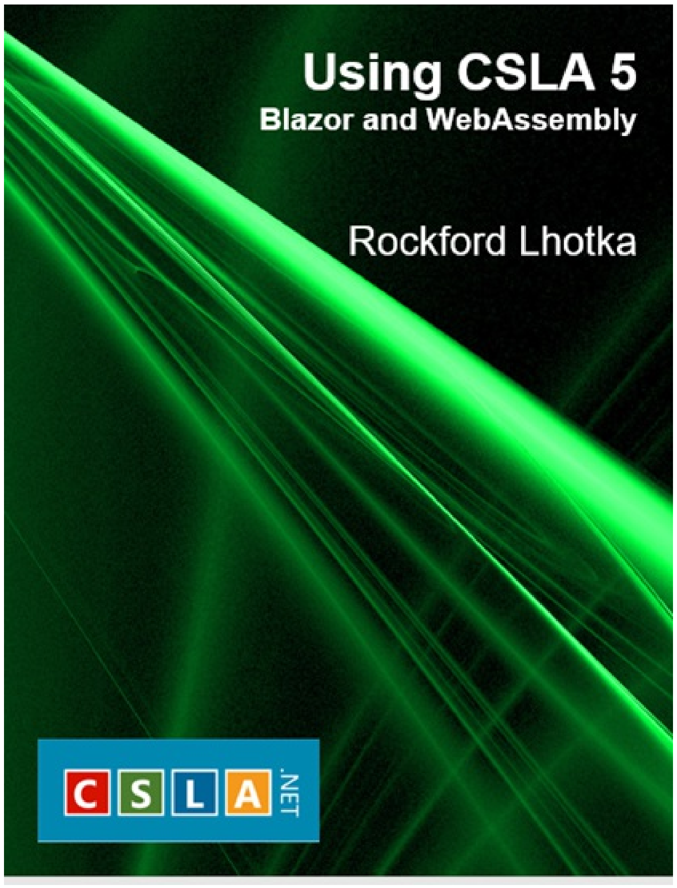
Using CSLA 5: Blazor and WebAssembly



The source code for this book (CSLA .NET version 5.1.0) is available at http:/cslanet.com.

Errata or other comments about this book should be emailed to errata@lhotka.net.

Revision: 200519

Acknowledgments

Magenic

Neither this book, nor CSLA .NET, would have been possible without support from Magenic. Magenic helps you get your digital products to market faster.

You can reach Magenic at https://magenic.com

CSLA .NET Community

CSLA .NET has attracted a community of very thoughtful, intelligent and dedicated people. You can find many of them on the CSLA .NET forum.

The capabilities and features described in this book come, in no small part, through the encouragement and feedback provided by this stellar community.

I would like to specifically acknowledge contributions by Andrew Hallmark (@TheCakeMonster) and @dazinator for their help with implementing Blazor support in CSLA .NET.

Thank you all!

Blazor Community

Blazor also has a very active community, especially in the Blazor gitter channel. Numerous people from this channel have been helpful in getting CSLA .NET working well with Blazor, and in writing this book.

I would like to specifically acknowledge @chrissainty, @tleylan, @shawty, @jspuij and @chucker for their thoughtful and timely help solving challenging problems.

About the Author

Rockford Lhotka is the author of more than 20 books on developing software using the Microsoft platform and technologies. He is a member of the Microsoft Regional Director and Microsoft MVP programs. Rockford speaks at many conferences and user groups around the world. He is the CTO at Magenic, a company that specializes getting your digital products to market faster.

About CSLA .NET

CSLA .NET is a software development framework designed to provide a home for your business logic.

CSLA .NET is open source, licensed under the MIT license.

Organization of the Book

This book is part of the *Using CSLA* book series. The content in this book is written with the assumption that you have read the first four books in the series:

1. *Using CSLA: CSLA .NET Overview*
2. *Using CSLA: Creating Business Objects*
3. *Using CSLA: Data Access*
4. *Using CSLA: Data Portal Configuration*

This book demonstrates how to create basic web applications using Blazor, WebAssembly, and CSLA .NET. The book uses Visual Studio 2019 with .NET Core 3.1, along with Blazor tooling and various NuGet packages.

Blazor supports server-side and client-side development. This book will cover both, but the primary focus will be on client-side Blazor with WebAssembly, enabling the creation of true smart client applications that can run in any modern browser on any operating system.

By the end of the book you will understand how to build basic server-side and client-side Blazor applications that use business domain objects created with CSLA .NET as the application's model.

Sample Code

Sample code used in the book is in the BlazorBook GitHub repository at https://github.com/MarimerLLC/BlazorBook.

Chapters

Chapter 1: Introduction to Blazor and WebAssembly

Blazor is a UI framework from Microsoft, designed to enable productivity in web app development. You can build server-side web apps using Blazor, but the real power of Blazor is that it can be used to create smart client applications that run in any modern browser on any operating system.

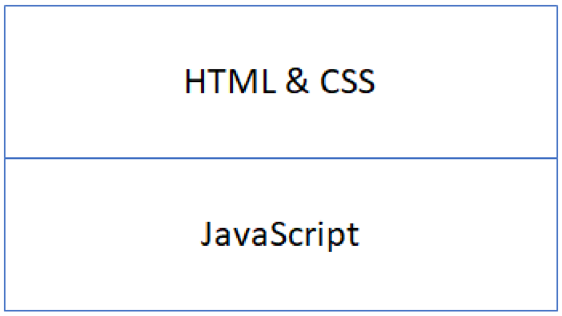
These client-side capabilities are possible thanks to an underlying technology called WebAssembly, combined with a version of .NET that runs in WebAssembly to allow us to run C# code natively in any modern browser.

It is important to lay the groundwork regarding WebAssembly before describing Blazor itself.

About WebAssembly

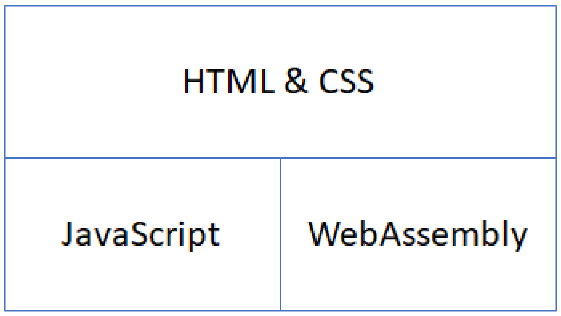
In the late 1990's, as the web started to become mainstream, there were two languages competing to become the standard for scripting client-side behaviors in the browser: JavaScript and VBscript. As time went on, JavaScript became the defacto standard scripting language.

In subsequent years, JavaScript has become more than just a scripting language, and is used as a first class application development language when building smart client apps (often called *single page apps* or SPAs) that run in any browser.



Not everyone finds JavaScript to be an ideal language for software development. As a result, many other languages have been created that transpile into JavaScript. Because browsers used to *only* support JavaScript, it was impossible to compile most languages for use in the browser, and only languages that can transpile into JavaScript have been available.

To overcome this limitation, the WebAssembly initiative brings the ability for a browser to execute JavaScript *and also* execute a type of assembly language code.



It is important to understand that WebAssembly is *not a plug-in* to any browser. WebAssembly is native to every modern browser, exactly like JavaScript. As a result, this technology is entirely *unlike* Flash or Silverlight or other technologies based on a plug-in architecture.

Another important side-effect of WebAssembly being a peer to JavaScript is that your code runs in the *same sandbox as JavaScript*, and so has exactly the same security profile. If you are comfortable running JavaScript in a browser you should be just as comfortable running WebAssembly in a browser.

All modern "evergreen" browsers currently support WebAssembly as well as JavaScript. These include:

- FireFox
- Chromium (Chrome and Edge)
- Safari
- Edge (pre-Chromium)

The power of WebAssembly is that high level languages like C, C++, and most others have always compiled into assembly language. When you compile a C program, the C compiler generates assembly language for the target hardware architecture, which is then compiled to a native binary for the computer.

Now your C code can be compiled to wasm, the WebAssembly assembly language. All browsers understand how to execute that wasm code. In fact, the browsers download the wasm code, then compile it to be native for the client device hardware, whether that is Intel, AMD, or ARM.

The result is that your high level language code ends up running as a native binary on the client device!

Numerous languages currently compile to wasm, including:

- C and C++
- C#
- GoLang
- Java
- PHP
- Python
- Rust
- Swift

and many others.

WebAssembly has the potential to broaden our ability to develop software for the browser by eliminating the single-language mono-culture enforced by JavaScript. This is a very exciting technology that will almost certainly lead to innovation and substantial change around client-side development into the future.

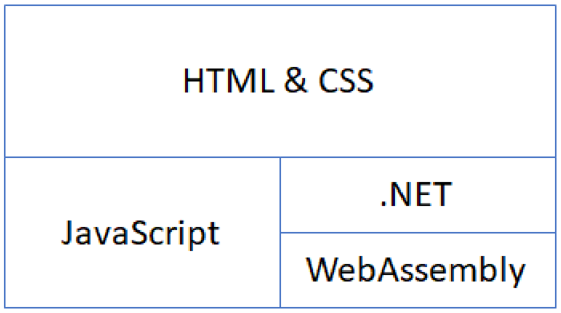
Remember that WebAssembly is a peer to JavaScript; both are nothing more than a way to execute code in a browser. Neither are a UI framework, which is where technologies like Angular, React, Vue, and Blazor come into the picture.

Before discussing the Blazor UI framework however, it is important to understand how .NET runs in the browser via WebAssembly.

.NET Support for WebAssembly

For many of us, the ability to run C or C++ code natively in the browser is an academic curiosity, because we use C# and .NET. Fortunately there is a version of .NET that runs in WebAssembly, and that means that our C# code can run in any modern browser.

Today the way .NET runs in WebAssembly is via the mono open source implementation of .NET. This is also part of Microsoft's strategy for .NET 5, the next major release of .NET planned for 2020.



Microsoft has indicated that .NET 5 will bring .NET Core and mono together, so it is possible that future .NET WebAssembly apps may run on .NET 5 instead of mono. In the near future however, you should expect that your code will run on mono. This should not be a concern, as .NET code in Xamarin runs on mono for iOS, Android, and other platforms. The mono runtime is widely used and stable.

The mono runtime implements .NET Standard 2.1. As a result, the majority of .NET Standard 2.0 and 2.1 assemblies, projects, and NuGet packages are available for your use when building WebAssembly-based apps. This includes CSLA .NET, because the `Csla` NuGet package includes a .NET Standard 2.0 implementation that runs in mono on WebAssembly.

About Blazor

Blazor is a .NET based UI framework designed to help you build rich, interactive web experiences for your users. It runs on top of ASP.NET Core on the server, or on top of mono on WebAssembly on the client.

A Blazor UI is created using HTML, CSS, and C#. Your components are created using a variant of the Razor syntax that has been in use with ASP.NET MVC for many years. In modern ASP.NET MVC, and the new Razor Pages model, pages are defined using `cshtml` files. In the Blazor Razor Components model, components are defined using `razor` files. This is because the Razor syntax used by MVC and Blazor are very similar, *but not identical*.

It is important to understand that Blazor is designed to create highly interactive user experiences, similar to those you might have created in the past with WPF, UWP, Angular, or React. As a result, your .NET code runs in the same context as your components, and can react instantly to UI events that occur due to user interaction. For example, when the user clicks or taps a button, your code is immediately invoked to handle that click event.

Contrast this to a web site model where user interactions in the browser can only be processed by your code after a postback transfers control from the browser to the web server. Blazor doesn't follow this model, and instead follows a model where your code is invoked immediately for all events.

If you are coming from a web development background, Blazor creates what is commonly known as a Single Page Application (SPA). Instead of using Angular, React, or a similar JavaScript-based UI framework, Blazor provides a .NET-based UI framework.

If you are coming from a Windows smart client background, Blazor creates a user experience very comparable to Windows Forms, WPF, or UWP. Instead of forms or XAML pages, you use Blazor to create Razor Components: components or pages with which the user interacts.

Regardless of your previous experience, Blazor provides a UI framework that allows you to build rich, interactive user experiences based on .NET, HTML, and CSS.

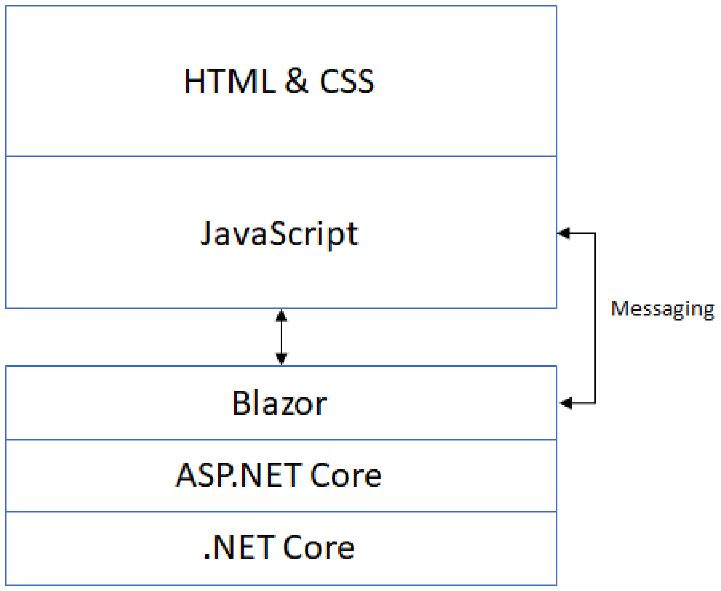
You can choose whether to run your code on the server or on the client. If your users have low-end client devices you may choose to run the code on the server so the browser is little more than a terminal used to render HTML and CSS. If your users have more powerful client devices you may choose to run the code on the client to offload processing power from your server, thus improving scalability and harnessing the substantial compute power available on modern PCs, laptops, tablets, and phones.

Regardless of whether you run your code on the server or client, what is running are your Razor components. The server-side and client-side WebAssembly Blazor environments are basically hosts for Razor Components.

Although Razor Components provides a virtually identical user experience between server-side and client-side Blazor, as a developer you should understand the underlying architecture supporting both scenarios.

Server-Side Blazor

Server-side Blazor rests on top of ASP.NET Core, and all your code executes on the web server. To provide rich client-side interactivity for the end user, a messaging protocol is used to send events back and forth between the browser and the web server.



When the user clicks a button, leaves an input field, or otherwise interacts with the app in the browser, events corresponding to those actions are immediately sent to the web server and your code is invoked to handle the event. These are not traditional postback operations, they are granular events that flow between browser and web server. Any changes your code makes to the page result in messages being immediately sent to the browser so the DOM (browser Document Object Model) can be updated.

To the user, it feels like your code is running on the client, even though it is actually running on the web server.

> For those who've been in this industry long enough, you'll recognize this model as being very similar to mainframe (3270 terminals) and minicomputer (VT terminals) software architectures that were mainstream before client/server and web development became popular. The app code runs on the server, and the browser (in this case) is used as a very capable terminal.

Server-side Blazor relies on SignalR to as a transport for the near-realtime messaging that flows between the browser client and the web server.

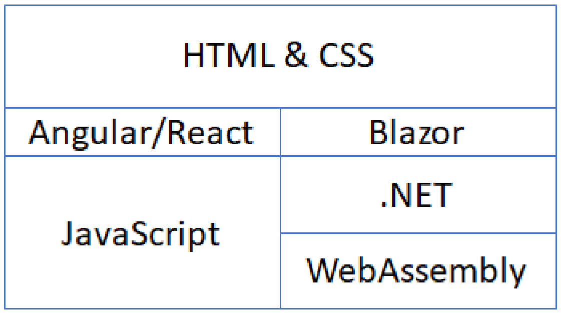
As a developer, you don't directly see or affect the messages used by Blazor to coordinate between the client and server, nor do you write the client-side code that manages the user experience. You write server-side Blazor code and allow the Blazor runtime to do all the work.

The end result is that the user enjoys a highly interactive user experience comparable to a smart client, but all the code runs on the server. This does imply that there's a constant stream of message traffic flowing back and forth between the client and server, which is a factor to keep in mind when you decide between server-side and client-side Blazor deployments for your app.

You should also be aware that server-side Blazor maintains state in memory on the web server. Some state is kept in memory for each user actively using the app. As as a result there are limits on how scalable a server-side Blazor app will be. The amount of memory and server resources consumed per-user varies greatly depending on how you implement your app and what the app is doing, so you'll need to do your own benchmarking to determine how many users can be supported by a web server.

Client-Side Blazor

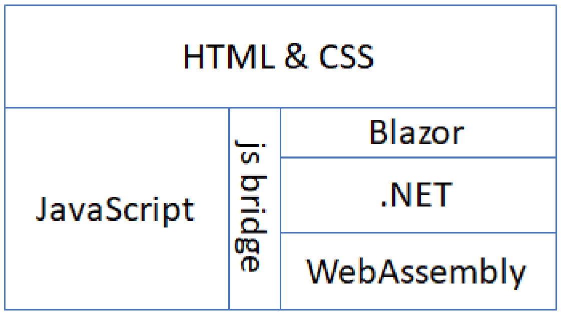
Client-side Blazor runs entirely in the browser, with the Blazor UI framework running on top of mono on WebAssembly. In this regard, client-side Blazor is very similar to Angular or React, in that it is a UI framework that runs in the browser. The primary difference is that your code will be C# instead of JavaScript or TypeScript.



Because all the UI code is running in the browser, there's no messaging needed between the browser and web server. In fact, there's no real need for a traditional web server at all, beyond deploying the html, css, and wasm files to the browser.

> You should be aware that platform or UI specific code will not run in WebAssembly; for example no Windows Forms, WPF, or Xamarin code. Similarly there is no provision for direct access to databases, so no use of `System.Data` to interact with your databases. If you need access to databases you will need to invoke services exposed from an application server.

Although most of the Blazor framework runs on WebAssembly, it does have the ability to interact with JavaScript that is also running in the browser.



This is important for a couple of reasons.

1. WebAssembly can't direct interact with the browser document object model (DOM)
2. You may want to invoke JavaScript libraries from your .NET code
3. You may want to invoke .NET methods from your JavaScript code

Because Blazor is largely a UI framework, it obviously needs to continually interact with the browser DOM, but that can't currently be done directly from WebAssembly. To address this requirement, Blazor includes a "JavaScript bridge" that interacts with the DOM on behalf of the code running in WebAssembly. This bridge passes messages and commands between the JavaScript/DOM and WebAssembly engines in the browser.

Client-side Blazor requires no state on the web server, which probably seems obvious because no web server is required. I call this out however, because it is a major difference between server-side and client-side Blazor.

> ⚠ As with any SPA, mobile, or smart client technology, if your code is running on a user's device it is vulnerable to hacking. This is an issue Windows developers have dealt with since the early 1990's, and mobile and SPA web developers have also dealt with for years. You should be aware of the consequences of running code on a client device, in terms of improved scalability and potentially reduced security.

When you create an app with client-side Blazor you are creating a smart client app, very comparable to Angular, React, WPF, Windows Forms, Xamarin, and other smart client development technologies. Well designed smart client apps that interact with services on an app server can scale far higher with less cost than traditional web sites. This is one key reason architectures based on smart clients are such a great approach.

Conclusion

It has become clear over the past several years that the browser is now the primary development target for client-side app development. WebAssembly breaks us free from the JavaScript mono-culture, allowing us to use many other programming languages, and their corresponding frameworks and tools, to develop software that runs in the browser.

One language and runtime that supports WebAssembly is C# with .NET. The majority of code that targets .NET Standard 2.0 and later will just work in WebAssembly, like it does on Windows, Mac, iOS, Android, and other operating environments.

Blazor is a UI framework built on top of .NET that leverages HTML and CSS for markup and layout, replacing the use of JavaScript or TypeScript with C#. It uses a slight variant of the Razor syntax that has been in use by ASP.NET MVC and Razor Pages developers on the server.

Chapter 2: Server-Side Blazor

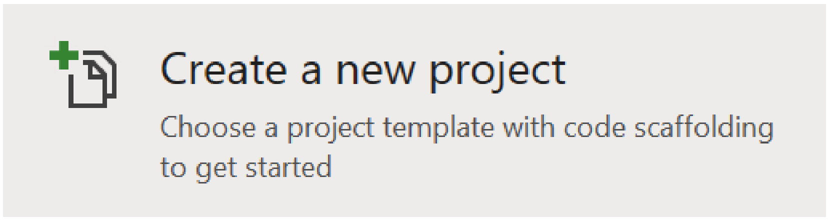
Starting with the basic understanding of Blazor from Chapter 1, I will walk through the basic features of server-side Blazor.

The released version of Visual Studio 2019, as of the end of 2019, directly supports the creation of server-side Blazor projects. Microsoft provides full support for server-side Blazor and Razor Components, alongside ASP.NET MVC, ASP.NET Razor Pages, and other ASP.NET features.

> The code for this example is the `BlazorServerExample` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Creating a Blazor Project

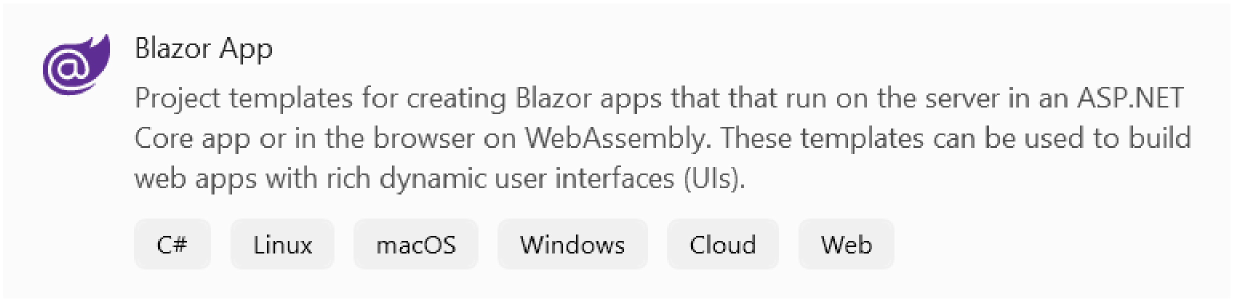
Open Visual Studio 2019 and create a new project.



Create a new project

Choose a project template with code scaffolding to get started

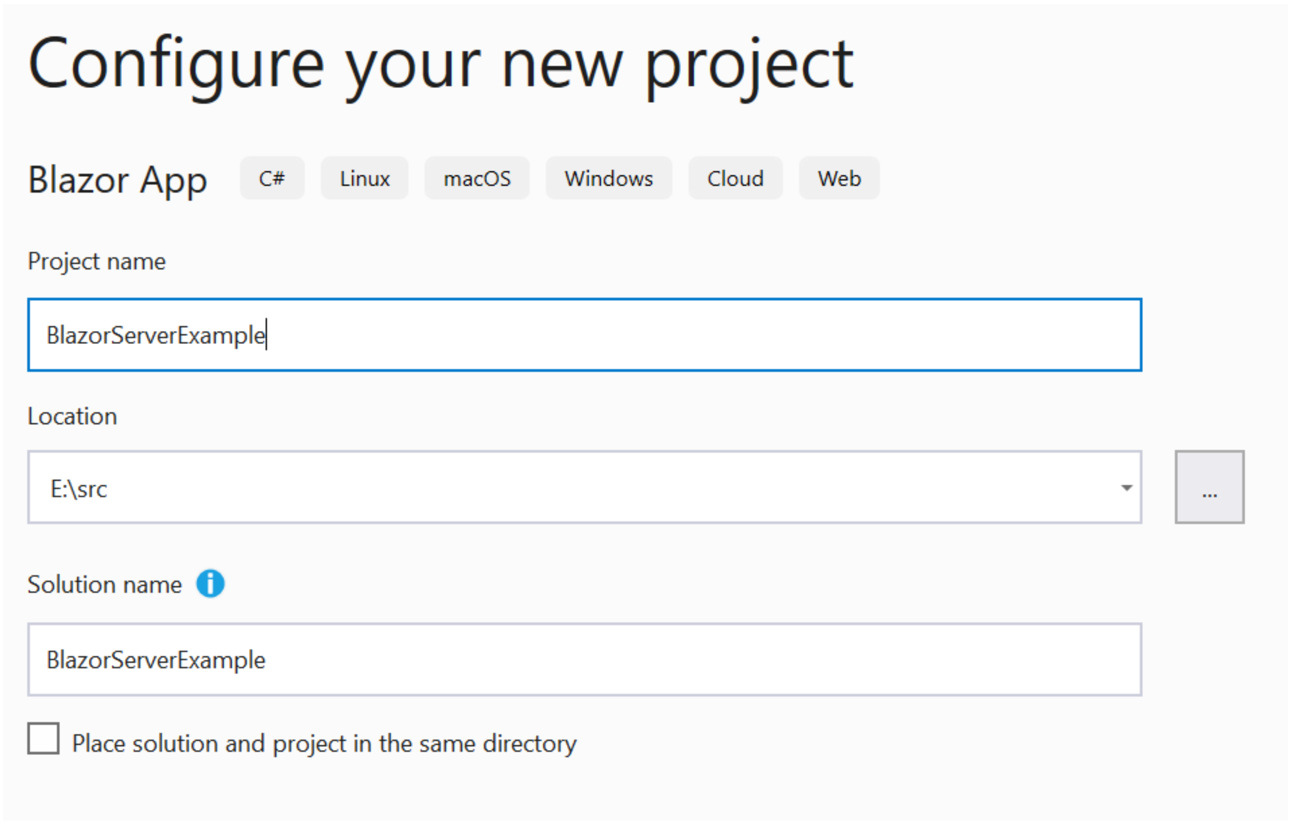
Choose to create a new Blazor project.



Blazor App

Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).

Name the new project `BlazorServerExample` and choose an appropriate location for the solution and project.


Configure your new project

Blazor App

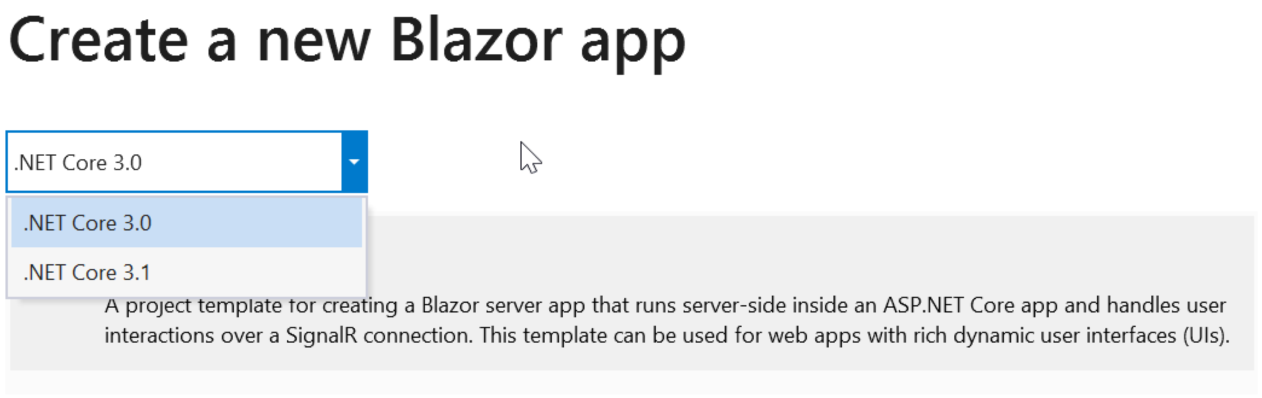
Project name: BlazorServerExample

Location: E:\src

Solution name: BlazorServerExample

☐ Place solution and project in the same directory

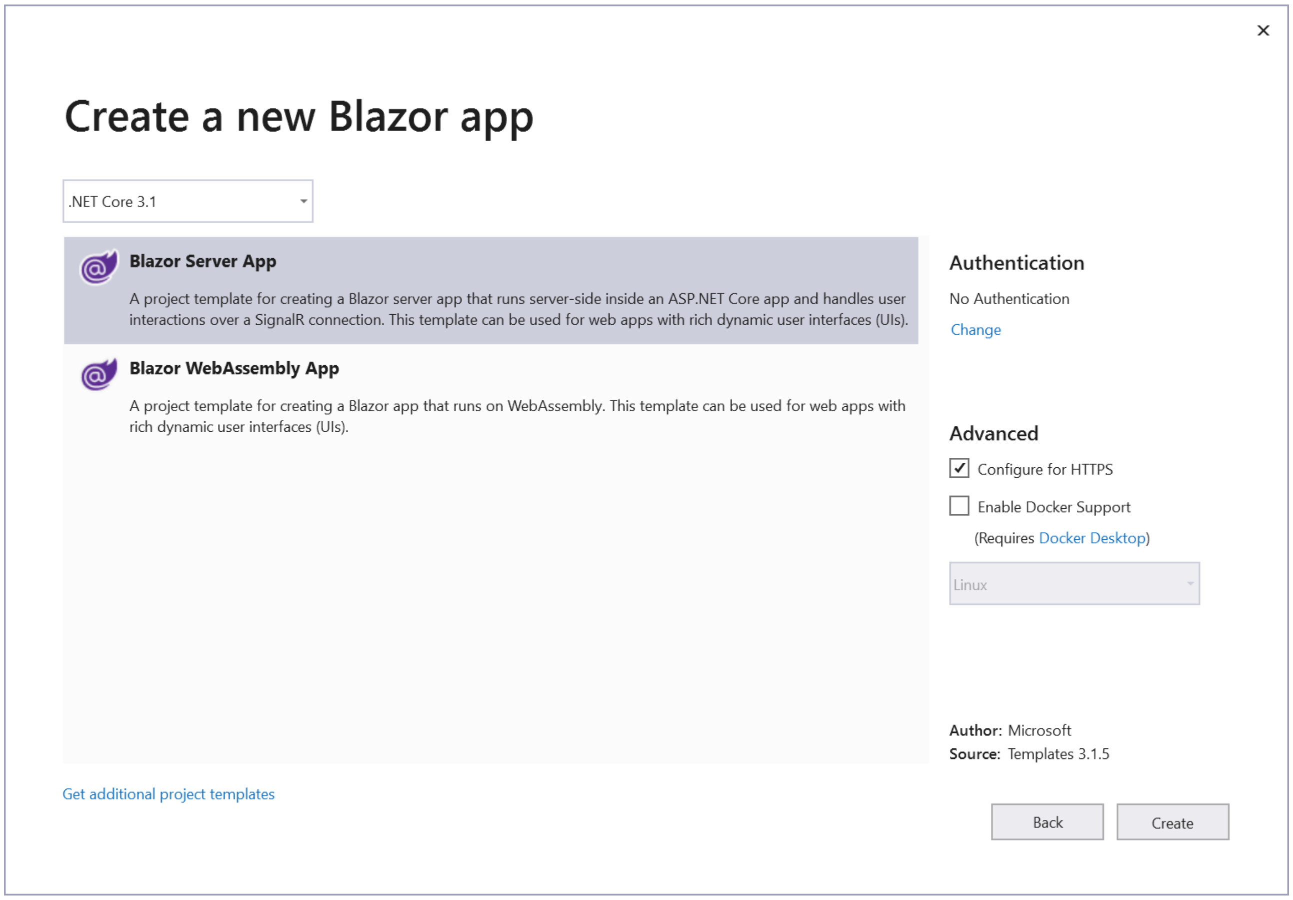
Visual Studio may offer to create a .NET Core 3.0 project, but .NET Core 3.1 is the LTS (long-term support) version of .NET Core, and so should be your preferred choice.


Create a new Blazor app

.NET Core 3.0

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Make sure to select the Blazor Server App option.



Create a new Blazor app

.NET Core 3.1

Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced

☑ Configure for HTTPS

☐ Enable Docker Support

(Requires Docker Desktop)

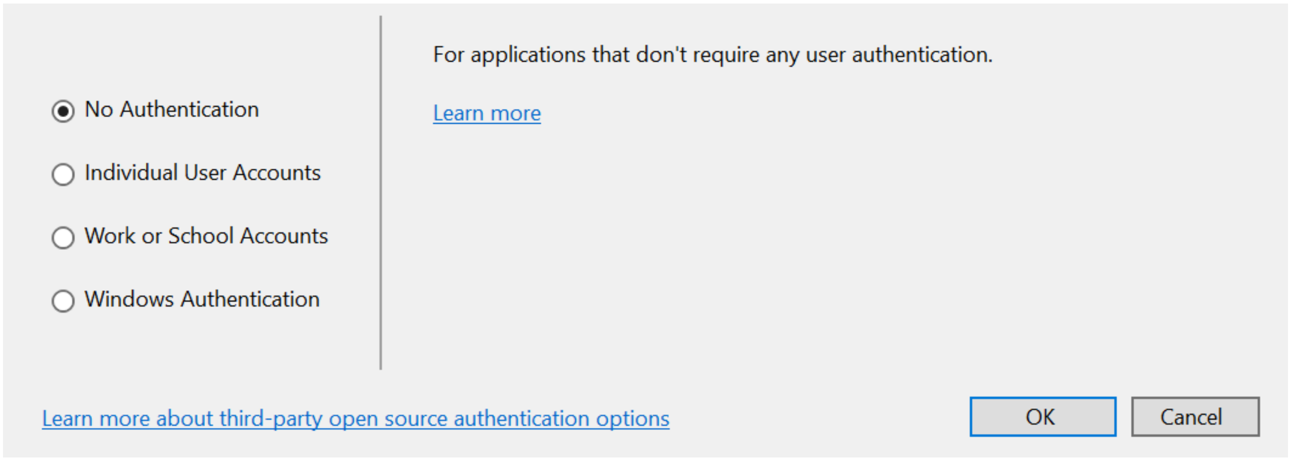
Linux

Author: Microsoft
Source: Templates 3.1.5

Get additional project templates

Back Create

The Authentication option allows you to select between various authentication options supported by the Microsoft project template.



[✓ No Authentication]

Individual User Accounts

Work or School Accounts

Windows Authentication

For applications that don't require any user authentication.

Learn more

Learn more about third-party open source authentication options

OK Cancel

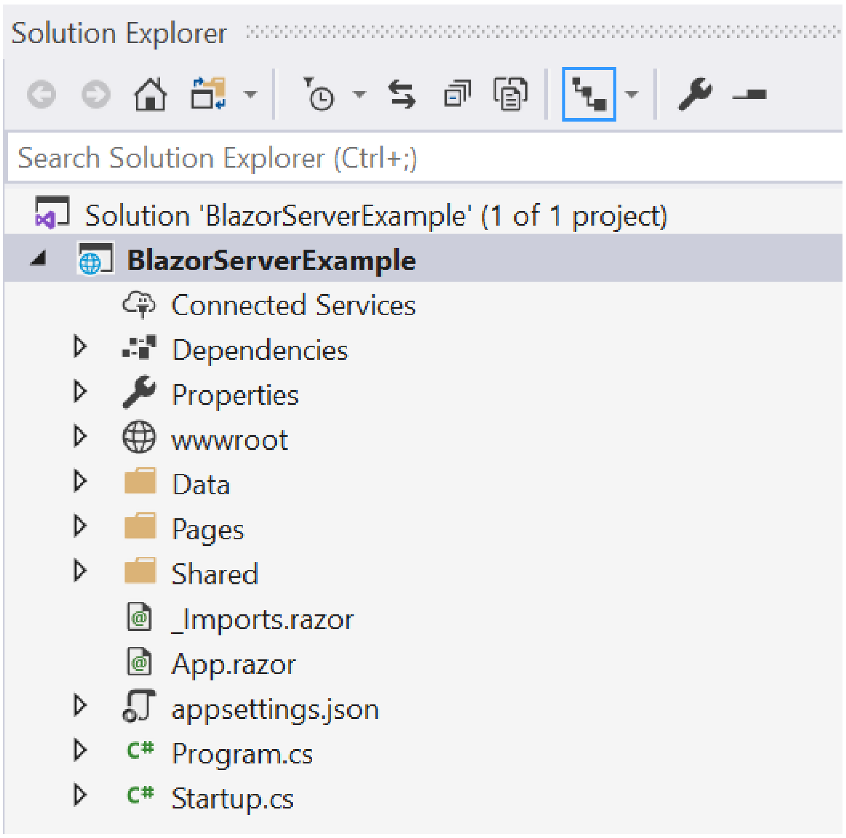
I will discuss the authentication options in Chapter 5. For now leave this at the default No Authentication value.

You will have the option to Configure for HTTPS, and the default is checked. Leave this alone so your web site is configured for HTTPS.

If you have Docker Desktop installed you will see the option to Enable Docker Support. In this example I will assume this option is not checked, so if it is available to you make sure to uncheck it.

Click the Create button to create the solution and project.

The result is a solution with the `BlazorServerExample` project.



Exploring the Project

The Blazor project template creates a fully functional app that demonstrates error handling, basic user interaction, and data retrieval and display.

BlazorServerExample Project File

The project file is a standard ASP.NET Core project file.

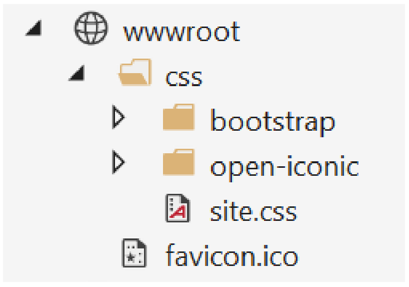
```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Notice how the project uses the `Microsoft.NET.Sdk.Web` namespace to bring in all the necessary references for ASP.NET. This includes support for server-side Blazor, as well as MVC and Razor Pages.

Because you chose to target .NET Core 3.1, the target framework for this project is `netcoreapp3.1`.

wwwroot Folder

The wwwroot folder in the project contains client-side content such as the web site's icon and CSS resources.

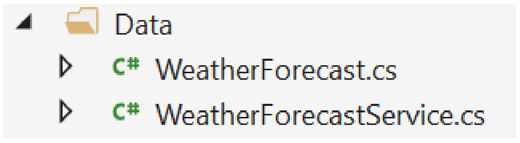

wwwroot
css
bootstrap
open-iconic
site.css
favicon.ico

Blazor itself does not require the use of bootstrap, but the templates are created around those styles. These are *not required* if you change the HTML in each page to use a different CSS framework and assets.

You can change or add CSS assets in this folder to control the layout and other aspects of your web UI.

Data Folder

The Data folder in the project contains sample data, and a mock service that creates the data. In a real app this service would most likely call a remote service via HTTP.


Data
WeatherForecast.cs
WeatherForecastService.cs

In a CSLA .NET based application there is not normally a Data folder, because the app will be built to use a formal business domain layer. I will discuss this in detail in Chapters 7 and 8.

WeatherForecast Class

The `WeatherForecast` class defines a DTO (data transfer object) that is intended to store strongly typed 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; }
}
```

The only behavior in this class beyond storing data, is that it calculates the Fahrenheit temperature from the Celsius temperature.

WeatherForecastService Class

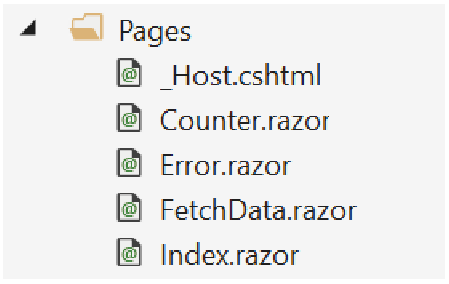
The `WeatherforecastService` class is a mock service that creates sample data.

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

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

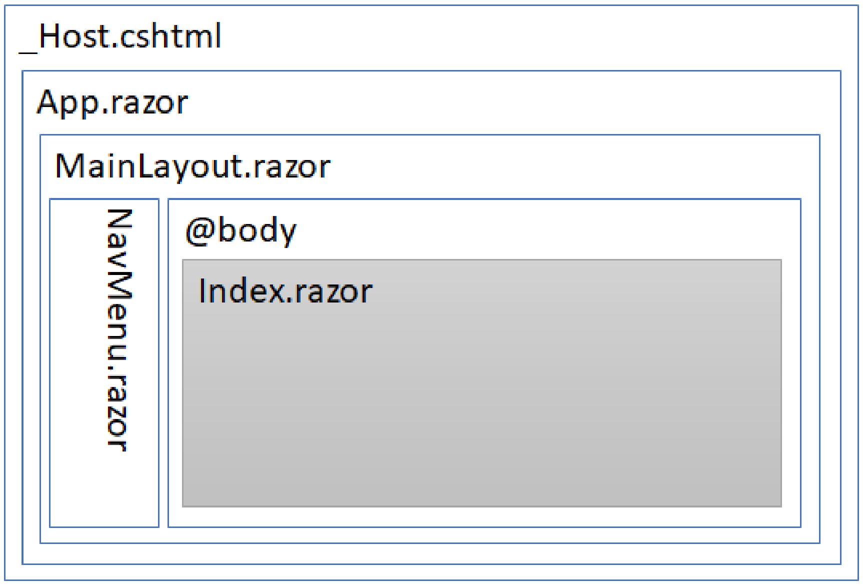
You can imagine how the `GetForecastAsync` method could make a call to a remote server endpoint to retrieve the data. The app consuming the data wouldn't be affected by such a change.

Pages Folder

The Pages folder contains most of the UI displayed to your users. The pages and components in this folder represent web pages, but you should think of them as smart-client "forms" or "pages" much like you would in a Windows Forms, WPF, Xamarin, UWP, Angular, or React app. Although these pages are created using HTML and Razor markup, the Blazor UI framework makes them *feel like* smart client pages, not traditional server-side web pages.



The pages in the Pages folder are rendered for the user in the browser by combining the HTML and Razor markup in each page, plus the CSS defined in the wwwroot folder, plus templates and components defined in the Shared folder I will discuss later in this chapter.



Blazor and Razor Components use a container-based, or nested, approach to constructing the content shown to the user. You can see that the primary content, in this diagram `Index.razor`, is contained within the `MainLayout.razor` component, replacing the `@body` tag. The `MainLayout` component is contained within an `App.razor` component, which is contained within the `_Host.cshtml` page.

The `_Host` and `Index` pages are found in the Pages folder, the other components come from the Shared folder.

_Host Page

The ASP.NET Blazor framework is based on the same ASP.NET Core technologies as MVC and Razor Pages. A server-side Blazor project is a specialized type of web site. As a result, it relies on ASP.NET Core Razor Pages to launch the web site and start up the Blazor framework on the client.

> Remember that server-side Blazor uses SignalR to provide near-realtime communication between the browser and web server. This implies that there is JavaScript code running in the browser to act as a SignalR client and to manage this messaging.

The _Host.cshtml file contains this Razor Pages bootstrap code, including some important elements for you as a Blazor developer.

> As I will discuss in Chapter 3, this is comparable to the `index.html` page in a client-side Blazor project.

This file sets the overall web site title.

```
<title>BlazorServerExample</title>
```

It also brings in the CSS used by the site.

```
<link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
<link href="css/site.css" rel="stylesheet" />
```

Next it controls how error text is displayed to the user in the case that the Blazor UI framework doesn't start properly. Once Blazor is running errors are displayed using the `Error` Razor page.

```
<div id="blazor-error-ui">
    <environment include="Staging,Production">
        An error has occurred. This application may no longer respond until reloaded.
    </environment>
    <environment include="Development">
        An unhandled exception has occurred. See browser dev tools for details.
    </environment>
    <a href="" class="reload">Reload</a>
    <a class="dismiss">🗙</a>
</div>
```

Finally, the Blazor client-side JavaScript framework is loaded to handle the SignalR messaging with the server.

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

In most cases you will only edit this file to update the web site title, improve SEO by changing meta tags, alter the CSS framework or files used by the app, and possibly alter or enhance the error text displayed to the user when the Blazor UI framework can't be loaded during app startup.

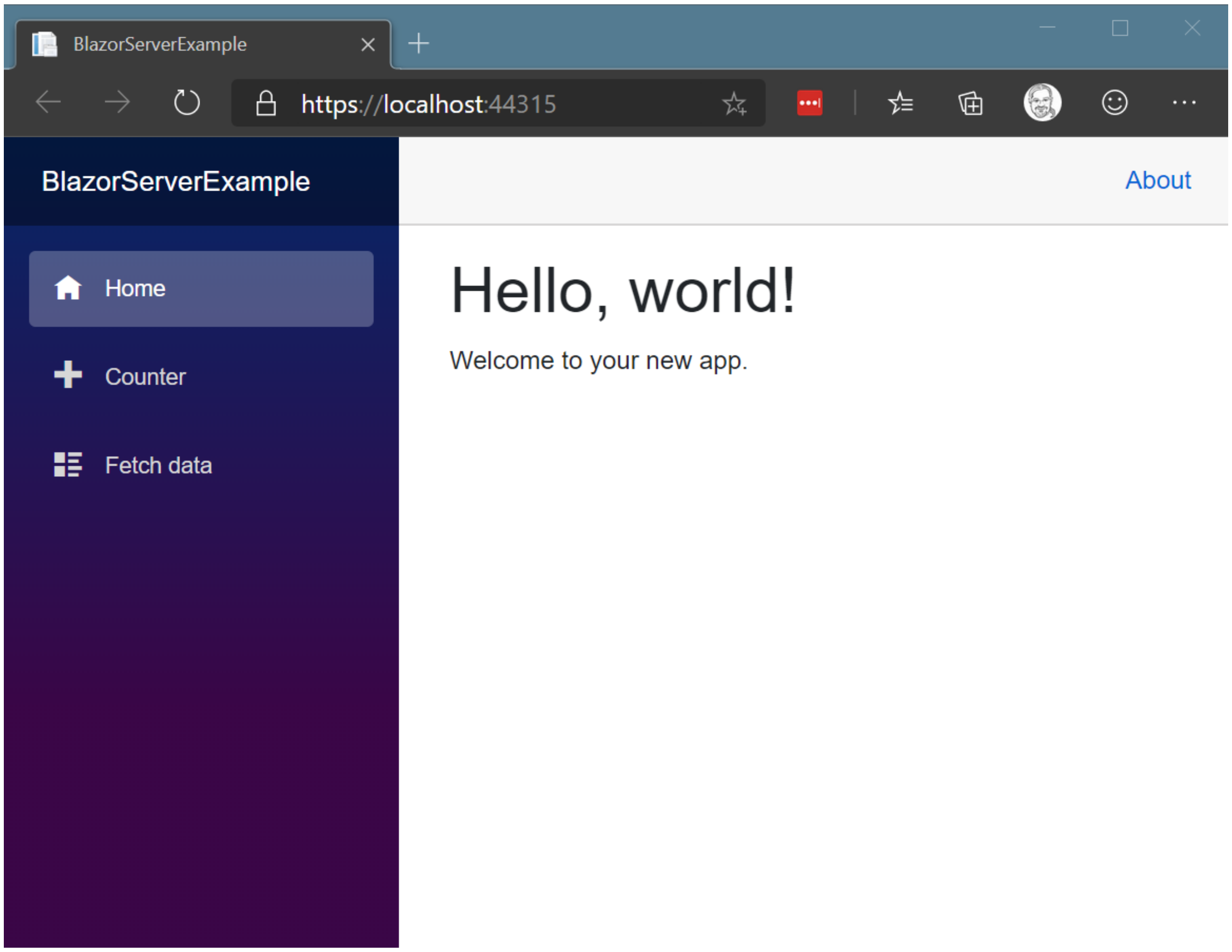
The other components end in a `.razor` postfix because they are handled by the Blazor UI framework, not by Razor

Pages. If the Blazor framework successfully loads during app startup everything from that point forward is handled by Blazor.

Index Component

Each page displayed to the user by Blazor is a combination of the page plus content from the Shared folder. I will discuss the Shared folder later. Right now it is important for you to understand that each page in the Pages folder represents only the specific page content, not other content such as headers, footers, navigation, or anything else outside the page itself.

The first page displayed to the user is the `Index` page.



The template for this page is very simple.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.
```

Notice the `@page` directive at the top of the code. This directive is part of the routing mechanism used by Blazor. In more advanced pages, later in this book, you will see that a page can have multiple `@page` directives if it handles more than one route. The `Index` page handles one route, the default, and so has one directive.

The rest of the page is HTML displaying content to the user. This page *could* have more advanced Razor markup as well, but the basic template simply displays text.

Error Component

The `Error` page is displayed to users when an unhandled error occurs within the app. Like all pages, it starts with a `@page` directive for routing.

```
@page "/error"
```

The rest of the content in the page is default text explaining how to get more detailed information by enabling developer mode. You will want to edit this content to provide meaningful and useful text for your users.

Counter Component

The `Counter` page implements functionality that allows the user to interact with the app. Specifically, it provides a button that the user can click on, and each click increments a numeric value.

```
@page "/counter"

<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++;
    }
}
```

There are two new concepts in this page: the properties on the `button` element, and the `@code` block.

The `button` element displays a standard HTML button, but the `@onclick` property is a special Razor markup item. This property causes Blazor to invoke the `IncrementCount` method each time the button is clicked.

Remember that the button is rendered in the browser, but all code runs on the web server. This means that each time the user clicks the button the Blazor JavaScript framework running in the browser sends a message to the server so the server knows that the `IncrementCount` method should be invoked.

The `@code` block is defining C# code *that runs on the web server*, and is invoked when the Blazor framework sends a message from the browser via SignalR.

> You should be aware that server-side Blazor does maintain state on the web server for each active user of the app. As a result, the state necessary to handle events from the client is available in memory and so event handling is fast and responsive. The potentially negative consequences to this architecture include memory consumption on the server for each user, and loss of in-memory user information if the web server goes down.

When the user clicks the button in the browser, Blazor sends a message to the server. On the server Blazor invokes the `IncrementCount` method in the `@code` block. That method increments the `currentCount` field.

Blazor automatically monitors the values of fields in a page, so it is immediately aware that `currentCount` has changed. The result is that Blazor sends a message to the browser using SignalR, so the Blazor framework code running in the browser can immediately update the value displayed to the user.

From the user's perspective this is all nearly instantaneous. The user will feel like this is a smart client app experience, with all the rich, interactive UI they'd expect from an Angular, React, Windows, or mobile app.

However, behind the scenes you, as a developer, need to remember that all code is running on the server, with Blazor providing the seamless illusion that everything is running on the client.

> It is important to remember that client-side Blazor really *does* run everything on the client!

FetchData Component

The `FetchData` page demonstrates how to have UI code invoke a service to retrieve data, and then how to display that data to the user. It also provides an opportunity for me to discuss how Blazor leverages the dependency injection (DI) service provided by .NET Core.

Near the top of the page there are a couple new directives.

```
@using BlazorServerExample.Data
@inject WeatherForecastService ForecastService
```

The `@using` directive works like the `using` statement in C#, bringing a namespace into scope for use within the page.

The `@inject` directive indicates that Blazor should use the .NET Core DI framework to inject an instance of the `WeatherForecastService` type with the variable name `ForecastService`. You can use the `ForecastService` variable throughout the page to interact with the object provided by dependency injection.

I will show you the code where the `WeatherForecastService` type is mapped to a specific instance later in this chapter when I discuss the `Startup.cs` file.

This service is used in the `@code` block for the page.

```
@code {
    private WeatherForecast[] forecasts;

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

As I discussed with the `Counter` page, fields declared within the page are available for data binding to the UI. In this case, the `forecasts` field is loaded with data from a remote service, so the array of forecast data can be displayed to the user.

You can look at the HTML and Razor markup in the page to see how the `forecasts` field is used.

It is important to understand that the page might render before data is loaded, and so the `forecasts` field will be `null`. Blazor will fail, rather ungracefully, to render a page if data binding tries to bind to a `null` value, so the markup in the page checks to see if the value is `null` before rendering.

```
@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
```

If the value is `null` a message is shown to the user to indicate that data is being loaded in the background, otherwise the value is not `null` and a `foreach` loop is used to build UI for each row of data in the array.

```
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
```

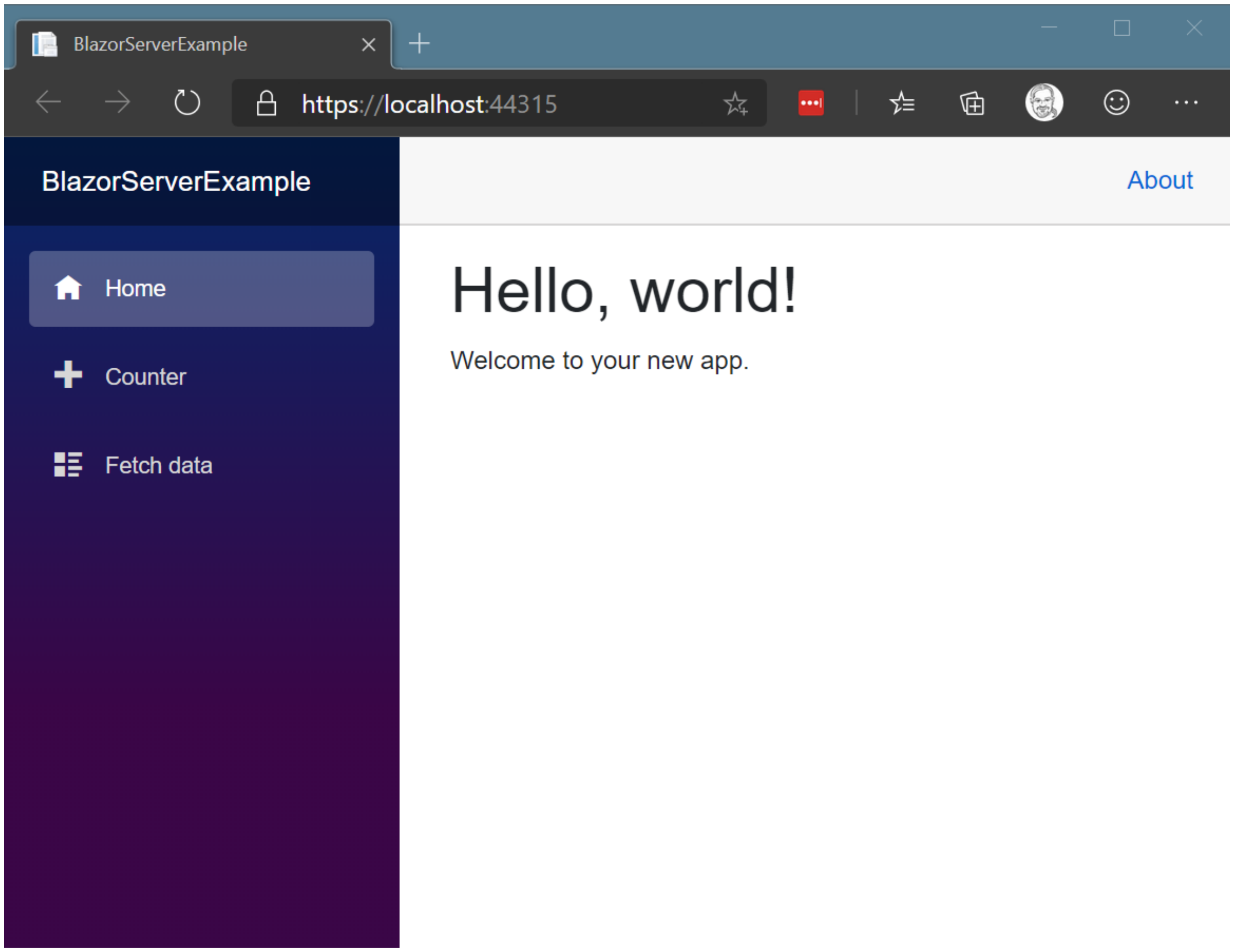
Because this is read-only data, the "data binding" used in this case are simple `@` directives to retrieve data values from each item in the row. Razor markup understands that it should apply `ToString` to any standalone value so that value can be rendered in the browser as text. This means that `@forecast.TemperatureC`, a numeric value, is converted to text and rendered in the browser for the user.

> If you have used Razor markup in ASP.NET MVC or Razor Pages this will be very familiar to you. Although there are Blazor-specific variations for some Razor markup, this use of Razor markup has remained unchanged for many years.

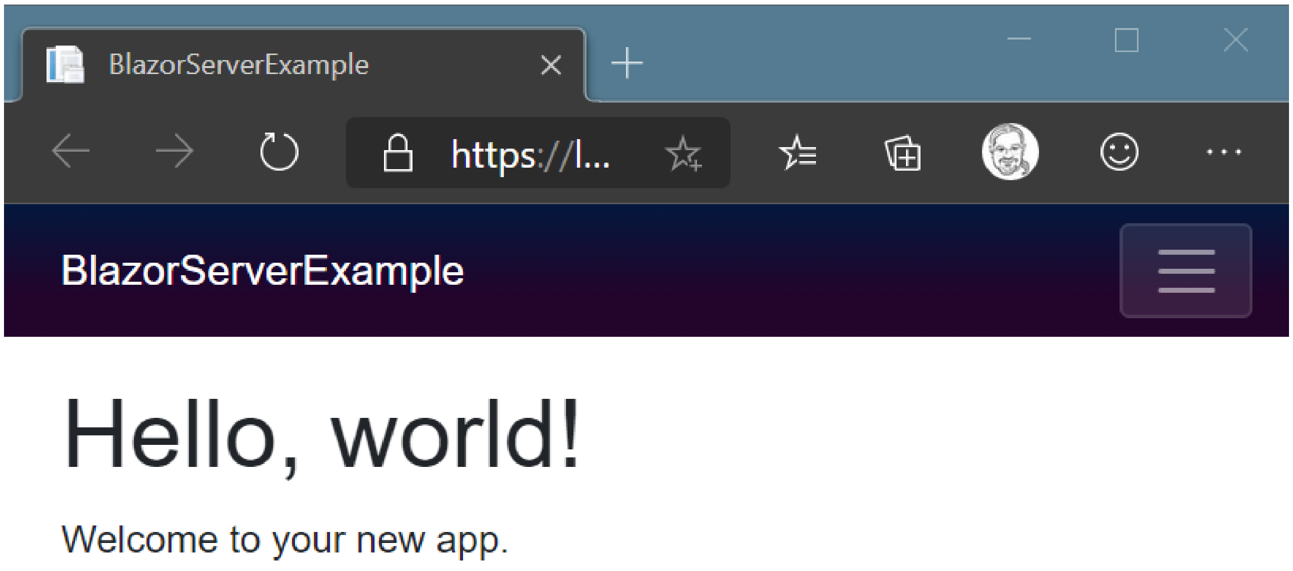
Later in this book I will discuss data binding in much more depth. Data binding can be used to not only display values, but to enable rich two-way bindings between UI elements and fields (or object properties) when creating data entry forms or other types of UI where the user provides input to the app.

Shared Folder

Earlier in this chapter I discussed the `Index` page and how what the user sees in the browser is a combination of each page plus content from the Shared folder. When you look at a page, all the content in the left-hand navigation area, the title and the header with About all come from the files in the Shared folder.



The default Blazor template provides the user with a responsive web experience. For example, when the browser window is small enough, the navigation area is moved to a hamburger menu so the user's focus remains on the content of each page.



Hello, world!

Welcome to your new app.

You can change all this content by editing the files in the Shared folder and the CSS from the wwwroot folder. The Shared folder contains `MainLayout.razor` and `NavMenu.razor` files you will edit to control the overall appearance and navigation for the app.

MainLayout Page

The `MainLayout` page defines the overall layout of all pages in the app.

```
@inherits LayoutComponentBase

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

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

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

You can see how this page specifies where the `NavMenu` and `@body` content will be displayed, as well as any global header content such as the About link.

Although I will discuss this in more detail in Chapter 4, you should be aware that nearly any page can be displayed as a component of another page, just like this page is displaying the content of the `NavMenu` page. The ability to compose pages by using other page/component content is one of the most important features of the Blazor UI framework.

NavMenu Page

The `NavMenu` page not only displays the navigation menu, it also implements events and code to handle expanding the menu when the user clicks the hamburger menu button when it is visible.

The page relies on the `navbar-toggler` CSS style to control whether the hamburger menu button is visible.

```
<div class="top-row pl-4 navbar navbar-dark">
    <a class="navbar-brand" href="">BlazorServerExample</a>
    <button class="navbar-toggler" @onclick="ToggleNavMenu">
        <span class="navbar-toggler-icon"></span>
    </button>
</div>
```

If the button is visible, then when the user clicks on the button the `ToggleNavMenu` method is invoked from the `@code` block on the server.

```
@code {
    private bool collapseNavMenu = true;

    private string NavMenuCssClass => collapseNavMenu ? "collapse" : null;

    private void ToggleNavMenu()
    {
        collapseNavMenu = !collapseNavMenu;
    }
}
```

This code toggles the `collapseNavMenu` value between `true` and `false`. More importantly, you can see how this value is used to change the `NavMenuCssClass` value between `collapse` and `null`. The `collapse` CSS style is used to alter the appearance of the navigation menu content in the page.

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
```

If the text of the `NavMenuCssClass` field is `collapse` then CSS controls whether the navigation menu content is visible to the user. If the value is `null` then the content is visible.

The `collapse` CSS is in the wwwroot folder in the `site.css` file. It is within a `@media` element so it is only expressed for certain screen sizes.

```
@media (min-width: 768px) {
```

When the width is 786px or greater then the sidebar will not collapse, otherwise it collapses.

```
    .sidebar .collapse {
        /* Never collapse the sidebar for wide screens */
        display: block;
    }
```

The interaction between the `NavMenu` HTML and Razor, the CSS styles, and the C# code works together to provide the user with a responsive experience for the navigation menu.

There are also a number of files at the project level. These files contain markup or code that affects the overall Blazor app.

_Imports Page

The `_Imports` page defines `@using` directives that apply to all pages in the app.

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using BlazorServerExample
@using BlazorServerExample.Shared
```

If you consistently use a namespace in your pages and want it to be globally available then you can add it to this file.

App Page

The `App` page manages routing for the app, and controls what the user sees if they navigate to an invalid URL.

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

You may edit this page to alter routing, alter the text displayed to the user if a route doesn't exist, and to implement some authentication and authorization features. I'll discuss authentication and authorization in Chapter 5.

Program.cs

A server-side Blazor app, like any ASP.NET Core web site, can be hosted by IIS, the .NET Core Kestrel web host, or in other hosting environments. In all cases, when the web site is started on the web server the first code that runs is the `Main` method in `Program.cs`.

```
public class Program
{
  public static void Main(string[] args)
  {
    CreateHostBuilder(args).Build().Run();
  }

  public static IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
          .ConfigureWebHostDefaults(webBuilder =>
          {
            webBuilder.UseStartup<Startup>();
          });
}
```

You will probably never edit this code, as it exists to properly bootstrap ASP.NET. If you want to affect how the app starts up or is configured, you will edit the code in `Startup.cs`.

Startup.cs and Configuration

The `Startup.cs` file is the first code run when ASP.NET starts up, and this file is where you will do any configuration or global startup code for the web site.

ConfigureServices Method

The `ConfigureServices` method allows you to configure the .NET Core dependency injection framework, providing the DI framework with type mappings for any types that will be injected into your pages or other code.

```
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddSingleton<WeatherForecastService>();
  }
```

Because this is a server-side Blazor web site, the `AddRazorPages` and `AddServerSideBlazor` methods are required.

The `AddSingleton` method is used to tell the DI framework about the `WeatherForecastService` type. Remember that this type is injected into the `FetchData` page. All that is necessary is for an instance of the type to be created and provided to any pages or code that requests the type, and this `AddSingleton` method call indicates that the DI framework should provide a single instance of the type to any classes requesting the type.

Later in this book you will see more advanced uses of dependency injection and configuration.

Configure Method

The `Configure` method runs after the `ConfigureServices` method, and provides an opportunity to provide other app configuration. Most of this code is standard for any ASP.NET Core web site.

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }
      else
      {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
      }

      app.UseHttpsRedirection();
      app.UseStaticFiles();

      app.UseRouting();

      app.UseEndpoints(endpoints =>
      {
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
      });
    }
```

The app is configured with the error page route, and to use HTTPS. It also is configured to serve up static files, and to use routing.

The `endpoints.MapBlazorHub` and `endpoints.MapFallbackToPage("/_Host")` lines are unique to server-side Blazor, and are required for Blazor to operate.

The `MapBlazorHub` call sets up the server-side endpoint for SignalR that is used by the client-side JavaScript Blazor code.

The `MapFallbackToPage` is triggered any time the user requests a route to a URL that doesn't exist as a physical file. Any time the user asks for content not represented by a physical file on the server they are redirected to the Blazor app. The `MapFallbacktoPage` call sets up the use of the `_Host.cshtml` page so it can bootstrap Blazor as I discussed earlier in this chapter.

Conclusion

At this point you should understand how to create a server-side Blazor project in Visual Studio, and all the folders and files that are provided in the default project template. Let's move on to discuss client-side Blazor projects.

Chapter 3: Blazor WebAssembly

In Chapter 2 you learned how to create a server-side Blazor project, and I discussed all the files and functionality included in the default project template. In this chapter I will go through the same process, focusing on client-side Blazor WebAssembly.

The project template is virtually identical between server-side and client-side Blazor. The only differences relate to how the app starts up and is configured, so there are differences in the `Program.cs` and `Startup.cs` files.

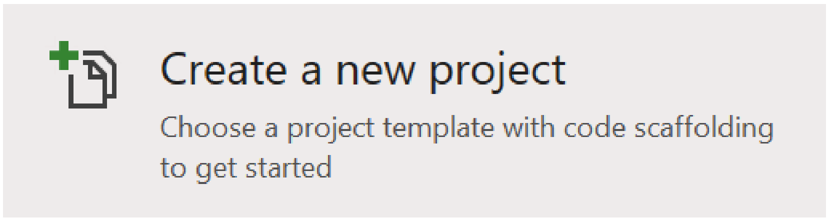
The way the app is hosted and executes is also very different. A server-side Blazor app runs all .NET code on the web server, relying on a Blazor JavaScript framework and SignalR messaging to provide the user with a rich, interactive experience. A client-side Blazor app runs entirely on the client, in the browser. All the .NET code in your app runs in the browser, and any web server is responsible only for deploying the static files containing your code to the browser.

> Your web server might also expose REST or other service endpoints, but from the perspective of the client-side Blazor app, the web server is nothing more than a way to deploy the app to the client device.

> The code for this example is the `BlazorClientExample` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Creating a Blazor Project

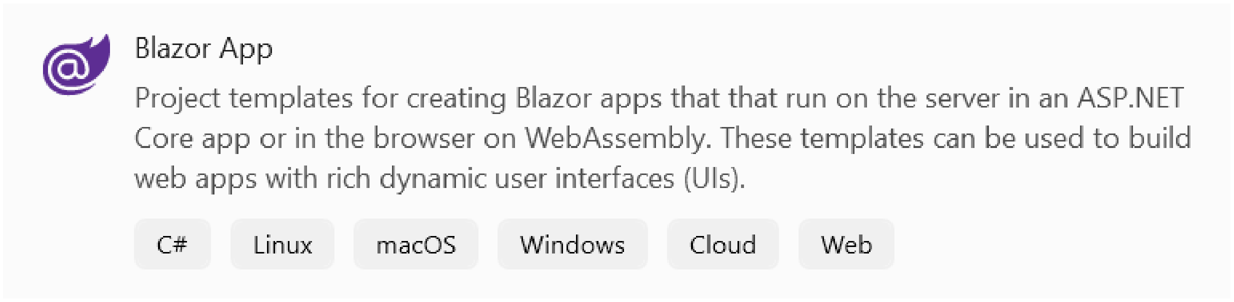
Open Visual Studio 2019 create a new project.



Create a new project

Choose a project template with code scaffolding to get started

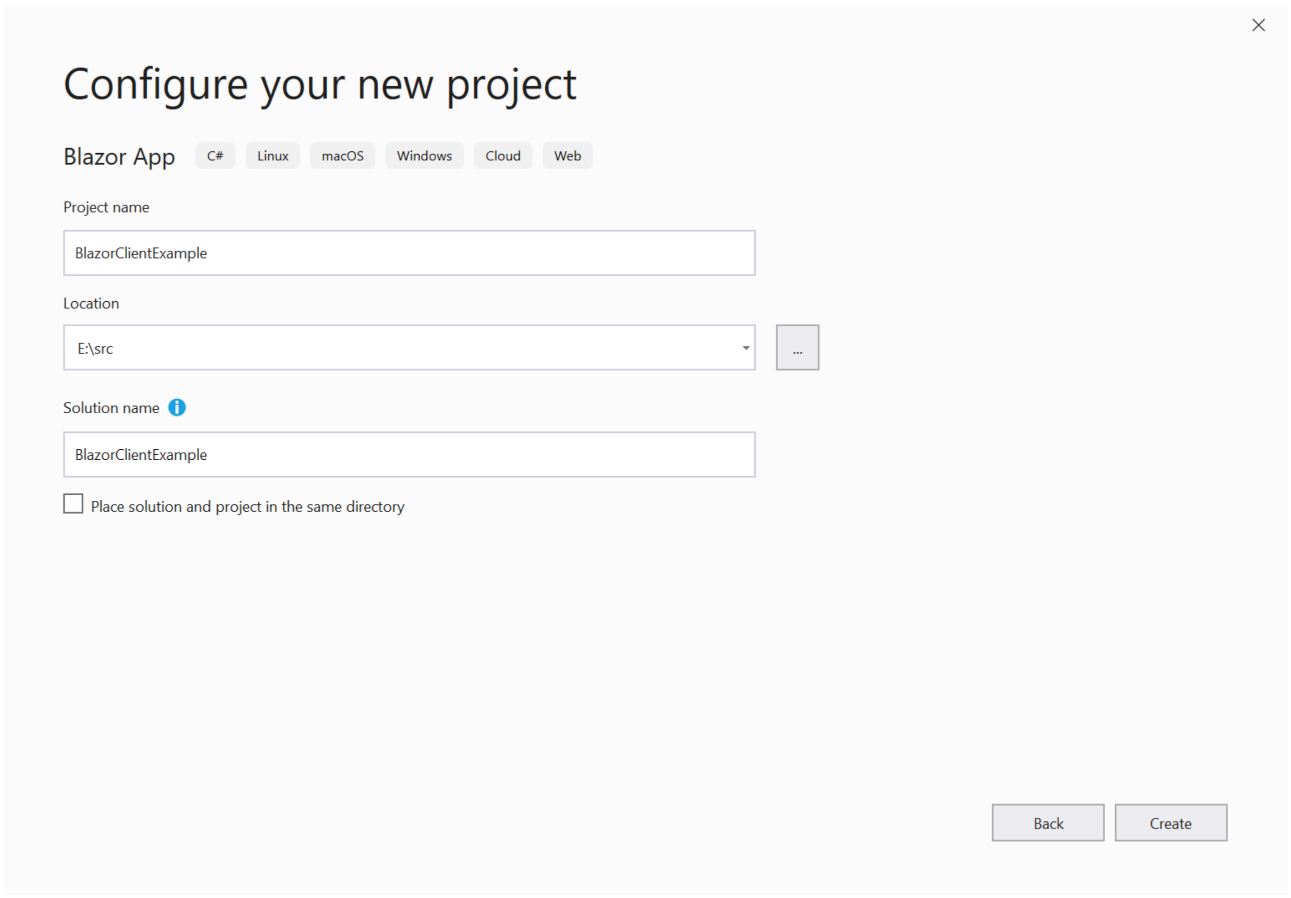
Choose to create a new Blazor project.



Blazor App

Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).

Name the new project `BlazorClientExample` and choose an appropriate location for the solution and project.


Configure your new project

Blazor App

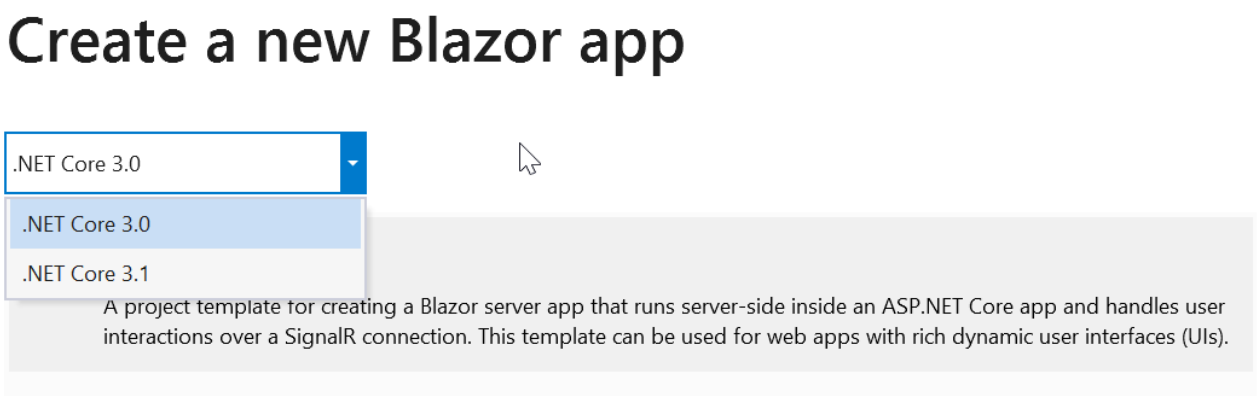
Project name: BlazorClientExample

Location: E:\src

Solution name: BlazorClientExample

☐ Place solution and project in the same directory

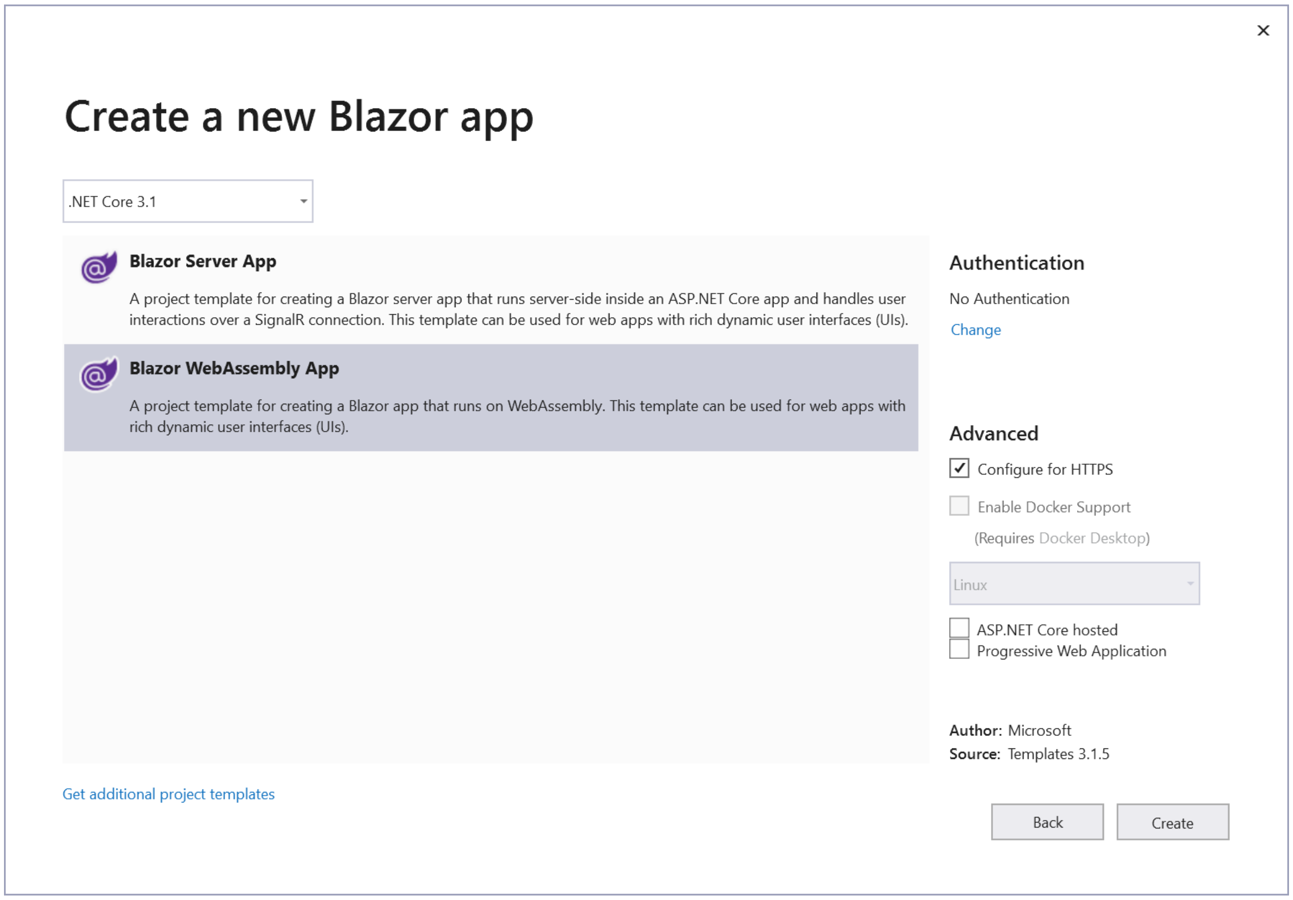
If .NET Core 3.1 is not selected, make sure to select that version. This is the LTS (long-term support) version of .NET Core, and you must select version 3.1 to create a client-side Blazor project.


Create a new Blazor app

.NET Core 3.0

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Now select the Blazor WebAssembly App template.


Create a new Blazor app

.NET Core 3.1

Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced

- [✓] Configure for HTTPS
- [ ] Enable Docker Support

(Requires Docker Desktop)

Linux

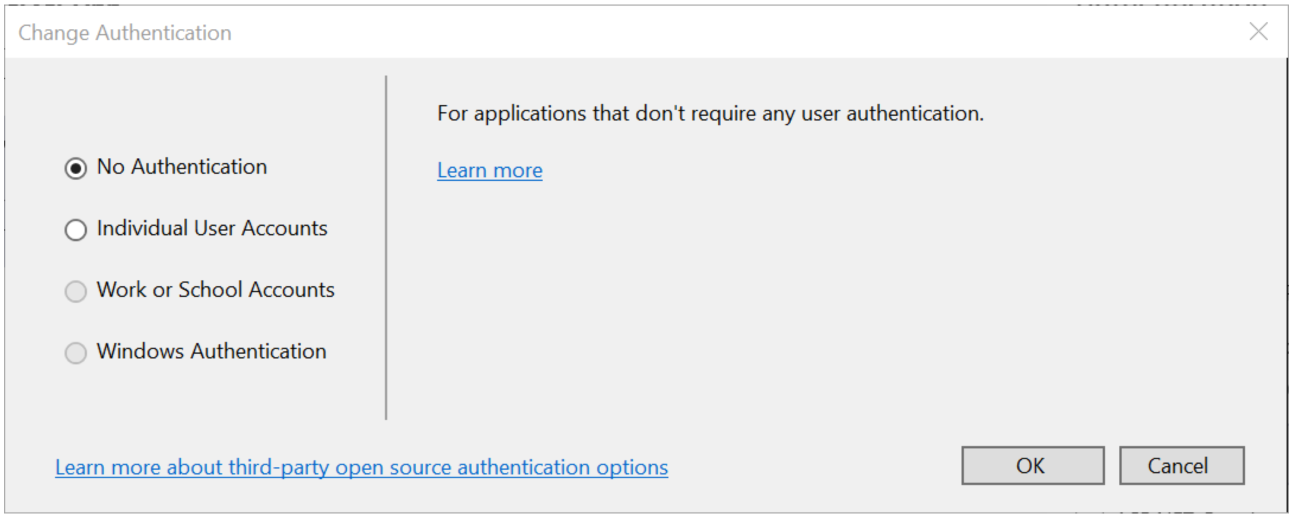
- [ ] ASP.NET Core hosted
- [ ] Progressive Web Application

Author: Microsoft
Source: Templates 3.1.5

Get additional project templates

Back Create

The client-side Blazor project template provides support for either no authentication or individual user account authentication.


Change Authentication

- [✓ No Authentication]
- Individual User Accounts
- Work or School Accounts
- Windows Authentication

For applications that don't require any user authentication.

Learn more

Learn more about third-party open source authentication options

OK Cancel

I will discuss the authentication options in Chapter 5. For now leave this at the default No Authentication value.

You can, and usually should, choose to configure your app for HTTPS so any communication between the web app and the server is encrypted.

If you have Docker Desktop installed you will see the option to Enable Docker Support, but it is always disabled. This is because you are creating an app that will run on the client, not on the server, and so Docker containers are not relevant.

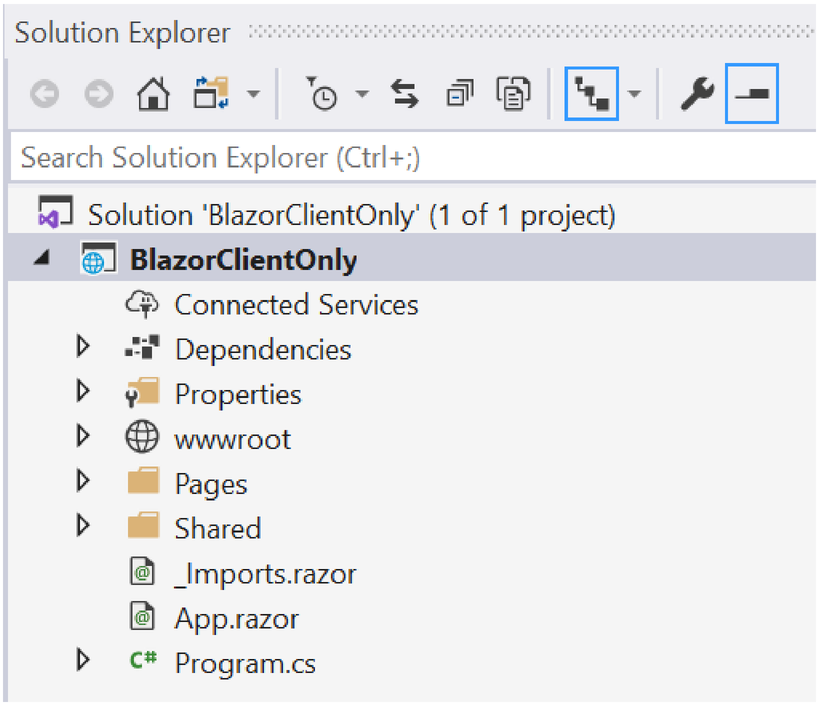
There is an option for the project to be ASP.NET Core Hosted. The choice you make here will change the generated solution in important ways.

⚠ You should check this option, as I will use the ASP.NET Core Hosted approach in this chapter.

The final option in the wizard is Progressive Web Application. If you check this box your app will have additional configuration for deployment to client devices as a PWA (progressive web application). I will discuss this feature later in this chapter.

Standalone WebAssembly Client Project

If you do *not* check the ASP.NET Core Hosted option, the resulting solution would contain one project: the client-side Blazor project.

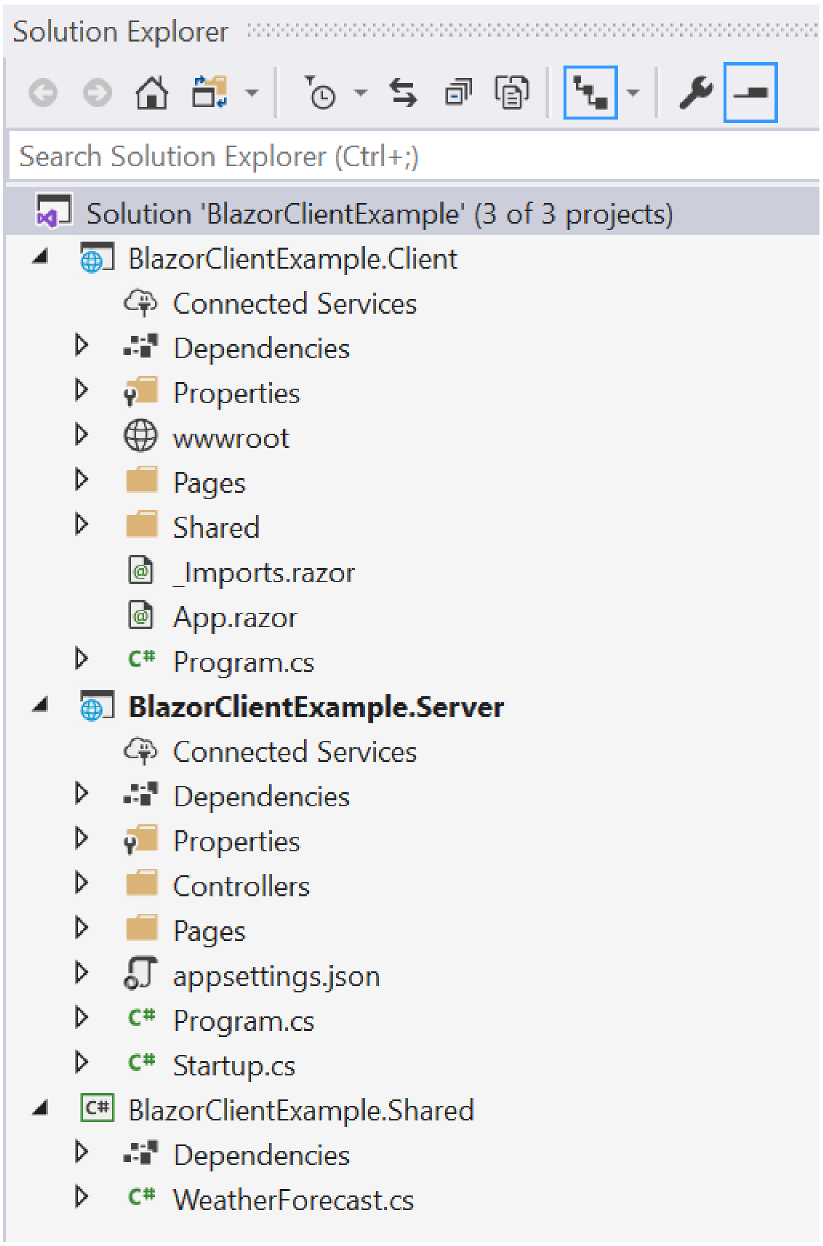


This is similar to the server-side project created in Chapter 2, though there are differences because it is a smart client project, not an ASP.NET Core web project.

When you press ctl-F5 to launch this project, the app is compiled and loaded into your browser. There actually is a tiny web server created for you behind the scenes, but it exists only to deploy the static files for your app onto the browser, and to provide access to other static files such as the sample data for the `FetchData` page.

ASP.NET Core Hosted Project

If you *do* check the ASP.NET Core Hosted option, the resulting solution will contain three projects: a client-side Blazor project, an ASP.NET Core web site that exposes a REST service, and a `BlazorClientExample.Shared` project that creates a .NET Standard 2.1 assembly containing code that runs on both the client and web server.



```
Solution 'BlazorClientExample' (3 of 3 projects)
  BlazorClientExample.Client
      Connected Services
      Dependencies
      Properties
      wwwroot
      Pages
      Shared
      _Imports.razor
      App.razor
      Program.cs
  BlazorClientExample.Server
      Connected Services
      Dependencies
      Properties
      Controllers
      Pages
      appsettings.json
      Program.cs
      Startup.cs
  BlazorClientExample.Shared
      Dependencies
      WeatherForecast.cs
```

The web server project in this solution has two roles. First, it is responsible for deploying the static files containing your app to the browser. Second, it exposes a web API service so the client-side Blazor app can make an `HttpClient` call to retrieve the sample data in the `FetchData` page.

When you launch this solution the web server project is started, just like any ASP.NET Core web project. When the browser loads, the web server will provide it with the static files containing the Blazor WebAssembly app. From that point forward the app is completely running in the browser. The only time the web server is used after the app is loaded, is if the user opens the `FetchData` page, triggering a call to the web server's web API endpoint.

Exploring the Standalone Client Project

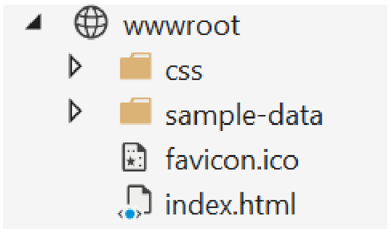
Although I had you select the option to create an ASP.NET Core Hosted solution, I do want to explain how the standalone client template sets up the project. I will focus only on the parts that are different from the ASP.NET Core Hosted solution.

Nearly all the code is identical between the standalone and ASP.NET Core Hosted options, other than how the `FetchData` page retrieves the sample data.

The code for this example is the `BlazorClientOnly` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Sample Data in wwwroot Folder

The files and content in the wwwroot folder are provided to the client-side Blazor app on demand as static files. In the standalone project the sample data is a static file named `weather.json` file in the wwwroot folder.



When you run the a standalone project a small web server is used to serve up static files to the browser, including the app itself, along with all the files in the wwwroot folder.

Retrieving Sample Data in FetchData Page

In the Pages folder there is a `FetchData` page.

Near the top of the page is a directive to inject an `HttpClient` instance via dependency injection.

```
@inject HttpClient Http
```

I'll discuss dependency injection and related configuration later in this chapter. For now it is enough to understand that the `Http` variable contains a reference to a properly initialized `HttpClient` object.

This page has a `@code` block containing C# code that is responsible for retrieving the sample data.

```
    protected override async Task OnInitializedAsync()
    {
        forecasts = await Http.GetJsonAsync<WeatherForecast[]>("sample-data/weather.json");
    }
```

The `GetJsonAsync` method of the `Http` object is used to make a web call to retrieve the static `weather.json` file using a relative URL.

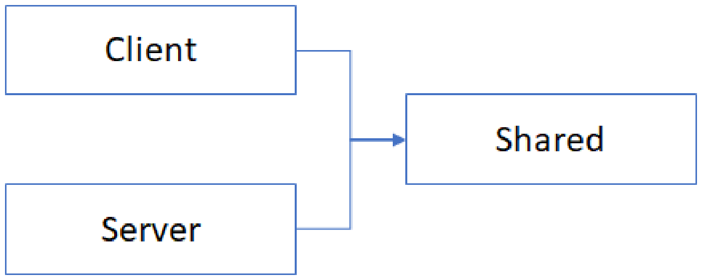
Now that you have seen the two small areas that are different in the standalone template, I will focus the rest of the chapter on exploring the ASP.NET Core Hosted solution.

Exploring the ASP.NET Core Hosted Solution

The solution contains three projects.

1. BlazorClientExample.Server - an ASP.NET Core web site project
2. BlazorClientExample.Shared - a .NET Standard 2.1 class library project
3. BlazorClientExample.Client - a client-side Blazor WebAssembly project

The server and client projects reference the shared project, so the types and code in the shared project is available on both the client and server.



> ℹ If you are familiar with CSLA .NET, the idea of sharing a common assembly between client and server is familiar. This is an approach I have been advocating since CSLA .NET 1.0 was released in 2002.

I will discuss each project and its role in the solution.

ASP.NET Core Web Server Project

As I discussed earlier in this chapter, the ASP.NET Core web server project fills two roles in the solution.

1. Delivering static files to the browser to deploy the client-side Blazor app
2. Exposing a service endpoint so the Blazor app can retrieve weather data

Because this is a standard ASP.NET Core web site, I will not walk through the entire project in detail. There are two parts of the project that exist to support the Blazor client app.

Startup Configuration for Blazor

The standard `Startup` class for configuring the ASP.NET Core web site includes some configuration specific to hosting a client-side Blazor app. These lines of code are in the `Configure` method.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseWebAssemblyDebugging();
  }
  else
  {
    app.UseExceptionHandler("/Error");
    app.UseHsts();
  }

  app.UseHttpsRedirection();
  app.UseBlazorFrameworkFiles();
  app.UseStaticFiles();

  app.UseRouting();

  app.UseEndpoints(endpoints =>
  {
    endpoints.MapRazorPages();
    endpoints.MapControllers();
    endpoints.MapFallbackToFile("index.html");
  });
}
```

The `UseWebAssemblyDebugging` method configures support for debugging Blazor code in the browser. You should understand that the debugging experience for client-side Blazor code is not as robust as server-side Blazor debugging. In Chapter 6 I will discuss a way to build a Blazor solution that can run as a server-side or client-side app. One benefit of that approach is that you can make use of the powerful server-side debugging during development, and still deploy and test your app using WebAssembly.

The `UseBlazorFrameworkFiles` method abstracts the steps necessary for the web server to properly tell the browser to load and launch the client-side Blazor app.

The `MapFallbackToFile` method ensures that when a browser first arrives at the site it is redirected to the `index.html` file in the `wwwroot` folder of the `BlazorClientExample.Client` project. That `index.html` file is a web page that bootstraps the Blazor WebAssembly runtime into the browser so the client-side app loads and runs on the client device.

The goal of this configuration is to ensure that the client-side Blazor app is properly launched in the browser during debugging or when a user attempts to navigate to a URL on the web server.

WeatherForecast Controller

In the Controllers folder the `WeatherForecastController` class implements a REST service endpoint that creates sample data and returns it to the calling code.

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
  private static readonly string[] Summaries = new[]
  {
          "Freezing", "Bracing", "Chilly",
          "Cool", "Mild", "Warm", "Balmy",
          "Hot", "Sweltering", "Scorching"
      };

  private readonly ILogger<WeatherForecastController> logger;

  public WeatherForecastController(ILogger<WeatherForecastController> logger)
  {
    this.logger = logger;
  }

  [HttpGet]
  public IEnumerable<WeatherForecast> Get()
  {
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = rng.Next(-20, 55),
      Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
  }
}
```

This is the same code to create random sample data that was used in the server-side Blazor project template in Chapter 2. In this case, the code is hosted in a server-side web service that can be called from the client app using HTTP.

Shared Project

The shared project is a .NET Standard 2.1 class library that implements code used by the client app and the web server. The solution template includes just one class in this project, but you can put any code into this project that is necessary for the client app and server to operate, including CSLA .NET business domain classes.

The `WeatherForecast` class is the same type used in all the Blazor project templates.

```
public class WeatherForecast
{
  public DateTime Date { get; set; }

  public int TemperatureC { get; set; }

  public string Summary { get; set; }

  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

This class defines strongly typed properties to store data, and implements the conversion from Celsius to Fahrenheit temperatures. It is used by the `WeatherForecastController` in the web server, and in the `FetchData` page in the client app.

Client-Side Blazor Project

The client-side Blazor project template creates a fully functional app that demonstrates error handling, basic user interaction, and data retrieval and display. The code in this project is very similar to the server-side Blazor project from Chapter 2, but there are important differences.

BlazorClientExample.Client Project File

The project file for the client-side Blazor project is more complex than its server-side equivalent from Chapter 2.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly"
                      Version="3.2.0-rc1.20223.4" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Build"
                      Version="3.2.0-rc1.20223.4" PrivateAssets="all" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"
                      Version="3.2.0-rc1.20223.4" PrivateAssets="all" />
    <PackageReference Include="System.Net.Http.Json"
                      Version="3.2.0-rc1.20217.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Shared\BlazorClientExample.Shared.csproj" />
  </ItemGroup>

</Project>
```

The project uses the `Microsoft.NET.Sdk.Web` namespace so it has access to ASP.NET functionality.

Notice that it targets the .NET Standard 2.1 framework instead of `netcoreapp3.1`. This is because the code will be running on top of the mono implementation of .NET, which is running on top of WebAssembly in the browser as I discussed in Chapter 1.

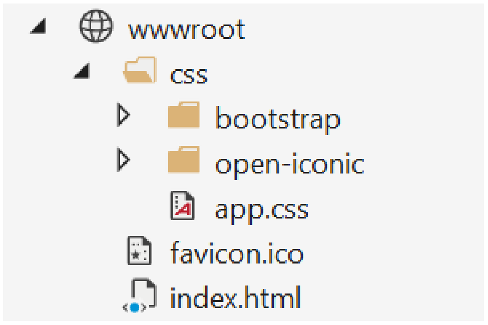
The .NET Framework, .NET Core, and mono all support .NET Standard 2.0 and 2.1, where only .NET Core 3.1 supports `netcoreapp3.1`. This means that any code that runs on mono should target .NET Standard 2.x instead of .NET Core 3.1.

The project also references a number of WebAssembly NuGet packages that provide support for client-side Blazor.

Finally, you can see where the project references the `BlazorClientExample.Shared` project, so the types in that project are available for use in the client-side app.

CSS Assets

The wwwroot folder in the project contains client-side content such as CSS resources. It also contains the start page for the app.



```
wwwroot
  css
    bootstrap
    open-iconic
    app.css
  favicon.ico
  index.html
```

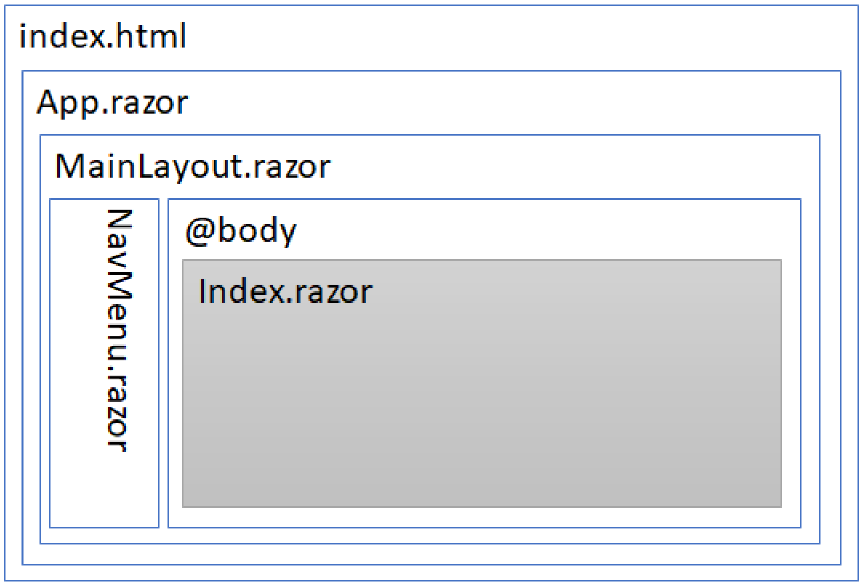
> Blazor itself does not require the use of bootstrap, but the templates are created around those styles. These are *not required* if you change the HTML in each page to use a different CSS framework and assets.

You can change or add CSS assets in this folder to control the layout and other aspects of your web UI.

Index HTML Page

The `index.html` page is the start page for the app. This is the page that is first loaded by the browser, and it hosts the Blazor WebAssembly framework and your app.

> This is the equivalent to the `_Host.cshtml` page in a server-side Blazor project.



In Chapter 2 I describe how Blazor and Razor Components use a container-based model for composing the content shown to the user. The `index.html` page is the outer-most container, and it hosts all the Blazor content.

Your page content, such as `Index.razor`, is contained within the `MainLayout.razor` component, which is contained within the `App.razor` component.

Here's the markup for the `index.html` page.

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0,
                   maximum-scale=1.0, user-scalable=no" />
    <title>BlazorClientExample</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
</head>

<body>
    <app>Loading...</app>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

This file sets the overall web site title.

```
    <title>BlazorClientExample</title>
```

It also brings in the CSS used by the site.

```
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
```

Next it displays some content shown to the user while the app is being loaded into the browser. Because this is a "single page app" (SPA), it can take some time to download the app's assets, resources, and code before the app can launch. You can think of this content as the "splash page" for your app.

If an error occurs during app download or startup, the `index` page controls how error text is displayed to the user.

```
    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>
```

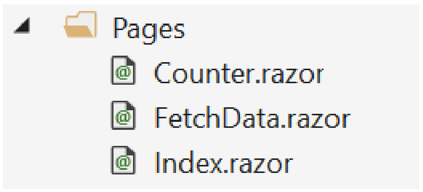
Finally, the Blazor framework JavaScript is loaded.

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

Although most of Blazor runs on mono in WebAssembly, it does use JavaScript code to bridge between the WebAssembly and JavaScript engines in the browser. I discussed the need for this in Chapter 1.

Pages Folder

The Pages folder contains most of the UI displayed to your users. The pages in this folder represent web pages, but you should think of them as smart-client "forms" or "pages" much like you would in a Windows Forms, WPF, Xamarin, UWP, Angular, or React app. Although these pages are created using HTML and Razor markup, the Blazor UI framework makes them *feel like* smart client pages, not traditional server-side web pages.



The pages in the Pages folder are rendered for the user in the browser by combining the HTML and Razor markup in each page, plus the CSS defined in the wwwroot folder, plus templates and components defined in the Shared folder I will discuss later in this chapter.

Notice that some pages from the server-side Blazor project are not needed in a client-side Blazor project.

The `_Host` page is replaced by the `index` page in the wwwroot folder, as I discussed earlier in this chapter.

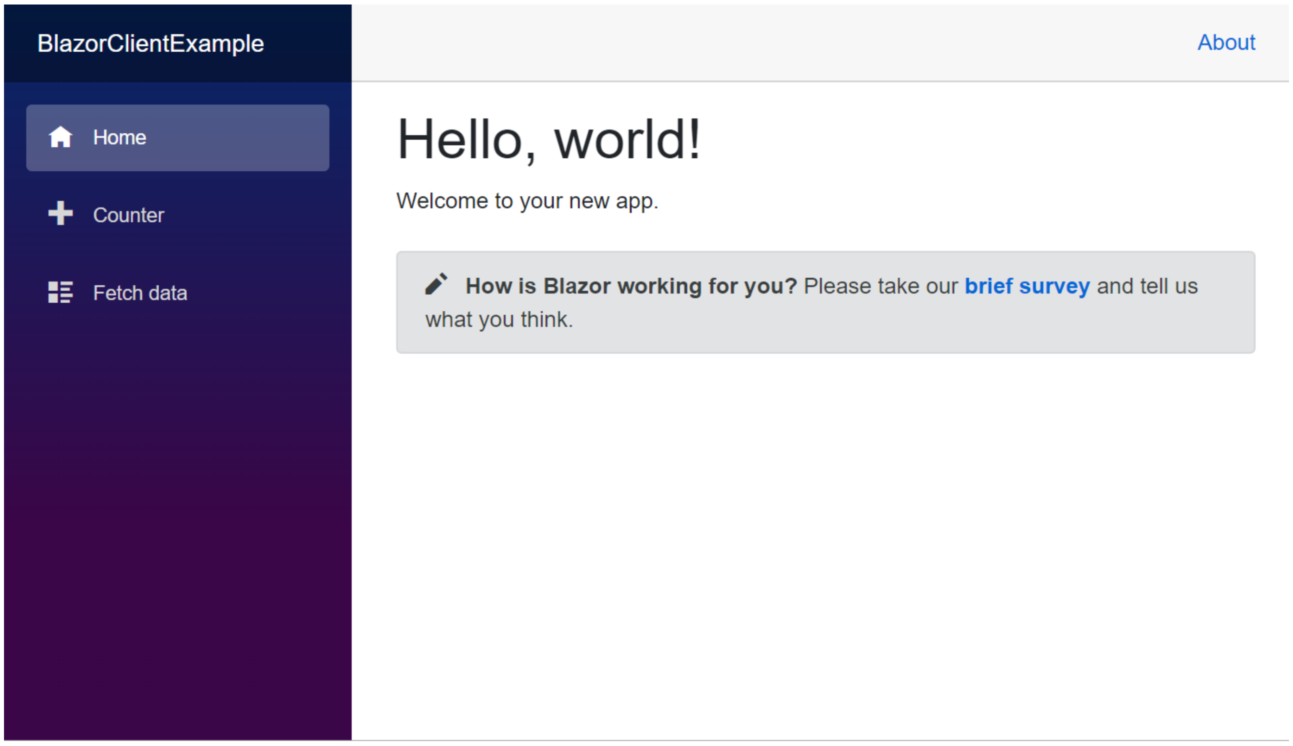
The primary need for an error page in a web site is due to the potential for unforeseen server errors. Because a client-side Blazor app runs entirely on the client device, there is no such thing as a server error, and so there is no need for the `Error` page.

The `Index`, `Counter`, and `FetchData` pages are very similar to their server-side counterparts.

Index Page

Each page displayed to the user by Blazor is a combination of the page plus content from the Shared folder. I will discuss the Shared folder later. Right now it is important for you to understand that each page in the Pages folder represents only the specific page content, not other content such as headers, footers, navigation, or anything else outside the page itself.

The first page displayed to the user is the `Index` page.



Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our **brief survey** and tell us what you think.

The template for this page is very simple.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Notice the `@page` directive at the top of the code. This directive is part of the routing mechanism used by Blazor. In more advanced pages, later in this book, you will see that a page can have multiple `@page` directives if it handles more than one route. The `Index` page handles one route, the default, and so has one directive.

Most of the page is HTML displaying content to the user. This page *could* have more advanced Razor markup as well, but the basic template simply displays text.

The `SurveyPrompt` element is referencing something called a *Blazor component*, which is a reusable bit of UI. I'll discuss the `SurveyPrompt` component later in this chapter, as it is located in the Shared folder of the project. And I'll discuss how you can create and use Blazor components in more depth in Chapter 4.

Counter Page

The `Counter` page implements functionality that allows the user to interact with the app. Specifically, it provides a button that the user can click on, and each click increments a numeric value.

```
@page "/counter"

<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++;
    }
}
```

There are two new concepts in this page: the properties on the `button` element, and the `@code` block.

The `button` element displays a standard HTML button, but the `@onclick` property is a special Razor markup item. This property causes Blazor to invoke the `IncrementCount` method each time the button is clicked.

Remember that all rendering and code executes on the client in the browser. This means that each time the user clicks the button the `IncrementCount` method is directly invoked on the client. The method increments the `currentCount` field.

Blazor automatically monitors the values of fields in a page, so it is immediately aware that `currentCount` has changed. Because the value has changed, Blazor updates the display so the user sees the change.

FetchData Page

The `FetchData` page demonstrates how to have UI code invoke a service to retrieve data, and then how to display that data to the user. It also provides an opportunity for me to discuss how Blazor leverages the dependency injection (DI) service provided by .NET Core.

Near the top of the page there are a couple new directives.

```
@page "/fetchdata"
@using BlazorClientExample.Shared
@inject HttpClient Http
```

The `@using` directive works like the `using` statement in C#, bringing a namespace into scope for use within the page.

The `@inject` directive indicates that Blazor should use the .NET Core DI framework to inject an instance of the `HttpClient` type with the variable name `Http`. You can use the `Http` variable throughout the page to interact with the object provided by dependency injection.

I will show you the code where the `HttpClient` type is mapped to a specific instance later in this chapter when I discuss the `Startup.cs` file.

The `HttpClient` instance is used in the `@code` block in the page.

```
@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await Http.GetJsonAsync<WeatherForecast[]>("WeatherForecast");
    }

}
```

You can see how the `GetJsonAsync` method is used to call the code in the ASP.NET Core web site's `WeatherForecastController`. Remember that controller generates sample data and returns it as JSON.

As I discussed with the `Counter` page, fields declared within the page are available for data binding to the UI. In this case, the `forecasts` field is loaded with data from a remote service, so the array of forecast data can be displayed to the user.

You can look at the HTML and Razor markup in the page to see how the `forecasts` field is used. A `foreach` loop is used to build UI for each row of data in the array.

```
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
```

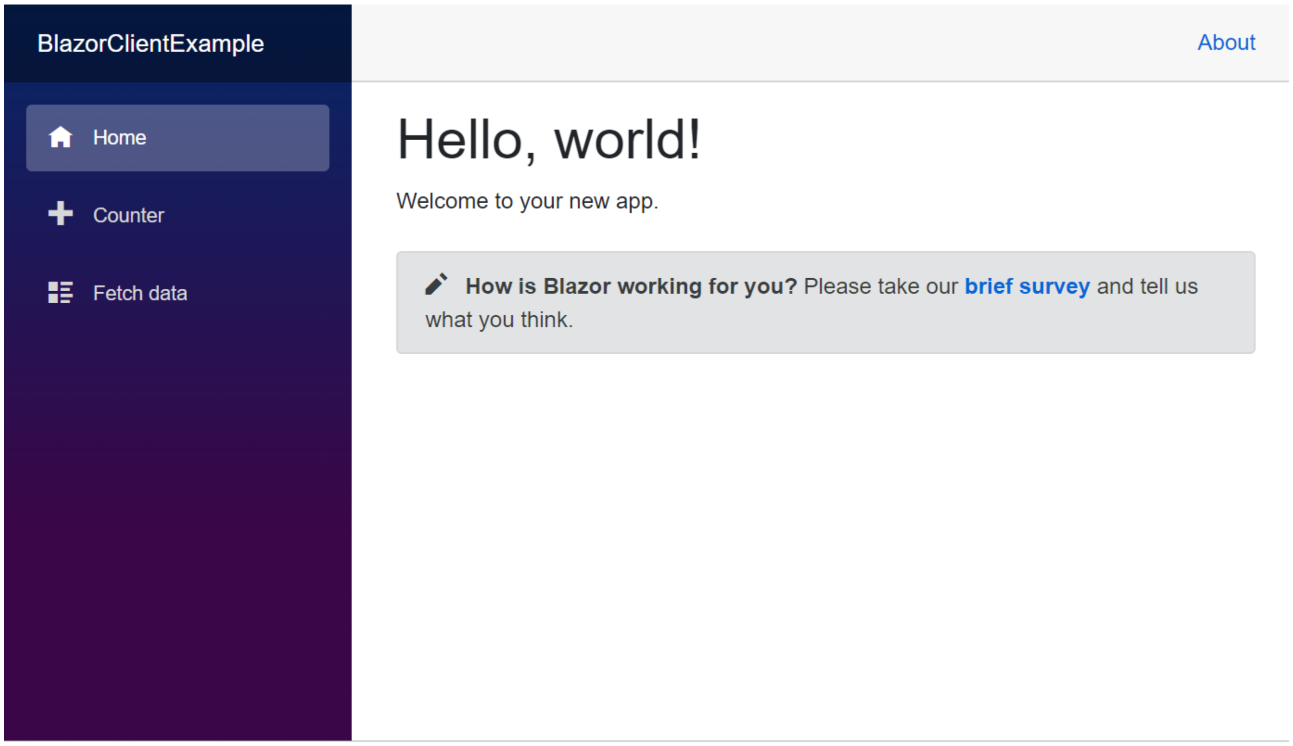
Because this is read-only data, the "data binding" used in this case are simple `@` directives to retrieve data values from each item in the row. Razor markup understands that it should apply `ToString` to any standalone value so that value can be rendered in the browser as text. This means that `@forecast.TemperatureC`, a numeric value, is converted to text and rendered in the browser for the user.

> If you have used Razor markup in ASP.NET MVC or Razor Pages this will be very familiar to you. Although there are Blazor-specific variations for some Razor markup, this use of Razor markup has remained unchanged for many years.

Later in this book I will discuss data binding in much more depth. Data binding can be used to not only display values, but to enable rich two-way bindings between UI elements and fields (or object properties) when creating data entry forms or other types of UI where the user provides input to the app.

Shared Folder

Earlier in this chapter I discussed how the browser displays a combination of each page plus content from the Shared folder. When you look at a page, all the content in the left-hand navigation area, the title and the header with About all come from the files in the Shared folder.

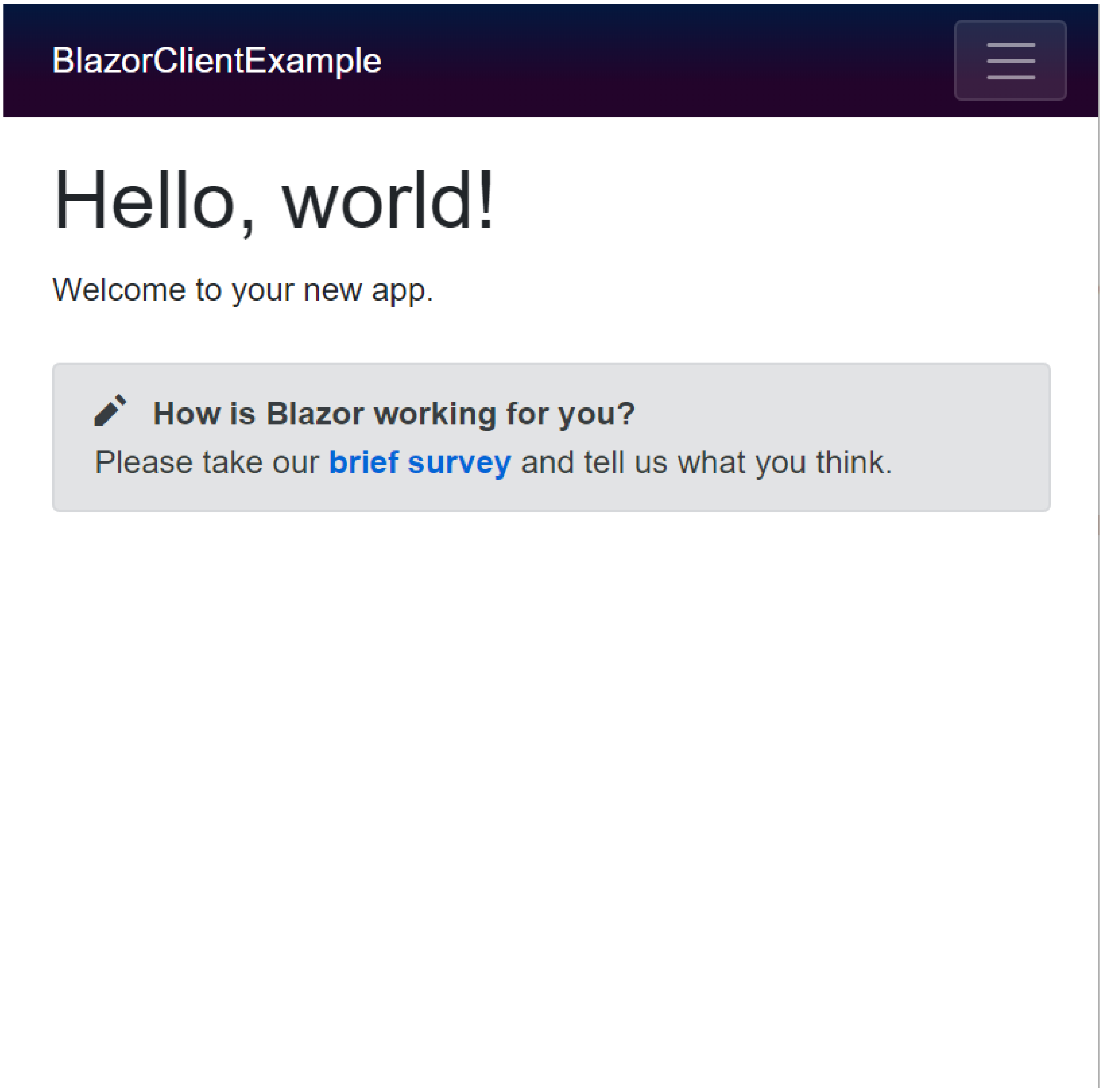


Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our **brief survey** and tell us what you think.

The default Blazor template provides the user with a responsive web experience. For example, when the browser window is small enough, the navigation area is moved to a hamburger menu so the user's focus remains on the content of each page.


BlazorClientExample

Hello, world!

Welcome to your new app.

How is Blazor working for you?

Please take our brief survey and tell us what you think.

You can change all this content by editing the files in the Shared folder and the CSS from the wwwroot folder. The Shared folder contains `MainLayout.razor` and `NavMenu.razor` files you will edit to control the overall appearance and navigation for the app.

The Shared folder also contains the `SurveyPrompt` component used by the `Index` page.

MainLayout Page

The `MainLayout` page defines the overall layout of all pages in the app.

```
@inherits LayoutComponentBase

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

<div class="main">
    <div class="top-row px-4">
        <a href="http://blazor.net" target="_blank" class="ml-md-auto">About</a>
    </div>

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

You can see how this page specifies where the `NavMenu` and `@body` content will be displayed, as well as any global header content such as the About link.

Although I will discuss this in more detail later in the chapter, you should be aware that nearly any page can be displayed as a component of another page, just like this page is displaying the content of the `NavMenu` page. The ability to compose pages by using other page/component content is one of the most important features of the Blazor UI framework.

NavMenu Page

The `NavMenu` page not only displays the navigation menu, it also implements events and code to handle expanding the menu when the user clicks the hamburger menu button when it is visible.

The page relies on the `navbar-toggler` CSS style to control whether the hamburger menu button is visible.

```
<div class="top-row pl-4 navbar navbar-dark">
    <a class="navbar-brand" href="">BlazorClientExample</a>
    <button class="navbar-toggler" @onclick="ToggleNavMenu">
        <span class="navbar-toggler-icon"></span>
    </button>
</div>
```

If the button is visible, then when the user clicks on the button the `ToggleNavMenu` method is invoked from the `@code` block.

```
@code {
    private bool collapseNavMenu = true;

    private string NavMenuCssClass => collapseNavMenu ? "collapse" : null;

    private void ToggleNavMenu()
    {
        collapseNavMenu = !collapseNavMenu;
    }
}
```

This code toggles the `collapseNavMenu` value between `true` and `false`. More importantly, you can see how this value is used to change the `NavMenuCssClass` value between `collapse` and `null`. The `collapse` CSS style is used to alter the appearance of the navigation menu content in the page.

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
```

If the text of the `NavMenuCssClass` field is `collapse` then CSS controls whether the navigation menu content is visible to the user. If the value is `null` then the content is visible.

The `collapse` CSS is in the wwwroot folder in the `site.css` file. It is within a `@media` element so it is only expressed for certain screen sizes.

```
@media (min-width: 768px) {
```

When the width is 786px or greater then the sidebar will not collapse, otherwise it collapses.

```
    .sidebar .collapse {
        /* Never collapse the sidebar for wide screens */
        display: block;
    }
```

The interaction between the `NavMenu` HTML and Razor, the CSS styles, and the C# code works together to provide the user with a responsive experience for the navigation menu.

SurveyPrompt Component

I will discuss Blazor components in more detail in Chapter 4, but the `SurveyPrompt` component will give you a glimpse at what is possible. The component is basically a Blazor page, containing HTML, Razor markup, and code.

```
<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold"
        href="https://go.microsoft.com/fwlink/?linkid=2109206">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string Title { get; set; }
}
```

A component is a self-contained bit of UI functionality that can be placed into any other page or component. You might remember that the `Index` page makes use of this component.

```
<SurveyPrompt Title="How is Blazor working for you?" />
```

You can see how the component name (based on the file name) becomes an element name when used in another page. And you can see how the `Title` parameter in the component is decorated with a `Parameter` attribute so Blazor understands that this is a parameter value that is provided by the page when the component is used.

Again, I'll discuss these concepts in more depth in Chapter 4.

There are also a number of files at the project level. These files contain markup or code that affects the overall Blazor app.

_Imports Page

The `_Imports` page defines `@using` directives that apply to all pages in the app.

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using BlazorClientExample.Client
@using BlazorClientExample.Client.Shared
```

If you consistently use a namespace in your pages and want it to be globally available then you can add it to this file.

App Page

The `App` page manages routing for the app, and controls what the user sees if they navigate to an invalid URL.

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

You may edit this page to alter routing, alter the text displayed to the user if a route doesn't exist, and to implement some authentication and authorization features. I'll discuss authentication and authorization in Chapter 5.

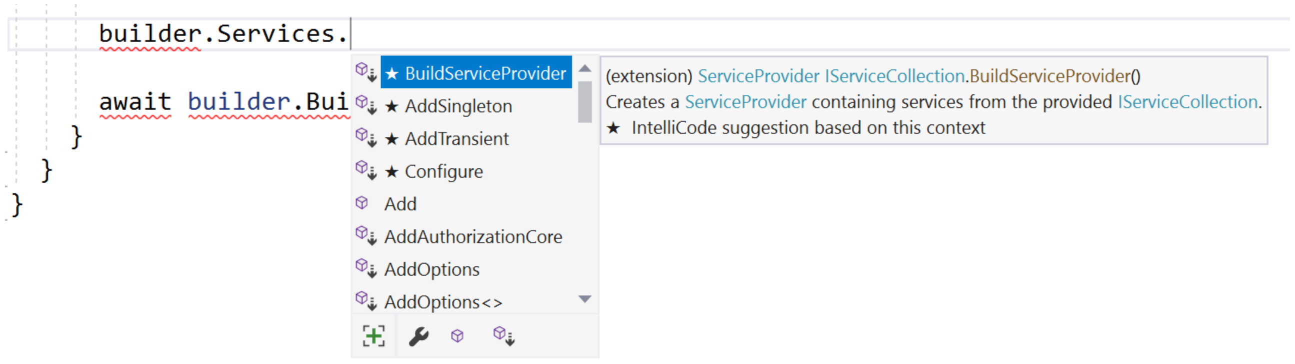
Program.cs

A client-side Blazor app, like most modern .NET Core apps, has `Main` method that acts as a single entry point for the app. This is the first code executed when the app is started.

This is also where you implement any configuration code for the app, such as adding services for use with dependency injection, or configuring services.

```
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");

      builder.Services.AddTransient(sp =>
        new HttpClient { BaseAddress =
                         new Uri(builder.HostEnvironment.BaseAddress) });

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

The `builder` field has a `Services` property that allows you to configure the .NET Core dependency injection framework, providing the DI framework with type mappings for any types that will be injected into your pages or other code.



```
        builder.Services.

        await builder.Bui
      }
    }
}
```

(extension) ServiceProvider IServiceCollection.BuildServiceProvider()
Creates a ServiceProvider containing services from the provided IServiceCollection.
★ IntelliCode suggestion based on this context

You can register types for DI using the `Services` property in the same way types are registered in the `Startup` class of an ASP.NET Core app. You can see how the Blazor template calls the `AddTransient` method to register the `HttpClient` type used in the `FetchData` page.

You can implement other configuration code in this method as needed by your app code. Later in this book you will see how to configure CSLA .NET by adding code to the `Main` method.

Blazor Progressive Web Application

A PWA (progressive web application) is a web site or web app that appears to the user much like a mobile or desktop app on their device. Some key features of PWA technology are that the app:

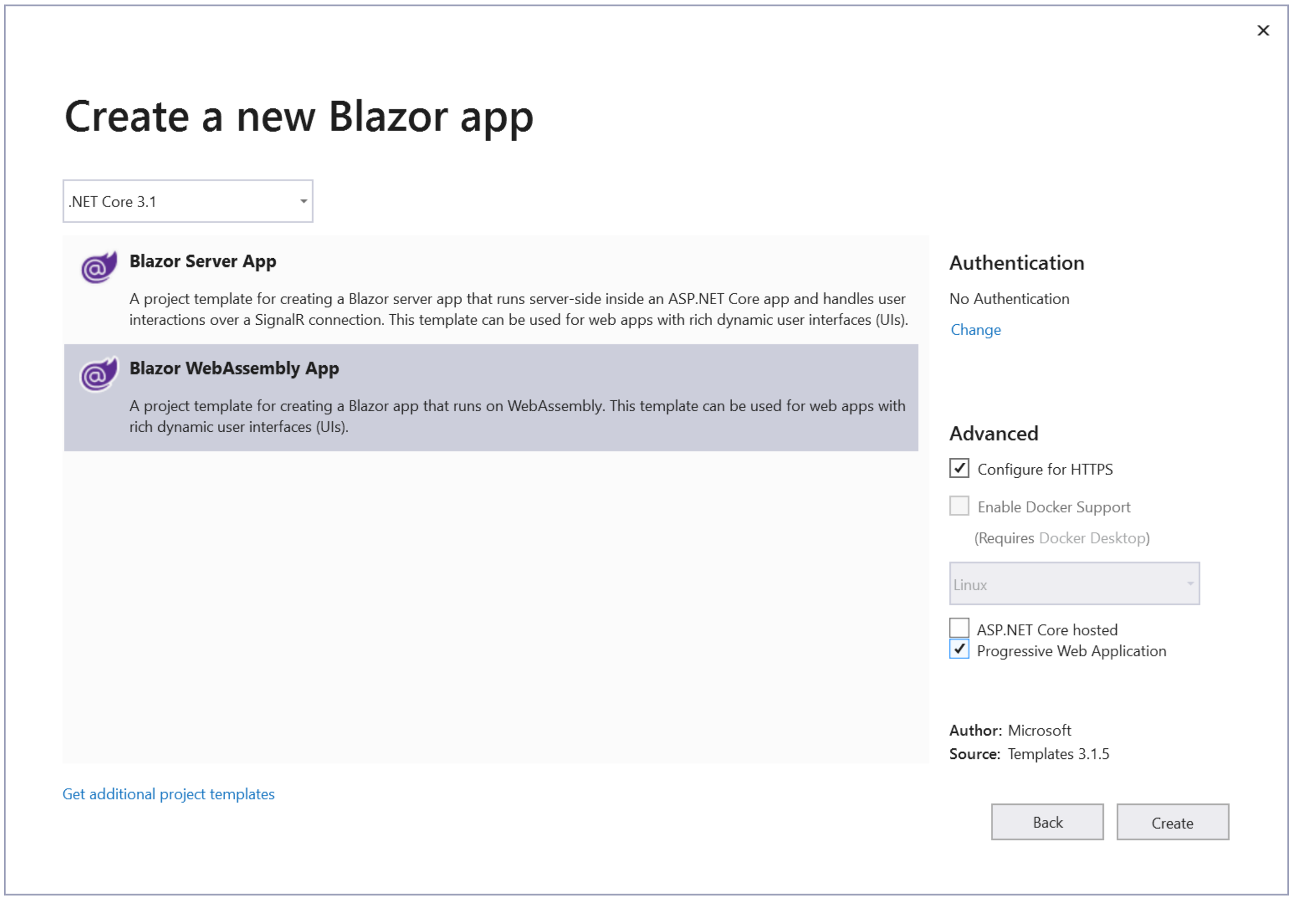
1. Has an icon or appears in the device launcher
2. Looks and feels like an app
3. Can be uninstalled like any other app
4. Can run online and offline
5. Has access to persistent storage on the device
6. May push notifications to the device
7. Leverages browser features on the device

Although many web apps can be converted into a PWA, it is more common for a PWA to be designed to work offline, store data locally on the device, and to sync data with services when online. In other words, when you choose to build a PWA with Blazor WebAssembly it is more than just checking the box in the Visual Studio wizard, there are also numerous application design choices you need to consider. I won't get into those details in this chapter, and will focus on the basic creation of a PWA with Blazor.

The user experience with PWAs varies depending on the user's operating system and browser. In this chapter I'll be using Windows 10 and the Chrome browser, but the deployment experience is similar on other browsers and operating systems.

> The code for this example is the `BlazorPwaExample` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Open Visual Studio 2019 and create a new Blazor project named `BlazorPwaExample`. Choose to create a Blazor WebAssembly App and make sure to check the Progressive Web Application option.


Create a new Blazor app

.NET Core 3.1

Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced

- [✓] Configure for HTTPS
- [ ] Enable Docker Support

(Requires Docker Desktop)

Linux

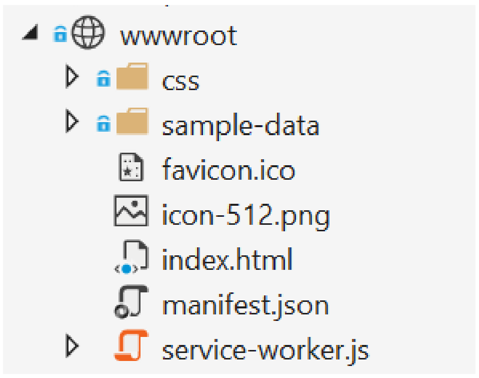
- [ ] ASP.NET Core hosted
- [✓] Progressive Web Application

Author: Microsoft
Source: Templates 3.1.5

Most of the project elements are identical to any standalone client-side Blazor app, including the initialization code in `Program.cs` and the Pages and Shared folders. I will only discuss the differences introduced to support PWA deployment in the `csproj` file and the wwwroot folder.

wwwroot Folder

The wwwroot folder contains a number of new files to support PWA deployment



These include `manifest.json`, `icon-512.png`, and `service-worker.js`. The `index.html` file is also different. I will discuss each of these files.

Deployment Manifest

The `manifest.json` file contains the information used by the client browser and operating system to install the PWA locally. The default file looks like this:

```json
{
  "name": "BlazorPwaExample",
  "short_name": "BlazorPwaExample",
  "start_url": "./",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#03173d",
  "icons": [
    {
      "src": "icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ]
}
```

The data in this file is used when installing the app.

- `name` - the name of the app displayed to the user
- `short_name` - a short name for the app
- `start_url` - URL used to launch the app from its icon
- `display` - preferred display mode (standalone, browser, fullscreen, minimal_ui)
- `background_color` - background color used until the css style is loaded
- `theme_color` - theme color that might be used by the host operating system for the app
- `icons` - icon graphics used when installing the app

You will usually customize these values as appropriate for your app requirements and appearance.

index.html File

The `index.html` file contains some extra elements to support PWA deployment.

```html
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0,
                   maximum-scale=1.0, user-scalable=no" />
    <title>BlazorPwaExample</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="manifest.json" rel="manifest" />
    <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" />
</head>

<body>
    <app>Loading...</app>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
    <script>navigator.serviceWorker.register('service-worker.js');</script>
</body>

</html>
```

The file adds `link` elements for the `manifest.json` and `icon-512.png` files.

```
    <link href="manifest.json" rel="manifest" />
    <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" />
```

It also registers the `service-worker.js` script.

```
    <script>navigator.serviceWorker.register('service-worker.js');</script>
```

The service worker is registered with the browser and is what enables much of the PWA functionality.

App Icon

Because a PWA is installed on the client device, the project now includes the icon graphic used on the client device in its start menu, launcher, or other experience provided by the device so the user can launch an app. You will want to replace the default `icon-512.png` image with one for your app.

You may also provide other icon graphics with different resolutions. Each icon file must be listed in the `icons` element of the `manifest.json` file.

Service Worker

Perhaps the most important feature provided by browsers to support PWAs is the concept of a *service worker*. You can think of a service worker as a proxy between your Blazor app and the network. The service worker will either service requests from a local cache when offline, or by communicating with the server when online.

The service worker can register events so you can write code to handle them and implement custom behaviors. These include events raised:

- `install` - when the service worker is first installed in a browser
- `activate` - each time the server worker is activated
- `fetch` - when you can fulfill a request for data from cache or a server call
- `push` - to implement push notifications from the server
- `sync` - to implement background sync operations to send data to the server

The staring template file demonstrates how to implement an empty `fetch` event handler.

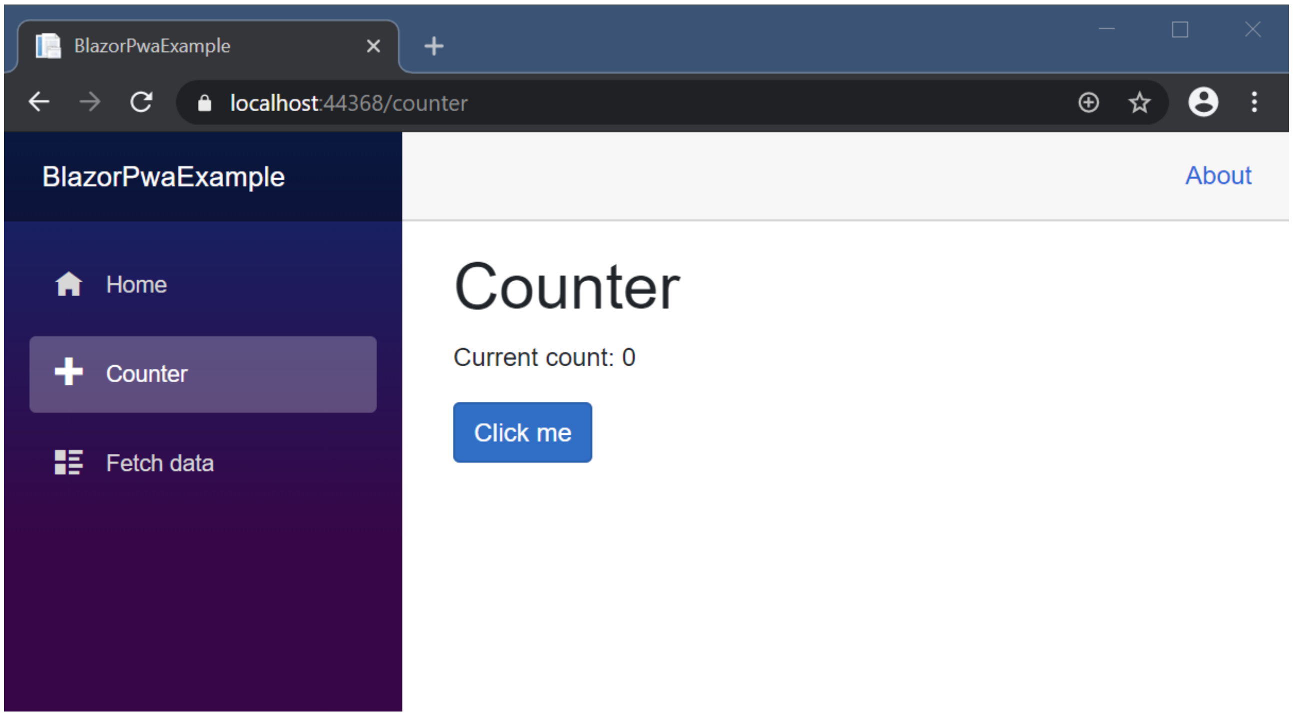
```
self.addEventListener('fetch', () => { });
```

I am not going to cover implementation of offline data caching, push notification implementation, or background sync operations in this chapter. The topic of designing and building production quality PWAs is large, and is outside this book's focus on Blazor.

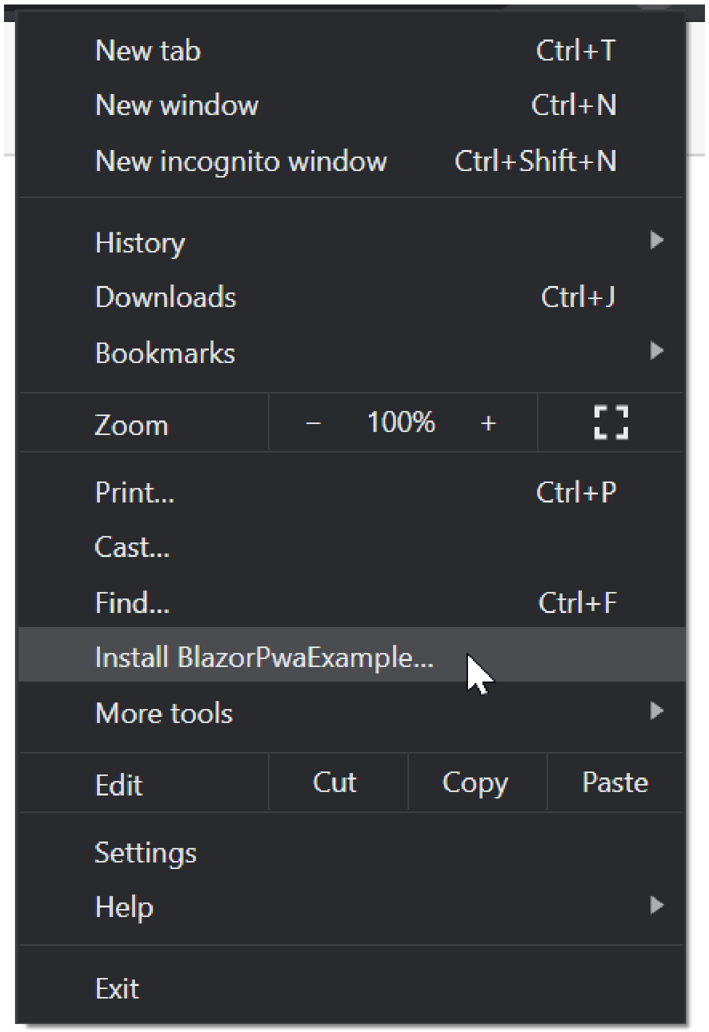
Installing a PWA

Because this project includes a `manifest.json` file and registers a service worker, it will be recognized as a PWA by all modern browsers on desktop and mobile devices. The user experience for installing a PWA is different for each operating system and browser, but the concepts are similar in all cases.

When you launch the project from Visual Studio you'll see the standard browser experience.

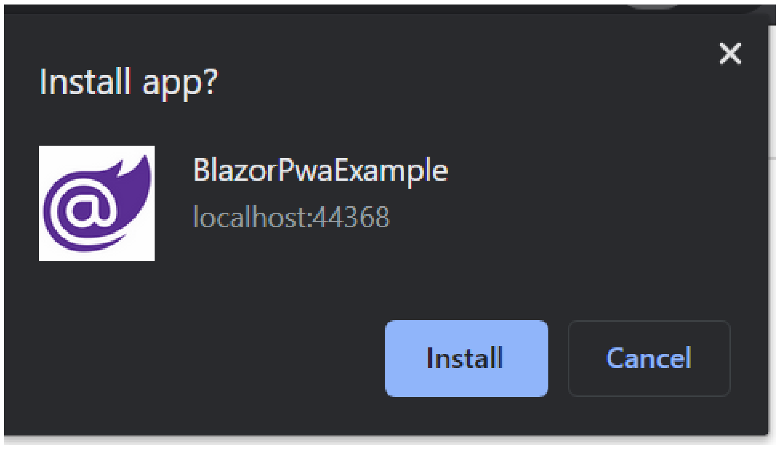


Clicking the ellipses to open the browser's page menu reveals an option to install the page as an app.



The name shown here comes from `manifest.json`, and the option is available as shown because the browser identifies this web site as a PWA.

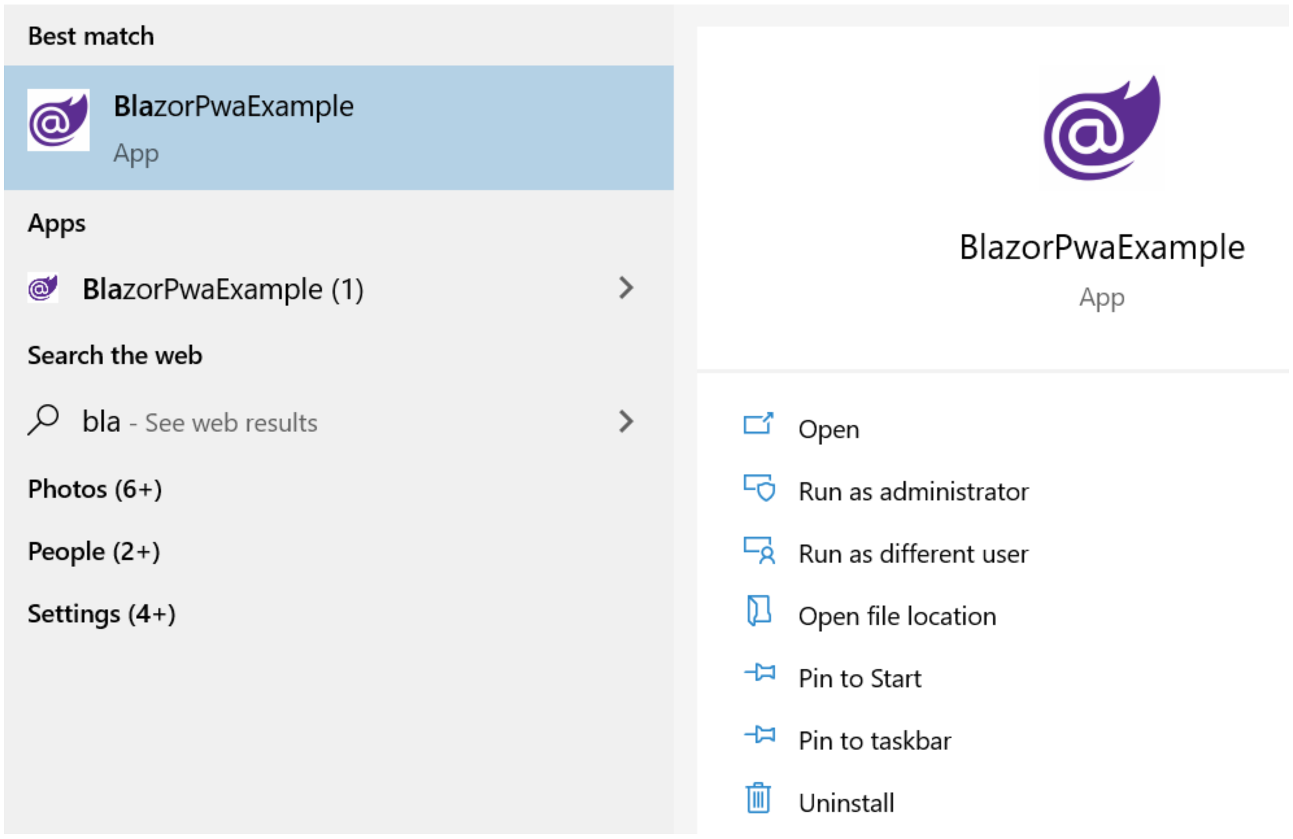
Clicking the install option in the menu results in a confirmation dialog.



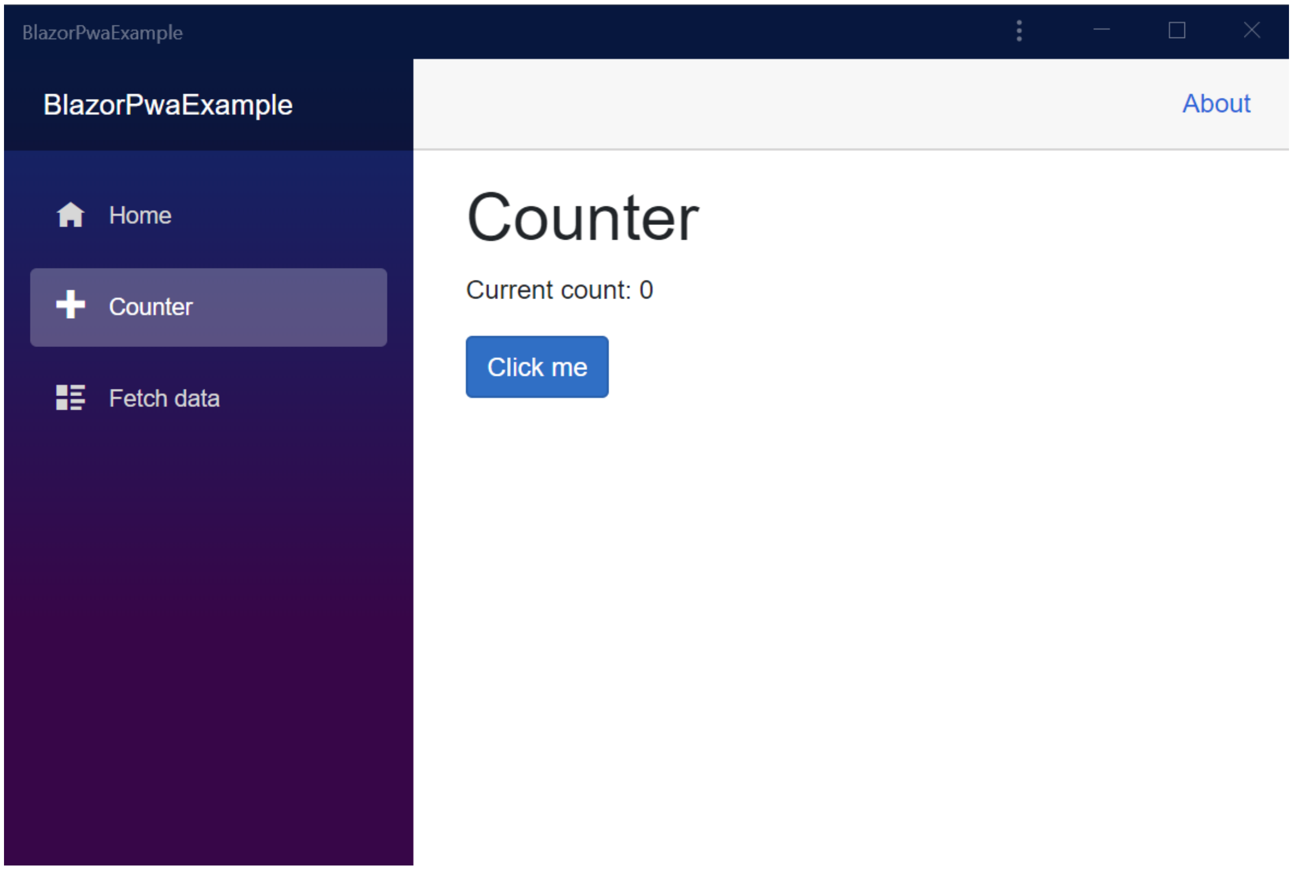
When the user confirms the install several things happen.

1. The browser installs the PWA as an app on the device
2. The operating system adds the app to any desktop, launcher, or start menu
3. The operating system launches the app in a new window

When doing this in the Chrome browser on Windows 10 the app is added to the Windows 10 start menu, and a shortcut is added to the user's desktop.



The original browser window remains open, and a new window is opened without the browser "chrome".



From this point forward the user can launch the app using the desktop shortcut or the Windows 10 start menu, just like any other app. Any app functionality that doesn't require access to a server will work while offline, and if the app has a robust service worker implementation some parts of the app that *do require* server access will continue to function.

The ability to deploy a Blazor WebAssembly app using PWA technology can be very powerful for some business scenarios.

Conclusion

At this point you should understand how to create a client-side Blazor project in Visual Studio, including a standalone app, or one hosted in ASP.NET Core. You should also understand all the folders and files that are provided in the default project templates. Let's move on to discuss Blazor components and how to create a Blazor class library that contains reusable UI elements.

Chapter 4: Blazor Features

In Chapter 2 and Chapter 3 you were exposed to some concepts that I glossed over, specifically:

1. Blazor components
2. Page and component lifecycle
3. Routing and navigation
4. Data binding

In this chapter I will discuss these Blazor features in more depth. In many ways these features are the centerpiece of what makes the Blazor UI framework so compelling and powerful.

Blazor Components

The Blazor UI framework is designed to support composition of a page from HTML, Razor markup, and Blazor components.

A Blazor component is a reusable unit of UI that can be composed into pages or other components.

It turns out that a page is also a component, so it is possible to treat a page as either a page that the user can navigate to directly, or as a component that can be embedded into another page.

Basic Creation and Use of Components

A component is encapsulated in a file, and the name of the file is the name of the component. Recall the `SurveyPrompt` component from the `BlazorClientExample.Client` project in Chapter 3.

```
<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold" href="https://go.microsoft.com/fwlink/?
linkid=2109206">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string Title { get; set; }
}
```

This defines some UI and behavior, and is contained in a file named `SurveyPrompt.razor`. The file name defines the component name, and the contents of the file represent the UI that can be composed into pages or other components. For example, it is used in the `Index` page.

```
<SurveyPrompt Title="How is Blazor working for you?" />
```

You can use the `SurveyPrompt` component in any page or component where you want to leverage its preexisting UI and behaviors.

Component Parameters

Components can accept parameters. The `SurveyPrompt` component accepts a `Title` parameter.

When you create a component that accepts a parameter you will use the `Parameter` attribute to declare any fields that should be treated as parameters. These fields must be `public` in scope.

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

This attribute indicates that when the component is used in another component or page that a value can be provided using the field name. So when the `SurveyPrompt` component is used in the `Index` page, the `Index` page provides a value for the `Title` parameter.

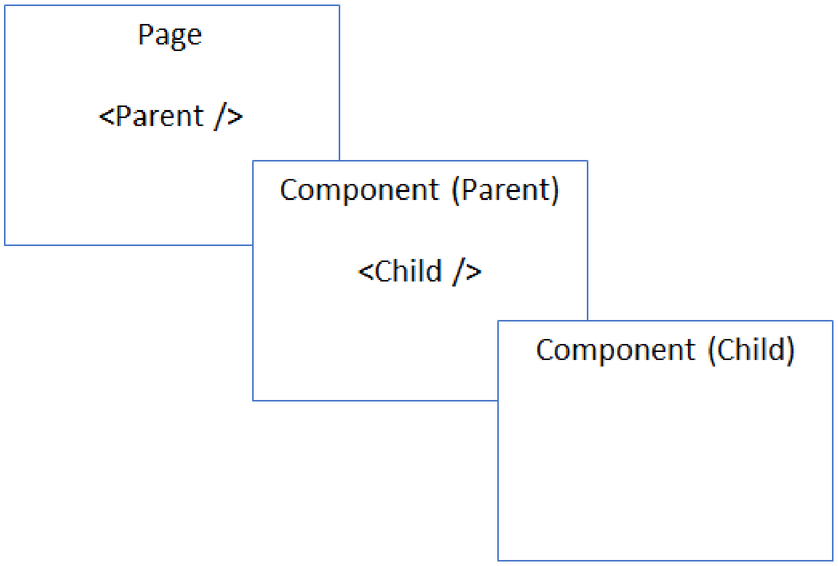
```
<SurveyPrompt Title="How is Blazor working for you?" />
```

A component can accept multiple parameters by declaring multiple fields, each with the `Parameter` attribute.

When a component is used in a page there is no requirement that a value be provided to any of the parameters. They are all optional. When you create a component you should provide default values or otherwise handle the case that no value was provided for parameters.

Cascading Parameters

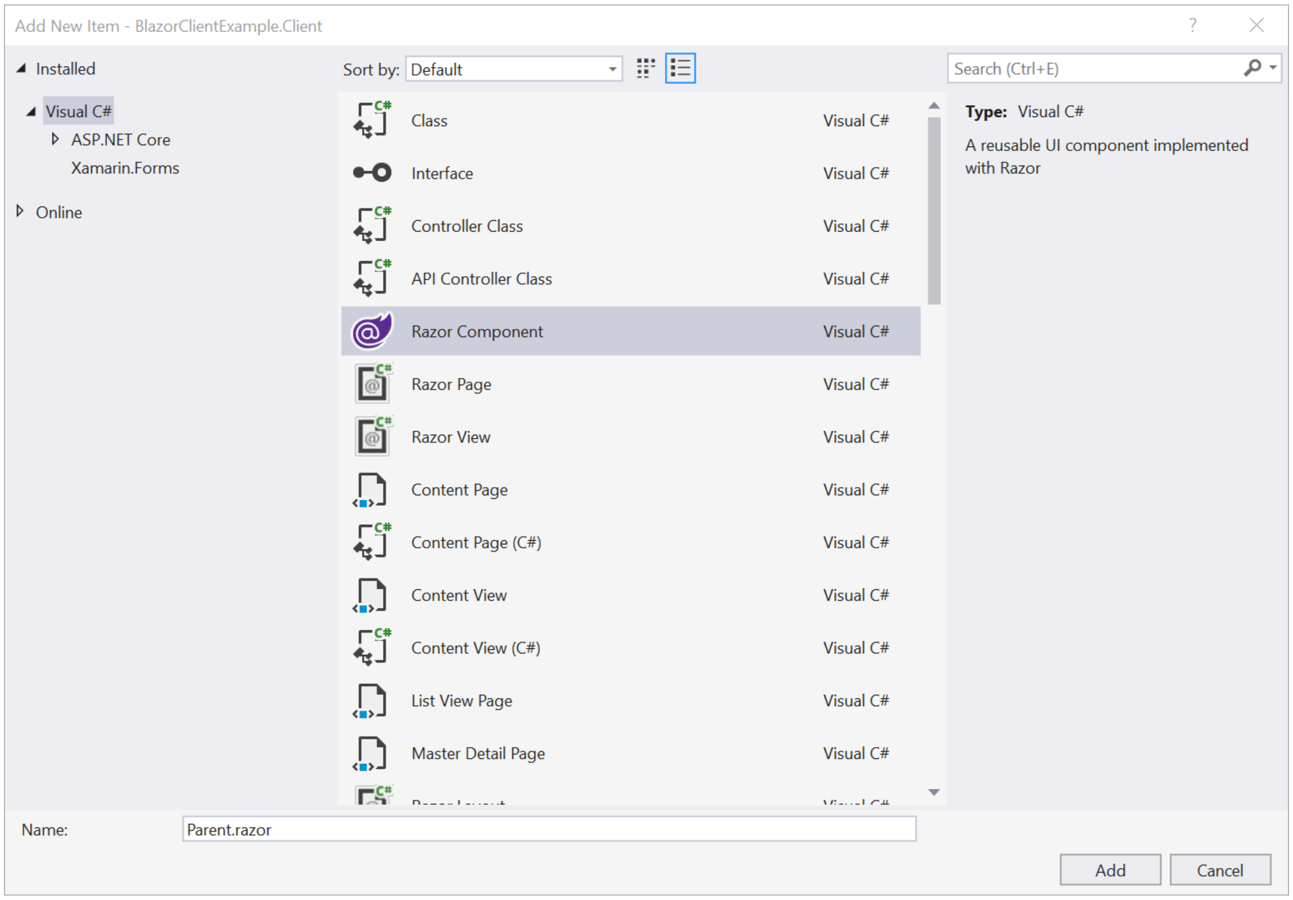
Because you can nest components inside other components there will be times when you need to pass a parameter from the page that's hosting the first component into the nested component.



Passing parameters *through* one component into a nested component is supported by using the concept of cascading parameters. This is supported by a Razor element called `CascadingValue` and a .NET attribute called `CascadingParameter`.

> i You should know that there are many variations on how you can declare and use cascading parameters. I am showing you just one approach in this chapter. In my opinion, this approach provides the best readability and maintainability, but there are cases where other options may be useful.

Add a `Parent` component to the `BlazorClientExample.Client` project from Chapter 3. Add it to the Shared folder by right-clicking on the Shared folder in Solution Explorer and choosing the Add | New Item option. In the New Item dialog choose the Razor Component item.



Edit the resulting component to accept and display a cascading parameter value.

```
<h3>Parent</h3>

<p>@Title (from parent)</p>

@code {
  [CascadingParameter(Name = "Title")] string Title { get; set; }
}
```

Notice the use of the `CascadingParameter` attribute. It has a `Name` parameter that defines the name of the cascading parameter value that should be mapped into this component's `Title` field. Also notice that this field does not have to be `public` like a regular parameter, and in fact you should prefer to make these fields `private` so their value only flows through the cascading parameter subsystem.

Now edit the `Index` page to define the cascading parameter value and to display the `Parent` component.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<CascadingValue Value="@Title" Name="Title">
  <Parent />
</CascadingValue>

@code {
    string Title = "42";
}
```

You can see how the `CascadingValue` element is used to define the context for the cascading value.

The `Value` parameter defines the expression that is used to retrieve the actual value. You can see how the `Title` field is set to a real value in the `@code` block.

The `Name` parameter defines the name by which this cascading parameter value is referenced in child components. This value matches the value used in the `CascadingParameter` attribute in the `Parent` component.

This probably seems like a lot of work to pass a parameter to the `Parent` component. The advantage of this approach is that you can nest another component inside `Parent`.

Add another component to the Shared folder. Name this one `Child`, and update its code to also use the cascading parameter value.

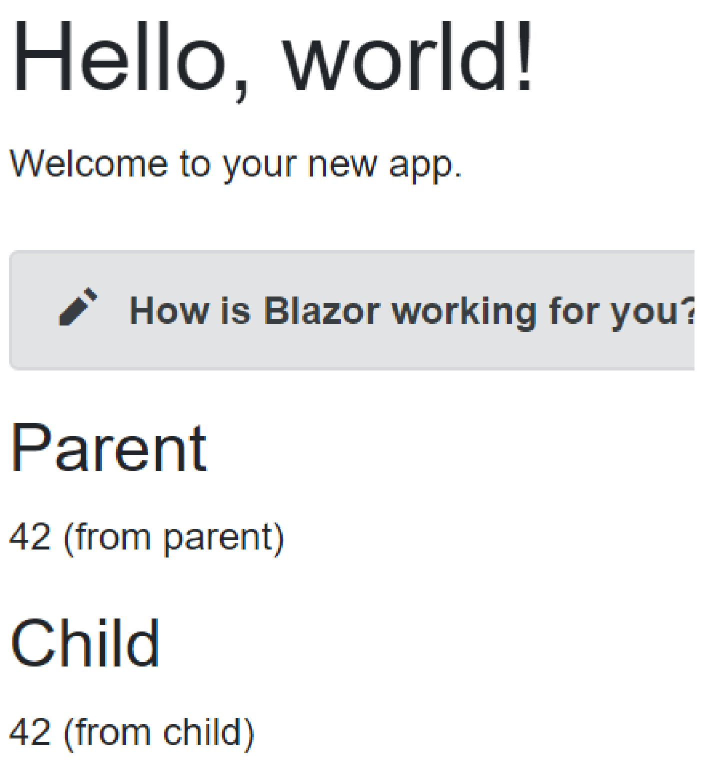
```
<h3>Child</h3>

<p>@Title (from child)</p>

@code {
  [CascadingParameter(Name = "Title")] string Title { get; set; }
}
```

You can see it uses the same `Name` value in the `CascadingParameter` attribute. This will work because all components loaded within the context of the top-level `CascadingView` element from the `Index` page will have access to this cascading parameter value.

Now update the `Parent` component to include the new `Child` component.

```
<h3>Parent</h3>

<p>@Title (from parent)</p>
<Child />

@code {
  [CascadingParameter(Name = "Title")] string Title { get; set; }
}
```

When you run the app you'll see how the cascading parameter value is displayed from the `Parent` and `Child` components.


Hello, world!

Welcome to your new app.

How is Blazor working for you?

Parent

42 (from parent)

Child

42 (from child)

The cascading parameter concept in Blazor enables powerful composition scenarios for building your UI from many components, allowing you to nest components without having a parent component needing to be aware of the parameters required by child components.

Pages as Components

Any Blazor page can be treated as a component. The only real difference between a "page" and a "component" is that pages have one or more `@page` directives so they support routing.

When used as a component, any `@page` directives in a page are ignored.

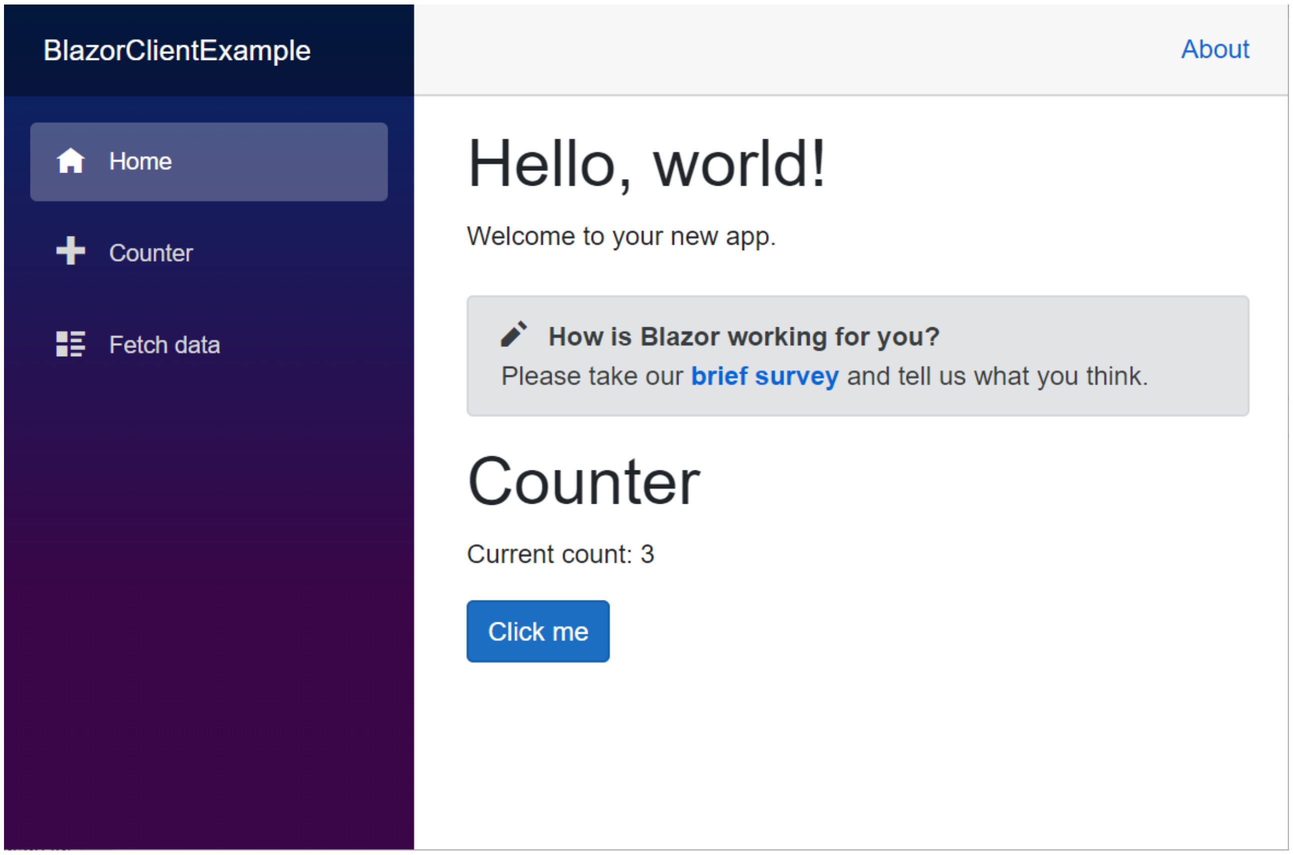
For example, in the `BlazorClientExample.Client` project there is a `Counter` page.

```
@page "/counter"

<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++;
    }
}
```

Because it has a `@page` directive the user can directly navigate to this page. You can also use this as a component in other pages. For example, you can compose the `Counter` component into the `Index` page.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<Counter />
```

The result is that the UI and behavior of the `Counter` page is included in the `Index` page.



Hello, world!

Welcome to your new app.

How is Blazor working for you?
Please take our brief survey and tell us what you think.

Counter

Current count: 3

The original `Counter` page is entirely unaffected. The `Counter` component in the `Index` page is a totally separate instance of the page/component.

> It is literally a separate instance. Each time a component or page is loaded, Blazor creates an instance of that component with its own private fields and state. Exactly what happens when you create instances of your own classes.

Page and Component Lifecycle

Like all UI frameworks, Blazor follows a specific lifecycle for pages and components. You can override lifecycle methods to implement behavior as a component is initialized, parameter values set, and rendered.

> Because pages are also components, I will refer to everything as a component in this part of the chapter.

Initialization

When a component instance is created you have the opportunity to initialize the instance.

- `OnInitialized`
- `OnInitializedAsync`

If your initialization is synchronous override the `OnInitialized` method, and if your initialization is asynchronous you must override the `OnInitializedAsync` method.

It is important to understand that initialization occurs exactly one time per component. You may be tempted to use these methods to load data values into your component, but I recommend you avoid that and instead load any data values after the parameters have been set.

Setting Parameters

After a component has been initialized, any parameters provided by the parent (or routing) will be set. You can override a method to alter how parameter values are set within your component.

- `SetParametersAsync`

By default parameter values are set based on `Parameter` and `CascasdingParameter` attributes in your component.

Once the parameters have been set you can override methods to initialize your component based on the now-available parameter values.

- `OnParametersSet`
- `OnParametersSetAsync`

Choose the appropriate method based on whether you will implement sync or async code.

I recommend using one of these methods to initialize your component, especially if you are loading the component with data from a service or other external source.

These methods are invoked before the component renders.

Component Rendering

Before your component is rendered Blazor invokes a method.

- `ShouldRender`

You can override `ShouldRender` and return a `bool` value indicating whether rendering should occur. If you return `false` then rendering will not occur.

After Blazor has rendered your component you can override methods.

- `OnAfterRender`
- `OnAfterRenderAsync`

Choose the appropriate method based on whether you will implement sync or async code.

> ⚠ These methods are not invoked during server-side prerendering.

These methods accept a `bool` parameter indicating whether this is the first time the component has been rendered. You can use this parameter to do first-time initialization of your component.

Forcing Component Rendering

You can force your component to be rendered by calling the `StateHasChanged` method in your code. This method tells Blazor that the component state has changed, triggering a render of the component.

Blazor is quite good about monitoring your component's fields for changes, and it will normally render the component when changes occur. This means you won't normally need to call the `StateHasChanged` method, because rendering will occur automatically.

It is possible that you will encounter scenarios, especially when calling async operations, where your component will not automatically refresh. If that happens, remember that you can call the `StateHasChanged` method to force a refresh.

Routing and Navigation

I will discuss three aspects to routing and navigation.

1. Getting parameter values from the URL
2. Navigating to pages by HTML elements
3. Navigating to pages in code

Let's talk about each concept.

Get Parameter Values from the URL

I discussed how the `@page` directive is what makes a component a page, and it is this directive that allows a user to navigate directly to a page. Blazor creates a route for your page based on the directives in your component.

For example, the `Index` page has a single directive.

```
@page "/"
```

This route indicates that the `Index` page is the default, root level, page.

The `Counter` page also has a directive, even though you have seen how this page can also be used as a component.

```
@page "/counter"
```

It is possible for a page to have multiple routes. Most commonly these alternate routes will be used to provide parameters to the page through the URL. Edit the `Counter` page to have an alternate route with a parameter.

```
@page "/counter"
@page "/counter/{start:int}"
```

This indicates that the user can navigate to `/counter` or something like `/counter/42`.

Now add a `start` field with a `Parameter` attribute, and an `OnParametersSet` method override in the `@code` block.

```
  [Parameter]
  public int start { get; set; }

  protected override void OnParametersSet()
  {
    currentCount = start;
  }
```

The `start` field is a parameter, and its name corresponds to the name used in the `@page` directive. This is how Blazor knows to map the parameter value from the route into the field.

Notice that the field is an `int` value, matching the constraint applied to the value in the `@page` directive. If you don't apply a constraint the default `string` type is used.

The result is that when the user navigates to the `Counter` page using the new route, the counter value will start at the provided value instead of zero.

Navigating to a Page with HTML

Passing parameters to a page through the URL is a very common pattern for navigation between pages in Blazor. This is often as simple as using HTML in a page to create a navigation link based on your pages.

Edit the `Index` page and add a link to the `Counter` page that uses the new route you just created.

```
<a href="/counter/42">Counter</a>
```

This is a simple `href` link using the relative URL and passing a parameter value to the target page.

Navigating to a Page with Code

Although it is very common to allow users to navigate to another page by clicking an `href` link, there are also times when you need to navigate to another page in code.

Blazor provides a `NavigationManager` type that encapsulates the navigation service. This type is normally injected into a page, so inject it into the `Index` page by adding this directive after the `@page` directive at the top of the file.

```
@inject NavigationManager NavigationManager
```

You can then add a `button` element to the page that runs some code.

```
<button @onclick="GoToCounter">Counter</button>
```

Finally, you can implement the `GoToCounter` method in the `@code` block.

```
  private void GoToCounter()
  {
    NavigationManager.NavigateTo("counter/123");
  }
```

This code doesn't do anything differently from the `href` link used in the last section, but it does allow you to do the navigation from code instead of markup.

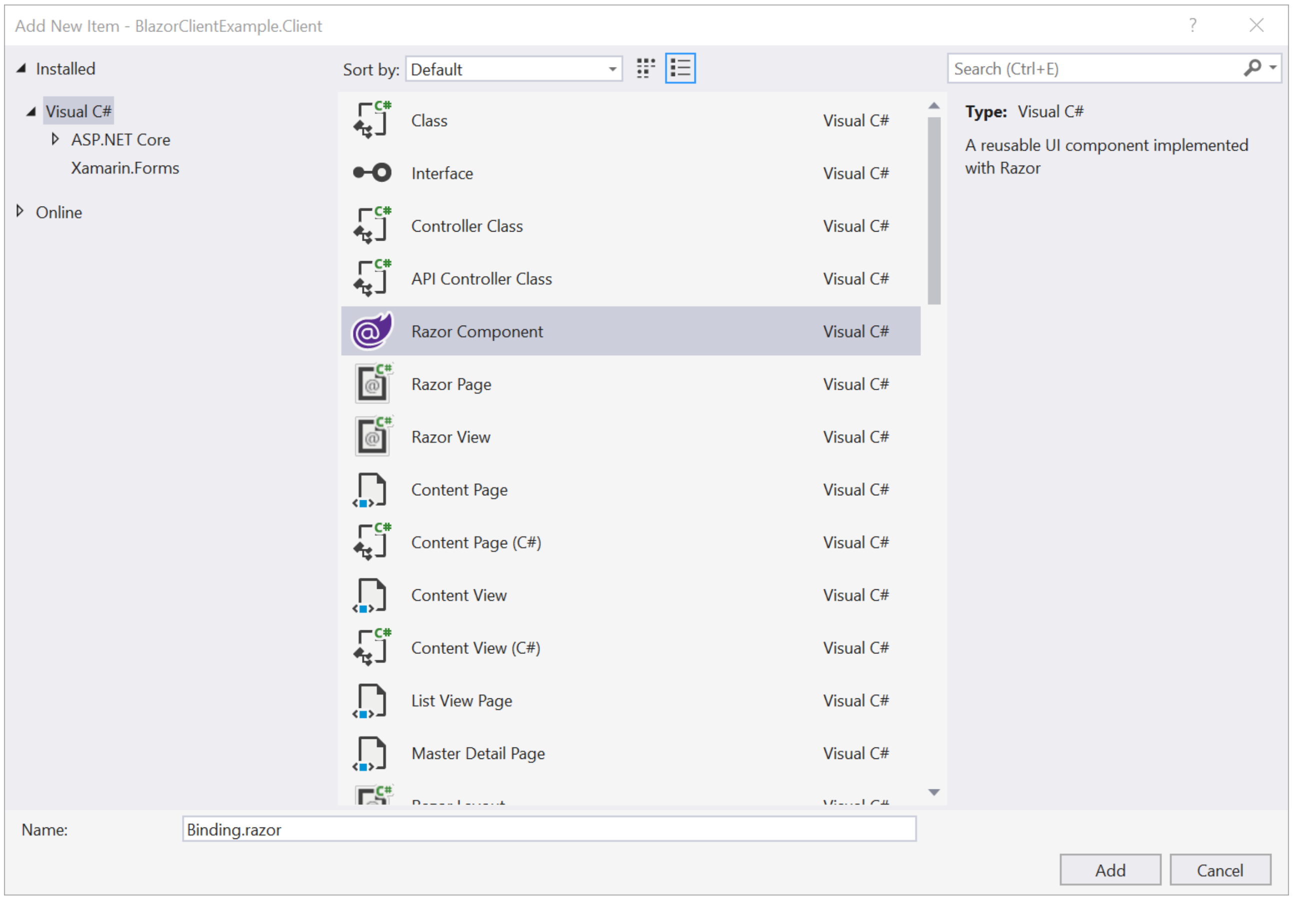
Data Binding

Creating data-focused user experiences requires getting data out of your objects or data model so it can be displayed in the UI. And it requires getting data out of the UI after being edited by the user, so the values can be put back into your objects or data model.

Although this can be done by hand, writing code to move the data back and forth between the UI view and your model, data binding technologies have been automating this process for more than two decades. It is also the case that nearly every data binding framework it better than its predecessors, and Blazor is no exception. The data binding model provided by Blazor is powerful and yet easy to use.

Blazor supports binding to read-only values, read-write values, and provides an `EditForm` component to make it easy to create simple data entry forms.

For the code in this section of the chapter, create a new page in the Pages folder named `Binding`.



Type: Visual C#

A reusable UI component implemented with Razor

Name: Binding.razor

Add a `@page` directive for routing.

```
@page "/binding"

<h3>Binding</h3>

@code {

}
```

Also add a `PersonEdit` class at the top level of the project, along side `Startup.cs`.

```
using System;

namespace BlazorClientExample.Client
{
  public class PersonEdit
  {
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
    public int Age { get => (DateTime.Now - Birthday).Days / 365; }
  }
}
```

I will use these types as the start point for the rest of this section of the chapter.

To make it easy to access the `Binding` page, edit the `NavMenu` page in the Shared folder and add an element to the existing `ul` list.

```
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="binding">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Binding
      </NavLink>
    </li>
```

This will add an item to the app's navigation menu so you can easily open the `Binding` page.

Displaying Values

You have already seen examples of one-way data binding to display values to the user in the `Counter` and `FetchData` pages.

One-way binding is where values flow from your fields, data model, or domain objects to the UI for display.

Edit the `Binding` component to declare a field based on the `PersonEdit` class and to create an instance of the class. Also add markup to display the properties of the `person` field.

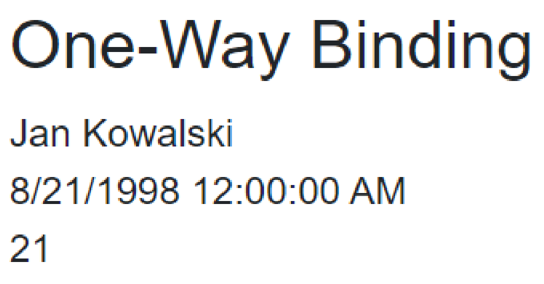
```
@page "/binding"

<h3>One-Way Binding</h3>
<p>
  @(person.Name)
  <br />
  @(person.Birthday)
  <br />
  @(person.Age)
</p>

@code {
  private PersonEdit person;

  protected override void OnParametersSet()
  {
    person = new PersonEdit
    {
      Name = "Jan Kowalski",
      Birthday = new DateTime(1998, 8, 21)
    };
  }
}
```

In this example I am using the `@()` syntax for the binding expression instead of the simpler `@` syntax. For simple types like `PersonEdit` you can use either approach. These lines are equivalent.

```
  @person.Name
  @(person.Name)
```

Later in this book I will be using generic types, and then you *must* use the `@()` syntax. If you use the simpler syntax then the Razor parser will get confused by the `<` and `>` characters because they are part of HTML as well as being how C# designates generic type values.

When you run the app and navigate to the `Binding` page you should see the values displayed.


One-Way Binding

Jan Kowalski

8/21/1998 12:00:00 AM

21

The output of the `Birthdate` property is not ideal. Fortunately that's easily addressed with a small change to the binding expression in the page.

```
@(person.Birthday.ToShortDateString())
```

This is possible because the way one-way data binding works is that Blazor executes the C# expression after the `@` and returns its `ToString` value. If a value is displayed in a format that is undesirable, you can use standard C# formatting techniques to change how the value is displayed.

Input Binding

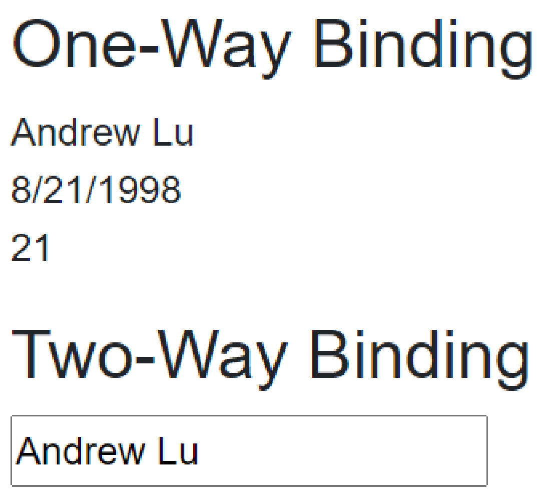
Displaying values using one-way binding leverages the same Razor markup that's been in use for many years. The two-way binding for input controls is new to Blazor, and is designed around providing interactive user experiences.

Basic Input Binding

Edit the `Binding` page and add an `input` element so the user can edit the `Name` property.

```
<h3>Two-Way Binding</h3>
<p>
  <input @bind="person.Name" />
</p>
```

When you run the app you will see the `Name` property value in both sections of the page. When you edit the name value and tab out of the input element you can see that the original one-way binding immediately updates.


One-Way Binding

Andrew Lu

8/21/1998

21

Two-Way Binding

Andrew Lu

This is because both UI elements are data bound to the same object property, and when that property is changed it automatically updates all data bound UI elements.

Controlling Input Binding

Depending on your user requirements you may need binding to happen when the user tabs out of the field, or on each key press.

Edit the `Binding` page and add another `input` element.

```
<h3>Two-Way Keypress Binding</h3>
<p>
  <input @bind-value="person.Name" @bind-value:event="oninput" />
</p>
```

When you run the app you can experiment with typing in the first input element and tabbing out to see the change. And you can type in this new input element and see how the `Name` property value (and all UI elements data bound to the property) update as you press each keystroke.

This applies not only to text `input` elements, but to other types of `input` element and other HTML elements that allow for user input, such as a dropdown list or checkbox.

EditForm Binding

Binding to individual properties is powerful. Blazor adds to this by providing an `EditForm` component that makes it easier to create data entry forms with a common display for any validation errors that might occur within the form.

> By default only data attribute validation is supported. In Chapter 7 I will discuss how you can use `EditForm` along with the richer set of business rules supported by CSLA .NET.

The `EditForm` component relies on a set of other input components, all of which inherit from `InputBase<T>`. These components include:

- `InputText` - allows user to enter a single text field
- `InputTextArea` - allows user to enter multiple lines of text
- `InputNumber` - allows user to enter a number
- `InputDate` - allows user to enter a date
- `InputCheckBox` - allows user to click a checkbox
- `InputSelect` - allows user to select a value from a dropdown list

You can use standard `input` tags in an `EditForm` too, but these input components abstract common behaviors required in most cases, and so they offer improved readability of code, as well as enhanced maintainability.

You can also create your own input components by inheriting from `InputBase`. That topic is outside the scope of this book.

Now let's create a data entry form using the `EditForm` component.

Creating a Basic Form

Add a `SimpleForm` Razor Component to the Pages folder in the project, and update its contents as shown here.

```
@page "/simpleform"

<h3>EditForm Example</h3>

@if (person == null)
{
<div>Loading...</div>
}
else
{
<EditForm Model="@person">
  <input type="submit" value="Save" class="btn btn-primary" />
</EditForm>
}

@code {
  private PersonEdit person;

  protected override void OnParametersSet()
  {
    person = new PersonEdit
    {
      Name = "Jan Kowalski",
      Birthday = new DateTime(1998, 8, 21)
    };
  }
}
```

The `@code` block is identical to the `binding` page you created earlier in this chapter. The markup is different however, because it sets up an `EditForm` component that is bound to the `person` field.

It is important to understand that the `person` field could be `null`, and that the `EditForm` component will throw an exception when attempting to bind to a `null` value. Because of this, you must always wrap any `EditForm` component in an `if` statement to handle the possible `null` value of the the `Model` property.

Within the form is a button so the user can submit the form values (after you add input components).

Also add markdown to the `NavMenu` component in the Shared folder so it is easy to navigate to this new page.

```
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="simpleform">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Simple form
      </NavLink>
    </li>
```

With that done, let's discuss how a form is submitted by the user.

Handling Form Submission

When the user clicks the button to submit the form, the form's contents are routed to the `@code` block. There are two ways to handle the form data.

First, you can handle the `EditForm` component's `OnSubmit` event. This is done by binding to the event.

```
<EditForm Model="@person" OnSubmit="@FormSubmitted">
```

To see the results of a simulated save operation, add a `div` tag above the `EditForm` component.

```
<div>@statusText</div>
```

And declare a `statusText` field in the `@code` block.

```
  private string statusText;
```

Then implement the event handler method in the `@code` block.

```
  private void FormSubmitted(EditContext editContext)
  {
    statusText = "Form submitted";
    // handle form data here
  }
```

In this `FormSubmitted` method you can use the `EditContext` parameter to trigger validation and see the results of validation. You can also access the form's `Model` property, making it possible to save the data from the form.

The second approach is to handle two events instead of a single event. These two events come from the `EditForm` component and allow you to implement code for valid input, and separate code if there's a validation error in the form.

1. `OnValidSubmit` - the form data is valid
2. `OnInvalidSubmit` - the form data is invalid due to a validation rule

To do this, start by binding the two events.

```
<EditForm Model="@person"
          OnValidSubmit="@FormValid"
          OnInvalidSubmit="@FormInvalid">
```

Then implement two event handler methods.

```
  private void FormValid(EditContext editContext)
  {
    statusText = "Form is valid";
    // handle valid form data here
  }

  private void FormInvalid(EditContext editContext)
  {
    statusText = "Form is not valid";
    // handle invalid form data here
  }
```

In this case validation occurs automatically, and the appropriate event is raised depending on whether the form data is valid or invalid. Notice that you still have access to the `EditContext` parameter, so you can interact with the validation and model data.

> ⚠ You must choose to use `OnSubmit` or to use the other two events, you can't mix the two models together.

In the rest of this chapter I will be using the `OnSubmit` event, with the `EditForm` component declared as shown.

```
<EditForm Model="@person" OnSubmit="@FormSubmitted">
```

The next step is adding the input components to the form.

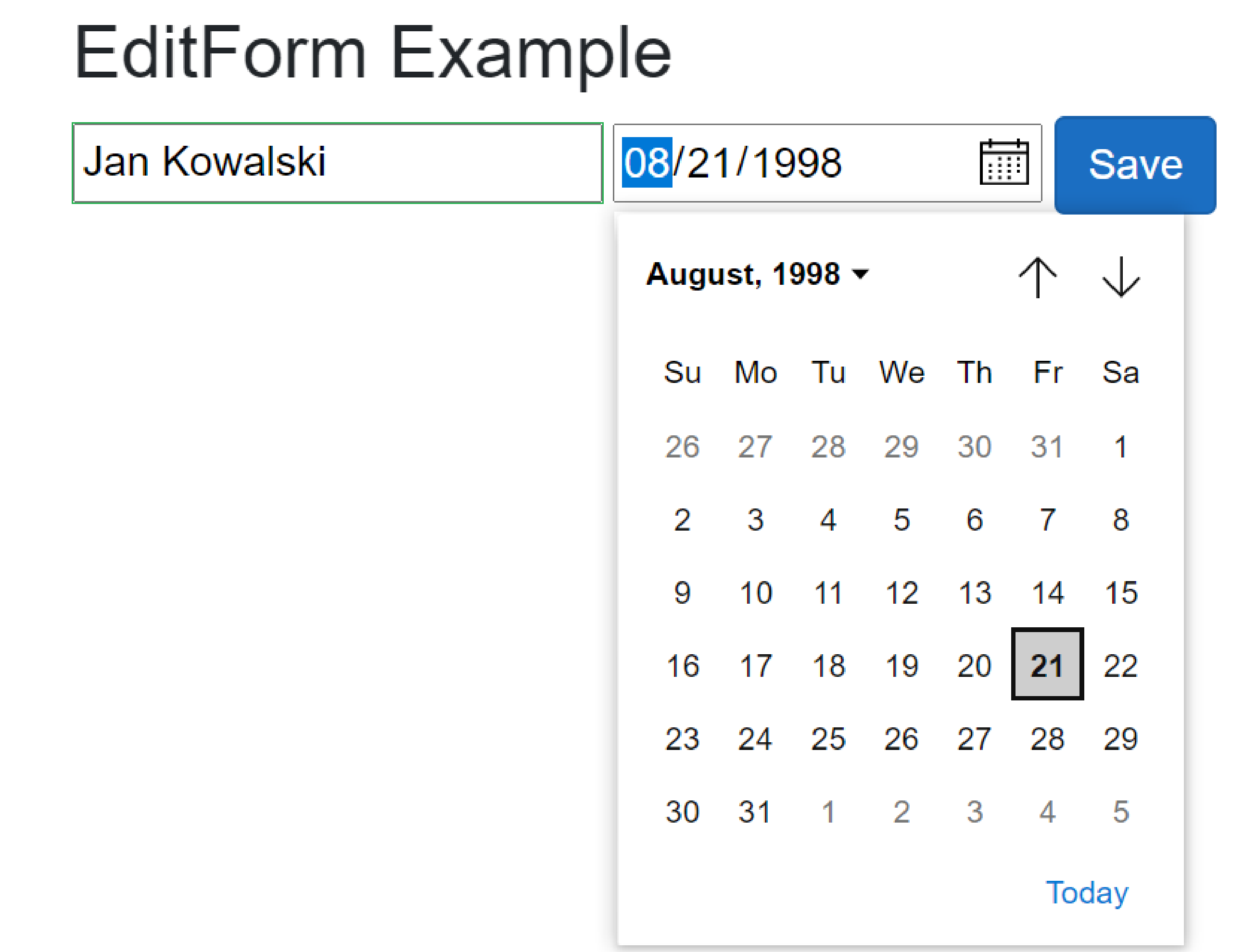
Binding Input Components to the Model

Each input component might have its own properties to allow you to control how the component works, but all of them use the `@bind-value` property to bind to the value of the form's model.

Edit the markup in the page as shown.

```
<EditForm Model="@person" OnSubmit="@FormSubmitted">
  <InputText @bind-Value=person.Name />
  <InputDate @bind-Value=person.Birthday
             ParsingErrorMessage="Birthday must be a date" />
  <div>@person.Name is @person.Age years old</div>
  <input type="submit" value="Save" class="btn btn-primary" />
</EditForm>
```

Running the app at this point will allow you to see and interact with the page.



Notice how the `InputText` component appears as a simple input element, and the `InputDate` component appears as a nice date input element in the browser. And the `Age` property is displayed using one-way binding in a `div` tag.

You can also see how the `Age` property updates in the object and display as the `Birthdate` property is changed. This demonstrates how Blazor is typically able to automatically detect any changes to the model and update the display accordingly.

> This automatic update the UI works because changing the `Age` property is as a result of a UI event (changing the `Birthdate` property). If a property on the model changes because of some background task (not a UI event) then the display may not be automatically updated.

At this point the form works to enter and submit data. In most apps a data entry form will at least implement some basic validation rules, so let's discuss how that works.

Running and Displaying Validation Rules

Blazor interacts with the data annotation attributes from the `System.ComponentModel.DataAnnotations` namespace. These attributes provide basic validation rules, including:

- `Required` - a `string` property is required
- `StringLength` - a `string` property has a max length
- `Range` - a property has min and max values
- `RegEx` - a property must match a regular expression

Edit the `PersonEdit` class and use the `Required` attribute to make the `Name` property required.

```
    [Required]
    public string Name { get; set; }
```

Then edit the `SimpleForm` page and add a `DataAnnotationsValidator` component inside the `EditForm` component.

```
<EditForm Model="@person" OnSubmit="@FormSubmitted">
  <DataAnnotationsValidator />
  <InputText @bind-Value=person.Name />
  <InputDate @bind-Value=person.Birthday
             ParsingErrorMessage="Birthday must be a date" />
  <div>@person.Name is @person.Age years old</div>
  <input type="submit" value="Save" class="btn btn-primary" />
</EditForm>
```

This new component will cause Blazor to run all the data annotations rules when the form is submitted.

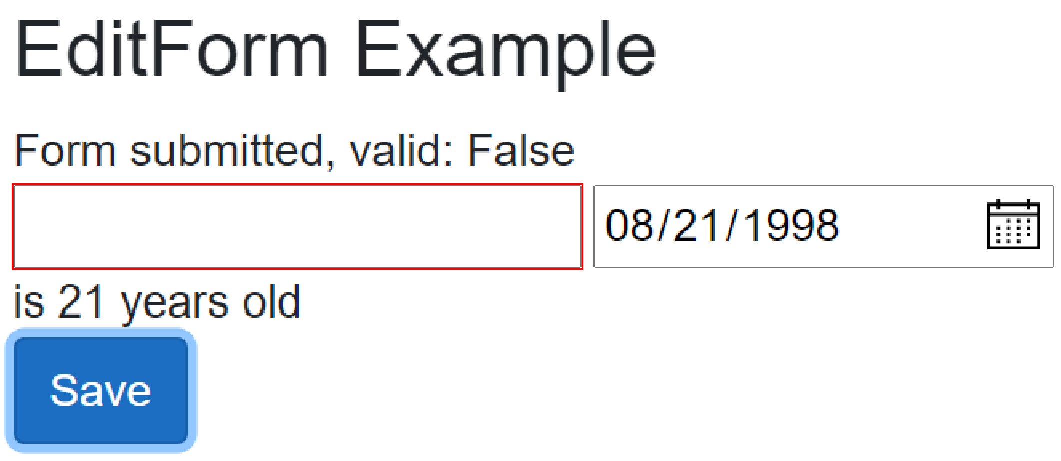
If you are using the `OnValidSubmit` and `OnInvalidSubmit` events on the `EditForm` component, the appropriate event will now be raised based on the validation rules.

If you are using the `OnSubmit` event, as in this chapter, you need to check the validation results in your event handler method.

```
  private void FormSubmitted(EditContext editContext)
  {
    statusText = $"Form submitted, valid: {!editContext.GetValidationMessages().Any()}";
    // handle form data here
  }
```

The `EditContext` parameter provides access to all the validation messages generated by the data annotations attributes. If there are no messages, then no rules have been broken.

Run the app, blank the `Name` field, and click the button to see the result.


EditForm Example

Form submitted, valid: False

08/21/1998

is 21 years old

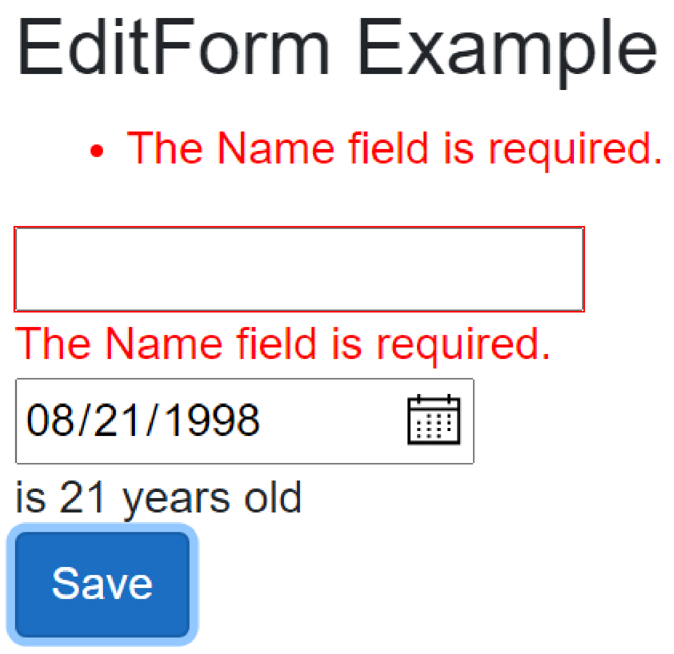
You could write code to display those messages to the user, but you don't need to do that work thanks to the `ValidationSummary` and `Validation` components.

Validation Display Components

Add `ValidationSummary` and `Validation` components to the `EditForm` component in the `SimpleForm` page.

```
<EditForm Model="@person" OnSubmit="@FormSubmitted">
  <DataAnnotationsValidator />
  <ValidationSummary />
  <InputText @bind-Value=person.Name />
  <ValidationMessage For=@(() => person.Name) />
  <InputDate @bind-Value=person.Birthday
             ParsingErrorMessage="Birthday must be a date" />
  <div>@person.Name is @person.Age years old</div>
  <input type="submit" value="Save" class="btn btn-primary" />
</EditForm>
```

Now run the app, blank the `Name` field, and click the button.


EditForm Example

- The Name field is required.

The Name field is required.

08/21/1998

is 21 years old

The bullet list is generated from the `ValidationSummary` component, and the single line message under the `Name` field is generated by the `Validation` component.

You can use either or both of these components depending on the type of UI you are creating for your users.

Notice that the `OnSubmit` event *was not raised* in this case. When you use any `ValidationSummary` or `Validation` components within an `EditForm` the `OnSubmit` event will only be raised if no validation rules are broken.

At this point you should understand how you can create pages to display and edit data using one-way or two-way data binding, or using the `EditForm` component.

Event Binding

Binding applies to events and methods much like it does to fields and properties with data. You have already seen examples of event binding to handle the click of a `button` element and events on the `EditForm` component.

Binding to a Method

For example, in the `Counter` page there's a button that binds to a method.

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

The method is often referred to as an *event handler* because it executes in response to the `button` element's `onclick` event.

```
  private void IncrementCount()
  {
    currentCount++;
  }
```

Binding to a method is the simplest scenario.

Binding to a Lambda Expression

You can use more advanced expressions in the markup to handle events with a lambda expression instead of a simple method call.

Add a new `button` element to the `Counter` page.

```
<button class="btn btn-primary" @onclick="() => currentCount--">Decrement</button>
```

When this element is clicked it doesn't call a named method, it directly runs the lambda expression. This expression decrements the `currentCount` value.

This can be a powerful technique when some simple state or flag needs to be altered in response to the user interacting with the page. For example, a button click might toggle a `bool` value which is data bound so it hides or displays a section of the page.

Passing Parameters to a Method

Another variation on this lambda concept is to pass parameter values to a method. The simple approach of binding an event to a method call doesn't allow for parameters to be passed to the method, but you can overcome that limitation using a lambda.

Add a new `button` element to pass a parameter.

```
<button class="btn btn-primary" @onclick="() => IncrementCount(5)">Add 5</button>
```

Then create an overload of the `IncrementCount` method that accepts a parameter.

```
private void IncrementCount(int increment)
{
  currentCount += increment;
}
```

Data binding in Blazor enables display and formatting of data, input of data, and binding code to events. These combine to enable very powerful and responsive UI scenarios with relatively little code.

Conclusion

In this chapter I discussed Blazor components and pages, the component lifecycle, routing and navigation, and data binding. These features will be used throughout the rest of the book as I talk about how CSLA .NET and Blazor work together, and finally walk through a complete app UI built using Blazor that leverages a CSLA based business domain layer. Before getting into CSLA .NET it is important to cover two more topics: authentication and authorization, and building multi-headed Blazor apps.

Chapter 5: Authentication and Authorization

Authentication is the process of having a user prove their identity. Authorization uses the now-known user identity to apply business rules about what the user can and can't do within the app.

The .NET platform supports role-based and claims-based authorization. In ASP.NET Core, including Blazor, the user's identity is maintained in a `ClaimsPrincipal` and `ClaimsIdentity` that supports both models.

> ⚠ Outside of ASP.NET Core it is possible to use the older `IPrincipal` and `IIdentity` types, which only support role-based authorization. This is not recommended however, because Microsoft continues to move toward supporting *only* `ClaimsPrincipal` and `ClaimsIdentity` types.

Server-Side Blazor Authentication

Server-side Blazor provides a number of authentication options to establish the user's identity. These include:

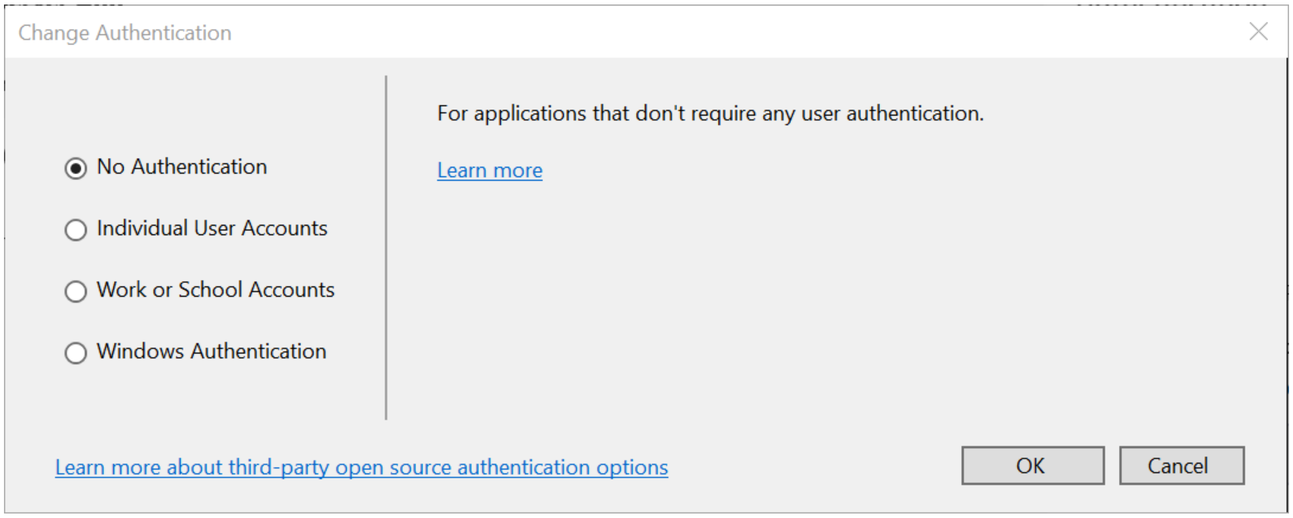
1. No authentication
2. Individual user accounts
3. Work or school accounts
4. Windows authentication

It is also possible to implement custom authentication with your own login and logout pages.

I will describe each option, but will focus on implementing custom authentication in this chapter.

No Authentication

This is the default option, and selecting this option means the project template will not add any authentication scaffolding into the project.



Change Authentication

[✓ No Authentication] Individual User Accounts Work or School Accounts Windows Authentication

For applications that don't require any user authentication.

Learn more

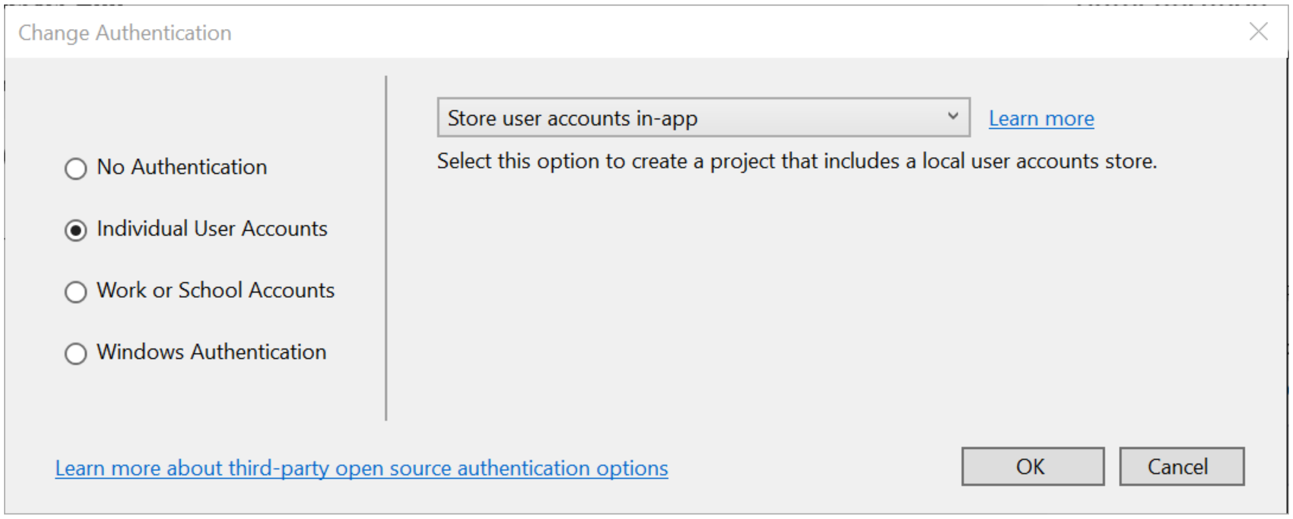
Learn more about third-party open source authentication options

OK Cancel

If you are implementing custom authentication this is the option you will choose.

Individual User Accounts

When this option is selected the project template adds support for the ASP.NET Core identity framework to your project. This framework stores user data in a SQL database and provides password recovery features as well as password-based user login.


Change Authentication

No Authentication [✓ Individual User Accounts] Work or School Accounts Windows Authentication

Store user accounts in-app

Learn more

Select this option to create a project that includes a local user accounts store.

Learn more about third-party open source authentication options

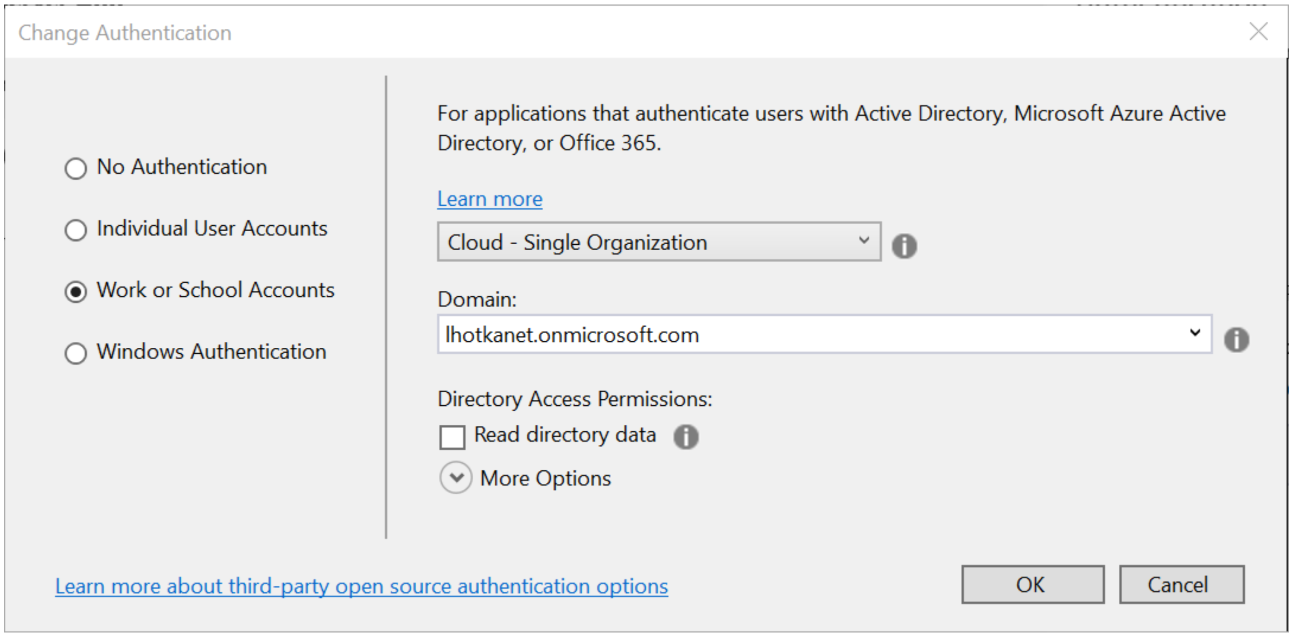
The dialog gives you the option to use a local or cloud-based SQL database to store the user information.

You will likely choose this option if you do not already have an authentication process in your organization, as this option provides a turnkey solution for user management. If the user is authenticated your app will get a user identity object representing the roles defined for the user in the SQL database tables.

Work or School Accounts

Microsoft 365 and Office 365 make use of Azure Active Directory (AAD) to enable single sign-on (SSO) for users of those services. Because many organizations use AAD to authenticate their users for other purposes, it is often attractive option because your users won't need a new login id or password. They will log into your app with the same credentials they use for Office 365 or Windows.

Choosing this option will require that you provide information about your AAD tenant.


Change Authentication

No Authentication Individual User Accounts [✓ Work or School Accounts] Windows Authentication

For applications that authenticate users with Active Directory, Microsoft Azure Active Directory, or Office 365.

Learn more

Cloud - Single Organization

Domain:
lhotkanet.onmicrosoft.com

Directory Access Permissions:
☐ Read directory data

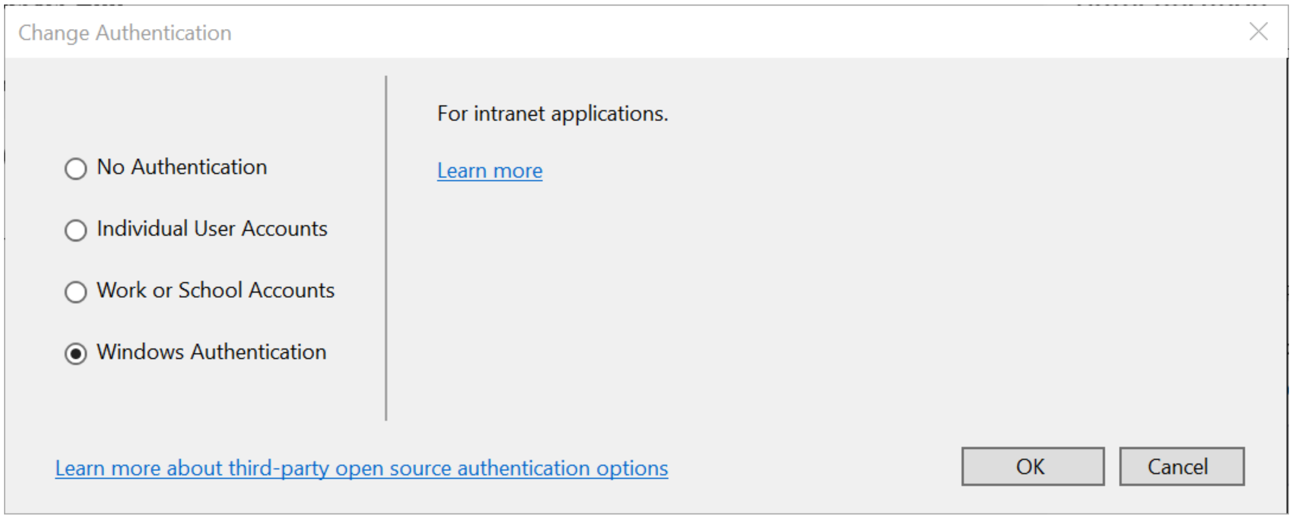
More Options

Learn more about third-party open source authentication options

Work or school authentication delegates the authentication process to Microsoft Azure. If the user is authenticated your app will get a user identity object representing the AAD groups the user belongs to in your organization.

Windows Authentication

Before AAD became the defacto standard, many organizations maintained their own private Windows Active Directory (AD) in their data centers. This authentication option in Blazor allows your users to log into your app using their regular AD credentials, probably the same user id and password they use to log into Windows or other intranet resources.


Change Authentication

For intranet applications.

Learn more

- No Authentication
- Individual User Accounts
- Work or School Accounts
- [✓ Windows Authentication]

Learn more about third-party open source authentication options

If your organization still uses AD to authenticate users this may be an attractive option.

At this point you should understand the basic capabilities of the built-in authentication options provided by server-side Blazor. If your oganization uses AAD or AD you should prefer to use those options as they provide single sign-on for your users. If you have no existing authentication model at all, the individual user account option may make sense.

If you do have an existing authentication mechanism in your organization it is possible to implement custom authentication.

Implementing Custom Authentication

Implementing custom authentication requires that you create login and logout pages in your app, routing to those pages as appropriate, and configuring the app for authentication and authorization.

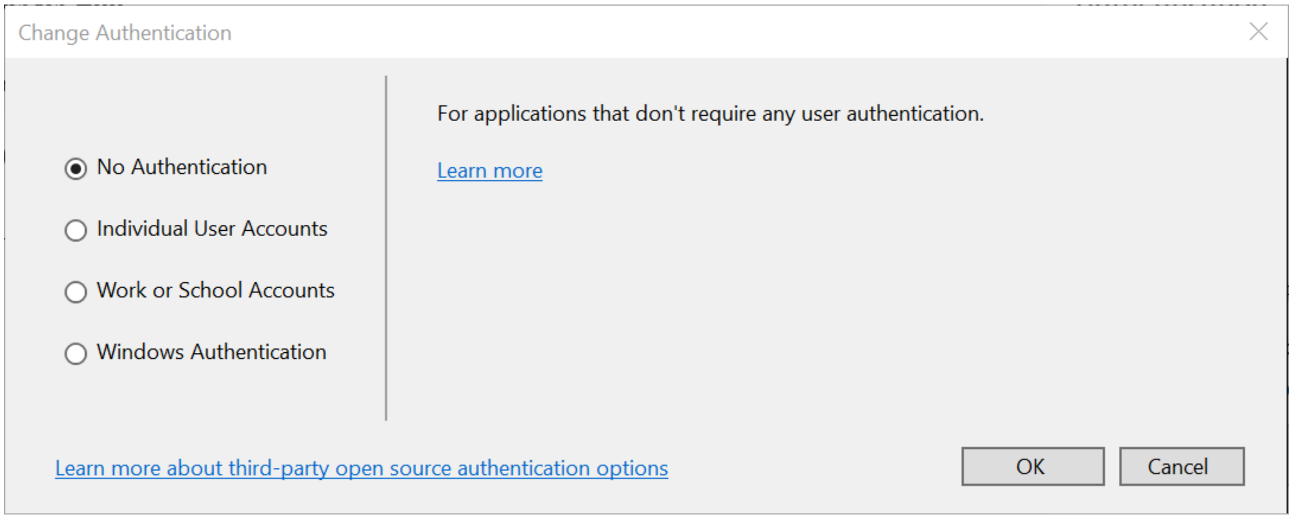
The specific steps you will use to authenticate your users and to retrieve their roles or claims will depend on your existing authentication mechanism. For the purposes of learning how the process works I will be using a set of hard-coded username and password values.

Open Visual Studio 2019 and create a new server-side Blazor project named `BlazorAuthServer` as described in Chapter 2.

> The code for this example is the `BlazorAuthServer` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Select No Authentication

When you get to the Create a new Blazor app dialog, notice the Change link under Authentication. You don't need to click it, because the default is No Authentication, but feel free to click the link and explore the options.


Change Authentication

(● No Authentication) ○ Individual User Accounts ○ Work or School Accounts ○ Windows Authentication

For applications that don't require any user authentication.

Learn more

Learn more about third-party open source authentication options

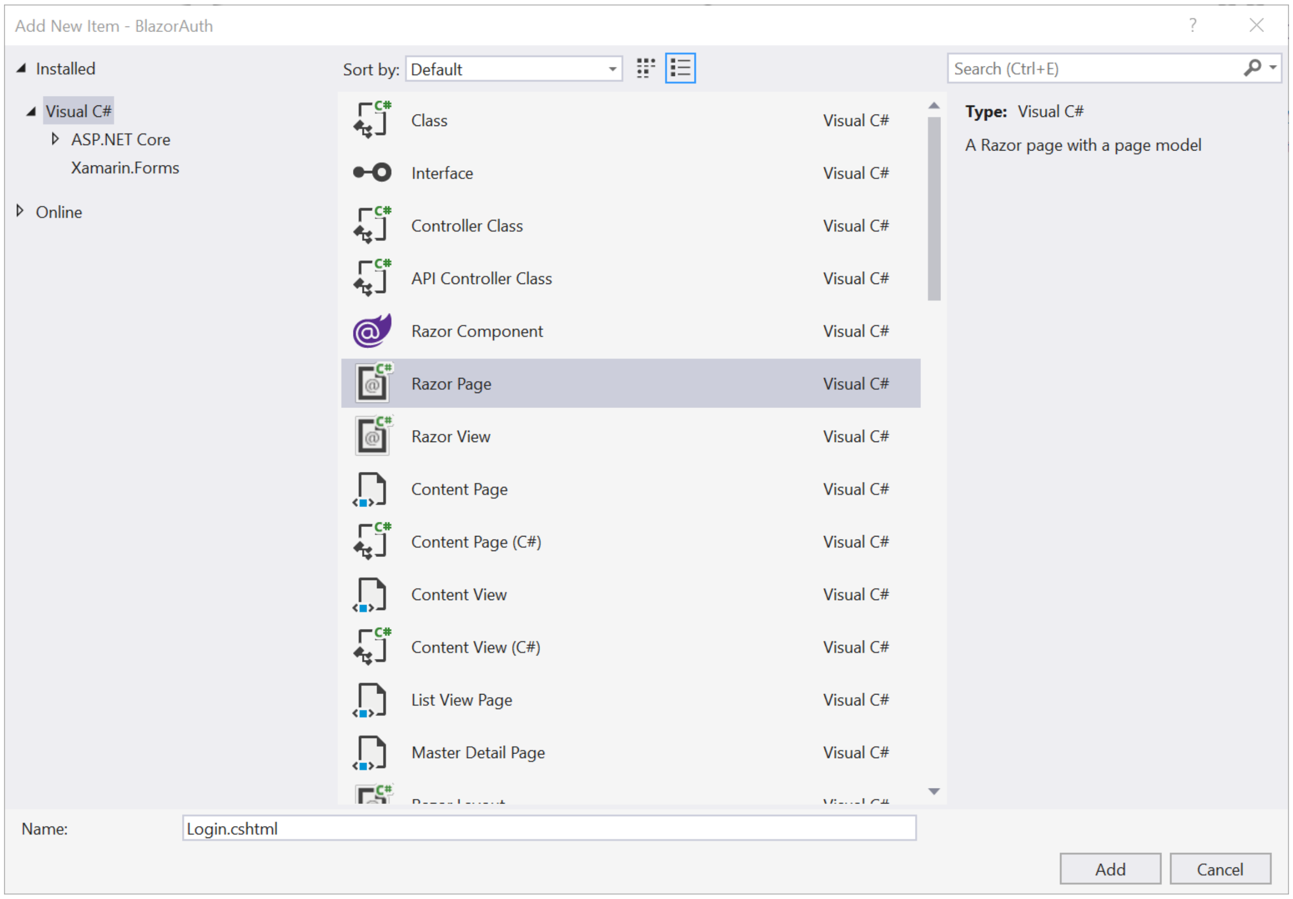
Once the solution is open in Visual Studio you can proceed to the next step.

Login Page Implementation

The process of authentication needs to occur outside the Blazor framework itself, so your login and logout pages will be Razor Pages with a `.cshtml` extension.

I prefer to keep my non-Blazor pages separate from my Blazor pages, so in the project create an Areas\Account\Pages folder structure to hold the `Login` and `Logout` pages.

Add a new Razor Page item to the Areas\Account\Pages folder in the project, named `Login`.


Add New Item - BlazorAuth

Installed
Visual C#
ASP.NET Core
Xamarin.Forms
Online

Sort by: Default

Class	Visual C#
Interface	Visual C#
Controller Class	Visual C#
API Controller Class	Visual C#
Razor Component	Visual C#
Razor Page	Visual C#
Razor View	Visual C#
Content Page	Visual C#
Content Page (C#)	Visual C#
Content View	Visual C#
Content View (C#)	Visual C#
List View Page	Visual C#
Master Detail Page	Visual C#

Search (Ctrl+E)

Type: Visual C#

A Razor page with a page model

Name: Login.cshtml

Here is the complete code for the page content. I'll discuss the important parts.

```
@page
@model BlazorAuthServer.Account.Pages.LoginModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Blazor Custom Authentication</title>
  <base href="~/" />
  <link rel="stylesheet" href="~/css/bootstrap/bootstrap.min.css" />
  <link rel="stylesheet" href="~/css/bootstrap4-toggle.min.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
  <app>
    <div class="sidebar">
    </div>
    <div class="main">
      <div class="top-row px-4">
        <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
      </div>
      <div class="content px-4">
        @if (!string.IsNullOrWhiteSpace(Model.ErrorText))
        {
          <div class="alert alert-danger">@Model.ErrorText</div>
        }
        <div>
          <form method="post">
            <div class="form-group">
              <label asp-for="Username"></label>
              <input asp-for="Username" class="form-control">
              <div class="invalid-feedback"></div>
            </div>

            <div class="form-group">
              <label asp-for="Password"></label>
              <input asp-for="Password" class="form-control" type="password">
              <div class="invalid-feedback"></div>
            </div>

            <button type="submit" class="btn btn-primary">Login</button>
          </form>
        </div>
      </div>
    </div>
  </app>
</body>
</html>
```

The page displays `Model.ErrorText` to provide the user with any error information.

Within the `form` element the page allows the user to enter username and password values using standard Razor markup data binding. The `Login` button triggers a postback to the server that is handled by the `OnPostAsync` method in the page's code.

Here's the code behind the page.

```
[AllowAnonymous]
public class LoginModel : PageModel
{
  [BindProperty]
  public string Username { get; set; }
  [BindProperty]
  public string Password { get; set; }
  [BindProperty]
  public string ErrorText { get; set; }

  private static Dictionary<string, string> Users = new Dictionary<string, string>
  {
    { "rocky", "mypassword" },
    { "andrew", "otherpassword" }
  };

  public async Task<IActionResult> OnPostAsync()
  {
    if (!ModelState.IsValid)
    {
      ErrorText = "Form has validation errors.";
      return Page();
    }

    if (!string.IsNullOrWhiteSpace(Username) &&
        Users.TryGetValue(Username.ToLower(), out string pw))
    {
      if (pw == Password)
      {
        var identity = new ClaimsIdentity("password");
        identity.AddClaim(new Claim(ClaimTypes.Name, Username.ToLower()));
        var principal = new ClaimsPrincipal(identity);
        var authProperties = new AuthenticationProperties();
        await HttpContext.SignInAsync(
          CookieAuthenticationDefaults.AuthenticationScheme,
          principal,
          authProperties);

        return LocalRedirect(Url.Content("~/"));
      }
    }
    ErrorText = "Invalid credentials";
    return Page();
  }
}
```

Notice the use of the `AllowAnonymous` attribute. This is important because the user won't be authenticated when they login, so they are an anonymous user. The `Login` page won't allow an anonymous user to do a postback, so this attribute is necessary for the page to operate.

Also notice that the "security database" in this sample is a static dictionary of hard-coded username/password values. In a real app you would be interacting with a database, LDAP server, or some other location where the user's identity can be verified.

The `BindProperty` attribute is used to indicate the fields available for data binding in the page. These fields correspond to the data binding expressions in the Razor markup.

The `OnPostAsync` method is invoked when the user triggers a postback by clicking the `Login` button on the page. If the user entered a value for the `Username` field, that value is used to look up the required password from the static dictionary.

Again, in a real app this is where you would make any appropriate database or service call to validate the user's credentials to ensure they are valid.

If the user's credentials are valid, the next step is to create a `ClaimsIdentity` object.

```
        var identity = new ClaimsIdentity("password");
        identity.AddClaim(new Claim(ClaimTypes.Name, Username.ToLower()));
```

At a minimum, this identity object will contain the type of authentication used (in this case `password`) and the username as a claim of type `ClaimTypes.Name`. You will probably add other claims as well depending on how your authorization rules will use those claims.

> ℹ Claims might be a role, a group, a department, or any other identifying fact about the user that will be used to determine if or how the user will be allowed to interact with the app.

Once you have an identity object, the next step is to create a `ClaimsPrincipal` object to contain the identity.

```
        var principal = new ClaimsPrincipal(identity);
```

Within .NET the user principal is always the primary object used to manage the user's identity, roles, and claims.

Finally, it is necessary to create a browser cookie to maintain the user's identity over time. The use of a cookie for this purpose is a time-tested standard within the web server world, because each request from the browser to the web server always includes the web site's cookies.

```
        var authProperties = new AuthenticationProperties();
        await HttpContext.SignInAsync(
          CookieAuthenticationDefaults.AuthenticationScheme,
          principal,
          authProperties);
```

This means that each time the browser sends a request to the web server this cookie can be used to determine the user's identity.

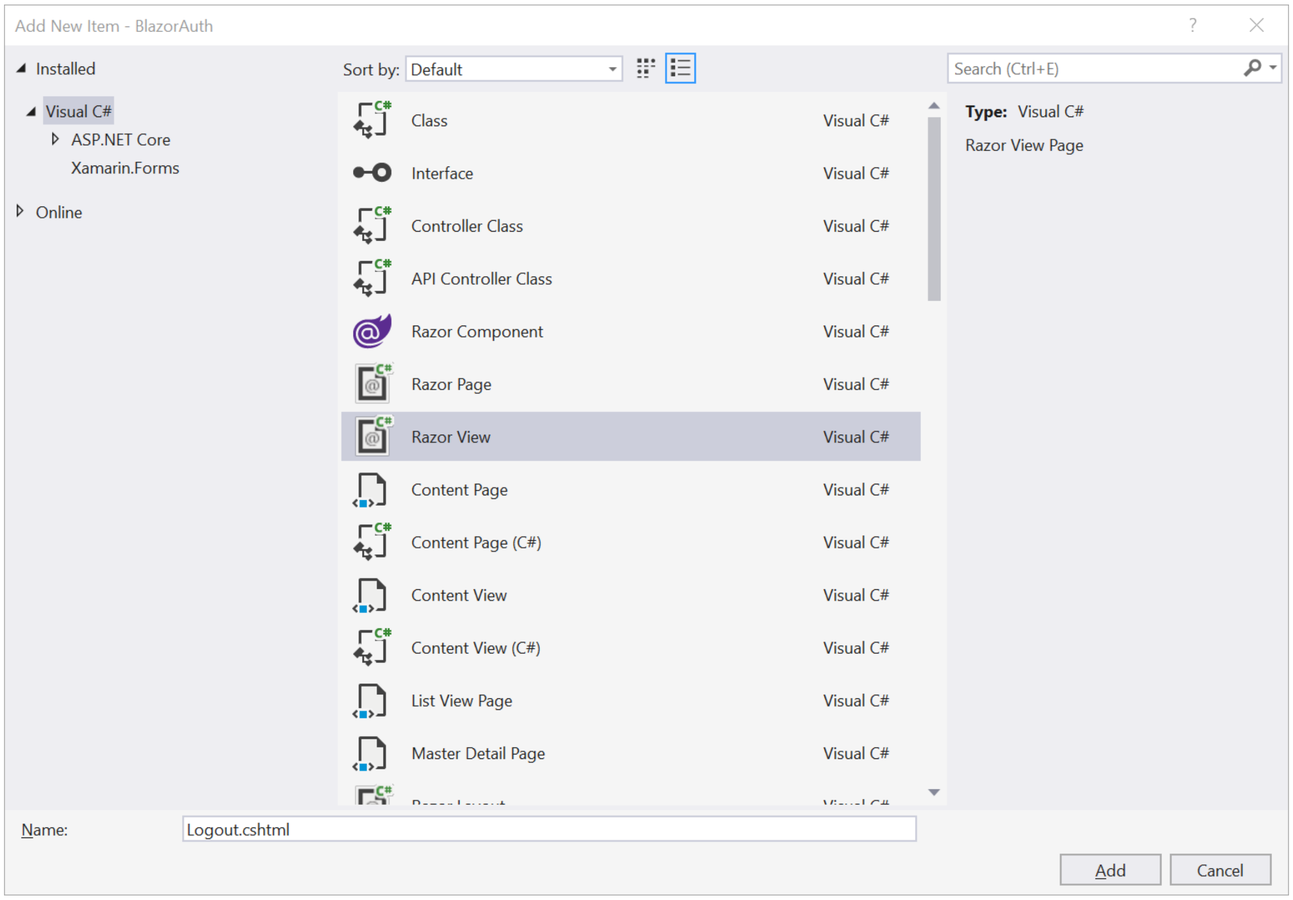
Now that the user is logged in, the page redirects to the Blazor app start page.

```
        return LocalRedirect(Url.Content("~/"));
```

Once routing and configuration are complete, this page will allow the user to log into the app.

Logout Page Implementation

Allowing the user to log out of the app is even simpler, because no UI is required. Add a `Logout` page to the Areas/Accounts/Pages folder.


Add New Item - BlazorAuth

Installed
- Visual C#
  - ASP.NET Core
  - Xamarin.Forms
- Online

Sort by: Default

Search (Ctrl+E)

Class	Visual C#
Interface	Visual C#
Controller Class	Visual C#
API Controller Class	Visual C#
Razor Component	Visual C#
Razor Page	Visual C#
Razor View	Visual C#
Content Page	Visual C#
Content Page (C#)	Visual C#
Content View	Visual C#
Content View (C#)	Visual C#
List View Page	Visual C#
Master Detail Page	Visual C#

Type: Visual C#

Razor View Page

Name: Logout.cshtml

Notice that this is a Razor View item, not a Razor Page. This is because there's no need for any UI, just some code to clear out the cookie.

```
@page
@using Microsoft.AspNetCore.Authentication
@using Microsoft.AspNetCore.Authentication.Cookies
@attribute [IgnoreAntiforgeryToken]
@functions {
  public async Task<IActionResult> OnPost()
  {
    if (User.Identity.IsAuthenticated)
    {
      await HttpContext.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);
    }

    return Redirect("~/");
  }
}
```

The code removes the authentication cookie.

```
      await HttpContext.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);
```

And then redirects the user back to the main page.

```
    return Redirect("~/");
```

The result is that the user is logged out and is treated as an anonymous user because there's no longer an authentication cookie.

The user identity needs to be made available to all Blazor components in the app before it can be used.

App Content

As I discussed in Chapter 4, Blazor uses cascading parameters to provide values to UI elements, and that is true for the user identity as well.

To make the user's identity available to all UI elements in the app, edit the `App` page and wrap the contents in a `CascadingAuthenticationState` element. This element provides the user identity as a cascading parameter to all Blazor components in the app.

Also change the `RouteView` element to an `AuthorizeRouteView` element.

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

The `AuthorizeRouteView` element changes the behavior of Blazor routing so users are not allowed to see pages that require authorization.

At this point pages exist to allow the user to log in and out of the app, but there's no routing or navigation for the user to reach these pages.

MainLayout Content

Open the `MainLayout` page in the Shared folder and edit it to match the following.

```
@inherits LayoutComponentBase

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

<div class="main">
  <div class="top-row px-4">
    <AuthorizeView>
      <Authorized>
        Hello, @context.User.Identity.Name
        <form method="post" action="/Account/Logout">
          <button type="submit" class="nav-link btn btn-link">Log out</button>
        </form>
      </Authorized>
      <NotAuthorized>
        <a href="/Account/Login">Log in</a>
      </NotAuthorized>
    </AuthorizeView>
    <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
  </div>

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

I will discuss the `AuthorizeView`, `Authorized`, and `NotAuthorized` elements later in this chapter. You can infer what they do however, and understand that the Log out and Log in links will appear in the app header depending on whether the user is currently logged in.

Startup Configuration

The final step before the custom authentication implementation will work is to do some configuration as the web site starts up. This is done in the `Startup` class.

In the `ConfigureServices` method, the `AddAuthentication` method is used to add and configure authentication.

```
services.AddAuthentication(options =>
{
  options.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultChallengeScheme = CookieAuthenticationDefaults.AuthenticationScheme;
})
  .AddCookie();
```

This must occur before adding the Razor Pages and server-side Blazor services.

Then in the `Configure` method the web site needs to be configured to use authentication and authorization.

```
app.UseAuthentication();
app.UseAuthorization();
```

These methods must be called after the `UseRouting` method is called, and before setting up the endpoints.

At this point you should be able to run the app and successfully log in and out by using the links in upper-right corner of the UI.

Client-Side Blazor Authentication

Client-side Blazor supports two authentication options to establish the user's identity. These include:

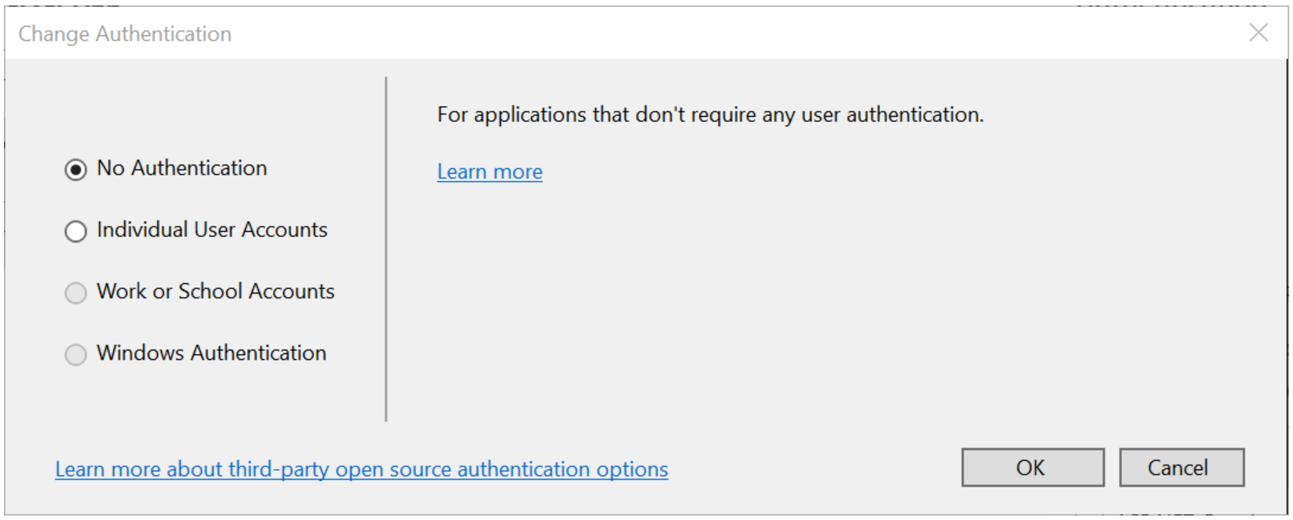
1. No authentication
2. Individual user accounts

It is also possible to implement custom authentication with your own login and logout pages.

I will describe each option, but will focus on implementing custom authentication in this chapter.

No Authentication

This is the default option, and selecting this option means the project template will not add any authentication scaffolding into the project.

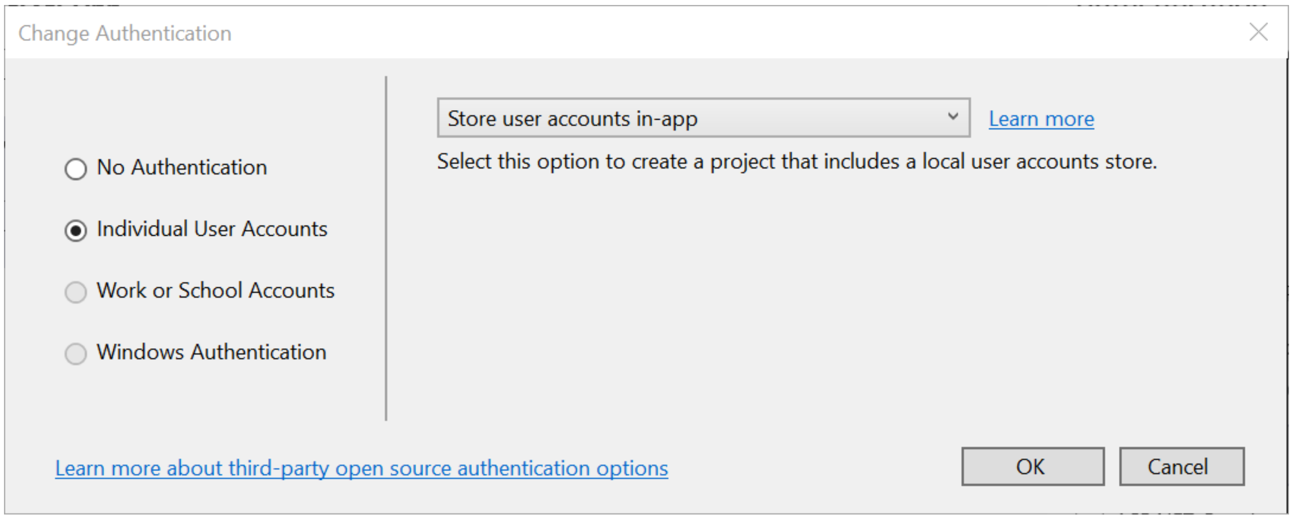


For applications that don't require any user authentication.

If you are implementing custom authentication this is the option you will choose.

Individual User Accounts

When this option is selected the project template adds support for the ASP.NET Core identity framework to your project. This framework stores user data in a SQL database and provides password recovery features as well as password-based user login.


Change Authentication

- No Authentication
- [✓ Individual User Accounts]
- Work or School Accounts
- Windows Authentication

Store user accounts in-app

Learn more

Select this option to create a project that includes a local user accounts store.

Learn more about third-party open source authentication options

The dialog gives you the option to use a local or cloud-based SQL database to store the user information.

You will likely choose this option if you do not already have an authentication process in your organization, as this option provides a turnkey solution for user management. If the user is authenticated your app will get a user identity object representing the roles defined for the user in the SQL database tables. A client-side Blazor app runs entirely in the browser on the client device, just like smart client apps have for decades with Windows Forms, WPF, and other technologies.

Custom Authentication

In nearly every case, when a user logs in using a smart client app, the app calls some server-side service to verify the user's credentials, and to retrieve any roles, claims, or other user information if those credentials are valid. There is no standard solution to this scenario, so it is up to you to provide the service and the client-side code that calls the service.

To implement custom authentication, make sure to select the No Authentication option within the Visual Studio wizard when creating your Blazor project.

> ⚠ Code running on a client device is always vulnerable to being compromised by a malicious actor. The user principal and identity can be used on the client to provide a rich user experience, but you can not count on these values from a security perspective.

Because ASP.NET and Blazor authorization assume the use of the `ClaimsPrincipal` and `ClaimsIdentity` types, the goal of any client-side authentication implementation should be to create a principal and identity in the client app by using those types.

There are a number of steps required to implement client-side authentication in Blazor, including:

1. Define message types shared between client and server
2. Implement server-side authentication controller (service)
3. Implement client-side types to manage user identity

4. Implement client-side login and logout pages
5. Add client-side navigation to login and logout pages

I will walk you through a very basic example to explain each of these steps, highlighting where you would write actual user authentication code in a real app.

Open Visual Studio 2019 and create a new ASP.NET Core hosted client-side Blazor project named `BlazorAuthClient` as described in Chapter 3.

> The code for this example is in GitHub: https://github.com/rockfordlhotka/BlazorAuthClient

Authentication Controller Implementation

The actual process of validating the user's credentials to determine if they are valid, and retrieving any claims, roles, or other information about the user, is all handled in a service that runs on a server. In this example you will implement the service in the ASP.NET Core web site that hosts the client-side Blazor app. This web site already provides the `WeatherForecast` service to the client app.

Before implementing the controller, you need to create the classes that define the messages passed from the client app to the service, and returned from the service to the client.

UserCredentials Class

Add a `UserCredentials` class to the `BlazorAuthClient.Shared` project. This will define the message passed from the client app to the service.

```
public class UserCredentials
{
  public string Username { get; set; }
  public string Password { get; set; }
}
```

The client app will get the user's credentials and send them to the service for validation.

UserIdentity Class

If the user's credentials are valid, the service should return all user information necessary for the client to create a `ClaimsIdentity` object. Create a `UserIdentity` class in the `BlazorAuthClient.Shared` project.

```
public class UserIdentity
{
  public string Name { get; set; }
  public string AuthenticationType { get; set; }
  public bool IsAuthenticated { get; set; }
  public List<ClaimInfo> Claims { get; set; } = new List<ClaimInfo>();
}

public class ClaimInfo
{
  public string ClaimType { get; set; }
  public string Claim { get; set; }
}
```

Mostly what needs to be returned is a list of claims, represented by the `ClaimInfo` class. To create a `ClaimsIdentity` class you also need to provide the authentication type, and as a convenience the message will also return a `bool` indicating whether the user was successfully authenticated.

Controller Implementation

Using the message types you just defined, it is now possible to implement the `AuthenticationController`. Add a class to the `BlazorAuthClient.Server` project in the Controllers folder.

```
[ApiController]
[Route("[controller]")]
public class AuthenticationController
{
  private static Dictionary<string, string> Users = new Dictionary<string, string>
  {
    { "rocky", "mypassword" },
    { "andrew", "otherpassword" }
  };

  [HttpPost]
  public UserIdentity Post(UserCredentials credentials)
  {
    var result = new UserIdentity();
    if (!string.IsNullOrWhiteSpace(credentials.Username) &&
        Users.TryGetValue(credentials.Username.ToLower(), out string pw) &&
        pw == credentials.Password)
    {
      result.IsAuthenticated = true;
      result.AuthenticationType = "password";
      result.Name = credentials.Username.ToLower();
      result.Claims.Add(new ClaimInfo
        { ClaimType = ClaimTypes.Name, Claim = credentials.Username.ToLower() });
    }
    return result;
  }
}
```

The implementation of this service is virtually identical to the `Login` page implementation from the server-side example. You can see how the username and password are validated. If they are valid a claim is added for the username. As always, this is the minimum claim necessary for a valid user identity.

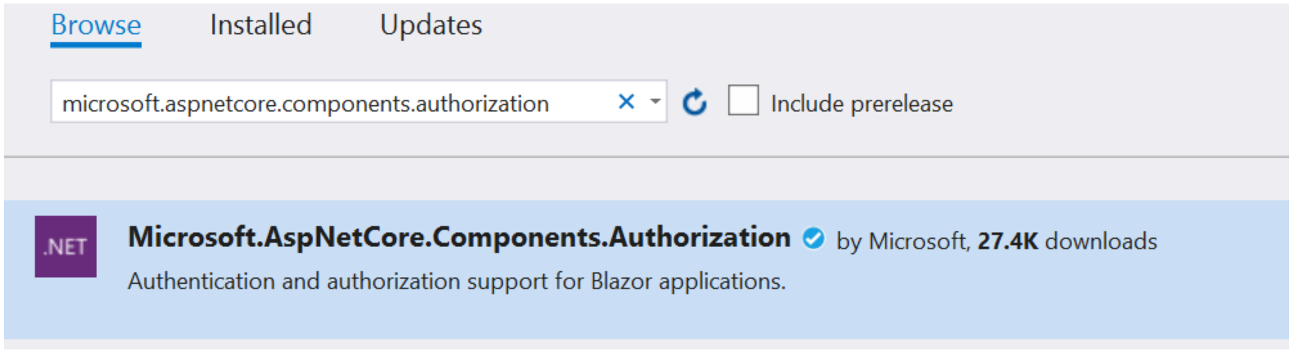
I am using a dictionary of hard-coded values as a "user database". In a real app you would access a database or some other service to validate the user's credentials and retrieve the user's claims or roles.

This service returns a `UserIdentity` message regardless of whether the user was authenticated or not. Both are "valid" responses from the service, and it is the responsibility of the calling code to examine the `IsAuthenticated` value to determine if the result represents a valid user identity.

Now let's move to the client-side Blazor project and implement the authentication that relies on this service.

Package References and Global Namespaces

Before you can implement authentication in a client-side Blazor project it is necessary to add a reference to the `Microsoft.AspNetCore.Components.Authorization` NuGet package.



Microsoft.AspNetCore.Components.Authorization by Microsoft, 27.4K downloads
Authentication and authorization support for Blazor applications.

With that done, edit the `_Imports.razor` file in the client-side project to add the following lines.

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
```

These namespaces contain the types necessary for authentication and authorization in the client-side components.

Managing the User Identity

In a server-side Blazor app you can rely on the `HttpContext` object to maintain the current user's identity and make it available to your code. Client-side Blazor doesn't have an `HttpContext` and so it is necessary to manage the user's identity in a different manner.

There are two concepts at play here. One is the need to keep the user's identity in memory so it is available throughout the lifetime of the app. The other is that Blazor defines something called an `AuthenticationStateProvider`, which is how all the Blazor components gain access to the current user identity.

User Service Implementation

You should understand that maintaining the user identity in memory is not the *only* option. You can also choose to store and retrieve the information from disk or a cookie. The important thing is that you need to keep the information available on the client as long as the app is running. I find that the easiest way to do this is by keeping the `ClaimsPrincipal` and `ClaimsIdentity` objects in memory.

To maintain the user identity in memory you will implement a class that manages the state, and configure this type as a singleton in the dependency injection subsystem.

Add a `CurrentUserService` class to the top level of the client-side project.

```
public class CurrentUserService
{
  private ClaimsPrincipal _currentUser;

  public event EventHandler<CurrentUserChangedEventArgs> CurrentUserChanged;

  public CurrentUserService()
  {
    CurrentUser = new ClaimsPrincipal(new ClaimsIdentity());
  }

  public ClaimsPrincipal CurrentUser
  {
    get
    {
      return _currentUser;
    }
    set
    {
      _currentUser = value;
      CurrentUserChanged?.Invoke(
        this, new CurrentUserChangedEventArgs() { NewUser = value });
    }
  }

  public class CurrentUserChangedEventArgs : EventArgs
  {
    public ClaimsPrincipal NewUser { get; set; }
  }
}
```

This service keeps a reference to the current `ClaimsPrincipal` object that represents the user.

It also exposes an event you can use to know when the current user has changed, which will be useful in keeping the UI up to date as the user logs in and out of the app.

AuthenticationStateProvider Implementation

Client-side Blazor relies on a `AuthenticationStateProvider` service to access the current user identity when necessary. All the built-in authorization elements and attributes make use of this service.

You need to create a subclass of this type to provide the current user identity upon request. Add a `CustomAuthenticationStateProvider` class to the top level of the client-side project.

```
public class CustomAuthenticationStateProvider : AuthenticationStateProvider
{
  private readonly CurrentUserService _currentUserService;

  public CustomAuthenticationStateProvider(CurrentUserService currentUserService)
  {
    _currentUserService = currentUserService;
    _currentUserService.CurrentUserChanged += (sender, e) =>
    {
      var authState = Task.FromResult(new AuthenticationState(e.NewUser));
      NotifyAuthenticationStateChanged(authState);
    };
  }

  public override Task<AuthenticationState> GetAuthenticationStateAsync()
  {
    return Task.FromResult(new AuthenticationState(_currentUserService.CurrentUser));
  }
}
```

The important part of this class is the `GetAuthenticationStateAsync` method. This is the method called by Blazor any time the framework needs access to the current user identity.

Startup Configuration

Now that you have types to keep the user's identity in memory, and to provide the identity to Blazor on-demand, it is necessary to configure the app to use these types.

In a server-side Blazor app edit the `Startup` class and add these lines to the `ConfigureServices` method.

```
    services.AddAuthorizationCore();
    services.AddSingleton<
      AuthenticationStateProvider, CustomAuthenticationStateProvider>();
    services.AddSingleton<CurrentUserService>();
```

In a client-side Blazor app edit the `Program` class and add these lines to the `Main` method.

```
    builder.Services.AddAuthorizationCore();
    builder.Services.AddSingleton<
      AuthenticationStateProvider, CustomAuthenticationStateProvider>();
    builder.Services.AddSingleton<CurrentUserService>();
```

The `AddAuthorizationCore` method tells Blazor that it can access the user identity as necessary.

The next line adds a singleton object, so one instance for the lifetime of the app, so Blazor understands to use the `CustomAuthenticationStateProvider` to retrieve the current user identity when necessary.

And finally, a singleton for `CurrentUserService` is added, because that is required by the `CustomAuthenticationStateProvider` type. The `CurrentUserService` instance will be used by pages within the app as well.

At this point the basic plumbing is in place so Blazor understands that it can use authorization, and has the type information necessary to retrieve the user identity. Let's implement the pages and navigation so the user can log in and out of the app.

Login Page

Add a `Login` Blazor Component to the Pages folder. It will need a `@page` directive along with some `@using` and `@inject` directives.

```
@page "/login"
@using BlazorAuthClient.Shared
@using System.Security.Claims
@inject HttpClient Http
@inject NavigationManager NavigationManager
@inject CurrentUserService CurrentUserService
```

Notice how the `CurrentUserService` singleton is injected, allowing the `Login` page to change the current user identity if appropriate.

The Razor markup in the page needs to show any error text, and allow the user to enter their credentials.

```
<h3>Login</h3>

@if (!string.IsNullOrWhiteSpace(ErrorText))
{
  <div class="alert-danger">@ErrorText</div>
}

<table class="table">
  <thead>
    <tr>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Username</td>
      <td><input @bind="Credentials.Username" /></td>
    </tr>
    <tr>
      <td>Password</td>
      <td><input @bind="Credentials.Password" type="password" /></td>
    </tr>
  </tbody>
</table>
```

Finally, the `@code` block needs to expose fields for data binding and implement the `LoginUser` method to handle the `button` element's click event.

```
private UserCredentials Credentials { get; set; } = new UserCredentials();
private string ErrorText { get; set; }

private async void LoginUser()
{
  ErrorText = string.Empty;
  try
  {
    var userinfo = await Http.PostJsonAsync<UserIdentity>(
      "Authentication", Credentials);
    if (userinfo.IsAuthenticated)
    {
      var identity = new ClaimsIdentity(userinfo.AuthenticationType);
      foreach (var item in userinfo.Claims)
        identity.AddClaim(new Claim(item.ClaimType, item.Claim));
      CurrentUserService.CurrentUser =
        new System.Security.Claims.ClaimsPrincipal(identity);
      NavigationManager.NavigateTo("/");
    }
    else
    {
      ErrorText = "Invalid credentials";
    }
  }
  catch (Exception ex)
  {
    ErrorText = ex.Message;
  }
  StateHasChanged();
}
```

The `LoginUser` method sends the credentials to the `Authentication` service in the web site.

```
    var userinfo = await Http.PostJsonAsync<UserIdentity>(
      "Authentication", Credentials);
```

The `Credentials` field will have the username and password values because it is data bound to the UI elements, and the `PostJsonAsync` method automatically converts the data to JSON and sends it to the service. The service returns a `UserIdentity` object indicating whether the credentials were valid.

If the credentials were valid the `IsAuthenticated` property will be `true` and the code can create a `ClaimsIdentity` based on the information returned from the service.

```
      var identity = new ClaimsIdentity(userinfo.AuthenticationType);
      foreach (var item in userinfo.Claims)
        identity.AddClaim(new Claim(item.ClaimType, item.Claim));
```

Using this new `ClaimsIdentity`, the next step is to create a `ClaimsPrincipal` and update the current user identity for the app by using the `CurrentUserService`.

```
      CurrentUserService.CurrentUser =
        new System.Security.Claims.ClaimsPrincipal(identity);
```

Finally, the `NavigationManager` instance injected at the top of the page is used to navigate to the start page for the app.

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

If the user is *not* successfully logged in the `ErrorText` field is updated with any exception message or other information so the user can tell why their credentials weren't accepted.

Note the explicit `StateHasChanged` method call at the bottom of the method. This will only run if the user was not logged in and `ErrorText` was updated. This call is necessary because Blazor is not able to automatically detect that the `ErrorText` value changed. The `StateHasChanged` method call ensures that the user will immediately see the error message.

Logout Page

Similar to the server-side example, the client-side `Logout` page has no UI and is just code. Add a `Logout` Blazor Component to the Pages folder.

```
@page "/logout"
@using System.Security.Claims
@inject NavigationManager NavigationManager
@inject CurrentUserService CurrentUserService

@code {
  protected override void OnInitialized()
  {
    var identity = new ClaimsIdentity();
    CurrentUserService.CurrentUser =
      new System.Security.Claims.ClaimsPrincipal(identity);
    base.OnInitialized();
    NavigationManager.NavigateTo("/");
  }
}
```

This page gets the `CurrentUserService` singleton and sets the current user identity to an empty (unauthenticated) `ClaimsPrincipal`.

```
    var identity = new ClaimsIdentity();
    CurrentUserService.CurrentUser =
      new System.Security.Claims.ClaimsPrincipal(identity);
```

It then navigates to the home page for the app.

App Content

The Blazor `AuthenticationStateProvider` makes the current user identity available to UI elements. As I discussed in Chapter 4, Blazor uses cascading parameters to provide this sort of value to UI elements, and that is true for user state as well.

To make the user's identity available to all UI elements in the app, edit the `App` page and wrap the contents in a `CascadingAuthenticationState` element. This element provides the user identity as a cascading parameter to all Blazor components in the app.

Also change the `RouteView` element to an `AuthorizeRouteView` element.

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

The `AuthorizeRouteView` element changes the behavior of Blazor routing so users are not allowed to see pages that require authorization.

MainLayout Content

The last step is to update the `MainLayout` page so the user can navigate to the `Login` and `Logout` pages, and to ensure that the current username is displayed when the user is logged into the app.

Add an `@inject` directive to the page.

```
@inject CurrentUserService CurrentUserService
```

And add a `@code` block that makes use of the `CurrentUserService` singleton to refresh the username display any time the user's identity is changed.

```
protected override void OnInitialized()
{
  CurrentUserService.CurrentUserChanged +=
    (sender, args) => StateHasChanged();
  base.OnInitialized();
}
```

Remember that the `OnInitialized` method is invoked one time as the component is loaded. The `MainLayout` component is loaded just once during the lifetime of the app, so this event handler will be set up just one time.

The event handler calls the `StateHasChanged` method so Blazor understands that the `MainLayout` state needs to be rendered any time the user identity changes.

In the Razor markup of the page, the `AuthorizeView`, `Authorized`, `NotAuthorized`, and `Authorizing` elements are used to update the display.

```
  <AuthorizeView>
    <Authorized>
      Hello, @context.User.Identity.Name
      <a href="Logout">Log out</a>
    </Authorized>
    <NotAuthorized>
      <a href="Login">Log in</a>
    </NotAuthorized>
    <Authorizing>
      Authorizing...
    </Authorizing>
  </AuthorizeView>
```

I will discuss these elements in more detail later in this chapter. For now it is sufficient to understand that they control the content shown to the user based on whether the user is currently authorized.

You should now be able to run the app and log in and out by using the hard-coded credentials in the `AuthorizationController` class.

Blazor Authorization

Authorization within Blazor, and .NET in general, is built on the assumption that the user's identity, claims, roles, and other information is available through the current principal object. The user must be logged into the app for the principal to be available.

The Blazor authorization behaviors are the same for server-side and client-side Blazor. In this section of the chapter I will use the `BlazorAuthServer` project to demonstrate how authorization works, but the same attributes and elements function in client-side Blazor as well unless noted otherwise.

App Configuration

Before authorization will work within Blazor, you need to have configured the app for authorization. This is different for server-side and client-side Blazor.

Server-Side Configuration

Before using authorization in server-side Blazor there are some requirements.

1. Authentication must be configured and implemented
2. Authorization must be configured
3. The `App` page must be correct

I have already discussed server-side authentication and how to edit the `App` page to enable authorization.

The only step I want to call out here is that in the `Startup` class, in the `Configure` method, you must call the `UserAuthorization` method to enable authorization.

```
    app.UseAuthorization();
```

This, combined with the other requirements, will enable authorization within your components and code.

Client-Side Configuration

As with server-side Blazor, when using authorization in server-side Blazor there are requirements that must be met.

1. Authentication must be configured and implemented
2. Authorization must be configured
3. The `App` page must be correct

I have already discussed client-side authentication and how to edit the `App` page to enable authorization.

You also must configure authorization in the `Program` class.

```
  public static async Task Main(string[] args)
  {
    var builder = WebAssemblyHostBuilder.CreateDefault(args);
    builder.RootComponents.Add<App>("app");

    builder.Services.AddSingleton(
      new HttpClient {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
    builder.Services.AddOptions();
    builder.Services.AddAuthorizationCore();
    builder.Services.AddSingleton
      <AuthenticationStateProvider, CustomAuthenticationStateProvider>();
    builder.Services.AddSingleton<CurrentUserService>();

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

Notice that the `Main` method includes authorization as a service, along with the authentication and current user service types discussed in this chapter.

```
    builder.Services.AddOptions();
    builder.Services.AddAuthorizationCore();
    builder.Services.AddSingleton
      <AuthenticationStateProvider, CustomAuthenticationStateProvider>();
    builder.Services.AddSingleton<CurrentUserService>();
```

This, combined with the other requirements, will enable authorization within your components and code.

AuthorizeRouteView Element

You saw earlier in the chapter how the `App` page uses the `AuthorizeRouteView` element so the user is prevented from viewing pages if they aren't authorized. This element works together with the `Authorize` attribute I will discuss later in this chapter.

It is possible to globally control what the user sees in place of a page when they are not authorized, or when authorization is still occurring. This is handled by the `NotAuthorized` and `Authorizing` child elements.

> ⚠ In the current version of Blazor these child attributes only work in client-side Blazor. Microsoft has indicated that these attributes will work in a future version of server-side Blazor.

In client-side Blazor you can edit the `App` page to use these elements within the `AuthorizeRouteView` element.

```
<AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
  <NotAuthorized>
    Hey, you shouldn't be here!
  </NotAuthorized>
  <Authorizing>
    Authorizing...
  </Authorizing>
</AuthorizeRouteView>
```

Perhaps the most important is the `NotAuthorized` element, because it allows you to override the default Not authorized text that is otherwise displayed.

Authorize Attribute

The default behavior for Blazor components and pages is to allow any user to view the content. You can use the `Authorize` attribute to prevent anonymous access to a page, or to require a specific role or policy.

This attribute is applied to a component using the `@attribute` directive, and controls whether the user can view the entire page or component.

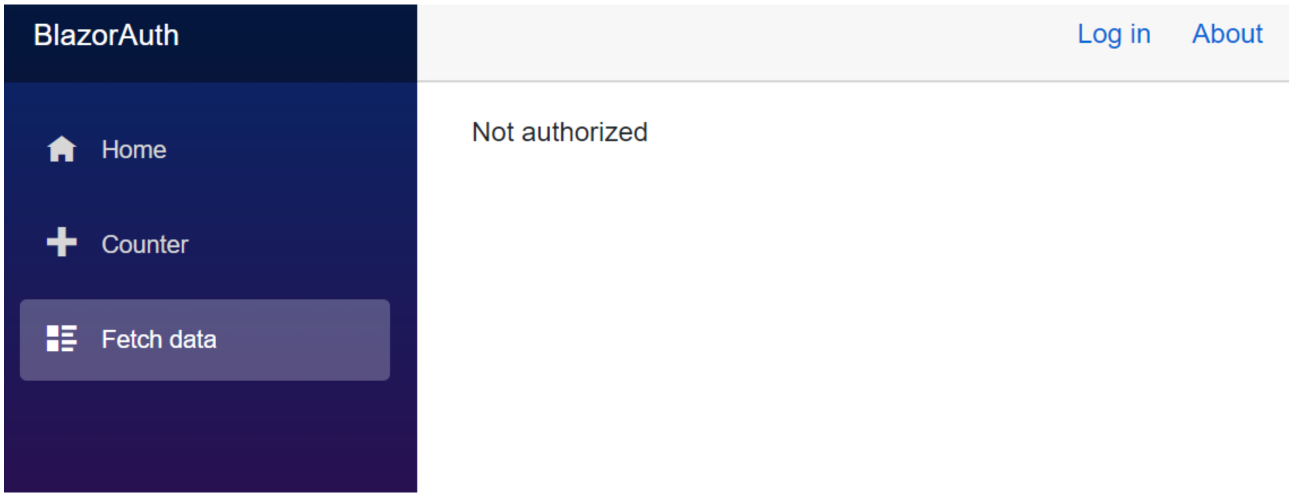
> ℹ I will discuss the `AuthorizeView` element later in this chapter. It allows you more control over authorization *within* the content of a page or component.

Prevent Anonymous Users

The simplest scenario is the use of the `Authorize` attribute to ensure that only an authorized user can access a page. Edit the `FetchData` page and add this directive.

```
@attribute [Authorize]
```

Run the app, make sure you are logged out, and try to navigate to the `FetchData` page. You should be prevented from seeing the page.



Not authorized

If you log in and navigate to the page the content will appear. This is because the user is authenticated, and thus meets the basic requirement to be authorized.

Require Roles

The `Authorize` attribute can also be used to require that the user be a member of a role. Edit the `FetchData` page and change the `Authorize` attribute as follows.

```
@attribute [Authorize(Roles = "admin")]
```

The string value for the `Roles` parameter is a comma-delimited list of roles. If the user is a member of any of the roles they will be authorized. Role names are case-sensitive.

Because Blazor uses the `ClaimsIdentity` type, roles are a type of claim. When creating a `ClaimsIdentity` with roles, you will add roles like this.

```
  identity.AddClaim(new Claim(ClaimTypes.Role, "admin"));
```

Edit the code for the `Login` page in the Areas\Account\Pages folder and add this line of code right after adding the `ClaimTypes.Role` claim.

You should now be able to run the app, log in, and view the `FetchData` page.

Require Policy

Role-based authorization has been in use for decades, and is well understood. However, it is a little inflexible, in that the role names need to be hard-coded throughout the pages of the app.

Policies provide a useful level of abstraction to roles, but go well beyond just roles. For example, you can define a policy that requires or allows anonymous users, requires specific email addresses, looks at a user's country of origin, and any other claims associated with the user's identity.

The `Authorize` attribute to use a policy looks like this.

```
@attribute [Authorize(Policy = "SpainOnly")]
```

Policies themselves are registered with Blazor in the `Program` class by adding services in the `Main` method. For example, here are three possible policies.

```
      builder.Services.AddAuthorization(config =>
      {
        config.AddPolicy("IsAuthenticated",
          policy => policy.RequireAuthenticatedUser());
        config.AddPolicy("IsAdmin",
          policy => policy.RequireRole("admin", "supervisor", "manager"));
        config.AddPolicy("SpainOnly",
          policy => policy.RequireClaim(ClaimTypes.Country, "es"));
      });
```

Policies provide a more elegant and maintainable way to define authorization rules. It is as easy to apply a policy using the `Authorize` attribute as it is a role, but the long-term benefits mean you should prefer to use policies.

The primary drawback to the `Authorize` attribute is that it applies to entire pages or components. In many apps it is far more common to allow users to see *part of a page* or interact with certain elements of data, rather than blocking the user entirely.

AuthorizeView Element

The `AuthorizeView` element and its child `Authorized`, `NotAuthorized`, and `Authorizing` elements provide more granular control over how authorization is applied within a page or component.

Detecting Anonymous Users

At a basic level the `AuthorizeView` element detects whether the current user is logged in or not.

Edit the `Index` page to add these elements.

```
@page "/"

<h1>Hello, world!</h1>

<p>Welcome to your new app.</p>

<AuthorizeView>
  <Authorized>
    User is authorized
  </Authorized>
  <NotAuthorized>
    User is not authorized
  </NotAuthorized>
  <Authorizing>
    Authorization is occurring
  </Authorizing>
</AuthorizeView>
```

Now run the app and notice how the main page switches its display depending on whether you are logged in or not.

As you would expect, when the user is logged in the content in the `Authorized` block is displayed, and when they are not then the content in the `NotAuthorized` block is displayed.

If asynchronous authorization is implemented at some point (a topic outside the scope of this book), the content in the `Authorizing` block would be displayed.

The `AuthorizeView` element allows you to provide `Roles` and `Policy` properties, much like the `Authorize` attribute.

Role Based Authorization

For example, you can specify a comma-delimited list of roles to the `AuthorizeView` element.

```
<AuthorizeView Roles="admin, manager">
```

Direct use of hard-coded role values has the same limitations as with the `Authorize` attribute.

Policy Based Authorization

The `AuthorizeView` element also allows you to provide a policy name.

```
<AuthorizeView Policy="IsAdmin">
```

This technique has the same maintainability and abstraction benefits as using the `Policy` property with the `Authorize` attribute.

You can use the `AuthorizeView` element in pages and components to control what content is rendered to the current user depending on whether they are authorized.

Block Anonymous Users App-Wide

In server-side Blazor you can use the `AuthorizedView` element to prevent anonymous users from accessing *all* pages in your app. To do this, edit the `MainLayout` page in the Shared folder and update the section of content that displays the `@body` content.

> ⚠ This technique will not work on client-side Blazor, because it will prevent the user from accessing the `Login` page. The server-side `Login` page is a Razor Page, not a Blazor page (Razor Component), so this technique works on the server.

```
<div class="content px-4">
  <AuthorizeView>
    <Authorized>
      @Body
    </Authorized>
    <NotAuthorized>
      Hey, this isn't for you!
    </NotAuthorized>
    <Authorizing>
      Authorizing...
    </Authorizing>
  </AuthorizeView>
</div>
```

This will prevent all Blazor pages from being rendered until the user has logged into the app.

At this point you should understand the requirements and configuration necessary for server-side and client-side authorization, along with the attributes and elements you can use to control what is rendered for the user.

Conclusion

In this chapter I discussed Blazor authentication for server-side and client-side scenarios. You have seen how authentication is quite different between these two scenarios. Once the user has been authenticated and the current user's identity is available to the app you should now understand how authorization can be applied through the use of roles and other claims.

Chapter 6 will cover the creation of a Blazor app that has both server-side and client-side experiences, with a common UI shared between both apps.

Chapter 6: Multi-Headed Blazor Solutions

It is possible to create a Blazor solution that includes multiple "heads". In this context a head is a particular UI platform, such as server-side or client-side hosting of your code. The goal is to create a solution where most of the app code is common, and is leveraged by multiple UI platforms.

With Blazor the common scenario is to have a server-side Blazor project (head), a client-side Blazor project (head), and a Blazor Class Library containing the common UI implementation. This allows you to maintain most of the code once, but allow users to interact with your app using server-side or client-side Blazor deployments.

> A similar concept can be found in Xamarin, where a solution often contains an iOS head, and Android head, a Windows UWP head, and the class library with all the common UI implementation for the app.

There are three big reasons this is important.

1. You may choose to have users interact with the server-side or client-side head depending on the capabilities of their client device, bandwidth, or other environmental factors
2. You may want to add other heads, such as using Electron or other UI technologies that become available in the future
3. It is often easier to debug Blazor UI code using server-side Blazor, but your planned user experience might be client-side Blazor; so developer productivity is enhanced if they can debug with server-side and test with client-side Blazor against the same codebase

In Chapter 3 you saw how the Blazor WebAssembly project template in Visual Studio creates three projects:

1. Client-side Blazor project
2. ASP.NET Core host project
3. Shared code project

A multi-headed solution is similar, but instead of a shared *code* project, there is a shared *UI* project:

1. Shared Blazor UI project
2. Client-side Blazor project
3. Server-side Blazor project

The client-side and server-side Blazor projects are the "heads", one for WebAssembly and the other for ASP.NET Core. They both reference the shared UI project and so have access to the shared UI components.

If you want to support more heads, such as the experimental WebWindow or Electron, you can add more Blazor projects that also reference the shared UI and code projects.

Unfortunately there is no pre-built template for a multi-headed solution. The good news is that it isn't hard to set up your own multi-headed solution.

Creating a Blazor Solution

In this chapter I will show how to start with an ASP.NET Core hosted Blazor WebAssembly solution and use it to create a multi-headed solution. The high level steps are:

1. Create an blank solution
2. Add a Server-side Blazor project
3. Add a Client-side Blazor project

4. Add a Shared Blazor UI project
5. Move Blazor pages and components to the shared UI project

Once the basic solution has been created I will demonstrate how to provide different implementations of key functionality in the client and server heads of the app. I'll use the standard forecast data from the Blazor template:

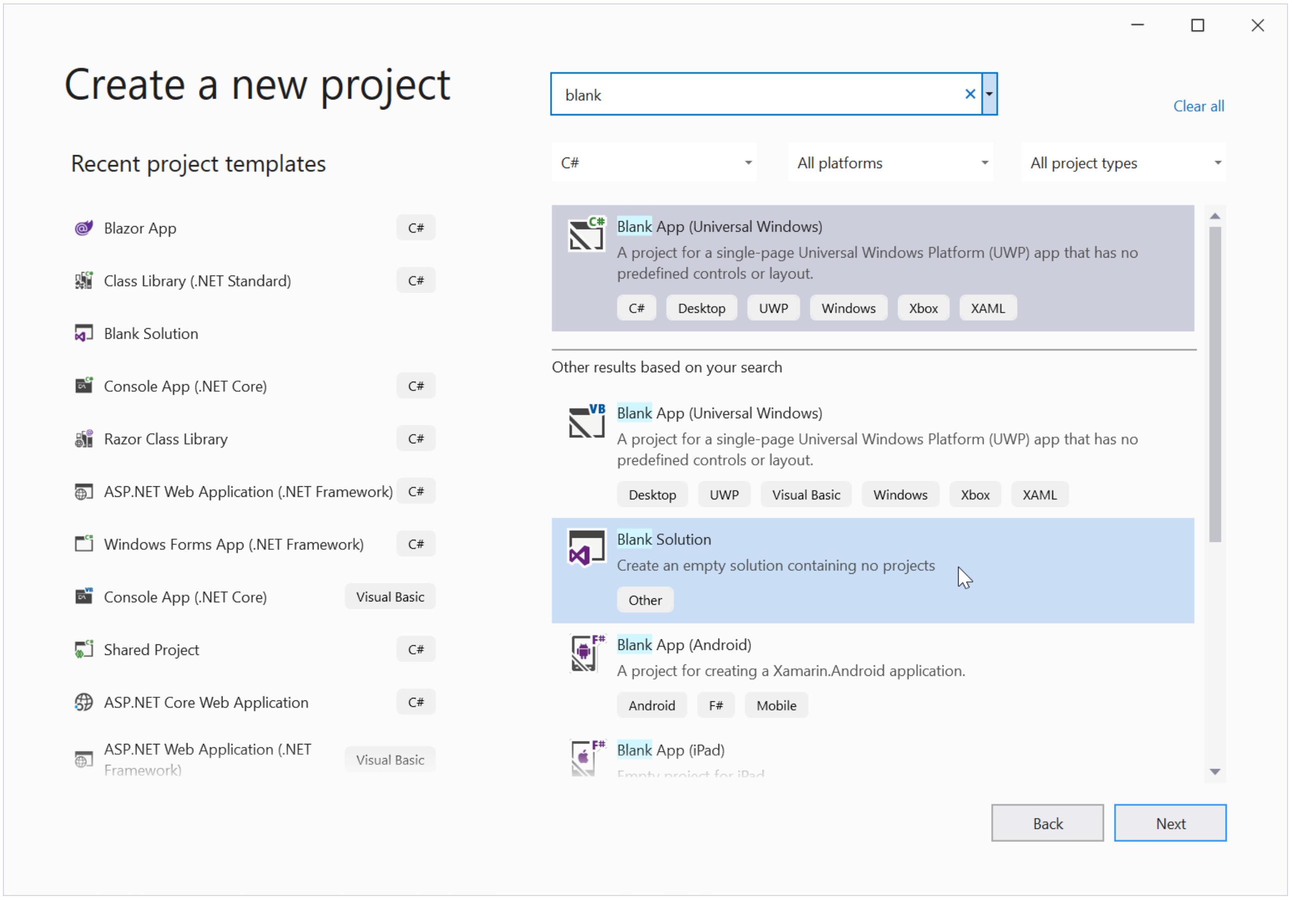
1. Add a Shared code project
2. Update the Server-side Blazor project to host a controller for use by the Client-side Blazor app
3. Add service abstraction for retrieving weather forecast data
4. Add client-side implementation to retrieve weather forecast data
5. Add server-side implementation to retrieve weather forecast data

> The code for this example is the `BlazorMultiHead` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

I will explain each step.

Create Blank Solution

Open Visual Studio and choose to create a new project. Search for and select the option to create a Blank Solution.



Create a new project

blank

Clear all

Recent project templates

Blazor App	C#
Class Library (.NET Standard)	C#
Blank Solution	
Console App (.NET Core)	C#
Razor Class Library	C#
ASP.NET Web Application (.NET Framework)	C#
Windows Forms App (.NET Framework)	C#
Console App (.NET Core)	Visual Basic
Shared Project	C#
ASP.NET Core Web Application	C#
ASP.NET Web Application (.NET Framework)	Visual Basic

C# | All platforms | All project types

Blank App (Universal Windows)
A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.
C# Desktop UWP Windows Xbox XAML

Other results based on your search

Blank App (Universal Windows)
A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.
Desktop UWP Visual Basic Windows Xbox XAML

Blank Solution
Create an empty solution containing no projects
Other

Blank App (Android)
A project for creating a Xamarin.Android application.
Android F# Mobile

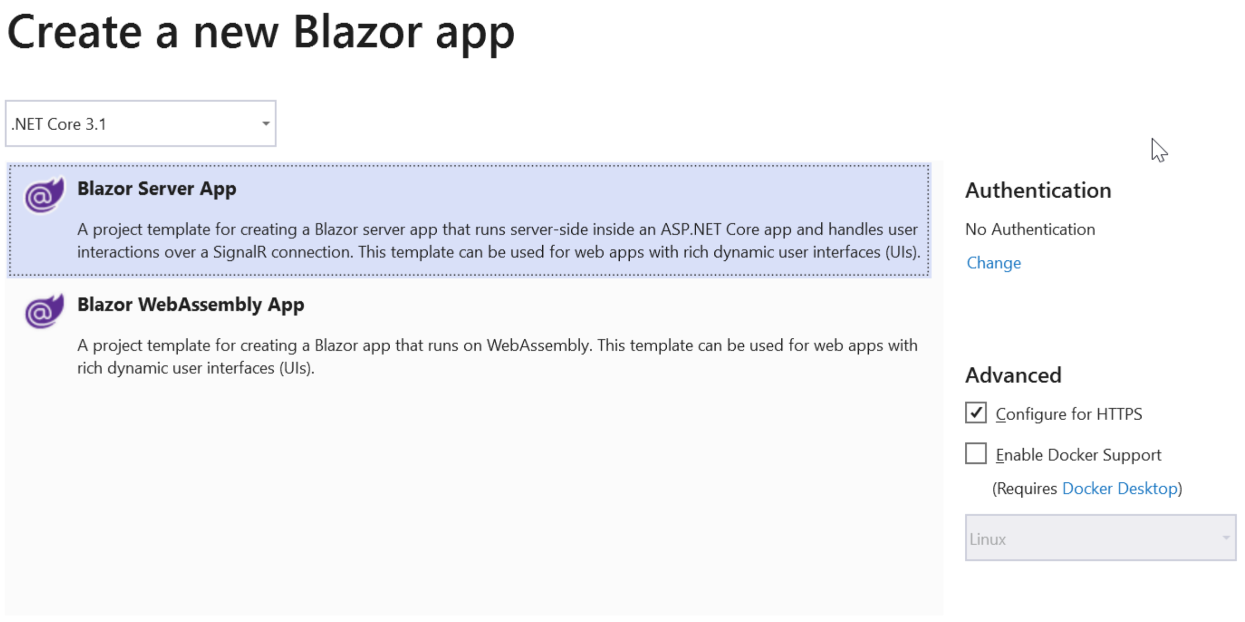
Blank App (iPad)
Empty project for iPad

Back Next

Name the new solution `BlazorMultiHead`.

Add Server-side Blazor Project

Add a new project to the solution named `BlazorMultiHead.Server` using the Blazor Server App template.


Create a new Blazor app

.NET Core 3.1

Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced

[✓] Configure for HTTPS

[ ] Enable Docker Support

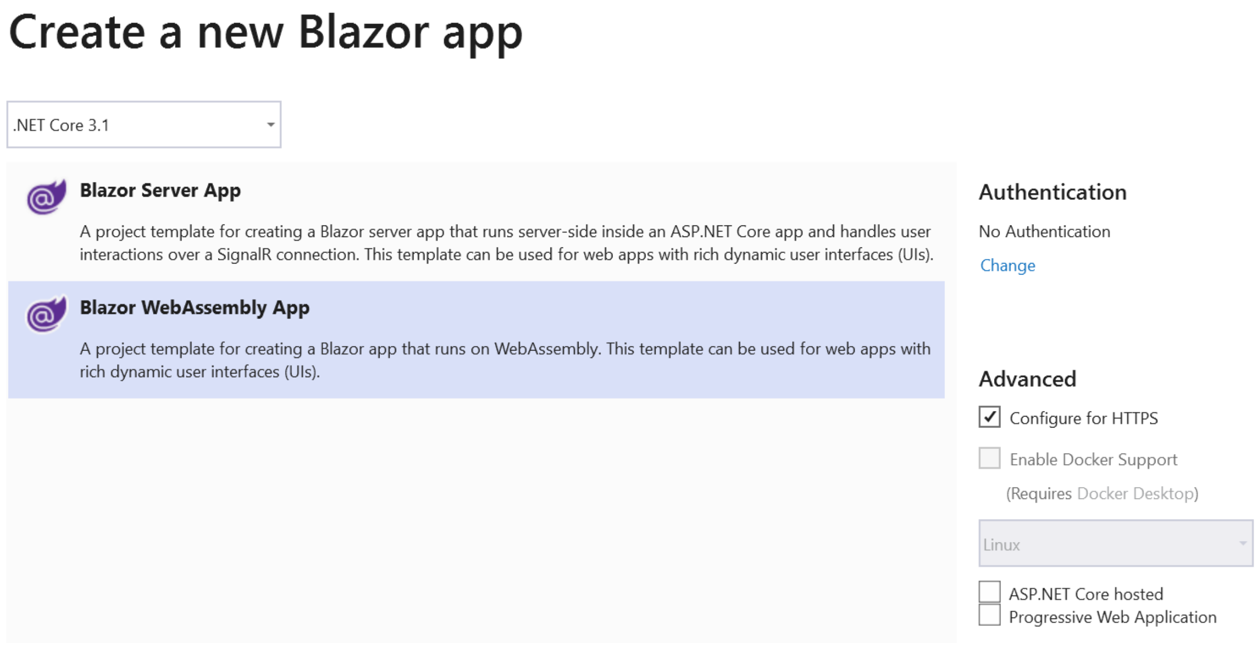
(Requires Docker Desktop)

Linux

This project, once updated, will act as the server-side Blazor head for the solution.

Add Client-side Blazor WebAssembly Solution

Add a new project to the solution named `BlazorMultiHead.Client` using the Blazor WebAssembly App template.


Create a new Blazor app

.NET Core 3.1

Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced

[✓] Configure for HTTPS

[ ] Enable Docker Support

(Requires Docker Desktop)

Linux

[ ] ASP.NET Core hosted

[ ] Progressive Web Application

This project, once updated, will act as the client-side Blazor head for the solution.

Add Shared UI Project

Now that you have the two "head" projects, it is time to add the project that will contain all common UI pages and components for both heads.

The steps include:

1. Add a shared UI project
2. Configure the project to support Blazor
3. Copy the Blazor UI elements to the new project

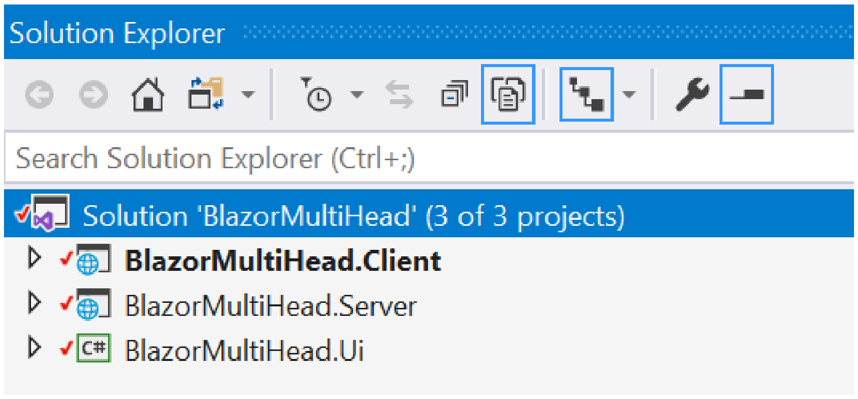
4. Remove the Blazor UI elements from the server-side and client-side projects

I am using a .NET Standard Class Library project in this example. You can also start with a Razor Class Library, but the simplicity of the .NET Standard Class Library template makes the process very straightforward.

Create Shared UI Project

Add a new .NET Standard Class Library project to the solution. Name the project `BlazorMultiHead.Ui`.

With this project added the solution now contains three projects.


Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'BlazorMultiHead' (3 of 3 projects)
- **BlazorMultiHead.Client**
- BlazorMultiHead.Server
- BlazorMultiHead.Ui

As I mentioned, the new `BlazorMultiHead.Ui` project isn't ready yet. Let's make a few changes so it can contain Blazor UI components.

The default project template includes a `Class1.cs` file. Use Solution Explorer to remove this file, leaving a totally empty project.

The `csproj` file needs some changes. Double-click on the project node in Solution Explorer to open the project file.

Replace the first line so it references the Razor SDK.

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

This is required for a class library to host Blazor UI components.

Edit the `PropertyGroup` element to match the following.

```
  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>
```

The `TargetFramework` element targets the `netstandard2.1` target. Most .NET Standard libraries should target `netstandard2.0`, but Blazor relies on features in the `netstandard2.1` target.

The `RazorLangVersion` element mirror the settings from the standard Blazor WebAssembly project template, and indicates the version of Razor to use.

Because this project targets `netstandard2.1` it can be referenced from other `netstandard2.1` projects such as the client-side Blazor UI project, and also from `netcoreapp3.1` projects such as the server-side Blazor UI project.

Finally, the project needs to reference some NuGet packages, so add an `ItemGroup` element.

```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components" Version="3.1.3" />
    <PackageReference Include="Microsoft.AspNetCore.Components.Web" Version="3.1.3" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient"
                      Version="3.2.0-preview3.20168.3" />
    <PackageReference Include="System.Net.Http.Json" Version="3.2.0-rc1.20217.1" />
  </ItemGroup>
```

These components are required when creating Blazor UI components.

With these changes the shared UI project is ready to host Blazor UI components. The `csproj` file should now look like this.

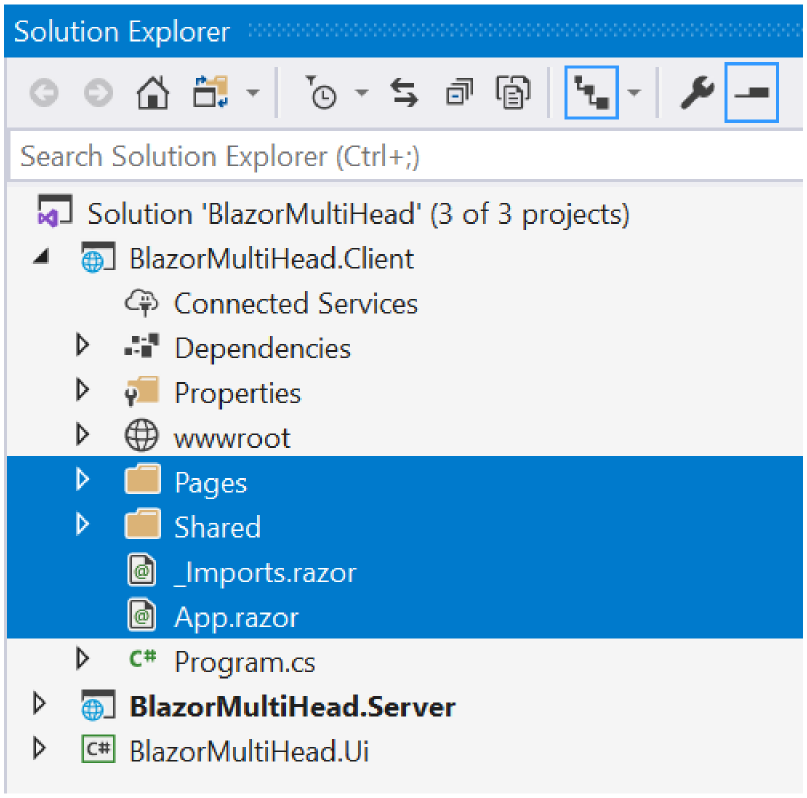
```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components" Version="3.1.3" />
    <PackageReference Include="Microsoft.AspNetCore.Components.Web" Version="3.1.3" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient"
                      Version="3.2.0-preview3.20168.3" />
    <PackageReference Include="System.Net.Http.Json" Version="3.2.0-rc1.20217.1" />
  </ItemGroup>

</Project>
```

The next step is to move pages and components from either of the existing Blazor app projects to this new shared UI project.

Move Pages and Components to Project

In this example I am choosing to copy the pages and components from the `BlazorMultiHead.Client` project to the new `BlazorMultiHead.Ui` shared project. You can copy the files from the server-side Blazor project too, the process is similar.

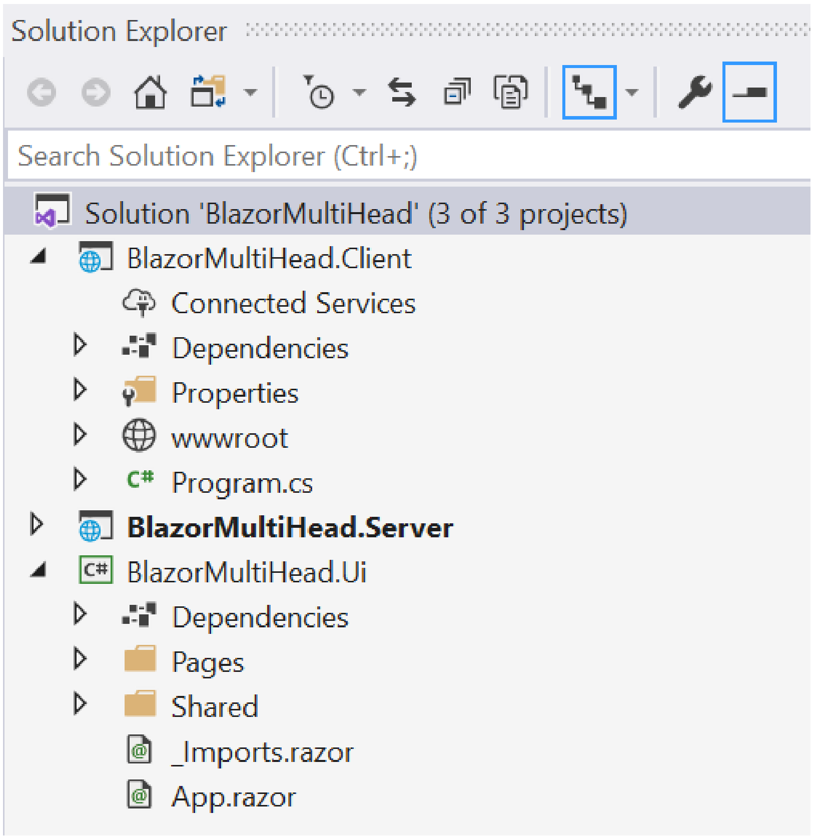
You can use Solution Explorer to drag-and-drop the content from the client project to the new shared UI project.



Copy the following:

- `Pages` folder
- `Shared` folder
- `_Imports.razor` file
- `App.razor` file

Once the files have been copied to the shared UI project you need to delete them from the original client project.



Solution 'BlazorMultiHead' (3 of 3 projects)
- BlazorMultiHead.Client
  - Connected Services
  - Dependencies
  - Properties
  - wwwroot
  - Program.cs
- **BlazorMultiHead.Server**
- BlazorMultiHead.Ui
  - Dependencies
  - Pages
  - Shared
  - _Imports.razor
  - App.razor

The project will not build yet, but it is close.

Update Moved Code

There are just a few changes you need to make to the code to change namespaces and now-missing types.

The `_Imports.razor` file uses namespaces from the old client project.

```
@using BlazorMultiHead.Client
@using BlazorMultiHead.Client.Shared
```

Update these to the new namespace.

```
@using BlazorMultiHead.Ui
@using BlazorMultiHead.Ui.Shared
```

The `_Imports.razor` file also references a WebAssembly namespace that isn't available in the shared UI project, so remove this line.

```
@using Microsoft.AspNetCore.Components.WebAssembly.Http
```

Open the `App.razor` page and find the reference to the `Program` type. This type is no longer available.

```
<Router AppAssembly="@typeof(Program).Assembly">
```

The `Program` type is being used to find the name of the assembly that contains the Blazor components. Any type that is guaranteed to exist, and which is in the new shared UI project will work. The obvious type that always exists is the `App` type itself, so change the code to use that type.

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

At this point the `BlazorMultiHead.Ui` project should build, though the `BlazorMultiHead.Client` project will not.

Reference Shared UI Project

At this point you have moved the Blazor UI components from the old client project to the new shared UI project. As a result the old client project has no UI components to display. Let's fix that.

Return to the `BlazorMultiHead.Client` project and add a reference to the `BlazorMultiHead.Ui` project.

> ℹ You can do this with the Add Project References dialog, or by editing the `csproj` file.

```
<ItemGroup>
  <ProjectReference Include="..\BlazorMultiHead.Ui\BlazorMultiHead.Ui.csproj" />
</ItemGroup>
```

The updated `BlazorMultiHead.Client` project file should now include an `ItemGroup` block similar to what I am showing here.

The client project will not build yet, because the `Program` class can't find the `App` type now that it is in a new namespace. Add a `using` statement at the top of the client project's `Program` class.

```
using BlazorMultiHead.Ui;
```

Now you should be able to successfully rebuild the solution.

Set the client-side Blazor project to be the startup project in Solution Explorer and run the solution. The client-side Blazor app should work as expected.

Update ASP.NET Core Project for Blazor

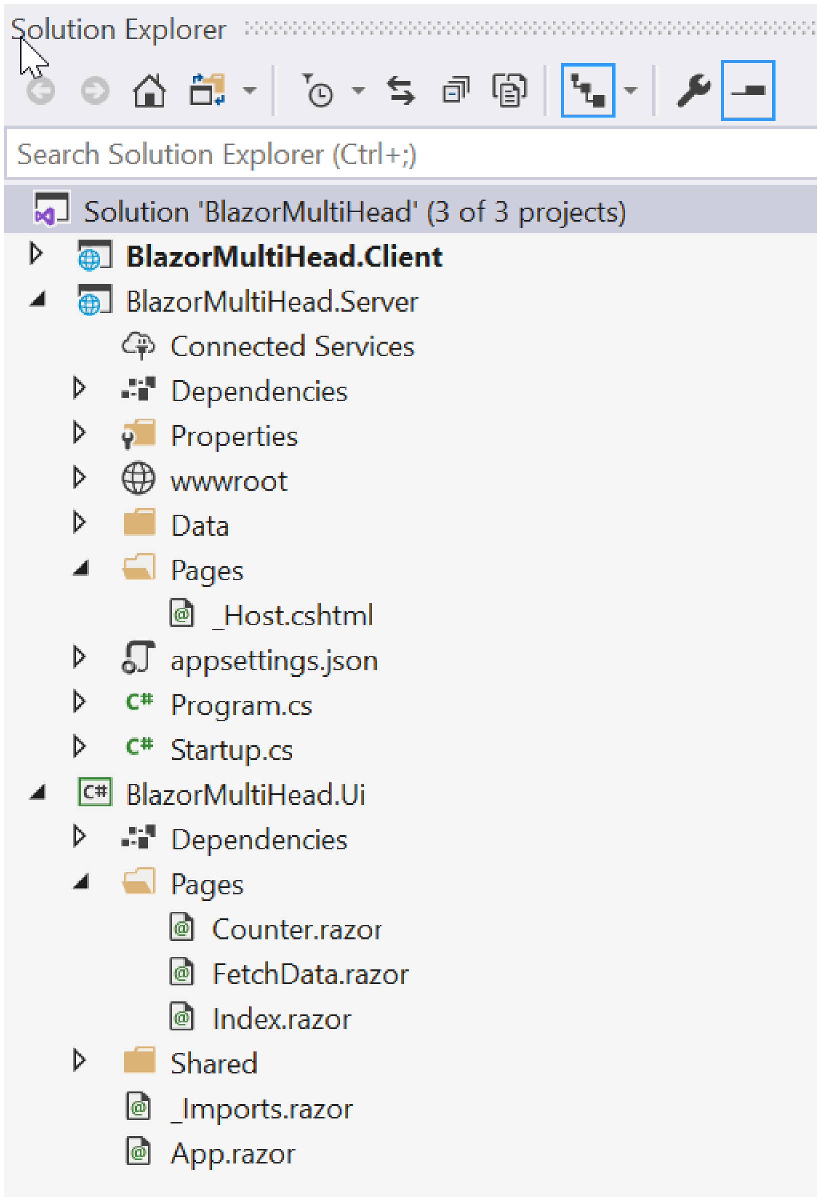
At this point you have a shared UI project, and the client-side Blazor head uses the pages and components in that shared project. The next step is to update the server-side Blazor project to do the same thing.

Remove Duplicate Pages and Components

The first step is to use Solution Explorer to remove all files in the `Pages` folder except for the `_Host.cshtml` file. This is the only file or type not already provided by the shared UI project.

Remove the entire `Shared` folder, as all these types are provided by the shared UI project.

Also remove the `_Imports.razor` and `App.razor` files from the project, as these files are in the shared UI project.



At this point the server-side Blazor project should only contain types unique to ASP.NET Core. It will rely on the shared UI project for all common UI types.

Reference the Shared UI Project

To make the shared UI types available to the server-side project, add a reference to the shared UI project from the server project. You can do this with the Add Project References dialog or by editing the project file.

When you are done the project file should contain an `ItemGroup` block similar to this.

```
<ItemGroup>
  <ProjectReference Include="..\BlazorMultiHead.Ui\BlazorMultiHead.Ui.csproj" />
</ItemGroup>
```

The project still won't build, because the `_Host.cshtml` file can't find the `App` type now that the type has moved to the shared UI project. Add a `@using` statement to `_Host.cshtml` to make this type available.

```
@using BlazorMultiHead.Ui
```

The solution should now build, and at this point you can set the server-side Blazor project to be the startup project in Solution Explorer and run the app.

The `Index` and `Counter` pages will work, but the `FetchData` page will fail. This is because the implementation of this page is from the client-side project, and won't work in the server-side project. I will discuss how to address this issue.

Per-Head Behaviors

Most of your UI pages and components will work identically regardless of whether they are hosted in a server-side or client-side Blazor app. Sometimes there do need to be differences between implementations.

The `FetchData` page is a good example, because the client-side app gets the data from a static file, and the server-side app generates random data.

> Xamarin.Forms developers are very familiar with this issue and the technique I am using in this chapter. It is very common for the Android and iOS heads of a mobile app to have different implementations for certain functionality.

This example is contrived, because you could choose to have a common implementation for both UI apps, but I'm using this as an example of how to provide different implementations.

The steps required to provide per-head implementations of a behavior are as follows.

1. Define any shared data types
2. Define a service interface
3. Use the shared types and interface in all shared UI code
4. Implement the service interface in each head project
5. Configure each head project for dependency injection

I will discuss each step.

Define Shared Data Types

Types that are shared between projects need to be defined in a common project. The existing `BlazorMultiHead.Ui` project is referenced by the server-side Blazor project, and so it can contain types needed by the UI and by the web server.

The type needed is the `WeatherForecast` type used by the `FetchData` page. Add a `Data` folder to the shared UI project, and add this `WeatherForecast` class in that folder.

```
using System;

namespace BlazorMultiHead.Ui.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; }
  }
}
```

This is the standard template code provided by the Blazor project templates. The class defines the data used by the `FetchData` page.

Define Service Interface

The client-side and server-side UI projects will provide different implementations of the code used to retrieve weather forecast data. The shared UI code won't know which implementation will be used at runtime, and so it relies on an interface type.

Add a `Services` folder in the shared UI project, and in that folder add a `IForecastService` definition.

```
using BlazorMultiHead.Ui.Data;
using System;
using System.Threading.Tasks;

namespace BlazorMultiHead.Ui.Services
{
  public interface IForecastService
  {
    Task<WeatherForecast[]> GetForecastAsync(DateTime startDate);
  }
}
```

This interface defines a `GetForecastAsync` method that can be called to get an array of `WeatherForecast` objects for use in the `FetchData` page.

Update the FetchData Page

The `FetchData` page in the shared UI project currently includes the original client-side implementation, and it needs to be updated to use this new `IForecastService` interface so its behavior can be different in the client-side and server-side Blazor deployments.

Find the line that injects the `HttpClient` instance.

```
@inject HttpClient Http
```

Replace that line with code to inject an instance of an object that implements the `IForecastService` interface.

```
@using Data
@using Services
@inject IForecastService ForecastService
```

Then replace all code in the `@code` block of the page to use the injected `ForecastService` object to get the data.

```
@code {
  private WeatherForecast[] forecasts;

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

The Razor markup in the page doesn't need to change, because it already displays data from an array of `WeatherForcast` objects.

At this point you have implemented a shared data type, an interface, and UI code that relies on that type and interface. All that remains is to implement the interface in the two app head projects.

Create Client-side Service Implementation

Add a `Services` folder to the `BlazorMultiHead.Client` project, and add a `ForecastService` class in that folder.

```
using BlazorMultiHead.Ui.Data;
using BlazorMultiHead.Ui.Services;
using Microsoft.AspNetCore.Components;
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class ForecastService : IForecastService
{
  private HttpClient Http;
  public ForecastService(HttpClient httpClient)
  {
    Http = httpClient;
  }

  public async Task<WeatherForecast[]> GetForecastAsync(DateTime time)
  {
    return await Http.GetJsonAsync<WeatherForecast[]>(
      "sample-data/weather.json");
  }
}
```

This class implements the `IForecastService` interface, and so implements the `GetForecastAsync` method. The implementation used here is from the standard Blazor WebAssembly project template, so it retrieves static data from a `sample-data/weather.json` file in the `wwwroot` directory of the client-side Blazor project.

The class relies on dependency injection to get an instance of the `HttpClient` object necessary to call the `GetJsonAsync` method.

Before this type can be used it needs to be registered for dependency injection as the client-side app starts up.

In the client project, edit the `Main` method in the `Program` class and register the `HttpClient` and `IForecastService` types.

```
      builder.Services.AddSingleton(new HttpClient {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
      builder.Services.AddSingleton<
        IForecastService, ForecastService>();
```

Now when the `FetchData` page requests an instance of `IForecastService` it will be provided with the `ForecastService` object implementation in the client-side Blazor project.

Create Server-side Service Implementation

The server-side project already has a service implementation to create random weather data. This is in the existing Data folder. Edit the `WeatherForecastService` class to implement the `IForecastService` interface.

```
  public class WeatherForecastService : IForecastService
```

You will need to add some `using` statements at the top of the file.

```
using BlazorMultiHead.Ui.Data;
using BlazorMultiHead.Ui.Services;
```

You can also delete the `WeatherForecast` class in the Data folder, because the code is now using the one from the shared UI project.

The server-side project needs to ensure that dependency injection relates the `WeatherForecastService` type with the `IForecastService` interface type. This is done in the `Startup` class. Edit the `ConfigureServices` method to change how the type is registered.

```
services.AddSingleton<IForecastService, ForecastService>();
```

Now when the `FetchData` page requests an instance of `IForecastService` it will be provided with the `ForecastService` object implementation in the server-side Blazor project.

You can use this technique to provide different implementations of a behavior on each Blazor app, while still having nearly all the code shared in the shared UI project.

Conclusion

Server-side Blazor provides superior debugging, and so is an attractive environment for app development. In many cases you will want to deploy to users using WebAssembly and client-side Blazor to take full advantage of the hardware and resources on client devices.

Being able to easily switch between server-side and client-side Blazor to run the exact same Blazor UI code enables easy debugging and flexible deployment. In the future Blazor may support other heads as well, so having your solution designed to support multiple heads is not only beneficial today, but also in the future.

The last two chapters of this book will focus on how CSLA .NET supports Blazor, and how you can use Blazor and CSLA .NET together to create powerful apps that leverage the best features of both frameworks.

Chapter 7: Blazor and CSLA .NET

CSLA .NET is an open source framework that provides a home for your business logic. Just like Blazor provides a first-class framework for creating a user interface and Entity Framework is a first-class framework for data access, CSLA .NET is a first-class framework for your business logic. You can learn about CSLA .NET from the Using CSLA: Introduction to CSLA .NET book.

Starting with version 5.2, CSLA .NET supports Blazor on both the server and client. In this chapter you will learn how to create server-side and client-side Blazor apps that leverage a reusable business logic layer created using CSLA .NET.

That same CSLA-based business layer can be used to create apps using Xamarin, Windows technologies, ASP.NET web sites, services, and more. The intent is that you create your business logic once, and it can be reused anywhere .NET can run.

Using CSLA .NET With Blazor

One of the core features of CSLA .NET is that business domain types created using CSLA automatically work well with all the UI frameworks supported by .NET. This means that when you create business types they'll support data binding and other common concepts from Windows Forms all the way to modern platforms like Blazor.

When you create a Blazor app with CSLA-based domain types, your business objects will automatically support data binding, along with enabling your app to easily take advantage of the validation, authorization, and other business rules encapsulated in your business layer.

Another important feature of CSLA .NET is how the data portal enables location transparency. This is the idea that the logical "server-side" code in your app might physically run on the client device, a web server, or a dedicated app server.

This is extremely valuable with Blazor, because a server-side Blazor deployment will probably run the logical "server-side" code directly in the web server that's hosting the Blazor app, but a client-side Blazor deployment of the same app will probably run the logical "server-side" code on an actual app server.

Because the CSLA data portal abstracts this entire concept, your code will be unchanged between server-side and client-side deployments, even though there may be configuration changes indicating that the logical "server-side" code will run in different locations.

I will discuss how the CSLA rules engine and data portal are used by Blazor apps in this chapter. To start with, let's discuss how to use and configure CSLA in your Blazor projects.

NuGet Packages

CSLA .NET provides "helper packages" for all supported UI frameworks, including Blazor, ASP.NET Core, UWP, WPF, and more. The Blazor NuGet package is named `Csla.Blazor`, and provides numerous types to help streamline the use of CSLA when building a Blazor project.

When building server-side Blazor projects it is also necessary to use the `Csla.AspNetCore` package, because server-side Blazor is hosted in ASP.NET Core.

Both of these packages depend on the core `Csla` NuGet package.

In a Blazor solution you may have the following project types, each with their own package dependency requirements.

1. Server-side Blazor project
   - Reference the `Csla.AspNetCore` package
   - Reference the `Csla.Blazor` package
2. Client-side Blazor project
   - Reference the `Csla.Blazor.WebAssembly` package
3. Shared Blazor UI (Razor Components) project
   - Reference the `Csla.Blazor` package
4. Business library project (containing only domain business types)
   - Reference the `Csla` package
5. Data access projects (implementing any data access logic)
   - Reference the `Csla` package

Notice that the business and data access projects do not reference any UI-specific helper packages. This is because your business domain types and data access types should be entirely platform neutral. Those types should work behind any UI framework supported by .NET.

Although the `Csla.AspNetCore` package is required for a server-side Blazor project, you won't directly use types from that package in your Blazor code. If you write any MVC or server-side Razor pages then those types are valuable. I discuss how CSLA .NET supports ASP.NET Core app development in the *Using CSLA: ASP.NET Core* book.

In this chapter I will focus on the types provided by the `Csla.Blazor` and `Csla.Blazor.WebAssembly` packages and how you can use them when building Blazor apps.

CSLA .NET Configuration

Most modern components, libraries, and frameworks require some configuration as the app starts up. This is true for CSLA .NET as well, and the configuration is slightly different for server-side and client-side Blazor.

Server-Side Blazor Configuration

Server-side Blazor projects rely on the ASP.NET Core configuration model where configuration occurs in the `Startup.cs` file.

Add a `using` statement to the top of the file.

```
using Csla.Configuration;
```

In the `ConfigureServices` method it is necessary to add the CSLA services for use in dependency injection. This is done by adding the following line of code after configuring things like MVC, Razor Pages, and Blazor.

```
  services.AddCsla().WithBlazorServerSupport();
```

In the `Configure` method it is necessary to configure the CSLA services and other configuration. Add this line of code after configuring MVC, Razor Page, and Blazor.

```
  app.UseCsla();
```

The `UseCsla` extension method has an overload you can optionally use to provide additional configuration for CSLA .NET. One of the most commmon scenarios is configuring the data portal to call an app server instead of running the logical server-side code on the web server.

```
  app.UseCsla((config) => config
    .DataPortal()
      .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy), "https://myserver/api/dataportal")
  );
```

Many other aspects of CSLA .NET can be configured as well, but most of the configuration should default correctly for use within your server-side Blazor app.

Client-Side Blazor Configuration

Client-side Blazor projects have configuration in the `Program` class.

Add a `using` statement to the top of the file.

```
using Csla.Configuration;
```

Configuration is performed in the `Main` method and relies on the `builder` variable created by the template code. Configure CSLA before the `RunAsync` method is invoked.

```
  builder.Services.AddOptions();
  builder.Services.AddAuthorizationCore();
  builder.UseCsla();
```

The `UseCsla` extension method adds all required services for use by dependency injection, and performs other configuration necessary for CSLA to work properly in a Blazor WebAssembly app.

CSLA .NET supports authorization, and it is necessary to call `AddOptions` and `AddAuthorizationCore` before using any Blazor authorization features. If you don't call these methods your app won't run because CSLA will be unable to initialize.

The method also has an overload you can optionally use to provide additional configuration for CSLA .NET. One of the most common scenarios is configuring the data portal to call an app server instead of running the logical server-side code on the web server.

```
  app.UseCsla((config) => config
    .DataPortal()
      .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy), "https://myserver/api/dataportal")
  );
```

Many other aspects of CSLA .NET can be configured as well, but most of the configuration should default correctly for use within your client-side Blazor app.

Data Portal and Data Access

The data portal is a key component of CSLA .NET, and a complete discussion of data portal configuration is in the *Using CSLA: Data Portal Configuration* book. In this chapter I will discuss only data portal configuration concepts that are unique to Blazor.

The data portal supports location transparency, allowing you to change your app between 1-tier, 2-tier, and n-tier physical deployments by changing *configuration* without affecting your code. When deploying Blazor apps there are four primary scenarios to consider.

1. Client-side Blazor With Local Data Portal
2. Client-side Blazor With App Server
3. Server-side Blazor With Local Data Portal
4. Server-side Blazor With App Server

The data portal defaults to "local mode", where the logical "server-side" code physically runs in the same location as your UI code. You can configure the data portal to run the logical "server-side" code on a server that is physically separate from where the UI code is running.

In this section of the chapter I use diagrams indicating the use of a ViewModel type from the MVVM design pattern. You *do not* need to use MVVM, and later in the chapter I will discuss alternatives. However, if you don't use MVVM then *something* will fill the role of the Interface Control layer in each diagram.

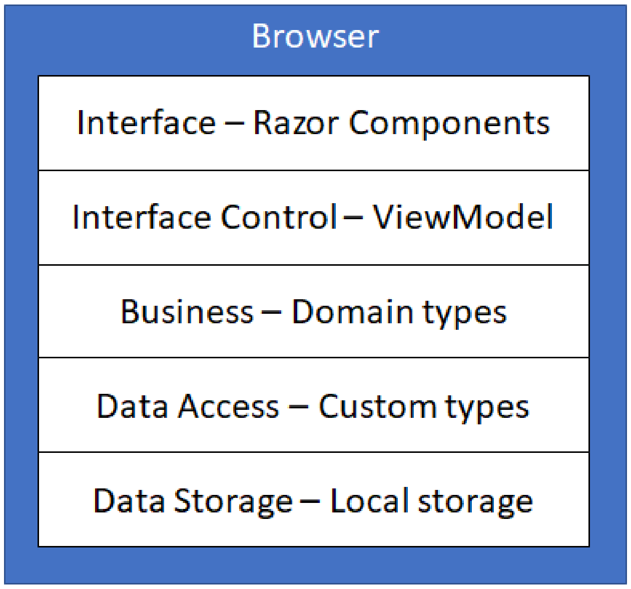
If the data portal is configured to communicate with a remote server, you are also able to choose the network transport protocol that should be used. The most common choice is to use HTTP, but you can also use gRPC, WCF, or other network communication technologies.

When working with client-side Blazor, only HTTP/HTTPS is supported as a network transport protocol.

Let's discuss each scenario, assuming the use of the *encapsulated invocation* data access model as discussed in the *Using CSLA: Data Access* book.

Client-side Blazor With Local Data Portal

A client-side Blazor app can store data locally on the client device.


Browser
Interface – Razor Components
Interface Control – ViewModel
Business – Domain types
Data Access – Custom types
Data Storage – Local storage

Normally in a client-side Blazor app data is stored using browser local storage to save data on the device that is running the browser. There are numerous NuGet packages available that provide ways to access browser local storage, and the use of those packages is outside the scope of this book.

The data portal abstracts the concept of domain business object persistence, allowing you to use whatever package or library you choose to interact with browser local storage.

The default data portal configuration is to run the logical server-side code locally, in the same location as your UI code. This means that the default is for your logical server-side code to run in the browser on the client device.

Because your logical server-side code will be running on the client device, this means the Data Access Layer (DAL) will be deployed to the client along with the rest of the app.

The DAL code will normally define an interface for persistence. To keep this example as simple as possible I will focus only on retrieving data.

```
public interface IPersonDal
{
  PersonInfo Get(int id);
}
```

The DAL will also implement this interface.

```
public class PersonDal : IPersonDal
{
  public PersonInfo Get(int id)
  {
    // get data from browser local storage
    // then put that data into a PersonInfo DTO for
    // transport to the business domain object
    return new PersonInfo { Id = id, Name = "Rocky" };
  }
}
```

This is where you must implement code to interact with browser local storage. Again, there are numerous NuGet packages available, and the use of those packages is outside the scope of this book.

Now that the DAL exists, the client-side Blazor app needs to be configured to use those types. This is done in the `Main` method of the `Program` class.

```
  public static async Task Main(string[] args)
  {
    var builder = WebAssemblyHostBuilder.CreateDefault(args);
    builder.RootComponents.Add<App>("app");

    builder.UseCsla();

    builder.Services.AddTransient<IPersonDal, PersonDal>();

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

You can see how the `AddTransient` method is used to tell dependency injection to return an instance of the `PersonDal` type any time the `IPersonDal` type is requested.

Finally, in the business library you will implement a `PersonEdit` domain type that makes use of the DAL.

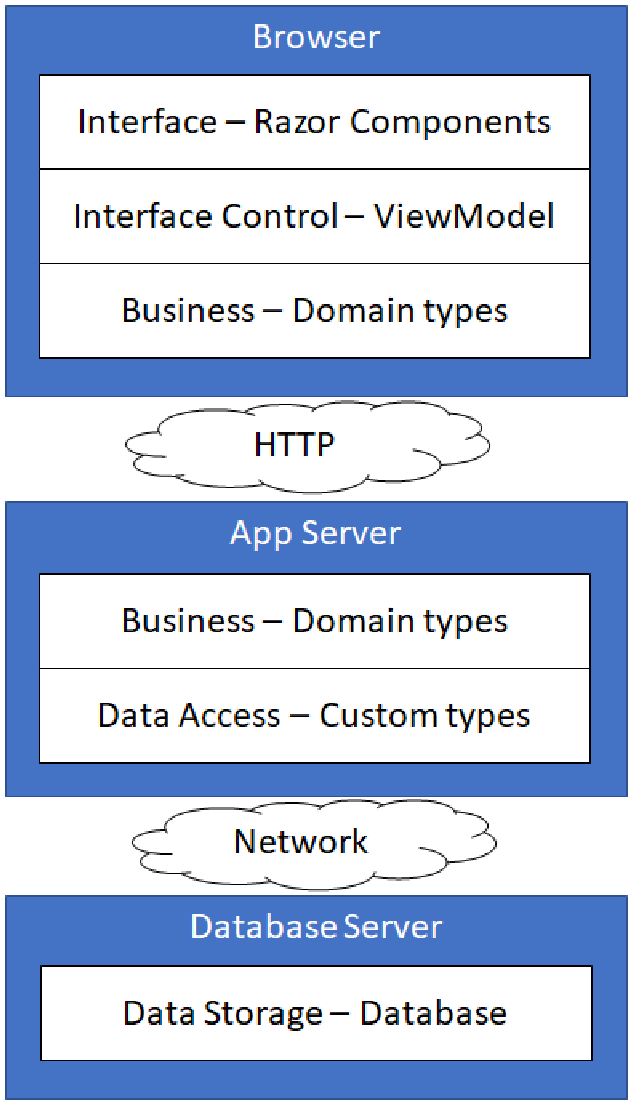
```
[Serializable]
public class PersonEdit : BusinessBase<PersonEdit>
{
  public static readonly PropertyInfo<int> IdProperty = RegisterProperty<int>(nameof(Id));
  public int Id
  {
    get => GetProperty(IdProperty);
    private set => LoadProperty(IdProperty, value);
  }

  public static readonly PropertyInfo<string> NameProperty = RegisterProperty<string>(nameof(Name));
  public string Name
  {
    get => GetProperty(NameProperty);
    set => SetProperty(NameProperty, value);
  }

  [Fetch]
  private void Fetch(int id, [Inject] IPersonDal dal)
  {
    var data = dal.Get(id);
    using (BypassPropertyChecks)
    {
      Id = data.Id;
      Name = data.Name;
    }
  }
}
```

Notice that the `Fetch` method relies on dependency injection to get an instance of the `IPersonDal` type, and then uses that object to get the persisted data so the domain object can be loaded with the information.

In this implementation, all the code runs on the client device, relying on the DAL code to persist any data locally on the device.

Client-side Blazor With App Server

Perhaps the most common deployment scenario will be a client-side Blazor app that interacts with an application server, and that app server interacts with the database. The app server might be the same physical web server used to deploy the Blazor WebAssembly app, or it might be a separate dedicated app server.



Either way, some configuration is needed on the client, and the app server must expose a service endpoint to host the data portal.

Implementing the Data Access Types

You should keep in mind that with this deployment model, the DAL assembly *does not* need to be deployed to the client device. The DAL assembly is only invoked on the app server, and should only be deployed to the app server.

Because the DAL code is running on an app server, it is very likely that the DAL implementation will be interacting with a database instead of some local file system. This will affect the implementation of your concrete DAL types.

In most cases you will use dependency injection to get access to your underlying data access object, such as an Entity Framework `ObjectContext` instance, and then you will use that type to interact with the database.

```
public class PersonDal : IPersonDal
{
  public ObjectContext ctx { get; set; }
  public PersonDal(ObjectContext context)
  {
      ctx = context;
  }

  public PersonInfo Get(int id)
  {
    var result = (from r in ctx.People
                  where r.Id == id
                  select new PersonInfo
                  {
                    Id = r.Id,
                    Name = r.Name
                  }).FirstOrDefault();
    if (result == null)
      throw new DataNotFoundException("Person");
    return result;
  }
}
```

Notice how the `PersonDal` type from earlier in the chapter has been enhanced to accept an EF type from dependency injection, and then to use that type when implementing the `Get` method. Also notice how the same `PersonInfo` DTO type is used to pass the data to the `Fetch` method in the business class. This means the business class is totally unaffected by the use of an EF-based DAL vs the earlier DAL implementation that relied on browser local storage.

HTTP Data Portal Channel

Before I discuss the specific configuration details for client and server, you should be aware that the only network transport protocol supported is the HTTP protocol via the `HttpProxy` in the `Csla.DataPortalClient` namespace.

Also, there is a limitation within the .NET (mono) runtime in WebAssembly that restricts data being passed over the .NET `HttpClient` object to be text only. No binary data is allowed. This restriction may be lifted at some point in the future, but today you need to configure the server-side data portal endpoint to understand that all data will be text-based, not binary.

Client-side App Configuration

In your client-side Blazor app it is necessary to configure the data portal so it uses an HTTP data portal channel to communicate with a remote app server. This is done in the `Main` method of the `Program` class, while configuring the app to use CSLA .NET.

```
app.UseCsla((config) => config
  .DataPortal()
    .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy), "https://myserver/api/dataportal")
);
```

In this example the `DefaultProxy` method is invoked to indicate that the `HttpProxy` type should be used to communicate with the app server, and a URL to the app server endpoint is provided.

The `UseCsla` method automatically configures the `HttpProxy` type to send and receive text-based data instead of binary data. The data transferred *is binary*, it is just base64 encoded into text so it can be passed over the network via the `HttpClient` object provided by Blazor.

App Server Configuration

You must also set up the app server with an endpoint and some configuration. In most cases the app server will be hosted by ASP.NET Core, whether this be the web server used to deploy the client-side Blazor app, or a dedicated app server.

The data portal endpoint is a controller in the `Controllers` directory of the ASP.NET Core project. Because this endpoint will send and receive text-based data instead of binary data, it must not be the same as the data portal endpoint controller used to support other types of client (such as Xarmarin, UWP, ASP.NET Core, etc).

```
[Route("api/[controller]")]
[ApiController]
public class DataPortalTextController : Csla.Server.Hosts.HttpPortalController
{
  public DataPortalTextController()
  {
    UseTextSerialization = true;
  }
}
```

A "regular" data portal endpoint is the exact same code as shown here, except that you don't set the `UseTextSerialization` property to `true`. By default the `HttpPortalController` base class sends and receives binary data, but when the `UseTextSerialization` property is set, it will send and receive base64 encoded binary data instead.

You must also configure the app server by adding code to the `Startup` class. In the `ConfigureServices` method it is necessary to configure the CSLA .NET services, and also to configure the web server to allow the use of synchronous methods.

```
      // If using Kestrel:
      services.Configure<KestrelServerOptions>(options =>
      {
        options.AllowSynchronousIO = true;
      });

      // If using IIS:
      services.Configure<IISServerOptions>(options =>
      {
        options.AllowSynchronousIO = true;
      });

      services.AddCsla().WithBlazorServerSupport();

      services.AddTransient<IPersonDal, PersonDal>();

      // also add EF types to services collection
```

The code that sets `AllowSynchronousIO` to `true` is required because the data portal currently uses some synchronous methods to serialize and deserialize data. By default, ASP.NET Core 3.0 or higher prevents the use of synchronous methods, and so it is necessary to enable their use.

The `AddCsla` method adds the necessary CSLA .NET types to the collection of services available for dependency injection, allowing the data portal to function on the app server.

Notice that the `IPersonDal` type is added so dependency injection can properly inject the correct DAL object for use in your business class's `Fetch` method.

Because the updated `PersonDal` type makes use of Entity Framework, you will also need to configure EF and add all relevant EF types to the services collection.

> There is no requirement by CSLA .NET to use EF. You can use raw ADO.NET, Dapper, EF, or any other technology you choose for interacting with the data store.

It is also necessary to add a line of code to the `Configure` method.

```
    app.UseCsla();
```

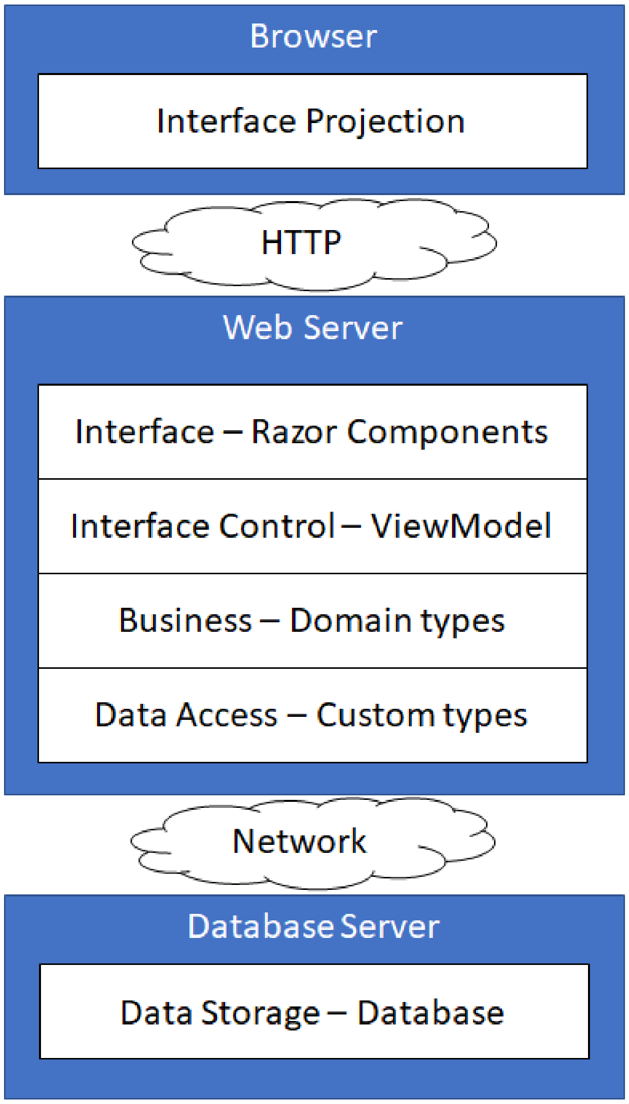
This performs other configuration steps necessary for CSLA .NET to function properly within an ASP.NET Core app.

At this point you should understand how to configure a client-side Blazor app for use with a local or remote data portal, and in the case of a remote data portal, you now know how to configure a data portal endpoint for use by Blazor apps.

Now let's discuss how to set up and configure the data portal for server-side Blazor apps.

Server-side Blazor With Local Data Portal

A server-side Blazor app runs entirely on the web server. When you use a local data portal configuration on a web server, your logical server-side code will run on the web server. This requires that the web server be able to directly interact with your database server.



The data portal defaults to running in "local mode", so no explicit configuration is required for the data portal to run locally. However, you do need to configure your DAL types on the web server for use by dependency injection.

Because the logical server-side code will be running on the web server, you need to deploy the DAL assembly to the web server, along with the Blazor app and the business library assembly.

Based on the `PersonEdit`, `PersonDal`, and `IPersonDal` types I discussed earlier in this chapter, you can configure the app for server-side Blazor deployment by altering the methods in the `Startup` class of the Blazor project.

In the `ConfigureServices` method you need to add the CSLA .NET types and the DAL type.

```
services.AddCsla().WithBlazorServerSupport();

services.AddTransient<IPersonDal, PersonDal>();

// also add EF types to services collection
```

The `AddCsla` method adds the necessary CSLA .NET types to the collection of services available for dependency injection, and the `IPersonDal` type is added so dependency injection can properly inject the correct DAL object for use in your business class's `Fetch` method. Also, the updated `PersonDal` type makes use of Entity Framework, you will also need to configure EF and add all relevant EF types to the services collection.

> There is no requirement by CSLA .NET to use EF. You can use raw ADO.NET, Dapper, EF, or any other technology you choose for interacting with the data store.

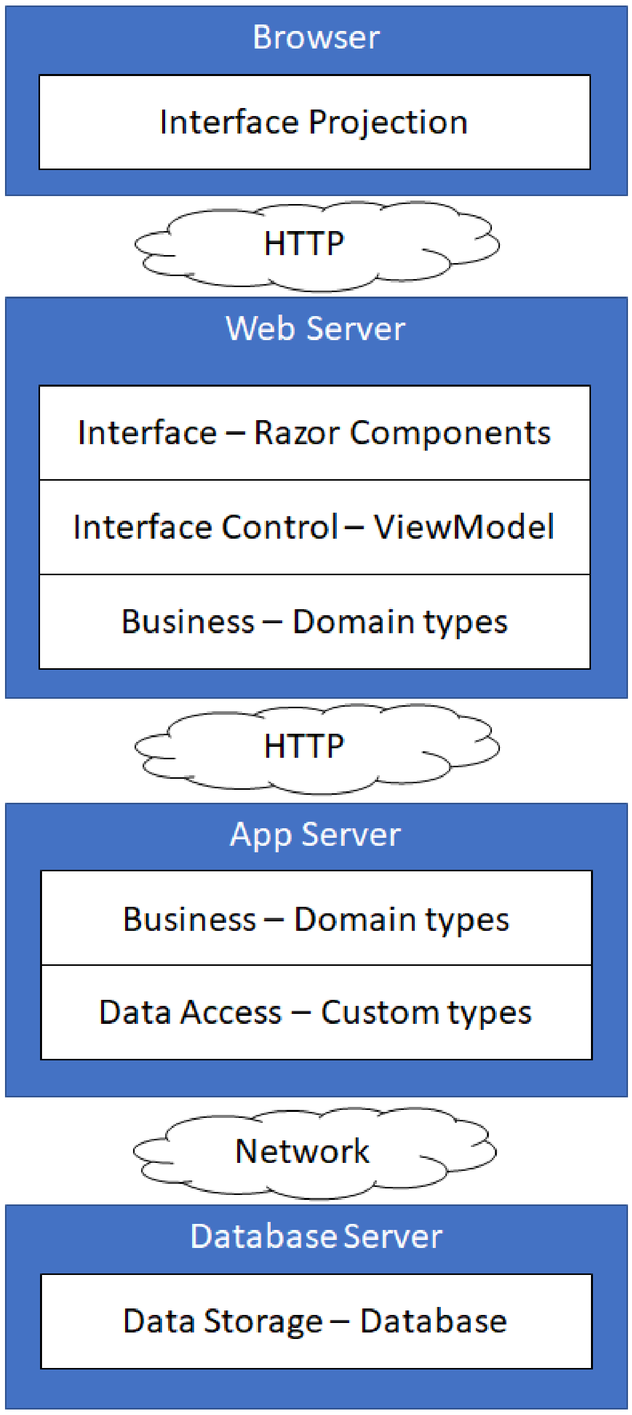
In the `Configure` method you need to indicate the use of CSLA .NET.

```
app.UseCsla();
```

At this point you should have a server-side Blazor app where the presentation, business, and data access layers are all deployed to, and run on, the web server.

Server-side Blazor With App Server

A less common deployment is where the Blazor app runs on the web server and communicates with an app server to manage data access.



If you need this type of deployment it is easy to set up by configuring the web server and app server.

Web Server Configuration

The web server runs the Blazor app, and so must contain your Blazor UI components and a reference to the assembly containing your business domain types. As with any server-side Blazor app that uses CSLA .NET, the `Startup` class needs some configuration code.

In this deployment the DAL assembly should not be deployed to the web server, as it is only used on the app server.

In the `ConfigureServices` method you must configure CSLA .NET.

```
services.AddCsla().WithBlazorServerSupport();
```

The `Configure` method needs not just the CSLA configuration, but also some data portal configuration to indicate that a remote app server should be used.

```
app.UseCsla((config) => config
  .DataPortal()
    .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy), "https://myserver/api/dataportal")
);
```

> This sample code shows using `HttpProxy`, but you can use any network transport that is supported by .NET Core, including `GrpcProxy` and other data portal channels that may be created in the future.

This configures the app to use CSLA .NET, and to run all logical server-side logic on the remote app server.

App Server Configuration

The app server needs a data portal endpoint and some configuration. The most common scenario is for the app server to be hosted by ASP.NET Core, so setting up a data portal endpoint is done by adding a controller to the `Controllers` directory in the web project.

```
[Route("api/[controller]")]
[ApiController]
public class DataPortalController : Csla.Server.Hosts.HttpPortalController
{  }
```

The `HttpPortalController` type completely abstracts the endpoint implementation, so you don't need to put any custom code here at all.

In the app server project's `Startup` class add the following to the `ConfigureServices` method.

```
    // If using Kestrel:
    services.Configure<KestrelServerOptions>(options =>
    {
      options.AllowSynchronousIO = true;
    });

    // If using IIS:
    services.Configure<IISServerOptions>(options =>
    {
      options.AllowSynchronousIO = true;
    });

    services.AddCsla().WithBlazorServerSupport();

    services.AddTransient<IPersonDal, PersonDal>();

    // also add EF types to services collection
```

This is the same configuration used earlier in this chapter for configuring the Blazor web server as an app server. The `AddCsla` method adds the CSLA types, a resolution for the `IPersonDal` type is added, and you need to add any necessary types for Entity Framework.

> There is no requirement by CSLA .NET to use EF. You can use raw ADO.NET, Dapper, EF, or any other technology you choose for interacting with the data store.

Then in the `Configure` method add a line of code.

```
app.UseCsla();
```

At this point you have configured the Blazor web server and an app server so they work together via the CSLA data portal so logical client-side work runs on the web server and logical server-side work runs on the app server.

You should now have an understanding of the four primary deployment models for Blazor apps supported by CSLA .NET and the data portal. Let's move on and discuss how business domain types created using CSLA .NET fit into a Blazor app architecture.

Creating Blazor Pages with CSLA .NET Objects

The Blazor UI framework supports data binding for read-only and read-write properties. When you create business domain types using CSLA .NET you will typically select base classes that support read-only or read-write object stereotypes.

I will discuss different Blazor UI architectures you can use when dealing with read-write data, but first I'll discuss the simpler read-only scenario.

Read-Only Values

When dealing with read-only properties, Blazor data binding is typically straightforward. For example.

```
<div>@person.Name</div>
```

The `@person.Name` expression will render the value of the `Name` property in the browser.

You should be aware that the CSLA rules engine supports authorization rules that run when a property getter is invoked. In other words, the current user may not be authorized to view the `Name` property.

At one level this is automatic, because the property getter will return the default value for the type if the current user isn't authorized to see the value. But you may want to provide a better experience, and later in this chapter I will discuss how you can use the CSLA authorization API to enhance the user experience to hide or disable UI elements if the current user isn't authorized to view or edit specific properties.

When you are creating data entry pages where data binding is used against read-write properties you have some choices. The Blazor UI framework supports two different ways to create a data entry form.

1. The Blazor `EditForm` component
2. The MVVM (model-view-viewmodel) design pattern

The `EditForm` component is a great way to create simple data entry forms. The `Csla.Blazor` namespaces includes helper types to integrate CSLA .NET rules into the UI experience.

The MVVM design pattern offers all the power provided by the CSLA .NET rules engine with the potential for a great deal of UI code reuse.

Although I will discuss both options in this chapter, my general recommendation is to use the MVVM design pattern, and that's what I'll use in Chapter 8.

> The code for this example is the `BlazorCslaExample` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Creating Pages with the EditForm Component

Blazor includes the `EditForm` component to help create simple data entry forms. I discussed this component in

Chapter 4.

At a basic level, CSLA-based domain objects work as any other object within the context of an `EditForm`. You don't need to do anything special to use a CSLA business object as the model for an `EditForm` component.

However, you will use the data portal to create, retrieve, and update the model, and the `Csla.Blazor` namespace includes helper components to allow the `EditForm` component to take full advantage of CSLA .NET business rules.

Getting the Model

In the page's `OnParametersSetAsync` method you will typically invoke the `CreateAsync` or `FetchAsync` data portal method to get an instance of the business class. You can see this code in the `BlazorCslaExample` project in the Pages folder in the `PersonEditForm` page.

```
protected override async Task OnParametersSetAsync()
{
  try
  {
    person = await Csla.DataPortal.FetchAsync<PersonEdit>(42);
    statusText = "Loaded";
  }
  catch (Exception ex)
  {
    statusText = $"ERROR: {ex.ToString()}";
  }
}
```

This code assumes that `person` and `statusText` fields have been declared in the `@code` block, similar to the coding approach used in Chapter 4.

Saving the Model

Saving changes when the form is submitted is a handled by your `OnSubmit` or `OnValidSubmit` event handler.

```
private async void FormSubmit(EditContext context)
{
  statusText = "Saving...";
  await person.SaveAndMergeAsync();
  statusText = "Saved";
  StateHasChanged();
}
```

The `SaveAndMergeAsync` method is asynchronous, so the `FormSubmit` method is marked with the `async` keyword, enabling the use of the `await` keyword.

However, there's a timing issue here that will cause Blazor data binding to miss any updates to the UI that occur after the `await` statement. The `FormSubmit` method is called synchronously by .NET, and the method will return *immediately* when the `SaveAndMergeAsync` method is invoked. It *will not wait* until the method completes.

The Blazor UI framework will update the UI based on any changes made when the `FormSubmit` method completes, and will be entirely unaware of any changes made when the `SaveAndMergeAsync` method completes, or after that line of code in the `FormSubmit` method.

To ensure that the UI is updated to reflect any changes to objects or fields (such as `statusText`) it is necessary to call the Blazor `StateHasChanged` method. This method tells the Blazor UI framework that state has changed, so it updates the UI display based on the new values.

Validation

You might recall from Chapter 4 that the `EditForm` component supports data annotations attributes. However, CSLA business rules can be much more complex than simple data annotations, and so the `Csla.Blazor` namespace includes helper types to enhance the display of broken rules to include those from CSLA .NET.

These components are direct replacements for their data annotation equivalents.

- `CslaValidator` should be used in place of `DataAnnontationValidator`
- `CslaValidationMessages` should be used in place of `ValidationMessages`

As a result, the markup within the `EditForm` component for editing a CSLA-based `PersonEdit` type looks like this.

```
<EditForm Model="@person" OnSubmit="@FormSubmit">
  <CslaValidator />
  <ValidationSummary />
  <InputText @bind-Value="person.Name" />
  <CslaValidationMessages For="() => person.Name" />
  <InputDate @bind-Value="person.Birthdate" />
  <div>@person.Name is @person.Age years old</div>
  <input type="submit" value="Save" class="btn btn-primary"
         disabled="@(!person.IsSavable)"  />
</EditForm>
```

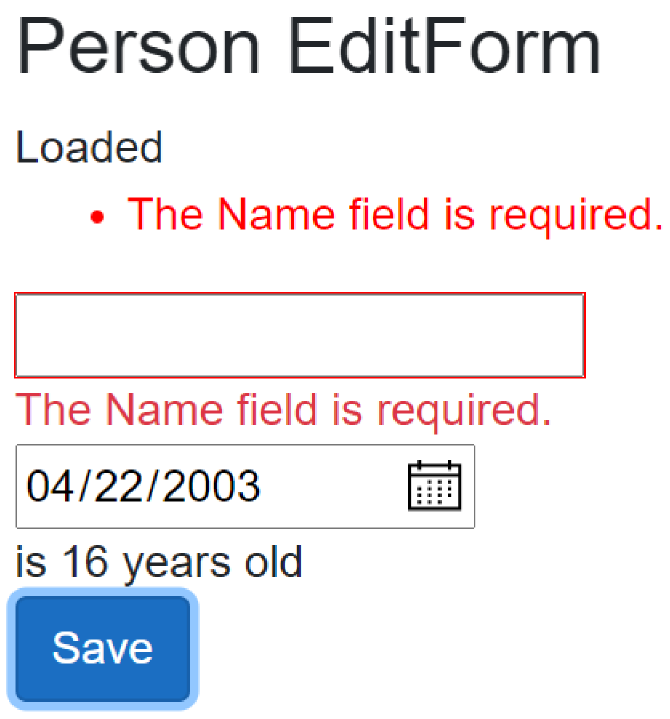
Notice the use of the `IsSavable` metastate property from the CSLA business object. This property is data bound to the submit button element's `disabled` property. As a result, the button is only available for the user to click when the underlying business object can be saved.

As a refresher, a CSLA object can only be saved when:

- The object has no broken validation rules (`Error` severity)
- Data in the object has been changed, or there's reason to believe the data doesn't match what is in the data store
- The current user is authorized to edit the object

The *Using CSLA: Creating Business Objects* book has full details on CSLA metastate properties. I am using the `IsSavable` property here to demonstrate how easy and powerful it can be to use data binding against those properties to manipulate the UI and provide a great user experience.

When this code is run and the form is submitted with a blank `Name` field, the result is that the validation messages are displayed.



Person EditForm

Loaded

- The Name field is required.

The Name field is required.

04/22/2003

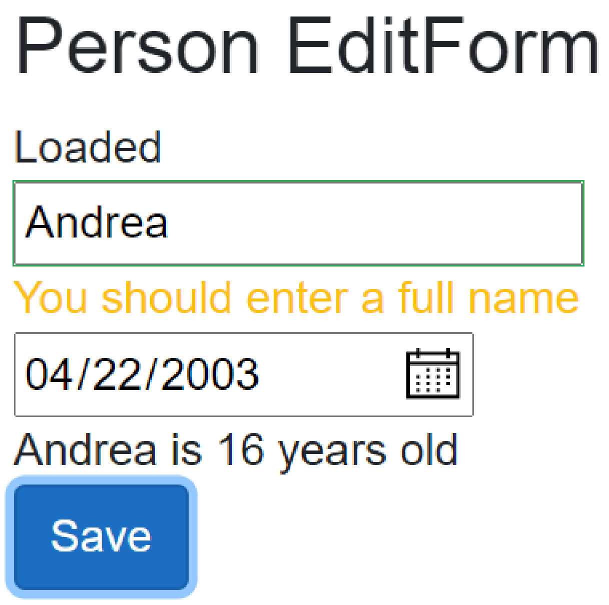
is 16 years old

Although this appears to be the same behavior as with the data annotations attributes in Chapter 4, it is important to understand that *all* CSLA validation rules are handled and their results displayed. That includes data annotations attributes, because CSLA honors those too, but it also includes any custom rules you've created within your code.

For example, the `PersonEdit` class in the code for this chapter includes a custom rule named `NoSingleName` that generates warning text if it appears that a full name was not entered.

```
public class NoSingleName : BusinessRule
{
  public NoSingleName(IPropertyInfo property)
    : base(property)
  {
    InputProperties.Add(property);
  }

  protected override void Execute(IRuleContext context)
  {
    var name = (string)context.InputPropertyValues[PrimaryProperty];
    if (!name.Trim().Contains(" "))
      context.AddWarningResult("You should enter a full name");
  }
}
```

The `CslaValidationMessages` component automatically displays this message.



Person EditForm

Loaded

Andrea

You should enter a full name

04/22/2003

Andrea is 16 years old

The standard `ValidationSummary` component only displays broken rules with an `Error` severity, and will not display `Warning` or `Information` severity rule messages.

EditContext Extensions

If you are writing code to implement validation by invoking methods on the `EditContext` parameter passed to the `OnSubmit`, `OnValidSubmit`, and `OnInvalidSubmit` event handlers from an `EditForm` component, the `Csla.Blazor` namespace provides the `AddCslaValidation` extension method.

The `AddCslaValidation` method is analogous to the `AddDataAnnotationsValidation` method, and is used to add CSLA validation to the context.

In most cases however, you will use the `CslaValidator` component as shown in this chapter.

At this point you should understand how the `EditForm` component works with CSLA-based domain types, and how you can use the helper types from the `Csla.Blazor` namespace to create data entry forms.

Creating Pages with MVVM

The model-view-viewmodel design pattern has been a mainstream concept for well over a decade at this point, and was pioneered with the rise of the XAML markup language used by the Windows Presentation Foundation (WPF), Universal Windows Platform (UWP) and now WinUI, and Xamarin.Forms.

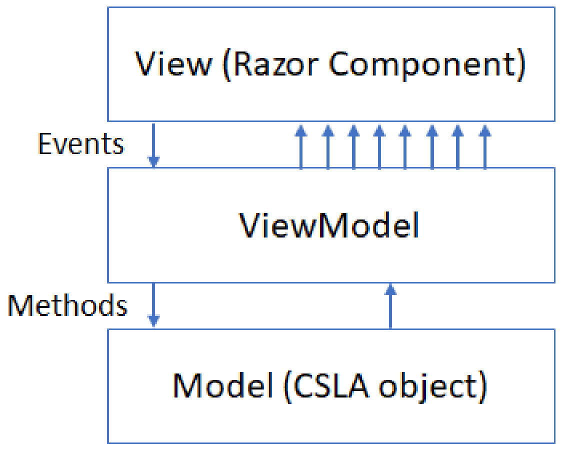
There are many justifications for the pattern, but there are two major reasons the pattern works so well with CSLA .NET and Blazor:

1. Code in a viewmodel class is more easily tested than code in or behind a Razor Component
2. The `Csla.Blazor` namespace provides a `ViewModel` type that helps abstract complexity when creating a Razor Component that binds to a CSLA-based domain business object

The basic idea of the MVVM design pattern is that there's an object between the view (the Blazor page) and the model (the CSLA business object) that can handle any events or interaction from the UI, reducing or eliminating the code in or behind the Blazor component.

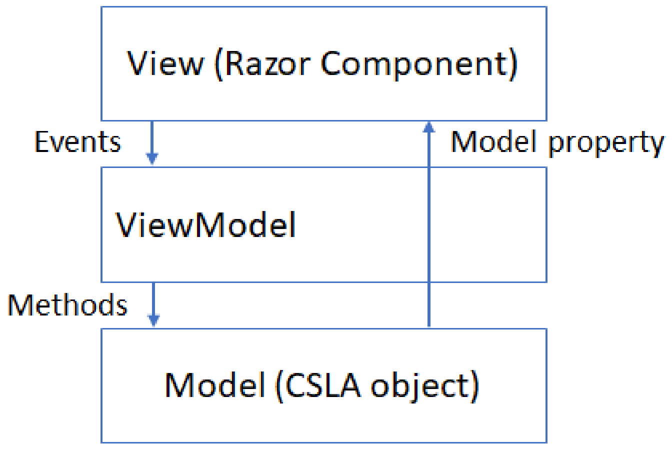
By putting nearly all code that responds to UI events into its own class, testability of your code is improved. It is possible to write unit tests for a viewmodel type much more easily than for a Razor Component.

In the simplest, and most brute force, MVVM implementations, the viewmodel class contains a lot of code, replicating all the properties of the underlying model object.



Although that approach can be useful if the model is poorly designed, when using CSLA .NET you should strive to design your domain business types based on good object-oriented design, focused on the business domain and user scenario. As a result, your domain types should be well designed and ready to bind directly to the view.

To this end, the `ViewModel` type provided by CSLA exposes a `Model` property so the view can bind directly to the business object without a lot of wasted coding effort.



In both cases the viewmodel object handles events raised by the view and implements behavior to properly handle those events. The `ViewModel` type provided by CSLA includes implementations for a number of standard behaviors, and you can add your own if necessary.

ViewModel Class

The `Csla.Blazor` namespace includes a `ViewModel` type that provides standard viewmodel behaviors necessary for most Blazor pages. This type is designed to work with dependency injection, and is automatically available to your code if you've configured CSLA in your project as discussed earlier in this chapter.

In a Razor Component you get access to a `ViewModel` instance by using the `@inject` directive. For example, in the sample project's Page folder the `PersonEditMvvm` page contains this code.

```
@page "/personeditmvvm"
@page "/personeditmvvm/{id}"
@using Data
@using Csla.Blazor
@inject ViewModel<PersonEdit> vm
```

The `@inject` statement ensures that a `ViewModel<PersonEdit>` instance is available to the component. This object will help manage the lifetime of the business object.

Initializing the Component

In the `@code` block for the page an override of the `Initialize` method is used to set up some basic event handling.

```
protected override void OnInitialized()
{
  vm.Saved += () => statusText = "Saved";
  vm.ModelPropertyChanged += async (s, e) => await InvokeAsync(() => StateHasChanged());
}
```

Handling the `Saved` event is optional, but in many cases you will find it useful to know when the user has submitted the form and the object has been saved. In this example I am updating some text on the page, but as you'll see in Chapter 8, this is also an ideal location to implement navigation to another page after saving the object.

The `ModelPropertyChanged` event must be handled as shown. When the event is raised the `StateHasChanged` method needs to be invoked. Background tasks might update the state of the business object so the `InvokeAsync` method is used to ensure that the `StateHasChanged` method is invoked in the Blazor UI context.

It is quite realistic for a CSLA-based domain object to be updated by background tasks. The data portal often runs in the background, as do any async rules you implement as part of your business logic. Handling the `ModelPropertyChanged` event ensures that the user's display always matches the properties of the underlying business domain object.

You can also handle the viewmodel's `ModelChanging` and `ModelChanged` events in the `Initialize` method. These events are raised before and after the `Model` property changes, allowing you to perform any necessary work in those scenarios.

Getting the Model Object

As always, the `OnParametersSetAsync` method is the event you should use to initialize the component's data. In this case that means using the `ViewModel` object to create or fetch the business object.

```
protected override async Task OnParametersSetAsync()
{
  if (id == 0)
    await vm.RefreshAsync(() => Csla.DataPortal.CreateAsync<PersonEdit>());
  else
    await vm.RefreshAsync(() => Csla.DataPortal.FetchAsync<PersonEdit>(id));
}
```

The `RefreshAsync` method requires that you provide a delegate (method) that returns a `Task<T>` where `T` is the business domain type. The async data portal methods, and any async static factory methods you implement return `Task<T>` as required.

You can see how this code checks the `id` parameter value passed via routing to the current page. If this value is `0` the `CreateAsync` method is called to create a new domain object. Otherwise `FetchAsync` is called to retrieve the existing object corresponding to the `id` parameter value.

The result is that the `Model` property of the viewmodel ends up referencing your `PersonEdit` business object.

It is possible for an exception to occur during the `RefreshAsync` method. In this case you will probably want to display that information to the user. This is easily done by binding a UI element to the `ViewModelErrorText` property of the `ViewModel` object.

```
<div class="text-danger">@vm.ViewModelErrorText</div>
```

For more detailed debugging information the `ViewModel` type also implements an `Exception` property that you can use in your code to access any `Exception` that caused the viewmodel to fail to get or save the model.

Saving the Model Object

The `ViewModel` type implements a `SaveAsync` method that saves the business object. It is intended for use via data binding from a save or submit button in the page. For example:

```
  <button @onclick="vm.SaveAsync" disabled="@(!vm.Model.IsSavable)">Save</button>
```

If the save operation is successful the `Saved` event is raised. Remember that this event is handled in the component's `Initialize` method.

It is possible for an exception to occur during the save process. As with the `RefreshAsync` method, any error information is available from the viewmodel's `ViewModelErrorText` property. Also, the `Exception` property references any `Exception` that caused the save operation to fail.

No other code is required, the rest of the Blazor page relies on data binding to interact with the model and viewmodel.

GetPropertyInfo Method

A CSLA domain object typically has properties, and each property has a set of metastate associated with that property. You can write code to directly interact with the low-level CSLA APIs to access that metastate, but CSLA provides helper functionality for all supported UI frameworks to simplify the process.

The `ViewModel` type implements a `GetPropertyInfo` method that gives you access to the value and metastate values for a property. In C# code you can call the method using any of these overloads:

```
  Csla.Blazor.PropertyInfo property = vm.GetPropertyInfo(() => vm.PropertyName);
  Csla.Blazor.PropertyInfo property = vm.GetPropertyInfo(nameof(vm.PropertyName));
  Csla.Blazor.PropertyInfo property = vm.GetPropertyInfo("PropertyName");
```

I recommend using the lambda expression technique, as it provides strong typing, the the most versatility of all options.

The `PropertyInfo` type from the `Csla.Blazor` namespace provides access to the property value and metastate, including:

- `Value` - read-write access to the property value
- `FriendlyName` - gets the friendly display name for the property
- `IsBusy` - returns `true` if an async rule is running for this property
- `ErrorText` - gets any `Error` severity broken rule messages
- `WarningText` - gets any `Warning` severity broken rule messages
- `InformationText` - gets any `Information` severity broken rule messages
- `CanRead` - returns `true` if the current user is authorized to read the property
- `CanWrite` - returns `true` if the current user is authorized to alter the property

In markup you can access the same information. For example, this code displays the `ErrorText` value for a `Name` property:

```
  <div class="alert-danger">@vm.GetPropertyInfo(() => vm.Name).ErrorText</div>
```

The `GetPropertyInfo` method and all the information available from the `PropertyInfo` type are designed to allow you to create reusable Blazor components based on your UI requirements. I will discuss how to create these components later in this chapter.

ViewModel and Authorization

As I discussed in Chapter 5, Blazor has component-level authorization features that are quite useful, if somewhat coarse-grained. The rules engine in CSLA provides for per-type and per-property authorization rules.

The `CanRead` and `CanWrite` properties of the `PropertyInfo` type allow you to alter the UI based on whether the current user is allowed to read or write to a specific business object property. I will show how those properties can be used to create reusable UI components later in this chapter.

At a per-type level you can use properties provided by the `ViewModel` type.

- `CanGetObject` - returns `true` if the user is allowed to fetch and view the model
- `CanEditObject` - returns `true` if the user is allowed to edit the model
- `CanDeleteObject` - returns `true` if the user is allowed to delete the model's data

You can use these properties to alter the UI so it is clear to the user what they are and are not allowed to do on any given page or component in your app.

At this point you should have a high level understanding of the features of the `ViewModel` type and how it supports building pages and components. I will now show how to create reusable UI components based on these features.

Building Reusable UI Components with ViewModel

One of the most powerful features of Blazor is how easy it is to create reusable UI components. I discussed this concept in Chapter 4, and now I will walk through how to create powerful UI components that make use of the CSLA metastate properties.

Building a Label Component

To start with, a data entry form usually displays some sort of label for each data entry field. This label value can come from the friendly display name maintained by CSLA for each business object property. These friendly names might be defined by an attribute or via code, but the UI doesn't really care, because that's all abstracted inside the business layer.

In the Shared folder of the `BlazorCslaExample` project there is a `LabelText` component.

```
@Property.FriendlyName

@code {
  [Parameter]
  public Csla.Blazor.IPropertyInfo Property { get; set; }
}
```

This component requires a `Property` parameter of type `IPropertyInfo`. I listed the properties available from the `IPropertyInfo` type earlier in this chapter, and you can see here how the `FriendlyName` property is used to display the friendly display name for the property.

The `PersonEditMvvm` page in the Pages folder uses this component to display a label.

```
  <div>
    <LabelText Property="vm.GetPropertyInfo(() => vm.Model.Id)" />
    is @vm.Model.Id
  </div>
```

This is preferable to hard-coding label names in each page or component, because CSLA supports concepts such as localization of the friendly name based on the user's current UI culture. Additionally this approach avoids repeating the label text through the app, making the app easier to maintain over time.

Building a Display Only Table Row Component

Many data entry forms are constructed using an HTML `table` so the labels and input fields line up in a professional manner. In such cases it is possible to compose UI components together. For example, in the Shared folder the `DisplayRow` component makes use of the `LabelText` component.

```
@if (Property.CanRead)
{
  <tr>
    <td><LabelText Property="@Property" /></td>
    <td>@Property.Value</td>
  </tr>
}

@code {
  [Parameter]
  public Csla.Blazor.IPropertyInfo Property { get; set; }
}
```

This layout assumes that the first table column is the label and the second column is the value to be displayed.

The component requires a parameter of type `IPropertyInfo` so it has access to the property value and metastate.

Notice how the `LabelText` component is reused to display the friendly name, passing the `Property` value through to the sub-component.

The `CanRead` metastate property is used to determine whether the current user is allowed to see this property value. If the user isn't authorized to see the value then no row is rendered into the table.

You might choose to display something different in the case that the user isn't authorized, but the benefit of having this `DisplayRow` component is that it provides a central location to change such UI behavior across the entire app.

The `DisplayRow` component is used in the `PersonEditMvvm` page.

```
    <DisplayRow Property="vm.GetPropertyInfo(() => vm.Model.Id)" />
```

Note that this component is used within an existing `table` element.

```
  <table>
    <thead>
      <tr>
        <th></th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      <DisplayRow Property="vm.GetPropertyInfo(() => vm.Model.Id)" />
    </tbody>
  </table>
```

This is a requirement because the `DisplayRow` component renders content that goes inside a `table`. If the component is used outside a `table` the result will be a runtime rendering error.

Building an Input Component

Most data forms support data entry. The `TextInput` component in the Shared folder implements basic text entry functionality.

```
<div>
  <input @bind-value="TextValue" @bind-value:event="oninput" type="@InputType"
         disabled="@(!Property.CanWrite)" /><br />
  <span class="text-danger">@Property.ErrorText</span>
  <span class="text-warning">@Property.WarningText</span>
  <span class="text-info">@Property.InformationText</span>
</div>

@code {
  [Parameter]
  public Csla.Blazor.IPropertyInfo Property { get; set; }
  [Parameter]
  public string InputType { get; set; } = "text";

  private string TextValue
  {
    get => (string)Property.Value;
    set => Property.Value = value;
  }
}
```

This component is a little more complex because it not only supports text input, but also CSLA validation messaging for all severities.

The text input is handled by rendering a standard `input` element.

```
  <input @bind-value="TextValue" @bind-value:event="oninput" type="@InputType"
         disabled="@(!Property.CanWrite)" /><br />
```

The `@bind-value` property binds the element's text to the text value of the business object property. A `TextValue` property is used within the component to cast the property value to a `string`, so properties of many types can be edited by the user.

The `@bind-value:event="oninput"` property ensures that data binding occurs on each keystroke, instead of the default behavior of binding when the user leaves the field in the browser. This provides for maximum interactivity in the user experience.

The `type="@InputType"` property binds the `input` element's `type` to a dynamic parameter value. The default value is `text`, but having this be a parameter to the component allows the UI developer to use this component for things like password input.

Finally, the `disabled` property is bound to the business object property's `CanWrite` metastate value. This ensures that the user will only be allowed to edit the value if they are authorized.

Next the three severities of validation messages are displayed, each with the appropriate style.

```
  <span class="text-danger">@Property.ErrorText</span>
  <span class="text-warning">@Property.WarningText</span>
  <span class="text-info">@Property.InformationText</span>
```

This component is not directly used in any page, but it is used to create the `TextInputRow` component.

Building a Table Row Component

Like the `DisplayRow` component discussed earlier, the `TextInputRow` component renders a `table` row that supports editing a text value.

```
@if (Property.CanRead)
{
  <tr>
    <td><LabelText Property="@Property" /></td>
    <td>
      <TextInput Property="@Property" InputType="@InputType" />
    </td>
  </tr>
}

@code {
  [Parameter]
  public Csla.Blazor.IPropertyInfo Property { get; set; }
  [Parameter]
  public string InputType { get; set; } = "text";
}
```

This component is not very complex, because much of the work has already been implemented in the `LabelText` and `TextInput` components.

This component is used in the `PersonEditMvvm` page.

```
      <TextInputRow Property="vm.GetPropertyInfo(() => vm.Model.Name)" />
```

Using the Components

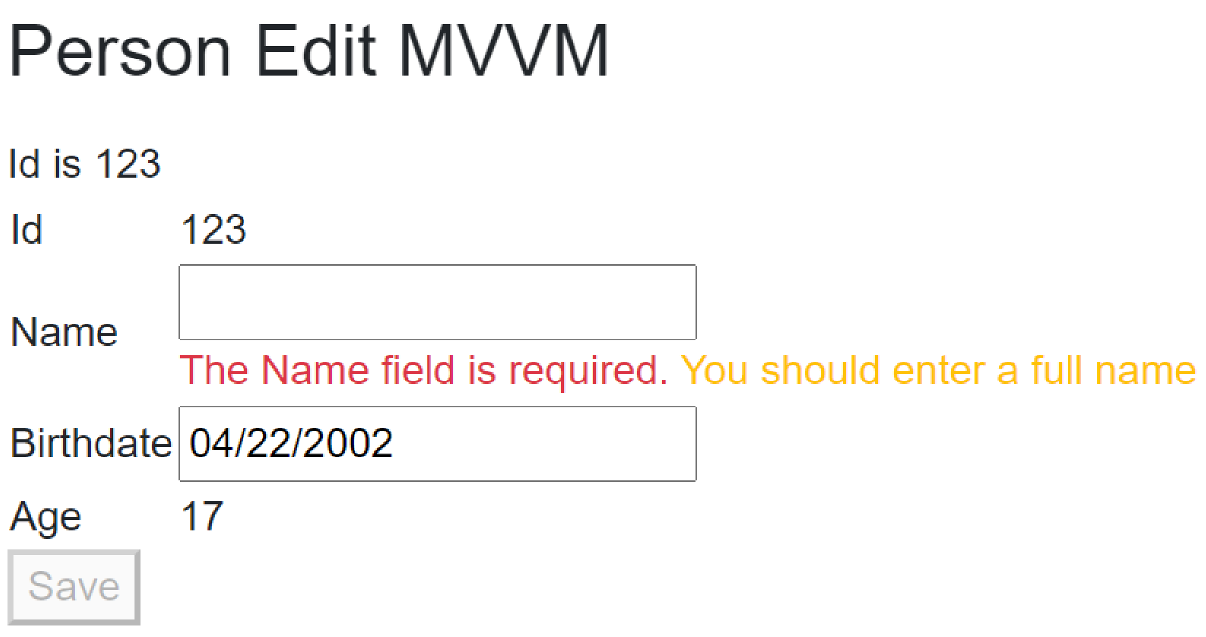
The sample project also includes `DateInput` and `DateInputRow` components. They follow the same pattern as the text input components. The resulting markup in the `PersonEditMvvm` page for the data entry table is this:

```
  <table>
    <thead>
      <tr>
        <th></th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      <DisplayRow Property="vm.GetPropertyInfo(() => vm.Model.Id)" />
      <TextInputRow Property="vm.GetPropertyInfo(() => vm.Model.Name)" />
      <DateInputRow Property="vm.GetPropertyInfo(() => vm.Model.Birthdate)" />
      <DisplayRow Property="vm.GetPropertyInfo(() => vm.Model.Age)" />
    </tbody>
  </table>
```

Running the app, navigate to the page with this URL:

```
http://localhost:59771/personeditmvvm/123
```

Then clear the `Name` field to trigger the rules.


Person Edit MVVM

Id is 123

Id 123

Name

The Name field is required. You should enter a full name

Birthdate 04/22/2002

Age 17

You can see how the validation messages are displayed, and how the save button is disabled because the model object is not currently valid.

At this point you should understand how to create reusable UI components based on the features of the `ViewModel` type. I will use this MVVM technique extensively in Chapter 8.

Next, it is important to understand how CSLA supports user authentication with Blazor.

Authentication

I discussed Blazor authentication in Chapter 5. However, if you are basing your app on a set of domain business types, your object model may also support authentication. CSLA .NET has always provided support for authentication, including extending user identity information to smart client platforms when necessary.

In the case of client-side Blazor it is necessary to extend the user identity to the client, otherwise it is not possible to run authorization business rules on the client device. In the case of server-side Blazor you may also need to extend the user's identity from the app server to the web server, though it is more likely that you will rely on the underlying operating system for such impersonation.

In Chapter 5 I talked about how server-side Blazor exclusively uses the `ClaimsPrincipal` and `ClaimsIdentity` types, and how client-side Blazor has no inherent concept of authentication.

Accessing User Identity in CSLA .NET

Over the years .NET has provided different APIs for accessing the current user identity, otherwise called a principal. One important goal of CSLA .NET is to provide a consistent coding pattern across all the .NET implementations and platforms, and so CSLA has always had an abstraction for accessing the user's identity.

```
var currentUser = Csla.ApplicationContext.User;
```

This `User` property adapts to the underlying .NET implementation and platform, providing a common way to get at the user identity regardless of whether your code is running in Windows Forms, ASP.NET Core, Blazor, Xamarin, or some other environment.

When writing apps using CSLA .NET you should always rely on this `User` property to gain access to the current user identity. This ensures that your code is portable across operating systems, .NET implementations, and platforms.

Server-side Blazor Authentication

When you build a server-side Blazor app, you should use the standard server-side authentication models discussed in Chapter 5. Those models all set the underlying ASP.NET Core user identity, and when CSLA code is running within ASP.NET Core the `User` property from `ApplicationContext` always returns that ASP.NET Core user identity.

In other words, for server-side Blazor you don't have to do anything special to enable the authorization rules in CSLA .NET, because they rely on the same user identity as ASP.NET Core and Blazor itself.

Client-side Blazor Authentication

Client-side Blazor is a little more complex, because client-side Blazor has no built-in authentication models. In Chapter 5 I discussed how you can implement authentication by delegating the responsibility to the web server, and then retrieving and maintaining the user principal on the client.

> ⚠ Any code running on a client device is more vulnerable to hacking than server-side code. You need to evaluate whether running any code on a client device fits within your risk profile. Keep in mind that the risk profile for client-side Blazor is the same as Angular, React, Windows Forms, WPF, UWP, Xamarin, and any other smart-client deployment target.

CSLA .NET makes client-side Blazor authentication easier thanks to built-in support for the concept. The data portal, when using the default `MobileFormatter` for serialization, is able to automatically serialize a copy of the server-side `ClaimsPrincipal` to the Blazor client running in the browser.

The basic flow of client-side Blazor authentication with CSLA is this:

1. Configure the app for authentication
2. Get the user's credentials from user input
3. Validate the user credentials using a business domain type
4. Use the user validation info to create a `ClaimsIdentity` instance
5. Create a `ClaimsPrincipal` instance using the new `ClaimsIdentity` object
6. Set the `Csla.ApplicationContext.User` property to the `ClaimsPrincipal` object

There are many variations on how you might implement this process, depending on the type of user credentials you are collecting, the type of identity class you've implemented, and how you are verifying the user credentials.

For example, you might be verifying the user credentials against a database, an LDAP server, an identity server, using Windows Active Directory, or Azure Active Directory.

In this chapter I will show a basic implementation of this process.

> The code for this example is the `BlazorCslaAuthentication` solution in GitHub: https://github.com/MarimerLLC/BlazorBook

Configure the App for Authentication

Chapter 5 covers the steps for configuring client-side Blazor to use authentication. When using CSLA some of the types are provided for you.

In a client-side Blazor app edit the `Program` class and add these lines to the `Main` method before calling the `UseCsla` method.

```
  builder.Services.AddOptions();
  builder.Services.AddAuthorizationCore();
  builder.Services.AddSingleton
    <AuthenticationStateProvider, CslaAuthenticationStateProvider>();
  builder.Services.AddSingleton<CslaUserService>();
```

The standard `AddOptions` and `AddAuthorizationCore` methods configure Blazor for authorization, and are required for CSLA .NET to properly initialize.

The next line adds a singleton object, so one instance for the lifetime of the app, so Blazor understands to use the `CslaAuthenticationStateProvider` to retrieve the current user identity when necessary. This type is in the `Csla.Blazor` namespace, and integrates with CSLA to properly manage the current user identity.

And finally, a singleton for `CslaUserService` is added, because that is required by the `CslaAuthenticationStateProvider` type. The `CslaUserService` instance may be used by pages within the app as well.

Make sure to add a line to the `_Imports.razor` component in the Shared folder as well.

```
@using Microsoft.AspNetCore.Components.Authorization
```

This brings the namespace into scope for use in Blazor components.

Finally, the content in `App.razor` needs to be wrapped with a `CascadingAuthenticationState` component, and the `RouteView` must be replaced with an `AuthorizeRouteView` as discussed in Chapter 5.

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

This completes the basic app configuration necessary to use authentication and authorization in a client-side Blazor app.

Implementing a Custom Identity Type

It is necessary to implement a custom identity type because this is where you will validate the credentials supplied by the user. There are many ways you might validate credentials, and in this example the code does no actual validation. In Chapter 8 you will see a similar class that validates the credentials against a database.

Look at the `CredentialValidator` class in the `BlazorCslaAuthentication.Shared` project. It is a standard CSLA read-only class that implements a number of properties necessary for creating a `ClaimsPrincipal` if the user credentials are valid.

- `Name` - validated user name
- `AuthenticationType` - type of validation used
- `Roles` - list of role names for the user

The interesting part of the class is the `Fetch` method. This is where the user credentials are validated and appropriate data is loaded into the `CredentialValidator` object.

```
    [Fetch]
    private void Fetch(UserCredentials credentials)
    {
      // validate credentials here
      if (!string.IsNullOrWhiteSpace(credentials.Username))
      {
        Name = credentials.Username;
        AuthenticationType = "Custom";
        Roles = new Csla.Core.MobileList<string>
        {
          "StandardUser",
          "PersonCreator"
        };
      }
    }
```

In a real application you will replace the `// validate credentials here` comment with code to actually validate the user credentials against your database, LDAP server, or other credential store.

If the credentials are valid then the object is loaded with appropriate values:

```
        Name = credentials.Username;
        AuthenticationType = "Custom";
        Roles = new Csla.Core.MobileList<string>
        {
          "StandardUser",
          "PersonCreator"
        };
```

When a `ClaimsIdentity` instance is loaded with data, it considers itself to represent an authenticated user only if the `AuthenticationType` property contains a text value. I will show the code to create a `ClaimsIdentity` instance based on the results of the `CredentialValidator` in the next section of this chapter.

Implementing a Login Page

A login page is typically just a way for the user to enter their username and password, or other credentials. As such, this is just another type of data entry form that can be built using a domain type.

Before creating the page itself, a domain type to collect the credentials is required. In the sample app's Data folder this is the `UserCredentials` class:

```
  [Serializable]
  public class UserCredentials : BusinessBase<UserCredentials>
  {
    public static readonly PropertyInfo<string> UsernameProperty = RegisterProperty<string>
(nameof(Username));
    [Required]
    public string Username
    {
      get => GetProperty(UsernameProperty);
      set => SetProperty(UsernameProperty, value);
    }

    public static readonly PropertyInfo<string> PasswordProperty = RegisterProperty<string>
(nameof(Password));
    [Required]
    public string Password
    {
      get => GetProperty(PasswordProperty);
      set => SetProperty(PasswordProperty, value);
    }

    [Create]
    [RunLocal]
    private void Create()
    { }
  }
```

This is a standard business class with two required `string` properties. Because this type will only be used to bind to a form, it'll never be saved, so there's no need for fetch or update data portal methods.

In the Pages folder, the `Login` page uses the MVVM pattern I discussed earlier in this chapter to provide the user with an edit experience over the `UserCredentials` type. I will discuss only the parts of this page that are unique.

This includes reuse of the `TextInputRow` UI component. Notice how the password field has an input type of `password`.

```
  <TextInputRow Property="vm.GetPropertyInfo(nameof(vm.Model.Password))"
                InputType="password" />
```

The page also relies on `NavigationManager` and `CslaUserService` instances, which it gets via injection.

```
@inject CslaUserService userService
@inject NavigationManager nav
```

The biggest difference from what I discussed earlier in the chapter is that the submit button doesn't ask the viewmodel object to save the model, and instead it raises an event to a standard event handler method.

```
  <button @onclick="VerifyCredentials" disabled="@(!vm.Model.IsSavable)">Save</button>
```

The `onclick` event is handled by a `VerifyCredentials` method that does the login operation.

```
  private async void VerifyCredentials()
  {
    var userInfo = await DataPortal.FetchAsync<CredentialValidator>(vm.Model);
    var identity = new ClaimsIdentity(userInfo.AuthenticationType);
    identity.AddClaim(new Claim(ClaimTypes.Name, userInfo.Name));
    if (userInfo.Roles != null)
      foreach (var item in userInfo.Roles)
        identity.AddClaim(new Claim(ClaimTypes.Role, item));
    var principal = new System.Security.Claims.ClaimsPrincipal(identity);
    userService.CurrentUser = principal;
    StateHasChanged();
    nav.NavigateTo("/");
  }
```

The data portal is used to fetch a `CredentialValidator` instance. As I discussed earlier in this chapter, the resulting object will contain information about whether the credentials are or are not valid.

Modern .NET, and Blazor, are designed with the assumption that the user identity is a `ClaimsPrincipal` containing a `ClaimsIdentity`. The following code uses the properties of the `CredentialValidator` to create a new `ClaimsIdentity` and `ClaimsPrincipal` to represent the current user.

```
    var identity = new ClaimsIdentity(userInfo.AuthenticationType);
    identity.AddClaim(new Claim(ClaimTypes.Name, userInfo.Name));
    if (userInfo.Roles != null)
      foreach (var item in userInfo.Roles)
        identity.AddClaim(new Claim(ClaimTypes.Role, item));
    var principal = new System.Security.Claims.ClaimsPrincipal(identity);
```

It is important to understand that if the `AuthenticationType` property is empty or null, the `ClaimsPrincipal` will represent an unauthenticated user. If the `AuthenticationType` property contains a text value the identity represents an authenticated user.

Because the data portal is used to asynchronously validate the credentials on the server, the `StateHasChanged` method must be called so Blazor realizes it needs to update the UI.

Finally, the Blazor navigation manager is used to navigate to the root page for the app.

Implementing a Logout Page

The `Logout` page, in the Pages folder, is similar to the `Login` page.

```
@page "/logout"
@using System.Security.Claims
@using Csla.Blazor.Client.Authentication
@inject CslaUserService userService
@inject NavigationManager nav

<h3>Logout</h3>

Are you sure you want to log out?

<button @onclick="LogoutUser" class="btn btn-primary">Logout</button>

@code {
  private void LogoutUser()
  {
    userService.CurrentUser =
      new ClaimsPrincipal(new ClaimsIdentity());
    nav.NavigateTo("/");
  }
}
```

This implementation acts as a confirmation page. You can also create this page without any visible UI elements and just log out the user as the page loads.

```
@page "/logout"
@using System.Security.Claims
@using Csla.Blazor.Client.Authentication
@inject CslaUserService userService
@inject NavigationManager nav

@code {
  protected override void Initialize()
  {
    userService.CurrentUser =
      new ClaimsPrincipal(new ClaimsIdentity());
    nav.NavigateTo("/");
  }
}
```

Either way, the page sets the `CurrentUser` property of the `CslaUserService` object to an unauthenticated `ClaimsPrincipal` and navigates to the home page for the app.

Because this code has no async server calls, it is not necessary to call `StateHasChanged` to update the UI.

At this point you should understand how CSLA .NET integrates with server-side and client-side Blazor to enable authentication of a user. Once a user has been authenticated it is possible to implement authorization rules to control what the user can and can not do within your app.

Authorization

Authorization relies on the user being authenticated, so there is a `ClaimsPrincipal` available from `Csla.ApplicationContext.User` on server or client. This enables the use of the helper types in the `Csla.Blazor` namespace.

Authorization is generally per-component. If you want more fine-grained authorization you should use the features from the `Csla.Blazor.ViewModel` type or the `Csla.Rules` namespace.

I discussed the per-component Blazor authorization functionality in Chapter 5. That behavior relies on the user being authenticated, and is unaffected by whether you use your own custom authentication or the CSLA-based authentication approach discussed in this chapter.

If you want more fine-grained authorization behaviors that rely on the business rules in your CSLA-based domain business layer you should use the helper types from the `Csla.Blazor` namespace. These include:

- `CslaPermissionsHandler` - Blazor permissions handler automatically configured by CSLA
- `CslaPermissionPolicy` - Blazor permission policy automatically configured by CSLA
- `CslaPermissionRequirement` - Permission definition used by `CslaPolicy` and `HasPermission`
- `CslaPolicy` - Blazor authorization policy used to check per-type CSLA authorization rules
- `HasPermission` - attribute used on a Blazor component to check per-type CSLA authorization rules

Of these types, you will directly use `CslaPolicy` and `HasPermission`.

Understanding Blazor Authorization

Before discussing these types it is important to understand how Blazor implements authorization rules based on policies.

A policy is just a string that names an authorization policy. When you use the standard `AuthorizeView` component in Blazor, one option is to provide a policy name.

```
  <AuthorizeView Policy="MyPolicy">
```

The Blazor authorization system relies on a list of policy handlers to enforce policies. The authorization subsystem loops through the list of handlers, asking each if it can handle the requested policy. If a handler can handle the policy, it provides a `true`/`false` answer.

CslaPolicy

CSLA authorization rules are relatively complex, at least compared to the simple string-based model supported by Blazor. To overcome this, CSLA includes helper code to encode authorization policies into a string value that works with the Blazor authentication model.

A per-type authorization rule in CSLA has action and target elements. The action element indicates whether the current user can:

- Create an instance of the target
- Get/view the target
- Edit/save the target
- Delete the target

The target is typically a business domain type.

The `CslaPolicy` type exposes a `GetPolicy` method that accepts the action and target and returns a string value that encodes both values.

```
string policyName = CslaPolicy.GetPolicy(
                      AuthorizationActions.CreateObject,
                      typeof(PersonEdit));
```

This string value is then used as a policy name anywhere the Blazor authorization subsystem requires such a name.

The `CslaPermissionsHandler` type, automatically configured by CSLA, handles these encoded policies, providing the appropriate response based on the specified action and target. It does this by invoking the CSLA API, asking the business layer whether the current user is authorized to perform the requested action on the indicated target.

The `BlazorCslaAuthentication` sample demonstrates this functionality. In the `Index` page the following markup uses `CslaPolicy` to create a policy name.

```
<AuthorizeView
  Policy="@(CslaPolicy.GetPolicy(AuthorizationActions.CreateObject,
                                typeof(PersonEdit)))">
  <Authorized>
    User can create a person object
  </Authorized>
  <NotAuthorized>
    User can NOT create a person object
  </NotAuthorized>
</AuthorizeView>
```

The project includes a `PersonEdit` class that defines per-type rules for the `PersonEdit` class.

```
  [ObjectAuthorizationRules]
  public static void PerTypeRules()
  {
    Csla.Rules.BusinessRules.AddRule(
      typeof(PersonEdit),
      new Csla.Rules.CommonRules.IsInRole(
        Csla.Rules.AuthorizationActions.CreateObject,
        "PersonCreator"));
  }
```

This is standard CSLA business layer code, indicating that only users who are members of the "PersonCreator" role are allowed to create new instances of the `PersonEdit` type.

You can use the `AuthorizeView` element to display different UI content to the user based on whether the user is allowed to create, view, edit, or delete business domain types.

HasPermission

The `HasPermission` attribute operates at a component level. Adding this attribute to the top of a component uses your business domain layer's authorization rules to grant or deny access to the component as a whole.

```
@attribute [HasPermission(AuthorizationActions.CreateObject, typeof(PersonEdit))]
```

This is comparable to the default Blazor behavior, but doesn't require hard-coding the roles or authorization policies into your UI layers. Instead this approach takes advantage of the rules already encoded into your business layer.

Conclusion

CSLA .NET and Blazor are a perfect match! Blazor is a UI framework built directly on top of the core architectural concepts I've been championing with CSLA since 1996.

In Chapter 8 I will discuss the ProjectTracker reference app, and how it implements server-side and client-side Blazor apps based on everything discussed in Chapters 1 through 7.

Chapter 8: ProjectTracker UI Using Blazor

In Chapters 1-6 I discussed the basic use of Blazor, including data binding, multi-headed deployments, authentication, and authorization. In Chapter 7 I discussed how CSLA .NET supports Blazor UI development with the helper types in the `Csla.Blazor` and `Csla.Blazor.WebAssembly` NuGet packages.

In this chapter I will use the ProjectTracker reference app as an example of how to build server-side and client-side Blazor apps, with a common UI codebase, that makes use of a CSLA-based business domain layer.

> The code for this example is the `Samples/ProjectTracker` solution in the CSLA .NET GitHub repo: https://github.com/MarimerLLC/csla

The ProjectTracker app includes a data access layer, a business layer, and numerous apps for various types of UI technologies. All the apps make use of those common business and data layers, and the existing ProjectTracker app server that exposes CSLA data portal endpoints.

The solution, with the Blazor UI implementation, exists in GitHub. I will describe how the Blazor UI projects were created, configured, and implemented.

Create the Blazor UI Projects

In the ProjectTracker sample, the Blazor UI consists of three projects, created following the concepts discussed in Chapter 6.

- Shared Blazor UI project
- Client-side Blazor UI project
- Server-side Blazor UI project

I will detail the creation of each project.

Create a Common Blazor UI Project

The common UI project is named `ProjectTracker.Ui.Blazor`, and it is a Class Library project set up as a shared UI project like I discussed in Chapter 6. Here is the relevant code from the `csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
    <LangVersion>7.3</LangVersion>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>
```

The project relies on types from the `Csla.Blazor` NuGet package and other Microsoft packages:

```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components" Version="3.1.4" />
    <PackageReference Include="Microsoft.AspNetCore.Components.Web" Version="3.1.4" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient"
                      Version="3.2.0-preview3.20168.3" />
    <PackageReference Include="Csla.Blazor" Version="5.2.0" />
    <PackageReference Include="Microsoft.AspNetCore.Components.Authorization"
                      Version="3.1.4" />
  </ItemGroup>
```

Because this project will implement the Blazor UI, it needs access to the existing business domain layer in the

`ProjectTracker.BusinessLibrary.NetStandard` project. It also needs access to the abstract types from the data access layer in the `ProjectTracker.Dal` project. Here is the relevant code from the `csproj` file:

```
<ItemGroup>
  <ProjectReference Include="..\ProjectTracker.BusinessLibrary.Netstandard
    \ProjectTracker.BusinessLibrary.Netstandard.csproj" />
  <ProjectReference Include="..\ProjectTracker.Dal\ProjectTracker.Dal.csproj" />
</ItemGroup>
```

I won't discuss the business or data access layers in this book, focusing instead on how to leverage the business domain types when creating a Blazor UI.

IsServerSide Property

Sometimes it is necessary to have different code or behavior depending on whether the app is deployed using client-side WebAssembly or server-side ASP.NET Core. In this app there's different UI markup for the login process on the client and server side deployments.

One easy way to address this requirement is to add an `IsServerSide` property to the `App` class as defined in the `App.razor` file. You'll see this in the `@code` block.

```
public static bool IsServerSide { get; set; }
```

This value will be set as the client-side and server-side apps start up, and it is used in the `MainLayout.razor` component.

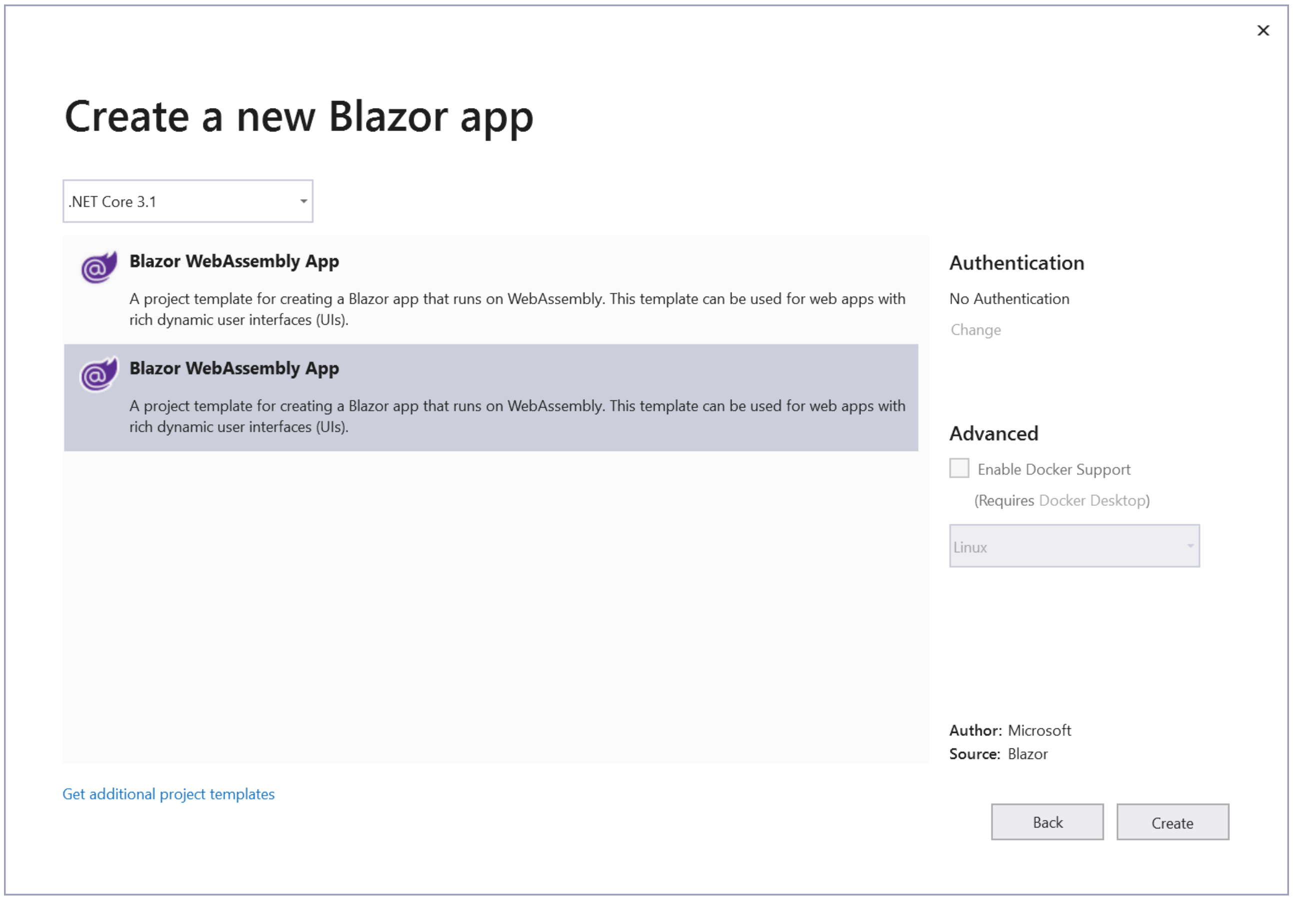
Let's talk about the client-side and server-side app projects.

Create a Client-Side Blazor Project

The `ProjectTracker.Ui.Blazor.Client` project is a standalone Blazor WebAssembly project.

Creating the Project

It was created in Visual Studio by not selecting the "ASP.NET Core hosted" option in the wizard.


Create a new Blazor app

.NET Core 3.1

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Advanced

Enable Docker Support

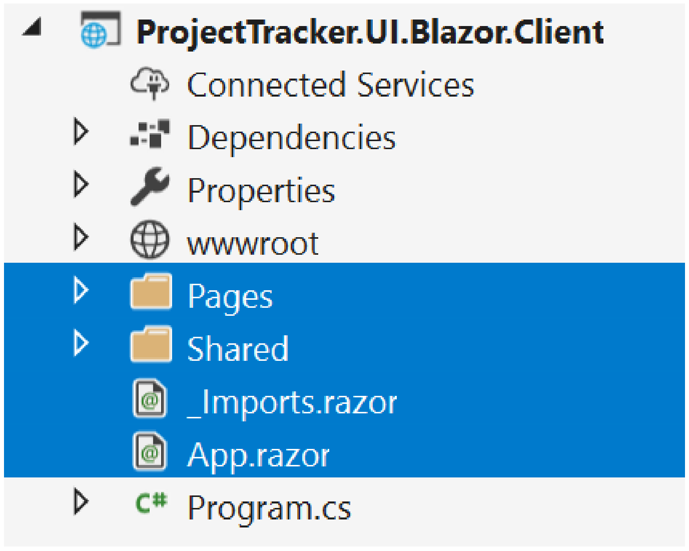
(Requires Docker Desktop)

Author: Microsoft
Source: Blazor

This means that the project can be directly launched from Visual Studio, or published and deployed to nearly any static web server host.

Removing Template Items

The resulting project contains Blazor components and other elements that are not needed because this project will get its UI elements from the shared project.



Remove the

- Pages folder
- Shared folder
- `_Imports.razor` file

- `App.Razor` file

All that is required is the `Program.cs` file for startup configuration, and the wwwroot folder that contains static CSS and other resources required by the UI.

Project References

The project references the `Csla.Blazor.WebAssembly` NuGet package and other standard client-side Blazor packages. And it references the shared UI project. All this is in the `csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Csla.Blazor.WebAssembly" Version="5.2.0" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="3.2.0" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Build"
                      Version="3.2.0" PrivateAssets="all" />
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"
                      Version="3.2.0" PrivateAssets="all" />
    <PackageReference Include="System.Net.Http.Json" Version="3.2.0-preview5.20210.3" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient"
                      Version="3.2.0-preview3.20168.3" />

  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\ProjectTracker.Ui.Blazor\ProjectTracker.Ui.Blazor.csproj" />
  </ItemGroup>

</Project>
```

As I discussed in Chapters 3 and 6, client-side Blazor WebAssembly projects target the `netstandard2.1` framework.

App Configuration

The `Program` class contains the startup configuration for the client-side Blazor app. The configuration is the same as discussed in Chapter 7.

```
      builder.Services.AddAuthorizationCore();
      builder.Services.AddOptions();
      builder.Services.AddSingleton
        <AuthenticationStateProvider, CslaAuthenticationStateProvider>();
      builder.Services.AddSingleton<CslaUserService>();
      App.IsServerSide = false;
      builder.UseCsla(c => c
        .DataPortal()
          .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy),
                        "http://localhost:8040/api/dataportaltext/"));
```

The `AddOptions` and `AddAuthorizationCore` methods are required to configure Blazor authentication and authorization. The `CslaAuthenticationStateProvider` and `CslaUserService` are necessary services for authentication to work as discussed in Chapter 7.

The `App.IsServerSide` property is set to `false` to indicate that the app is running in a client-side WebAssembly deployment.

The `UseCsla` method configures CSLA .NET to work properly in a client-side Blazor app, and this code also configures the data portal to communicate with the existing ProjectTracker app server.

At this point the client-side Blazor project is complete, but there's one more important detail to cover: configuring the app server.

Configuring the App Server

The ProjectTracker app server is in the `ProjectTracker.AppServerCore` project. This is an ASP.NET Core project that exposes data portal endpoints for use by all the different UI apps in the ProjectTracker solution.

There are two changes necessary to use this app server from client-side Blazor: a text-based data portal endpoint, and CORS (cross-origin resource sharing) configuration.

Text-Based Data Portal Endpoint

Remember from Chapter 7 that Blazor WebAssembly can only transfer text-based data via HTTP. This means any app server must expose a data portal endpoint that sends and receives text-encoded data.

In the project's Controllers folder the `DataPortalTextController` class implements this endpoint.

```
[Route("api/[controller]")]
[ApiController]
public class DataPortalTextController : Csla.Server.Hosts.HttpPortalController
{
  public DataPortalTextController()
  {
    UseTextSerialization = true;
  }
}
```

When running the ProjectTracker solution in debug mode on your local computer, the URL for this endpoint is `http://localhost:8040/api/dataportaltext/`.

Contrast this to the `DataPortalController` that uses the default binary behavior.

```
[Route("api/[controller]")]
[ApiController]
public class DataPortalController : Csla.Server.Hosts.HttpPortalController
{
}
```

When running the ProjectTracker solution in debug mode on your local computer, the URL for this endpoint is `http://localhost:8040/api/dataportal/`.

Most client apps can transfer data in binary format and so use this second endpoint. The Blazor WebAssembly app will use the first endpoint to transfer the same data, but text-encoded.

CORS Configuration

By default, ASP.NET Core prevents the use of cross-origin resource sharing (CORS). This means that, by default, browser-hosted apps are not allowed to directly call the service endpoints exposes by the app server. That is true for any JavaScript-based technologies such as Angular and React, as well as WebAssembly-based technologies such as Blazor.

Enabling CORS in ASP.NET Core requires editing the `Startup` class and adding code to the `ConfigureServices` method:

```
services.AddCors(options =>
{
  options.AddPolicy(BlazorClientPolicy,
    builder =>
    {
      builder
      .AllowAnyOrigin()
      .AllowAnyHeader()
      .AllowAnyMethod();
    });
});
```

This must be added before the call to the `AddMvc` method.

In the `AddCors` method call you can provide more targeted policies to control exactly which clients are allowed to call your service endpoints. In this example there are no restrictions.

Similarly, the `Configure` method needs a new line of code:

```
app.UseCors(BlazorClientPolicy);
```

This line is required before calling the `UseMvc` method.

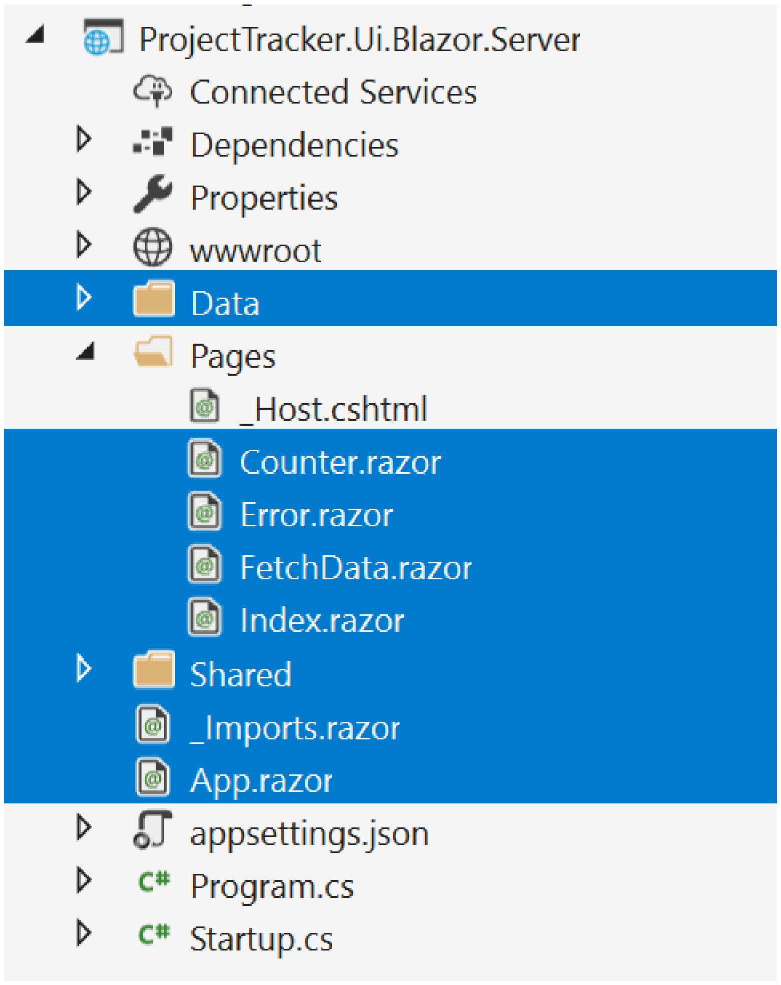
At this point the client-side Blazor app is complete, and the app server has been enhanced to allow WebAssembly clients to call the services endpoints it is hosting.

Create a Server-Side Blazor Project

The server-side Blazor UI project in the ProjectTracker solution is very similar to the projects from Chapters 2 and 6. The `ProjectTracker.Ui.Blazor.Server` project is a standard server-side Blazor project.

Removing Template Items

Because this project references the shared UI project, the UI components have been removed from the server-side project.



The following are removed:

- Data folder
- `Counter.razor` from the Pages folder
- `Error.razor` from the Pages folder
- `FetchData.razor` from the Pages folder
- `Index.razor` from the Pages folder
- `_Imports.razor` file
- `App.razor` file

These are not needed because the Blazor UI will come from the shared UI project.

Project References

The server-side project references the shared UI project in its `csproj` file:

```
<ItemGroup>
  <ProjectReference Include="..\ProjectTracker.Ui.Blazor\ProjectTracker.Ui.Blazor.csproj" />
</ItemGroup>
```

Because this is a .NET Core 3.1 project, the reference to `ProjectTracker.Ui.Blazor` will pull in the .NET Core 3.1 DLL from that project.

The server-side project also references the `Csla.AspNetCore` and `Csla.Blazor` packages along with other required packages:

```
<ItemGroup>
  <PackageReference Include="Csla.AspNetCore" Version="5.2.0" />
  <PackageReference Include="Csla.Blazor" Version="5.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient"
                    Version="3.2.0-preview3.20168.3" />
  <PackageReference Include="Microsoft.AspNetCore.Components.Authorization" Version="3.1.4" />
  <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="3.1.4" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="3.1.2" />
</ItemGroup>
```

As I discussed in Chapter 7, when creating a server-side Blazor project with CSLA .NET both the `Csla.AspNetCore` and `Csla.Blazor` packages are required. This is because the app is both ASP.NET Core and also Blazor at the same time.

App Configuration

Server-side Blazor app configuration occurs in the `Startup` class.

The `ConfigureServices` method needs two new lines of code:

```
      services.AddHttpContextAccessor();
      services.AddCsla().WithBlazorServerSupport();
```

The `AddHttpContextAccessor` method allows the use of `HttpContext` throughout the app. This is required because CSLA .NET uses the `HttpContext` instance.

The `AddCsla` method adds services required by CSLA .NET to function.

And the `Configure` method needs to configure for authentication and authorization:

```
      app.UseAuthentication();
      app.UseAuthorization();
```

These lines must be added before the `UseEndpoints` method call.

The code also sets the `IsServerSide` property to `true` so it is possible to know that this app is running from a server-side deployment:

```
      App.IsServerSide = true;
```

At the end of the method the code configures CSLA .NET for use in a server-side Blazor app:

```
      app.UseCsla(c => c
        .DataPortal()
          .DefaultProxy(typeof(Csla.DataPortalClient.HttpProxy),
          "http://localhost:8040/api/dataportal/"));
```

The `UseCsla` method configures CSLA .NET to function in an ASP.NET Core Blazor app, and also configures the data portal to interact with the app server.

Notice that the data portal is configured to call the binary data portal endpoint. Because the Blazor code is running in ASP.NET Core it can use binary data transfers. Only client-side Blazor running in WebAssembly is restricted to using text-based encoding.

At this point the server-side Blazor app is configured. The next step is to implement the Blazor UI itself, in the shared UI project.

Implementing the Blazor UI

The Blazor UI is implemented in the shared `ProjectTracker.Ui.Blazor` project. All the pages and components in the app are implemented in the shared project except for the server-side authentication.

The ProjectTracker app is designed to demonstrate several common user scenarios, including:

- User views and selects from a list of items
- User edits a list of items
- User creates or edits an object graph with child items
- User can log in and out of the app

I will discuss the implementation of each of these user scenarios in the context of the ProjectTracker app.

> There are *many* ways to design and implement any user experience. The choices I've made in the design of the ProjectTracker UI are just one option, and you may choose other navigation, display, or interaction patterns. That's fine, and you should be able to adapt the concepts from this chapter to many other types of Blazor UI.

Let's start by discussing one of the most common user scenarios: displaying a read-only list.

Read-Only Lists

One of the most common user scenarios is for the user to view a list of items, and possible select an item from the list with which to interact. That interaction might be to view details, edit, delete, or otherwise affect the selected item from the list.

CSLA .NET supports this concept at the business layer with the read-only list stereotype, relying on the `ReadOnlyList` base type to create business objects for use by the UI app.

In the `ProjectTracker.Library` namespace, the `ProjectList` and `ResourceList` classes are examples of the read-only list stereotype. Each of them relies on a child type, `ProjectInfo` and `ResourceInfo` respectively, to contain the necessary information for each item in the list.

In the `ProjectTracker.Ui.Blazor` project these types are exposed to the user in the `Projects` and `Resources` pages in the Pages folder. These pages use the same implementation, and so I will only walk through the `Projects` component.

Getting to the Page

The `NavMenu` component in the Shared folder includes a link so the user can access the `Projects` page.

```
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="projects">
        <span class="oi oi-plus" aria-hidden="true"></span> Projects
      </NavLink>
    </li>
```

There's nothing new or special about this navigation link.

Projects Page Header

At the top of the `Projects` page there's code to handle routing, bring in namespaces, and inject needed instances:

```
@page "/projects"
@using  Csla.Blazor
@using  Csla.Rules
@inject ViewModel<ProjectTracker.Library.ProjectList> vm
```

As discussed in Chapter 7, in this chapter I am using the MVVM design pattern that relies on the `ViewModel` type from the `Csla.Blazor` namespace. In this case the component requires access to a viewmodel object representing the `ProjectList` business type.

Displaying Page Level Errors

The page has markup to display the list of items to the user. Near the top of the markup is a line to display any page-level errors that might occur.

```
<p class="text-danger">@vm.ViewModelErrorText</p>
```

This line binds to the viewmodel object's `ViewModelErrorText` property, which will contain any error messages encountered by the `ViewModel` type as it attempts to retrieve or save the model object.

Handling a Null Model

As the page first loads the `Model` property of the viewmodel object will be null. As I discussed in Chapter 4, Blazor data binding can't handle a null binding target, so it is necessary to check the value and display alternate content until the value is no longer null:

```
@if (vm.Model == null)
{
  <p>Loading...</p>
}
```

This way, as the page first appears and is waiting to get any data from the app server or database, the user will see "Loading..." on the page.

The rest of the markup is in an `else` clause and is visible when the `Model` property is not null.

Allow User to Add New Item

Users may or may not be authorized to add new items to the list. This business rule is encoded in the business domain layer as a per-type rule associated with the `ProjectEdit` type. The UI should reflect this rule by only showing the user a "new item" link if they are authorized to add a new item:

```
  <AuthorizeView Policy="@(CslaPolicy.GetPolicy(AuthorizationActions.CreateObject,
                                              typeof(ProjectTracker.Library.ProjectEdit)))">
    <p><a href="projectedit">Add project</a></p>
  </AuthorizeView>
```

The Blazor `AuthorizeView` component is used, along with the `CslaPolicy` type discussed in Chapter 7, to determine whether the current user is authorized to create new projects, and whether to display the "Add project" link on the page.

If the user clicks the "Add project" link the app will navigate to the `ProjectEdit` page. I will discuss that page later in this chapter.

Displaying the List of Items

The next block of markup defines a `table` with three columns:

1. Display item `Id` property
2. Display item `Name` property
3. Provide the user with any actions allowed on the item

Displaying the list of `Id` and `Name` values is straightforward:

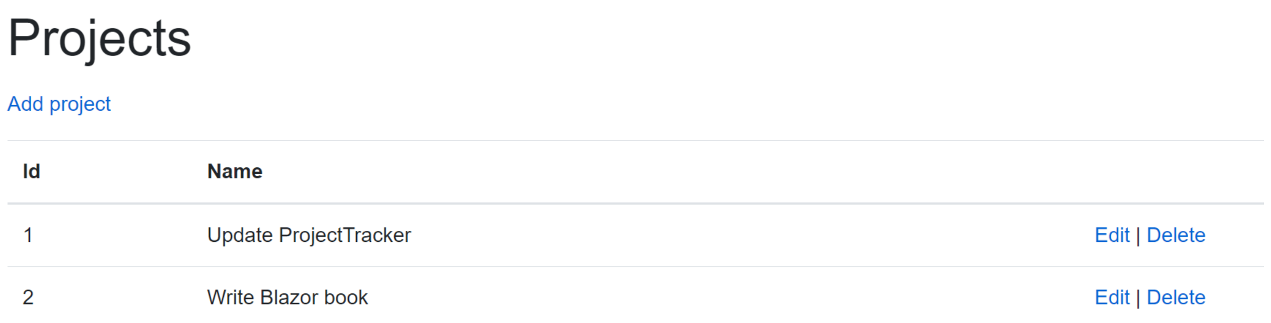
```
    @foreach (var item in vm.Model)
    {
      <tr>
        <td>@item.Id</td>
        <td>@item.Name</td>
```

A `foreach` loop is used, and each item in the list is rendered as a row in the table.

The third column is more complex however, because authorization business rules must be honored. The current user may or may not be allowed to edit or delete each item in the list.

```
        <td>
            <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
                AuthorizationActions.EditObject,
                typeof(ProjectTracker.Library.ProjectEdit)))">
              <a href="projectedit/@item.Id">Edit</a>
              <span> | </span>
            </AuthorizeView>
            <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
                AuthorizationActions.DeleteObject,
                typeof(ProjectTracker.Library.ProjectEdit)))">
              <a href="javascript: void(0);"
                @onclick="() => SelectForDelete(item.Id)">Delete</a>
            </AuthorizeView>
        </td>
```

The `AuthorizeView` component is used with the `CslaPolicy` type to determine whether the business rules encoded in the business layer allow the current user to edit or delete each row.


Projects

Add project

Id	Name	
1	Update ProjectTracker	Edit | Delete
2	Write Blazor book	Edit | Delete

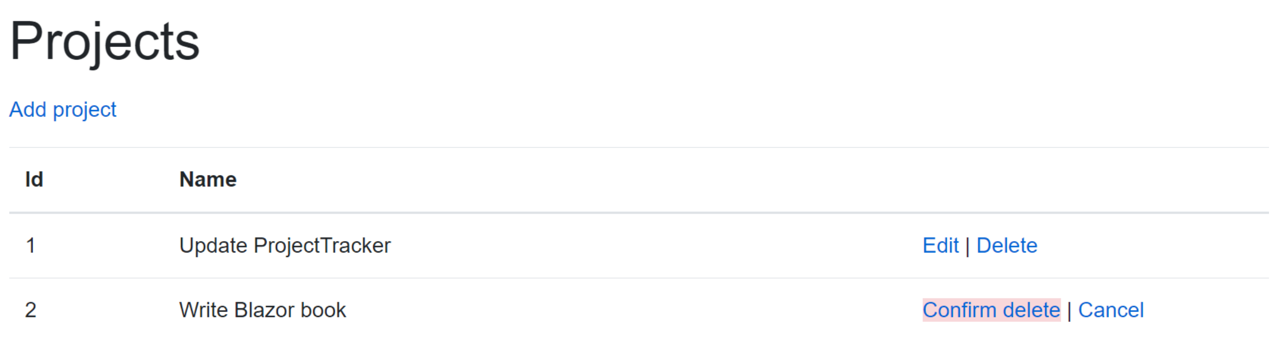
If the user is authorized to edit an item they will see an "Edit" link. Clicking that link will navigate the user to the `Projectedit` page. I will discuss that page later in this chapter.

If the user is authorized to delete an item they will see a "Delete" link. Clicking that link will call a `SelectForDelete` method in the page's code.

Implementing Delete Confirmation

When the user indicates they want to delete an item by clicking the "Delete" link on a row I want to confirm that they really mean to delete the item.

There are many ways to implement such a confirmation step, and in this page I have chosen to use an in-place confirmation model.


Projects

Add project

Id	Name	
1	Update ProjectTracker	Edit | Delete
2	Write Blazor book	Confirm delete | Cancel

Blazor makes this easy to implement with data binding.

When the "Delete" link is clicked a `SelectForDelete` method is invoked.

```
  <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
    AuthorizationActions.DeleteObject,
    typeof(ProjectTracker.Library.ProjectEdit)))">
  <a href="javascript: void(0);"
    @onclick="() => SelectForDelete(item.Id)">Delete</a>
  </AuthorizeView>
```

Notice that the `@onclick` event is handled by a lambda expression that invokes the `SelectForDelete` method. This is necessary because the method requires a paramter, and a lambda expression allows passing in a parameter to a method.

The `SelectForDelete` records the `Id` property value of the item selected:

```
private void SelectForDelete(int id)
{
  itemSelectedForDeletion = id;
}
```

The `itemSelectedForDeletion` field is used in the page to determine which markup should be rendered for the selected row:

```
  @if (item.Id == itemSelectedForDeletion)
  {
    <span class="alert-danger"><a href="javascript: void(0);"
      @onclick="() => Delete(item.Id)">Confirm delete</a></span>
    <span> | </span><a href="javascript: void(0);"
      @onclick="() => itemSelectedForDeletion = -1">Cancel</a>
  }
  else
  {
    <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
         AuthorizationActions.EditObject,
         typeof(ProjectTracker.Library.ProjectEdit)))">
      <a href="projectedit/@item.Id">Edit</a>
      <span> | </span>
    </AuthorizeView>
    <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
         AuthorizationActions.DeleteObject,
         typeof(ProjectTracker.Library.ProjectEdit)))">
      <a href="javascript: void(0);"
        @onclick="() => SelectForDelete(item.Id)">Delete</a>
    </AuthorizeView>
  }
```

If the current row matches the selected row then a delete confirmation is rendered:

```
    @if (item.Id == itemSelectedForDeletion)
    {
      <span class="alert-danger"><a href="javascript: void(0);" @onclick="() => Delete(item.Id)">Confirm
delete</a></span>
      <span> | </span><a href="javascript: void(0);"
        @onclick="() => itemSelectedForDeletion = -1">Cancel</a>
    }
```

Otherwise the a standard row is rendered.

If the user now clicks the "Confirm delete" link the `Delete` method is called:

```
  private async void Delete(int id)
  {
    itemSelectedForDeletion = -1;
    vm.Model = null;
    await ProjectTracker.Library.ProjectEdit.DeleteProjectAsync(id);
    await vm.RefreshAsync(() =>
      Csla.DataPortal.FetchAsync<ProjectTracker.Library.ProjectList>());
    StateHasChanged();
  }
```

This method uses the existing business layer API to call the `DeleteProjectAsync` method of the `ProjectEdit` domain type. That method implements the behavior necessary to delete a project from the system.

Once the project has been deleted it is necessary to reload the `Model` object so the list reflects the change. This is handled by calling the `RefreshAsync` method on the viewmodel object.

Because the `Delete` method calls asynchronous methods, it is necessary to tell Blazor to refresh the display by calling the `StateHasChanged` method.

If the user clicks the "Cancel" link the implementation is simpler. All that is necessary is to set the `itemSelectedForDeletion` field to an invalid value such as `-1`. This ensures that it will never match any row in the `ProjectList` collection and so the confirmation markup will never appear to the user.

The `Projects` and `Resources` pages demonstrate how to display read-only lists to the user, and to add actions for the user to take based on authorization rules from your business layer.

In-Place Editing of Lists

The next user scenario I will discuss, implemented in the `Roles` page, is where the user is viewing a list of items and needs to edit one of the items. In some ways this is very similar to the way the delete confirmation behavior was implemented in the `Projects` page.

This in-place editing concept works well if there are only a couple fields for the user to edit. If more fields need to be edited or displayed, then you should create an edit form as discussed later in this chapter.

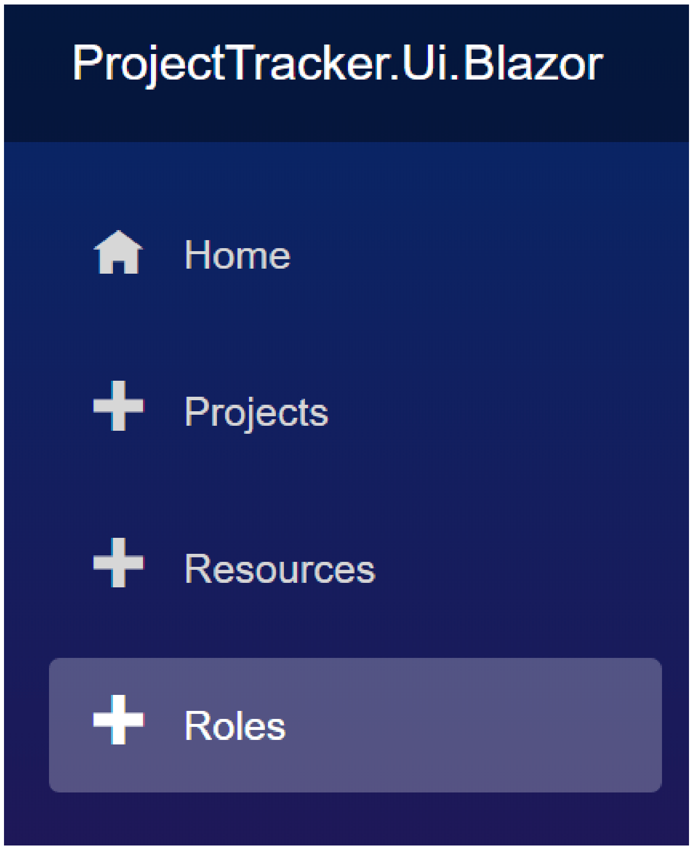
In the ProjectTracker example only users in the "Administrator" role are allowed to edit the list of roles represented by the `RoleEditList` type in the business layer. This is a read-write collection that allows an authorized user to add, edit, and remove items.

Authorizing Navigation

Because only certain users are allowed to view and edit the list of roles, the `NavMenu` component makes use of the `AuthorizeView` component to check the CSLA-based authorization rule for the menu item.

```
<AuthorizeView Policy="@(CslaPolicy.GetPolicy(
    AuthorizationActions.EditObject,
    typeof(ProjectTracker.Library.Admin.RoleEditList)))">
  <li class="nav-item px-3">
    <NavLink class="nav-link" href="roles">
      <span class="oi oi-plus" aria-hidden="true"></span> Roles
    </NavLink>
  </li>
</AuthorizeView>
```

This ensures that only authorized users will see the "Roles" link.



When the user clicks on the "Roles" link they are presented with the `Roles` page.

The `Roles` page has an authorization attribute to prevent unauthorized users from navigating directly to the URL:

```
@attribute [HasPermission(
  AuthorizationActions.EditObject, typeof(RoleEditList))]
```

In both cases the business layer API is used to determine if the current user is authorized to edit an object of type `RoleEditList` so there are no explicit rules in the UI, and yet the UI is able to alter the user experience based on the rules.

Page Layout

The page also relies on a `ViewModel` instance for the `RoleEditList` type:

```
@inject ViewModel<RoleEditList> vm
```

As with all pages that dynamically load content for data binding, any viewmodel errors are displayed and the `Model` property is check for a null value:

```
<p class="text-danger">@vm.ViewModelErrorText</p>

@if (vm.Model == null)
{
  <p>Loading...</p>
}
```

If the `Model` value is not null then the user might be presented with a link to add new items.

```
  <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
      AuthorizationActions.CreateObject,
      typeof(ProjectTracker.Library.Admin.RoleEditList)))">
    <p><a href="javascript: void(0);" @onclick="AddRole">Add role</a></p>
  </AuthorizeView>
```

The `AuthorizeView` component is used to determine whether to display the link to the current user, based on authorization rules implemented in the business layer.

If the user clicks the "Add role" link an `AddRole` method is called. I will discuss this method and how a new item is added to the list later in this chapter.

As with the `Projects` page, a `table` is used to define three columns for the `Id` and `Name` properties, and for potential user actions.

```
  <table class="table">
    <thead>
      <tr>
        <th>Id</th>
        <th>Name</th>
        <th></th>
      </tr>
    </thead>
```

The rest of the page markup is a little more complex than the `Projects` page. This is because each row of content can be displayed in one of several modes:

- Display the item plus actions
- In-place editing of the item
- In-place confirmation of item deletion

A `foreach` loop is used to display each item in the list, and page-level `itemSelectedForEdit` and `itemSelectedForDeletion` fields are used to indicate the rows that should be rendered for edit or confirmation of delete.

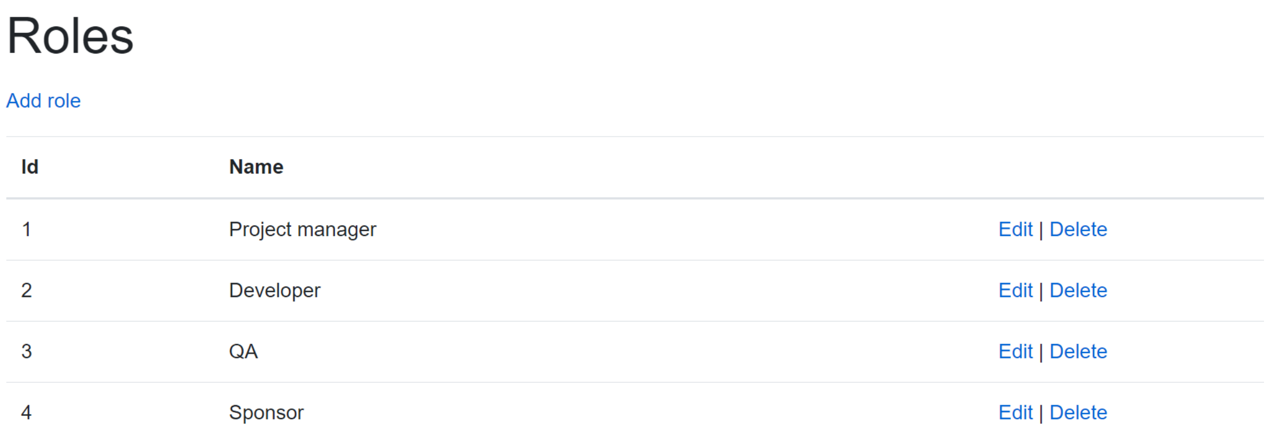
Let's look at the markup and code for each mode.

Display Item

If an item is not selected for edit or delete, its data is displayed to the user, along with links for authorized actions.

```
    <td>@item.Id</td>
    <td>@item.Name</td>
    <AuthorizeView Policy="@(CslaPolicy.GetPolicy(
         AuthorizationActions.EditObject,
         typeof(ProjectTracker.Library.Admin.RoleEditList)))">
      <Authorized>
        <td>
          <a href="javascript: void(0);"
            @onclick="() => SelectForEdit(item.Id)">Edit</a>
          <span> | </span>
          <a href="javascript: void(0);"
            @onclick="() => SelectForDelete(item.Id)">Delete</a>
        </td>
      </Authorized>
      <NotAuthorized>
        <td></td>
      </NotAuthorized>
    </AuthorizeView>
```

The `AuthorizeView` component is used to check the authorization rules for the current user. The "Edit" and "Delete" links are only visible to authorized users.



Roles

Add role

Id	Name	
1	Project manager	Edit | Delete
2	Developer	Edit | Delete
3	QA	Edit | Delete
4	Sponsor	Edit | Delete

If the user clicks the "Edit" or "Delete" links the code calls `SelectForEdit` or `SelectForDelete` methods respectively, passing the current item's `Id` property as a parameter.

I will discuss these two methods in the context of their specific UI.

Delete Item

The `SelectForDelete` method is very similar to the one in the `Projects` page:

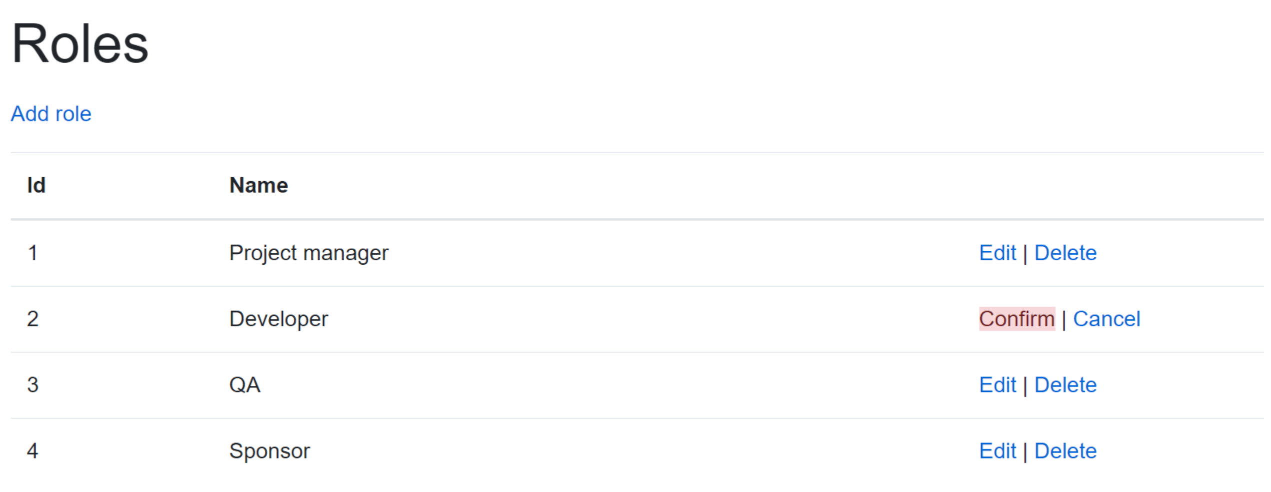
```
private void SelectForDelete(int id)
{
  addingRole = false;
  EditCancelled();
  itemSelectedForDeletion = id;
}
```

The difference is that this code makes sure any current add or edit operation is stopped before marking an item for deletion.

When the list of items is rendered, any item marked for deletion will render with a confirmation link instead of the default actions.

```
        @if (item.Id == itemSelectedForDeletion)
        {
          <td>@item.Id</td>
          <td>@item.Name</td>
          <td>
            <a class="alert-danger"
               href="javascript: void(0);"
               @onclick="() => Delete(item.Id)">Confirm</a>
            <span> | </span>
            <a href="javascript: void(0);"
               @onclick="() => itemSelectedForDeletion = -1">Cancel</a>
          </td>
        }
```

This is similar to the `Projects` page earlier in this chapter.



Roles

Add role

Id	Name	
1	Project manager	Edit | Delete
2	Developer	Confirm | Cancel
3	QA	Edit | Delete
4	Sponsor	Edit | Delete

The `Delete` method is invoked if the user clicks the "Confirm" link.

```
private async void Delete(int id)
{
  itemSelectedForDeletion = -1;
  var item = vm.Model.Where(r => r.Id == id).FirstOrDefault();
  if (item != null)
    vm.Model.Remove(item);
  await SaveAndRefresh();
}
```

This method finds the selected item in the list and removes it from the list. Remember that the `RoleEditList` type is an editable business list and so its items are manipulated in memory, and then the list is saved.

The `SaveAndRefresh` method is called to save the list and refresh the UI.

```
private async Task SaveAndRefresh()
{
  await vm.SaveAsync();
  await vm.RefreshAsync(() =>
    Csla.DataPortal.FetchAsync<RoleEditList>());
  StateHasChanged();
}
```

The `SaveAsync` method saves any changes in the list by calling the data portal. The `RefreshAsync` method reloads the list from the app server so it reflects any changes to the underlying database.

Finally, because these are asynchronous method calls, the `StateHasChanged` method is called so Blazor understands that it must refresh all data binding.

Edit Item

Editing an item is a little more complex because changes made to an item occur in memory (potentially in a browser) on the client, and if the user chooses to cancel edits made to a row there must be some way to reset the item to its original state.

This can be done by reloading the list with a data portal call, but it is usually more effective to just restore the original values without calling a remote service. CSLA .NET supports this concept using a concept called *n-level undo* as discussed in the *Using CSLA: Creating Business Objects* book.

The idea is that a snapshot of the row object's state is taken before allowing the user to edit any values. If the user accepts their edits then the snapshot is discarded. If the user cancels their edits then that original snapshot is used to restore the object to its original values.

When the user clicks the "Edit" link on an item in the list a `SelectForEdit` method is invoked:

```
private void SelectForEdit(int id)
{
  addingRole = false;
  EditCancelled();
  itemSelectedForDeletion = -1;
  itemSelectedForEdit = id;
  var item = vm.Model.Where(r => r.Id == id).FirstOrDefault();
  if (item != null)
    item.BeginEdit();
}
```

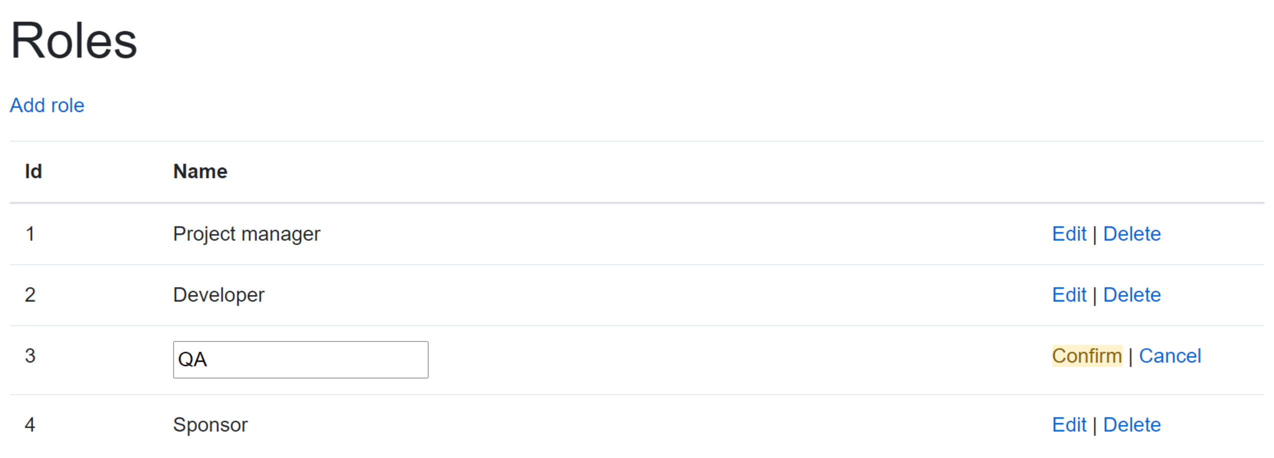
This method makes sure no new item is being added or existing item being edited or deleted, then it sets the `itemSelectedForEdit` field to indicate that this row is being edited.

Finally the item is found in the collection and a `BeginEdit` method is called. This method takes a snapshot of the object's current state for later use.

In the markup, the `itemSelectedForEdit` field is used to render the selected row for editing:

```
else if (item.Id == itemSelectedForEdit)
{
  <td>@item.Id</td>
  <td>
    <TextInput Property="@(vm.GetPropertyInfo(() => item.Name))" />
  </td>
  <td>
    <a class="alert-warning" href="javascript: void(0);"
      @onclick="EditAccepted">Confirm</a>
    <span> | </span>
    <a href="javascript: void(0);"
      @onclick="EditCancelled">Cancel</a>
  </td>
}
```

This markup renders input controls for each editable property of the business object, along with "Confirm" and "Cancel" action links.


Roles

Add role

Id	Name	
1	Project manager	Edit | Delete
2	Developer	Edit | Delete
3	QA	Confirm | Cancel
4	Sponsor	Edit | Delete

If the user clicks the "Cancel" link the `EditCancelled` method is invoked:

```
private void EditCancelled()
{
  if (itemSelectedForEdit > -1)
  {
    var item = vm.Model
      .Where(r => r.Id == itemSelectedForEdit)
      .FirstOrDefault();
    if (item != null)
      item.CancelEdit();
    itemSelectedForEdit = -1;
  }
}
```

This method finds the item currently being edited and calls its `CancelEdit` method. This method restores the object's original state from the snapshot created by calling the `BeginEdit` method earlier.

If the user clicks the "Confirm" button the `EditAccepted` method is called:

```
private async void EditAccepted()
{
  var item = vm.Model.Where(
    r => r.Id == itemSelectedForEdit).FirstOrDefault();
  if (item != null)
    item.ApplyEdit();
  itemSelectedForEdit = -1;
  await SaveAndRefresh();
}
```

This method finds the item being edited and calls its `ApplyEdit` method. This method discards the snapshot taken by the `BeginEdit` method, effectively committing the user's changes in memory.

The method then calls the `SaveAndRefresh` method to save the list and refresh the UI. This is the same method called in the delete operation.

Whether the user clicks "Confirm" or "Cancel" or chooses to edit or delete another row, this row is no longer selected for editing.

Add a New Item

The last action available to the user in this page is to click an "Add role" link to add a new item to the list. When the user clicks that link an `AddRole` method is called:

```
private void AddRole()
{
  itemSelectedForDeletion = -1;
  EditCancelled();
  newRole = Csla.DataPortal.CreateChild<RoleEdit>();
  addingRole = true;
}
```

This method ensures no existing item is being edited or deleted, then it creates a new child object for the list:

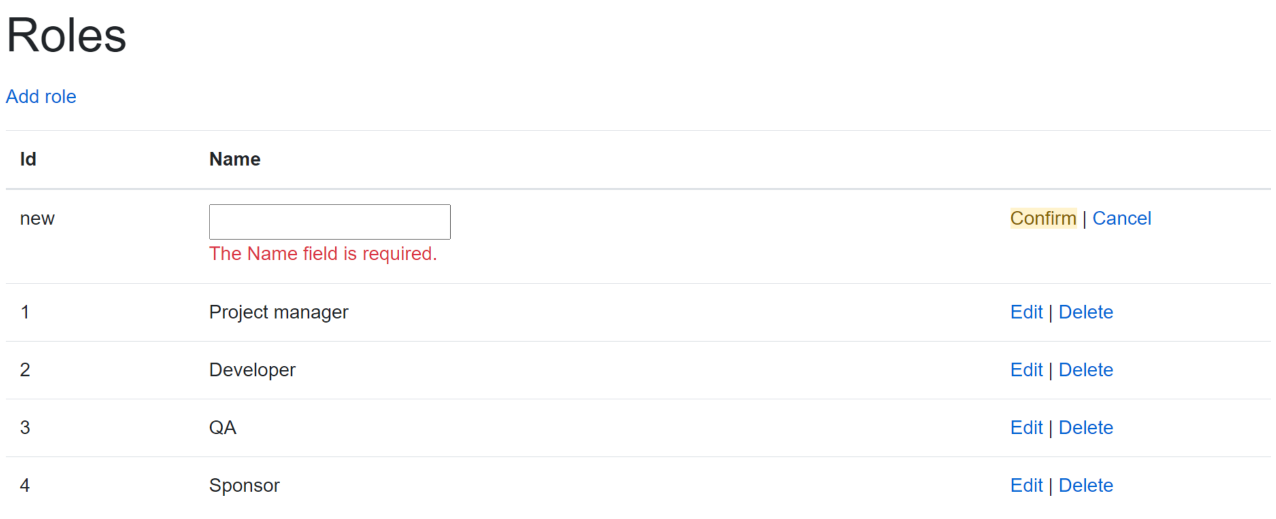
```
  newRole = Csla.DataPortal.CreateChild<RoleEdit>();
```

It is important to understand that this new `RoleEdit` item *has not been added to the list*, it is referenced by the `newRole` field in the page.

The `addingRole` field is set so Blazor renders UI for the user to add a new item. Here is the markdown:

```
@if (addingRole)
{
  <tr>
    <td>new</td>
    <td>
      <TextInput Property="@(vm.GetPropertyInfo(() => newRole.Name))" />
    </td>
    <td>
      <a class="alert-warning" href="javascript: void(0);"
        @onclick="AddAccepted">Confirm</a>
      <span> | </span>
      <a href="javascript: void(0);"
        @onclick="AddCancelled">Cancel</a>
    </td>
  </tr>
}
```

This markup is *above* the `foreach` loop that displays the existing items from the list. That means when the user clicks the "New role" link that the new item appears at the top of the list.


Roles

Add role

Id	Name	
new	The Name field is required.	Confirm | Cancel
1	Project manager	Edit | Delete
2	Developer	Edit | Delete
3	QA	Edit | Delete
4	Sponsor	Edit | Delete

As with other edit pages, the `TextInput` component is used to provide the user with a rich editing experience, including the display of validation messages from the business layer. In this case though, the way the `GetPropertyInfo` method is called is a little different:

```
<TextInput Property="@(vm.GetPropertyInfo(() => newRole.Name))" />
```

Notice that the target of the lambda is the `newRole` object. Even though this object has not yet been added to the viewmodel object's `Model` property the `GetPropertyInfo` is able to bind to the `Name` property as expected.

After editing the row, the user can click on the "Cancel" button, calling the `AddCancelled` method:

```
private void AddCancelled()
{
  addingRole = false;
  newRole = null;
}
```

This method discards the new object without adding it to the list or calling the data portal, and it sets the `addingRole` field so Blazor stops rendering the UI for adding a new item.

The user can click on the "Confirm" link to run the `AddAccepted` method:

```
private async void AddAccepted()
{
  addingRole = false;
  vm.Model.Add(newRole);
  await SaveAndRefresh();
}
```

This method sets the `addingRole` field to disable the add new item UI. Then it adds the `newRole` object to the collection. Finally, it calls the `SaveAndRefresh` method to save the list through the data portal and to then refresh the UI with any changes to the underlying data.

At this point you should understand how to display a read-only list with actions, and how to create an in-place editing experience for an editable collection of simple data. Now I'll discuss how to create more complex edit pages, including master-detail relationships.

Master-Detail Pages

The `ProjectEdit` and `ResourceEdit` pages in the ProjectTracker app implement master-child edit experiences. They are the same, and so I will focus on the `ProjectEdit` page.

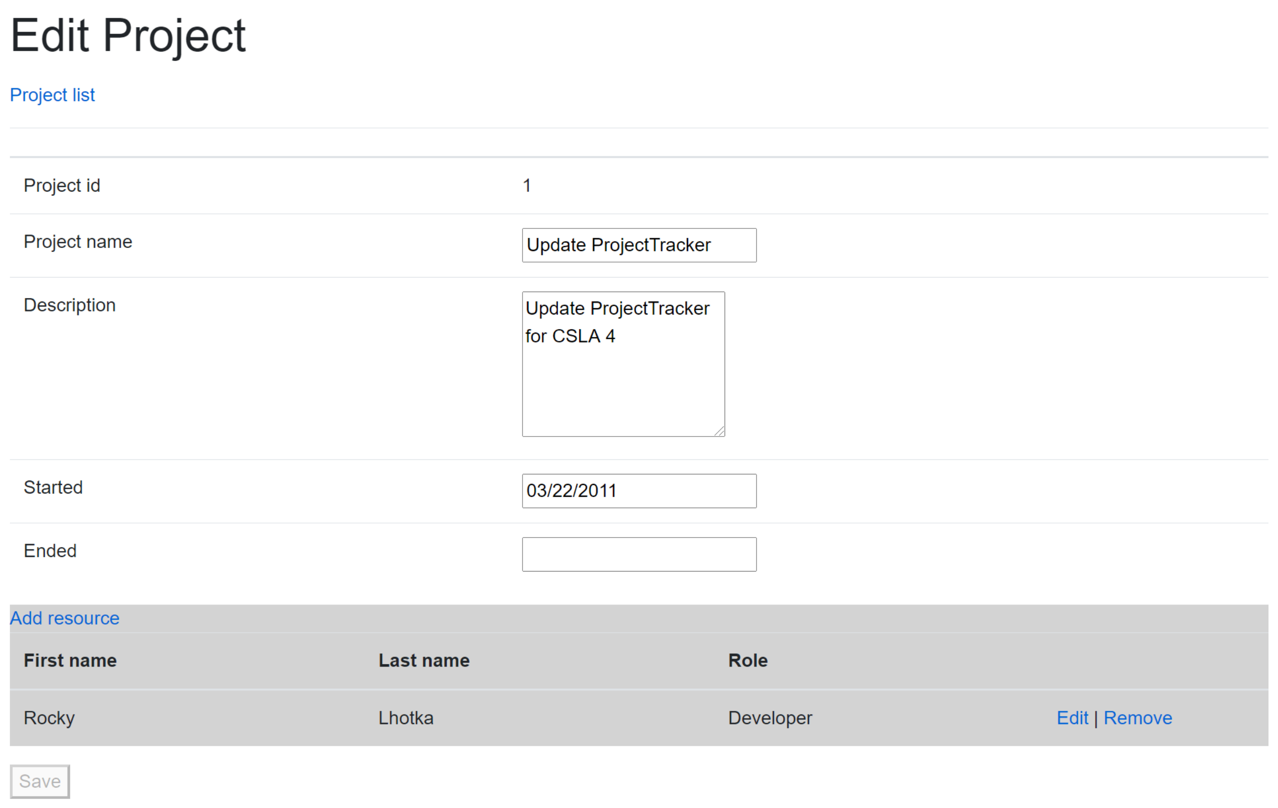
I've chosen to implement the `ProjectEdit` page to use an explicit "Save" button. This means that all edits made by the user occur in memory and are not sent to the server until the user clicks that "Save" button. Also, the "Save" button will not be available for the user to click until the business object graph is in a saveable state.

This page makes use of the `ProjectEdit` business domain type from the business layer. That type encapsulates the business rules and behaviors necessary to edit a project, and is the root object for an object graph that allows the user to view, add, edit, and remove resources from the project.

Page Layout

The `ProjectEdit` page has several areas of UI functionality:

- Page header
- Edit the `ProjectEdit` object properties
- Display list of assigned resources, with associated actions


Edit Project

Project list

Project id	1
Project name	Update ProjectTracker
Description	Update ProjectTracker for CSLA 4
Started	03/22/2011
Ended	

Add resource

First name	Last name	Role	
Rocky	Lhotka	Developer	Edit | Remove

Save

Page Header

The page header markup is similar to the previous pages:

```
@page "/projectedit"
@page "/projectedit/{id:int}"
@inject Csla.Blazor.ViewModel<ProjectTracker.Library.ProjectEdit> vm
@inject NavigationManager NavigationManager

<h1>Edit Project</h1>

<p class="text-danger">@vm.ViewModelErrorText</p>

<p>
  <a href="projects">Project list</a>
</p>

@if (vm.Model == null)
{
  <p>Loading...</p>
}
else
```

Key differences from the earlier pages in this chapter are that this page makes use of the Blazor `NavigationManager` type, and provides a "Project list" page that navigates to the `Projects` page so the user can see a list of projects.

Edit Project Properties

The page uses a `table` to create a two-column edit experience for the properties of the `ProjectEdit` business object. The first column is the friendly name of the property, and the second column allows the user to edit the property value. These are all created using the reusable UI components discussed in Chapter 7.

```
<table class="table">
  <thead>
    <tr>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <LabelRow Property="vm.GetPropertyInfo(() => vm.Model.Id)" />
    <TextInputRow Property="vm.GetPropertyInfo(() => vm.Model.Name)" />
    <TextAreaRow rows="5" Property="@(vm.GetPropertyInfo(() => vm.Model.Description))" />
    <DateInputRow Property="@(vm.GetPropertyInfo(() => vm.Model.Started))" />
    <DateInputRow Property="@(vm.GetPropertyInfo(() => vm.Model.Ended))"
                  EmptyIsMin="false" />
  </tbody>
</table>
```

The UI components used here understand how to interact with CSLA-based authorization and validation rules, adjusting the page content to reflect what the user can and can't view or edit.

I do want to call your attention to the `Ended` property.

```
    <DateInputRow Property="@(vm.GetPropertyInfo(() => vm.Model.Ended))"
                  EmptyIsMin="false" />
```

The `Started` and `Ended` properties of the `ProjectEdit` business class use the CSLA .NET `SmartDate` data type, which understands the concept of an "empty date". For comparison purposes, when a `SmartDate` is "empty" the value is either smaller than, or larger than, the smallest or largest possible date. By default an "empty" value is the smallest value.

In the case of the `Ended` date it is important that an "empty" date be the largest possible date, because there is a business rule that says a project must start before it ends. If the user enters a start date with an empty end date, that rule should pass.

The `DateInputRow` component relies on a `DateInput` component, and that component understands how to work with a `SmartDate` value. In particular, the `DateInput` component has code to convert the business object's property value into and out of a text value:

```
private Csla.SmartDate DateValue;

private string TextValue
{
  get
  {
    DateValue = new Csla.SmartDate((DateTime?)Property.Value, EmptyIsMin);
    DateValue.FormatString = FormatString;
    return DateValue.Text;
  }
  set
  {
    try
    {
      DateValue.Text = value;
    }
    catch (ArgumentException)
    { /* invalid text entry, don't update value */ }
    if (DateValue.IsEmpty)
      Property.Value = null;
    else
      Property.Value = DateValue.Date;
  }
}
```

Notice how the `EmptyIsMin` value is passed through to the `SmartDate` type so it behaves as required.

Display Assigned Resources

The light gray block of content on the page is also a `table`, with each row displaying a resource assigned to the project. The `ProjectEdit` business class has an `AssignedResources` property that provides access to the list of assigned resources.

The assigned resources display area of the page will change, depending on the user's actions. I am using a technique similar to the in-place delete and edit implementations shown earlier in this chapter. In this case the region of the UI will be one of:

- Displaying the list of assigned resources
- Show a list of resources that *could be* assigned to the project
- Allow the user to edit an assigned resource

I chose to use an `enum` to define these three UI modes:

```
public enum SubViewModes
{
  Default = 0,
  Select = 1,
  Details = 2
}
```

In the `ProjectEdit` page there is a field used to control which view is rendered.

```
private SubViewModes viewMode = SubViewModes.Default;
```

I will discuss how each mode is implemented.

Display Assigned Resources

The default mode is to display the list of resources assigned to the project.

```
  else if (viewMode == SubViewModes.Default)
  {
    <a href="javascript: void(0);"
      @onclick="SelectResource">Add resource</a>
    <table class="table">
      <thead>
        <tr>
          <th>First name</th>
          <th>Last name</th>
          <th>Role</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        @foreach (var item in vm.Model.Resources)
        {
          <tr>
            <td>@item.FirstName</td>
            <td>@item.LastName</td>
            <td>@item.RoleName</td>
            <td><a href="javascript: void(0);"
              @onclick="() => EditResource(item.ResourceId)">Edit</a> |
              <a href="javascript: void(0);"
              @onclick="() => RemoveResource(item.ResourceId)">Remove</a></td>
          </tr>
        }
      </tbody>
    </table>
  }
```

The result is a four column display.

First name	Last name	Role	
Rocky	Lhotka	Developer	Edit | Remove

The user can see the first name, last name, and role of the resource, along with actions available for the item.

As with the previous pages in this chapter, the action links invoke methods in the page's code, changing the mode of the display to allow the user to add, edit, or remove the item.

I will discuss the `EditResource` and `RemoveResource` methods later in this chapter. First I will walk through the process of assigning a new resource to the project.

Assign a Resource

At the top of the default display is a link so the user can assign a resource to the project.

```
      <a href="javascript: void(0);" @onclick="SelectResource">Add resource</a>
```

The `SelectResource` method is invoked when this link is clicked.

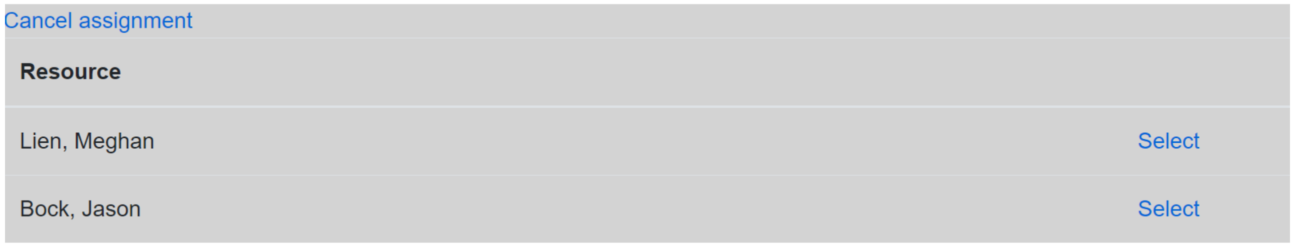
```
  private async void SelectResource()
  {
    viewMode = SubViewModes.Select;
    _resourceList = (await ProjectTracker.Library.ResourceList.GetResourceListAsync())
                      .Where(r => !vm.Model.Resources.Contains(r.Id)).ToList();
    StateHasChanged();
  }
```

This method changes the display mode to `Select` and it loads a list of resources from the data portal. The `Where` clause is used so the list won't include resources that are already assigned to the project.

Here is the markup for the `Select` UI mode:

```
    else if (viewMode == SubViewModes.Select)
    {
      @if (_resourceList == null)
      {
        <p>Loading resource list...</p>
      }
      else
      {
        <a href="javascript: void(0);"
          @onclick="ShowDefaultView">Cancel assignment</a>
        <table class="table">
          <thead>
            <tr>
              <th>Resource</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
            @foreach (var item in _resourceList)
            {
              <tr>
                <td>@item.Name</td>
                <td><a href="javascript: void(0);"
                  @onclick="() => AssignRole(item.Id)">Select</a></td>
              </tr>
            }
          </tbody>
        </table>
      }
    }
```

The resulting UI has a "Cancel assignment" link and a two column table listing the resource name and a "Select" action link.


Cancel assignment

Resource	
Lien, Meghan	Select
Bock, Jason	Select

If the user clicks the "Cancel assignment" link the `ShowDefaultView` method is called:

```
private void ShowDefaultView()
{
  if (selectedResource != null)
    selectedResource.CancelEdit();
  viewMode = SubViewModes.Default;
}
```

This resets the view to the default mode, so the user goes back to seeing the list of resources already assigned to the project. No new assignment occurs, and the `CancelEdit` method ensures that any changes the user might have made to this object's state are reset to a snapshot value.

If the user clicks the "Select" link on a listed resource, the `AssignRole` method is invoked, passing the selected item's `Id` property value as a parameter:

```
private async void AssignRole(int resourceId)
{
  selectedResource =
    (await ProjectTracker.Library.ProjectResourceEditCreator.
     GetProjectResourceEditCreatorAsync(resourceId)).Result;
  selectedResource.BeginEdit();
  viewMode = SubViewModes.Details;
  StateHasChanged();
}
```

The `AssignRole` method uses the `ProjectResourceEditCreator` type from the business library. This type implements a command that creates a `ProjectResourceEdit` object for a specific resource. This new object is the correct type to become a child object in the `ProjectEdit` object graph, but you can see that the code doesn't add it to the list of assigned resources.

Instead, the code changes the UI display mode to `Details` so the user can select the new resource's role on the project, and also choose to confirm assigning the resource (or not).

First though, it calls the `BeginEdit` method on the business object to take a snapshot of its current state. This is necessary because the user can choose to cancel the assignment operation later in the UI workflow.

Edit Resource Assignment Details

There are two ways the UI display mode can be set to `Details`. One is the result of the user selecting a resource to assign to the project. The other is if the user clicks the "Edit" action link on a resource that is already assigned to the project.

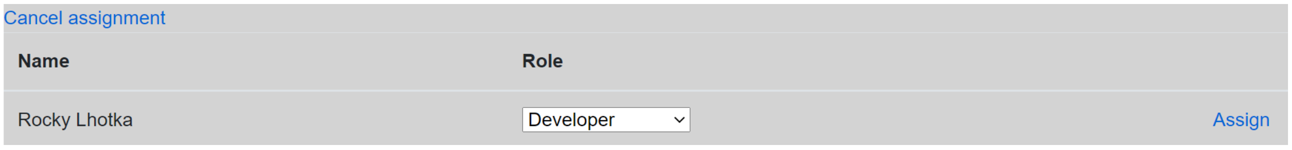
Clicking that "Edit" button invokes the `EditResource` method:

```
private void EditResource(int resourceId)
{
  selectedResource = vm.Model.Resources
    .Where(r => r.ResourceId == resourceId).FirstOrDefault();
  if (selectedResource != null)
  {
    selectedResource.BeginEdit();
    viewMode = SubViewModes.Details;
  }
}
```

Either way, the page's `selectedResource` field will be set to the `ProjectResourceEdit` object representing the resource the user selected, and that object will have had `BeginEdit` called to take a snapshot of the object's previous state. The UI will display the details about that assignment:

```
  @if (viewMode == SubViewModes.Details)
  {
    <a href="javascript: void(0);"
      @onclick="ShowDefaultView">Cancel assignment</a>
    <table class="table">
      <thead>
        <tr>
          <th>Name</th>
          <th>Role</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>@selectedResource.FirstName @selectedResource.LastName</td>
          <td>
            <select @bind="selectedResource.Role">
              @foreach
                (var item in ProjectTracker.Library.RoleList.GetCachedList())
              {
                <option value="@item.Key">@item.Value</option>
              }
            </select>
          </td>
          <td><a href="javascript: void(0);"
            @onclick="AddResource">Assign</a></td>
        </tr>
      </tbody>
    </table>
  }
```

In this display mode the user is shown the resource name, and is allowed to edit the role the resource will play on the project. There is also a "Cancel assignment" link to cancel any changes, and an "Assign" action link to confirm the changes.


Cancel assignment

Name	Role	
Rocky Lhotka	Developer	Assign

The business layer includes a read-only collection `RoleList` type that is used to get the list of possible roles. This list is normally cached in memory in the app, with the assumption being that the list of roles changes rarely.

The list of roles is used to create a dropdown combobox UI element by using the `select` and `option` elements.

If the user clicks the "Cancel assignment" link, the `ShowDefaultView` method is called. I discussed this method earlier, talking about how it changes the UI display mode back to `Default` without saving any changes made by the user.

In this case remember that the user could be in `Details` mode because they are adding or editing a resource assignment. In both cases, reverting to the `Default` UI mode without explicitly saving changes to the business object graph means that any changes are discarded.

If the user clicks the "Assign" link the `AddResource` method is invoked:

```
private void AddResource()
{
  selectedResource.ApplyEdit();
  if (!vm.Model.Resources.Contains(selectedResource.ResourceId))
    vm.Model.Resources.Add(selectedResource);
  ShowDefaultView();
}
```

This method calls the `ApplyEdit` method to accept any changes to the object's state made by the user.

It then checks to see if the list of assigned resources already contains the `selectedResource` item. If it does not, then the item is added to the collection.

Finally, the `ShowDefaultView` method is called to display the list of assigned resources to the user.

Remove a Resource Assignment

The final action the user can take with an assigned resource is to remove the assignment. In the `Default` UI mode there is a "Remove" action link on each row, and if the user clicks that link it calls a `RemoveResource` method:

```
private void RemoveResource(int resourceId)
{
  vm.Model.Resources.Remove(resourceId);
}
```

I chose not to implement any confirmation in this UI, so this method directly removes the selected item from the list of resources assigned to the project.

> I didn't confirm removal in this case, because no changes to the `ProjectEdit` object are committed to the database until the user clicks the "Save" button to save all changes.

At this point you should understand how the user is able to edit details about a project, and add, edit, or remove resources assigned to the project. All that remains now is the "Save" button so the user can save their changes to the server.

Save Button

The "Save" button is like the save or submit buttons I discussed in Chapter 7:

```
<button @onclick="vm.SaveAsync"
        disabled="@(!vm.Model.IsSavable)">Save</button>
```

The `@onclick` event is handled by the viewmodel object's `SaveAsync` method. Any errors that occur during that process are reflected by the viewmodel object's `ViewModelErrorText` property and so are displayed to the user at the top of the page.

The `disabled` property is data bound to the `IsSaveable` property of the `Model` object. This ensures that the "Save" button is only enabled if the `ProjectEdit` business object graph has no broken validation or authorization rules.

You should now be able to create pages for displaying read-only lists, editing simple editable business lists, and editing more complex master-child object graphs.

In the last section of this chapter I will discuss the implementation of authentication in the ProjectTracker apps.

Authentication

In Chapters 5 and 7 I discussed Blazor and CSLA .NET authentication.

As required by server-side and client-side Blazor, the `App.razor` component in the shared UI project wraps all content in a `CascadingAuthenticationState` element, and replaces the `RouteView` component with the `AuthorizeRouteView` component:

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

Remember that the `App.razor` component includes an `IsServerSide` property:

```
@code {
  public static bool IsServerSide { get; set; }
}
```

This property is set as the client-side or server-side app starts up so it is possible, within the code in the shared UI project, to know whether the code is hosted by ASP.NET Core or WebAssembly.

In the Shared folder of the shared UI project, the `MainLayout` component also includes authentication-related markup:

```
  <div class="top-row px-4">
    <AuthorizeView>
      <Authorized>
        Hello, @context.User.Identity.Name
        @if (App.IsServerSide)
        {
          <form method="post" action="/Account/Logout">
            <button type="submit" class="nav-link btn btn-link">Log out</button>
          </form>
        }
        else
        {
          <a href="Account/Logout">Log out</a>
        }
      </Authorized>
      <NotAuthorized>
        <a href="/Account/Login">Log in</a>
      </NotAuthorized>
    </AuthorizeView>
    <a href="https://cslanet.com" target="_blank">About</a>
  </div>
```

This markup uses the `AuthorizeView` component to display UI to allow the user to log in if they aren't logged in, or to log out if they are logged in.

Notice the use of the `IsServerSide` property in the `App.razor` component. The markup for server-side authentication is different from client-side authentication markup, and this property allows Blazor to render the appropriate content for each environment.

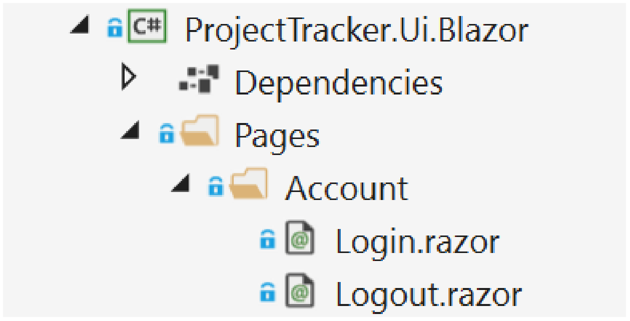
Because this markup is in the main layout, in the header section of the UI, it is available to the user on all pages within the app.

The configuration and pages necessary to implement authentication are different for client-side and server-side Blazor.

Client-side Authentication

The client-side Blazor ProjectTracker app follows the patterns from Chapters 5 and 7.

The shared UI project includes an Account folder in the Pages folder where the `Login` and `Logout` pages are located.



```
ProjectTracker.Ui.Blazor
  Dependencies
  Pages
    Account
      Login.razor
      Logout.razor
```

These pages are ignored for server-side Blazor, but are used to implement the authentication process for client-side Blazor.

Login Page

The `Login` page makes use of an editable business class named `Credentials` from the `ProjectTracker.Library` namespace to collect the user credentials. In this case a username and password. The page binds to this type using the same MVVM pattern I've been using throughout this chapter:

```
<table class="table">
  <thead>
    <tr>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <TextInputRow Property="vm.GetPropertyInfo(() => vm.Model.Username)" />
    <TextInputRow Property="vm.GetPropertyInfo(() => vm.Model.Password)"
                  InputType="password" />
  </tbody>
</table>
<button @onclick="LoginUser"
        disabled="@(!vm.Model.IsSavable)">Login</button>
```

When the user clicks the "Login" button the `LoginUser` method is called:

```
  private async void LoginUser()
  {
    var validator =
      await Csla.DataPortal.FetchAsync<CredentialValidator>(vm.Model);
    var principal = validator.GetPrincipal();

    if (principal.Identity.IsAuthenticated)
    {
      userService.CurrentUser = principal;
      NavigationManager.NavigateTo("/");
    }
    else
    {
      ErrorText = "Invalid credentials";
      StateHasChanged();
    }
  }
```

As I discussed in Chapter 7, this method calls the data portal to get a custom identity object. The `CredentialValidator` type is implemented in the business layer and is used by all the ProjectTracker UI apps. The `CredentialValidator` type includes a `GetPrincipal` method that creates a `ClaimsIdentity` and `ClaimsPrincipal` based on the user information:

```
    public ClaimsPrincipal GetPrincipal()
    {
      var identity = new ClaimsIdentity(AuthenticationType);
      if (!string.IsNullOrWhiteSpace(Name))
      {
        identity.AddClaim(new Claim(ClaimTypes.Name, Name));
        if (Roles != null)
          foreach (var item in Roles)
            identity.AddClaim(new Claim(ClaimTypes.Role, item));
      }
      return new ClaimsPrincipal(identity);
    }
```

The resulting principal object is used to set as current user identity by using the user service.

Logout Page

The `Logout` page has no visible UI at all. When the app navigates to this page the user is logged out and the current user principal is set to an unauthenticated `ClaimsPrincipal` object:

```
@page "/account/logout"
@inject Csla.Blazor.Client.Authentication.CslaUserService userService
@inject NavigationManager  NavigationManager

@code {
  protected override void OnInitialized()
  {
    userService.CurrentUser =
      new System.Security.Claims.ClaimsPrincipal();
    NavigationManager.NavigateTo("/");
  }
}
```

Once the current user is set to an unauthenticated identity, the app navigates to the home page.

Server-side Authentication

The server-side Blazor ProjectTracker app also follows the patterns from Chapters 5 and 7.

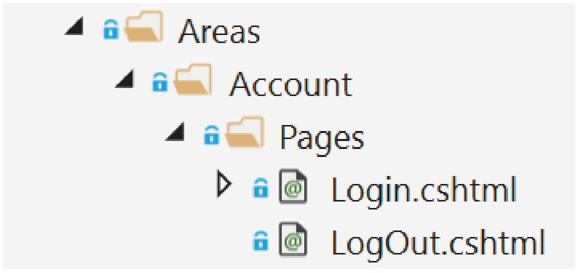
The server-side app is configured for authentication. In the `ConfigureServices` method of the `Startup` class, before any other methods are called, the `AddAuthorization` method is invoked:

```
    services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
      .AddCookie();
```

Then in the `Configure` method of the `Startup` class, before the `UseEndpoints` method, authentication and authorization are configured:

```
    app.UseAuthentication();
    app.UseAuthorization();
```

Server-side Blazor relies on ASP.NET Core Razor Pages on the server to authenticate the user. These pages are in the server-side Blazor UI project in the Areas\Account folder.



These pages follow the same model from Chapter 5, so I will discuss only highlights in this chapter.

Login Page

The `Login` page is bound to the same `Credentials` type from the shared business layer to collect the username and password credentials. The markup for the page binds to these properties:

```
  <form method="post">
    <div class="form-group">
      <label asp-for="credentials.Username"></label>
      <input asp-for="credentials.Username" class="form-control">
      <div class="invalid-feedback"></div>
    </div>
    <div class="form-group">
      <label asp-for="credentials.Password"></label>
      <input asp-for="credentials.Password" class="form-control" type="password">
      <div class="invalid-feedback"></div>
    </div>
    <button type="submit" class="btn btn-primary">Login</button>
  </form>
```

When the "Login" button is clicked the page's postback method is called:

```
    public async Task<IActionResult> OnPostAsync()
    {
      if (!ModelState.IsValid)
      {
        AlertMessage = "Form has validation errors.";
        return Page();
      }
      if (User.Identity.IsAuthenticated)
      {
        await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
      }

      var validator =
        await Csla.DataPortal.FetchAsync<CredentialValidator>(credentials);
      var principal = validator.GetPrincipal();

      Csla.ApplicationContext.User = principal;

      if (Csla.ApplicationContext.User.Identity.IsAuthenticated)
      {
        AuthenticationProperties authProperties = new AuthenticationProperties();
        await HttpContext.SignInAsync(
          CookieAuthenticationDefaults.AuthenticationScheme,
          principal,
          authProperties);

        string returnUrl = Url.Content("~/");
        return LocalRedirect(returnUrl);
      }
      else
      {
        AlertMessage = "Credentials could not be verified.";
        return Page();
      }
    }
```

If a user was already logged into the app, they are logged out:

```
      if (User.Identity.IsAuthenticated)
      {
        await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
      }
```

The next step is to use the data portal validate the credentials:

```
      var validator =
        await Csla.DataPortal.FetchAsync<CredentialValidator>(credentials);
```

Notice that this is the same technique used in the client-side Blazor app.

ASP.NET Core, like client-side Blazor, only understands the `ClaimsPrincipal` and `ClaimsIdentity` types. So the `GetPrincipal` method of the `CredentialValidator` object is used to create a new `ClaimsPrincipal`, which is then set as the current user:

```
  var identity = new ClaimsIdentity(validator.AuthenticationType);
  var principal = validator.GetPrincipal();

  Csla.ApplicationContext.User = principal;

  if (principal.Identity.IsAuthenticated)
  {
    AuthenticationProperties authProperties = new AuthenticationProperties();
    await HttpContext.SignInAsync(
      CookieAuthenticationDefaults.AuthenticationScheme,
      principal,
      authProperties);

    string returnUrl = Url.Content("~/");
    return LocalRedirect(returnUrl);
  }
```

When running in an ASP.NET Core environment, the `User` property of `Csla.ApplicationContext` relies on the ASP.NET Core `HttpContext` object to maintain the user identity. Setting the `User` property merely sets the underlying `HttpContext` principal.

If the new user identity is authenticated, the `SignInAsync` method of `HttpContext` is used to create a cookie to maintain the user identity between page requests.

The web site then navigates to the home page of the Blazor app.

Logout Page

As with the client-side Blazor implementation, the server-side `Logout` page has no visible display elements. When the app navigates to this page, the current user identity is set to an unauthenticated value:

```
@page
@using Microsoft.AspNetCore.Authentication
@using Microsoft.AspNetCore.Authentication.Cookies
@attribute [IgnoreAntiforgeryToken]
@functions {
  public async Task<IActionResult> OnPost()
  {
    if (User.Identity.IsAuthenticated)
    {
      await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
    }

    return Redirect("~/");
  }
}
```

The `SignOutAsync` method of the `HttpContext` object is used to remove the authenticated user cookie, meaning that the user is effectively logged out of the app.

The web site then navigates to the home page of the Blazor app.

At this point you should understand how to create a business app using CSLA .NET and Blazor.

Conclusion

In this chapter I have discussed how to create a business app using Blazor and CSLA .NET. This ProjectTracker sample supports server-side and client-side Blazor with a shared UI project containing all the Blazor components for both deployments.

The ProjectTracker app demonstrates how to create pages to display lists of data, edit simple lists of data, and to edit more complex master-detail relationships. You can also see how client-side and server-side authentication are implemented.

This book covers the use of Blazor to create server-side and client-side apps that make use of the Blazor UI framework, including its powerful data binding, event binding, and method binding mechanisms.

It also covers the use of the helper types in the `Csla.Blazor` namespace. These types are designed to make it easy to build Blazor apps that take full advantage of the validation, authorization, and other business rules provided by the CSLA .NET rules engine.

You have also seen how the CSLA .NET data portal provides a powerful abstraction, allowing you to deploy the same Blazor app server-side or client-side, ranging from 1-tier to 2-tier to n-tier deployments.

Blazor is an exciting new technology that seems designed to take the best possible advantage of all CSLA .NET has to offer!