Blazor apps are based on *components*. A component in Blazor is an element of UI, such as a page, dialog, or data entry form.

Components are .NET C# classes built into .NET assemblies that:

- Define flexible UI rendering logic.
- Handle user events.
- Can be nested and reused.
- Can be shared and distributed as Razor class libraries or NuGet packages.

The component class is usually written in the form of a Razor markup page with a `.razor` file extension. Components in Blazor are formally referred to as *Razor components*, informally as *Blazor components*. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor allows you to switch between HTML markup and C# in the same file with IntelliSense programming support in Visual Studio.

Blazor uses natural HTML tags for UI composition. The following Razor markup demonstrates a component that increments a counter when the user selects a button.

razor

```razor
<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++;
    }
}
```

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

Build a full-stack web app with Blazor

Blazor Web Apps provide a component-based architecture with server-side rendering and full client-side interactivity in a single solution, where you can switch between server-side and client-side rendering modes and even mix them in the same page.

Blazor Web Apps can quickly deliver UI to the browser by statically rendering HTML content from the server in response to requests. The page loads fast because UI rendering is performed quickly on the server without the need to download a large JavaScript bundle. Blazor can also further improve the user experience with various progressive enhancements to server rendering, such as enhanced navigation with form posts and streaming rendering of asynchronously-generated content.

Blazor supports *interactive* server-side rendering (interactive SSR), where UI interactions are handled from the server over a real-time connection with the browser. Interactive SSR enables a rich user experience like one would expect from a client app but without the need to create API endpoints to access server resources. Page content for interactive pages is prerendered, where content on the server is initially generated and sent to the client 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.

Blazor Web Apps support interactivity with client-side rendering (CSR) that relies on a .NET runtime built with WebAssembly that you can download with your app. When running Blazor on WebAssembly, your .NET code can access the full functionality of the browser and interop with JavaScript. Your .NET code runs in the browser's security sandbox with the protections that the sandbox provides against malicious actions on the client machine.

Blazor apps can entirely target running on WebAssembly in the browser without the involvement of a server. For a *standalone Blazor WebAssembly app*, assets are deployed as static files to a web server or service capable of serving static content to clients. Once downloaded, standalone Blazor WebAssembly apps can be cached and executed offline as a Progressive Web App (PWA).

Build a native client app with Blazor Hybrid

Blazor Hybrid enables using Razor components in a native client app with a blend of native and web technologies for web, mobile, and desktop platforms. Code runs natively in the .NET process and renders web UI to an embedded Web View control using a local interop channel. WebAssembly isn't used in Hybrid apps. Hybrid apps are built with .NET Multi-platform App UI (.NET MAUI), which is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.

The Blazor Hybrid supports Windows Presentation Foundation (WPF) and Windows Forms to transition apps from earlier technology to .NET MAUI.

Next steps

Blazor Tutorial - Build your first Blazor app

ASP.NET Core Blazor supported platforms

ASP.NET Core Blazor supported platforms

Article • 07/01/2024

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

Blazor is supported in the browsers shown in the following table on both mobile and desktop platforms.

Expand table

Browser	Version
Apple Safari	Current†
Google Chrome	Current†
Microsoft Edge	Current†
Mozilla Firefox	Current†

†*Current* refers to the latest version of the browser.

For Blazor Hybrid apps, we test on and support the latest platform Web View control versions:

- Microsoft Edge WebView2 on Windows
- Chrome on Android
- Safari on iOS and macOS

Additional resources

- ASP.NET Core Blazor hosting models
- ASP.NET Core SignalR supported platforms

Tooling for ASP.NET Core Blazor

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 tools for building Blazor apps using several tools:

- Visual Studio (VS): The most comprehensive integrated development environment (IDE) for .NET developers on Windows. Includes an array of tools and features to elevate and enhance every stage of software development.
- Visual Studio Code (VS Code) is an open source, cross-platform code editor that can be used to develop Blazor apps.
- .NET CLI: The .NET command-line interface (CLI) is a cross-platform toolchain for developing, building, running, and publishing .NET applications. The .NET CLI is included with the .NET SDK and runs on any platform supported by the SDK.

Select the pivot of this article that matches your tooling choice.

To create a Blazor app with Visual Studio, use the following guidance:

- Install the latest version of Visual Studio with the **ASP.NET and web development** workload.
- Create a new project using one of the available Blazor templates:
  - **Blazor Web App**: Creates a Blazor web app that supports interactive server-side rendering (interactive SSR) and client-side rendering (CSR). The Blazor Web App template is recommended for getting started with Blazor to learn about server-side and client-side Blazor features.
  - **Blazor WebAssembly Standalone App**: Creates a standalone client web app that can be deployed as a static site.

Select **Next**.

- Provide a **Project name** and confirm that the **Location** is correct.

- For more information on the options in the **Additional information** dialog, see the Blazor project templates and template options section.

> **Note**
>
> The hosted Blazor WebAssembly project template isn't available in ASP.NET Core 8.0 or later. To create a hosted Blazor WebAssembly app, a **Framework** option earlier than .NET 8.0 must be selected with the **ASP.NET Core Hosted** checkbox.

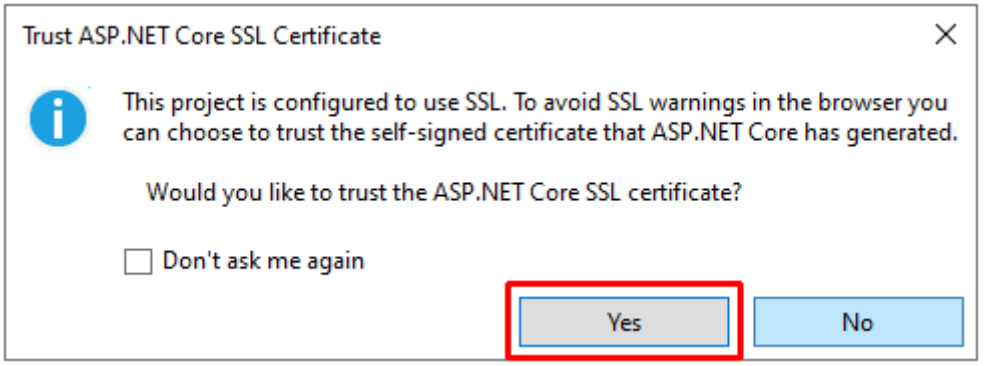
- Select **Create**.

Run the app

> **Important**
>
> When executing a Blazor Web App, run the app from the solution's server project, which is the project with a name that doesn't end in `.Client`.

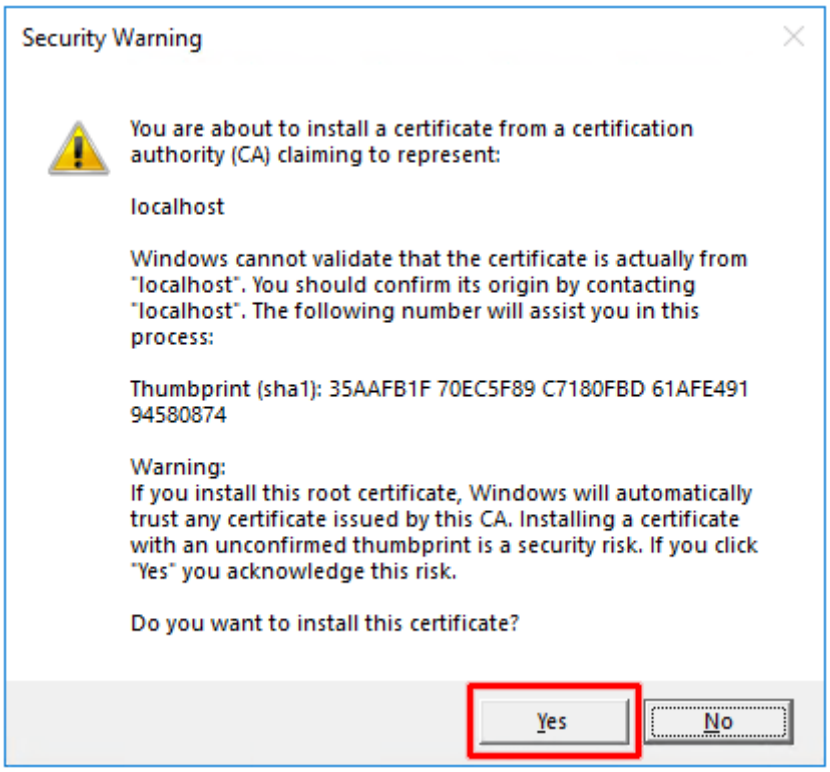
Press `Ctrl`+`F5` on the keyboard to run the app without the debugger.

Visual Studio displays the following dialog when a project isn't configured to use SSL:



Select **Yes** if you trust the ASP.NET Core SSL certificate.

The following dialog is displayed:



Select **Yes** to acknowledge the risk and install the certificate.

Visual Studio:

- Compiles and runs the app.
- Launches the default browser at `https://localhost:{PORT}`, which displays the app's UI. The `{PORT}` placeholder is the random port assigned at app creation. If you need to change the port due to a local port conflict, change the port in the project's `Properties/launchSettings.json` file.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

    
  - Press `Shift`+`F5` on the keyboard.

Visual Studio solution file (`.sln`)

A *solution* is a container to organize one or more related code projects. Solution files use a unique format and aren't intended to be edited directly.

Visual Studio and Visual Studio Code (VS Code) use a solution file (`.sln`) to store settings for a solution. The .NET CLI doesn't organize projects using a solution file, but it can create solution files and list/modify the projects in solution files via the dotnet sln command. Other .NET CLI commands use the path of the solution file for various publishing, testing, and packaging commands.

For more information, see the following resources:

- Introduction to projects and solutions (Visual Studio documentation)
- What are solutions and projects in Visual Studio? (Visual Studio documentation)
- Project management (VS Code documentation)

Blazor project templates and template options

The Blazor framework provides project templates for creating new apps. The templates are used to create new Blazor projects and solutions regardless of the tooling that you select for Blazor development (Visual Studio, Visual Studio Code, or the .NET command-line interface (CLI)):

- Blazor Web App project template: `blazor`
- Standalone Blazor WebAssembly app project template: `blazorwasm`

For more information on Blazor project templates, see ASP.NET Core Blazor project structure.

Rendering terms and concepts used in the following subsections are introduced in the following sections of the *Fundamentals* overview article:

- Client and server rendering concepts
- Static and interactive rendering concepts
- Render modes

Detailed guidance on render modes is provided by the ASP.NET Core Blazor render modes article.

Interactive render mode

- Interactive server-side rendering (interactive SSR) is enabled with the **Server** option.

- To only enable interactivity with client-side rendering (CSR), use the **WebAssembly** option.
- To enable both interactive rendering modes and the ability to automatically switch between them at runtime, use the **Auto (Server and WebAssembly)** (automatic) render mode option.
- If interactivity is set to `None`, the generated app has no interactivity. The app is only configured for static server-side rendering.

The Interactive Auto render mode initially uses interactive SSR while the .NET app bundle and runtime are download to the browser. After the .NET WebAssembly runtime is activated, the render mode switches to Interactive WebAssembly rendering.

The Blazor Web App template enables both static and interactive SSR using a single project. If you also enable CSR, 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. Any components using the WebAssembly or automatic render modes must be built from the client project.

> **ⓘ 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.

Interactivity location

Interactivity location options:

- **Per page/component**: The default sets up interactivity per page or per component.
- **Global**: Using this option sets up interactivity globally for the entire app.

Interactivity location can only be set if **Interactive render mode** isn't `None` and authentication isn't enabled.

Sample pages

To include sample pages and a layout based on Bootstrap styling, use the **Include sample pages** option. Disable this option for project without sample pages and

Bootstrap styling.

Additional guidance on template options

- ASP.NET Core Blazor render modes
- The *.NET default templates for dotnet new* article in the .NET Core documentation:
  - blazor
  - blazorwasm
- Passing the help option (`-h` or `--help`) to the dotnet new CLI command in a command shell:
  - `dotnet new blazor -h`
  - `dotnet new blazorwasm -h`

Additional resources

- Visual Studio
- Visual Studio Code
- ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation
- .NET command-line interface (CLI)
- .NET SDK
- .NET Hot Reload support for ASP.NET Core
- ASP.NET Core Blazor hosting models
- ASP.NET Core Blazor project structure
- ASP.NET Core Blazor Hybrid tutorials

ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation

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 the build tools for standalone Blazor WebAssembly apps and how to compile an app ahead of deployment with ahead-of-time (AOT) compilation.

Although the article primarily focuses on standalone Blazor WebAssembly apps, the section on heap size for some mobile device browsers also applies to the client-side project (`.Client`) of a Blazor Web App.

.NET WebAssembly build tools

The .NET WebAssembly build tools are based on Emscripten, a compiler toolchain for the web platform. To install the build tools, use ***either*** of the following approaches:

- For the **ASP.NET and web development** workload in the Visual Studio installer, select the **.NET WebAssembly build tools** option from the list of optional components.
- Execute `dotnet workload install wasm-tools` in an administrative command shell.

> **Note**
>
> .NET WebAssembly build tools for .NET 6 projects
>
> The `wasm-tools` workload installs the build tools for the latest release. However, the current version of the build tools are incompatible with existing projects built with .NET 6. Projects using the build tools that must support both .NET 6 and a later release must use multi-targeting.

Use the `wasm-tools-net6` workload for .NET 6 projects when developing apps with the .NET 7 SDK. To install the `wasm-tools-net6` workload, execute the following command from an administrative command shell:

.NET CLI

```
dotnet workload install wasm-tools-net6
```

Ahead-of-time (AOT) compilation

Blazor WebAssembly supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size.

Without enabling AOT compilation, Blazor WebAssembly apps run on the browser using a .NET Intermediate Language (IL) interpreter implemented in WebAssembly with partial just-in-time (JIT) runtime support, informally referred to as the *Jiterpreter*. Because the .NET IL code is interpreted, apps typically run slower than they would on a server-side .NET JIT runtime without any IL interpretation. AOT compilation addresses this performance issue by compiling an app's .NET code directly into WebAssembly for native WebAssembly execution by the browser. The AOT performance improvement can yield dramatic improvements for apps that execute CPU-intensive tasks. The drawback to using AOT compilation is that AOT-compiled apps are generally larger than their IL-interpreted counterparts, so they usually take longer to download to the client when first requested.

For guidance on installing the .NET WebAssembly build tools, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

To enable WebAssembly AOT compilation, add the `<RunAOTCompilation>` property set to `true` to the Blazor WebAssembly app's project file:

XML

```
<PropertyGroup>
  <RunAOTCompilation>true</RunAOTCompilation>
</PropertyGroup>
```

To compile the app to WebAssembly, publish the app. Publishing the `Release` configuration ensures the .NET Intermediate Language (IL) linking is also run to reduce the size of the published app:

.NET CLI

```
dotnet publish -c Release
```

WebAssembly AOT compilation is only performed when the project is published. AOT compilation isn't used when the project is run during development (`Development` environment) because AOT compilation usually takes several minutes on small projects and potentially much longer for larger projects. Reducing the build time for AOT compilation is under development for future releases of ASP.NET Core.

The size of an AOT-compiled Blazor WebAssembly app is generally larger than the size of the app if compiled into .NET IL:

- Although the size difference depends on the app, most AOT-compiled apps are about twice the size of their IL-compiled versions. This means that using AOT compilation trades off load-time performance for runtime performance. Whether this tradeoff is worth using AOT compilation depends on your app. Blazor WebAssembly apps that are CPU intensive generally benefit the most from AOT compilation.

- The larger size of an AOT-compiled app is due to two conditions:
  - More code is required to represent high-level .NET IL instructions in native WebAssembly.
  - AOT doesn't trim out managed DLLs when the app is published. Blazor requires the DLLs for reflection metadata and to support certain .NET runtime features. Requiring the DLLs on the client increases the download size but provides a more compatible .NET experience.

> **Note**
>
> For Mono /WebAssembly MSBuild properties and targets, see WasmApp.Common.targets (dotnet/runtime GitHub repository). Official documentation for common MSBuild properties is planned per Document blazor msbuild configuration options (dotnet/docs #27395).

Trim .NET IL after ahead-of-time (AOT) compilation

The `WasmStripILAfterAOT` MSBuild option enables removing the .NET Intermediate Language (IL) for compiled methods after performing AOT compilation to

WebAssembly, which reduces the size of the `_framework` folder.

In the app's project file:

```xml
<PropertyGroup>
  <RunAOTCompilation>true</RunAOTCompilation>
  <WasmStripILAfterAOT>true</WasmStripILAfterAOT>
</PropertyGroup>
```

This setting trims away the IL code for most compiled methods, including methods from libraries and methods in the app. Not all compiled methods can be trimmed, as some are still required by the .NET interpreter at runtime.

To report a problem with the trimming option, open an issue on the dotnet/runtime GitHub repository.

Disable the trimming property if it prevents your app from running normally:

```xml
<WasmStripILAfterAOT>false</WasmStripILAfterAOT>
```

Heap size for some mobile device browsers

When building a Blazor app that runs on the client and targets mobile device browsers, especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Runtime relinking

One of the largest parts of a Blazor WebAssembly app is the WebAssembly-based .NET runtime (`dotnet.wasm`) that the browser must download when the app is first accessed by a user's browser. Relinking the .NET WebAssembly runtime trims unused runtime code and thus improves download speed.

Runtime relinking requires installation of the .NET WebAssembly build tools. For more information, see Tooling for ASP.NET Core Blazor.

With the .NET WebAssembly build tools installed, runtime relinking is performed automatically when an app is **published** in the `Release` configuration. The size reduction

is particularly dramatic when disabling globalization. For more information, see ASP.NET Core Blazor globalization and localization.

> **ⓘ Important**
>
> Runtime relinking trims class instance JavaScript-invokable .NET methods unless they're protected. For more information, see **Call .NET methods from JavaScript functions in ASP.NET Core Blazor**.

Single Instruction, Multiple Data (SIMD)

Blazor uses WebAssembly Single Instruction, Multiple Data (SIMD) to improve the throughput of vectorized computations by performing an operation on multiple pieces of data in parallel using a single instruction.

To disable SIMD, for example when targeting old browsers or browsers on mobile devices that don't support SIMD, set the `<WasmEnableSIMD>` property to `false` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>
  <WasmEnableSIMD>false</WasmEnableSIMD>
</PropertyGroup>
```

For more information, see Configuring and hosting .NET WebAssembly applications: SIMD - Single instruction, multiple data and note that the guidance isn't versioned and applies to the latest public release.

Exception handling

Exception handling is enabled by default. To disable exception handling, add the `<WasmEnableExceptionHandling>` property with a value of `false` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>
  <WasmEnableExceptionHandling>false</WasmEnableExceptionHandling>
</PropertyGroup>
```

For more information, see the following resources:

- Configuring and hosting .NET WebAssembly applications: EH - Exception handling
- Exception handling

Additional resources

- ASP.NET Core Blazor WebAssembly native dependencies
- Webcil packaging format for .NET assemblies

ASP.NET Core Blazor hosting models

Article • 09/10/2024

> **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 hosting models, primarily focused on Blazor Server and Blazor WebAssembly apps in versions of .NET earlier than .NET 8. The guidance in this article is relevant under all .NET releases for Blazor Hybrid apps that run on native mobile and desktop platforms. Blazor Web Apps in .NET 8 or later are better conceptualized by how Razor components are rendered, which is described as their *render mode*. Render modes are briefly touched on in the *Fundamentals* overview article and covered in detail in ASP.NET Core Blazor render modes of the *Components* node.

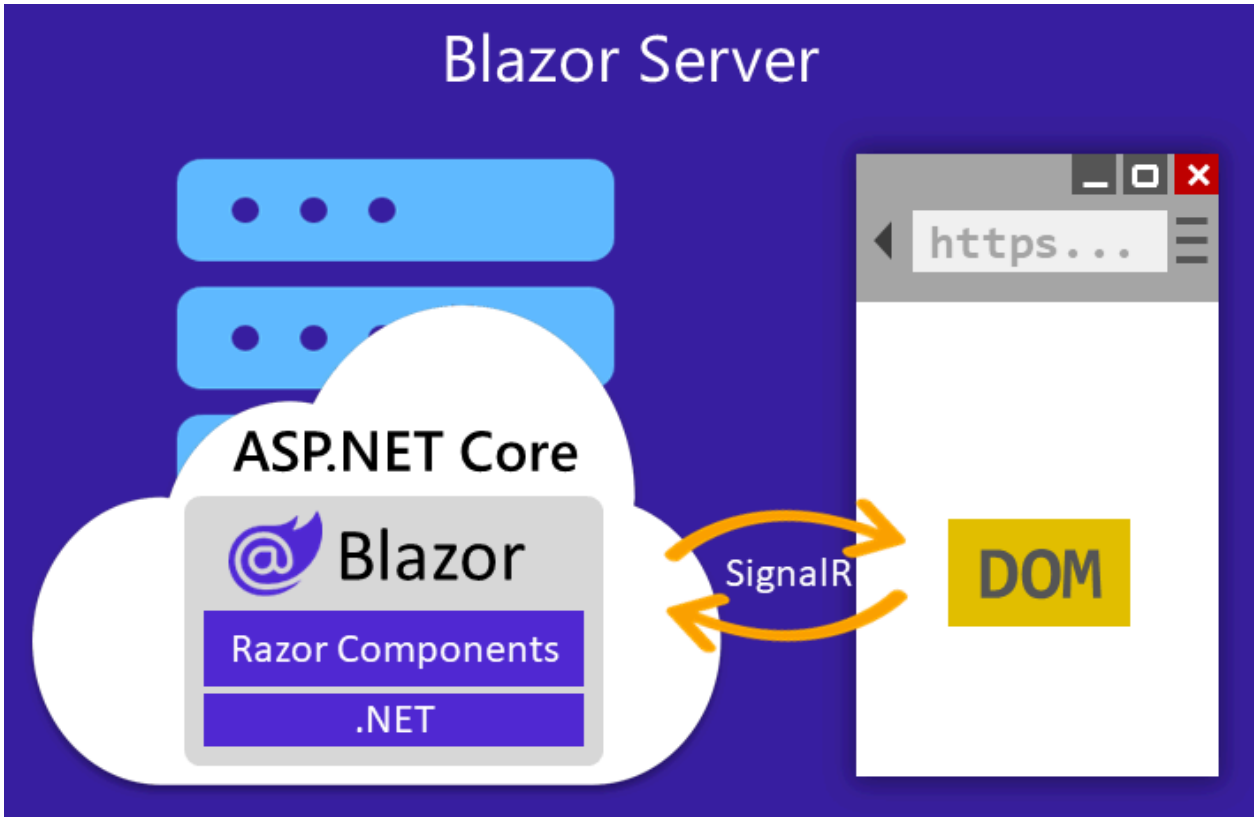
Blazor is a web framework for building web UI components (Razor components) that can be hosted in different ways. Razor components can run server-side in ASP.NET Core (*Blazor Server*) versus client-side in the browser on a WebAssembly -based .NET runtime (*Blazor WebAssembly*, *Blazor WASM*). You can also host Razor components in native mobile and desktop apps that render to an embedded Web View control (*Blazor Hybrid*). Regardless of the hosting model, the way you build Razor components *is the same*. The same Razor components can be used with any of the hosting models unchanged.

Blazor Server

With the Blazor Server hosting model, components are executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection using the WebSockets protocol. The state on the server associated with each connected client is called a *circuit*. Circuits aren't tied to a specific network connection and can tolerate temporary network interruptions and attempts by the client to reconnect to the server when the connection is lost.

In a traditional server-rendered app, opening the same app in multiple browser screens (tabs or `iframes`) typically doesn't translate into additional resource demands on the server. For the Blazor Server hosting model, each browser screen requires a separate

circuit and separate instances of server-managed component state. Blazor considers closing a browser tab or navigating to an external URL a *graceful* termination. In the event of a graceful termination, the circuit and associated resources are immediately released. A client may also disconnect non-gracefully, for instance due to a network interruption. Blazor Server stores disconnected circuits for a configurable interval to allow the client to reconnect.



On the client, the Blazor script establishes the SignalR connection with the server. The script is served from an embedded resource in the ASP.NET Core shared framework.

The Blazor Server hosting model offers several benefits:

- Download size is significantly smaller than when the Blazor WebAssembly hosting model is used, and the app loads much faster.
- The app takes full advantage of server capabilities, including the use of .NET Core APIs.
- .NET Core on the server is used to run the app, so existing .NET tooling, such as debugging, works as expected.
- Thin clients are supported. For example, Blazor Server works with browsers that don't support WebAssembly and on resource-constrained devices.
- The app's .NET/C# code base, including the app's component code, isn't served to clients.

The Blazor Server hosting model has the following limitations:

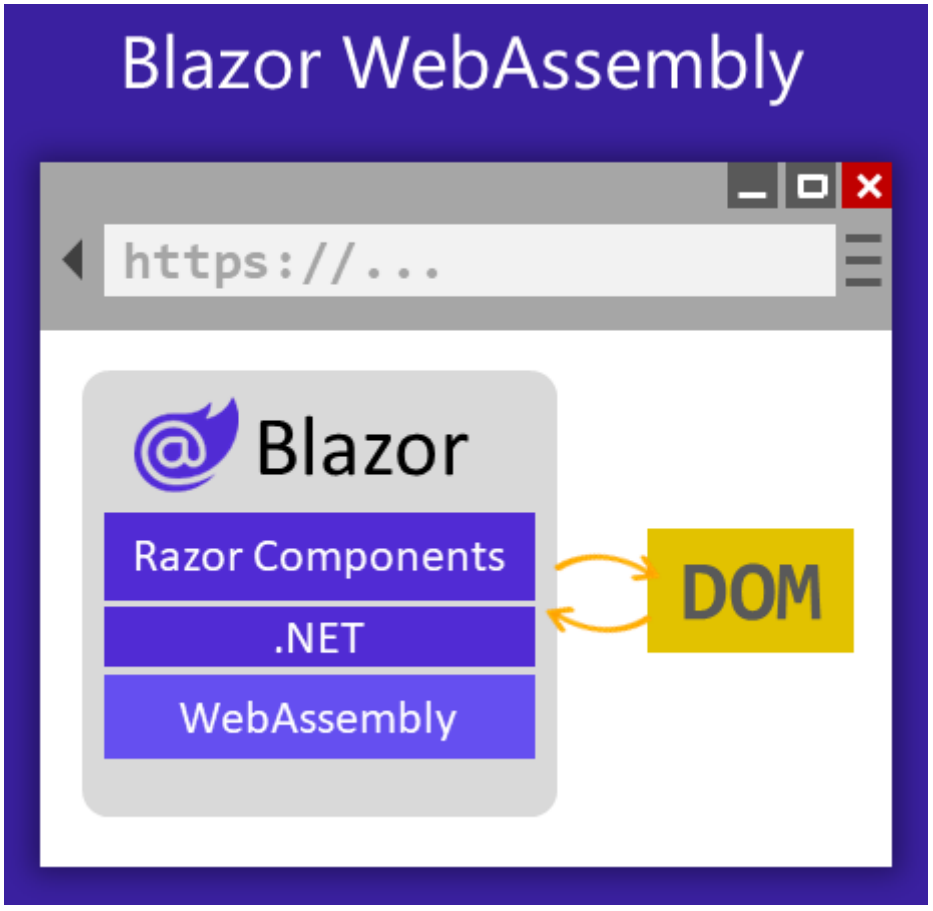
- Higher latency usually exists. Every user interaction involves a network hop.

- There's no offline support. If the client connection fails, interactivity fails.
- Scaling apps with many users requires server resources to handle multiple client connections and client state.
- An ASP.NET Core server is required to serve the app. Serverless deployment scenarios aren't possible, such as serving the app from a Content Delivery Network (CDN).

We recommend using the Azure SignalR Service for apps that adopt the Blazor Server hosting model. The service allows for scaling up a Blazor Server app to a large number of concurrent SignalR connections.

Blazor WebAssembly

The Blazor WebAssembly hosting model runs components client-side in the browser on a WebAssembly-based .NET runtime. Razor components, their dependencies, and the .NET runtime are downloaded to the browser. Components are executed directly on the browser UI thread. UI updates and event handling occur within the same process. Assets are deployed as static files to a web server or service capable of serving static content to clients.



Blazor web apps can use the Blazor WebAssembly hosting model to enable client-side interactivity. When an app is created that exclusively runs on the Blazor WebAssembly

hosting model without server-side rendering and interactivity, the app is called a *standalone* Blazor WebAssembly app.

When a standalone Blazor WebAssembly app uses a backend ASP.NET Core app to serve its files, the app is called a *hosted* Blazor WebAssembly app. Using hosted Blazor WebAssembly, you get a full-stack web development experience with .NET, including the ability to share code between the client and server apps, support for prerendering, and integration with MVC and Razor Pages. A hosted client app can interact with its backend server app over the network using a variety of messaging frameworks and protocols, such as web API, gRPC-web, and SignalR (Use ASP.NET Core SignalR with Blazor).

A Blazor WebAssembly app built as a Progressive Web App (PWA) uses modern browser APIs to enable many of the capabilities of a native client app, such as working offline, running in its own app window, launching from the host's operating system, receiving push notifications, and automatically updating in the background.

The Blazor script handles:

- Downloading the .NET runtime, Razor components, and the component's dependencies.
- Initialization of the runtime.

The size of the published app, its *payload size*, is a critical performance factor for an app's usability. A large app takes a relatively long time to download to a browser, which diminishes the user experience. Blazor WebAssembly optimizes payload size to reduce download times:

- Unused code is stripped out of the app when it's published by the Intermediate Language (IL) Trimmer.
- HTTP responses are compressed.
- The .NET runtime and assemblies are cached in the browser.

The Blazor WebAssembly hosting model offers several benefits:

- For standalone Blazor WebAssembly apps, there's no .NET server-side dependency after the app is downloaded from the server, so the app remains functional if the server goes offline.
- Client resources and capabilities are fully leveraged.
- Work is offloaded from the server to the client.
- For standalone Blazor WebAssembly apps, an ASP.NET Core web server isn't required to host the app. Serverless deployment scenarios are possible, such as serving the app from a Content Delivery Network (CDN).

The Blazor WebAssembly hosting model has the following limitations:

- Razor components are restricted to the capabilities of the browser.
- Capable client hardware and software (for example, WebAssembly support) is required.
- Download size is larger, and components take longer to load.
- Code sent to the client can't be protected from inspection and tampering by users.

The .NET Intermediate Language (IL) interpreter includes partial just-in-time (JIT) runtime support to achieve improved runtime performance. The JIT interpreter optimizes execution of interpreter bytecodes by replacing them with tiny blobs of WebAssembly code. The JIT interpreter is automatically enabled for Blazor WebAssembly apps except when debugging.

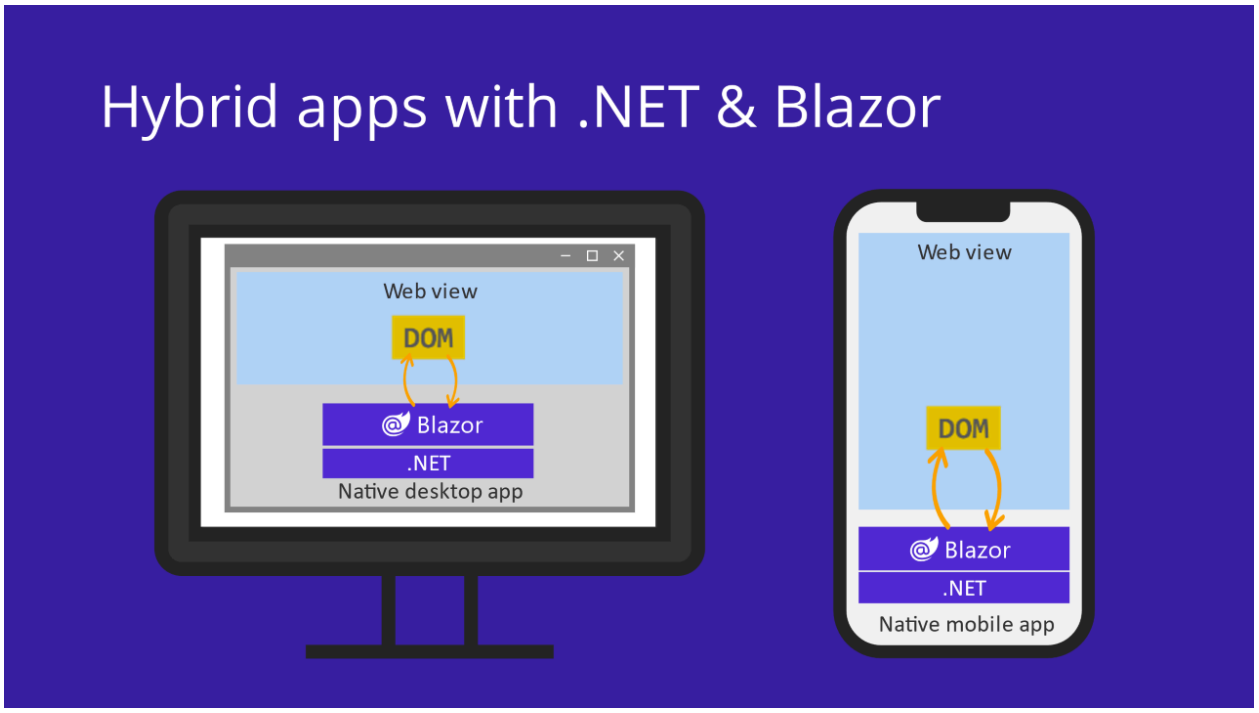
Blazor supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size. For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

The same .NET WebAssembly build tools used for AOT compilation also relink the .NET WebAssembly runtime to trim unused runtime code. Blazor also trims unused code from .NET framework libraries. The .NET compiler further precompresses a standalone Blazor WebAssembly app for a smaller app payload.

WebAssembly-rendered Razor components can use native dependencies built to run on WebAssembly.

Blazor Hybrid

Blazor can also be used to build native client apps using a hybrid approach. Hybrid apps are native apps that leverage web technologies for their functionality. In a Blazor Hybrid app, Razor components run directly in the native app (not on WebAssembly) along with any other .NET code and render web UI based on HTML and CSS to an embedded Web View control through a local interop channel.



Blazor Hybrid apps can be built using different .NET native app frameworks, including .NET MAUI, WPF, and Windows Forms. Blazor provides `BlazorWebView` controls for adding Razor components to apps built with these frameworks. Using Blazor with .NET MAUI offers a convenient way to build cross-platform Blazor Hybrid apps for mobile and desktop, while Blazor integration with WPF and Windows Forms can be a great way to modernize existing apps.

Because Blazor Hybrid apps are native apps, they can support functionality that isn't available with only the web platform. Blazor Hybrid apps have full access to native platform capabilities through normal .NET APIs. Blazor Hybrid apps can also share and reuse components with existing Blazor Server or Blazor WebAssembly apps. Blazor Hybrid apps combine the benefits of the web, native apps, and the .NET platform.

The Blazor Hybrid hosting model offers several benefits:

- Reuse existing components that can be shared across mobile, desktop, and web.
- Leverage web development skills, experience, and resources.
- Apps have full access to the native capabilities of the device.

The Blazor Hybrid hosting model has the following limitations:

- Separate native client apps must be built, deployed, and maintained for each target platform.
- Native client apps usually take longer to find, download, and install over accessing a web app in a browser.

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

For more information on Microsoft native client frameworks, see the following resources:

- .NET Multi-platform App UI (.NET MAUI)
- Windows Presentation Foundation (WPF)
- Windows Forms

Which Blazor hosting model should I choose?

A component's hosting model is set by its *render mode*, either at compile time or runtime, which is described with examples in ASP.NET Core Blazor render modes. The following table shows the primary considerations for setting the render mode to determine a component's hosting model. For standalone Blazor WebAssembly apps, all of the app's components are rendered on the client with the Blazor WebAssembly hosting model.

Blazor Hybrid apps include .NET MAUI, WPF, and Windows Forms framework apps.

Expand table

Feature	Blazor Server	Blazor WebAssembly (WASM)	Blazor Hybrid
Complete .NET API compatibility	✔	✘	✔
Direct access to server and network resources	✔	✘†	✘†
Small payload size with fast initial load time	✔	✘	✘
Near native execution speed	✔	✔‡	✔
App code secure and private on the server	✔	✘†	✘†
Run apps offline once downloaded	✘	✔	✔
Static site hosting	✘	✔	✘
Offloads processing to clients	✘	✔	✔
Full access to native client capabilities	✘	✘	✔
Web-based deployment	✔	✔	✘

†Blazor WebAssembly and Blazor Hybrid apps can use server-based APIs to access server/network resources and access private and secure app code.
‡Blazor WebAssembly only reaches near-native performance with ahead-of-time (AOT) compilation.

After you choose the app's hosting model, you can generate a Blazor Server or Blazor WebAssembly app from a Blazor project template. For more information, see Tooling for ASP.NET Core Blazor.

To create a Blazor Hybrid app, see the articles under ASP.NET Core Blazor Hybrid tutorials.

Complete .NET API compatibility

Components rendered for the Blazor Server hosting model and Blazor Hybrid apps have complete .NET API compatibility, while components rendered for Blazor WebAssembly are limited to a subset of .NET APIs. When an app's specification requires one or more .NET APIs that are unavailable to WebAssembly-rendered components, then choose to render components for Blazor Server or use Blazor Hybrid.

Direct access to server and network resources

Components rendered for the Blazor Server hosting model have direct access to server and network resources where the app is executing. Because components hosted using Blazor WebAssembly or Blazor Hybrid execute on a client, they don't have direct access to server and network resources. Components can access server and network resources *indirectly* via protected server-based APIs. Server-based APIs might be available via third-party libraries, packages, and services. Take into account the following considerations:

- Third-party libraries, packages, and services might be costly to implement and maintain, weakly supported, or introduce security risks.
- If one or more server-based APIs are developed internally by your organization, additional resources are required to build and maintain them.

Use the Blazor Server hosting model to avoid the need to expose APIs from the server environment.

Small payload size with fast initial load time

Rendering components from the server reduces the app payload size and improves initial load times. When a fast initial load time is desired, use the Blazor Server hosting

model or consider static server-side rendering.

Near native execution speed

Blazor Hybrid apps run using the .NET runtime natively on the target platform, which offers the best possible speed.

Components rendered for the Blazor WebAssembly hosting model, including Progressive Web Apps (PWAs), and standalone Blazor WebAssembly apps run using the .NET runtime for WebAssembly, which is slower than running directly on the platform. Consider using ahead-of-time (AOT) compiled to improve runtime performance when using Blazor WebAssembly.

App code secure and private on the server

Maintaining app code securely and privately on the server is a built-in feature of components rendered for the Blazor Server hosting model. Components rendered using the Blazor WebAssembly or Blazor Hybrid hosting models can use server-based APIs to access functionality that must be kept private and secure. The considerations for developing and maintaining server-based APIs described in the Direct access to server and network resources section apply. If the development and maintenance of server-based APIs isn't desirable for maintaining secure and private app code, render components for the Blazor Server hosting model.

Run apps offline once downloaded

Standalone Blazor WebAssembly apps built as Progressive Web Apps (PWAs) and Blazor Hybrid apps can run offline, which is particularly useful when clients aren't able to connect to the Internet. Components rendered for the Blazor Server hosting model fail to run when the connection to the server is lost. If an app must run offline, standalone Blazor WebAssembly and Blazor Hybrid are the best choices.

Static site hosting

Static site hosting is possible with standalone Blazor WebAssembly apps because they're downloaded to clients as a set of static files. Standalone Blazor WebAssembly apps don't require a server to execute server-side code in order to download and run and can be delivered via a Content Delivery Network (CDN) (for example, Azure CDN).

Although Blazor Hybrid apps are compiled into one or more self-contained deployment assets, the assets are usually provided to clients through a third-party app store. If static

hosting is an app requirement, select standalone Blazor WebAssembly.

Offloads processing to clients

Blazor WebAssembly and Blazor Hybrid apps execute on clients and thus offload processing to clients. Blazor Server apps execute on a server, so server resource demand typically increases with the number of users and the amount of processing required per user. When it's possible to offload most or all of an app's processing to clients and the app processes a significant amount of data, Blazor WebAssembly or Blazor Hybrid is the best choice.

Full access to native client capabilities

Blazor Hybrid apps have full access to native client API capabilities via .NET native app frameworks. In Blazor Hybrid apps, Razor components run directly in the native app, not on WebAssembly. When full client capabilities are a requirement, Blazor Hybrid is the best choice.

Web-based deployment

Blazor web apps are updated on the next app refresh from the browser.

Blazor Hybrid apps are native client apps that typically require an installer and platform-specific deployment mechanism.

Setting a component's hosting model

To set a component's hosting model to Blazor Server or Blazor WebAssembly at compile-time or dynamically at runtime, you set its *render mode*. Render modes are fully explained and demonstrated in the ASP.NET Core Blazor render modes article. We don't recommend that you jump from this article directly to the *Render modes* article without reading the content in the articles between these two articles. For example, render modes are more easily understood by looking at Razor component examples, but basic Razor component structure and function isn't covered until the ASP.NET Core Blazor fundamentals article is reached. It's also helpful to learn about Blazor's project templates and tooling before working with the component examples in the *Render modes* article.

ASP.NET Core Blazor tutorials

Article • 11/18/2024

> **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**.

The following tutorials provide basic working experiences for building Blazor apps.

For an overview of Blazor, see ASP.NET Core Blazor.

- Build your first Blazor app
- Build a Blazor todo list app (Blazor Web App)
- Build a Blazor movie database app (Overview) (Blazor Web App)
- Use ASP.NET Core SignalR with Blazor (Blazor Web App)
- ASP.NET Core Blazor Hybrid tutorials
- Microsoft Learn
  - Blazor Learning Path
  - Blazor Learn Modules

Build a Blazor todo list app

Article • 10/29/2024

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

This tutorial provides a basic working experience for building and modifying a Blazor app. For detailed Blazor guidance, see the Blazor reference documentation.

Learn how to:

- ✓ Create a todo list Blazor app project
- ✓ Modify Razor components
- ✓ Use event handling and data binding in components
- ✓ Use routing in a Blazor app

At the end of this tutorial, you'll have a working todo list app.

Prerequisites

Download and install .NET if it isn't already installed on the system or if the system doesn't have the latest version installed.

Create a Blazor app

Create a new Blazor Web App named `TodoList` in a command shell:

.NET CLI

```
dotnet new blazor -o TodoList
```

The `-o|--output` option creates a folder for the project. If you've created a folder for the project and the command shell is open in that folder, omit the `-o|--output` option and value to create the project.

The preceding command creates a folder named `TodoList` with the `-o|--output` option to hold the app. The `TodoList` folder is the *root folder* of the project. Change directories to the `TodoList` folder with the following command:

.NET CLI

```
cd TodoList
```

Build a todo list Blazor app

Add a new `Todo` Razor component to the app using the following command:

.NET CLI

```
dotnet new razorcomponent -n Todo -o Components/Pages
```

The `-n|--name` option in the preceding command specifies the name of the new Razor component. The new component is created in the project's `Components/Pages` folder with the `-o|--output` option.

> **ⓘ Important**
>
> Razor component file names require a capitalized first letter. Open the `Pages` folder and confirm that the `Todo` component file name starts with a capital letter `T`. The file name should be `Todo.razor`.

Open the `Todo` component in any file editor and make the following changes at the top of the file:

- Add an `@page` Razor directive with a relative URL of `/todo`.
- Enable interactivity on the page so that it isn't just statically rendered. The Interactive Server render mode enables the component to handle UI events from the server.
- Add a page title with the `PageTitle` component, which enables adding an HTML `<title>` element to the page.

`Todo.razor`:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

@code {

}
```

Save the `Todo.razor` file.

Add the `Todo` component to the navigation bar.

The `NavMenu` component is used in the app's layout. Layouts are components that allow you to avoid duplication of content in an app. The `NavLink` component provides a cue in the app's UI when the component URL is loaded by the app.

In the navigation element (`<nav>`) content of the `NavMenu` component, add the following `<div>` element for the `Todo` component.

In `Components/Layout/NavMenu.razor`:

razor

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

Save the `NavMenu.razor` file.

Build and run the app by executing the dotnet watch run command in the command shell from the `TodoList` folder. After the app is running, visit the new Todo page by selecting the **`Todo`** link in the app's navigation bar, which loads the page at `/todo`.

Leave the app running the command shell. Each time a file is saved, the app is automatically rebuilt, and the page in the browser is automatically reloaded.

Add a `TodoItem.cs` file to the root of the project (the `TodoList` folder) to hold a class that represents a todo item. Use the following C# code for the `TodoItem` class.

`TodoItem.cs`:

C#

```
public class TodoItem
{
    public string? Title { get; set; }
    public bool IsDone { get; set; }
}
```

> **Note**
>
> If using Visual Studio to create the `TodoItem.cs` file and `TodoItem` class, use ***either*** of the following approaches:
>
> - Remove the namespace that Visual Studio generates for the class.
> - Use the **Copy** button in the preceding code block and replace the entire contents of the file that Visual Studio generates.

Return to the `Todo` component and perform the following tasks:

- Add a field for the todo items in the `@code` block. The `Todo` component uses this field to maintain the state of the todo list.
- Add unordered list markup and a `foreach` loop to render each todo item as a list item (`<li>`).

`Components/Pages/Todo.razor`:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

@code {
    private List<TodoItem> todos = new();
}
```

The app requires UI elements for adding todo items to the list. Add a text input (`<input>`) and a button (`<button>`) below the unordered list (`<ul>...</ul>`):

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" />
<button>Add todo</button>

@code {
    private List<TodoItem> todos = new();
}
```

Save the `TodoItem.cs` file and the updated `Todo.razor` file. In the command shell, the app is automatically rebuilt when the files are saved. The browser reloads the page.

When the **`Add todo`** button is selected, nothing happens because an event handler isn't attached to the button.

Add an `AddTodo` method to the `Todo` component and register the method for the button using the `@onclick` attribute. The `AddTodo` C# method is called when the button is selected:

razor

```
<input placeholder="Something todo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private List<TodoItem> todos = new();

    private void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

To get the title of the new todo item, add a `newTodo` string field at the top of the `@code` block:

C#

```
private string? newTodo;
```

Modify the text `<input>` element to bind `newTodo` with the `@bind` attribute:

razor

```
<input placeholder="Something todo" @bind="newTodo" />
```

Update the `AddTodo` method to add the `TodoItem` with the specified title to the list. Clear the value of the text input by setting `newTodo` to an empty string:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private List<TodoItem> todos = new();
    private string? newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

Save the `Todo.razor` file. The app is automatically rebuilt in the command shell, and the page reloads in the browser.

The title text for each todo item can be made editable, and a checkbox can help the user keep track of completed items. Add a checkbox input for each todo item and bind its value to the `IsDone` property. Change `@todo.Title` to an `<input>` element bound to `todo.Title` with `@bind`:

razor

```razor
<ul>
      @foreach (var todo in todos)
      {
         <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
         </li>
      }
</ul>
```

Update the `<h3>` header to show a count of the number of todo items that aren't complete (`IsDone` is `false`). The Razor expression in the following header evaluates each time Blazor rerenders the component.

razor

```razor
<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>
```

The completed `Todo` component:

razor

```razor
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h1>Todo (@todos.Count(todo => !todo.IsDone))</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>
```

```
<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private List<TodoItem> todos = new();
    private string? newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

Save the `Todo.razor` file. The app is automatically rebuilt in the command shell, and the page reloads in the browser.

Add items, edit items, and mark todo items done to test the component.

When finished, shut down the app in the command shell. Many command shells accept the keyboard command `Ctrl`+`C` (Windows) or `⌘`+`C` (macOS) to stop an app.

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Next steps

In this tutorial, you learned how to:

- ✓ Create a todo list Blazor app project
- ✓ Modify Razor components
- ✓ Use event handling and data binding in components
- ✓ Use routing in a Blazor app

Learn about tooling for ASP.NET Core Blazor:

ASP.NET Core Blazor

Build a Blazor movie database app (Overview)

Article • 11/15/2024

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

This tutorial explains the basics of building a Blazor Web App with a database, Entity Framework (EF) Core, and user interactivity.

Parts of this series include:

1. Create a Blazor Web App
2. Add and scaffold a model
3. Learn about Razor components
4. Work with a database
5. Add validation
6. Add search
7. Add a new field
8. Add interactivity

At the end of the tutorial, you'll have a Blazor Web App that can display and manage movies in a movie database.

Secure authentication flow required for production apps

This tutorial uses a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor Web Apps, see the following resources:

- ASP.NET Core Blazor authentication and authorization

- ASP.NET Core Blazor authentication and authorization and the following articles in the *Server* security node
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
- Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Sample app

If you don't intend to create the demonstration app while reading the article, you can refer to the completed sample app in the Blazor samples GitHub repository (dotnet/blazor-samples). Select the latest version folder in the repository. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Article code examples

The line breaks of code examples shown in the ASP.NET Core documentation often don't match line breaks in scaffolded code generated by tooling for an app. This is due to an article publishing limitation. Lines of code in articles are generally limited to 85 characters in length, and we manually adjust the line length using line breaks to satisfy our publishing guidelines.

As you work through this tutorial or use any other ASP.NET Core article's code examples, you never need to adjust scaffolded code in your app to match the line breaks displayed in article code examples.

Report a tutorial issue

To open a documentation GitHub issue for an article of the series, use the **Open a documentation issue** link at the bottom of the article. Using the link to create your issue adds important tracking metadata to the issue and automatically pings the author of the article.

Support requests

We welcome feedback on the tutorial's articles, such as bug reports and comments on the article's text, but we're often unable to provide product support. If you run into a problem while following the tutorial, don't immediately open a documentation issue. Check the steps that you've taken against the article and compare your code to the sample app before opening an issue because many problems can be traced to missing a step or not following a step correctly.

For general questions about .NET and Blazor beyond the tutorial and reference documentation or to obtain assistance from the .NET community, converse with developers in public forums.

Next steps

Next: Create a Blazor Web App

Build a Blazor movie database app (Part 1 - Create a Blazor Web App)

Article • 11/15/2024

> **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 is the first part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers how to create a Blazor Web App that adopts static server-side rendering (static SSR). Static SSR means that content is rendered on the server and sent to the client for display in response to individual requests.

Prerequisites

Visual Studio (latest release) with the **ASP.NET and web development** workload

Create a Blazor Web App

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorWebAppMovies` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.

- Confirm that the **Location** for the app is suitable. Set the **Place solution and project in the same directory** checkbox to match your preferred solution file location. Select the **Next** button.

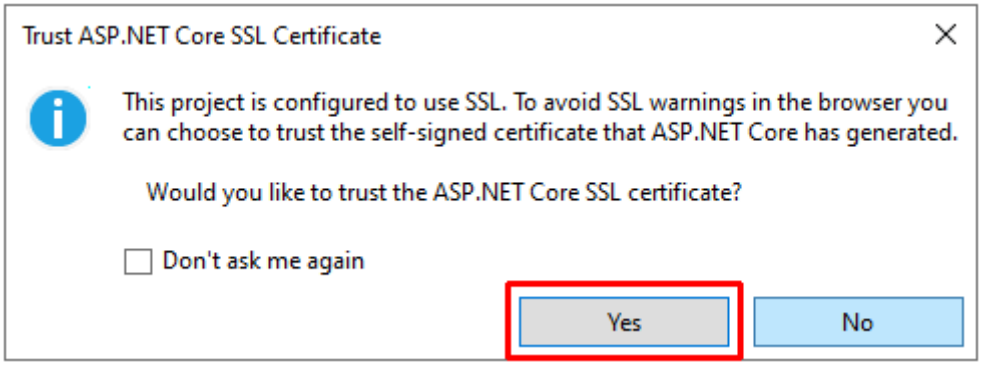
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Select **.NET 9.0 (Standard Term Support)**.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **Server**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected
  - Select **Create**.

The Visual Studio instructions in parts of this tutorial series use EF Core commands to add database migrations and update the database. EF Core commands are issued using Visual Studio Connected Services. More information is provided later in this tutorial series.

Run the app

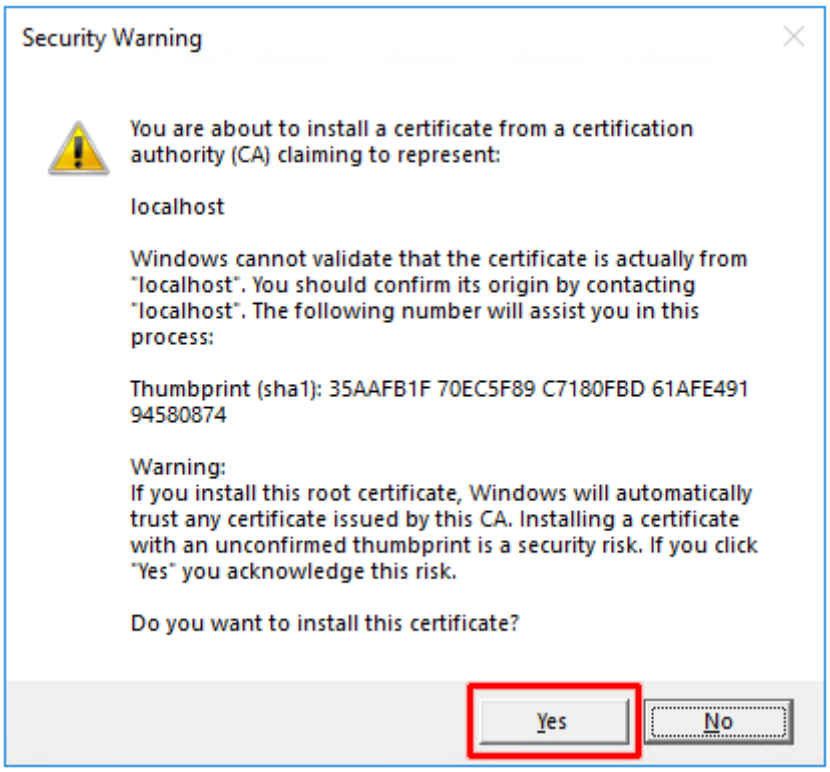
Press F5 on the keyboard to run the app.

Visual Studio displays the following dialog when a project isn't configured to use SSL:



Select **Yes** if you trust the ASP.NET Core SSL certificate.

The following dialog is displayed:



Select **Yes** to acknowledge the risk and install the certificate.

Visual Studio:

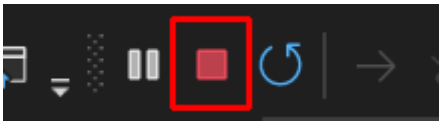
- Compiles and runs the app.
- Launches the default browser at `https://localhost:{PORT}`, which displays the app's UI. The `{PORT}` placeholder is the random port assigned to the app when the app is created. If you need to change the port due to a local port conflict, change the port in the project's `Properties/launchSettings.json` file.

Navigate the pages of the app to confirm that the app is working normally.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

    
  - Press `Shift`+`F5` on the keyboard.

Examine the project files

The following sections contain an overview of the project's folders and files.

If you're building the app, you don't need to make changes to the project files in the following sections. As you read the descriptions of the folders and files, examine them in the project.

If you're only reading the articles and not building the app, you can refer to the completed sample app in the Blazor samples GitHub repository (dotnet/blazor-samples). Select the latest version folder in the repository. The sample folder for this tutorial's project is named `BlazorWebAppMovies`. The sample app is the *finished version* of the app after following all of the steps of the tutorial series. Code in the sample doesn't always match steps of the tutorial before the end of the series.

`Properties` folder

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

`wwwroot` folder

The `wwwroot` folder contains static assets, such as image, JavaScript (`.js`), and stylesheet (`.css`) files.

`Components`, `Components/Pages`, and `Components/Layout` folders

These folders contain *Razor components*, often referred to as "components," and supporting files. A component is a self-contained portion of user interface (UI) with optional processing logic. Components can be nested, reused, and shared among projects.

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

Typically, components that are nested within other components and not directly reachable ("routable") at a URL are placed in the `Components` folder. Components that are routable via a URL are usually placed in the `Components/Pages` folder.

The `Components/Layout` folder contains the following layout components and stylesheets:

- `MainLayout` component (`MainLayout.razor`): The app's main layout component.
- `MainLayout.razor.css`: Stylesheet for the app's main layout.
- `NavMenu` component (`NavMenu.razor`): Implements sidebar navigation. This component uses several `NavLink` components to render navigation links to other Razor components.
- `NavMenu.razor.css`: Stylesheet for the app's navigation menu.

`Components/_Imports.razor` file

The `_Imports` file (`_Imports.razor`) includes common *Razor directives* to include in the app's Razor components. Razor directives are reserved keywords prefixed with `@` that appear in Razor markup and change the way component markup or component elements are compiled or function.

`Components/App.razor` file

The `App` component (`App.razor`) is the root component of the app that includes:

- HTML markup.
- The `Routes` component.
- The Blazor script (`<script>` tag for `blazor.web.js`).

The root component is the first component that the app loads.

`Components/Routes.razor` file

The `Routes` component (`Routes.razor`) sets up routing for the app.

`appsettings.json` file

The `appsettings.json` file contains configuration data, such as connection strings.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production

environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

`Program.cs` file

The `Program.cs` file contains code to create the app and configure the request processing pipeline of the app.

The order of the lines in the Blazor Web App project template changes across releases of .NET, so the order of the lines in the `Program.cs` file might not match the order of the lines covered in this section.

A WebApplicationBuilder creates the app with preconfigured defaults:

C#

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

Razor component services are added to the app by calling AddRazorComponents, which enables Razor components to render and execute code on the server, and AddInteractiveServerComponents adds services to support rendering Interactive Server components:

C#

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

The WebApplication (held by the `app` variable in the following code) is built:

C#

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

Next, the HTTP request pipeline is configured.

In the development environment:

- Exception Handler Middleware (UseExceptionHandler) processes errors and displays a developer exception page during development app runs.
- HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) processes HSTS.

```
C#

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}
```

HTTPS Redirection Middleware (UseHttpsRedirection) enforces the HTTPS protocol by redirecting HTTP requests to HTTPS if an HTTPS port is available:

```
C#

app.UseHttpsRedirection();
```

Antiforgery Middleware (UseAntiforgery) enforces antiforgery protection for form processing:

```
C#

app.UseAntiforgery();
```

Map Static Assets routing endpoint conventions (MapStaticAssets) maps static files, such as images, scripts, and stylesheets, produced during the build as endpoints:

```
C#

app.MapStaticAssets();
```

MapRazorComponents maps components defined in the root `App` component to the given .NET assembly and renders routable components, and AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) support for the app:

```
C#

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

> **Note**
>
> The extension methods AddInteractiveServerComponents on AddRazorComponents and AddInteractiveServerRenderMode on MapRazorComponents make the app capable of adopting interactive SSR, which isn't relevant until the last part of the tutorial series on interactivity. Over the next several articles, the app's components only adopt static SSR.

The app is run by calling Run on the WebApplication (`app`):

```
C#

app.Run();
```

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

When using VS Code or the .NET CLI, this tutorial series adopts insecure HTTP protocol to ease the transition of adopting SSL/HTTPS security for Linux and macOS users. For information on adopting SSL/HTTPS, see Enforce HTTPS in ASP.NET Core.

Next steps

Next: Add and scaffold a model

Build a Blazor movie database app (Part 2 - Add and scaffold a model)

Article • 10/08/2024

This article is the second part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

In this part of the tutorial series:

- A class is added to the app representing a movie in a database.
- Entity Framework Core (EF Core) services and tooling create a database context and database. EF Core is an object-relational mapper (O/RM) that simplifies data access. You write model classes first, and EF Core creates the database from the model classes, which is called *scaffolding*.
- Additional tooling scaffolds a Razor component-based UI for interacting with the movie records in the database via the database context.

Add a data model

In **Solution Explorer**, right-click the `BlazorWebAppMovies` project and select **Add** > **New Folder**. Name the folder `Models`.

Right-click the `Models` folder. Select **Add** > **Class**. Name the file `Movie.cs`. Use the following contents for the file.

`Models/Movie.cs`:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace BlazorWebAppMovies.Models;

public class Movie
{
    public int Id { get; set; }

    public string? Title { get; set; }

    public DateOnly ReleaseDate { get; set; }

    public string? Genre { get; set; }
```

```
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

The `Movie` class contains:

- A record identifier property (`Id`), which is required by EF Core and the database to track records. In the database, the `Id` property is the database record's primary key.
- Other properties that describe aspects of a movie:
  - Title (`Title`)
  - Release date (`ReleaseDate`)
  - Genre (`Genre`)
  - Price (`Price`)

The question mark on a `string` type indicates that the property is nullable (it can hold a `null` value).

The EF Core database provider selects data types based on the .NET types of the model's properties. The provider also takes into account other metadata provided by System.ComponentModel.DataAnnotations, which are a set of attribute classes placed above a model's property with the following format, where the `{ANNOTATION}` placeholder is the annotation name. You can place multiple annotations on the same line separated by commas inside the brackets, or you can place multiple annotations on separate lines, which is the approach adopted by this tutorial series.

C#

```
[{ANNOTATION}]
```

The `Price` property in the `Movie` class file has two data annotations:

C#

```
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

These annotations specify:

- That the property holds a currency data type.

- The database column is a decimal of 18 digits with two decimal places.

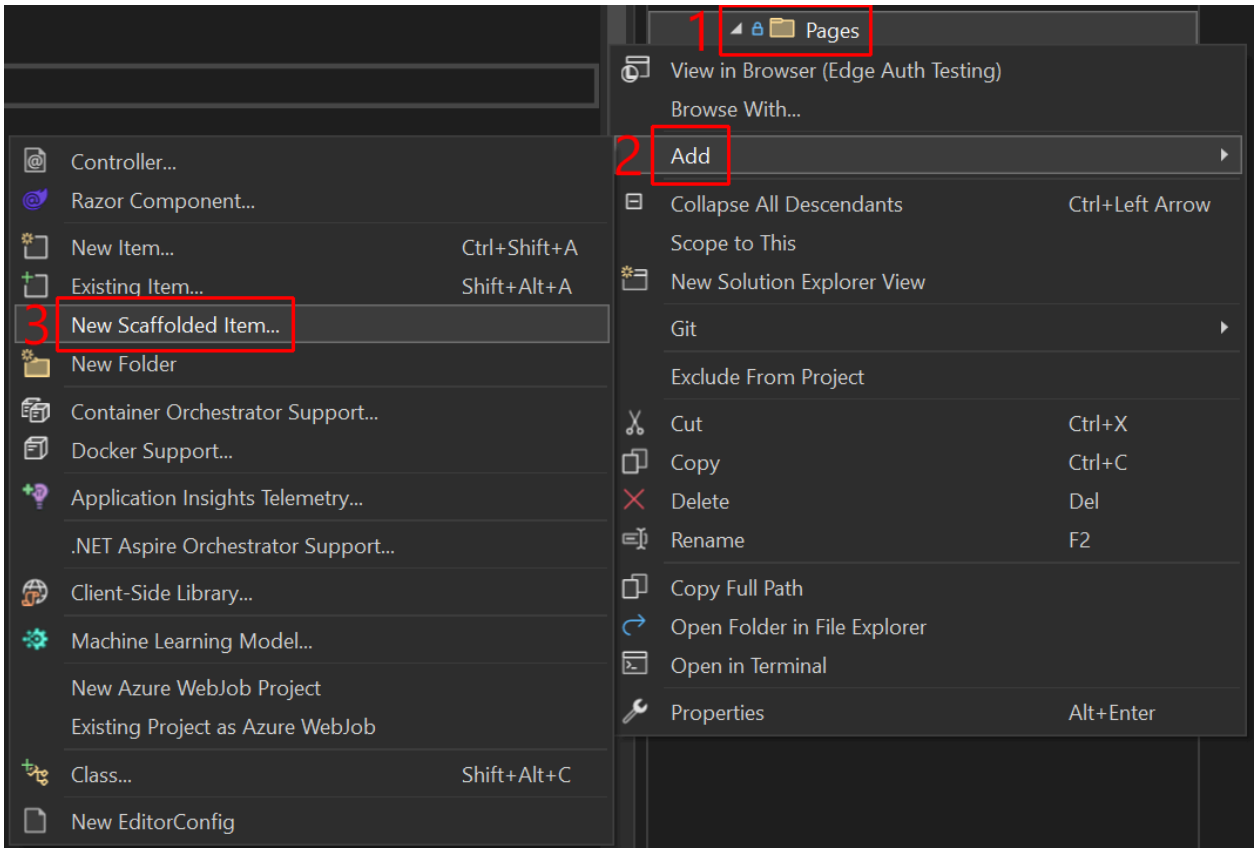
More information on data annotations, including adding data annotations for validation, is covered in a later part of the tutorial series.

Select **Build** > **Build Solution** from the menu bar or press F6 on the keyboard. Confirm in the **Output** panel that the build succeeded.

Scaffold the model

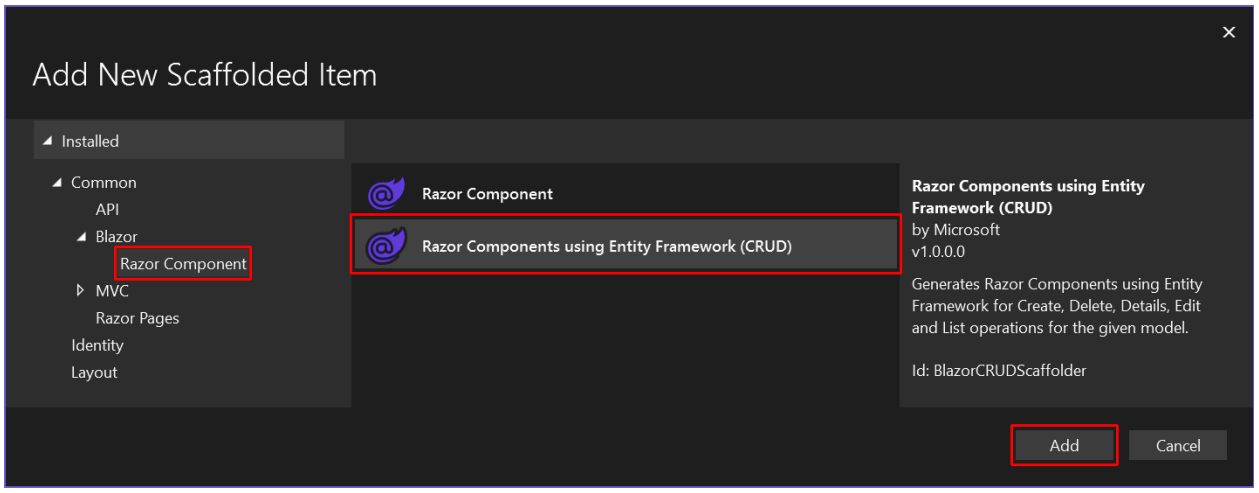
In this section, the `Movie` model is used to *scaffold* a database context and a UI for managing movies in the database. .NET scaffolding is a code generation framework for .NET applications. You use scaffolding to quickly add database and UI code that interacts with data models.

Right-click on the `Components/Pages` folder and select **Add** > **New Scaffolded Item**:



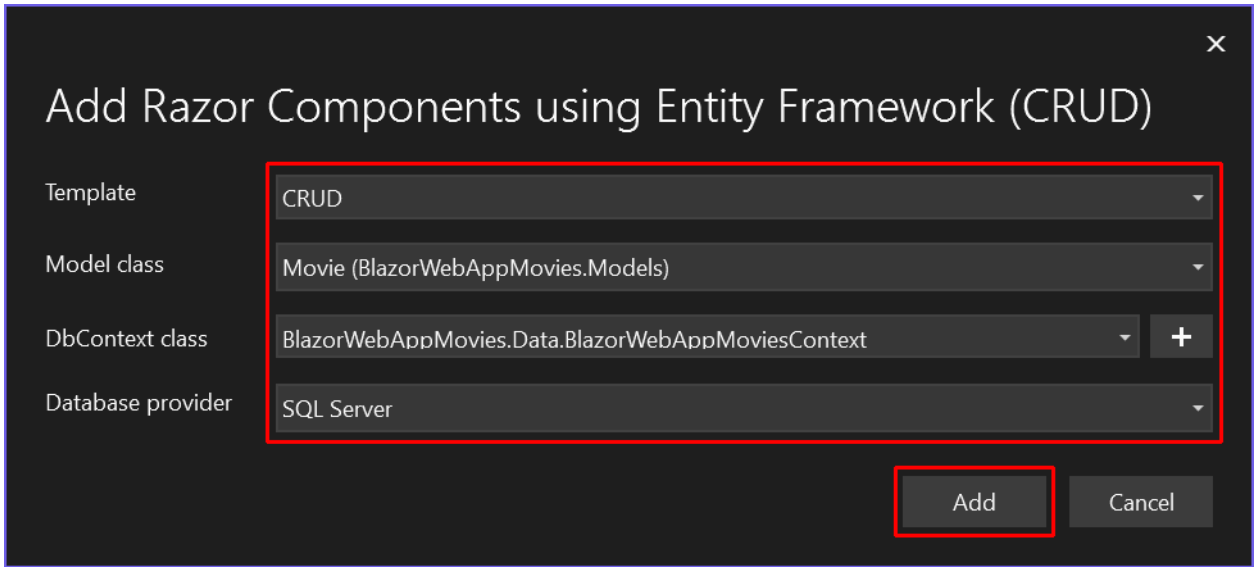
With the **Add New Scaffold Item** dialog open to **Installed** > **Common** > **Blazor** > **Razor Component**, select **Razor Components using Entity Framework (CRUD)**. Select the **Add** button.

CRUD is an acronym for Create, Read, Update, and Delete. The scaffolder produces create, edit, delete, details, and index components for the app.



Complete the **Add Razor Components using Entity Framework (CRUD)** dialog:

- The **Template** dropdown list includes other templates for specifically creating create, edit, delete, details, and list components. This dropdown list comes in handy when you only need to create a specific type of component scaffolded to a model class. Leave the **Template** dropdown list set to **CRUD** to scaffold a full set of components.
- In the **Model class** dropdown list, select **Movie (BlazorWebAppMovies.Models)**.
- For **DbContext class**, select the + (plus sign) button.
- In the **Add Data Context** modal dialog, the class name `BlazorWebAppMovies.Data.BlazorWebAppMoviesContext` is generated. Use the default generated value. Select the **Add** button.
- After the model dialog closes, the **Database provider** dropdown list defaults to **SQL Server**. When you're building apps in the future, you can select the appropriate provider for the database that you're using. The options include SQLite, PostgreSQL, and Azure Cosmos DB. Leave the **Database provider** dropdown list set to **SQL Server**.
- Select **Add**.



The `appsettings.json` file is updated with the connection string used to connect to a local database. In the following example, the `{CONNECTION STRING}` is the connection string automatically generated by the scaffolder:

JSON

```
"ConnectionStrings": {
  "BlazorWebAppMoviesContext": "{CONNECTION STRING}"
}
```

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Files created and updated by scaffolding

The scaffolding process creates the following component files and movie database context class file:

- `Components/Pages/MoviePages`
  - `Create.razor`: Creates new movie entities.
  - `Delete.razor`: Deletes a movie entity.
  - `Details.razor`: Shows movie entity details.
  - `Edit.razor`: Updates a movie entity.
  - `Index.razor`: Lists movie entities (records) in the database.
- `Data/BlazorWebAppMoviesContext.cs`: Database context file (DbContext).

The component files in the `MoviePages` folder are described in greater detail in the next part of this tutorial. The database context is described later in this article.

ASP.NET Core is built with dependency injection, which is a software design pattern for achieving Inversion of Control (IoC) between classes and their dependencies. Services, such as the EF Core database context, are registered with dependency injection during application startup. These services are injected into Razor components for use by the components.

The QuickGrid component is a Razor component for efficiently displaying data in tabular form. The scaffolder places a `QuickGrid` component in the `Index` component (`Components/Pages/Index.razor`) to display movie entities. Calling AddQuickGridEntityFrameworkAdapter on the service collection adds an EF Core adapter for `QuickGrid` to recognize EF Core-supplied IQueryable<T> instances and to resolve database queries asynchronously for efficiency.

In combination with UseDeveloperExceptionPage, AddDatabaseDeveloperPageExceptionFilter captures database-related exceptions that can be resolved by using Entity Framework migrations. When these exceptions occur, an HTML response is generated with details about possible actions to resolve the issue.

The following code is added to the `Program` file by the scaffolder:

C#

```
builder.Services.AddDbContextFactory<BlazorWebAppMoviesContext>(options =>
    options.UseSqlServer(

builder.Configuration.GetConnectionString("BlazorWebAppMoviesContext") ??
        throw new InvalidOperationException(
            "Connection string 'BlazorWebAppMoviesContext' not found.")));

builder.Services.AddQuickGridEntityFrameworkAdapter();

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

Static server-side rendering (static SSR) is enabled by calling:

- The service collection extension method AddRazorComponents to register services for server-side rendering (SSR) of Razor components.
- The request pipeline extension method MapRazorComponents, which maps the page components defined in the `App` component to the given assembly and renders the `App` component when a request matches a route to a component:

C#

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

```
...

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

The extension methods AddInteractiveServerComponents and AddInteractiveServerRenderMode make the app capable of adopting interactive SSR, which isn't relevant until the last part of the tutorial series on interactivity. Over the next several articles, the app's components only adopt static SSR.

Create the initial database schema using EF Core's migration feature

The migrations feature in EF Core:

- Creates the initial database schema.
- Incrementally updates the database schema to keep it synchronized with the app's data model. Existing data in the database is preserved.

EF Core adopts the *code-first* approach for database design and maintenance:

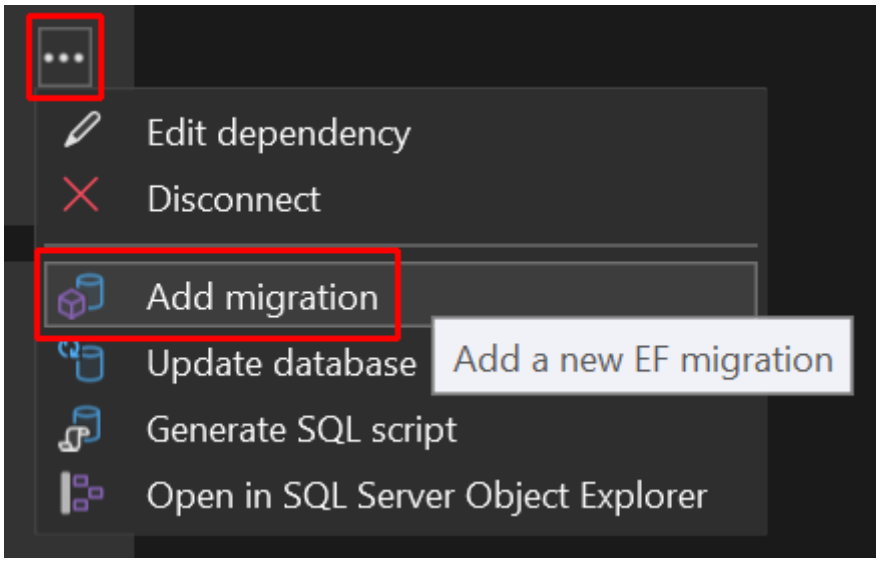
- Entity classes are created and updated first in the app.
- The database is created and updated from the app's entity classes.

This is the reverse procedure of *database-first* approaches, where the database is designed, built, and updated first. Adopting EF Core's code-first approach speeds up the process of app development because most of the difficult and time-consuming database creation and management procedures are handled transparently by the EF Core tooling, so you can focus on app development.

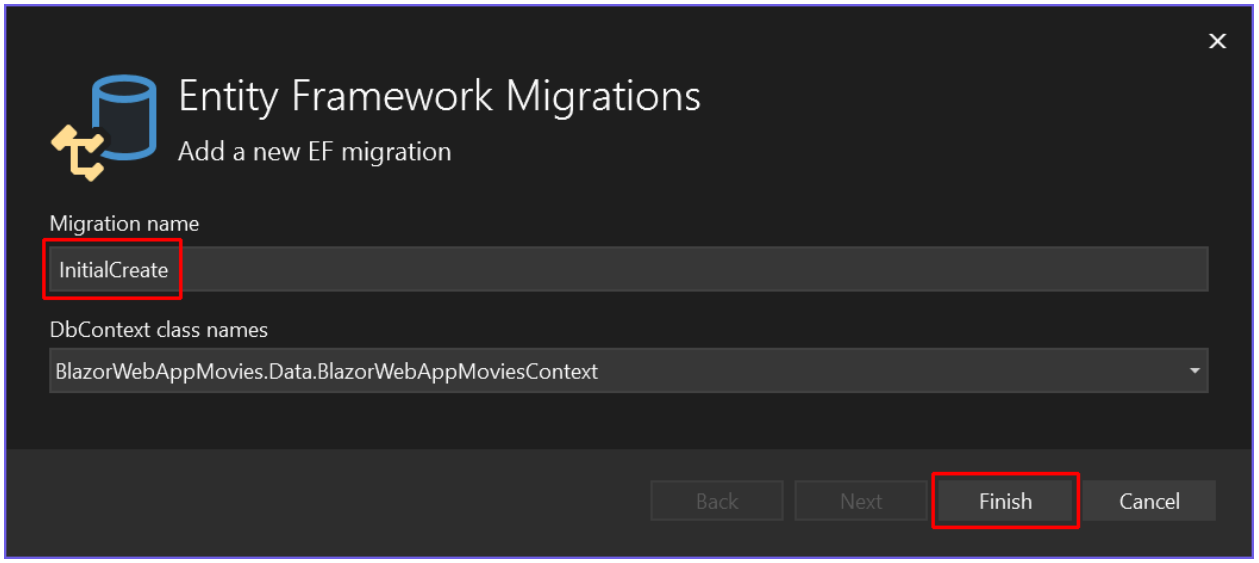
In this section, Visual Studio Connected Services are used to issue EF Core commands that:

- Add an initial migration.
- Update the database using the initial migration.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**:



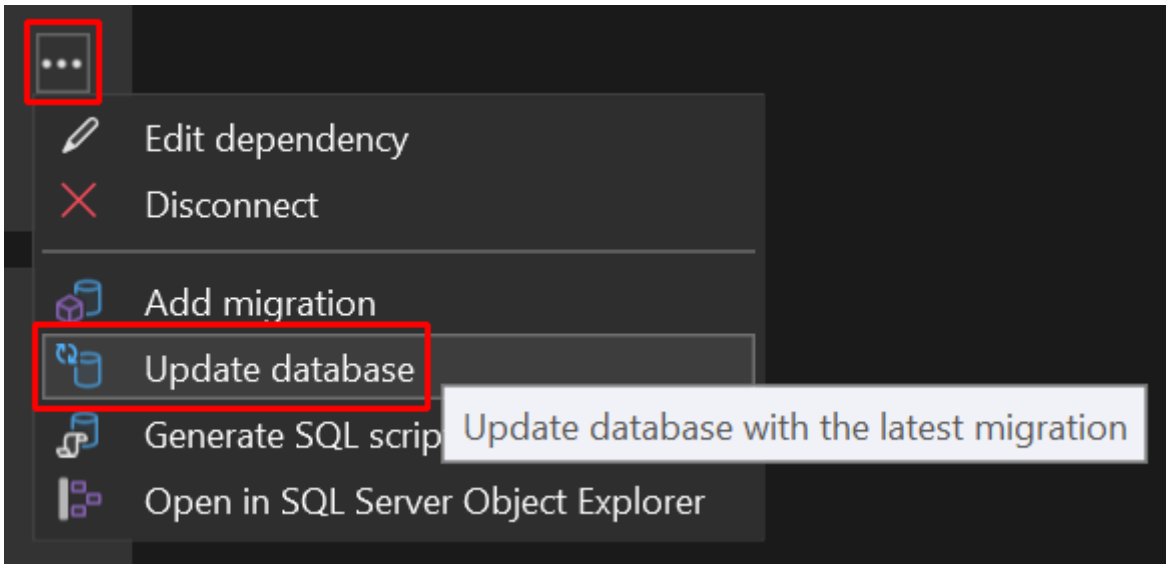
Give the migration a **Migration name** of `InitialCreate`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration:



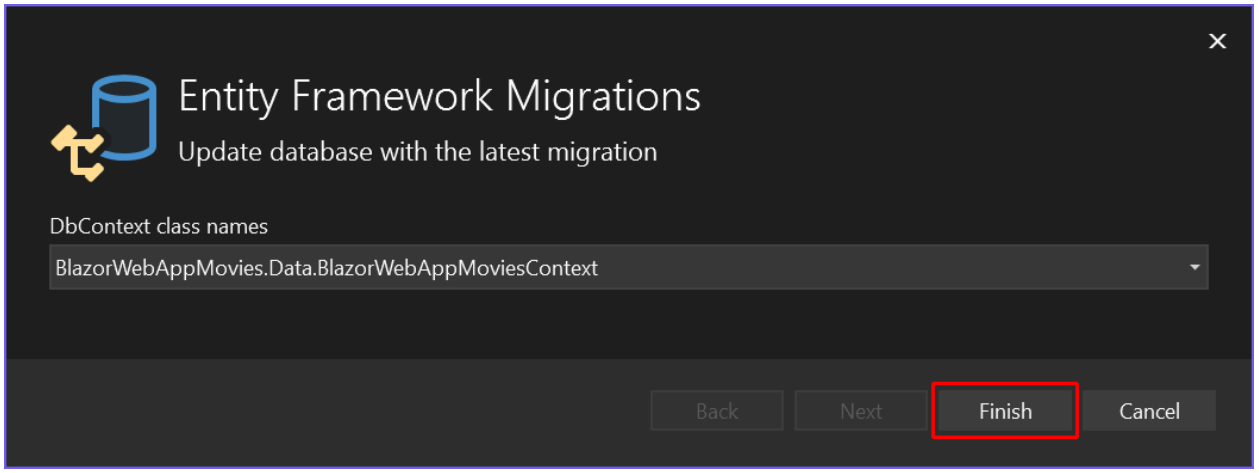
Select the **Close** button after the operation finishes.

Adding a migration generates code to create the initial database schema. The schema is based on the model specified in DbContext.

Select the ellipsis (`...`) again followed by the **Update database** command:



The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button:



The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_InitialCreate.cs` file, which creates the database. The `{TIME STAMP}` placeholder is a time stamp.

Select the **Close** button after the operation finishes.

The database context `BlazorWebAppMoviesContext` (`Data/BlazorWebAppMoviesContext.cs`):

- Derives from Microsoft.EntityFrameworkCore.DbContext.
- Specifies which entities are included in the data model.
- Coordinates EF Core functionality, such as CRUD operations, for the `Movie` model.
- Contains a DbSet<TEntity> property for the `Movie` entity set. In EF terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the connection string is read from the `appsettings.json` file.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

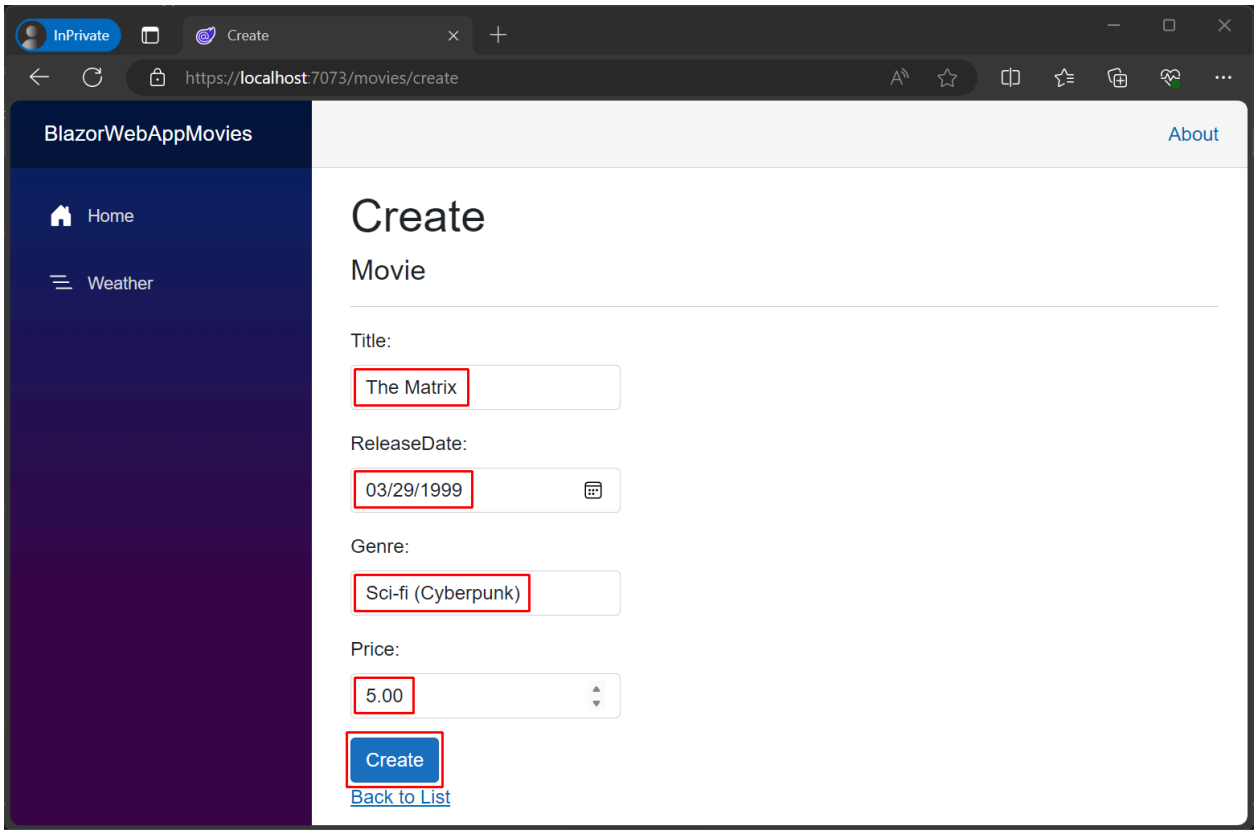
Test the app

Run the app.

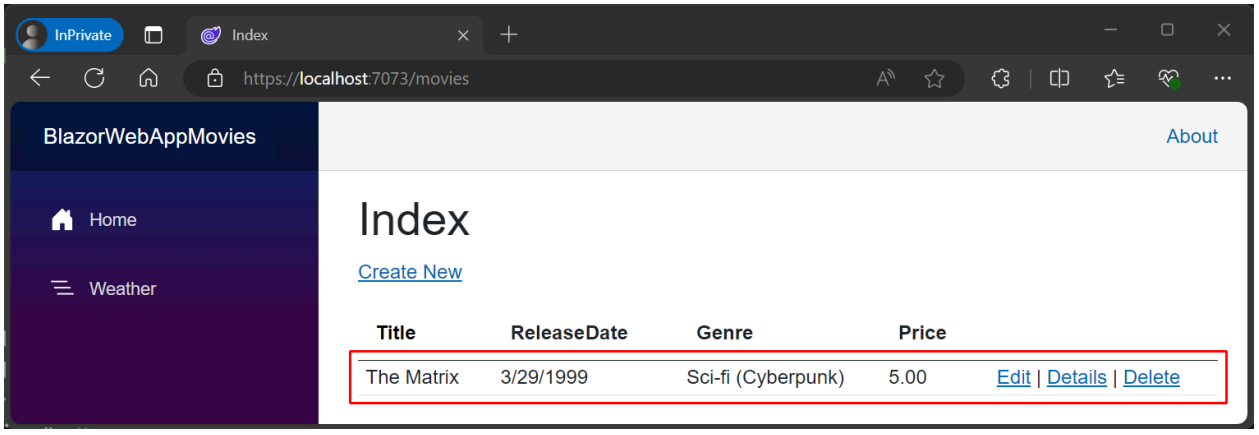
Add `/movies` to the URL in the browser's address bar to navigate to the movies `Index` page.

After the `Index` page loads, select the **Create New** link.

Add a movie to the database. In the following example, the classic sci-fi movie *The Matrix* (©1999 Warner Bros. Entertainment Inc.) is added as the first movie entry. Selecting the **Create** button adds the movie to the database:



When you select the **Create** button, the movie data is posted to the server and saved in the database. When the app returns to the `Index` page, the added entity appears:



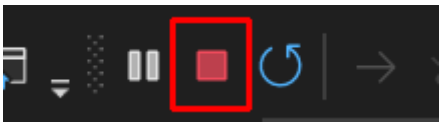
Open the `Edit` page. Edit the movie's record and save the changes.

Examine the `Delete` page, but don't delete *The Matrix* movie from the database. The presence of this movie record is valuable in the next step of the tutorial where rendered HTML is studied and some enhancements are made to the data displayed. If you already deleted the movie, add the movie to the database again using the `Create` page before proceeding to the next part of the tutorial series.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

  - Press `Shift`+`F5` on the keyboard.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

EF Core documentation:

- Entity Framework Core
- Entity Framework Core tools reference - .NET Core CLI
- Data Types
- SaveChangesAsync: The API document includes basic information on how entities are saved and change detection.
- Environment-based Startup class and methods
- dotnet aspnet-codegenerator
- Dependency injection in ASP.NET Core
- ASP.NET Core Blazor QuickGrid component

Legal

The Matrix is a copyright of Warner Bros. Entertainment Inc..

Next steps

Previous: Create a Blazor Web App
Next: Learn about Razor components

Build a Blazor movie database app (Part 3 - Learn about Razor components)

Article • 09/12/2024

This article is the third part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series examines the Razor components in the project that were scaffolded into the app. Improvements are made for the display of movie data.

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 (`.razor` file extension) for client-side UI logic and composition. Razor is a syntax for combining HTML markup with C# code designed for developer productivity.

Although developers and online resources use the term "Blazor components," the documentation uses the formal name "Razor components" (or just "components").

The anatomy of a Razor component has the following general pattern:

- At the top of the component definition (`.razor` file), various Razor directives specify how the component markup is compiled or functions.
- Next, Razor markup specifies how HTML is rendered, which includes ordinary HTML elements.
- Finally, an `@code` block contains C# code to define members for the component class, including component parameters and event handlers.

Consider the following `Welcome` component (`Welcome.razor`):

razor

```
@page "/welcome"

<PageTitle>Welcome!</PageTitle>

<h1>Welcome to Blazor!</h1>
```

```
<p>@welcomeMessage</p>

@code {
    private string welcomeMessage = "We ❤ Blazor!";
}
```

The first line represents an important Razor construct in Razor components, a *Razor directive*. A Razor directive is a reserved keyword prefixed with `@` that appears in Razor markup that changes the way component markup is compiled or functions. The `@page` Razor directive specifies the route template for the component. This component is reached in a browser at the relative URL `/welcome`. By convention, most of a component's directives are placed at the top of the component definition file.

The PageTitle component is a component built into the framework that specifies a page title.

"`Welcome to Blazor!`" is the first rendered body markup of the component per the content of the H1 heading element (`<h1>`).

Next, a welcome message is displayed using Razor syntax by prefixing the at symbol (`@`) to a C# variable (`welcomeMessage`).

The `@code` block contains the C# code of the component. `welcomeMessage` is a private string initialized with a value.

In the following sections of this article:

- Three components for webpage navigation and layout are described, the `NavMenu`, NavLink, and `MainLayout` components.
- The components created by scaffolding for CRUD operations on movie database entities are discussed.

`NavMenu` component for navigation

The `NavMenu` component (`Components/Layout/NavMenu.razor`) implements sidebar navigation using NavLink components, which render navigation links to other Razor components.

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.

NavLinkMatch.All assigned to the Match parameter configures the component to display an active CSS class when it matches the entire current URL.

The NavLink component built into the Blazor framework for any Blazor app to use, while the `NavMenu` component is only part of Blazor project templates.

`Components/Layout/NavMenu.razor`:

razor

```razor
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">BlazorWebAppMovies</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<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>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="weather">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Weather
            </NavLink>
        </div>
    </nav>
</div>
```

Notice in the `NavMenu` component's first `<div>` element the brand link text (`<a>` element content). Change the brand from `BlazorWebAppMovies` to `Sci-fi Movies`:

diff

```diff
- <a class="navbar-brand" href="">BlazorWebAppMovies</a>
+ <a class="navbar-brand" href="">Sci-fi Movies</a>
```

To allow users to reach the movies `Index` page, add a navigation menu entry to the `NavMenu` component. Immediately after the markup (`<div>`) for the `Weather` component's `NavLink`, add the following markup:

razor

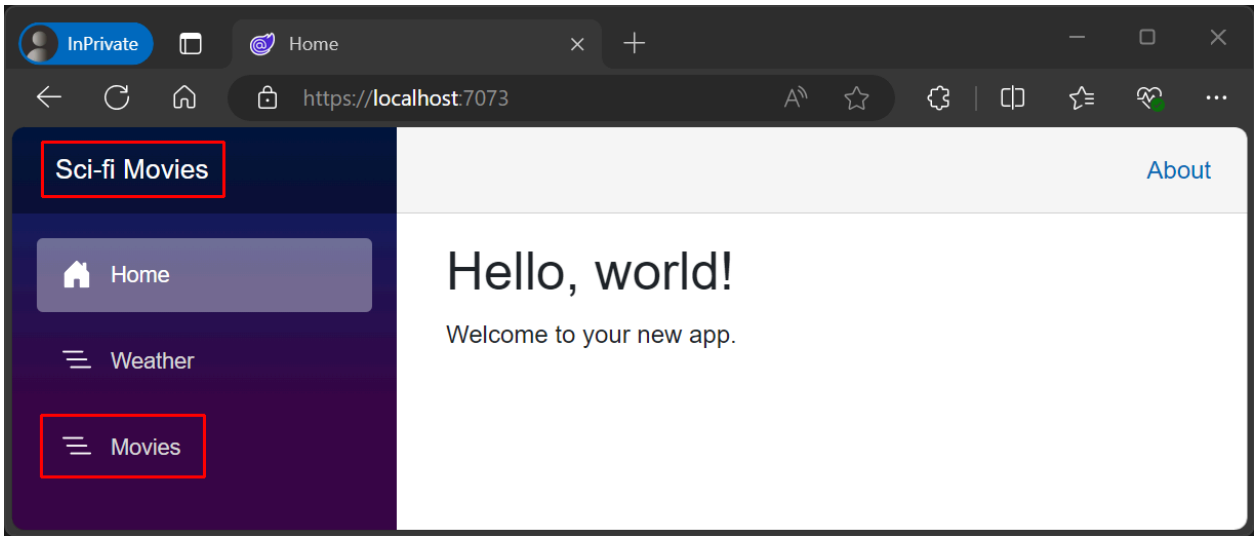
```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="movies">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Movies
    </NavLink>
</div>
```

The final `NavMenu` component after making the preceding changes:

razor

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">Sci-fi Movies</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<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>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="weather">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Weather
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="movies">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Movies
            </NavLink>
        </div>
    </nav>
</div>
```

Run the app to see the updated brand at the top of the sidebar navigation and a link to reach the movies page (**Movies**):



Stop the app by closing the browser's window.

`MainLayout` component for layout

The `MainLayout` component is the app's default layout. The `MainLayout` component inherits LayoutComponentBase, which is a base class for components that represent a layout. The app's components that use the layout are rendered where the Body (`@Body`) appears in the markup.

`Components/Layout/MainLayout.razor`:

razor

```
@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/core/"
target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

The `MainLayout` component adopts the following additional specifications:

- The `NavMenu` component is rendered in the sidebar. Notice that you only need to place an HTML tag with the component name to render a component at that location in Razor markup. This allows you to nest components within each other and within any HTML layout that you implement.
- The `<main>` element's content includes:
  - An **About** link that sends the user to the ASP.NET Core documentation landing page.
  - An `<article>` element with the Body (`@Body`) parameter, where components that use the layout are rendered.

The default layout (`MainLayout` component) is specified in the `Routes` component (`Components/Pages/Routes.razor`):

```razor
<RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)"
/>
```

Individual components are free to set their own non-default layout, and a layout can be applied to whole folder of components via an `_Imports.razor` file in the same folder. These features are covered in detail in the Blazor documentation.

Create, Read, Update, Delete (CRUD) components

The following sections explain the composition of the movie CRUD components and how they work.

`Index` component

Open the `Index` component definition file (`Components/Pages/Movies/Index.razor`) and examine the Razor directives at the top of the file.

The `@page` directive's route template indicates the URL for the page is `/movies`.

`@using` directives appear to access the following API:

- Microsoft.EntityFrameworkCore
- Microsoft.AspNetCore.Components.QuickGrid
- `BlazorWebAppMovies.Models`
- `BlazorWebAppMovies.Data`

The database context factory (`IDbContextFactory<BlazorWebAppMoviesContext>`) is injected into the component with the `@inject` directive. The factory approach requires that a database context be disposed, so the component implements the IAsyncDisposable interface with the `@implements` directive.

The page title is set via the Blazor framework's PageTitle component, and an H1 section heading is the first rendered element:

razor

```razor
<PageTitle>Index</PageTitle>

<h1>Index</h1>
```

A link is rendered to navigate to the `Create` page at `/movies/create`:

razor

```razor
<p>
    <a href="movies/create">Create New</a>
</p>
```

The QuickGrid component displays movie entities. The item provider is a `DbSet<Movie>` obtained from the created database context (CreateDbContext) of the injected database context factory (`DbFactory`). For each movie entity, the component displays the movie's title, release date, genre, and price. A column also holds links to edit, see details, and delete each movie entity.

razor

```razor
<QuickGrid Class="table" Items="context.Movie">
    <PropertyColumn Property="movie => movie.Title" />
    <PropertyColumn Property="movie => movie.ReleaseDate" />
    <PropertyColumn Property="movie => movie.Genre" />
    <PropertyColumn Property="movie => movie.Price" />

    <TemplateColumn Context="movie">
        <a href="@($"movies/edit?id={movie.Id}")">Edit</a> |
        <a href="@($"movies/details?id={movie.Id}")">Details</a> |
        <a href="@($"movies/delete?id={movie.Id}")">Delete</a>
    </TemplateColumn>
</QuickGrid>

@code {
    private BlazorWebAppMoviesContext context = default!;

    protected override void OnInitialized()
```

```
    {
        context = DbFactory.CreateDbContext();
    }

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

In the code block (`@code`):

- The `context` field holds the database context, typed as a `BlazorWebAppMoviesContext`.
- The `OnInitialized` lifecycle method assigns a created database context (CreateDbContext) from the injected factory (`DbFactory`) to the `context` variable.
- The asynchronous `DisposeAsync` method disposes of the database context when the component is disposed.

Notice how the context (`Context`) parameter of the TemplateColumn<TGridItem> specifies a parameter name (`movie`) for the context instance of the column. Specifying a name for the context instance makes the markup more readable (the default name for the context is simply `context`). `Movie` class properties are read from the context instance. For example, the movie identifier (`Id`) is available in `movie.Id`.

The at symbol (`@`) with parentheses (`@(...)`), which is called an *explicit Razor expression*, allows the `href` of each link to include the movie entity's `Id` property in the link query string as an *interpolated string* (`$...{...}...`). For a movie identifier (`Id`) of 7, the string value provided to the `href` to edit that movie is `movies/edit?id=7`. When the link is followed, the `id` field is read from the query string by the `Edit` component to load the movie.

For the movie example from the last part of the tutorial series, *The Matrix©*, the QuickGrid component renders the following HTML markup (some elements and attributes aren't present to simplify display). See how the explicit Razor expressions and interpolated strings produced the `href` values for the links to other pages. The movie's identifier in the database happens to be `3` for this example, so the `id` is `3` in the query strings for the `Edit`, `Details`, and `Delete` pages. You may see a different value when you run the app.

HTML

```
<table>
    <thead>
        <tr>
            <th>Title</th>
            <th>ReleaseDate</th>
```

```
            <th>Genre</th>
            <th>Price</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>The Matrix</td>
            <td>3/29/1999</td>
            <td>Sci-fi (Cyberpunk)</td>
            <td>5.00</td>
            <td>
                <a href="movies/edit?id=3">Edit</a> |
                <a href="movies/details?id=3">Details</a> |
                <a href="movies/delete?id=3">Delete</a>
            </td>
        </tr>
    </tbody>
</table>
```

The column names are taken from the `Movie` model properties, so the release date doesn't have a space between the words. Add a Title to the PropertyColumn<TGridItem,TProp> with a value that includes a space between the words:

razor

```
<PropertyColumn Property="movie => movie.ReleaseDate" Title="Release Date"
/>
```

Run the app to see that the column displays two words for the release date.

Stop the app by closing the browser's window.

`Details` component

Open the `Details` component definition file (`Components/Pages/Movies/Details.razor`).

The `@page` directive at the top of the file indicates the relative URL for the page is `/movies/details`. As before, the database context is injected, and namespaces are provided to access API (`BlazorWebAppMovies.Models` and `Microsoft.EntityFrameworkCore`). The `Details` component also injects the app's NavigationManager, which is used for a variety of navigation-related operations in components.

razor

```
@page "/movies/details"
@using Microsoft.EntityFrameworkCore
@using BlazorWebAppMovies.Models
@inject IDbContextFactory<BlazorWebAppMovies.Data.BlazorWebAppMoviesContext>
DbFactory
@inject NavigationManager NavigationManager
```

Details for a movie entity are only shown if the movie, located by its identifier (`Id`) from the query string, has been loaded for display. The presence of the movie in `movie` is checked with an `@if` Razor statement:

razor

```
@if (movie is null)
{
    <p><em>Loading...</em></p>
}
```

When the movie is loaded, it's displayed as a description list (MDN documentation) along with two links:

- The first link provides the user an opportunity to edit the entity.
- The second link allows the user to return to the movies `Index` page.

CSS classes aren't shown in the following example to simplify the Razor markup for display:

razor

```
<dl>
    <dt>Title</dt>
    <dd>@movie.Title</dd>
    <dt>ReleaseDate</dt>
    <dd>@movie.ReleaseDate</dd>
    <dt>Genre</dt>
    <dd>@movie.Genre</dd>
    <dt>Price</dt>
    <dd>@movie.Price</dd>
</dl>
<div>
    <a href="@($"/movies/edit?id={movie.Id}")">Edit</a> |
    <a href="@($"/movies")">Back to List</a>
</div>
</div>
```

Add a space to the content of the description term element (`<dt>`) for the movie's release date to separate the words:

diff

```
- <dt class="col-sm-2">ReleaseDate</dt>
+ <dt class="col-sm-2">Release Date</dt>
```

Examine the C# of the component's `@code` block:

C#

```
private Movie? movie;

[SupplyParameterFromQuery]
private int Id { get; set; }

protected override async Task OnInitializedAsync()
{
    using var context = DbFactory.CreateDbContext();
    movie = await context.Movie.FirstOrDefaultAsync(m => m.Id == Id);

    if (movie is null)
    {
        NavigationManager.NavigateTo("notfound");
    }
}
```

The `movie` variable is a private field of type `Movie`, which is a null-reference type (`?`), meaning that `movie` might be set to `null`.

The `Id` is a *component parameter* supplied from the component's query string due to the presence of the [SupplyParameterFromQuery] attribute. If the identifier is missing, `Id` defaults to zero (`0`).

`OnInitializedAsync` is the first component lifecycle method that we've seen. This method is executed when the component loads. FirstOrDefaultAsync is called on the database set (`DbSet<Movie>`) to retrieve the movie entity with and `Id` equal to the `Id` parameter that was set by the query string. If `movie` is `null`, NavigationManager.NavigateTo is used to navigate to a `notfound` endpoint.

There isn't an actual `notfound` endpoint (Razor component) in the app. When adopting server-side rendering (SSR), Blazor doesn't have a mechanism to return a 404 (Not Found) status code. As a temporary workaround, a 404 is generated by navigating to a non-existent endpoint. This scaffolded code is for your further implementation of a suitable result when not finding an entity. For example, you could have the component direct the user to a page where they can file an inquiry with your support team, or you could remove the injected NavigationManager and NavigationManager.NavigateTo

code and replace it with Razor markup and code that displays a message to the user that the entity wasn't found.

`Create` component

Open the `Create` component definition file (`Components/Pages/Movies/Create.razor`).

The component uses a built-in component called an EditForm, which renders a form for user input and includes validation features.

CSS classes aren't present in the following example to simplify the display:

razor

```
<EditForm method="post" Model="Movie" OnValidSubmit="AddMovie"
FormName="create" Enhance>
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label for="title">Title:</label>
        <InputText id="title" @bind-Value="Movie.Title" />
        <ValidationMessage For="() => Movie.Title" />
    </div>
    <div>
        <label for="releasedate">ReleaseDate:</label>
        <InputDate id="releasedate" @bind-Value="Movie.ReleaseDate" />
        <ValidationMessage For="() => Movie.ReleaseDate" />
    </div>
    <div>
        <label for="genre">Genre:</label>
        <InputText id="genre" @bind-Value="Movie.Genre" />
        <ValidationMessage For="() => Movie.Genre" />
    </div>
    <div>
        <label for="price">Price:</label>
        <InputNumber id="price" @bind-Value="Movie.Price" />
        <ValidationMessage For="() => Movie.Price" />
    </div>
    <button type="submit">Create</button>
</EditForm>
```

Add a space to the content of the label element (`<label>`) for the movie's release date to separate the words:

diff

```
- <label for="releasedate" class="form-label">ReleaseDate:</label>
+ <label for="releasedate" class="form-label">Release Date:</label>
```

The Model parameter is assigned the model, in this case `Movie`. OnValidSubmit specifies a method to invoke (`AddMovie`) when the form is submitted and the data is valid. By convention, every form should assign a FormName to prevent form collisions when multiple forms are present on a page. The Enhance flag activates a Blazor feature for server-side rendering (SSR) that submits the form without performing a full-page reload.

For validation:

- The DataAnnotationsValidator adds data annotations validation support, which is covered later in this tutorial series.
- The ValidationSummary component displays a list of validation messages.
- The ValidationMessage<TValue> components hold validation messages for the form's fields.

Blazor includes several form element components to assist you with creating forms, including EditForm and various input components, such as InputText, InputDate<TValue>, and InputNumber<TValue>. Each input component is bound to a model property with `@bind-Value` Razor syntax, where `Value` is a property in each input component.

In the component's `@code` block, C# code includes a `Movie` component parameter tied to the form via the [SupplyParameterFromForm] attribute.

The `AddMovie` method:

- Is called when the form is submitted.
- Adds the movie data bound to the form's model (`Movie`) if form validation passes.
- SaveChangesAsync is called on the database context to save the movie.
- NavigationManager is used to return the user to the movies `Index` page.

C#

```
@code {
    [SupplyParameterFromForm]
    private Movie Movie { get; set; } = new();

    private async Task AddMovie()
    {
        using var context = DbFactory.CreateDbContext();
        context.Movie.Add(Movie);
        await context.SaveChangesAsync();
        NavigationManager.NavigateTo("/movies");
    }
}
```

> ⚠ **Warning**
>
> Although it isn't a concern for the app in this tutorial, binding form data to entity data models can be susceptible to overposting attacks. Additional information on this subject appears later in this article.

`Delete` component

Open the `Delete` component definition file (`Components/Pages/Movies/Delete.razor`).

Add a space to the content of the description term element (`<dt>`) for the movie's release date to separate the words:

diff

```
- <dt class="col-sm-2">ReleaseDate</dt>
+ <dt class="col-sm-2">Release Date</dt>
```

Examine the Razor markup for the submit button of the EditForm (CSS class removed for simplicity):

razor

```
<button type="submit" disabled="@(movie is null)">Delete</button>
```

The **Delete** button sets its disabled HTML attribute (MDN documentation) based on the presence of the movie (not `null`) using an explicit Razor expression (`@(...)`).

In the C# code of the `@code` block, the `DeleteMovie` method removes the movie, saves the changes to the database, and navigates the user to the movies `Index` page. The exclamation point on the movie field (`movie!`) is the null-forgiving operator (C# Language Reference), which suppresses nullable warnings for `movie`.

C#

```
private async Task DeleteMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Movie.Remove(movie!);
    await context.SaveChangesAsync();
    NavigationManager.NavigateTo("/movies");
}
```

`Edit` component

Open the `Edit` component definition file (`Components/Pages/Movies/Edit.razor`).

Add a space to the content of the label element (`<label>`) for the movie's release date to separate the words:

diff

```
- <label for="releasedate" class="form-label">ReleaseDate:</label>
+ <label for="releasedate" class="form-label">Release Date:</label>
```

The component uses an EditForm similar to the `Create` component.

The movie entity's identifier `Id` is stored in a hidden field of the form:

razor

```
<input type="hidden" name="Movie.Id" value="@Movie.Id" />
```

Examine the C# code of the `@code` block:

C#

```
private async Task UpdateMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Attach(Movie!).State = EntityState.Modified;

    try
    {
        await context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie!.Id))
        {
            NavigationManager.NavigateTo("notfound");
        }
        else
        {
            throw;
        }
    }

    NavigationManager.NavigateTo("/movies");
}

private bool MovieExists(int id)
```

```
{
    using var context = DbFactory.CreateDbContext();
    return context.Movie.Any(e => e.Id == id);
}
```

The movie entity's EntityState is set to Modified, which signifies that the entity is tracked by the context, exists in the database, and that some or all of its property values are modified.

If there's a concurrency exception and the movie entity no longer exists at the time that changes are saved, the component redirects to the non-existent endpoint (`notfound`), which results in returning a 404 (Not Found) status code. You could change this code to notify the user that the movie no longer exists in the database or create a dedicated *Not Found* component and navigate the user to that endpoint. If the movie exists and a concurrency exception is thrown, for example when another user has already modified the entity, the exception is rethrown by the component with the throw statement (C# Language Reference). Additional guidance on handling concurrency with EF Core in Blazor apps is provided by the Blazor documentation.

> ⚠ **Warning**
>
> Although it isn't a concern for the app in this tutorial, binding form data to entity data models can be susceptible to overposting attacks. Additional information on this subject appears in the next section.

Mitigate overposting attacks

Statically-rendered server-side forms, such as those in the `Create` and `Edit` components, can be vulnerable to an *overposting* attack, also known as a *mass assignment* attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

In the example `Create` and `Edit` components of this tutorial, the `Movie` model doesn't include restricted properties for create and update operations, so overposting isn't a concern. However, it's important to keep overposting in mind when working with static SSR-based Blazor forms that you create and modify in the future.

To mitigate overposting, we recommend using a separate view model/data transfer object (DTO) for the form and database with create (insert) and update operations.

When the form is submitted, only properties of the view model/DTO are used by the component and C# code to modify the database. Any extra data included by a malicious user is discarded, so the malicious user is prevented from conducting an overposting attack.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- NavLink component
- ASP.NET Core Blazor layouts
- Razor directives (Razor syntax article) / Razor directives (Blazor documentation)
- ASP.NET Core Blazor QuickGrid component
- ASP.NET Core Blazor forms overview
- Concurrency with EF Core in Blazor apps
- ASP.NET Core Blazor globalization and localization: Explains how to render globalized and localized content to users in different cultures and languages, including how to accept comma number separators.

Next steps

Previous: Add and scaffold a model | Next: Work with a database

Build a Blazor movie database app (Part 4 - Work with a database)

Article • 10/21/2024

This article is the fourth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series focuses on the database context and directly working with the database's schema and data. Seeding the database with data is also covered.

Secure authentication flow required for production apps

This tutorial uses a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor Web Apps, see the following resources:

- ASP.NET Core Blazor authentication and authorization
- Secure ASP.NET Core server-side Blazor apps and the following articles in the *Server* security node
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
- Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Database context

The database context, `BlazorWebAppMoviesContext`, connects to the database and maps model objects to database records. The database context was created in the second part of this series. The scaffolded database context code appears in the `Program` file:

C#

```
builder.Services.AddDbContextFactory<BlazorWebAppMoviesContext>(options =>
    options.UseSqlServer(

builder.Configuration.GetConnectionString("BlazorWebAppMoviesContext") ??
        throw new InvalidOperationException(
            "Connection string 'BlazorWebAppMoviesContext' not found.")));
```

AddDbContextFactory registers a factory for the given context as a service in the app's service collection.

UseSqlServer or UseSqlite configures the context to connect to either a Microsoft SQL Server or SQLite database. Other providers are available to connect to additional types of databases.

GetConnectionString uses the ASP.NET Core Configuration system to read the `ConnectionStrings` key for the connection string name provided, which in the preceding example is `BlazorWebAppMoviesContext`.

For local development, configuration obtains the database connection string from the app settings file (`appsettings.json`). The `{CONNECTION STRING}` placeholder in the following example is the connection string:

JSON

```
"ConnectionStrings": {
  "BlazorWebAppMoviesContext": "{CONNECTION STRING}"
}
```

The following is an example connection string:

> Server=(localdb)\mssqllocaldb;Database=BlazorWebAppMoviesContext-00001111-aaaa-2222-bbbb-3333cccc4444;Trusted_Connection=True;MultipleActiveResultSets=true

When the app is deployed to a test/staging or production server, securely store the connection string outside of the project's configuration files.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production

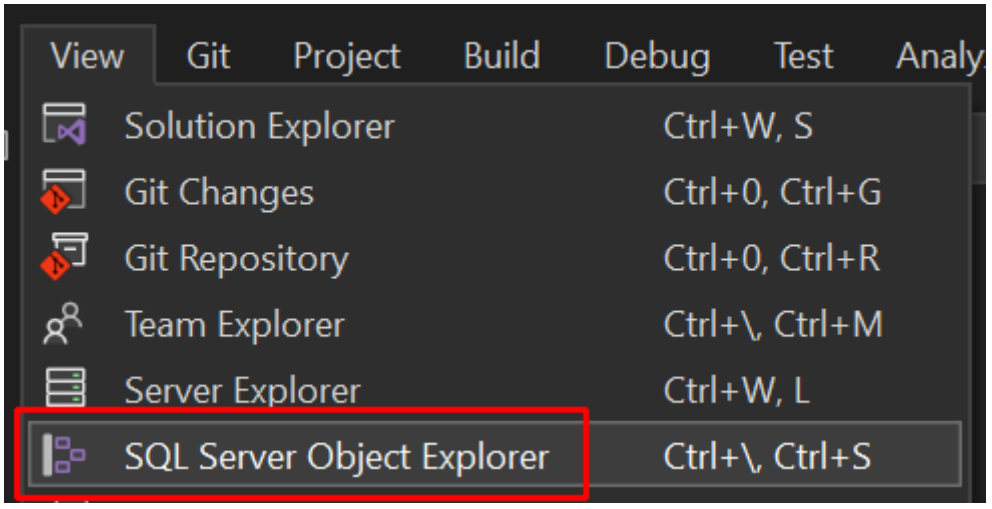
environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Database technology

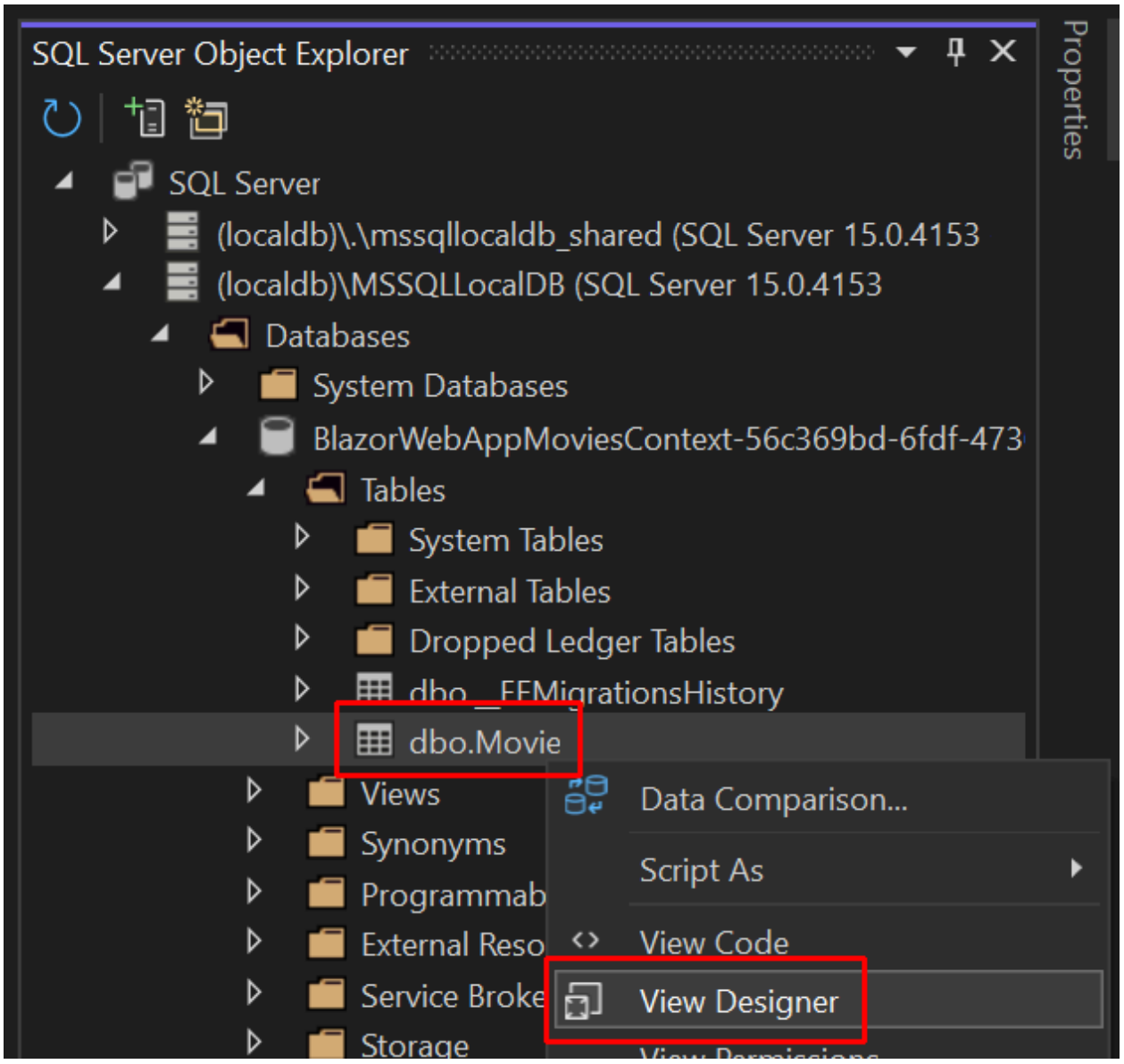
The Visual Studio version of this tutorial uses SQL Server.

SQL Server Express LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. Master database files (`*.mdf`) are placed in the `C:/Users/{USER}` directory, where the `{USER}` placeholder is the system's user ID.

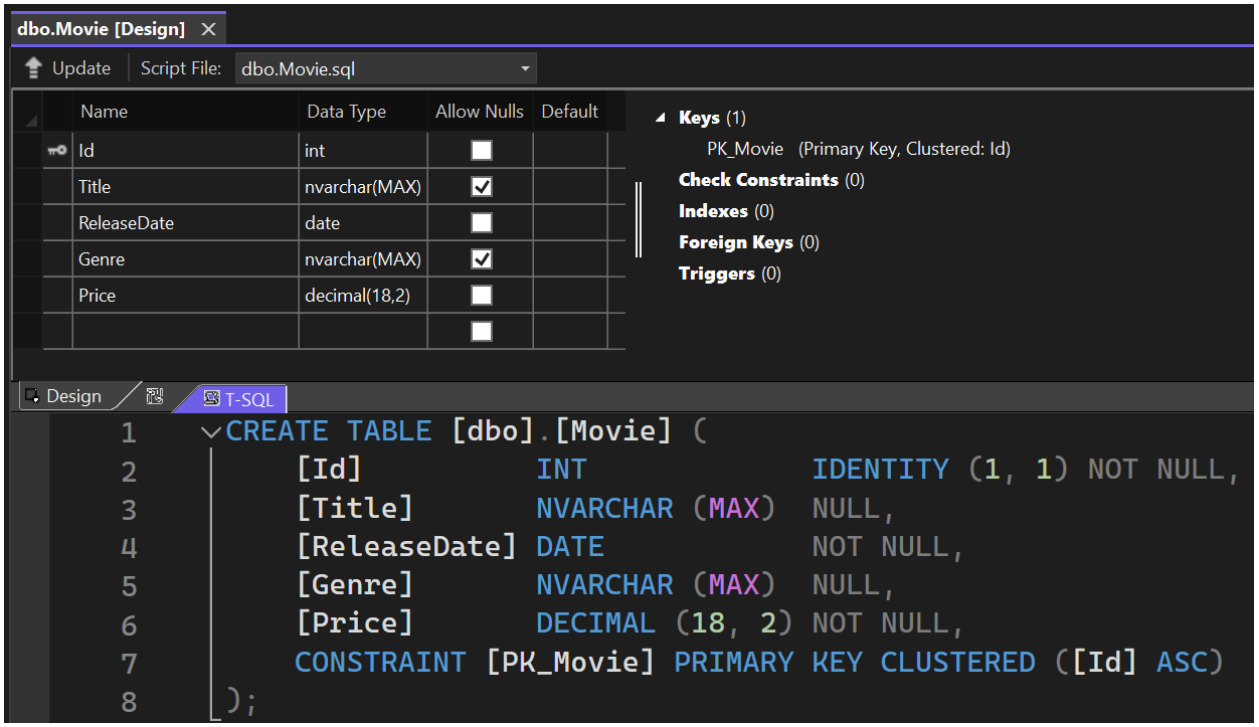
From the **View** menu, open **SQL Server Object Explorer** (SSOX).



Right-click on the `Movie` table and select **View Designer**:

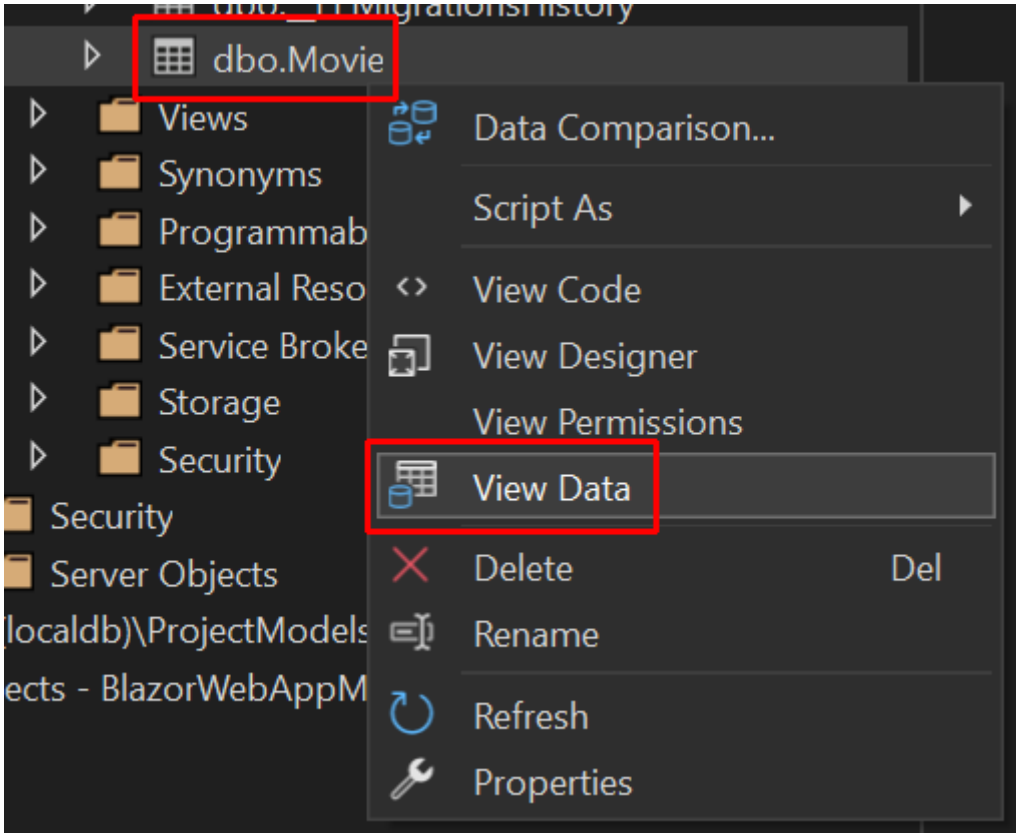


The **View Designer** opens:

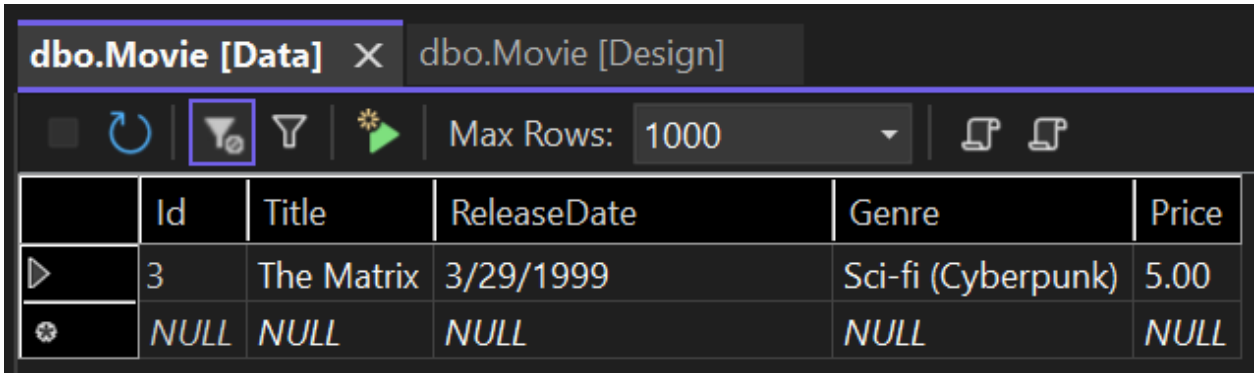


Note the key icon next to `ID`. EF creates a property named `ID` for the primary key.

Right-click on the `Movie` table and select **View Data**:



The table's data opens in a new tab in Visual Studio:



Seed the database

Seeding code can create a set of records for development testing or even be used to create the initial data for a new production database.

Create a new class named `SeedData` in the `Data` folder with the following code.

`Data/SeedData.cs`:

```
C#
```

```
using Microsoft.EntityFrameworkCore;
using BlazorWebAppMovies.Models;

namespace BlazorWebAppMovies.Data;

public class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using var context = new BlazorWebAppMoviesContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<BlazorWebAppMoviesContext>>());

        if (context == null || context.Movie == null)
        {
            throw new NullReferenceException(
                "Null BlazorWebAppMoviesContext or Movie DbSet");
        }

        if (context.Movie.Any())
        {
            return;
        }

        context.Movie.AddRange(
            new Movie
            {
                Title = "Mad Max",
                ReleaseDate = new DateOnly(1979, 4, 12),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 2.51M,
            },
            new Movie
            {
                Title = "The Road Warrior",
                ReleaseDate = new DateOnly(1981, 12, 24),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 2.78M,
            },
            new Movie
            {
                Title = "Mad Max: Beyond Thunderdome",
                ReleaseDate = new DateOnly(1985, 7, 10),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 3.55M,
            },
            new Movie
            {
                Title = "Mad Max: Fury Road",
                ReleaseDate = new DateOnly(2015, 5, 15),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 8.43M,
            },
            new Movie
```

```
            {
                Title = "Furiosa: A Mad Max Saga",
                ReleaseDate = new DateOnly(2024, 5, 24),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 13.49M,
            });

        context.SaveChanges();
    }
}
```

A database context instance is obtained from the dependency injection (DI) container. If movies are present, `return` is called to avoid seeding the database. When the database is empty, the *Mad Max* franchise (©Warner Bros. Entertainment) movies are seeded.

To execute the seed initializer, add the following code to the `Program` file immediately after the line that builds the app (`var app = builder.Build();`). The using statement ensures that the database context is disposed after the seeding operation completes.

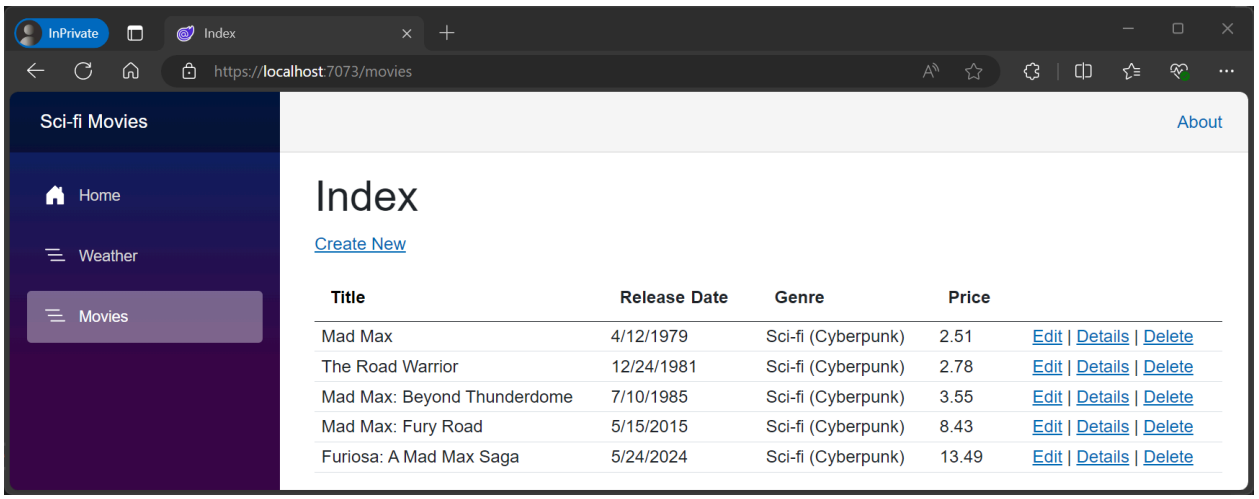
C#

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
```

If the database contains records from earlier testing, run the app and delete the entities that you created in the database. Stop the app by closing the browser's window.

When the database is empty, run the app.

Navigate to the movies `Index` page to see the seeded movies:



Bind a form to a model

Review the the `Edit` component (`Components/Pages/MoviePages/Edit.razor`).

When an HTTP GET request is made for the `Edit` component page (for example at the relative URL: `/movies/edit?id=6`):

- The OnInitializedAsync method fetches the movie with an `Id` of `6` from the database and assigns it to the `Movie` property.
- The EditForm.Model parameter specifies the top-level model object for the form. An edit context is constructed for the form using the assigned model.
- The form is displayed with the values from the movie.

When the `Edit` page is posted to the server, the form values on the page are bound to the `Movie` property because the [SupplyParameterFromForm] attribute is annotated on the `Movie` property:

C#

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

If the model state has errors when the form is posted, for example if `ReleaseDate` can't be converted into a date, the form is redisplayed with the submitted values. If no model errors exist, the movie is saved using the form's posted values.

Concurrency exception handling

Review the `UpdateMovie` method of the `Edit` component (`Components/Pages/MoviePages/Edit.razor`):

C#

```
private async Task UpdateMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Attach(Movie!).State = EntityState.Modified;

    try
    {
        await context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie!.Id))
```

```
            {
                NavigationManager.NavigateTo("notfound");
            }
            else
            {
                throw;
            }
        }

        NavigationManager.NavigateTo("/movies");
    }
```

Concurrency exceptions are detected when one client deletes the movie and a different client posts changes to the movie.

To test how concurrency is handled by the preceding code:

1. Select **Edit** for a movie, make changes, but don't select **Save**.
2. In a different browser window, open the app to the movie `Index` page and select the **Delete** link for the same movie to delete the movie.
3. In the previous browser window, post changes to the movie by selecting the **Save** button.
4. The browser is navigated to the `notfound` endpoint, which doesn't exist and yields a 404 (Not Found) result.

Additional guidance on handling concurrency with EF Core in Blazor apps is available in the Blazor documentation.

Stop the app

If the app is running, shut the app down by closing the browser's window.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Configuration articles:
  - Configuration in ASP.NET Core (ASP.NET Core Configuration system)
  - ASP.NET Core Blazor configuration (Blazor documentation)
  - Data seeding (EF Core documentation)
- Concurrency with EF Core in Blazor apps
- Database provider resources:
  - EF Core documentation
    - SQLite EF Core Database Provider Limitations
    - Customize migration code
  - SQLite ALTER TABLE statement (SQLite documentation)
- Blazor Web App security
  - ASP.NET Core Blazor authentication and authorization
  - Secure ASP.NET Core server-side Blazor apps and the following articles in the *Server* security node
  - Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
  - Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

Legal

Next steps

Previous: Learn about Razor components | Next: Add Validation

Build a Blazor movie database app (Part 5 - Add validation)

Article • 11/14/2024

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

This article is the fifth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series explains how metadata of the `Movie` model is used to validate user input in the forms that create and edit movies.

Validation using data annotations

Validation rules are specified on a model class using *data annotations*. The following list shows some of the System.ComponentModel.DataAnnotations attributes for user input validation of public properties in a form's model:

- [Required]: Require that the user provide a value.
- [StringLength]: Specifies the minimum and maximum length of characters. Note that a `MinimumLength` passed to the attribute doesn't make the string required (apply the [Required] attribute).
- [RegularExpression]: Specify a pattern to match for the user's input.
- [Range]: Specify the minimum and maximum values.

Value types, such as `decimal`, `int`, `float`, `DateOnly`, `TimeOnly`, and `DateTime`, are inherently required. Placing a [Required] attribute on value types isn't necessary.

Additional data annotations that you can use in your forms are covered by the Blazor reference documentation.

Add validation to the `Movie` model

Add the following data annotations to the `Movie` class properties. To update all of the properties at once, you can copy and paste the entire `Models/Movie.cs` file, which appears after the following code example.

diff

```
+ [Required]
+ [StringLength(60, MinimumLength = 3)]
  public string? Title { get; set; }

+ [Required]
* [StringLength(30)]
+ [RegularExpression(@"^[A-Z]+[a-zA-Z()\s-]*$")]
  public string? Genre { get; set; }

+ [Range(0, 100)]
  [DataType(DataType.Currency)]
  [Column(TypeName = "decimal(18, 2)")]
  public decimal Price { get; set; }
```

`Models/Movie.cs` file after the preceding data annotations are applied to the properties:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace BlazorWebAppMovies.Models;

public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(60, MinimumLength = 3)]
    public string? Title { get; set; }

    public DateOnly ReleaseDate { get; set; }

    [Required]
    [StringLength(30)]
    [RegularExpression(@"^[A-Z]+[a-zA-Z()\s-]*$")]
    public string? Genre { get; set; }

    [Range(0, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

The preceding validation rules are merely for demonstration and aren't optimal for a production system. For example, the preceding validation prevents entering a movie with only one or two characters and doesn't allow additional special characters for a movie's genre.

Create an EF Core migration and update the database

A data model schema defines how data is organized and connected within a relational database.

Adding the data annotations to the `Movie` class in the preceding section doesn't automatically result in matching changes to the database's schema.

Review the annotations applied to the `Title` property:

C#

```
[Required]
[StringLength(60, MinimumLength = 3)]
public string? Title { get; set; }
```

The difference between the model property and the database's schema are summarized in the following table. Neither constraint matches after the data annotation is applied to the `Movie` model.

Expand table

Constraint	Model `Title` property	Database `Title` column
Maximum length	60 characters	Byte pairs up to ~2 GB† NVARCHAR (MAX)
Required	✔ [Required]	✘ `NULL` is permitted in the column.

†Database character columns are defined by *size* (byte pairs). One byte-pair per character is used for characters defined in the Unicode range 0 to 65,535. However, individual characters outside of that Unicode range take multiple byte-pairs to store, so the actual number of characters that a column can store is arbitrary. The important concept for our purposes in comparing the `Title` property of the `Movie` model and the database schema for the `Title` column is that ~2 GB of stored byte-pairs in the

database far exceeds the 60 character limit set for the property. The database schema *should be adjusted downward* to match the app's constraint.

To match the `Movie` model's `Title` property length in the app, the database should indicate `NVARCHAR (60)` for the size of the `Title` column. The schema difference doesn't cause EF Core to throw an exception when the app is used because a user posting a 60 character movie title fits within the database's ~2 GB byte-pair limit for a movie title. However, consider the reverse situation where a model property is given a constraint larger than what the database permits and the user posts a string too long for a database character column: An exception is thrown by the database or data is truncated when the user posts the value. You should always keep the app's models aligned with the database's schema because a misaligned schema can cause exceptions and the storage of incorrect data.

Although the `Title` property is a nullable reference type (NRT), as indicated by the `?` on the `string` type (`string?`), the database shouldn't store a `NULL` value in its `Title` column due to the model's `Required` constraint. When the database's schema is updated in the next step, the database's `Title` column should reflect `NOT NULL` for the `Title` column to match the property. The important concept is that just because a model property is an NRT and can hold a `null` value in code doesn't mean that the database column's schema should be nullable (`NULL` permitted). These are independent conditions used for different purposes: NRTs are used to prevent coding errors with nullable types, while the database schema reflects the exact type and size of stored data.

To align the model and the database schema, create and apply an EF Core *database migration* with a migration name that identifies the migration changes. The migration name is similar to a commit message in a version control system. In the following command examples, the migration name "`NewMovieDataAnnotations`" reflects that new data annotations are added to the `Movie` model.

> **ⓘ Important**
>
> Make sure that the app isn't running for the next steps.
>
> Stopping the app when using Visual Studio only requires you to close the browser's window.
>
> When using VS Code, close the browser's window and stop the app in VS Code with **Run** > **Stop Debugging** or by pressing `Shift`+`F5` on the keyboard.

When using the .NET CLI, close the browser's window and stop the app in the command shell with `Ctrl`+`C` (Windows) or `⌘`+`C` (macOS).

In Visual Studio **Solution Explorer**, double-click **Connected Services**. In the **Service Dependencies** area, select the ellipsis (`...`) followed by **Add migration** in the **SQL Server Express LocalDB** area.

Give the migration a **Migration name** of `NewMovieDataAnnotations` to describe the migration. Wait for the database context to load in the **DbContext class names** field. Select **Finish** to create the migration. Select the **Close** button when the operation completes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load. Select the **Finish** button. Select the **Close** button when the operation completes.

After applying the migration, the model property and the database's schema match, as summarized in the following table.

Expand table

Constraint	**Model `Title` property**	**Database `Title` column**
Maximum length	60 characters	Sixty (60) byte pairs are permitted†. NVARCHAR (60)
Required	✔ [Required]	✔ `NOT NULL` is indicated for the column.

†Sixty (60) byte pairs is 60 characters if one byte-pair per character is used to store the movie title, which is true when using characters defined in the Unicode range 0 to 65,535.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core
- Migrations overview (EF Core documentation)
- nchar and nvarchar (Transact-SQL) (SQL Server documentation)
- Blazor enhanced forms

Next steps

Previous: Work with a database | Next: Add search

Build a Blazor movie database app (Part 6 - Add search)

Article • 11/14/2024

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

This article is the sixth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers adding a search feature to the movies `Index` component to filter movies by title.

Implement a filter feature for the `QuickGrid` component

The QuickGrid component is used by the movie `Index` component (`Components/MoviePages/Index.razor`) to display movies from the database:

```razor
<QuickGrid Class="table" Items="context.Movie">
    ...
</QuickGrid>
```

The Items parameter receives an `IQueryable<TGridItem>`, where `TGridItem` is the type of data represented by each row in the grid (`Movie`). Items is assigned a collection of movie entities (`DbSet<Movie>`) obtained from the created database context (CreateDbContext) of the injected database context factory (`DbFactory`).

To make the `QuickGrid` component filter on the movie title, the `Index` component should:

- Set a filter string as a *component parameter* from the query string.
- If the parameter has a value, filter the movies returned from the database.
- Provide an input for the user to provide the filter string and a button to trigger a reload using the filter.

Start by adding the following code to the `@code` block of the `Index` component (`MoviePages/Index.razor`):

C#

```
[SupplyParameterFromQuery]
private string? TitleFilter { get; set; }

private IQueryable<Movie> FilteredMovies =>
    context.Movie.Where(m => m.Title!.Contains(TitleFilter ??
string.Empty));
```

`TitleFilter` is the filter string. The property is provided the [SupplyParameterFromQuery] attribute, which lets Blazor know that the value of `TitleFilter` should be assigned from the query string when the query string contains a field of the same name (for example, `?titleFilter=road+warrior` yields a `TitleFilter` value of `road warrior`). Note that query string field names, such as `titleFilter`, aren't case sensitive.

The `FilteredMovies` property is an `IQueryable<Movie>`, which is the type for assignment to the `QuickGrid`'s Items parameter. The property filters the list of movies based on the supplied `TitleFilter`. If a `TitleFilter` isn't assigned a value from the query string (`TitleFilter` is `null`), an empty string (`string.Empty`) is used for the Contains clause. Therefore, no movies are filtered for display.

Change the `QuickGrid` component's Items parameter to use the `movies` collection:

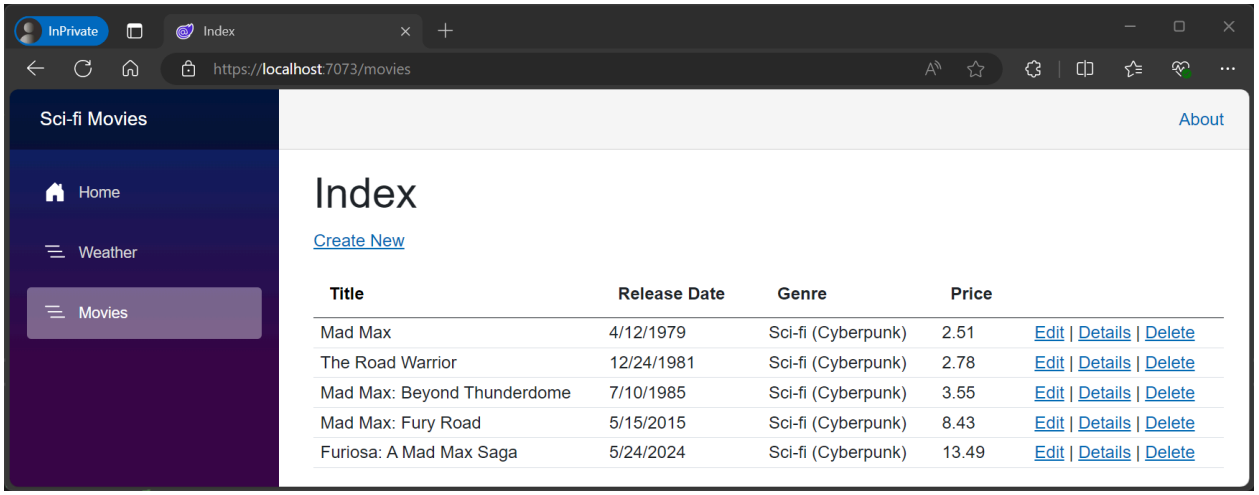
diff

```
- <QuickGrid Class="table" Items="context.Movie">
+ <QuickGrid Class="table" Items="FilteredMovies">
```

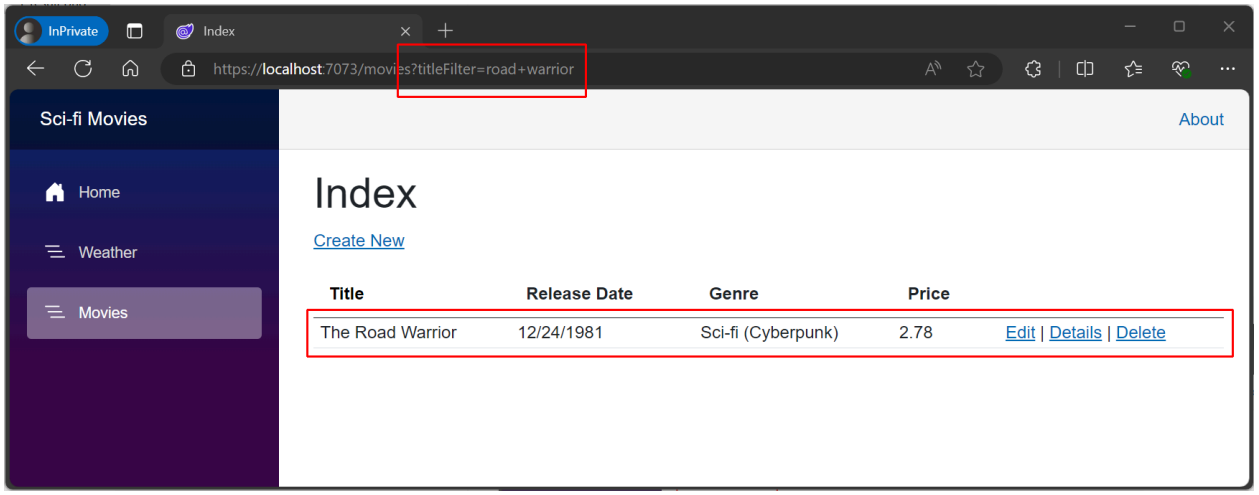
The `movie => movie.Title!.Contains(...)` code is a *lambda expression*. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where or Contains methods. LINQ queries aren't executed when they're defined or when they're modified by calling a method, such as Where, Contains, or OrderBy. Rather, query execution is deferred. The evaluation of an expression is delayed until its realized value is iterated.

The Contains method is run on the database, not in the C# code. The case sensitivity of the query depends on the database and the collation. For SQL Server, Contains maps to SQL LIKE, which is case insensitive. SQLite with default collation provides a mixture of case sensitive and case insensitive filtering, depending on the query. For information on making case insensitive SQLite queries, see the Additional resources section of this article.

Run the app and navigate to the movies `Index` page at `/movies`. The movies in the database load:



Append a query string to the URL in the address bar: `?titleFilter=road+warrior`. For example, the full URL appears as `https://localhost:7073/movies?titleFilter=road+warrior`, assuming the port number is `7073`. The filtered movie is displayed:



Next, give users a way to provide the `titleFilter` filter string via the component's UI. Add the following HTML under the H1 heading (`<h1>Index</h1>`). The following HTML reloads the page with the contents of the textbox as a query string value:

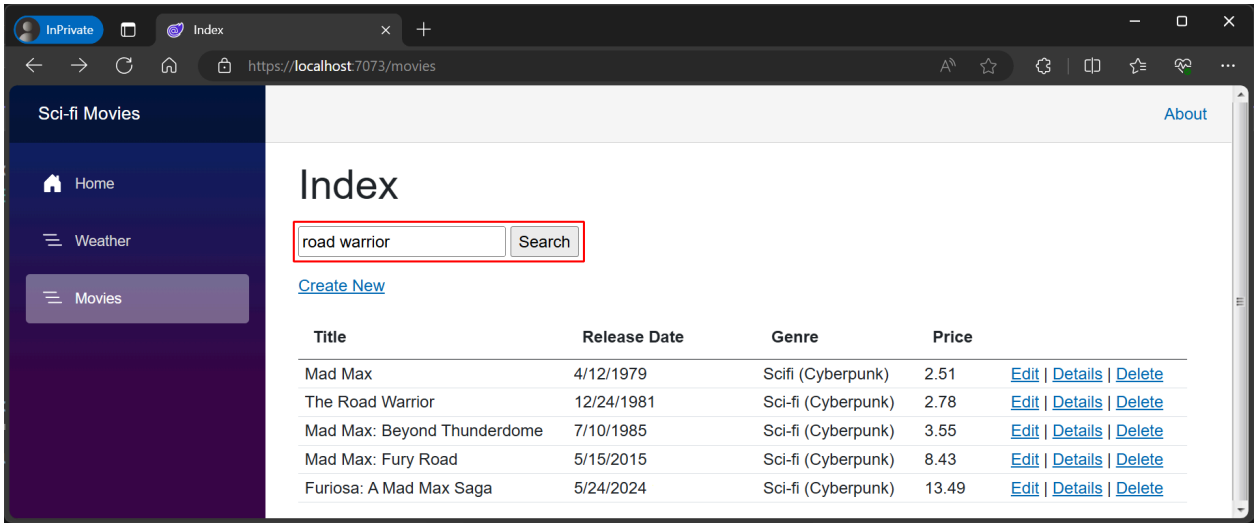
HTML

```
<div>
    <form action="/movies" data-enhance>
        <input type="search" name="titleFilter" />
        <input type="submit" value="Search" />
    </form>
</div>
```

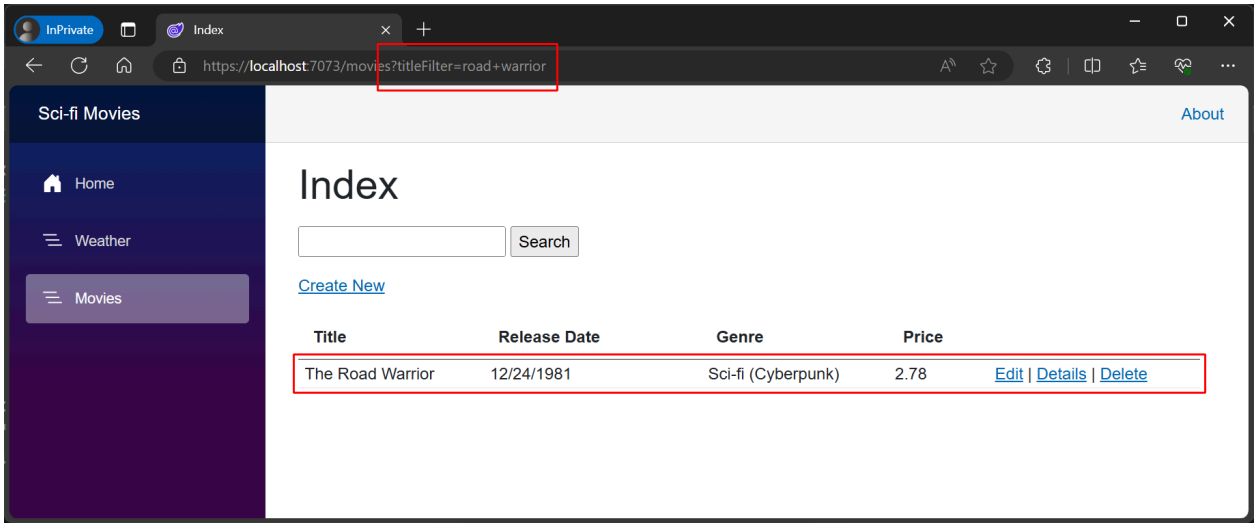
The `data-enhance` attribute applies *enhanced navigation* to the component, where Blazor intercepts the GET request and performs a fetch request instead. Blazor then patches the response content into the page, which avoids a full-page reload and preserves more of the page state. The page loads faster, usually without losing the user's scroll position.

Save the file that you're working on. Apply the change by either restarting the app or using Hot Reload to apply the change to the running app.

Type "`road warrior`" into the search box and select the **Search** button to filter the movies:



The result after searching on `road warrior`:



Notice that the search box loses the search value ("`road warrior`") when the movies are filtered. If you want to preserve the searched value, add the `data-permanent` attribute:

diff

```
- <form action="/movies" data-enhance>
+ <form action="/movies" data-enhance data-permanent>
```

Stop the app

Stop the app by closing the browser's window.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- LINQ documentation
- Write C# LINQ queries to query data (C# documentation)
- Lambda Expression (C# documentation
- Case insensitive SQLite queries
  - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
  - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
  - Collations and Case Sensitivity

Legal

Mad Max, *The Road Warrior*, *Mad Max: Beyond Thunderdome*, *Mad Max: Fury Road*, and *Furiosa: A Mad Max Saga* are trademarks and copyrights of Warner Bros. Entertainment.

Next steps

Previous: Add validation | Next: Add a new field

Build a Blazor movie database app (Part 7 - Add a new field)

Article • 09/23/2024

This article is the seventh part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers adding a new field to the movie class, CRUD pages, and database.

The database update is handled by EF Core migrations. EF Core transparently tracks changes to the database in a migration history table and automatically throws an exception if the app's model classes aren't synchronized with the database's tables and columns. EF Core migrations make it possible to quickly troubleshoot database consistency problems.

> **ⓘ Important**
>
> Confirm that the app isn't running for the next steps.

Add a movie rating to the app's model

Open the `Models/Movie.cs` file and add a `Rating` property with a regular expression that limits the value of `Rating` to the exact Motion Picture Association film rating designations:

C#

```
[Required]
[RegularExpression(@"^(G|PG|PG-13|R|NC-17)$")]
public string? Rating { get; set; }
```

Add the movie rating to the app's CRUD components

Open the `Create` component definition file (`Components/Pages/MoviePages/Create.razor`).

Add the following `<div>` block between the `<div>` block for `Price` and the create button (`<button>`):

razor

```
<div class="mb-3">
    <label for="rating" class="form-label">Rating:</label>
    <InputText id="rating" @bind-Value="Movie.Rating" class="form-control"
/>
    <ValidationMessage For="() => Movie.Rating" class="text-danger" />
</div>
```

Open the `Delete` component definition file (`Components/Pages/MoviePages/Delete.razor`).

Add the following description list (`<dl>`) block between the description list block for `Price` and the `EditForm` component:

razor

```
<dl class="row">
    <dt class="col-sm-2">Rating</dt>
    <dd class="col-sm-10">@movie.Rating</dd>
</dl>
```

Open the `Details` component definition file (`Components/Pages/MoviePages/Details.razor`).

Add the following description list term (`<dt>`) and description list element (`<dl>`) after the term and element for `Price` just inside the closing `</dl>` tag:

razor

```
<dt class="col-sm-2">Rating</dt>
<dd class="col-sm-10">@movie.Rating</dd>
```

Open the `Edit` component definition file (`Components/Pages/MoviePages/Edit.razor`).

Add the following `<div>` block between the `<div>` block for `Price` and the save button (`<button>`):

razor

```
<div class="mb-3">
    <label for="rating" class="form-label">Rating:</label>
```

```
    <InputText id="rating" @bind-Value="Movie.Rating" class="form-control"
/>
    <ValidationMessage For="() => Movie.Rating" class="text-danger" />
</div>
```

Open the `Index` component definition file (`Components/Pages/MoviePages/Index.razor`).

Update the `QuickGrid` component to include the movie rating. Add the following PropertyColumn<TGridItem,TProp> immediately after the column for `Price`:

razor

```
<PropertyColumn Property="movie => movie.Rating" />
```

Update the `SeedData` class (`Data/SeedData.cs`) to provide a default value for the new `Rating` property for reseeding operations.

The following change is for the *Mad Max* `new Movie` block. `Rating = "R",` is added to the block:

diff

```
new Movie
{
    Title = "Mad Max",
    ReleaseDate = DateOnly.Parse("1979-4-12"),
    Genre = "Sci-fi (Cyberpunk)",
    Price = 2.51M,
+   Rating = "R",
},
```

Add the `Rating` property to each of the other `new Movie` blocks in the same fashion. Here are the ratings of the remaining *Mad Max* movies when you add the new `Rating` lines:

- *The Road Warrior*: R
- *Mad Max: Beyond Thunderdome*: PG-13
- *Mad Max: Fury Road*: R
- *Furiosa: A Mad Max Saga*: R

Save all of the updated files.

Don't run the app yet. Build the app to confirm that there are no errors.

Select **Build** > **Rebuild Solution** from the menu bar.

Fix any errors in the app that were introduced by pasting the preceding markup and code before proceeding to the next step.

Update the database

If you were to try and run the app at this point, the app would fail with a SQL exception because the database doesn't include a `Rating` column in its `Movie` table. There are three approaches that you can take to resolve the discrepancy between the database's schema and the model's schema:

- Modify the schema of the database so that it matches the model class. The advantage of this approach is that it maintains the database's data. Adopt this approach using either database tooling or by creating a database change script, which are approaches not covered by this tutorial but can be learned using other articles. The downside to this approach is that it requires more time and is more prone to errors due to its increased complexity. *This tutorial doesn't adopt this approach.*
- Use EF Core to automatically drop and recreate the database using the new model class schema, losing the data stored in the database in the process. The database is reseeded with fresh data when the app is run. This approach allows you to quickly evolve the model and database schema together. Don't use this approach on a production database with data that must be preserved. *This tutorial doesn't adopt this approach.*
- Use an EF Core migration to update the database schema after the model is changed in the app. This approach is efficient and preserves the database's data. ***This tutorial adopts this approach.***

Create a migration to update the database's schema. The movie rating, as a `Rating` column, is added to the database's `Movie` table.

In Visual Studio **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name** of `AddRatingField` to describe the migration. Wait for the database context to load in the **DbContext class names** field. Select **Finish** to create the migration. Select **Close** when the operation is complete.

The migration:

- Compares the `Movie` model with the `Movie` database table schema.
- Creates code to migrate the database schema to match the model.

Creating the migration doesn't automatically provision a default value for the rating when the database is updated. However, you can manually make a change to the migration file to apply a default movie rating value, which can be helpful when there are many records that require a default value. In this case, all but one of the *Mad Max* movies is rated *R*, so a default value of "`R`" for the `Rating` column is an appropriate choice. The one movie that doesn't have an *R* rating can be updated later in the running app.

Examine the files in the `Migrations` folder of the project in Visual Studio's **Solution Explorer**. Open the migration file that adds the movie rating field, which has a file name of `{TIME STAMP}_AddRatingField.cs`, where the `{TIME STAMP}` placeholder is a time stamp (for example, `20240530123755_AddRatingField.cs`).

Find the `AddColumn` block that adds the rating column to the `Movie` table in the database. Modify the last line that applies a default value (`defaultValue`). Change it from an empty string (`""`) to an *R* movie rating (`"R"`):

diff

```
migrationBuilder.AddColumn<string>(
    name: "Rating",
    table: "Movie",
    type: "nvarchar(max)",
    nullable: false,
-   defaultValue: "");
+   defaultValue: "R");
```

Save the migration file.

In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load. Select the **Finish** button. Select the **Close** button when the operation completes.

Modify the one movie that isn't rated *R*:

1. Run the app.
2. Edit the *Mad Max: Beyond Thunderdome* movie.
3. Update the movie rating from `R` to `PG-13`. Save the change.

> ⓘ **Note**

An alternative to modifying the migration file is to delete the records in the database and rerun the app to reseed the database. The seeding code was modified earlier to supply default values. This approach is useful in cases where the assignment of default values to fields is better controlled or faster for you to implement with C# code during seeding. For more information see the Delete all database records and reseed the database section at the end of this article.

Run the app and verify you can create, edit, and display movies with the new movie rating field.

Troubleshoot

In the event that the database becomes corrupted, delete the database and use migrations to recreate the database:

1. Select the database in **SQL Server Object Explorer** (SSOX).
2. Right-click on the database, and select **Delete**. *Make sure that you select the correct database in the list.*
3. Check **Close existing connections**.
4. Select **OK**.
5. In **Solution Explorer**, double-click **Connected Services**. In the **Service Dependencies** area, select the ellipsis (`...`) followed by **Update database** in the **SQL Server Express LocalDB** area. Updating the database executes the existing migrations that recreate the database.

Delete all database records and reseed the database

This section describes an alternative process for updating the database with a default value for a new model property without modifying the migration file. Following the guidance in this section isn't necessary if you followed all of the steps in the Update the database section earlier in this article.

To delete all of the records in the database, use one of the following approaches:

- Run the app and use the delete links in the browser. This approach is reasonably fast when there are only a few records to delete.
- In Visual Studio from **SQL Server Object Explorer** (SSOX), delete the database records. With the database table visible, right-click the table and select **View Data**. When the table opens to show the movie records, select the first record. Hold the

`Shift` key and select the last record to select all of the records in the table. With all of the records selected, press the `Delete` key to delete the records. This approach is reasonably fast when there are many records to delete.

- In Visual Studio from SSOX, right-click the database and select **New Query**. A blank SQL file (`.sql`) opens. Paste the following command into the file: `DELETE FROM dbo.Movie;`. Select the green **Execute** triangle to execute the query or press `Ctrl`+`Shift`+`E` on the keyboard. In the **Message** tab, SSOX reports the number of rows affected, deleted in this case, by the query. You can save the query for later use if you wish. Otherwise, close the file without saving it. This approach is especially useful when the table is large and contains hundreds to thousands of records.

> **⊗ Caution**
>
> Use extreme caution when deleting records from a database. Deleting records is permanent without taking additional data loss mitigation steps. Production databases often provision automatic backup copies of data, either instantaneously as the database is modified or periodically, including with off-site copies and permanent physical storage of data.

After deleting all of the records, run the app. The initializer reseeds the database and includes the correct movie ratings for the `Rating` field based on the seeding code.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Migrations (EF Core documentation)
- Customize migration code

Legal

Mad Max, *The Road Warrior*, *Mad Max: Beyond Thunderdome*, *Mad Max: Fury Road*, and *Furiosa: A Mad Max Saga* are trademarks and copyrights of Warner Bros. Entertainment.

Next steps

Previous: Add Search | Next: Add interactivity

Build a Blazor movie database app (Part 8 - Add interactivity)

Article • 12/05/2024

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

This article is the eighth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

Up to this point in the tutorial, the entire app has been *enabled* for interactivity, but the app has only *adopted* interactivity in its `Counter` sample component. This part of the tutorial series explains how to adopt interactivity in the movie `Index` component.

> **ⓘ Important**
>
> Confirm that the app isn't running for the next steps.

Adopt interactivity

Interactivity means that a component has the capacity to process UI events via C# code, such as a button click. The 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. This tutorial adopts interactive server-side rendering (interactive SSR), also known generally as Interactive Server (`InteractiveServer`) rendering. Client-side rendering (CSR), which is inherently interactive, is covered in the Blazor reference documentation.

Interactive SSR enables a rich user experience like one would expect from a client app but without the need to create API endpoints to access server resources. UI interactions are handled by the server over a real-time SignalR connection with the browser. Page content for interactive pages is prerendered, where content on the server is initially generated and sent to the client without enabling event handlers for rendered controls.

With prerendering, 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.

Review the API in the `Program` file (`Program.cs`) that enables interactive SSR. Razor component services are added to the app to enable Razor components to render statically from the server (AddRazorComponents) and execute code with interactive SSR (AddInteractiveServerComponents):

```
C#

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

MapRazorComponents maps components defined in the root `App` component to the given .NET assembly and renders routable components, and AddInteractiveServerRenderMode configures the app's SignalR hub to support interactive SSR:

```
C#

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

Up to this point in the tutorial series, the calls to AddInteractiveServerComponents and AddInteractiveServerRenderMode weren't required for the movie pages because the app only adopted static SSR features for the movie components. This article explains how to adopt interactive features in the movie `Index` component.

When Blazor sets the type of rendering for a component, the rendering is referred to as the component's *render mode*. The following table shows the available render modes for rendering Razor components in a Blazor Web 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	✓

Name	Description	Render location	Interactive
Interactive Auto	Interactive SSR using Blazor Server initially and then CSR on subsequent visits after the Blazor bundle is downloaded.	Server, then client	✔

To apply a render mode to a component, the developer either uses the `@rendermode` directive or directive attribute on the component instance or on the component definition:

- The following example shows how to set the render mode on a component instance with the `@rendermode` directive attribute. The following example uses a hypothetical dialog (`Dialog`) component in a parent chat (`Chat`) component.

  Within the `Components/Pages/Chat.razor` file:

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

- The following example shows how to set the render mode on a component definition with the `@rendermode` directive. The following example shows setting the render mode in a hypothetical sales forecast (`SalesForecast`) component definition file (`.razor`).

  At the top of `Components/Pages/SalesForecast.razor` file:

  ```razor
  @page "/sales-forecast"
  @rendermode InteractiveServer
  ```

Using the preceding approaches, you can apply a render mode on a per-page/component basis. However, an entire app can adopt a single render mode via a root component that then by inheritance sets the render mode of every other component loaded. This is termed *global interactivity*, as opposed to *per-page/component interactivity*. Global interactivity is useful if most of the app requires interactive features. Global interactivity is usually applied via the `App` component, which is the root component of an app created from the Blazor Web App project template.

> ⓘ **Note**

More information on render modes and global interactivity is provided by Blazor's reference documentation. For the purposes of this tutorial, the app only adopts interactive SSR on a per-page/component basis. After the tutorial, you're free to use this app to study the other component render modes and the global interactivity location.

Open the movie `Index` component file (`Components/Pages/MoviePages/Index.razor`), and add the following `@rendermode` directive immediately after the `@page` directive to make the component interactive:

diff

```
@rendermode InteractiveServer
```

To see how making a component interactive enhances the user experience, let's provide three enhancements to the app in the following sections:

- Add pagination to the movie `QuickGrid` component.
- Make the `QuickGrid` component *sortable*.
- Replace the HTML form for filtering movies by title with C# code that:
  - Runs on the server.
  - Renders content interactively over the underlying SignalR connection.

Add pagination to the `QuickGrid`

The `QuickGrid` component can page data from the database.

Open the `Index` component (`Components/Pages/Movies/Index.razor`). Add a PaginationState instance to the `@code` block. Because the tutorial only uses five movie records for demonstration, set the ItemsPerPage to `2` items in order to demonstrate pagination. Normally, the number of items to display would be set to a higher value or set dynamically via a dropdown list.

C#

```
private PaginationState pagination = new PaginationState { ItemsPerPage = 2
};
```

Set the `QuickGrid` component's Pagination property to `pagination`:

diff

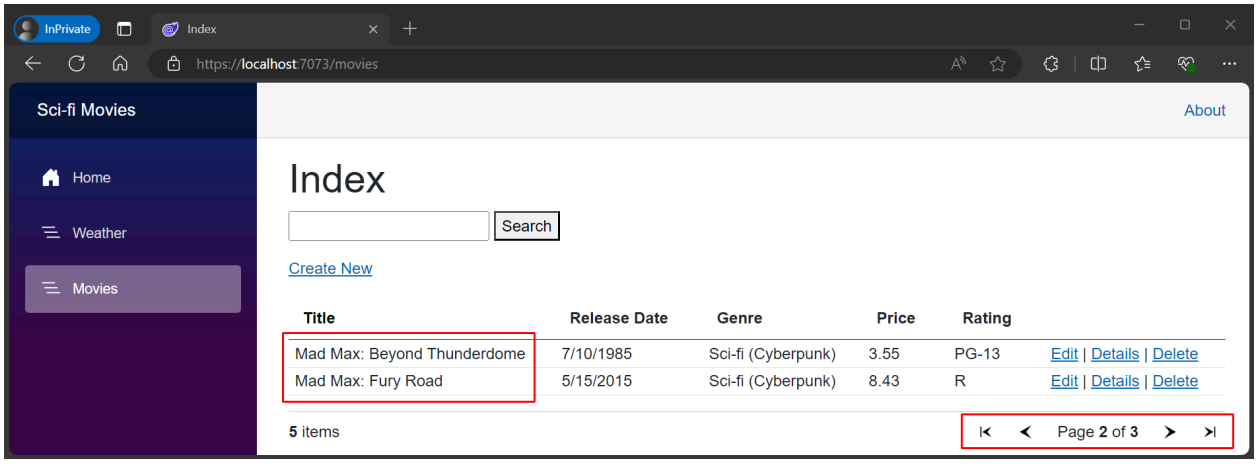
```
- <QuickGrid Class="table" Items="FilteredMovies">
+ <QuickGrid Class="table" Items="FilteredMovies" Pagination="pagination">
```

To provide a UI for pagination below the `QuickGrid` component, add a Paginator component below the `QuickGrid` component. Set the Paginator.State to `pagination`:

razor

```
<Paginator State="pagination" />
```

Run the app and navigate to the movies `Index` page. You can page through the movie items at two movies per page:



The component is *interactive*. The page doesn't reload for paging to occur. The paging is performed live over the SignalR connection between the browser and the server, where the paging operation is performed on the server with the rendered result sent back to the client for the browser to display.

Change ItemsPerPage to a more reasonable value, such as five items per page:

diff

```
- private PaginationState pagination = new PaginationState { ItemsPerPage =
2 };
+ private PaginationState pagination = new PaginationState { ItemsPerPage =
5 };
```

Sortable `QuickGrid`

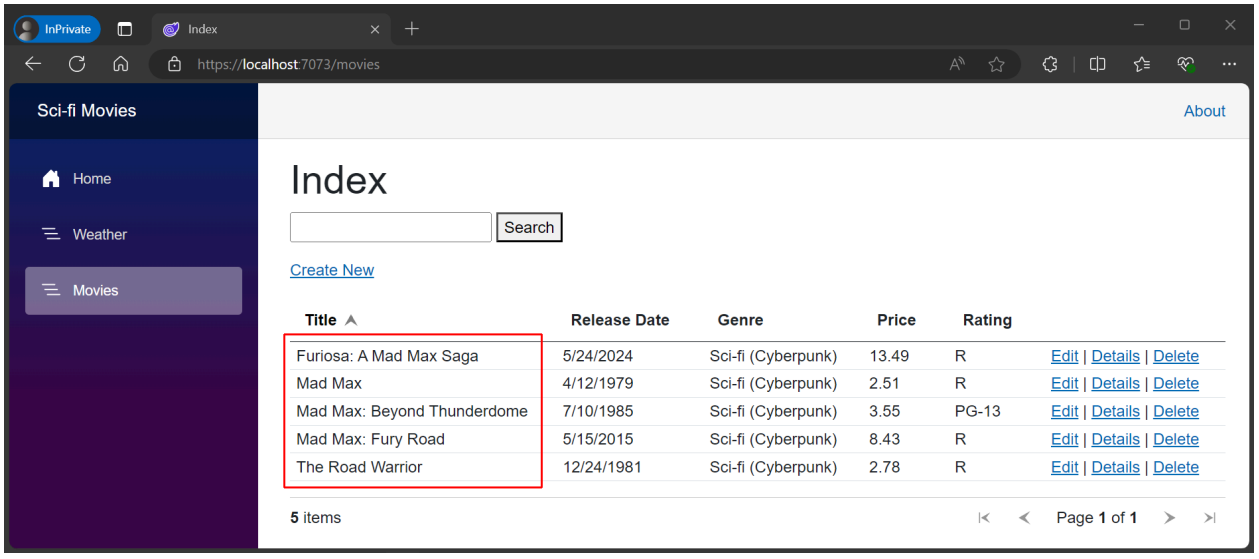
Open the `Index` component (`Components/Pages/Movies/Index.razor`).

Add `Sortable="true"` (Sortable) to the title PropertyColumn<TGridItem,TProp>:

diff

```
- <PropertyColumn Property="movie => movie.Title" />
+ <PropertyColumn Property="movie => movie.Title" Sortable="true" />
```

You can sort the `QuickGrid` by movie title by selecting the **Title** column. The page doesn't reload for sorting to occur. The sorting is performed live over the SignalR connection, where the sorting operation is performed on the server with the rendered result sent back to the client:



Use C# code and interactivity to search by title

In an earlier part of the tutorial series, the `Index` component was modified to allow the user to filter movies by title. This was accomplished by:

- Adding an HTML form that issues a GET request to the server with the user's title search string as a query string field-value pair (for example, `?titleFilter=road+warrior` if the user searches for "`road warrior`").
- Adding code to the component that obtains the title search string from the query string and uses it to filter the database's records.

The preceding approach is effective for a component that adopts static SSR, where the only interaction between the client and server is via HTTP requests. There was no live SignalR connection between the client and the server, and there was no way for the app on the server to process C# code *interactively* based on the user's actions in the component's UI and return content.

Now that the component is interactive, it can provide an improved user experience with Blazor features for binding and event handling.

Add a delegate event handler that the user can trigger to filter the database's movie records. The method uses the value of the `TitleFilter` property to perform the operation. If the user clears `TitleFilter` and searches, the method loads the entire movie list for display.

Delete the following lines from the `@code` block:

diff

```
- [SupplyParameterFromQuery]
- private string? TitleFilter { get; set; }

- private IQueryable<Movie> FilteredMovies =>
-     context.Movie.Where(m => m.Title!.Contains(TitleFilter ??
string.Empty));
```

Replace the deleted code with the following code:

C#

```
private string titleFilter = string.Empty;

private IQueryable<Movie> FilteredMovies =>
    context.Movie.Where(m => m.Title!.Contains(titleFilter));
```

Next, the component should bind the `titleFilter` field to an `<input>` element, so user input is stored in the `titleFilter` variable. Binding is achieved in Blazor with the `@bind` directive attribute.

Remove the HTML form from the component:

diff

```
- <form action="/movies" data-enhance>
-     <input type="search" name="titleFilter" />
-     <input type="submit" value="Search" />
- </form>
```

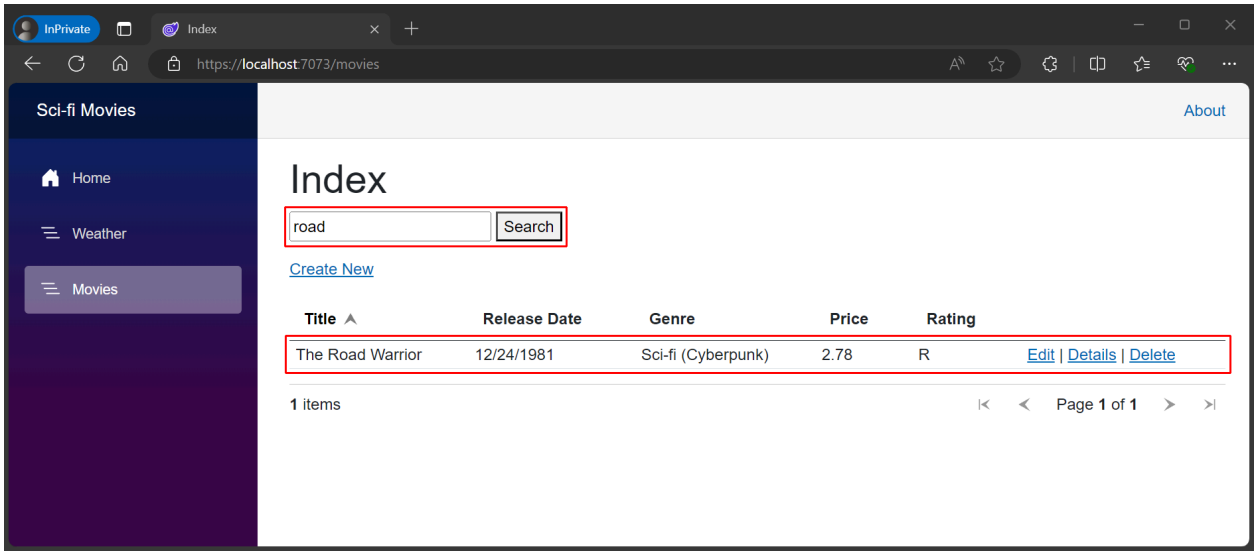
In its place, add the following Razor markup:

razor

```
<input type="search" @bind="titleFilter" @bind:event="oninput" />
```

`@bind:event="oninput"` performs binding for the HTML's `oninput` event, which fires when the `<input>` element's value is changed as a direct result of a user typing in the search box. The `QuickGrid` is bound to `FilteredMovies`. As `titleFilter` changes with the value of the search box, rerendering the `QuickGrid` bound to the `FilteredMovies` method filters movie entities based on the updated value of `titleFilter`.

Run the app, type "`road warrior`" into the search field and notice how the `QuickGrid` is filtered for each character entered until *The Road Warrior* movie is left when the search field reaches "`road `" ("road" followed by a space).



Filtering database records is performed on the server, and the server interactively sends back the HTML to display over the same SignalR connection. The page doesn't reload. The user feels like their interactions with the page are running code on the client. Actually, the code is running the server.

Instead of an HTML form, submitting a GET request in this scenario could've also used JavaScript to submit the request to the server, either using the Fetch API` or XMLHttpRequest API. In most cases, JavaScript can be replaced by using Blazor and C# in an interactive component.

Style the `QuickGrid` component

You can apply styles to the rendered `QuickGrid` component with a stylesheet isolated to the `Index` component using *CSS isolation*.

CSS isolation is applied by adding a stylesheet file using the file name format `{COMPONENT NAME}.razor.css`, where the `{COMPONENT NAME}` placeholder is the component name.

To apply styles to a child component, such as the `QuickGrid` component of the `Index` component, use the `::deep` pseudo-element.

In the `MoviePages` folder, add the following stylesheet for the `Index` component. Use `::deep` pseudo-elements to make the row height `3em` and vertically center the table cell content.

`Components/Pages/MoviePages/Index.razor.css`:

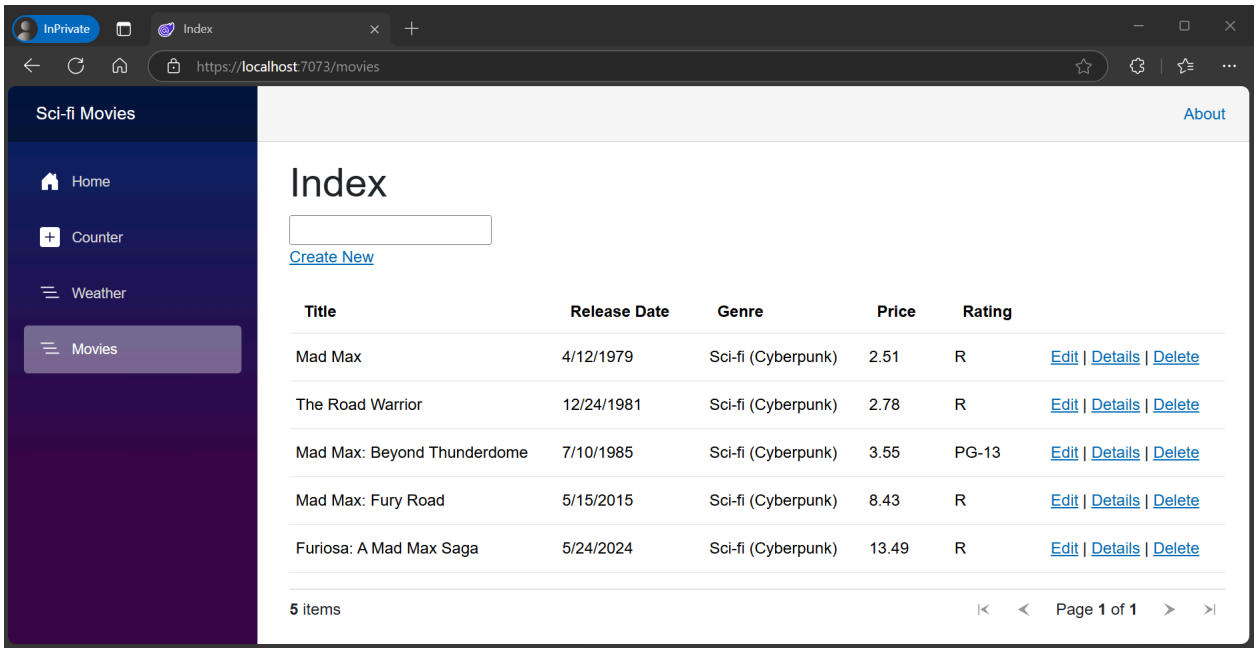
```css
::deep tr {
    height: 3em;
}

    ::deep tr > td {
        vertical-align: middle;
    }
```

The `::deep` pseudo-element only works with descendant elements, so the `QuickGrid` component must be wrapped with a `<div>` or some other block-level element in order to apply the styles to it.

In `Components/Pages/MoviePages/Index.razor`, place `<div>` tags around the `QuickGrid` component:

```diff
+ <div>
    <QuickGrid ...>
        ...
    </QuickGrid>
+ </div>
```

Blazor rewrites CSS selectors to match the markup rendered by the component. The rewritten CSS styles are bundled and produced as a static asset, so you don't need to take further action to apply the styles to the rendered `QuickGrid` component.



Clean up

When you're finished with the tutorial and delete the sample app from your local system, you can also delete the `BlazorWebAppMovies` database in Visual Studio's SQL Server Object Explorer (SSOX):

1. Access SSOX by selecting **View** > **SQL Server Object Explorer** from the menu bar.
2. Display the database list by selecting the triangles next to `SQL Server` > `(localdb)\MSSQLLocalDB` > `Databases`.
3. Right-click the `BlazorWebAppMovies` database in the list and select **Delete**.
4. Select the checkbox to close existing collections before selecting **OK**.

When deleting the database in SSOX, the database's physical files are removed from your Windows user folder.

Congratulations!

Congratulations on completing the tutorial series! We hope you enjoyed this tutorial on Blazor. Blazor offers many more features than we were able to cover in this series, and we invite you to explore the Blazor documentation, examples, and sample apps to learn more. *Happy coding with Blazor!*

Next steps

If you're new to Blazor, we recommend reading the following Blazor articles that cover important general information for Blazor development:

- ASP.NET Core Blazor
- ASP.NET Core Blazor supported platforms
- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- ASP.NET Core Blazor project structure
- ASP.NET Core Blazor fundamentals and the other *Fundamentals* node articles.
- ASP.NET Core Razor components and the other *Components* node articles.
- ASP.NET Core Blazor forms overview
- ASP.NET Core Blazor JavaScript interoperability (JS interop) and the other *JavaScript interop* articles.
- ASP.NET Core Blazor authentication and authorization
- Host and deploy ASP.NET Core Blazor
- ASP.NET Core Blazor with Entity Framework Core (EF Core) covers concurrency with EF Core in Blazor apps.

In the documentation website's sidebar navigation, articles are organized by subject matter and laid out in roughly in a general-to-specific or basic-to-complex order. The best approach when starting to learn about Blazor is to read down the table of contents from top to bottom.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Previous: Add a new field

Use ASP.NET Core SignalR with Blazor

Article • 06/21/2024

> **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 tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected

- Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

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

```
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
```

```
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> ⓘ Note
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

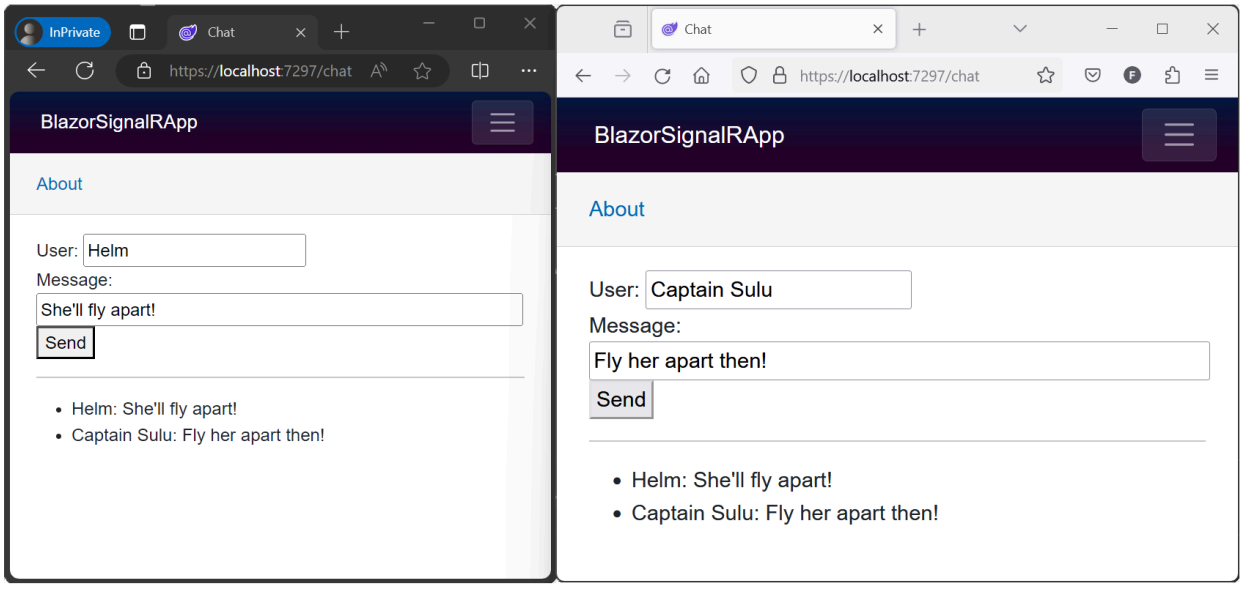
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor Hybrid

Article • 12/11/2024

> **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 ASP.NET Core Blazor Hybrid, a way to build interactive client-side web UI with .NET in an ASP.NET Core app.

Use *Blazor Hybrid* to blend desktop and mobile native client frameworks with .NET and Blazor.

In a Blazor Hybrid app, Razor components run natively on the device. Components render to an embedded Web View control through a local interop channel. Components don't run in the browser, and WebAssembly isn't involved. Razor components load and execute code quickly, and components have full access to the native capabilities of the device through the .NET platform. Component styles rendered in a Web View are platform dependent and may require you to account for rendering differences across platforms using custom stylesheets.

Blazor Hybrid articles cover subjects pertaining to integrating Razor components into native client frameworks.

Blazor Hybrid apps with .NET MAUI

Blazor Hybrid support is built into the .NET Multi-platform App UI (.NET MAUI) framework. .NET MAUI includes the BlazorWebView control that permits rendering Razor components into an embedded Web View. By using .NET MAUI and Blazor together, you can reuse one set of web UI components across mobile, desktop, and web.

Blazor Hybrid apps with WPF and Windows Forms

Blazor Hybrid apps can be built with Windows Presentation Foundation (WPF) and Windows Forms. Blazor provides `BlazorWebView` controls for both of these frameworks (WPF BlazorWebView, Windows Forms BlazorWebView). Razor components run natively in the Windows desktop and render to an embedded Web View. Using Blazor in WPF and Windows Forms enables you to add new UI to your existing Windows desktop apps that can be reused across platforms with .NET MAUI or on the web.

Web View configuration

Blazor Hybrid exposes the underlying Web View configuration for different platforms through events of the `BlazorWebView` control:

- `BlazorWebViewInitializing` provides access to the settings used to create the Web View on each platform, if settings are available.
- `BlazorWebViewInitialized` provides access to the Web View to allow further configuration of the settings.

Use the preferred patterns on each platform to attach event handlers to the events to execute your custom code.

API documentation:

- .NET MAUI
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized
- WPF
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized
- Windows Forms
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized

Unhandled exceptions in Windows Forms and WPF apps

This section only applies to Windows Forms and WPF Blazor Hybrid apps.

Create a callback for `UnhandledException` on the System.AppDomain.CurrentDomain property. The following example uses a compiler directive to display a MessageBox that either alerts the user that an error has occurred or shows the error information to the developer. Log the error information in `error.ExceptionObject`.

C#

```
AppDomain.CurrentDomain.UnhandledException += (sender, error) =>
{
#if DEBUG
    MessageBox.Show(text: error.ExceptionObject.ToString(), caption:
"Error");
#else
    MessageBox.Show(text: "An error has occurred.", caption: "Error");
#endif

    // Log the error information (error.ExceptionObject)
};
```

Globalization and localization

This section only applies to .NET MAUI Blazor Hybrid apps.

.NET MAUI configures the CurrentCulture and CurrentUICulture based on the device's ambient information.

IStringLocalizer and other API in the Microsoft.Extensions.Localization namespace generally work as expected, along with globalization formatting, parsing, and binding that relies on the user's culture.

When dynamically changing the app culture at runtime, the app must be reloaded to reflect the change in culture, which takes care of rerendering the root component and passing the new culture to rerendered child components.

.NET's resource system supports embedding localized images (as blobs) into an app, but Blazor Hybrid can't display the embedded images in Razor components at this time. Even if a user reads an image's bytes into a Stream using ResourceManager, the framework doesn't currently support rendering the retrieved image in a Razor component.

For more information, see the following resources:

- Localization (.NET MAUI documentation)
- Blazor Image component to display images that are not accessible through HTTP endpoints (dotnet/aspnetcore #25274)

Access scoped services from native UI

BlazorWebView has a TryDispatchAsync method that calls a specified `Action<ServiceProvider>` asynchronously and passes in the scoped services available in Razor components. This enables code from the native UI to access scoped services such as NavigationManager:

C#

```
private async void MyMauiButtonHandler(object sender, EventArgs e)
{
    var wasDispatchCalled = await _blazorWebView.TryDispatchAsync(sp =>
    {
        var navMan = sp.GetRequiredService<NavigationManager>();
        navMan.CallSomeNavigationApi(...);
    });

    if (!wasDispatchCalled)
    {
        ...
    }
}
```

When `wasDispatchCalled` is `false`, consider what to do if the call wasn't dispatched. Generally, the dispatch shouldn't fail. If it fails, OS resources might be exhausted. If resources are exhausted, consider logging a message, throwing an exception, and perhaps alerting the user.

Additional resources

- ASP.NET Core Blazor Hybrid tutorials
- .NET Multi-platform App UI (.NET MAUI)
- Windows Presentation Foundation (WPF)
- Windows Forms

ASP.NET Core Blazor Hybrid tutorials

Article • 11/18/2024

> **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**.

The following tutorials provide a basic working experience for building a Blazor Hybrid app:

- Build a .NET MAUI Blazor Hybrid app
- Build a .NET MAUI Blazor Hybrid app with a Blazor Web App
- Build a Windows Forms Blazor app
- Build a Windows Presentation Foundation (WPF) Blazor app

For an overview of Blazor and reference articles, see ASP.NET Core Blazor and the articles that follow it in the table of contents.

Build a .NET MAUI Blazor Hybrid app

Article • 02/13/2024

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

This tutorial shows you how to build and run a .NET MAUI Blazor Hybrid app. You learn how to:

- ✓ Create a .NET MAUI Blazor Hybrid app project in Visual Studio
- ✓ Run the app on Windows
- ✓ Run the app on an emulated mobile device in the Android Emulator

Prerequisites

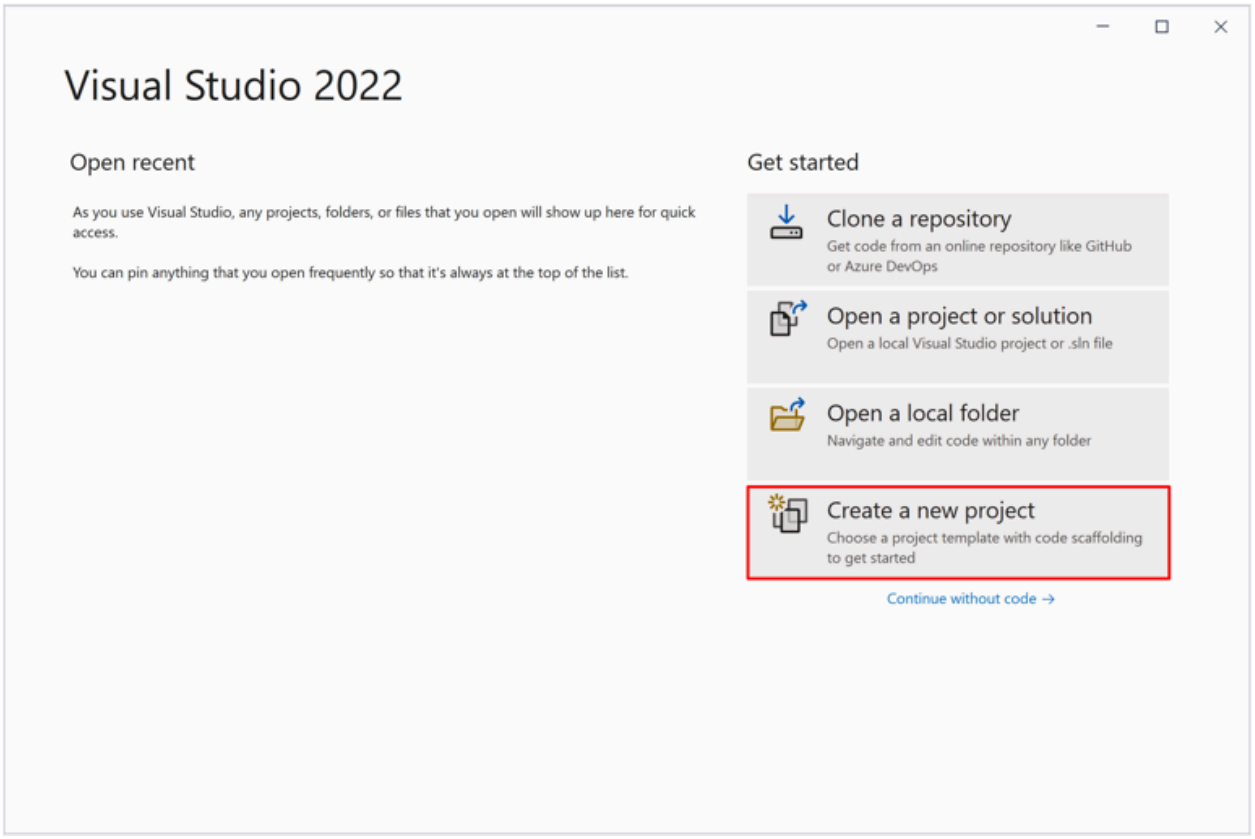
- Supported platforms (.NET MAUI documentation)
- Visual Studio with the **.NET Multi-platform App UI development** workload.
- Microsoft Edge WebView2: WebView2 is required on Windows when running a native app. When developing .NET MAUI Blazor Hybrid apps and only running them in Visual Studio's emulators, WebView2 isn't required.
- Enable hardware acceleration to improve the performance of the Android emulator.

For more information on prerequisites and installing software for this tutorial, see the following resources in the .NET MAUI documentation:

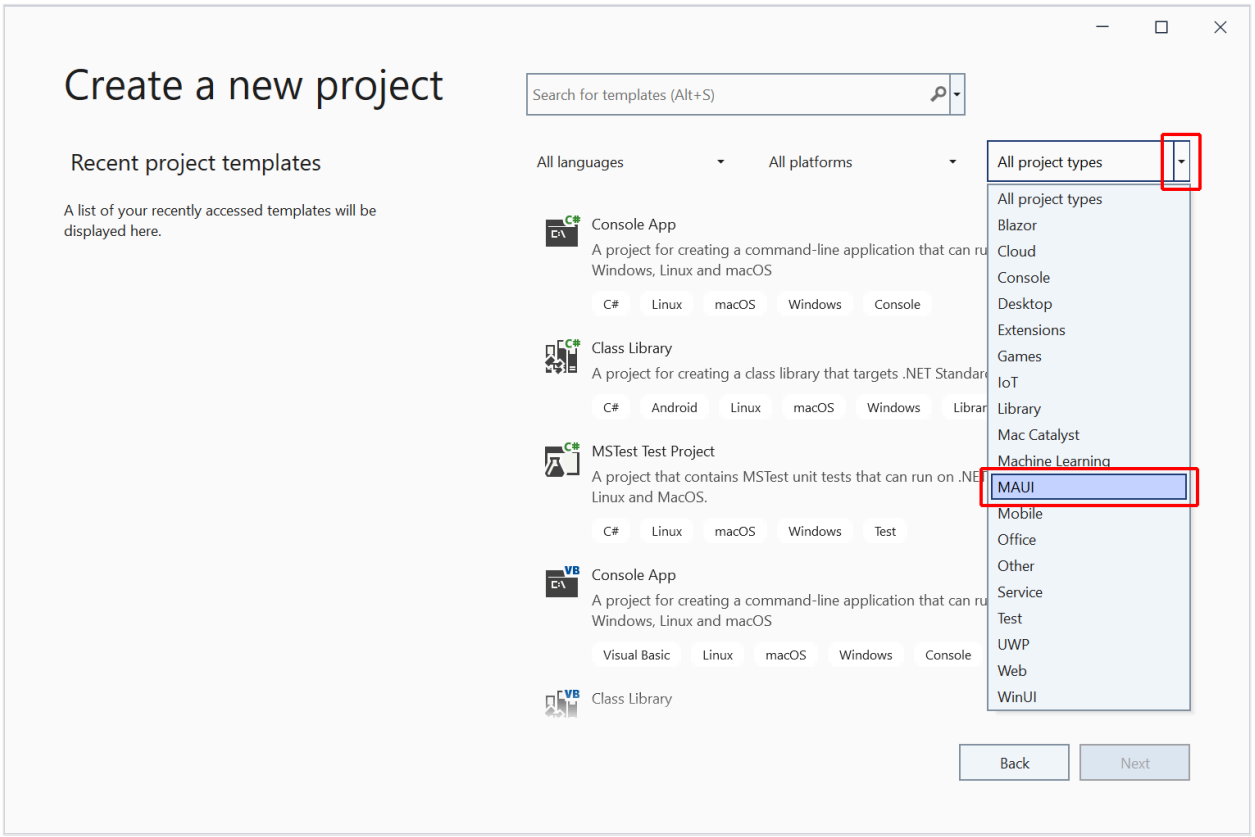
- Supported platforms for .NET MAUI apps
- Installation (Visual Studio)

Create a .NET MAUI Blazor Hybrid app

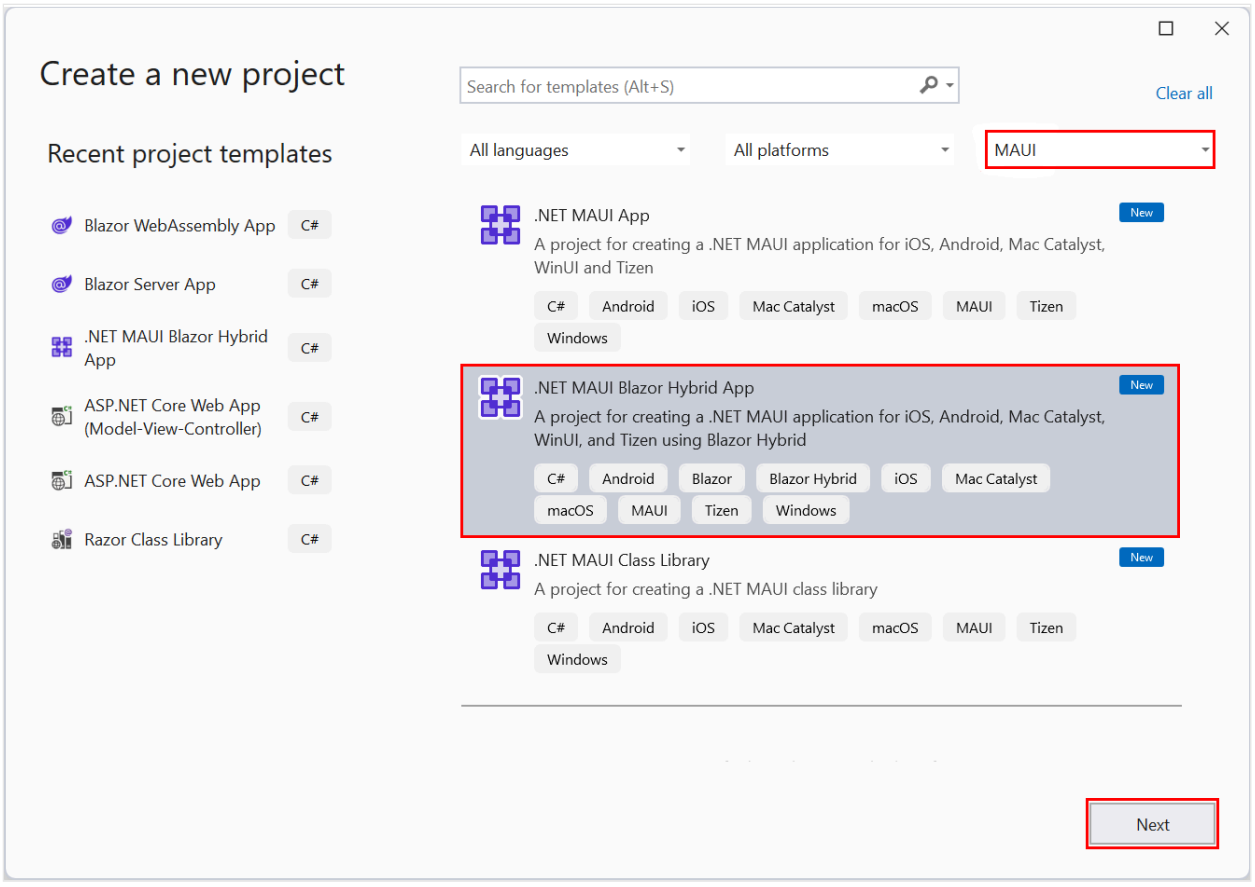
Launch Visual Studio. In the **Start Window**, select **Create a new project**:



In the **Create a new project** window, use the **Project type** dropdown to filter **MAUI** templates:



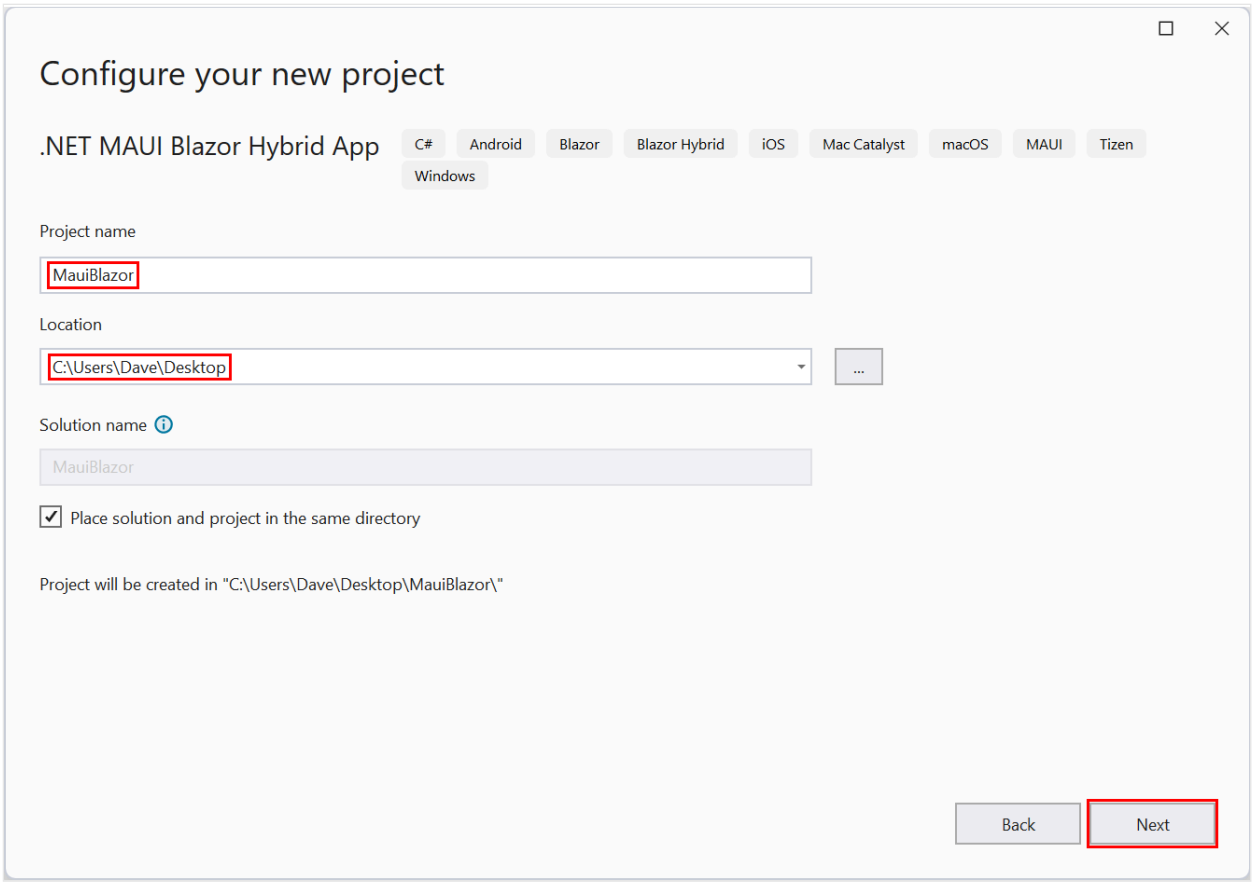
Select the **.NET MAUI Blazor Hybrid App** template and then select the **Next** button:



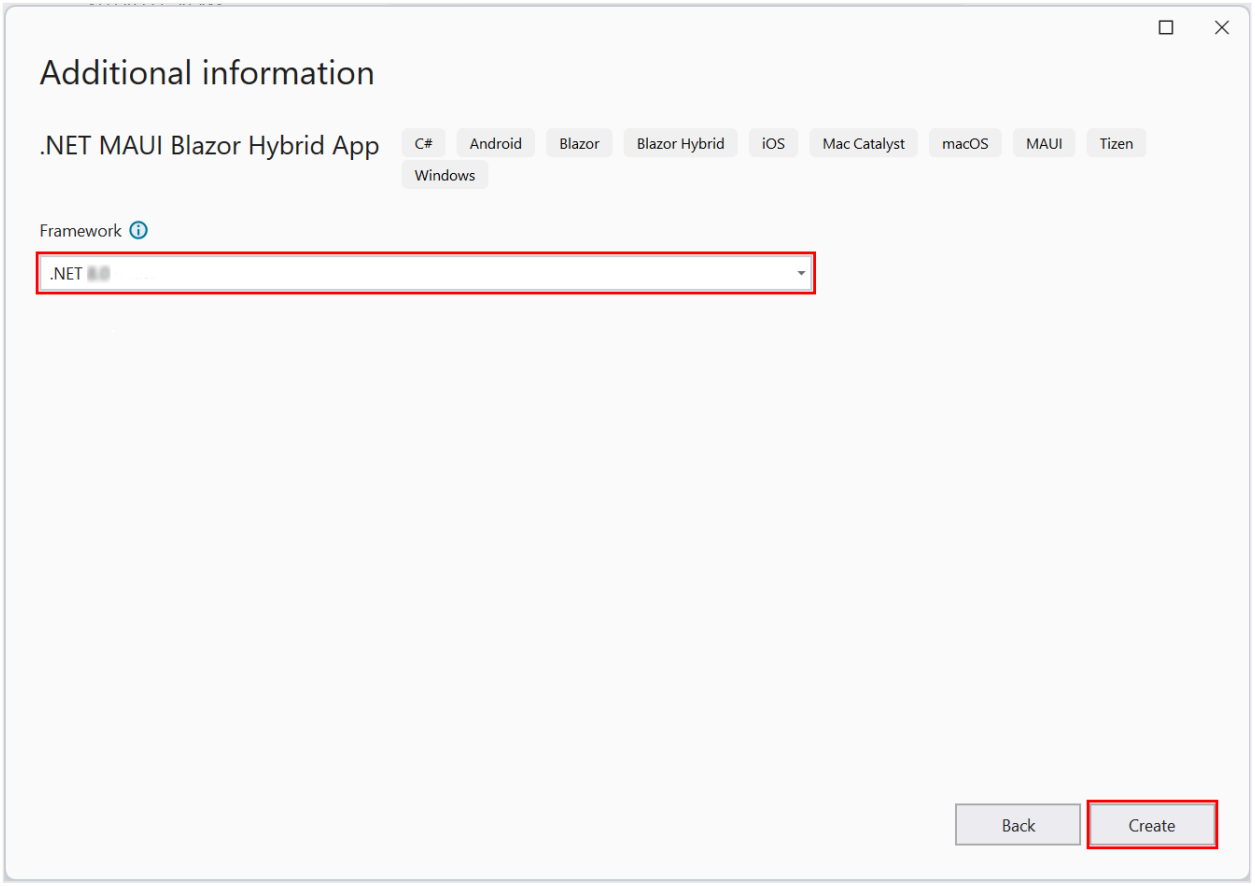
> **Note**
>
> In .NET 7 or earlier, the template is named **.NET MAUI Blazor App**.

In the **Configure your new project** dialog:

- Set the **Project name** to **MauiBlazor**.
- Choose a suitable location for the project.
- Select the **Next** button.

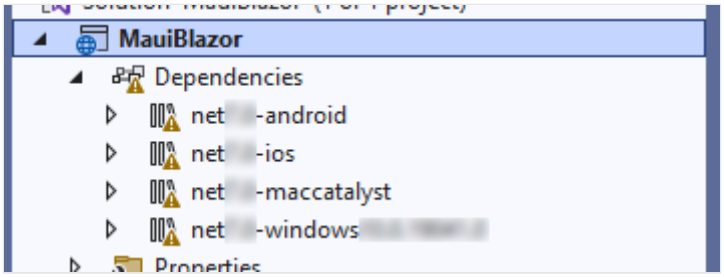


In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:

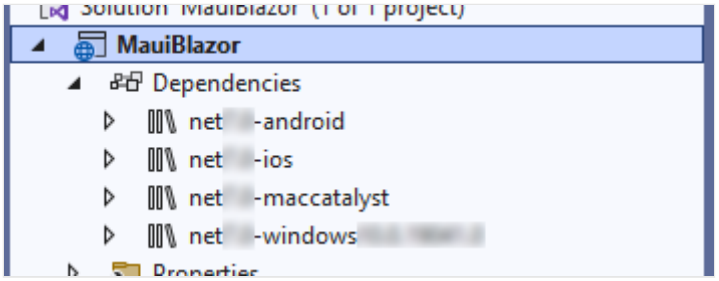


Wait for Visual Studio to create the project and restore the project's dependencies. Monitor the progress in **Solution Explorer** by opening the **Dependencies** entry.

Dependencies restoring:

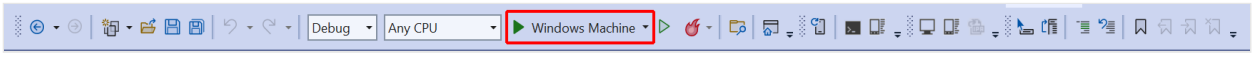


Dependencies restored:



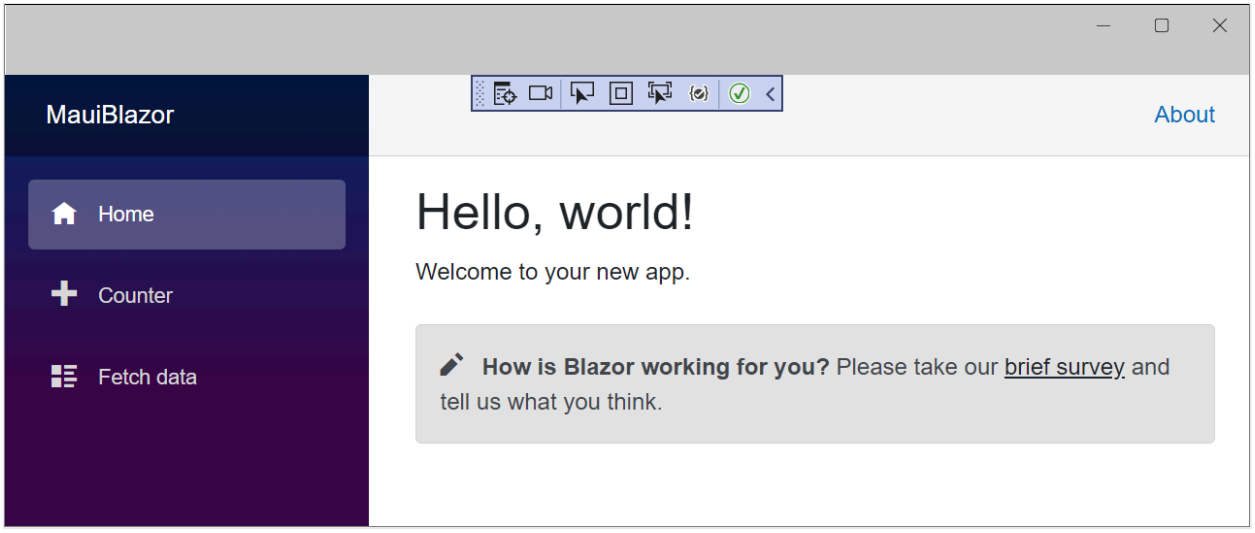
Run the app on Windows

In the Visual Studio toolbar, select the **Windows Machine** button to build and start the project:



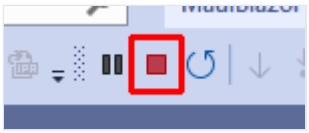
If Developer Mode isn't enabled, you're prompted to enable it in **Settings** > **For developers** > **Developer Mode** (Windows 10) or **Settings** > **Privacy & security** > **For developers** > **Developer Mode** (Windows 11). Set the switch to **On**.

The app running as a Windows desktop app:

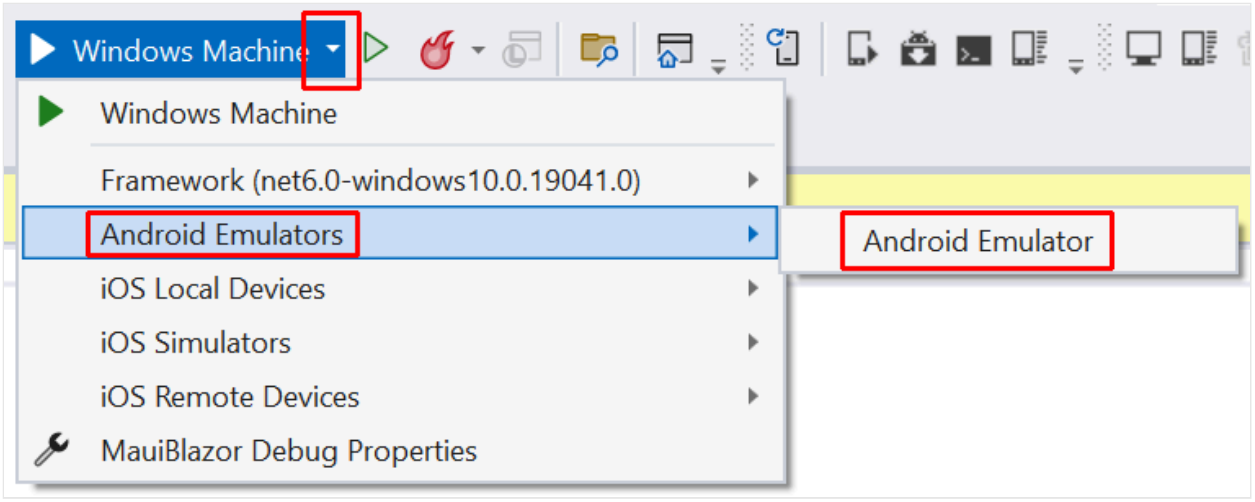


Run the app in the Android Emulator

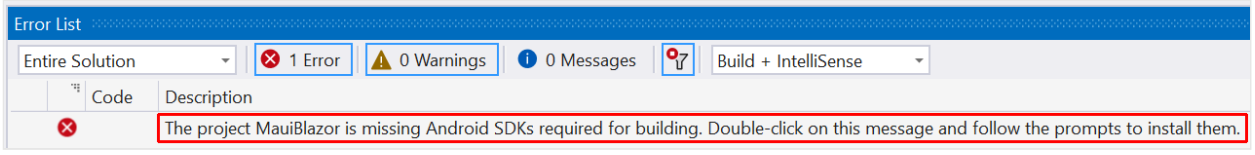
If you followed the guidance in the Run the app on Windows section, select the **Stop Debugging** button in the toolbar to stop the running Windows app:



In the Visual Studio toolbar, select the start configuration dropdown button. Select **Android Emulators** > **Android Emulator**:

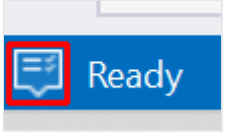


Android SDKs are required to build apps for Android. In the **Error List** panel, a message appears asking you to double-click the message to install the required Android SDKs:

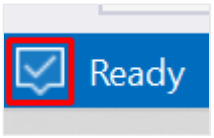


The **Android SDK License Acceptance** window appears, select the **Accept** button for each license that appears. An additional window appears for the **Android Emulator** and **SDK Patch Applier** licenses. Select the **Accept** button.

Wait for Visual Studio to download the Android SDK and Android Emulator. You can track the progress by selecting the background tasks indicator in the lower-left corner of the Visual Studio UI:



The indicator shows a checkmark when the background tasks are complete:



In the toolbar, select the **Android Emulator** button:

In the **Create a Default Android Device** window, select the **Create** button:

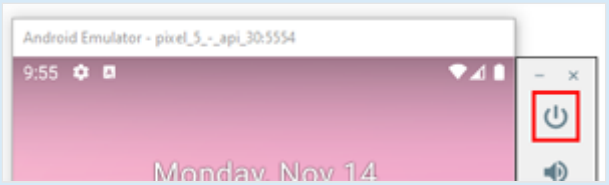
Wait for Visual Studio to download, unzip, and create an Android Emulator. When the Android phone emulator is ready, select the **Start** button.

> ⓘ **Note**
>
> Enable hardware acceleration to improve the performance of the Android emulator.

Close the **Android Device Manager** window. Wait until the emulated phone window appears, the Android OS loads, and the home screen appears.

> ⓘ **Important**
>
> The emulated phone must be powered on with the Android OS loaded in order to load and run the app in the debugger. If the emulated phone isn't running, turn on

the phone using either the `Ctrl`+`P` keyboard shortcut or by selecting the **Power** button in the UI:



In the Visual Studio toolbar, select the **Pixel 5 - {VERSION}** button to build and run the project, where the `{VERSION}` placeholder is the Android version. In the following example, the Android version is **API 30 (Android 11.0 - API 30)**, and a later version appears depending on the Android SDK installed:

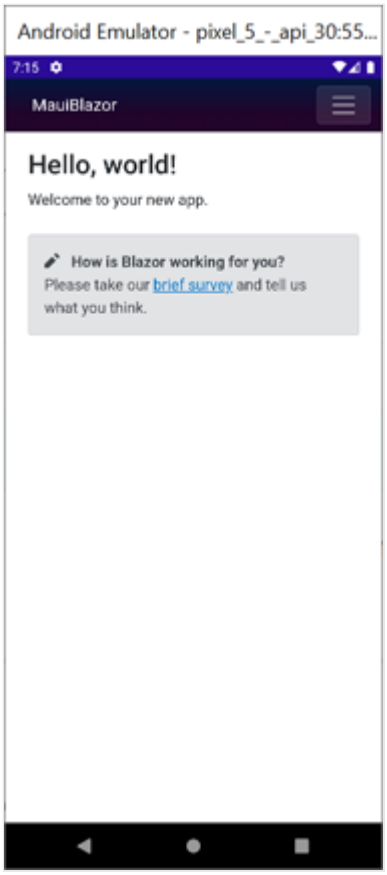
Visual Studio builds the project and deploys the app to the emulator.

Starting the emulator, loading the emulated phone and OS, and deploying and running the app can take several minutes depending on the speed of the system and whether or not hardware acceleration is enabled. You can monitor the progress of the deployment by inspecting Visual Studio's status bar at the bottom of the UI. The **Ready** indicator receives a checkmark and the emulator's deployment and app loading indicators disappear when the app is running:

During deployment:

During app startup:

The app running in the Android Emulator:



Next steps

In this tutorial, you learned how to:

- ✓ Create a .NET MAUI Blazor Hybrid app project in Visual Studio
- ✓ Run the app on Windows
- ✓ Run the app on an emulated mobile device in the Android Emulator

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

Build a .NET MAUI Blazor Hybrid app with a Blazor Web App

Article • 10/18/2024

This article shows you how to build a .NET MAUI Blazor Hybrid app with a Blazor Web App that uses a shared user interface via a Razor class library (RCL).

Prerequisites and preliminary steps

For prerequisites and preliminary steps, see Build a .NET MAUI Blazor Hybrid app. We recommend using the .NET MAUI Blazor Hybrid tutorial to set up your local system for .NET MAUI development before using the guidance in this article.

.NET MAUI Blazor Hybrid and Web App solution template

The .NET MAUI Blazor Hybrid and Web App solution template sets up a solution that targets Android, iOS, Mac, Windows and Web that reuses UI. You can choose a Blazor interactive render mode for the web app and it creates the appropriate projects for the app, including a Blazor Web App and a .NET MAUI Blazor Hybrid app. A shared Razor class library (RCL) maintains the Razor components for the app's UI. The template also provides sample code to show you how to use dependency injection to provide different interface implementations for the Blazor Hybrid and Blazor Web App, which is covered in the Using interfaces to support different device implementations section of this article.

If you haven't already installed the .NET MAUI workload, install it now. The .NET MAUI workload provides the project template:

.NET CLI

```
dotnet workload install maui
```

Create a solution from the project template with the following .NET CLI command:

.NET CLI

```
dotnet new maui-blazor-web -o MauiBlazorWeb -I Server
```

In the preceding command:

- The `-o|--output` option creates a new folder for the app named `MauiBlazorWeb`.
- The `-I|--InteractivityPlatform` option sets the interactivity render mode to Interactive Server (`InteractiveServer`). All three interactive Blazor render modes (`Server`, `WebAssembly`, and `Auto`) are supported by the project template. For more information, see the Use Blazor render modes section.

The app automatically adopts global interactivity, which is important because MAUI apps always run interactively and throw errors on Razor component pages that explicitly specify a render mode. For more information, see BlazorWebView needs a way to enable overriding ResolveComponentForRenderMode (dotnet/aspnetcore #51235).

Use Blazor render modes

Use the guidance in one of the following subsections that matches your app's specifications for applying Blazor render modes in the Blazor Web App but ignore the render mode assignments in the MAUI project.

Render mode specification subsections:

- Global Server interactivity
- Global Auto or WebAssembly interactivity

Global Server interactivity

- Interactive render mode: **Server**
- Interactivity location: **Global**
- Solution projects
  - MAUI (`MauiBlazorWeb`)
  - Blazor Web App (`MauiBlazorWeb.Web`)
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components without setting render modes in each component.

Project references: `MauiBlazorWeb` and `MauiBlazorWeb.Web` have a project reference to `MauiBlazorWeb.Shared`.

Global Auto or WebAssembly interactivity

- Interactive render mode: **Auto** or **WebAssembly**
- Interactivity location: **Global**

- Solution projects
  - MAUI (`MauiBlazorWeb`)
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components without setting render modes in each component.

Project references:

- `MauiBlazorWeb`, `MauiBlazorWeb.Web`, and `MauiBlazorWeb.Web.Client` projects have a project reference to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.

Per-page/component Server interactivity

- Interactive render mode: **Server**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb`): Calls `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` in `MauiProgram.cs`.
  - Blazor Web App (`MauiBlazorWeb.Web`): Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components that set the `InteractiveServer` render mode in each component.

`MauiBlazorWeb` and `MauiBlazorWeb.Web` have a project reference to `MauiBlazorWeb.Shared`.

Add the following `InteractiveRenderSettings` class to the RCL. The class properties are used to set component render modes.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI

project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs`:

```
C#

using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

```
C#

InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

```
diff

- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

> **Note**
>
> The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by **RenderMode** in the Blazor Web App's `_Import` file.

Per-page/component Auto interactivity

- Interactive render mode: **Auto**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb`): Calls `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` in `MauiProgram.cs`.
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`: Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components that set the `InteractiveAuto` render mode in each component.

Project references:

- `MauiBlazorWeb`, `MauiBlazorWeb.Web`, and `MauiBlazorWeb.Web.Client` have a project reference to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.

Add the following `InteractiveRenderSettings` class is added to the RCL. The class properties are used to set component render modes.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI

project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

C#

```
InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared.Client` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

diff

```
- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

> **Note**
>
> The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by **RenderMode** in the Blazor Web App's `_Import` file.

Per-page/component WebAssembly interactivity

- Interactive render mode: **WebAssembly**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb`)
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`: Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCLs
    - `MauiBlazorWeb.Shared`
    - `MauiBlazorWeb.Shared.Client`: Contains the shared Razor components that set the `InteractiveWebAssembly` render mode in each component. The `.Shared.Client` RCL is maintained separately from the `.Shared` RCL because the app should maintain the components that are required to run on WebAssembly separately from the components that run on server and that stay on the server.

Project references:

- `MauiBlazorWeb` and `MauiBlazorWeb.Web` have project references to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.
- `MauiBlazorWeb.Web.Client` and `MauiBlazorWeb.Shared` have a project reference to `MauiBlazorWeb.Shared.Client`.

Add the following AdditionalAssemblies parameter to the `Router` component instance for the `MauiBlazorWeb.Shared.Client` project assembly (via its `_Imports` file) in the `MauiBlazorWeb.Shared` project's `Routes.razor` file:

```
```

razor

```
<Router AppAssembly="@typeof(Routes).Assembly"
        AdditionalAssemblies="new [] {
typeof(MauiBlazorWeb.Shared.Client._Imports).Assembly }">
    <Found Context="routeData">
        <RouteView RouteData="@routeData"
DefaultLayout="@typeof(Components.Layout.MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
</Router>
```

Add the `MauiBlazorWeb.Shared.Client` project assembly (via its `_Imports` file) with the following AddAdditionalAssemblies call in the `MauiBlazorWeb.Web` project's `Program.cs` file:

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(MauiBlazorWeb.Shared._Imports).Assembly)

.AddAdditionalAssemblies(typeof(MauiBlazorWeb.Shared.Client._Imports).Assemb
ly);
```

Add the following `InteractiveRenderSettings` class is added to the `.Shared.Client` RCL. The class properties are used to set component render modes for server-based components.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs` (`.Shared.Client` RCL):

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
```

```
namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

A slightly different version of the `InteractiveRenderSettings` class is added to the `.Shared` RCL. In the class added to the `.Shared` RCL, `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` of the `.Shared.Client` RCL is called. This ensures that the render mode of WebAssembly components rendered on the MAUI client are unassigned (`null`) because they're interactive by default on the native client.

`InteractiveRenderSettings.cs` (`.Shared` RCL):

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared
{
    public static class InteractiveRenderSettings
    {
        public static IComponentRenderMode? InteractiveServer { get; set; }
=
            RenderMode.InteractiveServer;
        public static IComponentRenderMode? InteractiveAuto { get; set; } =
            RenderMode.InteractiveAuto;
        public static IComponentRenderMode? InteractiveWebAssembly { get;
set; } =
            RenderMode.InteractiveWebAssembly;

        public static void ConfigureBlazorHybridRenderModes()
        {
            InteractiveServer = null;
            InteractiveAuto = null;
```

```
            InteractiveWebAssembly = null;
            MauiBlazorWeb.Shared.Client.InteractiveRenderSettings
                .ConfigureBlazorHybridRenderModes();
        }
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

C#

```
InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared.Client` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

diff

```
- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

> **Note**
>
> The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by RenderMode in the Blazor Web App's `_Import` file.

Using interfaces to support different device implementations

The following example demonstrates how to use an interface to call into different implementations across the web app and the native (MAUI) app. The following example creates a component that displays the device form factor. Use the MAUI abstraction layer for native apps and provide an implementation for the web app.

In the Razor class library (RCL), a `Services` folder contains an `IFormFactor` interface.

`Services/IFormFactor.cs`:

C#

```
namespace MauiBlazorWeb.Shared.Services
{
    public interface IFormFactor
    {
        public string GetFormFactor();
        public string GetPlatform();
    }
}
```

The `Home` component (`Components/Pages/Home.razor`) of the RCL displays the form factor and platform.

`Components/Pages/Home.razor`:

razor

```
@page "/"
@using MauiBlazorWeb.Shared.Services
@inject IFormFactor FormFactor

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app running on <em>@factor</em> using
<em>@platform</em>.

@code {
    private string factor => FormFactor.GetFormFactor();
    private string platform => FormFactor.GetPlatform();
}
```

The web and native apps contain the implementations for `IFormFactor`.

In the Blazor Web App, a folder named `Services` contains the following `FormFactor.cs` file with the `FormFactor` implementation for web app use.

`Services/FormFactor.cs` (`MauiBlazorWeb.Web` project):

C#

```
using MauiBlazorWeb.Shared.Services;

namespace MauiBlazorWeb.Web.Services
{
```

```
    public class FormFactor : IFormFactor
    {
        public string GetFormFactor()
        {
            return "Web";
        }

        public string GetPlatform()
        {
            return Environment.OSVersion.ToString();
        }
    }
}
```

In the MAUI project, a folder named `Services` contains the following `FormFactor.cs` file with the `FormFactor` implementation for native use. The MAUI abstractions layer is used to write code that works on all native device platforms.

`Services/FormFactor.cs` (`MauiBlazorWeb` project):

C#

```
using MauiBlazorWeb.Shared.Services;

namespace MauiBlazorWeb.Services
{
    public class FormFactor : IFormFactor
    {
        public string GetFormFactor()
        {
            return DeviceInfo.Idiom.ToString();
        }

        public string GetPlatform()
        {
            return DeviceInfo.Platform.ToString() + " - " +
DeviceInfo.VersionString;
        }
    }
}
```

Dependency injection is used to obtain the implementations of these services.

In the MAUI project, the `MauiProgram.cs` file has following `using` statements at the top of the file:

C#

```
using MauiBlazorWeb.Services;
```

```
using MauiBlazorWeb.Shared.Interfaces;
```

Immediately before the call to `builder.Build()`, `FormFactor` is registered to add device-specific services used by the RCL:

```
C#

builder.Services.AddSingleton<IFormFactor, FormFactor>();
```

In the Blazor Web App, the `Program` file has the following `using` statements at the top of the file:

```
C#

using MauiBlazorWeb.Shared.Interfaces;
using MauiBlazorWeb.Web.Services;
```

Immediately before the call to `builder.Build()`, `FormFactor` is registered to add device-specific services used by the Blazor Web App:

```
C#

builder.Services.AddScoped<IFormFactor, FormFactor>();
```

If the solution also targets WebAssembly via a `.Web.Client` project, an implementation of the preceding API is also required in the `.Web.Client` project.

You can also use compiler preprocessor directives in your RCL to implement different UI depending on the device the app is running on. For this scenario, the app must multi-target the RCL just like the MAUI app does. For an example, see the BethMassi/BethTimeUntil GitHub repository.

Additional resources

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor render modes
- Reuse Razor components in ASP.NET Core Blazor Hybrid apps

Build a Windows Forms Blazor app

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 tutorial shows you how to build and run a Windows Forms Blazor app. You learn how to:

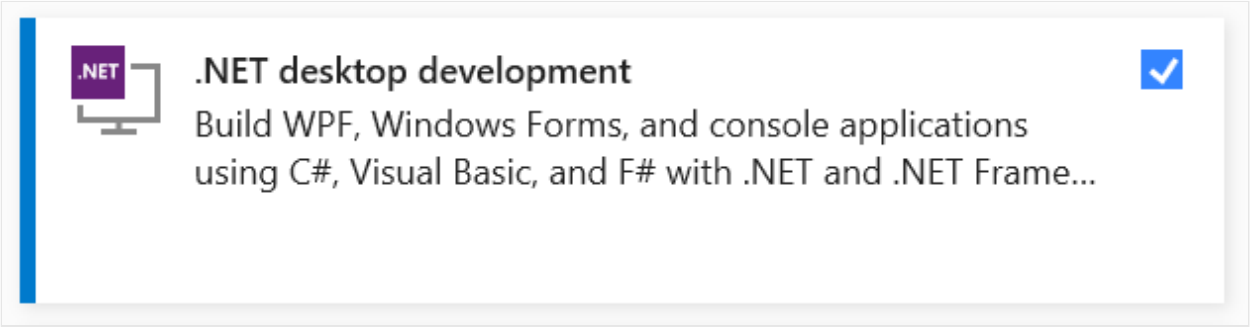
- ✓ Create a Windows Forms Blazor app project
- ✓ Run the app on Windows

Prerequisites

- Supported platforms (Windows Forms documentation)
- Visual Studio 2022 with the **.NET desktop development** workload

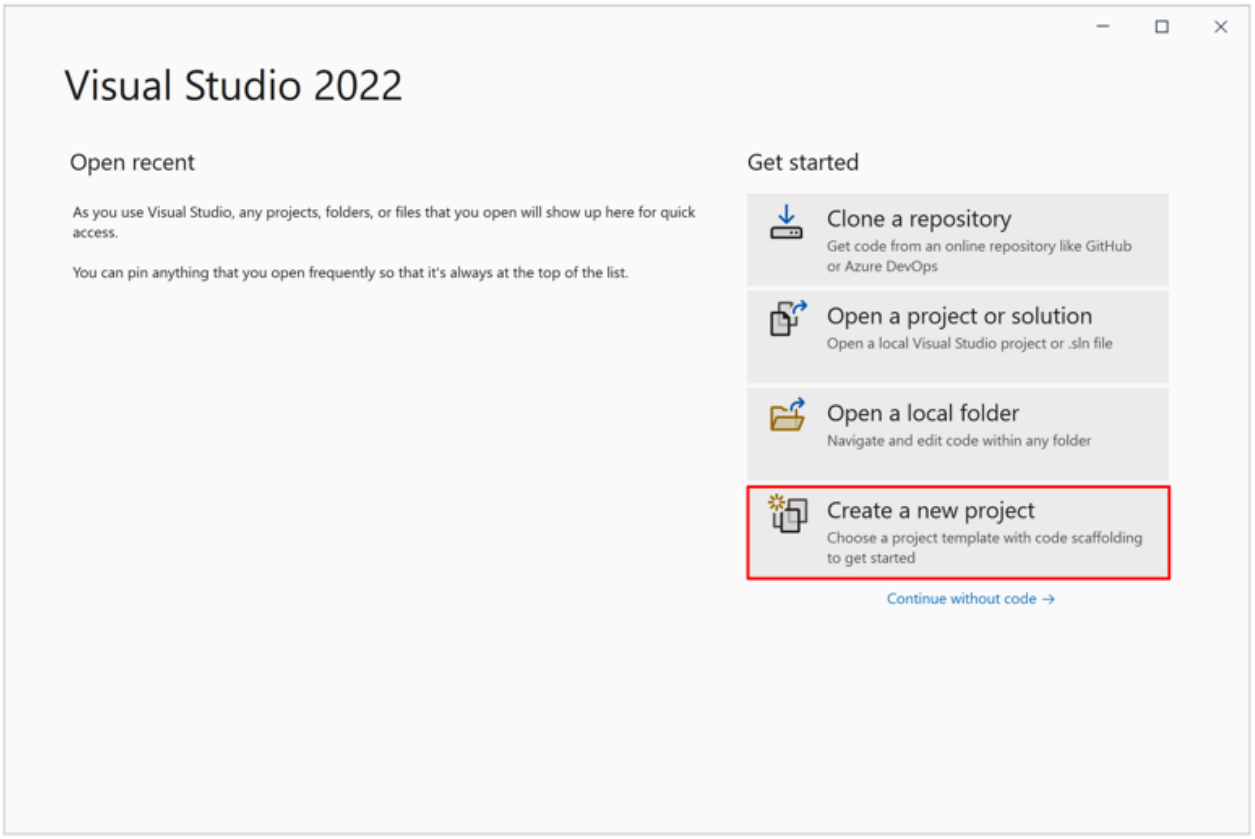
Visual Studio workload

If the **.NET desktop development** workload isn't installed, use the Visual Studio installer to install the workload. For more information, see Modify Visual Studio workloads, components, and language packs.

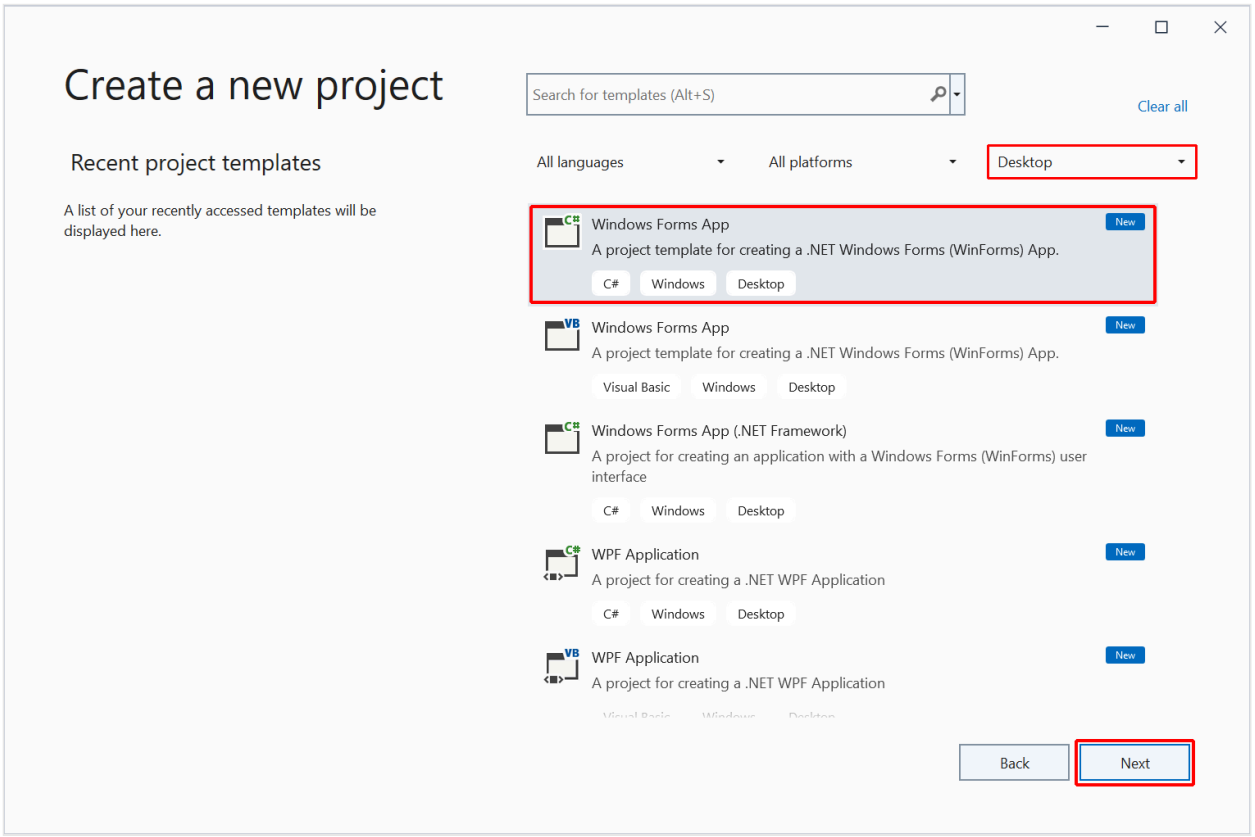


Create a Windows Forms Blazor project

Launch Visual Studio. In the **Start Window**, select **Create a new project**:



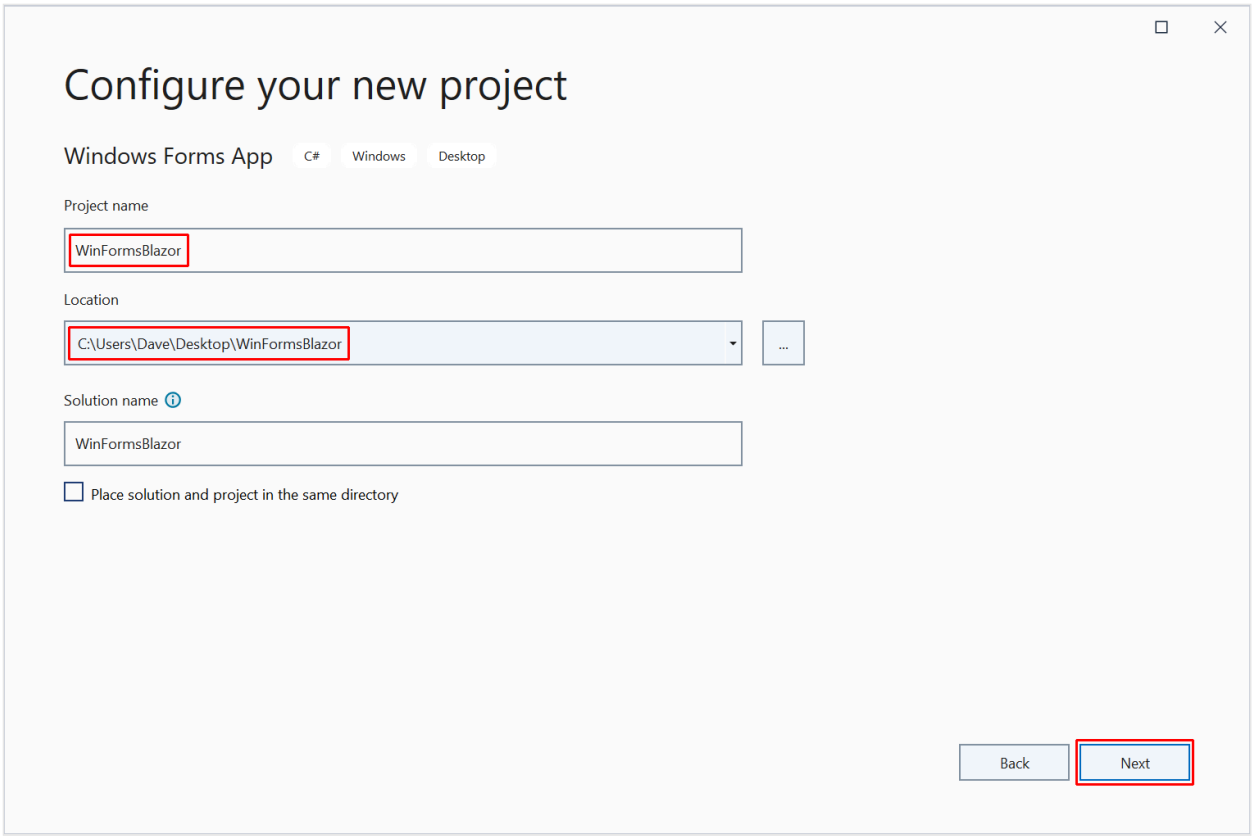
In the **Create a new project** dialog, filter the **Project type** dropdown to **Desktop**. Select the C# project template for **Windows Forms App** and select the **Next** button:



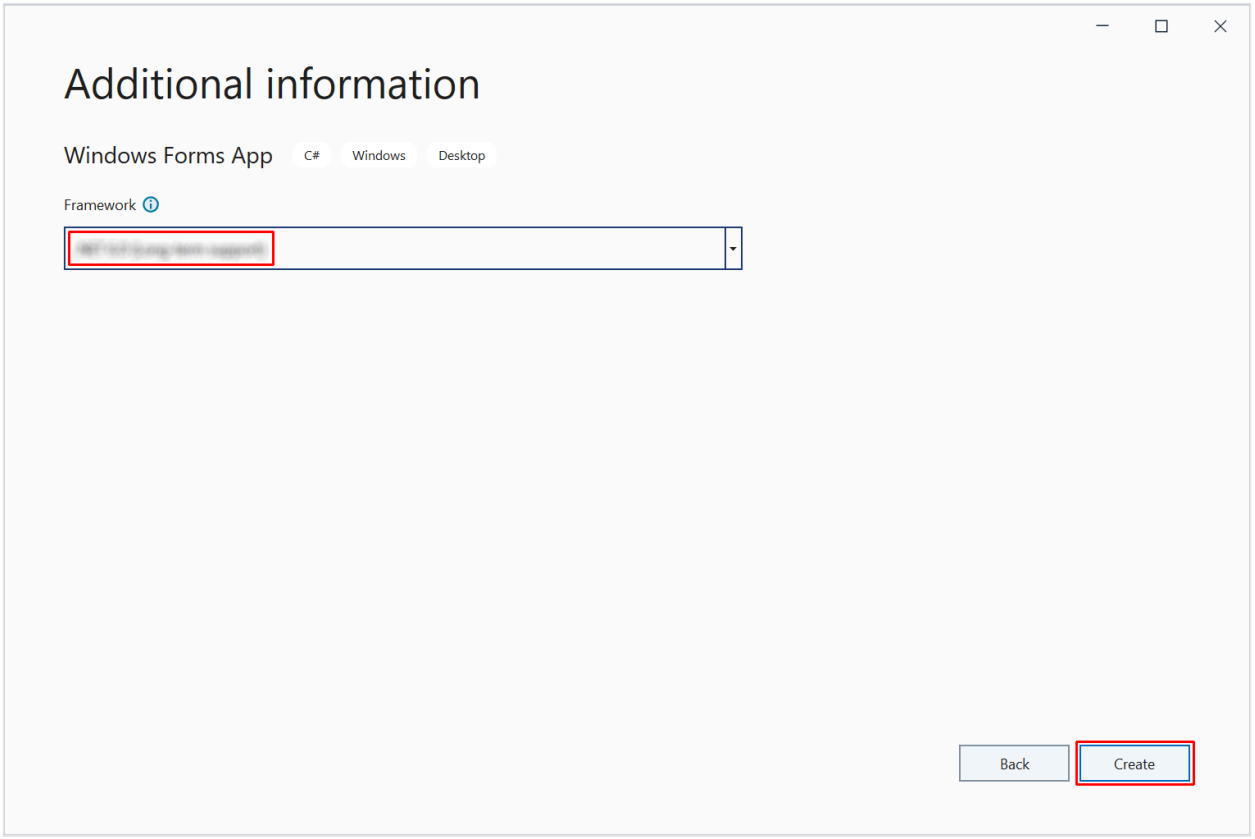
In the **Configure your new project** dialog:

- Set the **Project name** to **WinFormsBlazor**.
- Choose a suitable location for the project.

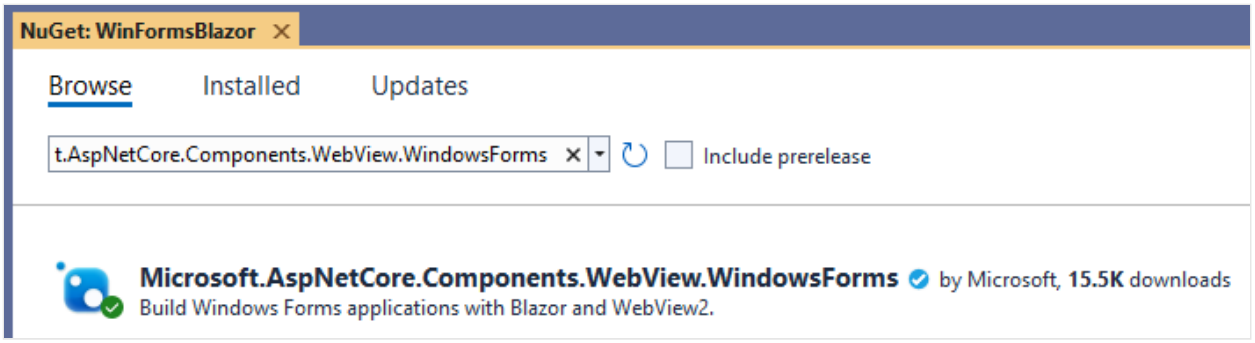
- Select the **Next** button.



In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:



Use NuGet Package Manager to install the Microsoft.AspNetCore.Components.WebView.WindowsForms NuGet package:



In **Solution Explorer**, right-click the project's name, **WinFormsBlazor**, and select **Edit Project File** to open the project file (`WinFormsBlazor.csproj`).

At the top of the project file, change the SDK to `Microsoft.NET.Sdk.Razor`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

Save the changes to the project file (`WinFormsBlazor.csproj`).

Add an `_Imports.razor` file to the root of the project with an @using directive for Microsoft.AspNetCore.Components.Web.

`_Imports.razor`:

razor

```razor
@using Microsoft.AspNetCore.Components.Web
```

Save the `_Imports.razor` file.

Add a `wwwroot` folder to the project.

Add an `index.html` file to the `wwwroot` folder with the following markup.

`wwwroot/index.html`:

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>WinFormsBlazor</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
```

```
    <link href="css/app.css" rel="stylesheet" />
    <link href="WinFormsBlazor.styles.css" rel="stylesheet" />
</head>

<body>

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

    <div id="blazor-error-ui" data-nosnippet>
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.webview.js"></script>

</body>

</html>
```

Inside the `wwwroot` folder, create a `css` folder to hold stylesheets.

Add an `app.css` stylesheet to the `wwwroot/css` folder with the following content.

`wwwroot/css/app.css`:

```css
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

h1:focus {
    outline: none;
}

a, .btn-link {
    color: #0071c1;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}
```

```
.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Inside the `wwwroot/css` folder, create a `bootstrap` folder. Inside the `bootstrap` folder, place a copy of `bootstrap.min.css`. You can obtain the latest version of `bootstrap.min.css` from the Bootstrap website. Because all of the content at the site is versioned in the URL, a direct link can't be provided here. Therefore, follow navigation bar links to **Docs** > **Download** to obtain `bootstrap.min.css`.

Add the following `Counter` component to the root of the project, which is the default `Counter` component found in Blazor project templates.

`Counter.razor`:

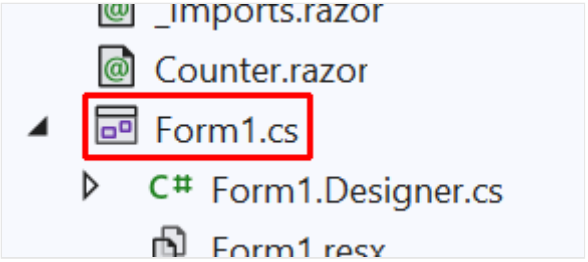
razor

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
```

```
    }
}
```

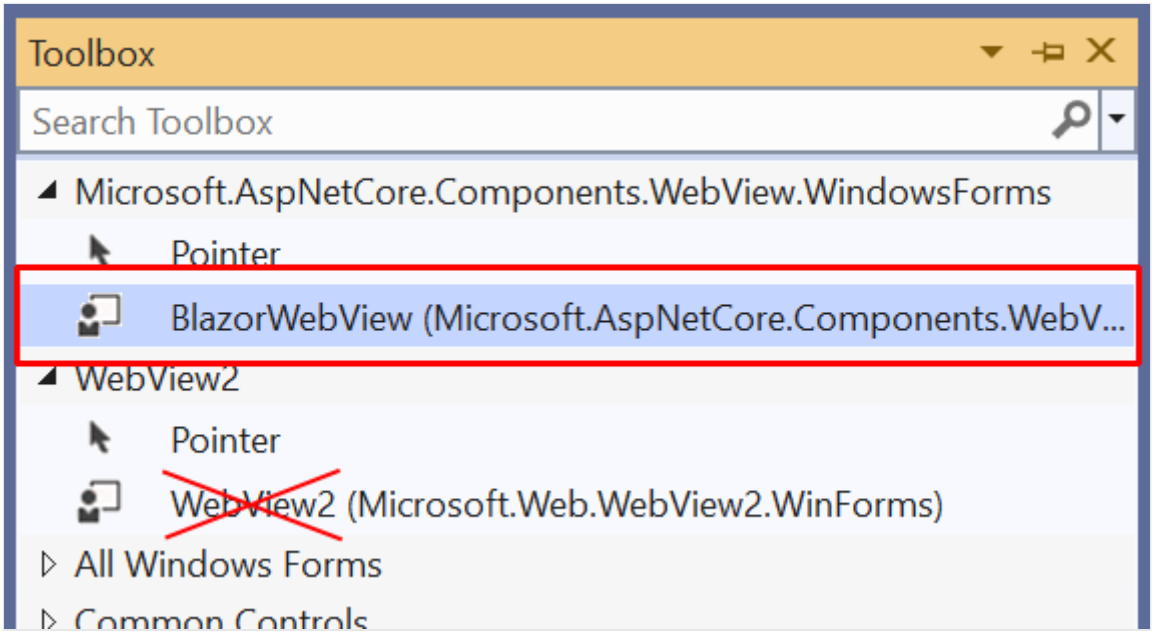
Save the `Counter` component (`Counter.razor`).

In **Solution Explorer**, double-click on the `Form1.cs` file to open the designer:

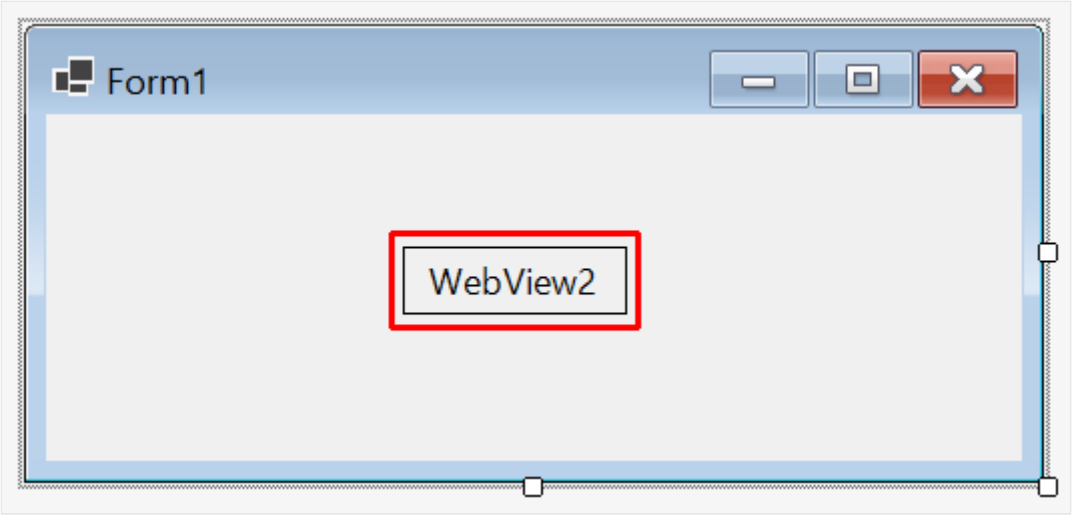


Open the **Toolbox** by either selecting the **Toolbox** button along the left edge of the Visual Studio window or selecting the **View** > **Toolbox** menu command.

Locate the **`BlazorWebView`** control under **`Microsoft.AspNetCore.Components.WebView.WindowsForms`**. Drag the BlazorWebView from the **Toolbox** into the `Form1` designer. Be careful not to accidentally drag a **`WebView2`** control into the form.

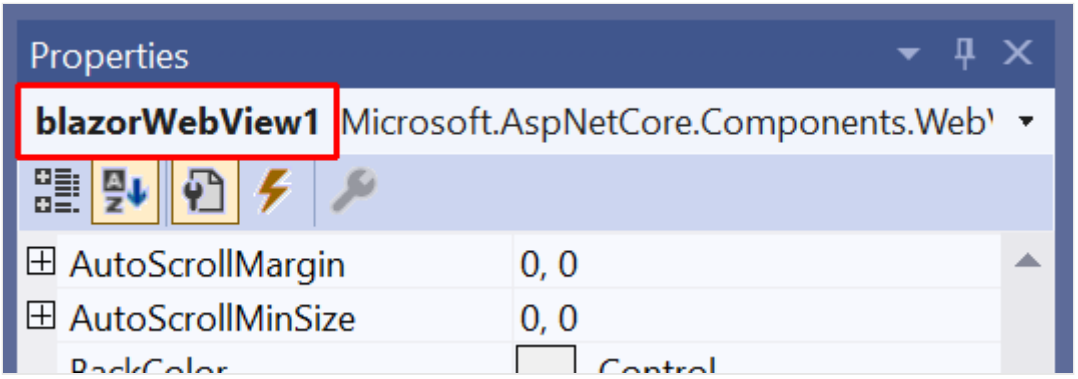


Visual Studio shows the BlazorWebView control in the form designer as `WebView2` and automatically names the control `blazorWebView1`:

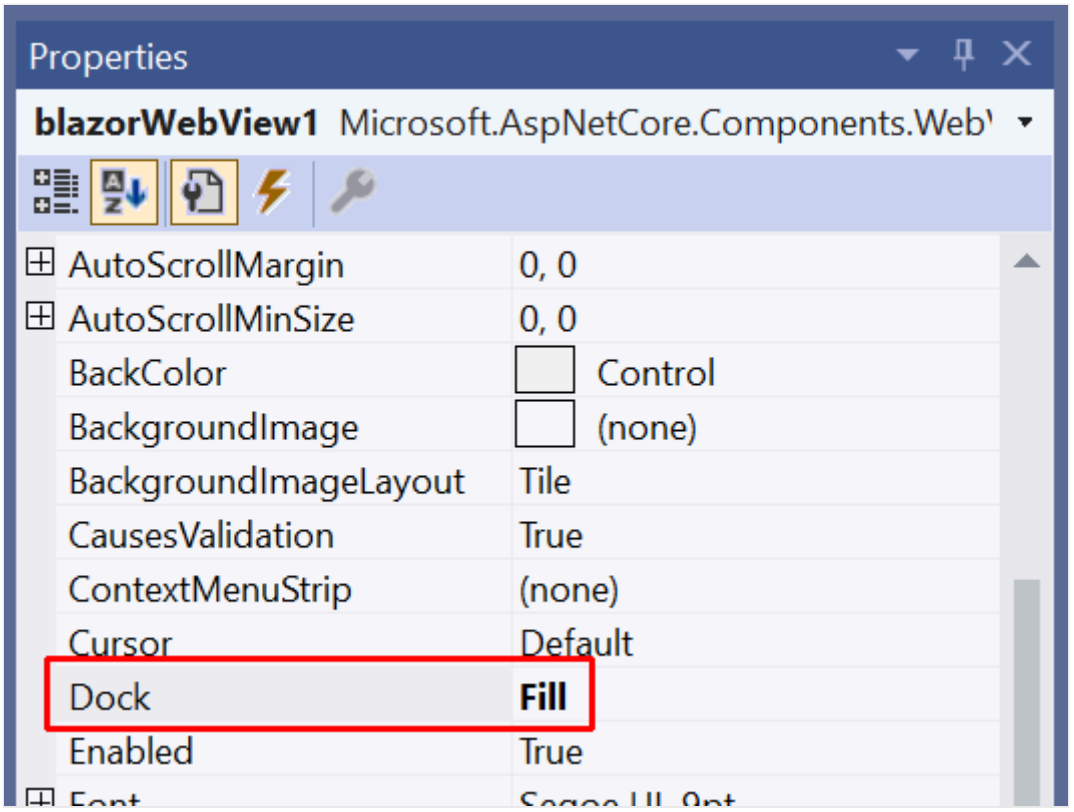


In `Form1`, select the BlazorWebView (`WebView2`) with a single click.

In the BlazorWebView's **Properties**, confirm that the control is named `blazorWebView1`. If the name isn't `blazorWebView1`, the wrong control was dragged from the **Toolbox**. Delete the `WebView2` control in `Form1` and drag the **`BlazorWebView` control** into the form.



In the control's properties, change the BlazorWebView's **Dock** value to **Fill**:



In the `Form1` designer, right-click `Form1` and select **View Code**.

Add namespaces for Microsoft.AspNetCore.Components.WebView.WindowsForms and Microsoft.Extensions.DependencyInjection to the top of the `Form1.cs` file:

C#

```
using Microsoft.AspNetCore.Components.WebView.WindowsForms;
using Microsoft.Extensions.DependencyInjection;
```

Inside the `Form1` constructor, after the `InitializeComponent` method call, add the following code:

C#

```
var services = new ServiceCollection();
services.AddWindowsFormsBlazorWebView();
blazorWebView1.HostPage = "wwwroot\\index.html";
blazorWebView1.Services = services.BuildServiceProvider();
blazorWebView1.RootComponents.Add<Counter>("#app");
```

> ⓘ Note
>
> The `InitializeComponent` method is automatically generated at app build time and added to the compilation object for the calling class.

The final, complete C# code of `Form1.cs` with a file-scoped namespace:

```C#
using Microsoft.AspNetCore.Components.WebView.WindowsForms;
using Microsoft.Extensions.DependencyInjection;

namespace WinFormsBlazor;

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        var services = new ServiceCollection();
        services.AddWindowsFormsBlazorWebView();
        blazorWebView1.HostPage = "wwwroot\\index.html";
        blazorWebView1.Services = services.BuildServiceProvider();
        blazorWebView1.RootComponents.Add<Counter>("#app");
    }
}
```

Run the app

Select the start button in the Visual Studio toolbar:

The app running on Windows:

Next steps

In this tutorial, you learned how to:

- ✓ Create a Windows Forms Blazor app project
- ✓ Run the app on Windows

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

Build a Windows Presentation Foundation (WPF) Blazor app

Article • 03/08/2024

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

This tutorial shows you how to build and run a WPF Blazor app. You learn how to:

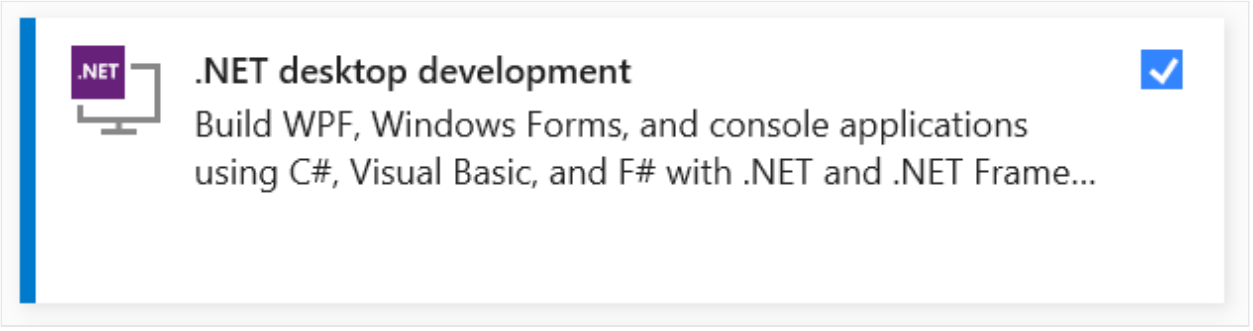
- ✓ Create a WPF Blazor app project
- ✓ Add a Razor component to the project
- ✓ Run the app on Windows

Prerequisites

- Supported platforms (WPF documentation)
- Visual Studio 2022 with the **.NET desktop development** workload

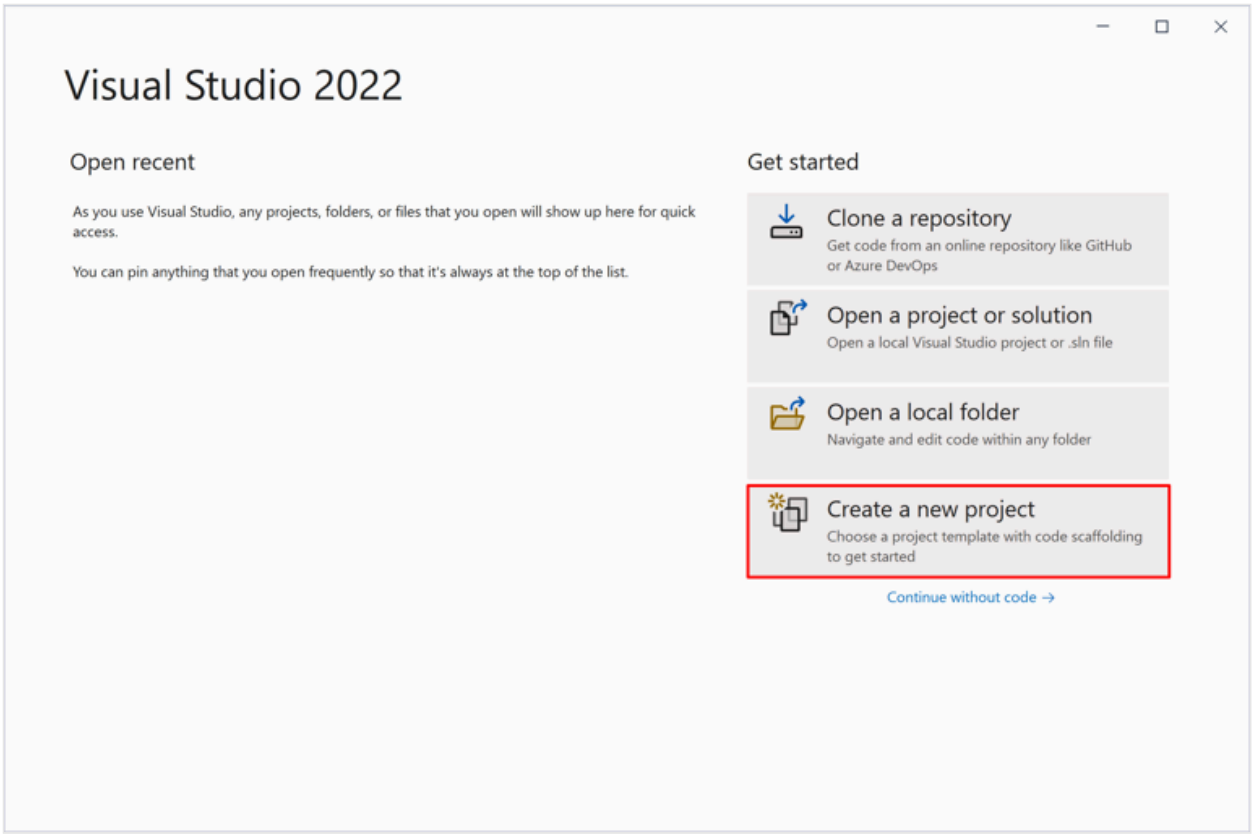
Visual Studio workload

If the **.NET desktop development** workload isn't installed, use the Visual Studio installer to install the workload. For more information, see Modify Visual Studio workloads, components, and language packs.

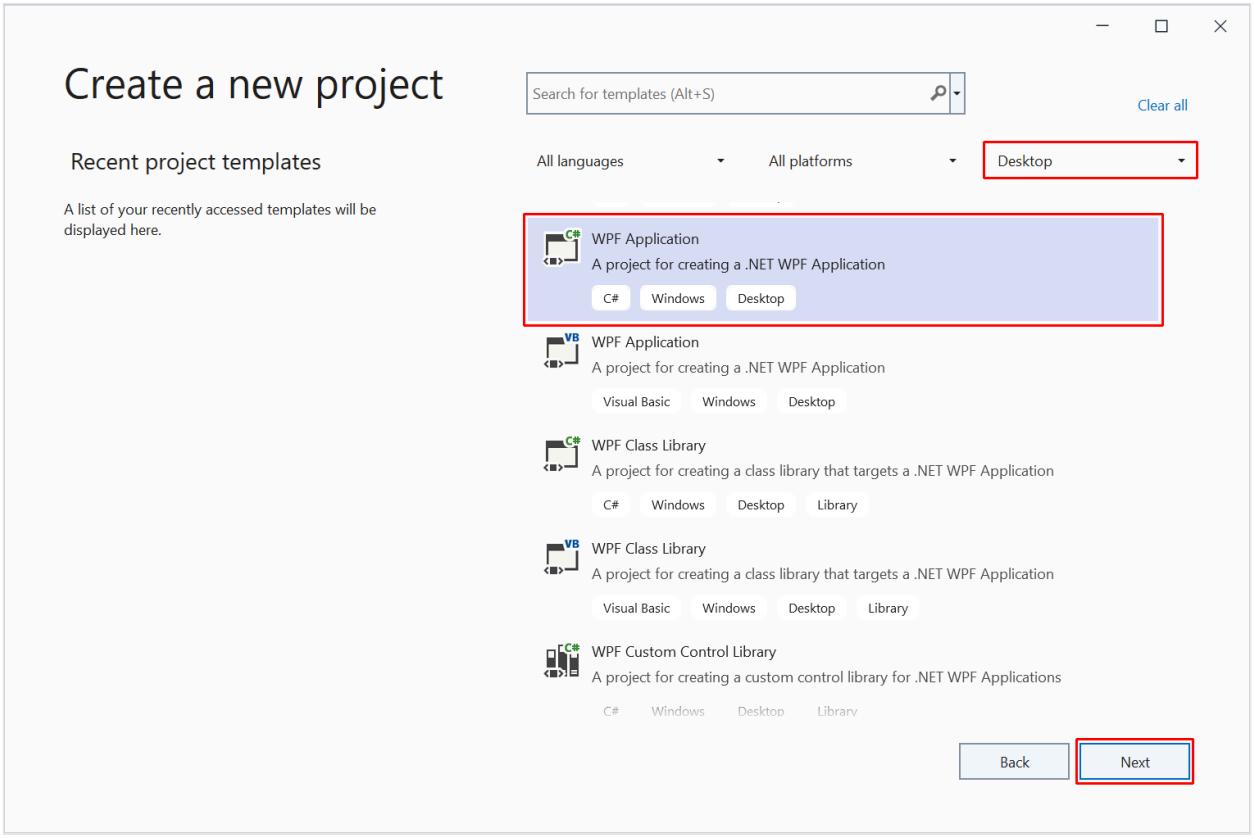


Create a WPF Blazor project

Launch Visual Studio. In the **Start Window**, select **Create a new project**:

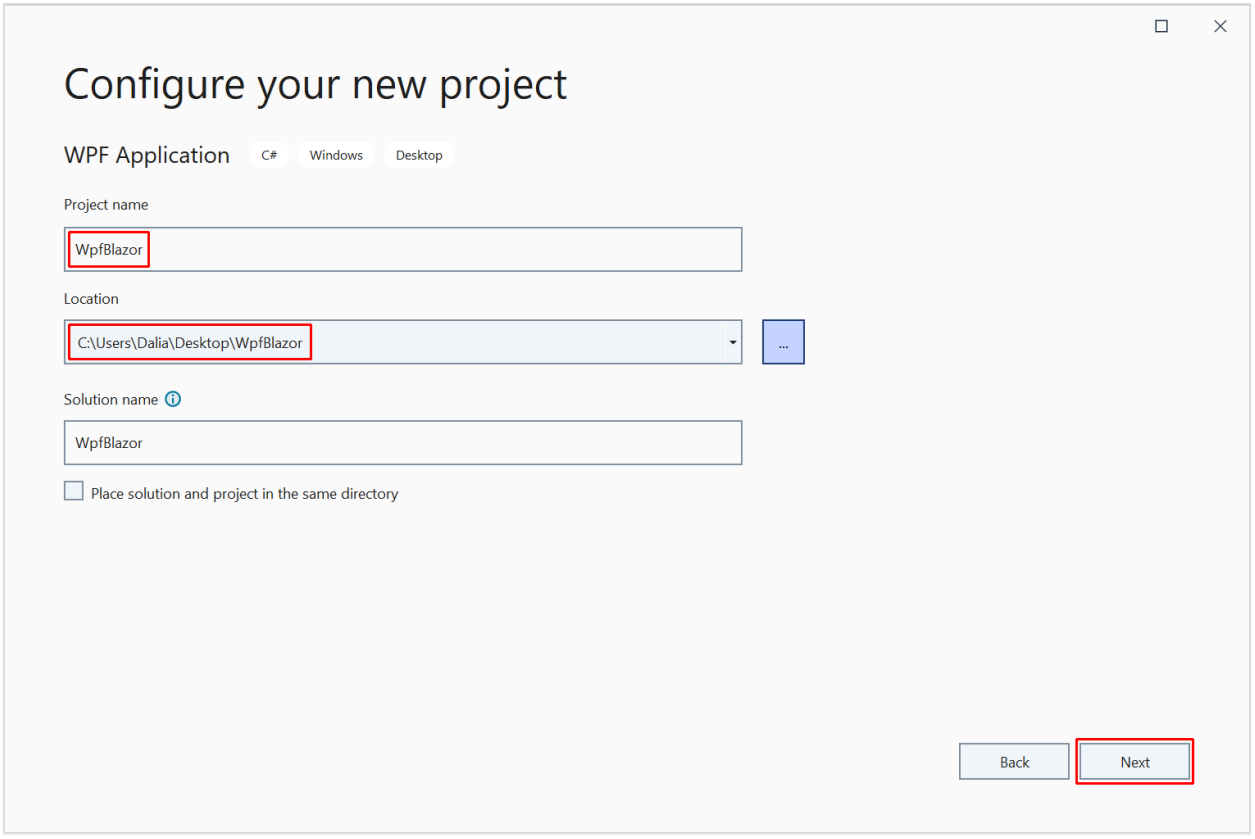


In the **Create a new project** dialog, filter the **Project type** dropdown to **Desktop**. Select the C# project template for **WPF Application** and select the **Next** button:

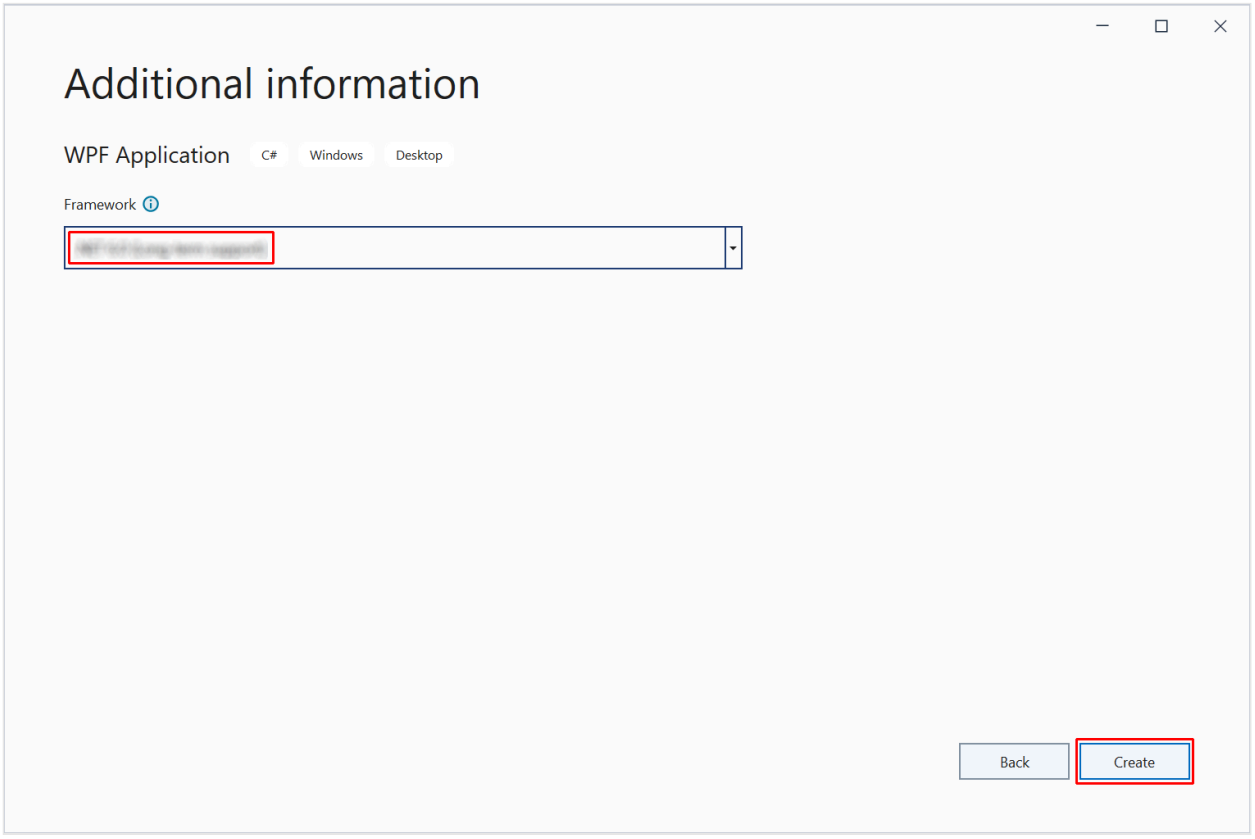


In the **Configure your new project** dialog:

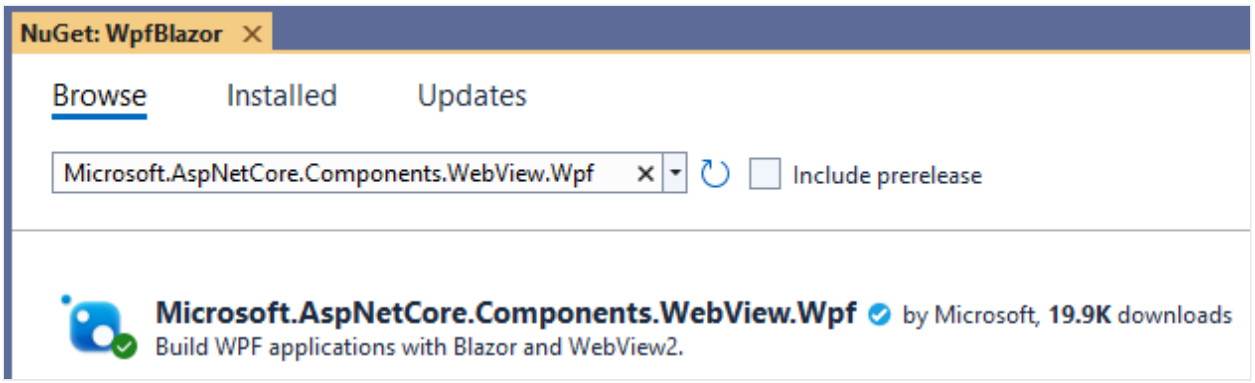
- Set the **Project name** to **WpfBlazor**.
- Choose a suitable location for the project.
- Select the **Next** button.



In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:



Use NuGet Package Manager to install the Microsoft.AspNetCore.Components.WebView.Wpf NuGet package:



In **Solution Explorer**, right-click the project's name, **WpfBlazor**, and select **Edit Project File** to open the project file (`WpfBlazor.csproj`).

At the top of the project file, change the SDK to `Microsoft.NET.Sdk.Razor`:

```xml
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

In the project file's existing `<PropertyGroup>` add the following markup to set the app's root namespace, which is `WpfBlazor` in this tutorial:

```xml
<RootNamespace>WpfBlazor</RootNamespace>
```

> **Note**
>
> The preceding guidance on setting the project's root namespace is a temporary workaround. For more information, see [Blazor][Wpf] Root namespace related issue (dotnet/maui #5861).

Save the changes to the project file (`WpfBlazor.csproj`).

Add an `_Imports.razor` file to the root of the project with an @using directive for Microsoft.AspNetCore.Components.Web.

`_Imports.razor`:

```razor
```

```
@using Microsoft.AspNetCore.Components.Web
```

Save the `_Imports.razor` file.

Add a `wwwroot` folder to the project.

Add an `index.html` file to the `wwwroot` folder with the following markup.

`wwwroot/index.html`:

HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>WpfBlazor</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="WpfBlazor.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui" data-nosnippet>
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>
    <script src="_framework/blazor.webview.js"></script>
</body>

</html>
```

Inside the `wwwroot` folder, create a `css` folder.

Add an `app.css` stylesheet to the `wwwroot/css` folder with the following content.

`wwwroot/css/app.css`:

css

```
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}
```

```
h1:focus {
    outline: none;
}

a, .btn-link {
    color: #0071c1;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}

.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Inside the `wwwroot/css` folder, create a `bootstrap` folder. Inside the `bootstrap` folder, place a copy of `bootstrap.min.css`. You can obtain the latest version of `bootstrap.min.css` from the Bootstrap website. Follow navigation bar links to **Docs** > **Download**. A direct link can't be provided here because all of the content at the site is versioned in the URL.

Add the following `Counter` component to the root of the project, which is the default `Counter` component found in Blazor project templates.

`Counter.razor`:

razor

```razor
<h1>Counter</h1>

<p>Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Save the `Counter` component (`Counter.razor`).

If the `MainWindow` designer isn't open, open it by double-clicking the `MainWindow.xaml` file in **Solution Explorer**. In the `MainWindow` designer, replace the XAML code with the following:

XAML

```xaml
<Window x:Class="WpfBlazor.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:blazor="clr-
namespace:Microsoft.AspNetCore.Components.WebView.Wpf;assembly=Microsoft.Asp
NetCore.Components.WebView.Wpf"
        xmlns:local="clr-namespace:WpfBlazor"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <blazor:BlazorWebView HostPage="wwwroot\index.html" Services="
{DynamicResource services}">
            <blazor:BlazorWebView.RootComponents>
                <blazor:RootComponent Selector="#app" ComponentType="{x:Type
local:Counter}" />
            </blazor:BlazorWebView.RootComponents>
        </blazor:BlazorWebView>
```

```
    </Grid>
</Window>
```

In **Solution Explorer**, right-click `MainWindow.xaml` and select **View Code**:

Add the namespace Microsoft.Extensions.DependencyInjection to the top of the `MainWindow.xaml.cs` file:

C#

```
using Microsoft.Extensions.DependencyInjection;
```

Inside the `MainWindow` constructor, after the `InitializeComponent` method call, add the following code:

C#

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddWpfBlazorWebView();
Resources.Add("services", serviceCollection.BuildServiceProvider());
```

> ⓘ **Note**
>
> The `InitializeComponent` method is automatically generated at app build time and added to the compilation object for the calling class.

The final, complete C# code of `MainWindow.xaml.cs` with a file-scoped namespace and comments removed:

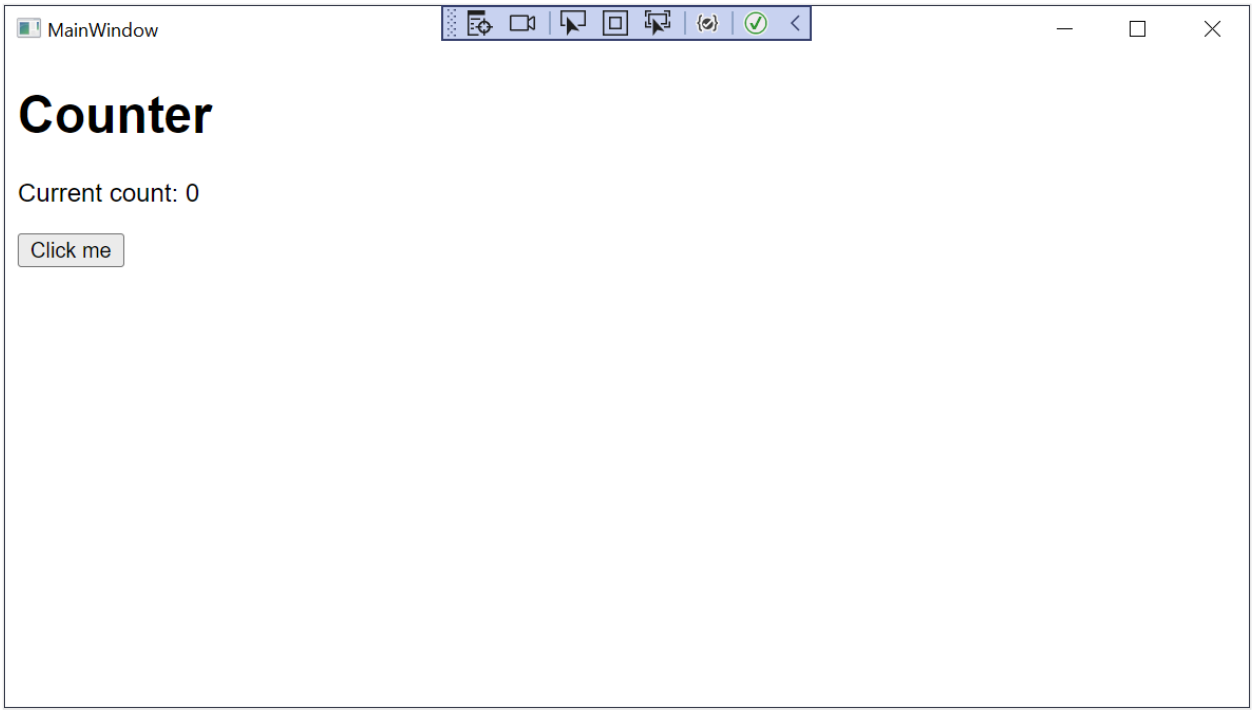
C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Extensions.DependencyInjection;

namespace WpfBlazor;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        var serviceCollection = new ServiceCollection();
        serviceCollection.AddWpfBlazorWebView();
        Resources.Add("services", serviceCollection.BuildServiceProvider());
    }
}
```

Run the app

Select the start button in the Visual Studio toolbar:

The app running on Windows:



Next steps

In this tutorial, you learned how to:

- ✓ Create a WPF Blazor app project
- ✓ Add a Razor component to the project
- ✓ Run the app on Windows

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

ASP.NET Core Blazor Hybrid routing and navigation

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 explains how to manage request routing and navigation in Blazor Hybrid apps.

URI request routing behavior

Default URI request routing behavior:

- A link is *internal* if the host name and scheme match between the app's origin URI and the request URI. When the host names and schemes don't match or if the link sets `target="_blank"`, the link is considered *external*.
- If the link is internal, the link is opened in the `BlazorWebView` by the app.
- If the link is external, the link is opened by an app determined by the device based on the device's registered handler for the link's scheme.
- For internal links that appear to request a file because the last segment of the URI uses dot notation (for example, `/file.x`, `/Maryia.Melnyk`, `/image.gif`) but don't point to any static content:
  - WPF and Windows Forms: The host page content is returned.
  - .NET MAUI: A 404 response is returned.

To change the link handling behavior for links that don't set `target="_blank"`, register the `UrlLoading` event and set the UrlLoadingEventArgs.UrlLoadingStrategy property. The UrlLoadingStrategy enumeration allows setting link handling behavior to any of the following values:

- OpenExternally: Load the URL using an app determined by the device. This is the default strategy for URIs with an external host.

- OpenInWebView: Load the URL within the `BlazorWebView`. This is the default strategy for URLs with a host matching the app origin. ***Don't use this strategy for external links unless you can ensure the destination URI is fully trusted.***
- CancelLoad: Cancels the current URL loading attempt.

The UrlLoadingEventArgs.Url property is used to get or dynamically set the URL.

> ⚠ **Warning**
>
> External links are opened in an app determined by the device. Opening external links within a `BlazorWebView` can introduce security vulnerabilities and shouldn't be enabled unless you can ensure that the external links are fully trusted.

API documentation:

- .NET MAUI: BlazorWebView.UrlLoading
- WPF: BlazorWebView.UrlLoading
- Windows Forms: BlazorWebView.UrlLoading

The Microsoft.AspNetCore.Components.WebView namespace is required for the following examples:

C#

```
using Microsoft.AspNetCore.Components.WebView;
```

Add the following event handler to the constructor of the `Page` where the `BlazorWebView` is created, which is `MainPage.xaml.cs` in an app created from the .NET MAUI project template.

C#

```
blazorWebView.UrlLoading +=
    (sender, urlLoadingEventArgs) =>
    {
        if (urlLoadingEventArgs.Url.Host != "0.0.0.0")
        {
            urlLoadingEventArgs.UrlLoadingStrategy =
                UrlLoadingStrategy.OpenInWebView;
        }
    };
```

Get or set a path for initial navigation

Use the `BlazorWebView.StartPath` property to get or set the path for initial navigation within the Blazor navigation context when the Razor component is finished loading. The default start path is the relative root URL path (`/`).

In the `MainPage` XAML markup (`MainPage.xaml`), specify the start path. The following example sets the path to a welcome page at `/welcome`:

XAML

```
<BlazorWebView ... StartPath="/welcome" ...>
    ...
<BlazorWebView>
```

Alternatively, the start path can be set in the `MainPage` constructor (`MainPage.xaml.cs`):

C#

```
blazorWebView.StartPath = "/welcome";
```

Navigation among pages and Razor components

This section explains how to navigate among .NET MAUI content pages and Razor components.

The .NET MAUI Blazor hybrid project template isn't a Shell-based app, so the URI-based navigation for Shell-based apps isn't suitable for a project based on the project template. The examples in this section use a NavigationPage to perform modeless or modal navigation.

In the following example:

- The namespace of the app is `MauiBlazor`, which matches the suggested project name of the Build a .NET MAUI Blazor Hybrid app tutorial.
- A ContentPage is placed in a new folder added to the app named `Views`.

In `App.xaml.cs`, create the `MainPage` as a NavigationPage by making the following change:

diff

```
- MainPage = new MainPage();
```

```
+ MainPage = new NavigationPage(new MainPage());
```

`Views/NavigationExample.xaml`:

XAML

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MauiBlazor"
             x:Class="MauiBlazor.Views.NavigationExample"
             Title="Navigation Example"
             BackgroundColor="{DynamicResource PageBackgroundColor}">
    <StackLayout>
        <Label Text="Navigation Example"
               VerticalOptions="Center"
               HorizontalOptions="Center"
               FontSize="24" />
        <Button x:Name="CloseButton"
                Clicked="CloseButton_Clicked"
                Text="Close" />
    </StackLayout>
</ContentPage>
```

In the following `NavigationExample` code file, the `CloseButton_Clicked` event handler for the close button calls PopAsync to pop the ContentPage off of the navigation stack.

`Views/NavigationExample.xaml.cs`:

C#

```
namespace MauiBlazor.Views;

public partial class NavigationExample : ContentPage
{
    public NavigationExample()
    {
        InitializeComponent();
    }

    private async void CloseButton_Clicked(object sender, EventArgs e)
    {
        await Navigation.PopAsync();
    }
}
```

In a Razor component:

- Add the namespace for the app's content pages. In the following example, the namespace is `MauiBlazor.Views`.

- Add an HTML `button` element with an @onclick event handler to open the content page. The event handler method is named `OpenPage`.
- In the event handler, call PushAsync to push the ContentPage, `NavigationExample`, onto the navigation stack.

The following example is based on the `Index` component in the .NET MAUI Blazor project template.

`Pages/Index.razor`:

razor

```
@page "/"
@using MauiBlazor.Views

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<button class="btn btn-primary" @onclick="OpenPage">Open</button>

@code {
    private async void OpenPage()
    {
        await App.Current.MainPage.Navigation.PushAsync(new
NavigationExample());
    }
}
```

To change the preceding example to modal navigation:

- In the `CloseButton_Clicked` method (`Views/NavigationExample.xaml.cs`), change PopAsync to PopModalAsync:

  diff

  ```
  - await Navigation.PopAsync();
  + await Navigation.PopModalAsync();
  ```

- In the `OpenPage` method (`Pages/Index.razor`), change PushAsync to PushModalAsync:

  diff

  ```
  - await App.Current.MainPage.Navigation.PushAsync(new
  NavigationExample());
  ```

```
+ await App.Current.MainPage.Navigation.PushModalAsync(new
NavigationExample());
```

For more information, see the following resources:

- NavigationPage article (.NET MAUI documentation)
- NavigationPage (API documentation)

App linking (deep linking)

It's often desirable to connect a website and a mobile app so that links on a website launch the mobile app and display content in the mobile app. App linking, also known as *deep linking*, is a technique that enables a mobile device to respond to a URI and launch content in a mobile app that's represented by the URI.

For more information, see the following articles in the .NET MAUI documentation:

- Android app links
- Apple universal links

ASP.NET Core Blazor Hybrid static files

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 how to consume static asset files in Blazor Hybrid apps.

In a Blazor Hybrid app, static files are *app resources*, accessed by Razor components using the following approaches:

- .NET MAUI: .NET MAUI file system helpers
- WPF and Windows Forms: ResourceManager

When static assets are only used in the Razor components, static assets can be consumed from the web root (`wwwroot` folder) in a similar way to Blazor WebAssembly and Blazor Server apps. For more information, see the Static assets limited to Razor components section.

.NET MAUI

In .NET MAUI apps, *raw assets* using the `MauiAsset` build action and .NET MAUI file system helpers are used for static assets.

> **Note**
>
> Interfaces, classes, and supporting types to work with storage on devices across all supported platforms for features such as choosing a file, saving preferences, and using secure storage are in the **Microsoft.Maui.Storage** namespace. The namespace is available throughout a MAUI Blazor Hybrid app, so there's no need to specify a `using` statement in a class file or an `@using` Razor directive in a Razor component for the namespace.

Place raw assets into the `Resources/Raw` folder of the app. The example in this section uses a static text file.

`Resources/Raw/Data.txt`:

```text
This is text from a static text file resource.
```

The following Razor component:

- Calls OpenAppPackageFileAsync to obtain a Stream for the resource.
- Reads the Stream with a StreamReader.
- Calls StreamReader.ReadToEndAsync to read the file.

`Pages/StaticAssetExample.razor`:

```razor
@page "/static-asset-example"
@using System.IO
@using Microsoft.Extensions.Logging
@inject ILogger<StaticAssetExample> Logger

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>

@code {
    public string dataResourceText = "Loading resource ...";

    protected override async Task OnInitializedAsync()
    {
        try
        {
            using var stream =
                await FileSystem.OpenAppPackageFileAsync("Data.txt");
            using var reader = new StreamReader(stream);

            dataResourceText = await reader.ReadToEndAsync();
        }
        catch (FileNotFoundException ex)
        {
            dataResourceText = "Data file not found.";
            Logger.LogError(ex, "'Resource/Raw/Data.txt' not found.");
        }
    }
}
```

For more information, see the following resources:

- Target multiple platforms from .NET MAUI single project (.NET MAUI documentation)

- Improve consistency with resizetizer (dotnet/maui #4367)

WPF

Place the asset into a folder of the app, typically at the project's root, such as a `Resources` folder. The example in this section uses a static text file.

`Resources/Data.txt`:

```text
This is text from a static text file resource.
```

If a `Properties` folder doesn't exist in the app, create a `Properties` folder in the root of the app.

If the `Properties` folder doesn't contain a resources file (`Resources.resx`), create the file in **Solution Explorer** with the **Add** > **New Item** contextual menu command.

Double-click the `Resource.resx` file.

Select **Strings** > **Files** from the dropdown list.

Select **Add Resource** > **Add Existing File**. If prompted by Visual Studio to confirm editing the file, select **Yes**. Navigate to the `Resources` folder, select the `Data.txt` file, and select **Open**.

In the following example component, ResourceManager.GetString obtains the string resource's text for display.

> ⚠ **Warning**
>
> Never use ResourceManager methods with untrusted data.

`StaticAssetExample.razor`:

```razor
@page "/static-asset-example"
@using System.Resources

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>
```

```
@code {
    public string dataResourceText = "Loading resource ...";

    protected override void OnInitialized()
    {
        var resources =
            new ResourceManager(typeof(WpfBlazor.Properties.Resources));

        dataResourceText = resources.GetString("Data") ?? "'Data' not
found.";
    }
}
```

Windows Forms

Place the asset into a folder of the app, typically at the project's root, such as a `Resources` folder. The example in this section uses a static text file.

`Resources/Data.txt`:

```
text
```

```
This is text from a static text file resource.
```

Examine the files associated with `Form1` in **Solution Explorer**. If `Form1` doesn't have a resource file (`.resx`), add a `Form1.resx` file with the **Add** > **New Item** contextual menu command.

Double-click the `Form1.resx` file.

Select **Strings** > **Files** from the dropdown list.

Select **Add Resource** > **Add Existing File**. If prompted by Visual Studio to confirm editing the file, select **Yes**. Navigate to the `Resources` folder, select the `Data.txt` file, and select **Open**.

In the following example component:

- The app's assembly name is `WinFormsBlazor`. The ResourceManager's base name is set to the assembly name of `Form1` (`WinFormsBlazor.Form1`).
- ResourceManager.GetString obtains the string resource's text for display.

> ⚠ **Warning**

Never use ResourceManager methods with untrusted data.

`StaticAssetExample.razor`:

razor

```razor
@page "/static-asset-example"
@using System.Resources

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>

@code {
    public string dataResourceText = "Loading resource ...";

    protected override async Task OnInitializedAsync()
    {
        var resources =
            new ResourceManager("WinFormsBlazor.Form1",
this.GetType().Assembly);

        dataResourceText = resources.GetString("Data") ?? "'Data' not
found.";
    }
}
```

Static assets limited to Razor components

A `BlazorWebView` control has a configured host file (HostPage), typically `wwwroot/index.html`. The HostPage path is relative to the project. All static web assets (scripts, CSS files, images, and other files) that are referenced from a `BlazorWebView` are relative to its configured HostPage.

Static web assets from a Razor class library (RCL) use special paths: `_content/{PACKAGE ID}/{PATH AND FILE NAME}`. The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. The `{PATH AND FILE NAME}` placeholder is path and file name under `wwwroot`. These paths are logically subpaths of the app's `wwwroot` folder, although they're actually coming from other packages or projects. Component-specific CSS style bundles are also built at the root of the `wwwroot` folder.

The web root of the HostPage determines which subset of static assets are available:

- `wwwroot/index.html` (*recommended*): All assets in the app's `wwwroot` folder are available (for example: `wwwroot/image.png` is available from `/image.png`), including subfolders (for example: `wwwroot/subfolder/image.png` is available from `/subfolder/image.png`). RCL static assets in the RCL's `wwwroot` folder are available (for example: `wwwroot/image.png` is available from the path `_content/{PACKAGE ID}/image.png`), including subfolders (for example: `wwwroot/subfolder/image.png` is available from the path `_content/{PACKAGE ID}/subfolder/image.png`).
- `wwwroot/{PATH}/index.html`: All assets in the app's `wwwroot/{PATH}` folder are available using app web root relative paths. RCL static assets in `wwwroot/{PATH}` aren't because they would be in a non-existent theoretical location, such as `../../_content/{PACKAGE ID}/{PATH}`, which isn't a supported relative path.
- `wwwroot/_content/{PACKAGE ID}/index.html`: All assets in the RCL's `wwwroot/{PATH}` folder are available using RCL web root relative paths. The app's static assets in `wwwroot/{PATH}` are aren't because they would be in a non-existent theoretical location, such as `../../{PATH}`, which isn't a supported relative path.

For most apps, we recommend placing the HostPage at the root of the `wwwroot` folder of the app, which provides the greatest flexibility for supplying static assets from the app, RCLs, and via subfolders of the app and RCLs.

The following examples demonstrate referencing static assets from the app's web root (`wwwroot` folder) with a HostPage rooted in the `wwwroot` folder.

`wwwroot/data.txt`:

text

```
This is text from a static text file resource.
```

`wwwroot/scripts.js`:

JavaScript

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

The following Jeep® image is also used in this section's example. You can right-click the following image to save it locally for use in a local test app.

`wwwroot/jeep-yj.png`:

In a Razor component:

- The static text file contents can be read using the following techniques:
  - .NET MAUI: .NET MAUI file system helpers (OpenAppPackageFileAsync)
  - WPF and Windows Forms: StreamReader.ReadToEndAsync
- JavaScript files are available at logical subpaths of `wwwroot` using `./` paths.
- The image can be the source attribute (`src`) of an image tag (`<img>`).

`StaticAssetExample2.razor`:

razor

```
@page "/static-asset-example-2"
@using Microsoft.Extensions.Logging
@implements IAsyncDisposable
@inject IJSRuntime JS
@inject ILogger<StaticAssetExample2> Logger

<h1>Static Asset Example 2</h1>

<h2>Read a file</h2>

<p>@dataResourceText</p>

<h2>Call JavaScript</h2>

<p>
    <button @onclick="TriggerPrompt">Trigger browser window prompt</button>
</p>

<p>@result</p>

<h2>Show an image</h2>

<p><img alt="1991 Jeep YJ" src="/jeep-yj.png" /></p>

<p>
    <em>Jeep</em> and <em>Jeep YJ</em> are registered trademarks of
```

```
    <a href="https://www.stellantis.com">FCA US LLC (Stellantis NV)</a>.
</p>

@code {
    private string dataResourceText = "Loading resource ...";
    private IJSObjectReference? module;
    private string result;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            dataResourceText = await ReadData();
        }
        catch (FileNotFoundException ex)
        {
            dataResourceText = "Data file not found.";
            Logger.LogError(ex, "'wwwroot/data.txt' not found.");
        }
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./scripts.js");
        }
    }

    private async Task TriggerPrompt()
    {
        result = await Prompt("Provide some text");
    }

    public async ValueTask<string> Prompt(string message) =>
        module is not null ?
            await module.InvokeAsync<string>("showPrompt", message) : null;

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}
```

In .NET MAUI apps, add the following `ReadData` method to the `@code` block of the preceding component:

```
C#
```

```
private async Task<string> ReadData()
{
    using var stream = await
FileSystem.OpenAppPackageFileAsync("wwwroot/data.txt");
    using var reader = new StreamReader(stream);

    return await reader.ReadToEndAsync();
}
```

In WPF and Windows Forms apps, add the following `ReadData` method to the `@code` block of the preceding component:

C#

```
private async Task<string> ReadData()
{
    using var reader = new StreamReader("wwwroot/data.txt");

    return await reader.ReadToEndAsync();
}
```

Collocated JavaScript files are also accessible at logical subpaths of `wwwroot`. Instead of using the script described earlier for the `showPrompt` function in `wwwroot/scripts.js`, the following collocated JavaScript file for the `StaticAssetExample2` component also makes the function available.

`Pages/StaticAssetExample2.razor.js`:

JavaScript

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

Modify the module object reference in the `StaticAssetExample2` component to use the collocated JavaScript file path (`./Pages/StaticAssetExample2.razor.js`):

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",
    "./Pages/StaticAssetExample2.razor.js");
```

Trademarks

Jeep and *Jeep YJ* are registered trademarks of FCA US LLC (Stellantis NV).

Additional resources

- ResourceManager
- Create resource files for .NET apps (.NET Fundamentals documentation)
- How to: Use resources in localizable apps (WPF documentation)

Use browser developer tools with 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 use browser developer tools with Blazor Hybrid apps.

Browser developer tools with .NET MAUI Blazor

Ensure the Blazor Hybrid project is configured to support browser developer tools. You can confirm developer tools support by searching the app for `AddBlazorWebViewDeveloperTools`.

If the project isn't already configured for browser developer tools, add support by:

1. Locating where the call to AddMauiBlazorWebView is made, likely within the app's `MauiProgram.cs` file.
2. At the top of the `MauiProgram.cs` file, confirm the presence of a `using` statement for Microsoft.Extensions.Logging. If the `using` statement isn't present, add it to the top of the file:

   C#

   ```
   using Microsoft.Extensions.Logging;
   ```

3. After the call to AddMauiBlazorWebView, add the following code:

   C#

   ```
   #if DEBUG
       builder.Services.AddBlazorWebViewDeveloperTools();
   ```

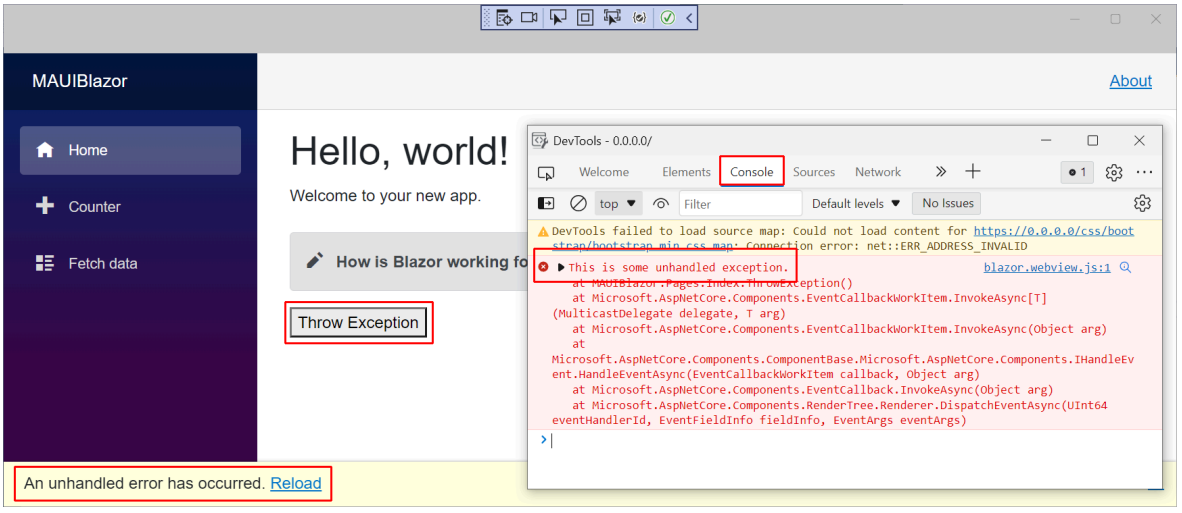
```
    builder.Logging.AddDebug();
#endif
```

To use browser developer tools with a Windows app:

1. Run the .NET MAUI Blazor Hybrid app for Windows and navigate to an app page that uses a BlazorWebView. The developer tools console is unavailable from ContentPages without a Blazor Web View.

2. Use the keyboard shortcut `Ctrl`+`Shift`+`I` to open browser developer tools.

3. Developer tools provide a variety of features for working with apps, including which assets the page requested, how long assets took to load, and the content of loaded assets. The following example shows the **Console** tab to see the console messages, which includes any exception messages generated by the framework or developer code:



Additional resources

- Chrome DevTools
- Microsoft Edge Developer Tools overview
- Safari Developer Help

Reuse Razor components in ASP.NET Core Blazor Hybrid

Article • 12/05/2024

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

This article explains how to author and organize Razor components for the web and Web Views in Blazor Hybrid apps.

Razor components work across hosting models (Blazor WebAssembly, Blazor Server, and in the Web View of Blazor Hybrid) and across platforms (Android, iOS, and Windows). Hosting models and platforms have unique capabilities that components can leverage, but components executing across hosting models and platforms must leverage unique capabilities separately, which the following examples demonstrate:

- Blazor WebAssembly supports synchronous JavaScript (JS) interop, which isn't supported by the strictly asynchronous JS interop communication channel in Blazor Server and Web Views of Blazor Hybrid apps.
- Components in a Blazor Server app can access services that are only available on the server, such as an Entity Framework database context.
- Components in a `BlazorWebView` can directly access native desktop and mobile device features, such as geolocation services. Blazor Server and Blazor WebAssembly apps must rely upon web API interfaces of apps on external servers to provide similar features.

Design principles

In order to author Razor components that can seamlessly work across hosting models and platforms, adhere to the following design principles:

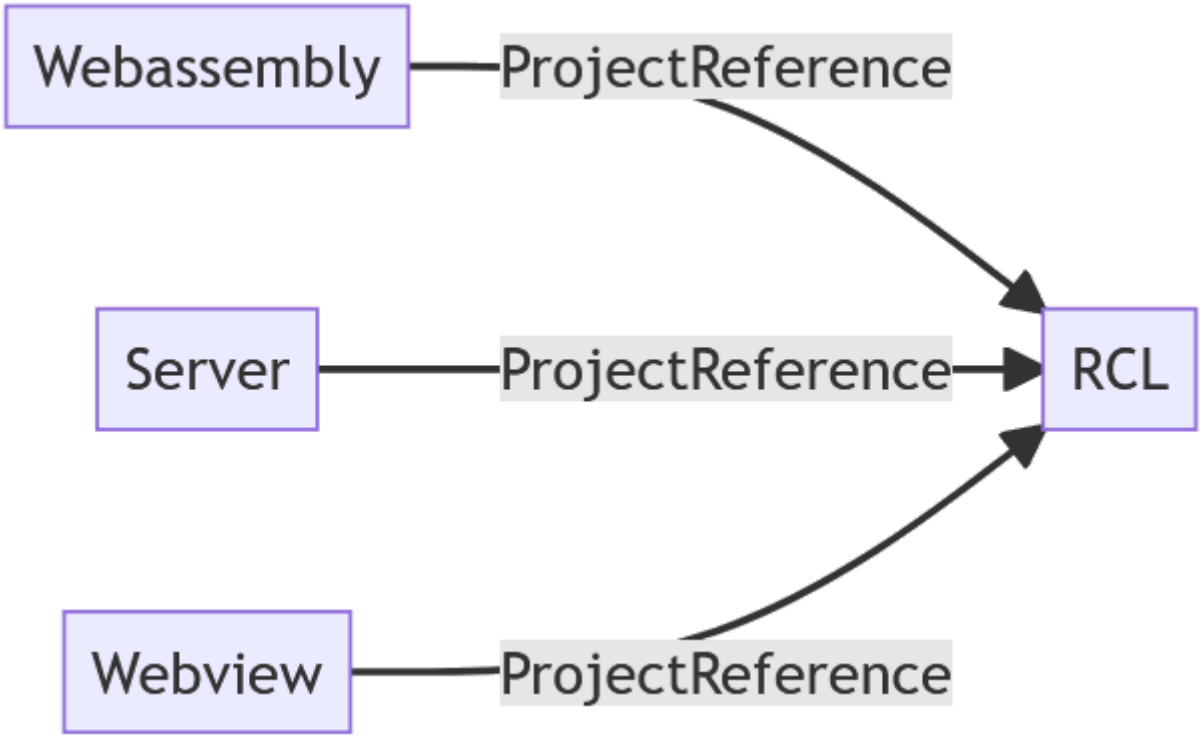
- Place shared UI code in Razor class libraries (RCLs), which are containers designed to maintain reusable pieces of UI for use across different hosting models and platforms.

- Implementations of unique features shouldn't exist in RCLs. Instead, the RCL should define abstractions (interfaces and base classes) that hosting models and platforms implement.
- Only opt-in to unique features by hosting model or platform. For example, Blazor WebAssembly supports the use of IJSInProcessRuntime and IJSInProcessObjectReference in a component as an optimization, but only use them with conditional casts and fallback implementations that rely on the universal IJSRuntime and IJSObjectReference abstractions that all hosting models and platforms support. For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor. For more information on IJSInProcessObjectReference, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.
- As a general rule, use CSS for HTML styling in components. The most common case is for consistency in the look and feel of an app. In places where UI styles must differ across hosting models or platforms, use CSS to style the differences.
- If some part of the UI requires additional or different content for a target hosting model or platform, the content can be encapsulated inside a component and rendered inside the RCL using DynamicComponent. Additional UI can also be provided to components via RenderFragment instances. For more information on RenderFragment, see Child content render fragments and Render fragments for reusable rendering logic.

Project code organization

As much as possible, place code and static content in Razor class libraries (RCLs). Each hosting model or platform references the RCL and registers individual implementations in the app's service collection that a Razor component might require.

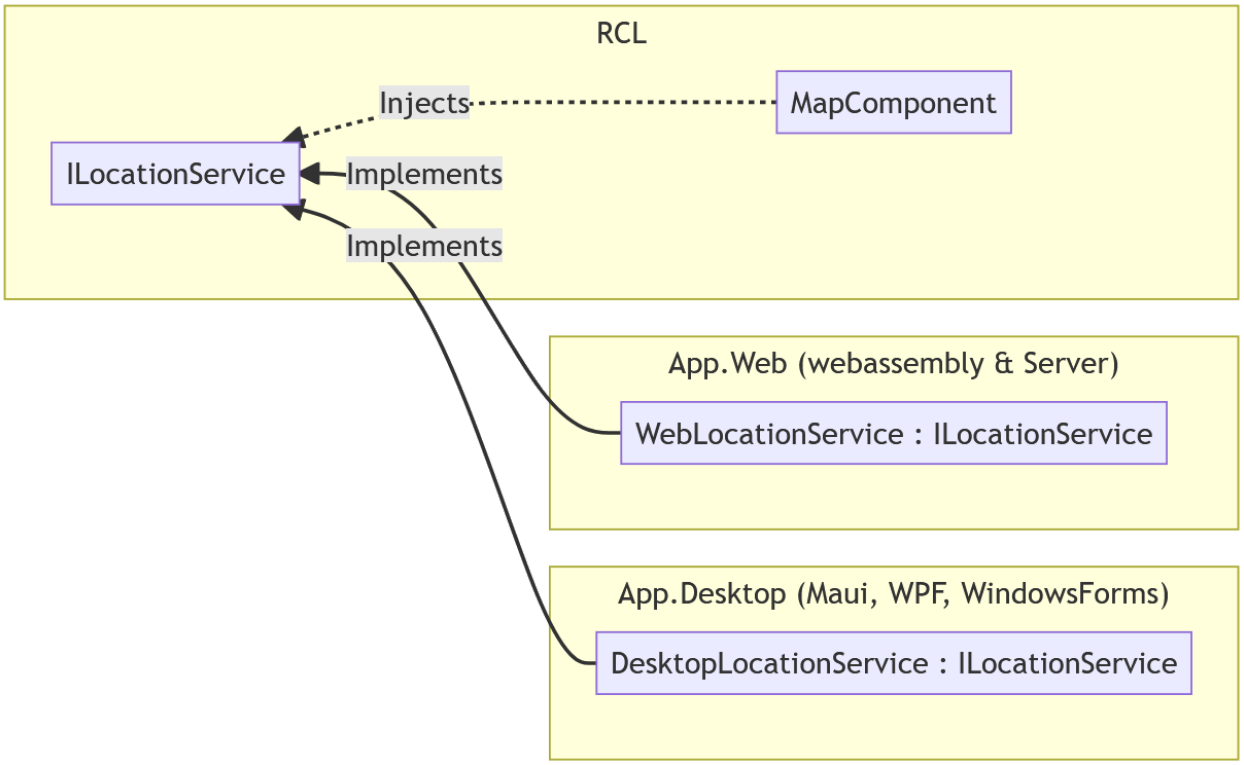
Each target assembly should contain only the code that is specific to that hosting model or platform along with the code that helps bootstrap the app.



Use abstractions for unique features

The following example demonstrates how to use an abstraction for a geolocation service by hosting model and platform.

- In a Razor class library (RCL) used by the app to obtain geolocation data for the user's location on a map, the `MapComponent` Razor component injects an `ILocationService` service abstraction.
- `App.Web` for Blazor WebAssembly and Blazor Server projects implement `ILocationService` as `WebLocationService`, which uses web API calls to obtain geolocation data.
- `App.Desktop` for .NET MAUI, WPF, and Windows Forms, implement `ILocationService` as `DesktopLocationService`. `DesktopLocationService` uses platform-specific device features to obtain geolocation data.



.NET MAUI Blazor platform-specific code

A common pattern in .NET MAUI is to create separate implementations for different platforms, such as defining partial classes with platform-specific implementations. For example, see the following diagram, where partial classes for `CameraService` are implemented in each of `CameraService.Windows.cs`, `CameraService.iOS.cs`, `CameraService.Android.cs`, and `CameraService.cs`:

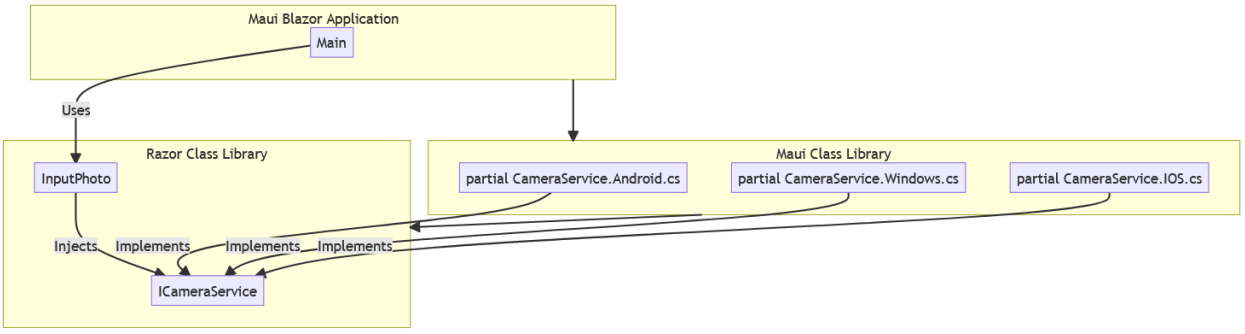
Where you want to pack platform-specific features into a class library that can be consumed by other apps, we recommend that you follow a similar approach to the one described in the preceding example and create an abstraction for the Razor component:

- Place the component in a Razor class library (RCL).
- From a .NET MAUI class library, reference the RCL and create the platform-specific implementations.
- Within the consuming app, reference the .NET MAUI class library.

The following example demonstrates the concepts for images in an app that organizes photographs:

- A .NET MAUI Blazor Hybrid app uses `InputPhoto` from an RCL that it references.
- The .NET MAUI app also references a .NET MAUI class library.

- `InputPhoto` in the RCL injects an `ICameraService` interface, which is defined in the RCL.
- `CameraService` partial class implementations for `ICameraService` are in the .NET MAUI class library (`CameraService.Windows.cs`, `CameraService.iOS.cs`, `CameraService.Android.cs`), which references the RCL.



For an example, see Build a .NET MAUI Blazor Hybrid app with a Blazor Web App.

Additional resources

- eShop Reference Application (AdventureWorks): The .NET MAUI Blazor Hybrid app is in the `src/HybridApp` folder.
  - For Azure hosting: Azure-Samples/eShopOnAzure GitHub repository
  - For general non-Azure hosting: dotnet/eShop GitHub repository.

Share assets across web and native clients using a Razor class library (RCL)

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**.

Use a Razor class library (RCL) to share Razor components, C# code, and static assets across web and native client projects.

This article builds on the general concepts found in the following articles:

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

The examples in this article share assets between a server-side Blazor app and a .NET MAUI Blazor Hybrid app in the same solution:

- Although a server-side Blazor app is used, the guidance applies equally to Blazor WebAssembly apps sharing assets with a Blazor Hybrid app.
- Projects are in the same solution, but an RCL can supply shared assets to projects outside of a solution.
- The RCL is added as a project to the solution, but any RCL can be published as a NuGet package. A NuGet package can supply shared assets to web and native client projects.
- The order that the projects are created isn't important. However, projects that rely on an RCL for assets must create a project reference to the RCL *after* the RCL is created.

For guidance on creating an RCL, see Consume ASP.NET Core Razor components from a Razor class library (RCL). Optionally, access the additional guidance on RCLs that apply broadly to ASP.NET Core apps in Reusable Razor UI in class libraries with ASP.NET Core.

Share web UI Razor components, code, and static assets

Components from an RCL can be simultaneously shared by web and native client apps built using Blazor. The guidance in Consume ASP.NET Core Razor components from a Razor class library (RCL) explains how to share Razor components using a Razor class library (RCL). The same guidance applies to reusing Razor components from an RCL in a Blazor Hybrid app.

Component namespaces are derived from the RCL's package ID or assembly name and the component's folder path within the RCL. For more information, see ASP.NET Core Razor components. @using directives can be placed in `_Imports.razor` files for components and code, as the following example demonstrates for an RCL named `SharedLibrary` with a `Shared` folder of shared Razor components and a `Data` folder of shared data classes:

razor

```
@using SharedLibrary
@using SharedLibrary.Shared
@using SharedLibrary.Data
```

Place shared static assets in the RCL's `wwwroot` folder and update static asset paths in the app to use the following path format:

> _content/{PACKAGE ID/ASSEMBLY NAME}/{PATH}/{FILE NAME}

Placeholders:

- `{PACKAGE ID/ASSEMBLY NAME}`: The package ID or assembly name of the RCL.
- `{PATH}`: Optional path within the RCL's `wwwroot` folder.
- `{FILE NAME}`: The file name of the static asset.

The preceding path format is also used in the app for static assets supplied by a NuGet package added to the RCL.

For an RCL named `SharedLibrary` and using the minified Bootstrap stylesheet as an example:

> _content/SharedLibrary/css/bootstrap/bootstrap.min.css

For additional information on how to share static assets across projects, see the following articles:

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

The root `index.html` file is usually specific to the app and should remain in the Blazor Hybrid app or the Blazor WebAssembly app. The `index.html` file typically isn't shared.

The root Razor Component (`App.razor` or `Main.razor`) can be shared, but often might need to be specific to the hosting app. For example, `App.razor` is different in the server-side Blazor and Blazor WebAssembly project templates when authentication is enabled. You can add the AdditionalAssemblies parameter to specify the location of any shared routable components, and you can specify a shared default layout component for the router by type name.

Provide code and services independent of hosting model

When code must differ across hosting models or target platforms, abstract the code as interfaces and inject the service implementations in each project.

The following weather data example abstracts different weather forecast service implementations:

- Using an HTTP request for Blazor Hybrid and Blazor WebAssembly.
- Requesting data directly for a server-side Blazor app.

The example uses the following specifications and conventions:

- The RCL is named `SharedLibrary` and contains the following folders and namespaces:
  - `Data`: Contains the `WeatherForecast` class, which serves as a model for weather data.
  - `Interfaces`: Contains the service interface for the service implementations, named `IWeatherForecastService`.
- The `FetchData` component is maintained in the `Pages` folder of the RCL, which is routable by any of the apps consuming the RCL.
- Each Blazor app maintains a service implementation that implements the `IWeatherForecastService` interface.

`Data/WeatherForecast.cs` in the RCL:

C#

```
namespace SharedLibrary.Data;

public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    public string? Summary { get; set; }
}
```

`Interfaces/IWeatherForecastService.cs` in the RCL:

C#

```
using SharedLibrary.Data;

namespace SharedLibrary.Interfaces;

public interface IWeatherForecastService
{
    Task<WeatherForecast[]?> GetForecastAsync(DateTime startDate);
}
```

The `_Imports.razor` file in the RCL includes the following added namespaces:

razor

```
@using SharedLibrary.Data
@using SharedLibrary.Interfaces
```

`Services/WeatherForecastService.cs` in the Blazor Hybrid and Blazor WebAssembly apps:

C#

```
using System.Net.Http.Json;
using SharedLibrary.Data;
using SharedLibrary.Interfaces;

namespace {APP NAMESPACE}.Services;

public class WeatherForecastService : IWeatherForecastService
{
    private readonly HttpClient http;

    public WeatherForecastService(HttpClient http)
    {
```