret isn't recommended for any ASP.NET Core app, especially client-side apps because the client secret can't be secured client-side to access the Azure Key Vault service.
- Azure App configuration provider: The provider isn't appropriate for a client-side app because the app doesn't run on a server in Azure.

For more information on configuration providers, see Configuration in ASP.NET Core.

> ⚠ **Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

App settings configuration

Configuration in app settings files are loaded by default. In the following example, a UI configuration value is stored in an app settings file and loaded by the Blazor framework automatically. The value is read by a component.

`wwwroot/appsettings.json`:

JSON

```
{
    "h1FontSize": "50px"
}
```

Inject an IConfiguration instance into a component to access the configuration data.

`ConfigExample.razor`:

```razor
@page "/config-example"
@inject IConfiguration Configuration

<PageTitle>Configuration</PageTitle>

<h1 style="font-size:@Configuration["h1FontSize"]">
    Configuration example (50px)
</h1>
```

Client security restrictions prevent direct access to files via user code, including settings files for app configuration. To read configuration files in addition to `appsettings.json`/`appsettings.{ENVIRONMENT}.json` from the `wwwroot` folder into configuration, use an HttpClient.

> ⚠ **Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

The following example reads a configuration file (`cars.json`) into the app's configuration.

`wwwroot/cars.json`:

```json
{
    "size": "tiny"
}
```

Add the namespace for Microsoft.Extensions.Configuration to the `Program` file:

```C#
using Microsoft.Extensions.Configuration;
```

Modify the existing HttpClient service registration to use the client to read the file:

C#

```
var http = new HttpClient()
{
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
};

builder.Services.AddScoped(sp => http);

using var response = await http.GetAsync("cars.json");
using var stream = await response.Content.ReadAsStreamAsync();

builder.Configuration.AddJsonStream(stream);
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

Memory Configuration Source

The following example uses a MemoryConfigurationSource in the `Program` file to supply additional configuration.

Add the namespace for Microsoft.Extensions.Configuration.Memory to the `Program` file:

C#

```
using Microsoft.Extensions.Configuration.Memory;
```

In the `Program` file:

C#

```
var vehicleData = new Dictionary<string, string?>()
{
    { "color", "blue" },
    { "type", "car" },
    { "wheels:count", "3" },
    { "wheels:brand", "Blazin" },
    { "wheels:brand:type", "rally" },
    { "wheels:year", "2008" },
};

var memoryConfig = new MemoryConfigurationSource { InitialData = vehicleData
};
```

```
builder.Configuration.Add(memoryConfig);
```

Inject an IConfiguration instance into a component to access the configuration data.

`MemoryConfig.razor`:

razor

```razor
@page "/memory-config"
@inject IConfiguration Configuration

<PageTitle>Memory Configuration</PageTitle>

<h1>Memory Configuration Example</h1>

<h2>General specifications</h2>

<ul>
    <li>Color: @Configuration["color"]</li>
    <li>Type: @Configuration["type"]</li>
</ul>

<h2>Wheels</h2>

<ul>
    <li>Count: @Configuration["wheels:count"]</li>
    <li>Brand: @Configuration["wheels:brand"]</li>
    <li>Type: @Configuration["wheels:brand:type"]</li>
    <li>Year: @Configuration["wheels:year"]</li>
</ul>
```

Obtain a section of the configuration in C# code with IConfiguration.GetSection. The following example obtains the `wheels` section for the configuration in the preceding example:

razor

```razor
@code {
    protected override void OnInitialized()
    {
        var wheelsSection = Configuration.GetSection("wheels");

        ...
    }
}
```

Authentication configuration

Provide ***public*** authentication configuration in an app settings file.

`wwwroot/appsettings.json`:

JSON

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

Load the configuration for an Identity provider with ConfigurationBinder.Bind in the `Program` file. The following example loads configuration for an OIDC provider:

C#

```
builder.Services.AddOidcAuthentication(options =>
    builder.Configuration.Bind("Local", options.ProviderOptions));
```

> ⚠ **Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

Logging configuration

This section applies to apps that configure logging via an app settings file in the `wwwroot` folder.

Add the Microsoft.Extensions.Logging.Configuration package to the app.

> ⓘ **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the app settings file, provide logging configuration. The logging configuration is loaded in the `Program` file.

`wwwroot/appsettings.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

In the `Program` file:

C#

```
builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("Logging"));
```

Host builder configuration

Read host builder configuration from WebAssemblyHostBuilder.Configuration in the `Program` file:

C#

```
var hostname = builder.Configuration["HostName"];
```

Cached configuration

Configuration files are cached for offline use. With Progressive Web Applications (PWAs), you can only update configuration files when creating a new deployment. Editing configuration files between deployments has no effect because:

- Users have cached versions of the files that they continue to use.
- The PWA's `service-worker.js` and `service-worker-assets.js` files must be rebuilt on compilation, which signal to the app on the user's next online visit that the app has been redeployed.

For more information on how background updates are handled by PWAs, see ASP.NET Core Blazor Progressive Web Application (PWA).

Options configuration

Options configuration requires adding a package reference for the Microsoft.Extensions.Options.ConfigurationExtensions NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Example:

```
C#

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));
```

Not all of the ASP.NET Core Options features are supported in Razor components. For example, IOptionsSnapshot<TOptions> and IOptionsMonitor<TOptions> configuration is supported, but recomputing option values for these interfaces isn't supported outside of reloading the app by either requesting the app in a new browser tab or selecting the browser's reload button. Merely calling StateHasChanged doesn't update snapshot or monitored option values when the underlying configuration changes.

ASP.NET Core Blazor dependency injection

Article • 10/04/2024

> **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 Rainer Stropek and Mike Rousos

This article explains how Blazor apps can inject services into components.

Dependency injection (DI) is a technique for accessing services configured in a central location:

- Framework-registered services can be injected directly into Razor components.
- Blazor apps define and register custom services and make them available throughout the app via DI.

> **Note**
>
> We recommend reading Dependency injection in ASP.NET Core before reading this topic.

Default services

The services shown in the following table are commonly used in Blazor apps.

Expand table

Service	Lifetime	Description
HttpClient	Scoped	Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.

Service	Lifetime	Description
		Client-side, an instance of HttpClient is registered by the app in the `Program` file and uses the browser for handling the HTTP traffic in the background. Server-side, an HttpClient isn't configured as a service by default. In server-side code, provide an HttpClient. For more information, see Call a web API from an ASP.NET Core Blazor app. An HttpClient is registered as a scoped service, not singleton. For more information, see the Service lifetime section.
IJSRuntime	**Client-side**: Singleton **Server-side**: Scoped The Blazor framework registers IJSRuntime in the app's service container.	Represents an instance of a JavaScript runtime where JavaScript calls are dispatched. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor. When seeking to inject the service into a singleton service on the server, take either of the following approaches: • Change the service registration to scoped to match IJSRuntime's registration, which is appropriate if the service deals with user-specific state. • Pass the IJSRuntime into the singleton service's implementation as an argument of its method calls instead of injecting it into the singleton.
NavigationManager	**Client-side**: Singleton **Server-side**: Scoped The Blazor framework registers NavigationManager in the app's service container.	Contains helpers for working with URIs and navigation state. For more information, see URI and navigation state helpers.

Additional services registered by the Blazor framework are described in the documentation where they're used to describe Blazor features, such as configuration and logging.

A custom service provider doesn't automatically provide the default services listed in the table. If you use a custom service provider and require any of the services shown in the table, add the required services to the new service provider.

Add client-side services

Configure services for the app's service collection in the `Program` file. In the following example, the `ExampleDependency` implementation is registered for `IExampleDependency`:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<IExampleDependency, ExampleDependency>();
...

await builder.Build().RunAsync();
```

After the host is built, services are available from the root DI scope before any components are rendered. This can be useful for running initialization logic before rendering content:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<WeatherService>();
...

var host = builder.Build();

var weatherService = host.Services.GetRequiredService<WeatherService>();
await weatherService.InitializeWeatherAsync();

await host.RunAsync();
```

The host provides a central configuration instance for the app. Building on the preceding example, the weather service's URL is passed from a default configuration source (for example, `appsettings.json`) to `InitializeWeatherAsync`:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<WeatherService>();
...
```

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

var weatherService = host.Services.GetRequiredService<WeatherService>();
await weatherService.InitializeWeatherAsync(
    host.Configuration["WeatherServiceUrl"]);

await host.RunAsync();
```

Add server-side services

After creating a new app, examine part of the `Program` file:

C#

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

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

The `builder` variable represents a WebApplicationBuilder with an IServiceCollection, which is a list of service descriptor objects. Services are added by providing service descriptors to the service collection. The following example demonstrates the concept with the `IDataAccess` interface and its concrete implementation `DataAccess`:

C#

```
builder.Services.AddSingleton<IDataAccess, DataAccess>();
```

Register common services

If one or more common services are required client- and server-side, you can place the common service registrations in a method client-side and call the method to register the services in both projects.

First, factor common service registrations into a separate method. For example, create a `ConfigureCommonServices` method client-side:

C#

```
public static void ConfigureCommonServices(IServiceCollection services)
{
```

```
    services.Add...;
}
```

For the client-side `Program` file, call `ConfigureCommonServices` to register the common services:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);

...

ConfigureCommonServices(builder.Services);
```

In the server-side `Program` file, call `ConfigureCommonServices` to register the common services:

C#

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

...

Client.Program.ConfigureCommonServices(builder.Services);
```

For an example of this approach, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Client-side services that fail during prerendering

This section only applies to WebAssembly components in Blazor Web Apps.

Blazor Web Apps normally prerender client-side WebAssembly components. If an app is run with a required service only registered in the `.Client` project, executing the app results in a runtime error similar to the following when a component attempts to use the required service during prerendering:

> InvalidOperationException: Cannot provide a value for {PROPERTY} on type '{ASSEMBLY}}.Client.Pages.{COMPONENT NAME}'. There is no registered service of type '{SERVICE}'.

Use ***either*** of the following approaches to resolve this problem:

- Register the service in the main project to make it available during component prerendering.
- If prerendering isn't required for the component, disable prerendering by following the guidance in ASP.NET Core Blazor render modes. If you adopt this approach, you don't need to register the service in the main project.

For more information, see Client-side services fail to resolve during prerendering.

Service lifetime

Services can be configured with the lifetimes shown in the following table.

Expand table

Lifetime	Description
Scoped	Client-side doesn't currently have a concept of DI scopes. `Scoped`-registered services behave like `Singleton` services. Server-side development supports the `Scoped` lifetime across HTTP requests but not across SignalR connection/circuit messages among components that are loaded on the client. The Razor Pages or MVC portion of the app treats scoped services normally and recreates the services on *each HTTP request* when navigating among pages or views or from a page or view to a component. Scoped services aren't reconstructed when navigating among components on the client, where the communication to the server takes place over the SignalR connection of the user's circuit, not via HTTP requests. In the following component scenarios on the client, scoped services are reconstructed because a new circuit is created for the user: • The user closes the browser's window. The user opens a new window and navigates back to the app. • The user closes a tab of the app in a browser window. The user opens a new tab and navigates back to the app. • The user selects the browser's reload/refresh button. For more information on preserving user state in server-side apps, see ASP.NET Core Blazor state management.
Singleton	DI creates a *single instance* of the service. All components requiring a `Singleton` service receive the same instance of the service.
Transient	Whenever a component obtains an instance of a `Transient` service from the service container, it receives a *new instance* of the service.

The DI system is based on the DI system in ASP.NET Core. For more information, see Dependency injection in ASP.NET Core.

Request a service in a component

For injecting services into components, Blazor supports constructor injection and property injection.

Constructor injection

After services are added to the service collection, inject one or more services into components with constructor injection. The following example injects the `NavigationManager` service.

`ConstructorInjection.razor`:

razor

```razor
@page "/constructor-injection"

<button @onclick="HandleClick">
    Take me to the Counter component
</button>
```

`ConstructorInjection.razor.cs`:

C#

```
using Microsoft.AspNetCore.Components;

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

Property injection

After services are added to the service collection, inject one or more services into components with the @inject Razor directive, which has two parameters:

- Type: The type of the service to inject.
- Property: The name of the property receiving the injected app service. The property doesn't require manual creation. The compiler creates the property.

For more information, see Dependency injection into views in ASP.NET Core.

Use multiple @inject statements to inject different services.

The following example demonstrates shows how to use the @inject directive. The service implementing `Services.NavigationManager` is injected into the component's property `Navigation`. Note how the code is only using the `NavigationManager` abstraction.

`PropertyInjection.razor`:

razor

```razor
@page "/property-injection"
@inject NavigationManager Navigation

<button @onclick="@(() => Navigation.NavigateTo("/counter"))">
    Take me to the Counter component
</button>
```

Internally, the generated property (`Navigation`) uses the [Inject] attribute. Typically, this attribute isn't used directly. If a base class is required for components and injected properties are also required for the base class, manually add the [Inject] attribute:

C#

```
using Microsoft.AspNetCore.Components;

public class ComponentBase : IComponent
{
    [Inject]
    protected NavigationManager Navigation { get; set; } = default!;

    ...
}
```

> **Note**
>
> Since injected services are expected to be available, the default literal with the null-forgiving operator (`default!`) is assigned in .NET 6 or later. For more information, see Nullable reference types (NRTs) and .NET compiler null-state static analysis.

In components derived from a base class, the @inject directive isn't required. The InjectAttribute of the base class is sufficient. The component only requires the @inherits directive. In the following example, any injected services of `CustomComponentBase` are available to the `Demo` component:

razor

```
@page "/demo"
@inherits CustomComponentBase
```

Use DI in services

Complex services might require additional services. In the following example, `DataAccess` requires the HttpClient default service. @inject (or the [Inject] attribute) isn't available for use in services. *Constructor injection* must be used instead. Required services are added by adding parameters to the service's constructor. When DI creates the service, it recognizes the services it requires in the constructor and provides them accordingly. In the following example, the constructor receives an HttpClient via DI. HttpClient is a default service.

C#

```
using System.Net.Http;

public class DataAccess : IDataAccess
{
    public DataAccess(HttpClient http)
    {
        ...
    }

    ...
}
```

Constructor injection is supported with primary constructors in C# 12 (.NET 8) or later:

C#

```
using System.Net.Http;

public class DataAccess(HttpClient http) : IDataAccess
{
    ...
}
```

Prerequisites for constructor injection:

- One constructor must exist whose arguments can all be fulfilled by DI. Additional parameters not covered by DI are allowed if they specify default values.
- The applicable constructor must be `public`.

- One applicable constructor must exist. In case of an ambiguity, DI throws an exception.

Inject keyed services into components

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

```
C#

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

Utility base component classes to manage a DI scope

In non-Blazor ASP.NET Core apps, scoped and transient services are typically scoped to the current request. After the request completes, scoped and transient services are disposed by the DI system.

In interactive server-side Blazor apps, the DI scope lasts for the duration of the circuit (the SignalR connection between the client and server), which can result in scoped and disposable transient services living much longer than the lifetime of a single component. Therefore, don't directly inject a scoped service into a component if you intend the service lifetime to match the lifetime of the component. Transient services injected into a component that don't implement IDisposable are garbage collected when the component is disposed. However, injected transient services *that implement IDisposable* are maintained by the DI container for the lifetime of the circuit, which prevents service garbage collection when the component is disposed and results in a memory leak. An alternative approach for scoped services based on the OwningComponentBase type is described later in this section, and disposable transient services shouldn't be used at all. For more information, see Design for solving transient disposables on Blazor Server (dotnet/aspnetcore #26676).

Even in client-side Blazor apps that don't operate over a circuit, services registered with a scoped lifetime are treated as singletons, so they live longer than scoped services in typical ASP.NET Core apps. Client-side disposable transient services also live longer than the components where they're injected because the DI container, which holds references to disposable services, persists for the lifetime of the app, preventing garbage collection

on the services. Although long-lived disposable transient services are of greater concern on the server, they should be avoided as client service registrations as well. Use of the OwningComponentBase type is also recommended for client-side scoped services to control service lifetime, and disposable transient services shouldn't be used at all.

An approach that limits a service lifetime is use of the OwningComponentBase type. OwningComponentBase is an abstract type derived from ComponentBase that creates a DI scope corresponding to the *lifetime of the component*. Using this scope, a component can inject services with a scoped lifetime and have them live as long as the component. When the component is destroyed, services from the component's scoped service provider are disposed as well. This can be useful for services reused within a component but not shared across components.

Two versions of OwningComponentBase type are available and described in the next two sections:

- OwningComponentBase
- OwningComponentBase<TService>

`OwningComponentBase`

OwningComponentBase is an abstract, disposable child of the ComponentBase type with a protected ScopedServices property of type IServiceProvider. The provider can be used to resolve services that are scoped to the lifetime of the component.

DI services injected into the component using @inject or the [Inject] attribute aren't created in the component's scope. To use the component's scope, services must be resolved using ScopedServices with either GetRequiredService or GetService. Any services resolved using the ScopedServices provider have their dependencies provided in the component's scope.

The following example demonstrates the difference between injecting a scoped service directly and resolving a service using ScopedServices on the server. The following interface and implementation for a time travel class include a `DT` property to hold a DateTime value. The implementation calls DateTime.Now to set `DT` when the `TimeTravel` class is instantiated.

`ITimeTravel.cs`:

C#

```
public interface ITimeTravel
{
```

```
    public DateTime DT { get; set; }
}
```

`TimeTravel.cs`:

C#

```
public class TimeTravel : ITimeTravel
{
    public DateTime DT { get; set; } = DateTime.Now;
}
```

The service is registered as scoped in the server-side `Program` file. Server-side, scoped services have a lifetime equal to the duration of the circuit.

In the `Program` file:

C#

```
builder.Services.AddScoped<ITimeTravel, TimeTravel>();
```

In the following `TimeTravel` component:

- The time travel service is directly injected with `@inject` as `TimeTravel1`.
- The service is also resolved separately with ScopedServices and GetRequiredService as `TimeTravel2`.

`TimeTravel.razor`:

razor

```
@page "/time-travel"
@inject ITimeTravel TimeTravel1
@inherits OwningComponentBase

<h1><code>OwningComponentBase</code> Example</h1>

<ul>
    <li>TimeTravel1.DT: @TimeTravel1?.DT</li>
    <li>TimeTravel2.DT: @TimeTravel2?.DT</li>
</ul>

@code {
    private ITimeTravel TimeTravel2 { get; set; } = default!;

    protected override void OnInitialized()
    {
        TimeTravel2 = ScopedServices.GetRequiredService<ITimeTravel>();
```

```
    }
}
```

Initially navigating to the `TimeTravel` component, the time travel service is instantiated twice when the component loads, and `TimeTravel1` and `TimeTravel2` have the same initial value:

> TimeTravel1.DT: 8/31/2022 2:54:45 PM
> TimeTravel2.DT: 8/31/2022 2:54:45 PM

When navigating away from the `TimeTravel` component to another component and back to the `TimeTravel` component:

- `TimeTravel1` is provided the same service instance that was created when the component first loaded, so the value of `DT` remains the same.
- `TimeTravel2` obtains a new `ITimeTravel` service instance in `TimeTravel2` with a new DT value.

> TimeTravel1.DT: 8/31/2022 2:54:45 PM
> TimeTravel2.DT: 8/31/2022 2:54:48 PM

`TimeTravel1` is tied to the user's circuit, which remains intact and isn't disposed until the underlying circuit is deconstructed. For example, the service is disposed if the circuit is disconnected for the disconnected circuit retention period.

In spite of the scoped service registration in the `Program` file and the longevity of the user's circuit, `TimeTravel2` receives a new `ITimeTravel` service instance each time the component is initialized.

`OwningComponentBase<TService>`

OwningComponentBase<TService> derives from OwningComponentBase and adds a Service property that returns an instance of `T` from the scoped DI provider. This type is a convenient way to access scoped services without using an instance of IServiceProvider when there's one primary service the app requires from the DI container using the component's scope. The ScopedServices property is available, so the app can get services of other types, if necessary.

```
razor
```

```
@page "/users"
@attribute [Authorize]
@inherits OwningComponentBase<AppDbContext>

<h1>Users (@Service.Users.Count())</h1>

<ul>
    @foreach (var user in Service.Users)
    {
        <li>@user.UserName</li>
    }
</ul>
```

Detect client-side transient disposables

Custom code can be added to a client-side Blazor app to detect disposable transient services in an app that should use OwningComponentBase. This approach is useful if you're concerned that code added to the app in the future consumes one or more transient disposable services, including services added by libraries. Demonstration code is available in the Blazor samples GitHub repository (how to download).

Inspect the following in .NET 6 or later versions of the `BlazorSample_WebAssembly` sample:

- `DetectIncorrectUsagesOfTransientDisposables.cs`
- `Services/TransientDisposableService.cs`
- In `Program.cs`:
  - The app's `Services` namespace is provided at the top of the file (`using BlazorSample.Services;`).
  - `DetectIncorrectUsageOfTransients` is called immediately after the `builder` is assigned from WebAssemblyHostBuilder.CreateDefault.
  - The `TransientDisposableService` is registered (`builder.Services.AddTransient<TransientDisposableService>();`).
  - `EnableTransientDisposableDetection` is called on the built host in the processing pipeline of the app (`host.EnableTransientDisposableDetection();`).
- The app registers the `TransientDisposableService` service without throwing an exception. However, attempting to resolve the service in `TransientService.razor` throws an InvalidOperationException when the framework attempts to construct an instance of `TransientDisposableService`.

Detect server-side transient disposables

Custom code can be added to a server-side Blazor app to detect server-side disposable transient services in an app that should use OwningComponentBase. This approach is useful if you're concerned that code added to the app in the future consumes one or more transient disposable services, including services added by libraries. Demonstration code is available in the Blazor samples GitHub repository (how to download).

Inspect the following in .NET 8 or later versions of the `BlazorSample_BlazorWebApp` sample:

- `DetectIncorrectUsagesOfTransientDisposables.cs`
- `Services/TransitiveTransientDisposableDependency.cs`:
- In `Program.cs`:
  - The app's `Services` namespace is provided at the top of the file (`using BlazorSample.Services;`).
  - `DetectIncorrectUsageOfTransients` is called on the host builder (`builder.DetectIncorrectUsageOfTransients();`).
  - The `TransientDependency` service is registered (`builder.Services.AddTransient<TransientDependency>();`).
  - The `TransitiveTransientDisposableDependency` is registered for `ITransitiveTransientDisposableDependency` (`builder.Services.AddTransient<ITransitiveTransientDisposableDependency, TransitiveTransientDisposableDependency>();`).
- The app registers the `TransientDependency` service without throwing an exception. However, attempting to resolve the service in `TransientService.razor` throws an InvalidOperationException when the framework attempts to construct an instance of `TransientDependency`.

Transient service registrations for `IHttpClientFactory`/`HttpClient` handlers

Transient service registrations for IHttpClientFactory/HttpClient handlers are recommended. If the app contains IHttpClientFactory/HttpClient handlers and uses the IRemoteAuthenticationBuilder<TRemoteAuthenticationState,TAccount> to add support for authentication, the following transient disposables for client-side authentication are also discovered, which is expected and can be ignored:

- BaseAddressAuthorizationMessageHandler
- AuthorizationMessageHandler

Other instances of IHttpClientFactory/HttpClient are also discovered. These instances can also be ignored.

The Blazor sample apps in the Blazor samples GitHub repository (how to download) demonstrate the code to detect transient disposables. However, the code is deactivated because the sample apps include IHttpClientFactory/HttpClient handlers.

To activate the demonstration code and witness its operation:

- Uncomment the transient disposable lines in `Program.cs`.
- Remove the conditional check in `NavLink.razor` that prevents the `TransientService` component from displaying in the app's navigation sidebar:

  diff

  ```
  - else if (name != "TransientService")
  + else
  ```

- Run the sample app and navigate to the `TransientService` component at `/transient-service`.

Use of an Entity Framework Core (EF Core) DbContext from DI

For more information, see ASP.NET Core Blazor with Entity Framework Core (EF Core).

Access server-side Blazor services from a different DI scope

Circuit activity handlers provide an approach for accessing scoped Blazor services from other non-Blazor dependency injection (DI) scopes, such as scopes created using IHttpClientFactory.

Prior to the release of ASP.NET Core in .NET 8, accessing circuit-scoped services from other dependency injection scopes required using a custom base component type. With circuit activity handlers, a custom base component type isn't required, as the following example demonstrates:

C#

```
public class CircuitServicesAccessor
{
    static readonly AsyncLocal<IServiceProvider> blazorServices = new();

    public IServiceProvider? Services
    {
        get => blazorServices.Value;
        set => blazorServices.Value = value;
    }
}

public class ServicesAccessorCircuitHandler(
    IServiceProvider services, CircuitServicesAccessor servicesAccessor)
    : CircuitHandler
{
    public override Func<CircuitInboundActivityContext, Task>
CreateInboundActivityHandler(
        Func<CircuitInboundActivityContext, Task> next) =>
            async context =>
            {
                servicesAccessor.Services = services;
                await next(context);
                servicesAccessor.Services = null;
            };
}

public static class CircuitServicesServiceCollectionExtensions
{
    public static IServiceCollection AddCircuitServicesAccessor(
        this IServiceCollection services)
    {
        services.AddScoped<CircuitServicesAccessor>();
        services.AddScoped<CircuitHandler, ServicesAccessorCircuitHandler>
();

        return services;
    }
}
```

Access the circuit-scoped services by injecting the `CircuitServicesAccessor` where it's needed.

For an example that shows how to access the AuthenticationStateProvider from a DelegatingHandler set up using IHttpClientFactory, see Server-side ASP.NET Core Blazor additional security scenarios.

Additional resources

- Service injection via a top-level imports file (_Imports.razor) in Blazor Web Apps

- Dependency injection in ASP.NET Core
- IDisposable guidance for Transient and shared instances
- Dependency injection into views in ASP.NET Core
- Primary constructors (C# Guide)

ASP.NET Core Blazor startup

Article • 09/27/2024

> **ⓘ 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 app startup configuration.

For general guidance on ASP.NET Core app configuration for server-side development, see Configuration in ASP.NET Core.

Startup process and configuration

The Blazor startup process is automatic and asynchronous via the Blazor script (`blazor.*.js`), where the `*` placeholder is:

- `web` for a Blazor Web App
- `server` for a Blazor Server app
- `webassembly` for a Blazor WebAssembly app

For the location of the script, see ASP.NET Core Blazor project structure.

To manually start Blazor:

Blazor Web App:

- Add an `autostart="false"` attribute and value to the Blazor `<script>` tag.
- Place a script that calls `Blazor.start()` after the Blazor `<script>` tag and inside the closing `</body>` tag.
- Place static server-side rendering (static SSR) options in the `ssr` property.
- Place server-side Blazor-SignalR circuit options in the `circuit` property.
- Place client-side WebAssembly options in the `webAssembly` property.

```
HTML
```

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  ...
  Blazor.start({
    ssr: {
      ...
    },
    circuit: {
      ...
    },
    webAssembly: {
      ...
    }
  });
  ...
</script>
```

Standalone Blazor WebAssembly and Blazor Server:

- Add an `autostart="false"` attribute and value to the Blazor `<script>` tag.
- Place a script that calls `Blazor.start()` after the Blazor `<script>` tag and inside the closing `</body>` tag.
- You can provide additional options in the `Blazor.start()` parameter.

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  ...
  Blazor.start({
    ...
  });
  ...
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

JavaScript initializers

JavaScript (JS) initializers execute logic before and after a Blazor app loads. JS initializers are useful in the following scenarios:

- Customizing how a Blazor app loads.
- Initializing libraries before Blazor starts up.
- Configuring Blazor settings.

JS initializers are detected as part of the build process and imported automatically. Use of JS initializers often removes the need to manually trigger script functions from the app when using Razor class libraries (RCLs).

To define a JS initializer, add a JS module to the project named `{NAME}.lib.module.js`, where the `{NAME}` placeholder is the assembly name, library name, or package identifier. Place the file in the project's web root, which is typically the `wwwroot` folder.

For Blazor Web Apps:

- `beforeWebStart(options)`: Called before the Blazor Web App starts. For example, `beforeWebStart` is used to customize the loading process, logging level, and other options. Receives the Blazor Web options (`options`).
- `afterWebStarted(blazor)`: Called after all `beforeWebStart` promises resolve. For example, `afterWebStarted` can be used to register Blazor event listeners and custom event types. The Blazor instance is passed to `afterWebStarted` as an argument (`blazor`).
- `beforeServerStart(options, extensions)`: Called before the first Server runtime is started. Receives SignalR circuit start options (`options`) and any extensions (`extensions`) added during publishing.
- `afterServerStarted(blazor)`: Called after the first Interactive Server runtime is started.
- `beforeWebAssemblyStart(options, extensions)`: Called before the Interactive WebAssembly runtime is started. Receives the Blazor options (`options`) and any extensions (`extensions`) added during publishing. For example, options can specify the use of a custom boot resource loader.
- `afterWebAssemblyStarted(blazor)`: Called after the Interactive WebAssembly runtime is started.

> **Note**
>
> Legacy JS initializers (`beforeStart`, `afterStarted`) aren't invoked by default in a Blazor Web App. You can enable the legacy initializers to run with the `enableClassicInitializers` option. However, legacy initializer execution is unpredictable.

```
HTML
```

```
<script>
  Blazor.start({ enableClassicInitializers: true });
</script>
```

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

- `beforeStart(options, extensions)`: Called before Blazor starts. For example, `beforeStart` is used to customize the loading process, logging level, and other options specific to the hosting model.
  - Client-side, `beforeStart` receives the Blazor options (`options`) and any extensions (`extensions`) added during publishing. For example, options can specify the use of a custom boot resource loader.
  - Server-side, `beforeStart` receives SignalR circuit start options (`options`).
  - In a BlazorWebView, no options are passed.
- `afterStarted(blazor)`: Called after Blazor is ready to receive calls from JS. For example, `afterStarted` is used to initialize libraries by making JS interop calls and registering custom elements. The Blazor instance is passed to `afterStarted` as an argument (`blazor`).

Additional .NET WebAssembly runtime callbacks:

- `onRuntimeConfigLoaded(config)`: Called when the boot configuration is downloaded. Allows the app to modify parameters (config) before the runtime starts (the parameter is `MonoConfig` from dotnet.d.ts):

  JavaScript

  ```
  export function onRuntimeConfigLoaded(config) {
    // Sample: Enable startup diagnostic logging when the URL contains
    // parameter debug=1
    const params = new URLSearchParams(location.search);
    if (params.get("debug") == "1") {
      config.diagnosticTracing = true;
    }
  }
  ```

- `onRuntimeReady({ getAssemblyExports, getConfig })`: Called after the .NET WebAssembly runtime has started (the parameter is `RuntimeAPI` from dotnet.d.ts):

  JavaScript

```
export function onRuntimeReady({ getAssemblyExports, getConfig }) {
  // Sample: After the runtime starts, but before Main method is
called,
  // call [JSExport]ed method.
  const config = getConfig();
  const exports = await getAssemblyExports(config.mainAssemblyName);
  exports.Sample.Greet();
}
```

Both callbacks can return a `Promise`, and the promise is awaited before the startup continues.

For the file name:

- If the JS initializers are consumed as a static asset in the project, use the format `{ASSEMBLY NAME}.lib.module.js`, where the `{ASSEMBLY NAME}` placeholder is the app's assembly name. For example, name the file `BlazorSample.lib.module.js` for a project with an assembly name of `BlazorSample`. Place the file in the app's `wwwroot` folder.
- If the JS initializers are consumed from an RCL, use the format `{LIBRARY NAME/PACKAGE ID}.lib.module.js`, where the `{LIBRARY NAME/PACKAGE ID}` placeholder is the project's library name or package identifier. For example, name the file `RazorClassLibrary1.lib.module.js` for an RCL with a package identifier of `RazorClassLibrary1`. Place the file in the library's `wwwroot` folder.

For Blazor Web Apps:

The following example demonstrates JS initializers that load custom scripts before and after the Blazor Web App has started by appending them to the `<head>` in `beforeWebStart` and `afterWebStarted`:

JavaScript

```
export function beforeWebStart() {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'beforeStartScripts.js');
  document.head.appendChild(customScript);
}

export function afterWebStarted() {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'afterStartedScripts.js');
  document.head.appendChild(customScript);
}
```

The preceding `beforeWebStart` example only guarantees that the custom script loads before Blazor starts. It doesn't guarantee that awaited promises in the script complete their execution before Blazor starts.

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

The following example demonstrates JS initializers that load custom scripts before and after Blazor has started by appending them to the `<head>` in `beforeStart` and `afterStarted`:

JavaScript

```
export function beforeStart(options, extensions) {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'beforeStartScripts.js');
  document.head.appendChild(customScript);
}

export function afterStarted(blazor) {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'afterStartedScripts.js');
  document.head.appendChild(customScript);
}
```

The preceding `beforeStart` example only guarantees that the custom script loads before Blazor starts. It doesn't guarantee that awaited promises in the script complete their execution before Blazor starts.

> **Note**
>
> MVC and Razor Pages apps don't automatically load JS initializers. However, developer code can include a script to fetch the app's manifest and trigger the load of the JS initializers.

For examples of JS initializers, see the following resources:

- ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)
- Use Razor components in JavaScript apps and SPA frameworks (*`quoteContainer2` example*)
- ASP.NET Core Blazor event handling (*Custom clipboard paste event example*)
- Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App
- Basic Test App in the ASP.NET Core GitHub repository (BasicTestApp.lib.module.js)

> **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).

Ensure libraries are loaded in a specific order

Append custom scripts to the `<head>` in `beforeStart` and `afterStarted` in the order that they should load.

The following example loads `script1.js` before `script2.js` and `script3.js` before `script4.js`:

JavaScript

```
export function beforeStart(options, extensions) {
    var customScript1 = document.createElement('script');
    customScript1.setAttribute('src', 'script1.js');
    document.head.appendChild(customScript1);

    var customScript2 = document.createElement('script');
    customScript2.setAttribute('src', 'script2.js');
    document.head.appendChild(customScript2);
}

export function afterStarted(blazor) {
    var customScript1 = document.createElement('script');
    customScript1.setAttribute('src', 'script3.js');
    document.head.appendChild(customScript1);

    var customScript2 = document.createElement('script');
    customScript2.setAttribute('src', 'script4.js');
    document.head.appendChild(customScript2);
}
```

Import additional modules

Use top-level import statements in the JS initializers file to import additional modules.

`additionalModule.js`:

JavaScript

```
export function logMessage() {
  console.log('logMessage is logging');
}
```

In the JS initializers file (`.lib.module.js`):

JavaScript

```
import { logMessage } from "/additionalModule.js";

export function beforeStart(options, extensions) {
  ...

  logMessage();
}
```

Import map

Import maps are supported by ASP.NET Core and Blazor.

Initialize Blazor when the document is ready

The following example starts Blazor when the document is ready:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  document.addEventListener("DOMContentLoaded", function() {
    Blazor.start();
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use then to chain to the Promise that results from a manual Blazor app start:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start().then(function () {
    ...
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> ⓘ Note
>
> For a library to automatically execute additional tasks after Blazor has started, use a JavaScript initializer. Use of a JS initializer doesn't require the consumer of the library to chain JS calls to Blazor's manual start.

Load client-side boot resources

When an app loads in the browser, the app downloads boot resources from the server:

- JavaScript code to bootstrap the app
- .NET runtime and assemblies
- Locale specific data

Customize how these boot resources are loaded using the `loadBootResource` API. The `loadBootResource` function overrides the built-in boot resource loading mechanism. Use `loadBootResource` for the following scenarios:

- Load static resources, such as timezone data or `dotnet.wasm`, from a CDN.
- Load compressed assemblies using an HTTP request and decompress them on the client for hosts that don't support fetching compressed contents from the server.
- Alias resources to a different name by redirecting each `fetch` request to a new name.

> ⓘ Note
>
> External sources must return the required Cross-Origin Resource Sharing (CORS) headers for browsers to allow cross-origin resource loading. CDNs usually provide the required headers.

`loadBootResource` parameters appear in the following table.

Expand table

Parameter	Description
`type`	The type of the resource. Permissible types include: `assembly`, `pdb`, `dotnetjs`, `dotnetwasm`, and `timezonedata`. You only need to specify types for custom behaviors. Types not specified to `loadBootResource` are loaded by the framework per their default loading behaviors. The `dotnetjs` boot resource (`dotnet.*.js`) must either return `null` for default loading behavior or a URI for the source of the `dotnetjs` boot resource.
`name`	The name of the resource.
`defaultUri`	The relative or absolute URI of the resource.
`integrity`	The integrity string representing the expected content in the response.

The `loadBootResource` function can return a URI string to override the loading process. In the following example, the following files from `bin/Release/{TARGET FRAMEWORK}/wwwroot/_framework` are served from a CDN at `https://cdn.example.com/blazorwebassembly/{VERSION}/`:

- `dotnet.*.js`
- `dotnet.wasm`
- Timezone data

The `{TARGET FRAMEWORK}` placeholder is the target framework moniker (for example, `net7.0`). The `{VERSION}` placeholder is the shared framework version (for example, `7.0.0`).

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        console.log(`Loading: '${type}', '${name}', '${defaultUri}',
'${integrity}'`);
        switch (type) {
          case 'dotnetjs':
          case 'dotnetwasm':
          case 'timezonedata':
```

```
            return
`https://cdn.example.com/blazorwebassembly/{VERSION}/${name}`;
        }
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      console.log(`Loading: '${type}', '${name}', '${defaultUri}',
'${integrity}'`);
      switch (type) {
        case 'dotnetjs':
        case 'dotnetwasm':
        case 'timezonedata':
          return
`https://cdn.example.com/blazorwebassembly/{VERSION}/${name}`;
      }
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

To customize more than just the URLs for boot resources, the `loadBootResource` function can call `fetch` directly and return the result. The following example adds a custom HTTP header to the outbound requests. To retain the default integrity checking behavior, pass through the `integrity` parameter.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        if (type == 'dotnetjs') {
          return null;
        } else {
```

```
          return fetch(defaultUri, {
            cache: 'no-cache',
            integrity: integrity,
            headers: { 'Custom-Header': 'Custom Value' }
          });
        }
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      if (type == 'dotnetjs') {
        return null;
      } else {
        return fetch(defaultUri, {
          cache: 'no-cache',
          integrity: integrity,
          headers: { 'Custom-Header': 'Custom Value' }
        });
      }
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

When the `loadBootResource` function returns `null`, Blazor uses the default loading behavior for the resource. For example, the preceding code returns `null` for the `dotnetjs` boot resource (`dotnet.*.js`) because the `dotnetjs` boot resource must either return `null` for default loading behavior or a URI for the source of the `dotnetjs` boot resource.

The `loadBootResource` function can also return a Response promise. For an example, see Host and deploy ASP.NET Core Blazor WebAssembly.

For more information, see ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching.

Control headers in C# code

Control headers at startup in C# code using the following approaches.

In the following examples, a Content Security Policy (CSP) is applied to the app via a CSP header. The `{POLICY STRING}` placeholder is the CSP policy string.

Server-side and prerendered client-side scenarios

Use ASP.NET Core Middleware to control the headers collection.

In the `Program` file:

C#

```
app.Use(async (context, next) =>
{
    context.Response.Headers.Append("Content-Security-Policy", "{POLICY
STRING}");
    await next();
});
```

The preceding example uses inline middleware, but you can also create a custom middleware class and call the middleware with an extension method in the `Program` file. For more information, see Write custom ASP.NET Core middleware.

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

Client-side loading progress indicators

A loading progress indicator shows the loading progress of the app to users, indicating that the app is loading normally and that the user should wait until loading is finished.

Blazor Web App loading progress

The loading progress indicator used in Blazor WebAssembly apps isn't present in an app created from the Blazor Web App project template. Usually, a loading progress indicator isn't desirable for interactive WebAssembly components because Blazor Web Apps prerender client-side components on the server for fast initial load times. For mixed-render-mode situations, the framework or developer code must also be careful to avoid the following problems:

- Showing multiple loading indicators on the same rendered page.
- Inadvertently discarding prerendered content while the .NET WebAssembly runtime is loading.

A future release of .NET might provide a framework-based loading progress indicator. In the meantime, you can add a custom loading progress indicator to a Blazor Web App.

Create a `LoadingProgress` component in the `.Client` app that calls OperatingSystem.IsBrowser:

- When `false`, display a loading progress indicator while the Blazor bundle is downloaded and before the Blazor runtime activates on the client.
- When `true`, render the requested component's content.

The following demonstration uses the loading progress indicator found in apps created from the Blazor WebAssembly template, including a modification of the styles that the template provides. The styles are loaded into the app's `<head>` content by the HeadContent component. For more information, see Control head content in ASP.NET Core Blazor apps.

`LoadingProgress.razor`:

razor

```
@if (!OperatingSystem.IsBrowser())
{
    <HeadContent>
        <style>
            .loading-progress {
                position: relative;
                display: block;
                width: 8rem;
                height: 8rem;
                margin: 20vh auto 1rem auto;
            }

                .loading-progress circle {
                    fill: none;
                    stroke: #e0e0e0;
                    stroke-width: 0.6rem;
                    transform-origin: 50% 50%;
                    transform: rotate(-90deg);
                }

                    .loading-progress circle:last-child {
                        stroke: #1b6ec2;
                        stroke-dasharray:
                            calc(3.142 * var(--blazor-load-percentage, 0%) *
0.8),
```

```
                            500%;
                        transition: stroke-dasharray 0.05s ease-in-out;
                    }

            .loading-progress-text {
                position: relative;
                text-align: center;
                font-weight: bold;
                top: -90px;
            }

                .loading-progress-text:after {
                    content: var(--blazor-load-percentage-text, "Loading");
                }

            code {
                color: #c02d76;
            }
        </style>
    </HeadContent>
    <svg class="loading-progress">
        <circle r="40%" cx="50%" cy="50%" />
        <circle r="40%" cx="50%" cy="50%" />
    </svg>
    <div class="loading-progress-text"></div>
}
else
{
    @ChildContent
}

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

In a component that adopts Interactive WebAssembly rendering, wrap the component's Razor markup with the `LoadingProgress` component. The following example demonstrates the approach with the `Counter` component of an app created from the Blazor Web App project template.

`Pages/Counter.razor`:

razor

```
@page "/counter"
@rendermode InteractiveWebAssembly

<PageTitle>Counter</PageTitle>

<LoadingProgress>
    <h1>Counter</h1>
```

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

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

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Blazor WebAssembly app loading progress

The project template contains Scalable Vector Graphics (SVG) and text indicators that show the loading progress of the app.

The progress indicators are implemented with HTML and CSS using two CSS custom properties (variables) provided by Blazor:

- `--blazor-load-percentage`: The percentage of app files loaded.
- `--blazor-load-percentage-text`: The percentage of app files loaded, rounded to the nearest whole number.

Using the preceding CSS variables, you can create custom progress indicators that match the styling of your app.

In the following example:

- `resourcesLoaded` is an instantaneous count of the resources loaded during app startup.
- `totalResources` is the total number of resources to load.

JavaScript

```
const percentage = resourcesLoaded / totalResources * 100;
document.documentElement.style.setProperty(
  '--blazor-load-percentage', `${percentage}%`);
document.documentElement.style.setProperty(
  '--blazor-load-percentage-text', `"${Math.floor(percentage)}%"`);
```

The default round progress indicator is implemented in HTML in the `wwwroot/index.html` file:

HTML

```
<div id="app">
    <svg class="loading-progress">
        <circle r="40%" cx="50%" cy="50%" />
        <circle r="40%" cx="50%" cy="50%" />
    </svg>
    <div class="loading-progress-text"></div>
</div>
```

To review the project template markup and styling for the default progress indicators, see the ASP.NET Core reference source:

- wwwroot/index.html
- app.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).

Instead of using the default round progress indicator, the following example shows how to implement a linear progress indicator.

Add the following styles to `wwwroot/css/app.css`:

css

```
.linear-progress {
    background: silver;
    width: 50vw;
    margin: 20% auto;
    height: 1rem;
    border-radius: 10rem;
    overflow: hidden;
    position: relative;
}

.linear-progress:after {
    content: '';
    position: absolute;
    inset: 0;
    background: blue;
    scale: var(--blazor-load-percentage, 0%) 100%;
```

```
    transform-origin: left top;
    transition: scale ease-out 0.5s;
}
```

A CSS variable (`var(...)`) is used to pass the value of `--blazor-load-percentage` to the `scale` property of a blue pseudo-element that indicates the loading progress of the app's files. As the app loads, `--blazor-load-percentage` is updated automatically, which dynamically changes the progress indicator's visual representation.

In `wwwroot/index.html`, remove the default SVG round indicator in `<div id="app">...</div>` and replace it with the following markup:

HTML

```
<div class="linear-progress"></div>
```

Configure the .NET WebAssembly runtime

In advanced programming scenarios, the `configureRuntime` function with the `dotnet` runtime host builder is used to configure the .NET WebAssembly runtime. For example, `dotnet.withEnvironmentVariable` sets an environment variable that:

- Configures the .NET WebAssembly runtime.
- Changes the behavior of a C library.

> **Note**
>
> A documentation request is pending in the `dotnet/runtime` GitHub repository for more information on environment variables that configure the .NET WebAssembly runtime or affect the behavior of C libraries. Although the documentation request is pending, more information and cross-links to additional resources are available in the request, Question/request for documentation on .NET WASM runtime env vars (dotnet/runtime #98225).

The `configureRuntime` function can also be used to enable integration with a browser profiler.

For the placeholders in the following examples that set an environment variable:

- The `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

- The `{NAME}` placeholder is the environment variable's name.
- The `{VALUE}` placeholder is the environment variable's value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      configureRuntime: dotnet => {
        dotnet.withEnvironmentVariable("{NAME}", "{VALUE}");
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureRuntime: dotnet => {
      dotnet.withEnvironmentVariable("{NAME}", "{VALUE}");
    }
  });
</script>
```

> **Note**
>
> The .NET runtime instance can be accessed using the .NET WebAssembly Runtime API (`Blazor.runtime`). For example, the app's build configuration can be obtained using `Blazor.runtime.runtimeBuildInfo.buildConfiguration`.
>
> For more information on the .NET WebAssembly runtime configuration, see the runtime's TypeScript definition file (dotnet.d.ts) in the dotnet/runtime GitHub repository.
>
> > **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).

Disable enhanced navigation and form handling

This section applies to Blazor Web Apps.

To disable enhanced navigation and form handling, set `disableDomPreservation` to `true` for `Blazor.start`:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    ssr: { disableDomPreservation: true }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Additional resources

- Environments: Set the app's environment
- SignalR (includes sections on SignalR startup configuration)
- Globalization and localization: Statically set the culture with Blazor.start() (*client-side only*)
- Host and deploy: Blazor WebAssembly: Compression

ASP.NET Core Blazor environments

Article • 09/16/2024

> **ⓘ 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 configure and read the environment in a Blazor app.

When running an app locally, the environment defaults to `Development`. When the app is published, the environment defaults to `Production`.

We recommend the following conventions:

- Always use the "`Development`" environment name for local development. This is because the ASP.NET Core framework expects exactly that name when configuring the app and tooling for local development runs of an app.
- For testing, staging, and production environments, always publish and deploy the app. You can use any environment naming scheme that you wish for published apps, but always use app setting file names with casing of the environment segment that exactly matches the environment name. For staging, use "`Staging`" (capital "S") as the environment name, and name the app settings file to match (`appsettings.Staging.json`). For production, use "`Production`" (capital "P") as the environment name, and name the app settings file to match (`appsettings.Production.json`).

Set the environment

The environment is set using any of the following approaches:

- Blazor Web App: Use any of the approaches described in Use multiple environments in ASP.NET Core for general ASP.NET Core apps.
- Blazor Web App or standalone Blazor WebAssembly: Blazor start configuration
- Standalone Blazor WebAssembly: blazor-environment header
- Blazor Web App or standalone Blazor WebAssembly: Azure App Service

On the client for a Blazor Web App, the environment is determined from the server via a middleware that communicates the environment to the browser via a header named `blazor-environment`. The header sets the environment when the WebAssemblyHost is created in the client-side `Program` file (WebAssemblyHostBuilder.CreateDefault).

For a standalone Blazor WebAssembly app running locally, the development server adds the `blazor-environment` header.

For app's running locally in development, the app defaults to the `Development` environment. Publishing the app defaults the environment to `Production`.

Set the client-side environment via Blazor startup configuration

The following example starts Blazor in the `Staging` environment if the hostname includes `localhost`. Otherwise, the environment is set to its default value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  if (window.location.hostname.includes("localhost")) {
    Blazor.start({
      webAssembly: {
        environment: "Staging"
      }
    });
  } else {
    Blazor.start();
  }
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> **Note**
>
> For Blazor Web Apps that set the `webAssembly` > `environment` property in `Blazor.start` configuration, it's wise to match the server-side environment to the environment set on the `environment` property. Otherwise, prerendering on the server will operate under a different environment than rendering on the client,

which results in arbitrary effects. For general guidance on setting the environment for a Blazor Web App, see Use multiple environments in ASP.NET Core.

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  if (window.location.hostname.includes("localhost")) {
    Blazor.start({
      environment: "Staging"
    });
  } else {
    Blazor.start();
  }
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Using the `environment` property overrides the environment set by the blazor-environment header.

The preceding approach sets the client's environment without changing the `blazor-environment` header's value, nor does it change the server project's console logging of the startup environment for a Blazor Web App that has adopted global Interactive WebAssembly rendering.

To log the environment to the console in either a standalone Blazor WebAssembly project or the `.Client` project of a Blazor Web App, place the following C# code in the `Program` file after the WebAssemblyHost is created with WebAssemblyHostBuilder.CreateDefault and before the line that builds and runs the project (`await builder.Build().RunAsync();`):

C#

```
Console.WriteLine(
    $"Client Hosting Environment: {builder.HostEnvironment.Environment}");
```

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

Set the client-side environment via header

Blazor WebAssembly apps can set the environment with the `blazor-environment` header.

In the following example for IIS, the custom header (`blazor-environment`) is added to the published `web.config` file. The `web.config` file is located in the `bin/Release/{TARGET FRAMEWORK}/publish` folder, where the `{TARGET FRAMEWORK}` placeholder is the target framework:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>

    ...

    <httpProtocol>
      <customHeaders>
        <add name="blazor-environment" value="Staging" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

> **Note**
>
> To use a custom `web.config` file for IIS that isn't overwritten when the app is published to the `publish` folder, see Host and deploy ASP.NET Core Blazor WebAssembly.
>
> Although the Blazor framework issues the header name in all lowercase letters (`blazor-environment`), you're welcome to use any casing that you desire. For example, a header name that capitalizes each word (`Blazor-Environment`) is supported.

Set the environment for Azure App Service

For a standalone Blazor WebAssembly app, you can set the environment manually via start configuration or the blazor-environment header.

For a server-side app, set the environment via an `ASPNETCORE_ENVIRONMENT` app setting in Azure:

1. ***Confirm that the casing of environment segments in app settings file names match their environment name casing exactly***. For example, the matching app

settings file name for the `Staging` environment is `appsettings.Staging.json`. If the file name is `appsettings.staging.json` (lowercase "`s`"), the file isn't located, and the settings in the file aren't used in the `Staging` environment.

2. For Visual Studio deployment, confirm that the app is deployed to the correct deployment slot. For an app named `BlazorAzureAppSample`, the app is deployed to the `Staging` deployment slot.

3. In the Azure portal for the environment's deployment slot, set the environment with the `ASPNETCORE_ENVIRONMENT` app setting. For an app named `BlazorAzureAppSample`, the staging App Service Slot is named `BlazorAzureAppSample/Staging`. For the `Staging` slot's configuration, create an app setting for `ASPNETCORE_ENVIRONMENT` with a value of `Staging`. **Deployment slot setting** is enabled for the setting.

When requested in a browser, the `BlazorAzureAppSample/Staging` app loads in the `Staging` environment at `https://blazorazureappsample-staging.azurewebsites.net`.

When the app is loaded in the browser, the response header collection for `blazor.boot.json` indicates that the `blazor-environment` header value is `Staging`.

App settings from the `appsettings.{ENVIRONMENT}.json` file are loaded by the app, where the `{ENVIRONMENT}` placeholder is the app's environment. In the preceding example, settings from the `appsettings.Staging.json` file are loaded.

Read the environment in a Blazor WebAssembly app

Obtain the app's environment in a component by injecting IWebAssemblyHostEnvironment and reading the Environment property.

`ReadEnvironment.razor`:

razor

```razor
@page "/read-environment"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IWebAssemblyHostEnvironment Env

<h1>Environment example</h1>

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

Read the environment client-side in a Blazor Web App

Assuming that prerendering isn't disabled for a component or the app, a component in the `.Client` project is prerendered on the server. Because the server doesn't have a registered IWebAssemblyHostEnvironment service, it isn't possible to inject the service and use the service implementation's host environment extension methods and properties during server prerendering. Injecting the service into an Interactive WebAssembly or Interactive Auto component results in the following runtime error:

> There is no registered service of type 'Microsoft.AspNetCore.Components.WebAssembly.Hosting.IWebAssemblyHostEnvironment'.

To address this, create a custom service implementation for IWebAssemblyHostEnvironment on the server. Add the following class to the server project.

`ServerHostEnvironment.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Microsoft.AspNetCore.Components;

public class ServerHostEnvironment(IWebHostEnvironment env,
NavigationManager nav) :
    IWebAssemblyHostEnvironment
{
    public string Environment => env.EnvironmentName;
    public string BaseAddress => nav.BaseUri;
}
```

In the server project's `Program` file, register the service:

C#

```
builder.Services.TryAddScoped<IWebAssemblyHostEnvironment,
ServerHostEnvironment>();
```

At this point, the IWebAssemblyHostEnvironment service can be injected into an interactive WebAssembly or interactive Auto component and used as shown in the Read the environment in a Blazor WebAssembly app section.

The preceding example can demonstrate that it's possible to have a different server environment than client environment, which isn't recommended and may lead to arbitrary results. When setting the environment in a Blazor Web App, it's best to match server and `.Client` project environments. Consider the following scenario in a test app:

- Implement the client-side (`webassembly`) environment property with the `Staging` environment via `Blazor.start`. See the Set the client-side environment via startup configuration section for an example.
- Don't change the server-side `Properties/launchSettings.json` file. Leave the `environmentVariables` section with the `ASPNETCORE_ENVIRONMENT` environment variable set to `Development`.

You can see the value of the IWebAssemblyHostEnvironment.Environment property change in the UI.

When prerendering occurs on the server, the component is rendered in the `Development` environment:

> **Environment:** Development

When the component is rerendered just a second or two later, after the Blazor bundle is downloaded and the .NET WebAssembly runtime activates, the values change to reflect that the client is operating in the `Staging` environment on the client:

> **Environment:** Staging

The preceding example demonstrates why we recommend setting the server environment to match the client environment for development, testing, and production deployments.

For more information, see the Client-side services fail to resolve during prerendering section of the *Render modes* article, which appears later in the Blazor documentation.

Read the client-side environment during startup

During startup, the WebAssemblyHostBuilder exposes the IWebAssemblyHostEnvironment through the HostEnvironment property, which enables environment-specific logic in host builder code.

In the `Program` file:

C#

```
if (builder.HostEnvironment.Environment == "Custom")
{
    ...
};
```

The following convenience extension methods provided through WebAssemblyHostEnvironmentExtensions permit checking the current environment for `Development`, `Production`, `Staging`, and custom environment names:

- IsDevelopment
- IsProduction
- IsStaging
- IsEnvironment

In the `Program` file:

C#

```
if (builder.HostEnvironment.IsStaging())
{
    ...
};

if (builder.HostEnvironment.IsEnvironment("Custom"))
{
    ...
};
```

The IWebAssemblyHostEnvironment.BaseAddress property can be used during startup when the NavigationManager service isn't available.

Additional resources

- ASP.NET Core Blazor startup
- Use multiple environments in ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor logging

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 app logging, including configuration and how to write log messages from Razor components.

Configuration

Logging configuration can be loaded from app settings files. For more information, see ASP.NET Core Blazor configuration.

At default log levels and without configuring additional logging providers:

- On the server, logging only occurs to the server-side .NET console in the `Development` environment at the LogLevel.Information level or higher.
- On the client, logging only occurs to the client-side browser developer tools console at the LogLevel.Information level or higher.

When the app is configured in the project file to use implicit namespaces (`<ImplicitUsings>enable</ImplicitUsings>`), a `using` directive for Microsoft.Extensions.Logging or any API in the LoggerExtensions class isn't required to support API Visual Studio IntelliSense completions or building apps. If implicit namespaces aren't enabled, Razor components must explicitly define @using directives for logging namespaces that aren't imported via the `_Imports.razor` file.

Log levels

Log levels conform to ASP.NET Core app log levels, which are listed in the API documentation at LogLevel.

Razor component logging

The following example:

- Injects an ILogger (`ILogger<Counter1>`) object to create a logger. The log's *category* is the fully qualified name of the component's type, `Counter`.
- Calls LogWarning to log at the Warning level.

`Counter1.razor`:

razor

```
@page "/counter-1"
@inject ILogger<Counter1> Logger

<PageTitle>Counter 1</PageTitle>

<h1>Counter 1</h1>

<p>Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        Logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

The following example demonstrates logging with an ILoggerFactory in components.

`Counter2.razor`:

razor

```
@page "/counter-2"
@inject ILoggerFactory LoggerFactory

<PageTitle>Counter 2</PageTitle>

<h1>Counter 2</h1>

<p>Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;
```

```
    private void IncrementCount()
    {
        var logger = LoggerFactory.CreateLogger<Counter2>();
        logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

Server-side logging

For general ASP.NET Core logging guidance, see Logging in .NET Core and ASP.NET Core.

Client-side logging

Not every feature of ASP.NET Core logging is supported client-side. For example, client-side components don't have access to the client's file system or network, so writing logs to the client's physical or network storage isn't possible. When using a third-party logging service designed to work with single-page apps (SPAs), follow the service's security guidance. Keep in mind that every piece of data, including keys or secrets stored client-side are ***insecure*** and can be easily discovered by malicious users.

Configure logging in client-side apps with the WebAssemblyHostBuilder.Logging property. The Logging property is of type ILoggingBuilder, so the extension methods of ILoggingBuilder are supported.

To set the minimum logging level, call LoggingBuilderExtensions.SetMinimumLevel on the host builder in the `Program` file with the LogLevel. The following example sets the minimum log level to Warning:

C#

```
builder.Logging.SetMinimumLevel(LogLevel.Warning);
```

Log in the client-side `Program` file

Logging is supported in client-side apps after the WebAssemblyHostBuilder is built using the framework's internal console logger provider (WebAssemblyConsoleLoggerProvider (reference source)).

In the `Program` file:

```
C#

var host = builder.Build();

var logger = host.Services.GetRequiredService<ILoggerFactory>()
    .CreateLogger<Program>();

logger.LogInformation("Logged after the app is built in the Program file.");

await host.RunAsync();
```

Developer tools console output:

```
info: Program[0]
Logged after the app is built in the Program file.
```

> **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).

Client-side log category

Log categories are supported.

The following example shows how to use log categories with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`) that injects an ILoggerFactory as `LoggerFactory`:

```
C#

var logger = LoggerFactory.CreateLogger("CustomCategory");
logger.LogWarning("Someone has clicked me!");
```

Developer tools console output:

> warn: CustomCategory[0]
> Someone has clicked me!

Client-side log event ID

Log event ID is supported.

The following example shows how to use log event IDs with the `Counter` component of an app created from a Blazor project template.

`LogEvent.cs`:

C#

```
public class LogEvent
{
    public const int Event1 = 1000;
    public const int Event2 = 1001;
}
```

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
logger.LogInformation(LogEvent.Event1, "Someone has clicked me!");
logger.LogWarning(LogEvent.Event2, "Someone has clicked me!");
```

Developer tools console output:

> info: BlazorSample.Pages.Counter[1000]
> Someone has clicked me!
> warn: BlazorSample.Pages.Counter[1001]
> Someone has clicked me!

Client-side log message template

Log message templates are supported:

The following example shows how to use log message templates with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
logger.LogInformation("Someone clicked me at {CurrentDT}!",
DateTime.UtcNow);
```

Developer tools console output:

```
info: BlazorSample.Pages.Counter[0]
Someone clicked me at 04/21/2022 12:15:57!
```

Client-side log exception parameters

Log exception parameters are supported.

The following example shows how to use log exception parameters with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
currentCount++;

try
{
    if (currentCount == 3)
    {
        currentCount = 4;
        throw new OperationCanceledException("Skip 3");
    }
}
catch (Exception ex)
{
    logger.LogWarning(ex, "Exception (currentCount: {Count})!",
currentCount);
}
```

Developer tools console output:

```
warn: BlazorSample.Pages.Counter[0]
Exception (currentCount: 4)!
System.OperationCanceledException: Skip 3
at BlazorSample.Pages.Counter.IncrementCount() in
C:UsersAlabaDesktopBlazorSamplePagesCounter.razor:line 28
```

Client-side filter function

Filter functions are supported.

The following example shows how to use a filter with the `Counter` component of an app created from a Blazor project template.

In the `Program` file:

C#

```
builder.Logging.AddFilter((provider, category, logLevel) =>
    category.Equals("CustomCategory2") && logLevel == LogLevel.Information);
```

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`) that injects an ILoggerFactory as `LoggerFactory`:

C#

```
var logger1 = LoggerFactory.CreateLogger("CustomCategory1");
logger1.LogInformation("Someone has clicked me!");

var logger2 = LoggerFactory.CreateLogger("CustomCategory1");
logger2.LogWarning("Someone has clicked me!");

var logger3 = LoggerFactory.CreateLogger("CustomCategory2");
logger3.LogInformation("Someone has clicked me!");

var logger4 = LoggerFactory.CreateLogger("CustomCategory2");
logger4.LogWarning("Someone has clicked me!");
```

In the developer tools console output, the filter only permits logging for the `CustomCategory2` category and Information log level message:

> info: CustomCategory2[0]
> Someone has clicked me!

The app can also configure log filtering for specific namespaces. For example, set the log level to Trace in the `Program` file:

C#

```
builder.Logging.SetMinimumLevel(LogLevel.Trace);
```

Normally at the Trace log level, developer tools console output at the **Verbose** level includes Microsoft.AspNetCore.Components.RenderTree logging messages, such as the following:

> dbug: Microsoft.AspNetCore.Components.RenderTree.Renderer[3]
> Rendering component 14 of type
> Microsoft.AspNetCore.Components.Web.HeadOutlet

In the `Program` file, logging messages specific to Microsoft.AspNetCore.Components.RenderTree can be disabled using ***either*** of the following approaches:

- C#

```
builder.Logging.AddFilter("Microsoft.AspNetCore.Components.RenderTree.*
", LogLevel.None);
```

- C#

```
builder.Services.PostConfigure<LoggerFilterOptions>(options =>
    options.Rules.Add(
        new LoggerFilterRule(null,

"Microsoft.AspNetCore.Components.RenderTree.*",
                             LogLevel.None,
                             null)
    ));
```

After ***either*** of the preceding filters is added to the app, the console output at the **Verbose** level doesn't show logging messages from the Microsoft.AspNetCore.Components.RenderTree API.

Client-side custom logger provider

The example in this section demonstrates a custom logger provider for further customization.

Add a package reference to the app for the Microsoft.Extensions.Logging.Configuration package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add the following custom logger configuration. The configuration establishes a `LogLevels` dictionary that sets a custom log format for three log levels: Information, Warning, and Error. A `LogFormat` enum is used to describe short (`LogFormat.Short`) and long (`LogFormat.Long`) formats.

`CustomLoggerConfiguration.cs`:

C#

```
using Microsoft.Extensions.Logging;

public class CustomLoggerConfiguration
{
    public int EventId { get; set; }

    public Dictionary<LogLevel, LogFormat> LogLevels { get; set; } =
        new()
        {
            [LogLevel.Information] = LogFormat.Short,
            [LogLevel.Warning] = LogFormat.Short,
            [LogLevel.Error] = LogFormat.Long
        };

    public enum LogFormat
    {
        Short,
        Long
    }
}
```

Add the following custom logger to the app. The `CustomLogger` outputs custom log formats based on the `logLevel` values defined in the preceding `CustomLoggerConfiguration` configuration.

C#

```
using Microsoft.Extensions.Logging;
using static CustomLoggerConfiguration;

public sealed class CustomLogger : ILogger
{
    private readonly string name;
    private readonly Func<CustomLoggerConfiguration> getCurrentConfig;

    public CustomLogger(
        string name,
        Func<CustomLoggerConfiguration> getCurrentConfig) =>
        (this.name, this.getCurrentConfig) = (name, getCurrentConfig);

    public IDisposable BeginScope<TState>(TState state) => default!;
```

```
    public bool IsEnabled(LogLevel logLevel) =>
        getCurrentConfig().LogLevels.ContainsKey(logLevel);

    public void Log<TState>(
        LogLevel logLevel,
        EventId eventId,
        TState state,
        Exception? exception,
        Func<TState, Exception?, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        CustomLoggerConfiguration config = getCurrentConfig();

        if (config.EventId == 0 || config.EventId == eventId.Id)
        {
            switch (config.LogLevels[logLevel])
            {
                case LogFormat.Short:
                    Console.WriteLine($"{name}: {formatter(state,
exception)}");
                    break;
                case LogFormat.Long:
                    Console.WriteLine($"[{eventId.Id, 2}: {logLevel, -12}]
{name} - {formatter(state, exception)}");
                    break;
                default:
                    // No-op
                    break;
            }
        }
    }
}
```

Add the following custom logger provider to the app. `CustomLoggerProvider` adopts an Options-based approach to configure the logger via built-in logging configuration features. For example, the app can set or change log formats via an `appsettings.json` file without requiring code changes to the custom logger, which is demonstrated at the end of this section.

`CustomLoggerProvider.cs`:

C#

```
using System.Collections.Concurrent;
using Microsoft.Extensions.Options;
```

```
[ProviderAlias("CustomLog")]
public sealed class CustomLoggerProvider : ILoggerProvider
{
    private readonly IDisposable onChangeToken;
    private CustomLoggerConfiguration config;
    private readonly ConcurrentDictionary<string, CustomLogger> loggers =
        new(StringComparer.OrdinalIgnoreCase);

    public CustomLoggerProvider(
        IOptionsMonitor<CustomLoggerConfiguration> config)
    {
        this.config = config.CurrentValue;
        onChangeToken = config.OnChange(updatedConfig => this.config =
updatedConfig);
    }

    public ILogger CreateLogger(string categoryName) =>
        loggers.GetOrAdd(categoryName, name => new CustomLogger(name,
GetCurrentConfig));

    private CustomLoggerConfiguration GetCurrentConfig() => config;

    public void Dispose()
    {
        loggers.Clear();
        onChangeToken.Dispose();
    }
}
```

Add the following custom logger extensions.

`CustomLoggerExtensions.cs`:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Configuration;

public static class CustomLoggerExtensions
{
    public static ILoggingBuilder AddCustomLogger(
        this ILoggingBuilder builder)
    {
        builder.AddConfiguration();

        builder.Services.TryAddEnumerable(
            ServiceDescriptor.Singleton<ILoggerProvider,
CustomLoggerProvider>());

        LoggerProviderOptions.RegisterProviderOptions
            <CustomLoggerConfiguration, CustomLoggerProvider>
```

```
(builder.Services);

        return builder;
    }
}
```

In the `Program` file on the host builder, clear the existing provider by calling ClearProviders and add the custom logging provider:

C#

```
builder.Logging.ClearProviders().AddCustomLogger();
```

In the following `CustomLoggerExample` component:

- The debug message isn't logged.
- The information message is logged in the short format (`LogFormat.Short`).
- The warning message is logged in the short format (`LogFormat.Short`).
- The error message is logged in the long format (`LogFormat.Long`).
- The trace message isn't logged.

`CustomLoggerExample.razor`:

razor

```
@page "/custom-logger-example"
@inject ILogger<CustomLoggerExample> Logger

<p>
    <button @onclick="LogMessages">Log Messages</button>
</p>

@code{
    private void LogMessages()
    {
        Logger.LogDebug(1, "This is a debug message.");
        Logger.LogInformation(3, "This is an information message.");
        Logger.LogWarning(5, "This is a warning message.");
        Logger.LogError(7, "This is an error message.");
        Logger.LogTrace(5!, "This is a trace message.");
    }
}
```

The following output is seen in the browser's developer tools console when the **`Log Messages`** button is selected. The log entries reflect the appropriate formats applied by the custom logger (the client app is named `LoggingTest`):

> LoggingTest.Pages.CustomLoggerExample: This is an information message.
> LoggingTest.Pages.CustomLoggerExample: This is a warning message.
> [7: Error] LoggingTest.Pages.CustomLoggerExample - This is an error message.

From a casual inspection of the preceding example, it's apparent that setting the log line formats via the dictionary in `CustomLoggerConfiguration` isn't strictly necessary. The line formats applied by the custom logger (`CustomLogger`) could have been applied by merely checking the `logLevel` in the `Log` method. The purpose of assigning the log format via configuration is that the developer can change the log format easily via app configuration, as the following example demonstrates.

In the client-side app, add or update the `appsettings.json` file to include logging configuration. Set the log format to `Long` for all three log levels:

JSON

```
{
  "Logging": {
    "CustomLog": {
      "LogLevels": {
        "Information": "Long",
        "Warning": "Long",
        "Error": "Long"
      }
    }
  }
}
```

In the preceding example, notice that the entry for the custom logger configuration is `CustomLog`, which was applied to the custom logger provider (`CustomLoggerProvider`) as an alias with `[ProviderAlias("CustomLog")]`. The logging configuration could have been applied with the name `CustomLoggerProvider` instead of `CustomLog`, but use of the alias `CustomLog` is more user friendly.

In the `Program` file, consume the logging configuration. Add the following code:

C#

```
builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("Logging"));
```

The call to LoggingBuilderConfigurationExtensions.AddConfiguration can be placed either before or after adding the custom logger provider.

Run the app again. Select the **`Log Messages`** button. Notice that the logging configuration is applied from the `appsettings.json` file. All three log entries are in the long (`LogFormat.Long`) format (the client app is named `LoggingTest`):

> [3: Information] LoggingTest.Pages.CustomLoggerExample - This is an information message.
> [5: Warning] LoggingTest.Pages.CustomLoggerExample - This is a warning message.
> [7: Error] LoggingTest.Pages.CustomLoggerExample - This is an error message.

Client-side log scopes

The developer tools console logger doesn't support log scopes. However, a custom logger can support log scopes. For an unsupported example that you can further develop to suit your needs, see the `BlazorWebAssemblyScopesLogger` sample app in the Blazor samples GitHub repository (how to download).

The sample app uses standard ASP.NET Core BeginScope logging syntax to indicate scopes for logged messages. The `Logger` service in the following example is an `ILogger<CustomLoggerExample>`, which is injected into the app's `CustomLoggerExample` component (`CustomLoggerExample.razor`).

C#

```
using (Logger.BeginScope("L1"))
{
    Logger.LogInformation(3, "INFO: ONE scope.");
}

using (Logger.BeginScope("L1"))
{
    using (Logger.BeginScope("L2"))
    {
        Logger.LogInformation(3, "INFO: TWO scopes.");
    }
}

using (Logger.BeginScope("L1"))
{
    using (Logger.BeginScope("L2"))
    {
        using (Logger.BeginScope("L3"))
        {
            Logger.LogInformation(3, "INFO: THREE scopes.");
        }
```

```
    }
}
```

Output:

```
[ 3: Information ] {CLASS} - INFO: ONE scope. => L1 blazor.webassembly.js:1:35542
[ 3: Information ] {CLASS} - INFO: TWO scopes. => L1 => L2
blazor.webassembly.js:1:35542
[ 3: Information ] {CLASS} - INFO: THREE scopes. => L1 => L2 => L3
```

The `{CLASS}` placeholder in the preceding example is `BlazorWebAssemblyScopesLogger.Pages.CustomLoggerExample`.

Prerendered component logging

Prerendered components execute component initialization code twice. Logging takes place server-side on the first execution of initialization code and client-side on the second execution of initialization code. Depending on the goal of logging during initialization, check logs server-side, client-side, or both.

SignalR client logging with the SignalR client builder

This section applies to server-side apps.

In Blazor script start configuration, pass in the `configureSignalR` configuration object that calls `configureLogging` with the log level.

For the `configureLogging` log level value, pass the argument as either the string or integer log level shown in the following table.

Expand table

LogLevel	String setting	Integer setting
Trace	`trace`	0
Debug	`debug`	1
Information	`information`	2
Warning	`warning`	3

LogLevel	String setting	Integer setting
Error	`error`	4
Critical	`critical`	5
None	`none`	6

Example 1: Set the Information log level with a string value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.configureLogging("information");
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.configureLogging("information");
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Example 2: Set the Information log level with an integer value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
```

```
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.configureLogging(2); // LogLevel.Information
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.configureLogging(2); // LogLevel.Information
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> ⓘ Note
>
> Using an integer to specify the logging level in Example 2, often referred to as a *magic number* or *magic constant*, is considered a poor coding practice because the integer doesn't clearly identify the logging level when viewing the source code. If minimizing the bytes transferred to the browser is a priority, using an integer might be justified (consider removing the comment in such cases).

For more information on Blazor startup (`Blazor.start()`), see ASP.NET Core Blazor startup.

SignalR client logging with app configuration

Set up app settings configuration as described in ASP.NET Core Blazor configuration. Place app settings files in `wwwroot` that contain a `Logging:LogLevel:HubConnection` app setting.

> ⓘ Note

As an alternative to using app settings, you can pass the LogLevel as the argument to LoggingBuilderExtensions.SetMinimumLevel when the hub connection is created in a Razor component. However, accidentally deploying the app to a production hosting environment with verbose logging may result in a performance penalty. We recommend using app settings to set the log level.

Provide a `Logging:LogLevel:HubConnection` app setting in the default `appsettings.json` file and in the `Development` environment app settings file. Use a typical less-verbose log level for the default, such as LogLevel.Warning. The default app settings value is what is used in `Staging` and `Production` environments if no app settings files for those environments are present. Use a verbose log level in the `Development` environment app settings file, such as LogLevel.Trace.

`wwwroot/appsettings.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "HubConnection": "Warning"
    }
  }
}
```

`wwwroot/appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "HubConnection": "Trace"
    }
  }
}
```

ⓘ Important

Configuration in the preceding app settings files is only used by the app if the guidance in ASP.NET Core Blazor configuration is followed.

At the top of the Razor component file (`.razor`):

- Inject an ILoggerProvider to add a `WebAssemblyConsoleLogger` to the logging providers passed to HubConnectionBuilder. Unlike a ConsoleLoggerProvider, `WebAssemblyConsoleLogger` is a wrapper around browser-specific logging APIs (for example, `console.log`). Use of `WebAssemblyConsoleLogger` makes logging possible within Mono inside a browser context.
- Inject an `IConfiguration` to read the `Logging:LogLevel:HubConnection` app setting.

> ⓘ Note
>
> `WebAssemblyConsoleLogger` is internal and not supported for direct use in developer code.

C#

```
@inject ILoggerProvider LoggerProvider
@inject IConfiguration Config
```

> ⓘ Note
>
> The following example is based on the demonstration in the SignalR with Blazor tutorial. Consult the tutorial for further details.

In the component's OnInitializedAsync method, use HubConnectionBuilderExtensions.ConfigureLogging to add the logging provider and set the minimum log level from configuration:

C#

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .ConfigureLogging(builder =>
        {
            builder.AddProvider(LoggerProvider);
            builder.SetMinimumLevel(
                Config.GetValue<LogLevel>
("Logging:LogLevel:HubConnection"));
```

```
        })
        .Build();

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

    await hubConnection.StartAsync();
}
```

> **Note**
>
> In the preceding example, `Navigation` is an injected NavigationManager.

For more information on setting the app's environment, see ASP.NET Core Blazor environments.

Client-side authentication logging

Log Blazor authentication messages at the LogLevel.Debug or LogLevel.Trace logging levels with a logging configuration in app settings or by using a log filter for Microsoft.AspNetCore.Components.WebAssembly.Authentication in the `Program` file.

Use ***either*** of the following approaches:

- In an app settings file (for example, `wwwroot/appsettings.Development.json`):

  JSON

  ```
  "Logging": {
    "LogLevel": {
      "Microsoft.AspNetCore.Components.WebAssembly.Authentication":
  "Debug"
    }
  }
  ```

  For more information on how to configure a client-side app to read app settings files, see ASP.NET Core Blazor configuration.

- Using a log filter, the following example:
  - Activates logging for the `Debug` build configuration using a C# preprocessor directive.
  - Logs Blazor authentication messages at the Debug log level.

  C#

```
#if DEBUG
    builder.Logging.AddFilter(
        "Microsoft.AspNetCore.Components.WebAssembly.Authentication",
        LogLevel.Debug);
#endif
```

> **Note**
>
> Razor components rendered on the client only log to the client-side browser developer tools console.

Additional resources

- Logging in .NET Core and ASP.NET Core
- Loglevel Enum (API documentation)
- Implement a custom logging provider in .NET
- Browser developer tools documentation:
  - Chrome DevTools
  - Microsoft Edge Developer Tools overview
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Handle errors 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.

This article describes how Blazor manages unhandled exceptions and how to develop apps that detect and handle errors.

Detailed errors during development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a light yellow bar at the bottom of the screen:

- During development, the bar directs you to the browser console, where you can see the exception.
- In production, the bar notifies the user that an error has occurred and recommends refreshing the browser.

The UI for this error handling experience is part of the Blazor project templates. Not all versions of the Blazor project templates use the data-nosnippet attribute to signal to browsers not to cache the contents of the error UI, but all versions of the Blazor documentation apply the attribute.

In a Blazor Web App, customize the experience in the `MainLayout` component. Because the Environment Tag Helper (for example, `<environment include="Production">...</environment>`) isn't supported in Razor components, the following example injects IHostEnvironment to configure error messages for different environments.

At the top of `MainLayout.razor`:

```
razor
```

```
@inject IHostEnvironment HostEnvironment
```

Create or modify the Blazor error UI markup:

razor

```
<div id="blazor-error-ui" data-nosnippet>
    @if (HostEnvironment.IsProduction())
    {
        <span>An error has occurred.</span>
    }
    else
    {
        <span>An unhandled exception occurred.</span>
    }
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>
```

In a Blazor WebAssembly app, customize the experience in the `wwwroot/index.html` file:

HTML

```
<div id="blazor-error-ui" data-nosnippet>
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>
```

The `blazor-error-ui` element is normally hidden due to the presence of the `display: none` style of the `blazor-error-ui` CSS class in the app's auto-generated stylesheet. When an error occurs, the framework applies `display: block` to the element.

Detailed circuit errors

This section applies to Blazor Web Apps operating over a circuit.

Client-side errors don't include the call stack and don't provide detail on the cause of the error, but server logs do contain such information. For development purposes, sensitive circuit error information can be made available to the client by enabling detailed errors.

Set CircuitOptions.DetailedErrors to `true`. For more information and an example, see ASP.NET Core Blazor SignalR guidance.

An alternative to setting CircuitOptions.DetailedErrors is to set the `DetailedErrors` configuration key to `true` in the app's `Development` environment settings file (`appsettings.Development.json`). Additionally, set SignalR server-side logging (`Microsoft.AspNetCore.SignalR`) to Debug or Trace for detailed SignalR logging.

`appsettings.Development.json`:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.AspNetCore.SignalR": "Debug"
    }
  }
}
```

The DetailedErrors configuration key can also be set to `true` using the `ASPNETCORE_DETAILEDERRORS` environment variable with a value of `true` on `Development`/`Staging` environment servers or on your local system.

> ⚠ **Warning**
>
> Always avoid exposing error information to clients on the Internet, which is a security risk.

Detailed errors for Razor component server-side rendering

This section applies to Blazor Web Apps.

Use the RazorComponentsServiceOptions.DetailedErrors option to control producing detailed information on errors for Razor component server-side rendering. The default value is `false`.

The following example enables detailed errors:

C#

```
builder.Services.AddRazorComponents(options =>
    options.DetailedErrors = builder.Environment.IsDevelopment());
```

> ⚠ **Warning**
>
> **Only enable detailed errors in the `Development` environment.** Detailed errors may contain sensitive information about the app that malicious users can use in an attack.
>
> The preceding example provides a degree of safety by setting the value of DetailedErrors based on the value returned by IsDevelopment. When the app is in the `Development` environment, DetailedErrors is set to `true`. This approach isn't foolproof because it's possible to host a production app on a public server in the `Development` environment.

Manage unhandled exceptions in developer code

For an app to continue after an error, the app must have error handling logic. Later sections of this article describe potential sources of unhandled exceptions.

In production, don't render framework exception messages or stack traces in the UI. Rendering exception messages or stack traces could:

- Disclose sensitive information to end users.
- Help a malicious user discover weaknesses in an app that can compromise the security of the app, server, or network.

Unhandled exceptions for circuits

This section applies to server-side apps operating over a circuit.

Razor components with server interactivity enabled are stateful on the server. While users interact with the component on the server, they maintain a connection to the server known as a *circuit*. The circuit holds active component instances, plus many other aspects of state, such as:

- The most recent rendered output of components.
- The current set of event-handling delegates that could be triggered by client-side events.

If a user opens the app in multiple browser tabs, the user creates multiple independent circuits.

Blazor treats most unhandled exceptions as fatal to the circuit where they occur. If a circuit is terminated due to an unhandled exception, the user can only continue to interact with the app by reloading the page to create a new circuit. Circuits outside of the one that's terminated, which are circuits for other users or other browser tabs, aren't affected. This scenario is similar to a desktop app that crashes. The crashed app must be restarted, but other apps aren't affected.

The framework terminates a circuit when an unhandled exception occurs for the following reasons:

- An unhandled exception often leaves the circuit in an undefined state.
- The app's normal operation can't be guaranteed after an unhandled exception.
- Security vulnerabilities may appear in the app if the circuit continues in an undefined state.

Global exception handling

For approaches to handling exceptions globally, see the following sections:

- Error boundaries: Applies to all Blazor apps.
- Alternative global exception handling: Applies to Blazor Server, Blazor WebAssembly, and Blazor Web Apps (8.0 or later) that adopt a global interactive render mode.

Error boundaries

Error boundaries provide a convenient approach for handling exceptions. The ErrorBoundary component:

- Renders its child content when an error hasn't occurred.
- Renders error UI when an unhandled exception is thrown by any component within the error boundary.

To define an error boundary, use the ErrorBoundary component to wrap one or more other components. The error boundary manages unhandled exceptions thrown by the components that it wraps.

```
razor
```

```
<ErrorBoundary>
    ...
</ErrorBoundary>
```

To implement an error boundary in a global fashion, add the boundary around the body content of the app's main layout.

In `MainLayout.razor`:

razor

```
<article class="content px-4">
    <ErrorBoundary>
        @Body
    </ErrorBoundary>
</article>
```

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

An interactive render mode can't be applied to the `MainLayout` component because the component's `Body` parameter is a RenderFragment delegate, which is arbitrary code and can't be serialized. To enable interactivity broadly for the `MainLayout` component and the rest of the components further down the component hierarchy, the app must adopt a global interactive render mode by applying the interactive render mode to the `HeadOutlet` and `Routes` component instances in the app's root component, which is typically the `App` component. The following example adopts the Interactive Server (`InteractiveServer`) render mode globally.

In `Components/App.razor`:

razor

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

...

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

If you prefer not to enable global interactivity, place the error boundary farther down the component hierarchy. The important concepts to keep in mind are that wherever the error boundary is placed:

- If the component where the error boundary is placed isn't interactive, the error boundary is only capable of activating on the server during static SSR. For example, the boundary can activate when an error is thrown in a component lifecycle method but not for an event triggered by user interactivity within the component, such as an error thrown by a button click handler.
- If the component where the error boundary is placed is interactive, the error boundary is capable of activating for interactive components that it wraps.

> **Note**
>
> The preceding considerations aren't relevant for standalone Blazor WebAssembly apps because the client-side rendering (CSR) of a Blazor WebAssembly app is completely interactive.

Consider the following example, where an exception thrown by an embedded counter component is caught by an error boundary in the `Home` component, which adopts an interactive render mode.

`EmbeddedCounter.razor`:

razor

```
<h1>Embedded 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++;

        if (currentCount > 5)
        {
            throw new InvalidOperationException("Current count is too
big!");
        }
    }
}
```

`Home.razor`:

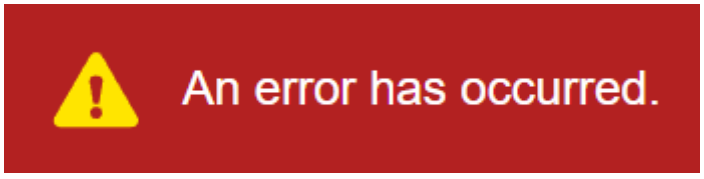
razor

```razor
@page "/"
@rendermode InteractiveServer

<PageTitle>Home</PageTitle>

<h1>Home</h1>

<ErrorBoundary>
    <EmbeddedCounter />
</ErrorBoundary>
```

If the unhandled exception is thrown for a `currentCount` over five:

- The error is logged normally (`System.InvalidOperationException: Current count is too big!`).
- The exception is handled by the error boundary.
- The default error UI is rendered by the error boundary.

The ErrorBoundary component renders an empty `<div>` element using the `blazor-error-boundary` CSS class for its error content. The colors, text, and icon for the default UI are defined in the app's stylesheet in the `wwwroot` folder, so you're free to customize the error UI.



To change the default error content:

- Wrap the components of the error boundary in the ChildContent property.
- Set the ErrorContent property to the error content.

The following example wraps the `EmbeddedCounter` component and supplies custom error content:

razor

```razor
<ErrorBoundary>
    <ChildContent>
        <EmbeddedCounter />
    </ChildContent>
    <ErrorContent>
        <p class="errorUI">😈 A rotten gremlin got us. Sorry!</p>
    </ErrorContent>
</ErrorBoundary>
```

For the preceding example, the app's stylesheet presumably includes an `errorUI` CSS class to style the content. The error content is rendered from the ErrorContent property without a block-level element. A block-level element, such as a division (`<div>`) or a paragraph (`<p>`) element, can wrap the error content markup, but it isn't required.

Optionally, use the context (`@context`) of the ErrorContent to obtain error data:

razor

```
<ErrorContent>
    @context.HelpLink
</ErrorContent>
```

The ErrorContent can also name the context. In the following example, the context is named `exception`:

razor

```
<ErrorContent Context="exception">
    @exception.HelpLink
</ErrorContent>
```

> ⚠ **Warning**
>
> Always avoid exposing error information to clients on the Internet, which is a security risk.

If the error boundary is defined in the app's layout, the error UI is seen regardless of which page the user navigates to after the error occurs. We recommend narrowly scoping error boundaries in most scenarios. If you broadly scope an error boundary, you can reset it to a non-error state on subsequent page navigation events by calling the error boundary's Recover method.

In `MainLayout.razor`:

- Add a field for the ErrorBoundary to capture a reference to it with the @ref attribute directive.
- In the OnParameterSet lifecycle method, you can trigger a recovery on the error boundary with Recover to clear the error when the user navigates to a different component.

razor

```
...

<ErrorBoundary @ref="errorBoundary">
    @Body
</ErrorBoundary>

...

@code {
    private ErrorBoundary? errorBoundary;

    protected override void OnParametersSet()
    {
        errorBoundary?.Recover();
    }
}
```

To avoid the infinite loop where recovering merely rerenders a component that throws the error again, don't call Recover from rendering logic. Only call Recover when:

- The user performs a UI gesture, such as selecting a button to indicate that they want to retry a procedure or when the user navigates to a new component.
- Additional logic that executes also clears the exception. When the component is rerendered, the error doesn't reoccur.

The following example permits the user to recover from the exception with a button:

razor

```
<ErrorBoundary @ref="errorBoundary">
    <ChildContent>
        <EmbeddedCounter />
    </ChildContent>
    <ErrorContent>
        <div class="alert alert-danger" role="alert">
            <p class="fs-3 fw-bold">😈 A rotten gremlin got us. Sorry!</p>
            <p>@context.HelpLink</p>
            <button class="btn btn-info" @onclick="_ =>
errorBoundary?.Recover()">
                Clear
            </button>
        </div>
    </ErrorContent>
</ErrorBoundary>

@code {
    private ErrorBoundary? errorBoundary;
}
```

You can also subclass ErrorBoundary for custom processing by overriding OnErrorAsync. The following example merely logs the error, but you can implement any error handling code you wish. You can remove the line that returns a CompletedTask if your code awaits an asynchronous task.

`CustomErrorBoundary.razor`:

razor

```razor
@inherits ErrorBoundary
@inject ILogger<CustomErrorBoundary> Logger

@if (CurrentException is null)
{
    @ChildContent
}
else if (ErrorContent is not null)
{
    @ErrorContent(CurrentException)
}

@code {
    protected override Task OnErrorAsync(Exception ex)
    {
        Logger.LogError(ex, "😈 A rotten gremlin got us. Sorry!");
        return Task.CompletedTask;
    }
}
```

The preceding example can also be implemented as a class.

`CustomErrorBoundary.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace BlazorSample;

public class CustomErrorBoundary : ErrorBoundary
{
    [Inject]
    ILogger<CustomErrorBoundary> Logger {  get; set; } = default!;

    protected override Task OnErrorAsync(Exception ex)
    {
        Logger.LogError(ex, "😈 A rotten gremlin got us. Sorry!");
        return Task.CompletedTask;
```

```
    }
}
```

Either of the preceding implementations used in a component:

razor

```razor
<CustomErrorBoundary>
    ...
</CustomErrorBoundary>
```

Alternative global exception handling

The approach described in this section applies to Blazor Server, Blazor WebAssembly, and Blazor Web Apps that adopt a global interactive render mode (`InteractiveServer`, `InteractiveWebAssembly`, or `InteractiveAuto`). The approach doesn't work with Blazor Web Apps that adopt per-page/component render modes or static server-side rendering (static SSR) because the approach relies on a CascadingValue/CascadingParameter, which don't work across render mode boundaries or with components that adopt static SSR.

An alternative to using Error boundaries (ErrorBoundary) is to pass a custom error component as a CascadingValue to child components. An advantage of using a component over using an injected service or a custom logger implementation is that a cascaded component can render content and apply CSS styles when an error occurs.

The following `ProcessError` component example merely logs errors, but methods of the component can process errors in any way required by the app, including through the use of multiple error processing methods.

`ProcessError.razor`:

razor

```razor
@inject ILogger<ProcessError> Logger

<CascadingValue Value="this">
    @ChildContent
</CascadingValue>

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

    public void LogError(Exception ex)
```

```
    {
        Logger.LogError("ProcessError.LogError: {Type} Message: {Message}",
            ex.GetType(), ex.Message);

        // Call StateHasChanged if LogError directly participates in
        // rendering. If LogError only logs or records the error,
        // there's no need to call StateHasChanged.
        //StateHasChanged();
    }
}
```

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

When using this approach in a Blazor Web App, open the `Routes` component and wrap the Router component (`<Router>...</Router>`) with the `ProcessError` component. This permits the `ProcessError` component to cascade down to any component of the app where the `ProcessError` component is received as a CascadingParameter.

In `Routes.razor`:

razor

```
<ProcessError>
    <Router ...>
        ...
    </Router>
</ProcessError>
```

When using this approach in a Blazor Server or Blazor WebAssembly app, open the `App` component, wrap the Router component (`<Router>...</Router>`) with the `ProcessError` component. This permits the `ProcessError` component to cascade down to any component of the app where the `ProcessError` component is received as a CascadingParameter.

In `App.razor`:

razor

```
<ProcessError>
    <Router ...>
        ...
    </Router>
</ProcessError>
```

To process errors in a component:

- Designate the `ProcessError` component as a CascadingParameter in the @code block. In an example `Counter` component in an app based on a Blazor project template, add the following `ProcessError` property:

  C#

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

- Call an error processing method in any `catch` block with an appropriate exception type. The example `ProcessError` component only offers a single `LogError` method, but the error processing component can provide any number of error processing methods to address alternative error processing requirements throughout the app. The following `Counter` component `@code` block example includes the `ProcessError` cascading parameter and traps an exception for logging when the count is greater than five:

  razor

  ```
  @code {
      private int currentCount = 0;

      [CascadingParameter]
      public ProcessError? ProcessError { get; set; }

      private void IncrementCount()
      {
          try
          {
              currentCount++;

              if (currentCount > 5)
              {
                  throw new InvalidOperationException("Current count is
  over five!");
              }
          }
          catch (Exception ex)
          {
              ProcessError?.LogError(ex);
          }
      }
  }
  ```

The logged error:

> fail: {COMPONENT NAMESPACE}.ProcessError[0]
> ProcessError.LogError: System.InvalidOperationException Message: Current count is over five!

If the `LogError` method directly participates in rendering, such as showing a custom error message bar or changing the CSS styles of the rendered elements, call StateHasChanged at the end of the `LogError` method to rerender the UI.

Because the approaches in this section handle errors with a try-catch statement, an app's SignalR connection between the client and server isn't broken when an error occurs and the circuit remains alive. Other unhandled exceptions remain fatal to a circuit. For more information, see the section on how a circuit reacts to unhandled exceptions.

Log errors with a persistent provider

If an unhandled exception occurs, the exception is logged to ILogger instances configured in the service container. Blazor apps log console output with the Console Logging Provider. Consider logging to a location on the server (or backend web API for client-side apps) with a provider that manages log size and log rotation. Alternatively, the app can use an Application Performance Management (APM) service, such as Azure Application Insights (Azure Monitor).

> **Note**
>
> Native Application Insights features to support client-side apps and native Blazor framework support for Google Analytics might become available in future releases of these technologies. For more information, see Support App Insights in Blazor WASM Client Side (microsoft/ApplicationInsights-dotnet #2143) and Web analytics and diagnostics (includes links to community implementations) (dotnet/aspnetcore #5461). In the meantime, a client-side app can use the Application Insights JavaScript SDK with JS interop to log errors directly to Application Insights from a client-side app.

During development in a Blazor app operating over a circuit, the app usually sends the full details of exceptions to the browser's console to aid in debugging. In production, detailed errors aren't sent to clients, but an exception's full details are logged on the server.

You must decide which incidents to log and the level of severity of logged incidents. Hostile users might be able to trigger errors deliberately. For example, don't log an incident from an error where an unknown `ProductId` is supplied in the URL of a component that displays product details. Not all errors should be treated as incidents for logging.

For more information, see the following articles:

- ASP.NET Core Blazor logging
- Handle errors in ASP.NET Core‡
- Create web APIs with ASP.NET Core

‡Applies to server-side Blazor apps and other server-side ASP.NET Core apps that are web API backend apps for Blazor. Client-side apps can trap and send error information on the client to a web API, which logs the error information to a persistent logging provider.

Places where errors may occur

Framework and app code may trigger unhandled exceptions in any of the following locations, which are described further in the following sections of this article:

- Component instantiation
- Lifecycle methods
- Rendering logic
- Event handlers
- Component disposal
- JavaScript interop
- Prerendering

Component instantiation

When Blazor creates an instance of a component:

- The component's constructor is invoked.
- The constructors of DI services supplied to the component's constructor via the @inject directive or the [Inject] attribute are invoked.

An error in an executed constructor or a setter for any `[Inject]` property results in an unhandled exception and stops the framework from instantiating the component. If the app is operating over a circuit, the circuit fails. If constructor logic may throw exceptions,

the app should trap the exceptions using a try-catch statement with error handling and logging.

Lifecycle methods

During the lifetime of a component, Blazor invokes lifecycle methods. If any lifecycle method throws an exception, synchronously or asynchronously, the exception is fatal to a circuit. For components to deal with errors in lifecycle methods, add error handling logic.

In the following example where OnParametersSetAsync calls a method to obtain a product:

- An exception thrown in the `ProductRepository.GetProductByIdAsync` method is handled by a try-catch statement.
- When the `catch` block is executed:
  - `loadFailed` is set to `true`, which is used to display an error message to the user.
  - The error is logged.

razor

```
@page "/product-details/{ProductId:int?}"
@inject ILogger<ProductDetails> Logger
@inject IProductRepository Product

<PageTitle>Product Details</PageTitle>

<h1>Product Details Example</h1>

@if (details != null)
{
    <h2>@details.ProductName</h2>
    <p>
        @details.Description
        <a href="@details.Url">Company Link</a>
    </p>

}
else if (loadFailed)
{
    <h1>Sorry, we could not load this product due to an error.</h1>
}
else
{
    <h1>Loading...</h1>
}

@code {
    private ProductDetail? details;
```

```
    private bool loadFailed;

    [Parameter]
    public int ProductId { get; set; }

    protected override async Task OnParametersSetAsync()
    {
        try
        {
            loadFailed = false;

            // Reset details to null to display the loading indicator
            details = null;

            details = await Product.GetProductByIdAsync(ProductId);
        }
        catch (Exception ex)
        {
            loadFailed = true;
            Logger.LogWarning(ex, "Failed to load product {ProductId}",
ProductId);
        }
    }

    public class ProductDetail
    {
        public string? ProductName { get; set; }
        public string? Description { get; set; }
        public string? Url { get; set; }
    }

    /*
    * Register the service in Program.cs:
    * using static BlazorSample.Components.Pages.ProductDetails;
    * builder.Services.AddScoped<IProductRepository, ProductRepository>();
    */

    public interface IProductRepository
    {
        public Task<ProductDetail> GetProductByIdAsync(int id);
    }

    public class ProductRepository : IProductRepository
    {
        public Task<ProductDetail> GetProductByIdAsync(int id)
        {
            return Task.FromResult(
                new ProductDetail()
                {
                    ProductName = "Flowbee ",
                    Description = "The Revolutionary Haircutting System
You've Come to Love!",
                    Url = "https://flowbee.com/"
                });
        }
```

```
    }
}
```

Rendering logic

The declarative markup in a Razor component file (`.razor`) is compiled into a C# method called BuildRenderTree. When a component renders, BuildRenderTree executes and builds up a data structure describing the elements, text, and child components of the rendered component.

Rendering logic can throw an exception. An example of this scenario occurs when `@someObject.PropertyName` is evaluated but `@someObject` is `null`. For Blazor apps operating over a circuit, an unhandled exception thrown by rendering logic is fatal to the app's circuit.

To prevent a NullReferenceException in rendering logic, check for a `null` object before accessing its members. In the following example, `person.Address` properties aren't accessed if `person.Address` is `null`:

razor

```
@if (person.Address != null)
{
    <div>@person.Address.Line1</div>
    <div>@person.Address.Line2</div>
    <div>@person.Address.City</div>
    <div>@person.Address.Country</div>
}
```

The preceding code assumes that `person` isn't `null`. Often, the structure of the code guarantees that an object exists at the time the component is rendered. In those cases, it isn't necessary to check for `null` in rendering logic. In the prior example, `person` might be guaranteed to exist because `person` is created when the component is instantiated, as the following example shows:

razor

```
@code {
    private Person person = new();

    ...
}
```

Event handlers

Client-side code triggers invocations of C# code when event handlers are created using:

- `@onclick`
- `@onchange`
- Other `@on...` attributes
- `@bind`

Event handler code might throw an unhandled exception in these scenarios.

If the app calls code that could fail for external reasons, trap exceptions using a try-catch statement with error handling and logging.

If an event handler throws an unhandled exception (for example, a database query fails) that isn't trapped and handled by developer code:

- The framework logs the exception.
- In a Blazor app operating over a circuit, the exception is fatal to the app's circuit.

Component disposal

A component may be removed from the UI, for example, because the user has navigated to another page. When a component that implements System.IDisposable is removed from the UI, the framework calls the component's Dispose method.

If the component's `Dispose` method throws an unhandled exception in a Blazor app operating over a circuit, the exception is fatal to the app's circuit.

If disposal logic may throw exceptions, the app should trap the exceptions using a try-catch statement with error handling and logging.

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

JavaScript interop

IJSRuntime is registered by the Blazor framework. IJSRuntime.InvokeAsync allows .NET code to make asynchronous calls to the JavaScript (JS) runtime in the user's browser.

The following conditions apply to error handling with InvokeAsync:

- If a call to InvokeAsync fails synchronously, a .NET exception occurs. A call to InvokeAsync may fail, for example, because the supplied arguments can't be

serialized. Developer code must catch the exception. If app code in an event handler or component lifecycle method doesn't handle an exception in a Blazor app operating over a circuit, the resulting exception is fatal to the app's circuit.

- If a call to InvokeAsync fails asynchronously, the .NET Task fails. A call to InvokeAsync may fail, for example, because the JS-side code throws an exception or returns a `Promise` that completed as `rejected`. Developer code must catch the exception. If using the await operator, consider wrapping the method call in a try-catch statement with error handling and logging. Otherwise in a Blazor app operating over a circuit, the failing code results in an unhandled exception that's fatal to the app's circuit.
- Calls to InvokeAsync must complete within a certain period or else the call times out. The default timeout period is one minute. The timeout protects the code against a loss in network connectivity or JS code that never sends back a completion message. If the call times out, the resulting System.Threading.Tasks fails with an OperationCanceledException. Trap and process the exception with logging.

Similarly, JS code may initiate calls to .NET methods indicated by the [JSInvokable] attribute. If these .NET methods throw an unhandled exception:

- In a Blazor app operating over a circuit, the exception isn't treated as fatal to the app's circuit.
- The JS-side `Promise` is rejected.

You have the option of using error handling code on either the .NET side or the JS side of the method call.

For more information, see the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Prerendering

Razor components are prerendered by default so that their rendered HTML markup is returned as part of the user's initial HTTP request.

In a Blazor app operating over a circuit, prerendering works by:

- Creating a new circuit for all of the prerendered components that are part of the same page.
- Generating the initial HTML.

- Treating the circuit as `disconnected` until the user's browser establishes a SignalR connection back to the same server. When the connection is established, interactivity on the circuit is resumed and the components' HTML markup is updated.

For prerendered client-side components, prerendering works by:

- Generating initial HTML on the server for all of the prerendered components that are part of the same page.
- Making the component interactive on the client after the browser has loaded the app's compiled code and the .NET runtime (if not already loaded) in the background.

If a component throws an unhandled exception during prerendering, for example, during a lifecycle method or in rendering logic:

- In a Blazor app operating over a circuit, the exception is fatal to the circuit. For prerendered client-side components, the exception prevents rendering the component.
- The exception is thrown up the call stack from the ComponentTagHelper.

Under normal circumstances when prerendering fails, continuing to build and render the component doesn't make sense because a working component can't be rendered.

To tolerate errors that may occur during prerendering, error handling logic must be placed inside a component that may throw exceptions. Use try-catch statements with error handling and logging. Instead of wrapping the ComponentTagHelper in a try-catch statement, place error handling logic in the component rendered by the ComponentTagHelper.

Advanced scenarios

Recursive rendering

Components can be nested recursively. This is useful for representing recursive data structures. For example, a `TreeNode` component can render more `TreeNode` components for each of the node's children.

When rendering recursively, avoid coding patterns that result in infinite recursion:

- Don't recursively render a data structure that contains a cycle. For example, don't render a tree node whose children includes itself.

- Don't create a chain of layouts that contain a cycle. For example, don't create a layout whose layout is itself.
- Don't allow an end user to violate recursion invariants (rules) through malicious data entry or JavaScript interop calls.

Infinite loops during rendering:

- Causes the rendering process to continue forever.
- Is equivalent to creating an unterminated loop.

In these scenarios, the Blazor fails and usually attempts to:

- Consume as much CPU time as permitted by the operating system, indefinitely.
- Consume an unlimited amount of memory. Consuming unlimited memory is equivalent to the scenario where an unterminated loop adds entries to a collection on every iteration.

To avoid infinite recursion patterns, ensure that recursive rendering code contains suitable stopping conditions.

Custom render tree logic

Most Razor components are implemented as Razor component files (`.razor`) and are compiled by the framework to produce logic that operates on a RenderTreeBuilder to render their output. However, a developer may manually implement RenderTreeBuilder logic using procedural C# code. For more information, see ASP.NET Core Blazor advanced scenarios (render tree construction).

> ⚠ **Warning**
>
> Use of manual render tree builder logic is considered an advanced and unsafe scenario, not recommended for general component development.

If RenderTreeBuilder code is written, the developer must guarantee the correctness of the code. For example, the developer must ensure that:

- Calls to OpenElement and CloseElement are correctly balanced.
- Attributes are only added in the correct places.

Incorrect manual render tree builder logic can cause arbitrary undefined behavior, including crashes, the app or server to stop responding, and security vulnerabilities.

Consider manual render tree builder logic on the same level of complexity and with the same level of *danger* as writing assembly code or Microsoft Intermediate Language (MSIL) instructions by hand.

Additional resources

- Handle caught exceptions outside of a Razor component's lifecycle
- ASP.NET Core Blazor logging
- Handle errors in ASP.NET Core†
- Create web APIs with ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

†Applies to backend ASP.NET Core web API apps that client-side Blazor apps use for logging.

ASP.NET Core Blazor SignalR guidance

Article • 09/27/2024

> **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 configure and manage SignalR connections in Blazor apps.

For general guidance on ASP.NET Core SignalR configuration, see the topics in the Overview of ASP.NET Core SignalR area of the documentation, especially ASP.NET Core SignalR configuration.

Server-side apps use ASP.NET Core SignalR to communicate with the browser. SignalR's hosting and scaling conditions apply to server-side apps.

Blazor works best when using WebSockets as the SignalR transport due to lower latency, reliability, and security. Long Polling is used by SignalR when WebSockets isn't available or when the app is explicitly configured to use Long Polling.

Azure SignalR Service with stateful reconnect

Stateful reconnect (WithStatefulReconnect) was released with .NET 8 but isn't currently supported for the Azure SignalR Service. For more information, see Stateful Reconnect Support? (Azure/azure-signalr #1878).

WebSocket compression for Interactive Server components

By default, Interactive Server components:

- Enable compression for WebSocket connections. DisableWebSocketCompression (default: `false`) controls WebSocket compression.
- Adopt a `frame-ancestors` Content Security Policy (CSP) directive set to `'self'`, which only permits embedding the app in an `<iframe>` of the origin from which

the app is served when compression is enabled or when a configuration for the WebSocket context is provided. `ContentSecurityFrameAncestorPolicy` controls the `frame-ancestors` CSP.

The `frame-ancestors` CSP can be removed manually by setting the value of ContentSecurityFrameAncestorsPolicy to `null`, as you may want to configure the CSP in a centralized way. When the `frame-ancestors` CSP is managed in a centralized fashion, care must be taken to apply a policy whenever the first document is rendered. We don't recommend removing the policy completely, as it might make the app vulnerable to attack.

Use ConfigureWebSocketAcceptContext to configure the WebSocketAcceptContext for the websocket connections used by the server components. By default, a policy that enables compression and sets a CSP for the frame ancestors defined in ContentSecurityFrameAncestorsPolicy is applied.

Usage examples:

Disable compression by setting DisableWebSocketCompression to `true`, which reduces the vulnerability of the app to attack but may result in reduced performance:

C#

```
builder.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode(o => o.DisableWebSocketCompression =
true)
```

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

C#

```
builder.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode(o =>
o.ContentSecurityFrameAncestorsPolicy = "'none'")
```

When compression is enabled, remove the `frame-ancestors` CSP by setting ContentSecurityFrameAncestorsPolicy to `null`. This scenario is only recommended for apps that set the CSP in a centralized way:

C#

```
builder.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode(o =>
o.ContentSecurityFrameAncestorsPolicy = null)
```

> **ⓘ Important**
>
> Browsers apply CSP directives from multiple CSP headers using the strictest policy directive value. Therefore, a developer can't add a weaker `frame-ancestors` policy than `'self'` on purpose or by mistake.
>
> Single quotes are required on the string value passed to ContentSecurityFrameAncestorsPolicy:
>
> ✗ `none`, `self`
>
> ✓ `'none'`, `'self'`
>
> Additional options include specifying one or more host sources and scheme sources.

For security implications, see Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering. For more information on the `frame-ancestors` directive, see CSP: frame-ancestors (MDN documentation).

Disable response compression for Hot Reload

When using Hot Reload, disable Response Compression Middleware in the `Development` environment. Whether or not the default code from a project template is used, always call UseResponseCompression first in the request processing pipeline.

In the `Program` file:

C#

```
if (!app.Environment.IsDevelopment())
{
    app.UseResponseCompression();
}
```

Client-side SignalR cross-origin negotiation for authentication

This section explains how to configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers.

Use SetBrowserRequestCredentials to set Include on cross-origin fetch requests.

`IncludeRequestCredentialsMessageHandler.cs`:

C#

```
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Http;

public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {

request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

Where a hub connection is built, assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

C#

```
private HubConnectionBuilder? hubConnection;

...

hubConnection = new HubConnectionBuilder()
    .WithUrl(new Uri(Navigation.ToAbsoluteUri("/chathub")), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler =
innerHandler };
    }).Build();
```

The preceding example configures the hub connection URL to the absolute URI address at `/chathub`. The URI can also be set via a string, for example

`https://signalr.example.com`, or via configuration. `Navigation` is an injected NavigationManager.

For more information, see ASP.NET Core SignalR configuration.

Client-side rendering

If prerendering is configured, prerendering occurs before the client connection to the server is established. For more information, see Prerender ASP.NET Core Razor components.

Prerendered state size and SignalR message size limit

A large prerendered state size may exceed the SignalR circuit message size limit, which results in the following:

- The SignalR circuit fails to initialize with an error on the client: Circuit host not initialized.
- The reconnection UI on the client appears when the circuit fails. Recovery isn't possible.

To resolve the problem, use ***either*** of the following approaches:

- Reduce the amount of data that you are putting into the prerendered state.
- Increase the SignalR message size limit. ***WARNING***: Increasing the limit may increase the risk of Denial of Service (DoS) attacks.

Additional client-side resources

- Secure a SignalR hub
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR configuration
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Use session affinity (sticky sessions) for server-side webfarm hosting

When more than one backend server is in use, the app must implement session affinity, also called *sticky sessions*. Session affinity ensures that a client's circuit reconnects to the

same server if the connection is dropped, which is important because client state is only held in the memory of the server that first established the client's circuit.

The following error is thrown by an app that hasn't enabled session affinity in a webfarm:

> Uncaught (in promise) Error: Invocation canceled due to the underlying connection being closed.

For more information on session affinity with Azure App Service hosting, see Host and deploy ASP.NET Core server-side Blazor apps.

Azure SignalR Service

The optional Azure SignalR Service works in conjunction with the app's SignalR hub for scaling up a server-side app to a large number of concurrent connections. In addition, the service's global reach and high-performance data centers significantly aid in reducing latency due to geography.

The service isn't required for Blazor apps hosted in Azure App Service or Azure Container Apps but can be helpful in other hosting environments:

- To facilitate connection scale out.
- Handle global distribution.

For more information, see Host and deploy ASP.NET Core server-side Blazor apps.

Server-side circuit handler options

Configure the circuit with CircuitOptions. View default values in the reference source.

> **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).

Read or set the options in the `Program` file with an options delegate to AddInteractiveServerComponents. The `{OPTION}` placeholder represents the option, and

the `{VALUE}` placeholder is the value.

In the `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents(options
=>
{
    options.{OPTION} = {VALUE};
});
```

To configure the HubConnectionContext, use HubConnectionContextOptions with AddHubOptions. View the defaults for the hub connection context options in reference source. For option descriptions in the SignalR documentation, see ASP.NET Core SignalR configuration. The `{OPTION}` placeholder represents the option, and the `{VALUE}` placeholder is the value.

> **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).

In the `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents().AddHu
bOptions(options =>
{
    options.{OPTION} = {VALUE};
});
```

> **Warning**
>
> The default value of MaximumReceiveMessageSize is 32 KB. Increasing the value may increase the risk of Denial of Service (DoS) attacks.
>
> Blazor relies on MaximumParallelInvocationsPerClient set to 1, which is the default value. For more information, see MaximumParallelInvocationsPerClient > 1 breaks

file upload in Blazor Server mode (dotnet/aspnetcore #53951).

For information on memory management, see Host and deploy ASP.NET Core server-side Blazor apps.

Blazor hub options

Configure MapBlazorHub options to control HttpConnectionDispatcherOptions of the Blazor hub. View the defaults for the hub connection dispatcher options in reference source. The `{OPTION}` placeholder represents the option, and the `{VALUE}` placeholder is the value.

> **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).

Place the call to `app.MapBlazorHub` after the call to `app.MapRazorComponents` in the app's `Program` file:

C#

```
app.MapBlazorHub(options =>
{
    options.{OPTION} = {VALUE};
});
```

Configuring the hub used by AddInteractiveServerRenderMode with MapBlazorHub fails with an `AmbiguousMatchException`:

> Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request matched multiple endpoints.

To workaround the problem for apps targeting .NET 8, give the custom-configured Blazor hub higher precedence using the WithOrder method:

C#

```
app.MapBlazorHub(options =>
{
    options.CloseOnAuthenticationExpiration = true;
}).WithOrder(-1);
```

For more information, see the following resources:

- MapBlazorHub configuration in NET8 throws a The request matched multiple endpoints exception (dotnet/aspnetcore #51698)
- Attempts to map multiple blazor entry points with MapBlazorHub causes Ambiguous Route Error. This worked with net7 (dotnet/aspnetcore #52156)

Maximum receive message size

This section only applies to projects that implement SignalR.

The maximum incoming SignalR message size permitted for hub methods is limited by the HubOptions.MaximumReceiveMessageSize (default: 32 KB). SignalR messages larger than MaximumReceiveMessageSize throw an error. The framework doesn't impose a limit on the size of a SignalR message from the hub to a client.

When SignalR logging isn't set to Debug or Trace, a message size error only appears in the browser's developer tools console:

> Error: Connection disconnected with error 'Error: Server returned an error on close: Connection closed with an error.'.

When SignalR server-side logging is set to Debug or Trace, server-side logging surfaces an InvalidDataException for a message size error.

`appsettings.Development.json`:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      ...
      "Microsoft.AspNetCore.SignalR": "Debug"
    }
  }
}
```

Error:

> System.IO.InvalidDataException: The maximum message size of 32768B was exceeded. The message size can be configured in AddHubOptions.

One approach involves increasing the limit by setting MaximumReceiveMessageSize in the `Program` file. The following example sets the maximum receive message size to 64 KB:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents()
    .AddHubOptions(options => options.MaximumReceiveMessageSize = 64 *
1024);
```

Increasing the SignalR incoming message size limit comes at the cost of requiring more server resources, and it increases the risk of Denial of Service (DoS) attacks. Additionally, reading a large amount of content in to memory as strings or byte arrays can also result in allocations that work poorly with the garbage collector, resulting in additional performance penalties.

A better option for reading large payloads is to send the content in smaller chunks and process the payload as a Stream. This can be used when reading large JavaScript (JS) interop JSON payloads or if JS interop data is available as raw bytes. For an example that demonstrates sending large binary payloads in server-side apps that uses techniques similar to the InputFile component, see the Binary Submit sample app and the Blazor InputLargeTextArea Component Sample.

> **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).

Forms that process large payloads over SignalR can also use streaming JS interop directly. For more information, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor. For a forms example that streams `<textarea>` data to the server, see Troubleshoot ASP.NET Core Blazor forms.

Consider the following guidance when developing code that transfers a large amount of data:

- Leverage the native streaming JS interop support to transfer data larger than the SignalR incoming message size limit:
  - Call JavaScript functions from .NET methods in ASP.NET Core Blazor
  - Call .NET methods from JavaScript functions in ASP.NET Core Blazor
  - Form payload example: Troubleshoot ASP.NET Core Blazor forms
- General tips:
  - Don't allocate large objects in JS and C# code.
  - Free consumed memory when the process is completed or cancelled.
  - Enforce the following additional requirements for security purposes:
    - Declare the maximum file or data size that can be passed.
    - Declare the minimum upload rate from the client to the server.
  - After the data is received by the server, the data can be:
    - Temporarily stored in a memory buffer until all of the segments are collected.
    - Consumed immediately. For example, the data can be stored immediately in a database or written to disk as each segment is received.

Blazor server-side Hub endpoint route configuration

In the `Program` file, call MapBlazorHub to map the Blazor Hub to the app's default path. The Blazor script (`blazor.*.js`) automatically points to the endpoint created by MapBlazorHub.

Reflect the server-side connection state in the UI

When the client detects that the connection has been lost, a default UI is displayed to the user while the client attempts to reconnect. If reconnection fails, the user is provided the option to retry.

To customize the UI, define a single element with an `id` of `components-reconnect-modal`. The following example places the element in the `App` component.

`App.razor`:

```
CSHTML
```

```
<div id="components-reconnect-modal">
    There was a problem with the connection!
</div>
```

> **Note**
>
> If more than one element with an `id` of `components-reconnect-modal` are rendered by the app, only the first rendered element receives CSS class changes to display or hide the element.

Add the following CSS styles to the site's stylesheet.

`wwwroot/app.css`:

```css
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show,
#components-reconnect-modal.components-reconnect-failed,
#components-reconnect-modal.components-reconnect-rejected {
    display: block;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element by the Blazor framework.

Expand table

CSS class	Indicates...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()` in JavaScript.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()` in JavaScript. This connection state may result when:

CSS class	Indicates...
	• A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the user's components are disposed. • The server is restarted, or the app's worker process is recycled.

Customize the delay before the reconnection UI appears by setting the `transition-delay` property in the site's CSS for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second).

`wwwroot/app.css`:

```css
#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

To display the current reconnect attempt, define an element with an `id` of `components-reconnect-current-attempt`. To display the maximum number of reconnect retries, define an element with an `id` of `components-reconnect-max-retries`. The following example places these elements inside a reconnect attempt modal element following the previous example.

```cshtml
<div id="components-reconnect-modal">
    There was a problem with the connection!
    (Current reconnect attempt:
    <span id="components-reconnect-current-attempt"></span> /
    <span id="components-reconnect-max-retries"></span>)
</div>
```

When the custom reconnect modal appears, it renders content similar to the following based on the preceding code:

```html
There was a problem with the connection! (Current reconnect attempt: 3 / 8)
```

Server-side rendering

By default, components are prerendered on the server before the client connection to the server is established. For more information, see Prerender ASP.NET Core Razor components.

Monitor server-side circuit activity

Monitor inbound circuit activity using the CreateInboundActivityHandler method on CircuitHandler. Inbound circuit activity is any activity sent from the browser to the server, such as UI events or JavaScript-to-.NET interop calls.

For example, you can use a circuit activity handler to detect if the client is idle and log its circuit ID (Circuit.Id):

C#

```
using Microsoft.AspNetCore.Components.Server.Circuits;
using Microsoft.Extensions.Options;
using Timer = System.Timers.Timer;

public sealed class IdleCircuitHandler : CircuitHandler, IDisposable
{
    private Circuit? currentCircuit;
    private readonly ILogger logger;
    private readonly Timer timer;

    public IdleCircuitHandler(ILogger<IdleCircuitHandler> logger,
        IOptions<IdleCircuitOptions> options)
    {
        timer = new Timer
        {
            Interval = options.Value.IdleTimeout.TotalMilliseconds,
            AutoReset = false
        };

        timer.Elapsed += CircuitIdle;
        this.logger = logger;
    }

    private void CircuitIdle(object? sender, System.Timers.ElapsedEventArgs
e)
    {
        logger.LogInformation("{CircuitId} is idle", currentCircuit?.Id);
    }

    public override Task OnCircuitOpenedAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        currentCircuit = circuit;

        return Task.CompletedTask;
```

```
    }

    public override Func<CircuitInboundActivityContext, Task>
CreateInboundActivityHandler(
        Func<CircuitInboundActivityContext, Task> next)
    {
        return context =>
        {
            timer.Stop();
            timer.Start();

            return next(context);
        };
    }

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

public class IdleCircuitOptions
{
    public TimeSpan IdleTimeout { get; set; } = TimeSpan.FromMinutes(5);
}

public static class IdleCircuitHandlerServiceCollectionExtensions
{
    public static IServiceCollection AddIdleCircuitHandler(
        this IServiceCollection services,
        Action<IdleCircuitOptions> configureOptions)
    {
        services.Configure(configureOptions);
        services.AddIdleCircuitHandler();

        return services;
    }

    public static IServiceCollection AddIdleCircuitHandler(
        this IServiceCollection services)
    {
        services.AddScoped<CircuitHandler, IdleCircuitHandler>();

        return services;
    }
}
```

Register the service in the `Program` file. The following example configures the default idle timeout of five minutes to five seconds in order to test the preceding `IdleCircuitHandler` implementation:

C#

```
builder.Services.AddIdleCircuitHandler(options =>
```

```
    options.IdleTimeout = TimeSpan.FromSeconds(5));
```

Circuit activity handlers also provide an approach for accessing scoped Blazor services from other non-Blazor dependency injection (DI) scopes. For more information and examples, see:

- ASP.NET Core Blazor dependency injection
- Server-side ASP.NET Core Blazor additional security scenarios

Blazor startup

Configure the manual start of a Blazor app's SignalR circuit in the `App.razor` file of a Blazor Web App:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.*.js` script.
- Place a script that calls `Blazor.start()` after the Blazor script is loaded and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start()` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

For more information, including how to initialize Blazor when the document is ready and how to chain to a JS Promise, see ASP.NET Core Blazor startup.

Configure SignalR timeouts and Keep-Alive on the client

Configure the following values for the client:

- `withServerTimeout`: Configures the server timeout in milliseconds. If this timeout elapses without receiving any messages from the server, the connection is terminated with an error. The default timeout value is 30 seconds. The server timeout should be at least double the value assigned to the Keep-Alive interval (`withKeepAliveInterval`).
- `withKeepAliveInterval`: Configures the Keep-Alive interval in milliseconds (default interval at which to ping the server). This setting allows the server to detect hard

disconnects, such as when a client unplugs their computer from the network. The ping occurs at most as often as the server pings. If the server pings every five seconds, assigning a value lower than `5000` (5 seconds) pings every five seconds. The default value is 15 seconds. The Keep-Alive interval should be less than or equal to half the value assigned to the server timeout (`withServerTimeout`).

The following example for the `App.razor` file (Blazor Web App) shows the assignment of default values.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.withServerTimeout(30000).withKeepAliveInterval(15000);
      }
    }
  });
</script>
```

The following example for the `Pages/_Host.cshtml` file (Blazor Server, all versions except ASP.NET Core in .NET 6) or `Pages/_Layout.cshtml` file (Blazor Server, ASP.NET Core in .NET 6).

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.withServerTimeout(30000).withKeepAliveInterval(15000);
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

When creating a hub connection in a component, set the ServerTimeout (default: 30 seconds) and KeepAliveInterval (default: 15 seconds) on the HubConnectionBuilder. Set the HandshakeTimeout (default: 15 seconds) on the built HubConnection. The following example shows the assignment of default values:

C#

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .WithServerTimeout(TimeSpan.FromSeconds(30))
        .WithKeepAliveInterval(TimeSpan.FromSeconds(15))
        .Build();

    hubConnection.HandshakeTimeout = TimeSpan.FromSeconds(15);

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

    await hubConnection.StartAsync();
}
```

When changing the values of the server timeout (ServerTimeout) or the Keep-Alive interval (KeepAliveInterval):

- The server timeout should be at least double the value assigned to the Keep-Alive interval.
- The Keep-Alive interval should be less than or equal to half the value assigned to the server timeout.

For more information, see the *Global deployment and connection failures* sections of the following articles:

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

Modify the server-side reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown`.
- Established/re-established connections use `onConnectionUp`.

Both `onConnectionDown` and `onConnectionUp` must be specified.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      reconnectionHandler: {
        onConnectionDown: (options, error) => console.error(error),
        onConnectionUp: () => console.log("Up, up, and away!")
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    reconnectionHandler: {
      onConnectionDown: (options, error) => console.error(error),
      onConnectionUp: () => console.log("Up, up, and away!")
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

Automatically refresh the page when server-side reconnection fails

The default reconnection behavior requires the user to take manual action to refresh the page after reconnection fails. However, a custom reconnection handler can be used to automatically refresh the page:

`App.razor`:

HTML

```
<div id="reconnect-modal" style="display: none;"></div>
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script src="boot.js"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

Create the following `wwwroot/boot.js` file.

Blazor Web App:

JavaScript

```
(() => {
  const maximumRetryCount = 3;
  const retryIntervalMilliseconds = 5000;
  const reconnectModal = document.getElementById('reconnect-modal');

  const startReconnectionProcess = () => {
    reconnectModal.style.display = 'block';

    let isCanceled = false;

    (async () => {
      for (let i = 0; i < maximumRetryCount; i++) {
        reconnectModal.innerText = `Attempting to reconnect: ${i + 1} of
${maximumRetryCount}`;

        await new Promise(resolve => setTimeout(resolve,
retryIntervalMilliseconds));

        if (isCanceled) {
          return;
        }

        try {
          const result = await Blazor.reconnect();
          if (!result) {
            // The server was reached, but the connection was rejected;
reload the page.
            location.reload();
```

```
            return;
          }

          // Successfully reconnected to the server.
          return;
        } catch {
          // Didn't reach the server; try again.
        }
      }

      // Retried too many times; reload the page.
      location.reload();
    })();

    return {
      cancel: () => {
        isCanceled = true;
        reconnectModal.style.display = 'none';
      },
    };
  };

  let currentReconnectionProcess = null;

  Blazor.start({
    circuit: {
      reconnectionHandler: {
        onConnectionDown: () => currentReconnectionProcess ??=
startReconnectionProcess(),
        onConnectionUp: () => {
          currentReconnectionProcess?.cancel();
          currentReconnectionProcess = null;
        }
      }
    }
  });
})();
```

Blazor Server:

JavaScript

```
(() => {
  const maximumRetryCount = 3;
  const retryIntervalMilliseconds = 5000;
  const reconnectModal = document.getElementById('reconnect-modal');

  const startReconnectionProcess = () => {
    reconnectModal.style.display = 'block';

    let isCanceled = false;

    (async () => {
```

```
      for (let i = 0; i < maximumRetryCount; i++) {
        reconnectModal.innerText = `Attempting to reconnect: ${i + 1} of
${maximumRetryCount}`;

        await new Promise(resolve => setTimeout(resolve,
retryIntervalMilliseconds));

        if (isCanceled) {
          return;
        }

        try {
          const result = await Blazor.reconnect();
          if (!result) {
            // The server was reached, but the connection was rejected;
reload the page.
            location.reload();
            return;
          }

          // Successfully reconnected to the server.
          return;
        } catch {
          // Didn't reach the server; try again.
        }
      }

      // Retried too many times; reload the page.
      location.reload();
    })();

    return {
      cancel: () => {
        isCanceled = true;
        reconnectModal.style.display = 'none';
      },
    };
  };

  let currentReconnectionProcess = null;

  Blazor.start({
    reconnectionHandler: {
      onConnectionDown: () => currentReconnectionProcess ??=
startReconnectionProcess(),
      onConnectionUp: () => {
        currentReconnectionProcess?.cancel();
        currentReconnectionProcess = null;
      }
    }
  });
})();
```

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

Adjust the server-side reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`).

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      reconnectionOptions: {
        maxRetries: 3,
        retryIntervalMilliseconds: 2000
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    reconnectionOptions: {
      maxRetries: 3,
      retryIntervalMilliseconds: 2000
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

When the user navigates back to an app with a disconnected circuit, reconnection is attempted immediately rather than waiting for the duration of the next reconnect interval. This behavior seeks to resume the connection as quickly as possible for the user.

The default reconnect timing uses a computed backoff strategy. The first several reconnection attempts occur in rapid succession before computed delays are introduced between attempts. The default logic for computing the retry interval is an implementation detail subject to change without notice, but you can find the default logic that the Blazor framework uses in the computeDefaultRetryInterval function (reference source).

> **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).

Customize the retry interval behavior by specifying a function to compute the retry interval. In the following exponential backoff example, the number of previous reconnection attempts is multiplied by 1,000 ms to calculate the retry interval. When the count of previous attempts to reconnect (`previousAttempts`) is greater than the maximum retry limit (`maxRetries`), `null` is assigned to the retry interval (`retryIntervalMilliseconds`) to cease further reconnection attempts:

JavaScript

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

An alternative is to specify the exact sequence of retry intervals. After the last specified retry interval, retries stop because the `retryIntervalMilliseconds` function returns `undefined`:

JavaScript

```
Blazor.start({
  circuit: {
    reconnectionOptions: {
      retryIntervalMilliseconds:
        Array.prototype.at.bind([0, 1000, 2000, 5000, 10000, 15000, 30000]),
```

```
    },
  },
});
```

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

Control when the reconnection UI appears

Controlling when the reconnection UI appears can be useful in the following situations:

- A deployed app frequently displays the reconnection UI due to ping timeouts caused by internal network or Internet latency, and you would like to increase the delay.
- An app should report to users that the connection has dropped sooner, and you would like to shorten the delay.

The timing of the appearance of the reconnection UI is influenced by adjusting the Keep-Alive interval and timeouts on the client. The reconnection UI appears when the server timeout is reached on the client (`withServerTimeout`, Client configuration section). However, changing the value of `withServerTimeout` requires changes to other Keep-Alive, timeout, and handshake settings described in the following guidance.

As general recommendations for the guidance that follows:

- The Keep-Alive interval should match between client and server configurations.
- Timeouts should be at least double the value assigned to the Keep-Alive interval.

Server configuration

Set the following:

- ClientTimeoutInterval (default: 30 seconds): The time window clients have to send a message before the server closes the connection.
- HandshakeTimeout (default: 15 seconds): The interval used by the server to timeout incoming handshake requests by clients.
- KeepAliveInterval (default: 15 seconds): The interval used by the server to send keep alive pings to connected clients. Note that there is also a Keep-Alive interval setting on the client, which should match the server's value.

The ClientTimeoutInterval and HandshakeTimeout can be increased, and the KeepAliveInterval can remain the same. The important consideration is that if you change the values, make sure that the timeouts are at least double the value of the Keep-Alive interval and that the Keep-Alive interval matches between server and client.

For more information, see the Configure SignalR timeouts and Keep-Alive on the client section.

In the following example:

- The ClientTimeoutInterval is increased to 60 seconds (default value: 30 seconds).
- The HandshakeTimeout is increased to 30 seconds (default value: 15 seconds).
- The KeepAliveInterval isn't set in developer code and uses its default value of 15 seconds. Decreasing the value of the Keep-Alive interval increases the frequency of communication pings, which increases the load on the app, server, and network. Care must be taken to avoid introducing poor performance when lowering the Keep-Alive interval.

Blazor Web App (.NET 8 or later) in the server project's `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents()
    .AddHubOptions(options =>
{
    options.ClientTimeoutInterval = TimeSpan.FromSeconds(60);
    options.HandshakeTimeout = TimeSpan.FromSeconds(30);
});
```

Blazor Server in the `Program` file:

```
C#

builder.Services.AddServerSideBlazor()
    .AddHubOptions(options =>
    {
        options.ClientTimeoutInterval = TimeSpan.FromSeconds(60);
        options.HandshakeTimeout = TimeSpan.FromSeconds(30);
    });
```

For more information, see the Server-side circuit handler options section.

Client configuration

Set the following:

- `withServerTimeout` (default: 30 seconds): Configures the server timeout, specified in milliseconds, for the circuit's hub connection.
- `withKeepAliveInterval` (default: 15 seconds): The interval, specified in milliseconds, at which the connection sends Keep-Alive messages.

The server timeout can be increased, and the Keep-Alive interval can remain the same. The important consideration is that if you change the values, make sure that the server timeout is at least double the value of the Keep-Alive interval and that the Keep-Alive interval values match between server and client. For more information, see the Configure SignalR timeouts and Keep-Alive on the client section.

In the following startup configuration example (location of the Blazor script), a custom value of 60 seconds is used for the server timeout. The Keep-Alive interval (`withKeepAliveInterval`) isn't set and uses its default value of 15 seconds.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.withServerTimeout(60000);
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.withServerTimeout(60000);
    }
  });
</script>
```

When creating a hub connection in a component, set the server timeout (WithServerTimeout, default: 30 seconds) on the HubConnectionBuilder. Set the HandshakeTimeout (default: 15 seconds) on the built HubConnection. Confirm that the timeouts are at least double the Keep-Alive interval (WithKeepAliveInterval/KeepAliveInterval) and that the Keep-Alive value matches between server and client.

The following example is based on the `Index` component in the SignalR with Blazor tutorial. The server timeout is increased to 60 seconds, and the handshake timeout is increased to 30 seconds. The Keep-Alive interval isn't set and uses its default value of 15 seconds.

C#

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .WithServerTimeout(TimeSpan.FromSeconds(60))
        .Build();

    hubConnection.HandshakeTimeout = TimeSpan.FromSeconds(30);

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

    await hubConnection.StartAsync();
}
```

Disconnect the Blazor circuit from the client

A Blazor circuit is disconnected when the unload page event is triggered. To disconnect the circuit for other scenarios on the client, invoke `Blazor.disconnect` in the appropriate event handler. In the following example, the circuit is disconnected when the page is hidden (pagehide event):

JavaScript

```
window.addEventListener('pagehide', () => {
  Blazor.disconnect();
});
```

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

Server-side circuit handler

You can define a *circuit handler*, which allows running code on changes to the state of a user's circuit. A circuit handler is implemented by deriving from CircuitHandler and registering the class in the app's service container. The following example of a circuit handler tracks open SignalR connections.

`TrackingCircuitHandler.cs`:

```
C#

using Microsoft.AspNetCore.Components.Server.Circuits;

public class TrackingCircuitHandler : CircuitHandler
{
    private HashSet<Circuit> circuits = new();

    public override Task OnConnectionUpAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Add(circuit);

        return Task.CompletedTask;
    }

    public override Task OnConnectionDownAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Remove(circuit);

        return Task.CompletedTask;
    }

    public int ConnectedCircuits => circuits.Count;
}
```

Circuit handlers are registered using DI. Scoped instances are created per instance of a circuit. Using the `TrackingCircuitHandler` in the preceding example, a singleton service is created because the state of all circuits must be tracked.

In the `Program` file:

```
C#

builder.Services.AddSingleton<CircuitHandler, TrackingCircuitHandler>();
```

If a custom circuit handler's methods throw an unhandled exception, the exception is fatal to the circuit. To tolerate exceptions in a handler's code or called methods, wrap the code in one or more try-catch statements with error handling and logging.

When a circuit ends because a user has disconnected and the framework is cleaning up the circuit state, the framework disposes of the circuit's DI scope. Disposing the scope disposes any circuit-scoped DI services that implement System.IDisposable. If any DI service throws an unhandled exception during disposal, the framework logs the exception. For more information, see ASP.NET Core Blazor dependency injection.