Progress® Telerik® UI for Blazor

Blazor, A Beginners Guide

A quick-start guide to productivity with Blazor

Ed Charbeneau

EBOOK

BLAZOR, A BEGINNERS GUIDE

A quick start guide to productivity with Blazor

Ed Charbeneau

DEDICATION

This book is dedicated to the .NET community. Without your support there is no Developer Advocacy, a title I work hard to achieve, ensuring the products we touch are the best they can be.

COPYRIGHT

Cover designed by Progress Software

Ed Charbeneau
Visit my website at www.EdCharbeneau.com

Produced by Progress Software
Visit the Telerik UI for Blazor at www.telerik.com/blazor-ui

First Edition: March 2020
Ed Charbeneau

ABOUT THIS BOOK

Blazor, A Beginners Guide is intended for developers with some .NET experience. If you're coming from a non-.NET development background, you may learn some .NET basics along the way, but supplemental material around C# and .NET would be of great help to you.

The book begins with the author's own perspective on WebAssembly, why it's important to have choices in web development and offers some insight on why Blazor was created. You'll then step through an overview where you'll get familiar with the framework and technology stack. As the book progresses you will see working examples that teach core concepts broadly. Later in the book, topics that require a deeper knowledge are revisited to grasp the fine details so you can make informed decisions about how your apps and components are built.

The intent of the book's layout is to get you productive quickly. You should be able to navigate and create Blazor projects with ease as you read through the chapters versus having to complete the book before starting a project.

PREFACE

At the 2017 Microsoft MVP (**M**ost **V**aluable **P**rofessional) Summit I watched Steve Sanderson unveil his experimental framework to a room full of .NET developers. I watched the room fill with excitement and give rounds of standing applause. The excitement validated my own opinion that this was something .NET developers like myself have always wanted for the web. I wrote this book to help the .NET community quickly grasp what Blazor has to offer and share in the excitement we all felt at the MVP Summit.

Since that moment I have made many blog posts about the topic which have become outdated with each monthly release of Blazor. I have taken the best of those blog posts, reviewed, updated, and rewrote portions of them into the book along with a wealth of brand-new material. By compiling my best works into a single piece as an e-book, this single source of knowledge can be kept up to date with new releases.

WEBASSEMBLY

You Are Here

Web development has always been synonymous
with JavaScript development.
That is, until now.

Web development has always been synonymous with JavaScript development. That is, until now. A new form of web development is starting to emerge that promises an alternative to JavaScript, **WebAssembly**. As a software developer with many years of experience in web development, this new direction has captured my interest.

WebAssembly (**wasm**) is a binary instruction format for web browsers that is designed to provide a compilation target for high-level languages like C++. In 2017, Microsoft began experimenting with WebAssembly to bring .NET to the browser using the **Mono run-time**.

WEBASSEMBLY

Mono provides the plumbing for .NET libraries (.dll's) to run on WebAssembly. Running on top of Mono is Blazor, a **single-page web app (SPA)** framework built on .NET. The WebAssembly-Mono-Blazor stack has potential to enable web developers with a full stack .NET application platform that doesn't require writing JavaScript, furthermore it doesn't depend on the user to install any browser plugins.

Introducing the concept of a "non-JavaScript web" immediately brings forth questions and rightfully so.

> *What Does WebAssembly Provide That JavaScript or TypeScript Doesn't?*

My answer comes with a great amount of bias and opinion, and I feel it should as not all developers, projects, and tools are the same. For me the answer is clear, and short, "choice." Opening web development beyond JavaScript means choice and the freedom to choose not only JavaScript, or .NET, but an even wider array of options. More precisely and personally, I have the choice to develop a web application using tools and languages that I'm already using elsewhere.

npm and Webpack Alternatives

One of the benefits of opening the web to .NET is that we now have alternatives to npm and Webpack. As a long time, .NET developer, I'm greeting NuGet package manager and MSBuild with excitement. For me, these technologies are less problematic, more familiar, and far more productive. While nothing is ever perfect, my relationship with NuGet and MSBuild has been mostly positive.

At first this may come with the impression that npm and Webpack are somehow bad, and that I'm advocating to abandon those tools, but the opposite holds true. npm and Webpack are great tools and they will likely be around for quite some time. If your JavaScript tools work well for you and the apps you create, then this is a wonderful thing. Having a long history of experience with the web, I understand why npm and Webpack exist and appreciation for what they have and will accomplish.

Reduced Learning Curve

One thing that shocked me about Blazor is how genuinely simple it feels to use. Blazor combines the ease of the Razor markup syntax with other .NET concepts. It has borrowed the best patterns from popular JavaScript frameworks like Angular and React while leveraging Razor templates and providing parity with other .NET conventions. This combination of features allows for the reuse of skills in a way that was unavailable before. The same could be said for Node developers who use one language and familiar concepts in full stack JavaScript apps.

Interoperability

Using WebAssembly doesn't mean that JavaScript can be completely avoided. WebAssembly must currently be loaded by JavaScript. (Yes, I can hear the record-

scratch) The necessity for JavaScript doesn't stop there either. WebAssembly doesn't have access to any platform APIs. In order to access platform APIs JavaScript is required.

> *WebAssembly must currently be loaded by JavaScript. (Yes, I can hear the record-scratch)*

WebAssembly applications can make calls to JavaScript, providing a migration path for APIs that are beyond the reach of pure WebAssembly. This feature is used in the Blazor framework as well. Because Blazor is new, the Blazor interop allows developers to fall back on JavaScript when there are shortcomings of WebAssembly itself, or because the Blazor framework does not provide access. A block diagram of the application stack can be seen in **Figure 1.**

The interop is an abstraction layer that many developers will work with in C#, and they will not need to worry that underlying technology is still executing JavaScript code. Over time the need for abstractions will decrease as WebAssembly matures. The details of using the JavaScript interop will be covered in full later in the book with examples.

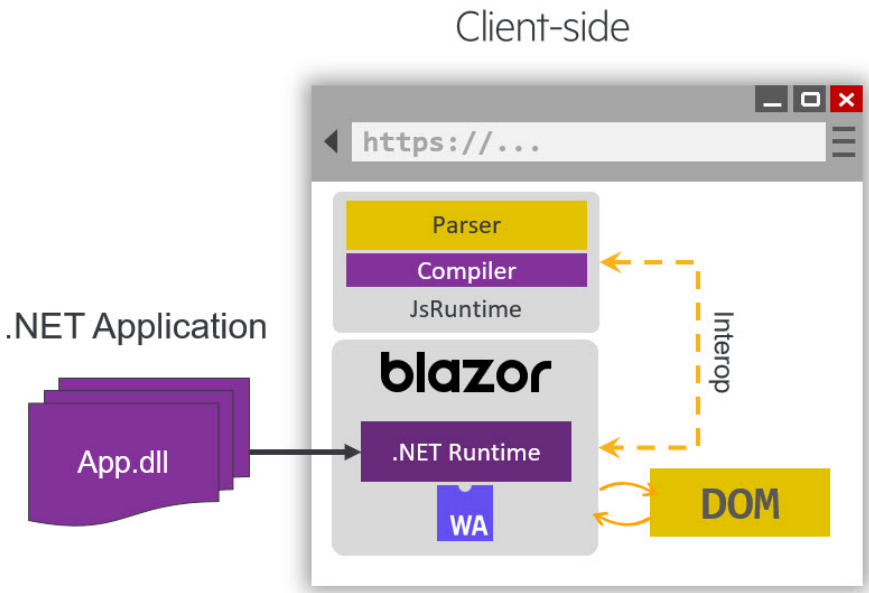


Figure 1: A block diagram of the Blazor framework and the web browser.

Moving Forward

If developing for the web using JavaScript alternatives is of interest to you, then WebAssembly and frameworks like Blazor are worth investing time in. These are still early days for WebAssembly and WebAssembly based technologies, but the promise of a widening ecosystem has gotten my attention. As a huge fan of web development, I want to see it move forward and expand ideas of how apps are written for the platform. The prospect of leaning on years of .NET experience to build apps in a way that makes me more productive is exciting to say the least. In addition, I have built a solid foundation of JavaScript skills as well, that I continue to grow each day. With this variety skills comes perspective and unique ways to solve problems as an engineer.

INTRODUCTION TO BLAZOR

In the previous chapter we learned the author's prospective on why Blazor is important to the web development ecosystem. In this chapter we'll shift our focus to learning the general concepts of the Blazor framework and the choices it gives us.

Challenging the status quo is Blazor, a new web framework that's not only powerful but productive too. Blazor uses .NET Core's architecture for common patterns and practices across the entire stack. Blazor leverages .NET Core's capability to run anywhere by supporting a client-side and server-side hosting model. This duality of Blazor provides choice while remaining flexible enough to switch between hosting modes easily. Blazor combines the ease of Razor with other .NET Core concepts like **dependency injection**, **configuration**, and **routing**. It has borrowed the best patterns from popular JavaScript frameworks like Angular and React while leveraging Razor templates and provided parity with other .NET conventions and tooling.

Razor Component Model

You'll find the Razor Component model very familiar to working with ASP.NET MVC or Razor Pages. Razor Components incorporates the Razor syntax with a new method of encapsulating markup into reusable components. With Razor Components you'll quickly create the building blocks of your application's user interface (UI). Components are .NET classes whose properties are component parameters; this basic system makes Razor Components easy to author. By breaking the component model down into three concepts: **directives**, **markup**, and **code**, we can understand how they are created.

While Razor Component, and *Blazor Component* are widely used interchangeably, the correct terminology is **Razor Component**. This is important when searching online as both terms are quite prevalent. However, throughout the book we will refer to components as Razor Components.

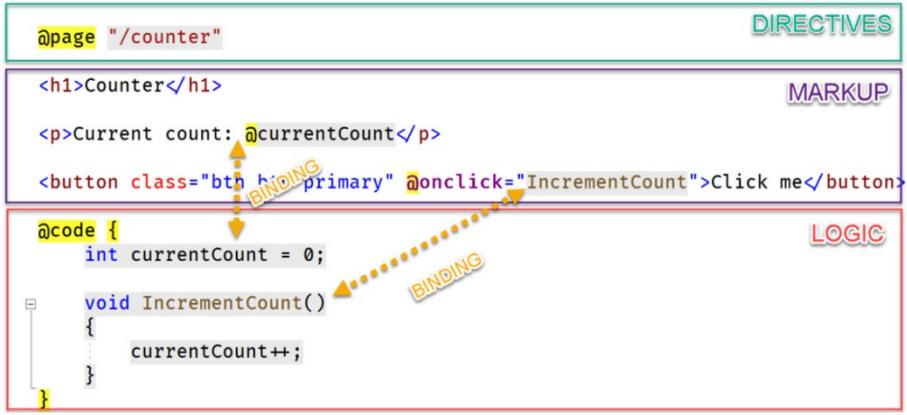


Figure 2: A breakdown of Razor Component, the most basic building block in a Blazor application.

In **Figure 2,** components use directives to add special functionality like routing or dependency injection. Syntax for directives is like what's used in ASP.NET MVC or Razor Pages. Component markup is primarily HTML which is enhanced through the Razor syntax. The Razor syntax allows C# to be used in-line with markup and can render values in the UI. The component's logic is written inside a *@code* block. This is where component parameters and data bound values are defined. Alternatively, code be referenced using a code-behind approach much like ASP.NET WebForms. We'll

cover the code-behind approach in a later chapter, but for now it's good to know the option exists. The Razor Component model allows us to break down our UI into manageable pieces. Components can then be assembled into larger UIs forming more complex components and pages.

The Tale of Two Blazors

Blazor initially started as a client-side single page application SPA framework.

Blazor initially started as a client-side SPA framework. The goal of Blazor is to run .NET applications in the browser using WebAssembly and the Razor syntax. This gave Blazor its name, a combination of *browser + Razor*. Over the course of development, the ASP.NET team added a server-side hosting model to Blazor. While each hosting model offers fundamentally different strengths, they both rely on the same underlying architecture. This approach enables developers to write most of their code independent of the hosting model. Ideally the only time code is can be identified as server or client centric is when data is being fetched. Let's take a closer look at the two hosting models to further understand their viability.

Client-Side: Blazor WebAssembly

On the client-side, Blazor is made possible by WebAssembly, this is referred to as a Blazor WebAssembly app. WebAssembly (wasm) is a binary instruction format for web browsers that is designed as a compilation target for high-level languages like C++. Blazor leverages this technology via the Mono runtime which is compiled to a WebAssembly module. As we can see in **Figure 3,** introducing the .NET runtime to the browser enables .NET libraries, dll's, to run directly on the client.

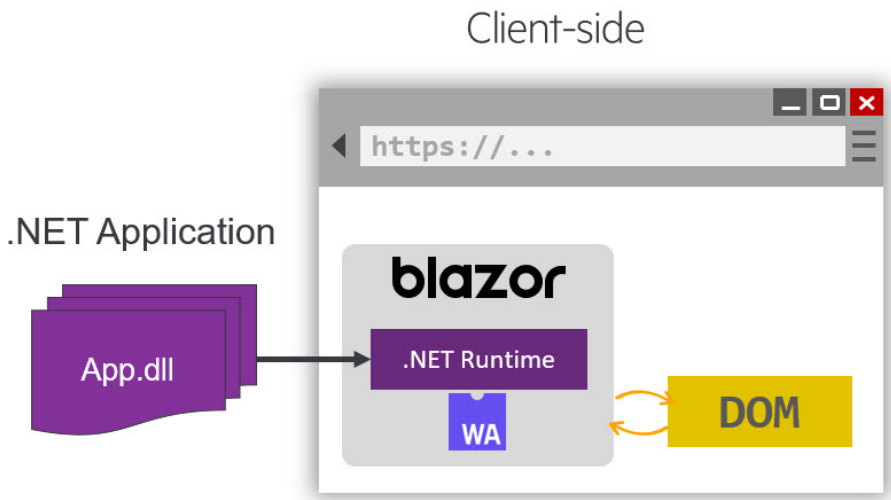


Figure 3: This diagram shows the .NET runtime inside the browser using WebAssembly. Blazor uses this runtime to work directly with standard .NET libraries.

There are tradeoffs with this approach in the way of performance and package size, but ultimately portability is the real strength here. Since Blazor applications use .NET libraries there is no need to recompile existing code or use special complier targeting. Application code can be shared across .NET projects resulting in writing less code. In a typical JavaScript front-end application, code for validation and data transfer is often duplicated because it is required in both .NET and JavaScript layers. This is not required in a Blazor application because both client and server code can reference the same libraries.

As with any client-side technology, Blazor WebAssembly apps rely on RESTful web services for data. Fetching data is done through **HttpClient**, the standard for the .NET ecosystem. When we make a http calls in Blazor the data is automatically serialized to the specified class object. Let's examine the *HttpClient's GetJsonAsync<T>* method call so we can understand its simplicity.

In a Web API application, we have a *WeatherForecast* controller with an endpoint that returns an IEnumerable of WeatherForecast. When the Get method is invoked ASP.NET automatically serializes the response into a JSON string.

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

```
{
[HttpGet]
public IEnumerable<WeatherForecast> Get() {…}
}
```

When fetching data from a Blazor application we create field to hold a *WeatherForecast* array. In this example we'll leverage the *OnInitializedAsync* lifetime method to populate the *forecasts* field. Using the *GetJsonAsync* we make a get request to the WeatherForecast controller. Because GetJsonAsync is called with the specified type *WeatherForecast[]*, it returns a serialized result of this type.

```
WeatherForecast[] forecasts;

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

Both the Web API and client application can use the same WeatherForecast class which results in less code, testing, and complexity.

We can further reduce complexity in our application by running Blazor server-side. When running server-side Blazor can use Entity Framework directly eliminating the need for a Web API application all together. We'll explore these options in later chapters once we become more familiar with the core concepts.

Blazor Server-Side with SignalR

A Blazor Server app takes a different approach building a .NET web application than Blazor WebAssembly. With a Blazor Server app, **instead of using WebAssembly** the browser is treated as a thin client. All the application code is running on the server using .NET Core runtime. To enable this thin-client operation, a light weight **SignalR** library is bootstrapped on the client that allows server code to send asynchronous updates using web sockets. Messages sent between the server and client only contain events and updates. As we can see in **Figure 4,** Blazor runs on the server and communicates with the browser over SignalR while the thin-client handles updates to the DOM (Document Object Model).

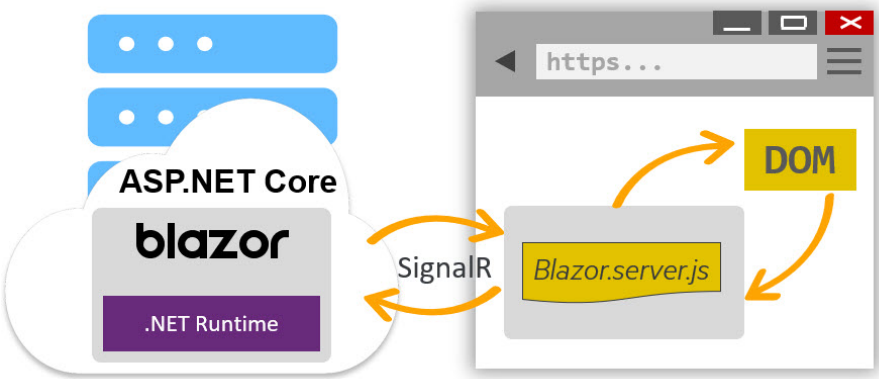


Figure 4: This diagram shows Blazor running server-side and interacting with the Browser using a SignalR connection.

Because the application code runs completely server-side the data layer may be tightly coupled without the need for a data layer. This approach is ideal for line-of-business applications where a persistent connection is available.

Blazor Server apps do not require a Web API service for communicating with a database or services when the data is on the same server. Instead of using HttpClient as with Blazor WebAssembly, we can instead inject services into the application for direct use. Ultimately Blazor Server applications require less abstractions and can be composed within a single project. Project size and scope can also affect the abstraction level, but it's nice to have the choice of staying uncomplicated.

Fetching data is a simple task from Blazor Server when using Entity Framework Core directly. We'll begin by creating a field which will hold our data. In this example we'll leverage the *OnInitializedAsync* lifetime method to populate the *blogs* field. Using the dbContext we make a call to *ToArrayAsync*. Because the application runs completely server side there's no Web API endpoint or concern for serialization and deserialization.

```
Blog[] blogs;

protected override async Task OnInitializedAsync()
{
    blogs = await dbContext.Blogs.ToArrayAsync();
}
```

Using Blazor gives you the choice of building a RESTful client-side application using .NET technologies instead of JavaScript, or a server-side .NET web application with reduced complexity.

Blazor Hosting Models: Comparing the strengths and tradeoffs Blazor hosting models.

Client-Side	Server-Side
+ Low-no server overhead	+ Tiny payload size
+ RESTful	+ Less abstraction
+ Offline & PWA capable	+ Pre-rendering by default
- Larger payload size	- Server overhead
- Disconnected environment	- Connection required

BLAZOR PROJECT TYPES

In the previous chapter, we discussed the duality of Blazor and the general concepts behind each hosting model. In this chapter we'll focus more on the project types that derive from the choices we're given by the framework.

So we all get started on the same page we'll begin with Blazor's prerequisites. Some of the tools we will be working with are installed by default with either the .NET SDK or Visual Studio IDE. However, some portions of Blazor are still in preview and require additional installation steps for the templates and code to function properly.

To get started with Blazor development you'll need the following installed on your machine:

1. The latest .NET Core 3.x SDK release.
2. The Blazor WebAssembly template
3. The latest Visual Studio 2019 Preview with the ASP.NET and web development workload.

For specific installation instructions it is best to follow the steps outlined in the official Microsoft documentation: Blazor.net This is because the docs can be kept current with the latest version numbers for each piece of software.

Once the required software is installed on our system, we're ready to explore the Blazor new project templates. To create a new Blazor application in Visual Studio, select *File* > *New Project* and choose **Blazor App** from the C*reate a new project* dialog.

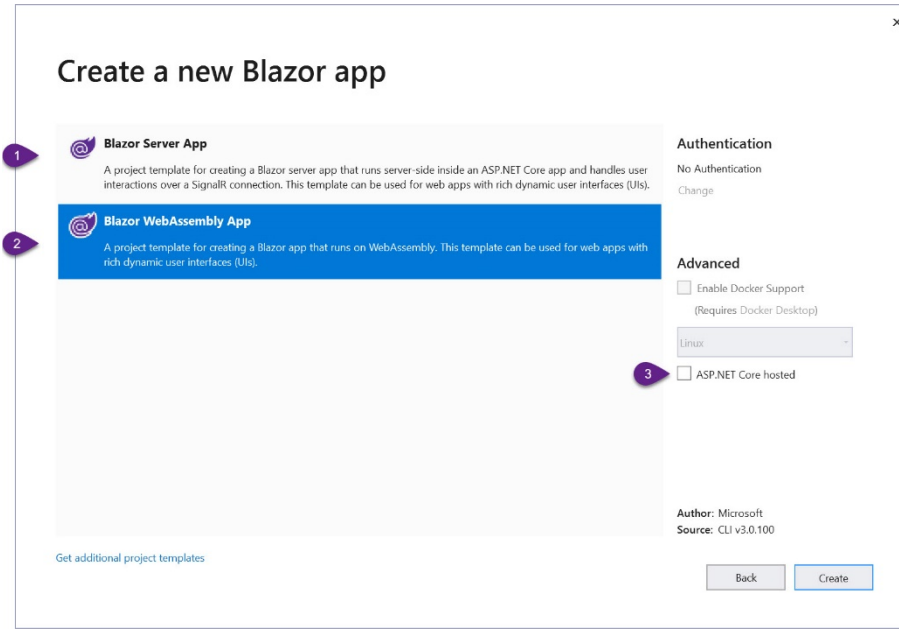


Figure 5: The create a new Blazor app dialog has three templates to choose from.

Let's breakdown the project types from the **Create a new Blazor app** dialog in **Figure 5**.

1. Blazor Server App, server-side
2. Blazor WebAssembly, client-side
3. Blazor WebAssembly, client-side with ASP.NET host and Web API (aka **Full-Stack**)

Now that we know where to find each project template, we can discuss the details of each selection.

Blazor WebAssembly

The Blazor client-side template can be compared to an HTML, CSS, and JavaScript application, like other SPA frameworks. However, in the case of Blazor, JavaScript is supplemented by .NET and C# via WebAssembly. Unlike ASP.NET MVC projects that

use Razor, Blazor (client-side) is **not** rendered on the server, instead the HTML is rendered to browser via the Blazor framework on the client. The browser processes everything in this project is a static resource.

Since the project can be treated as a set of static resources, it leads to some interesting possibilities that have yet to be seen in the ASP.NET ecosystem. Projects built with this template can be hosted on virtually any resource than can serve static files, for example: GitHub pages and Azure Storage: Static website hosting.

The template includes examples of how to get started with Blazor as well as basic page layouts, navigation controls, and styling from Bootstrap.

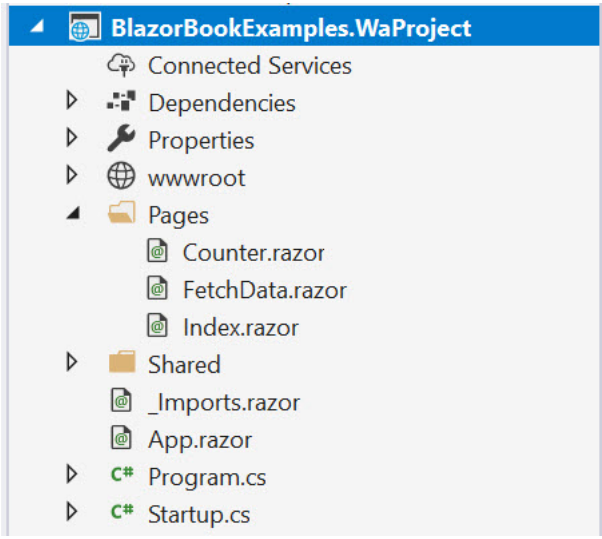


Figure 6: The solution contains a familiar project structure for ASP.NET applications with some Blazor specifics.

The project structure is simple with just the few resources outlined below:

- **/wwwroot**: web standard static resources including CSS, JavaScript, JSON, images, and HTML
- **/Pages**: Razor application pages/features
- **/Shared**: common components & page layouts
- **App.razor**: the root, a container component for the entire application
- **Program.cs**: application bootstrapping and configuration

At first glance, some familiar concepts may appear as Blazor uses an approach like ASP.NET apps, this can be seen in **Figure 6**. Not only are Program.cs a common feature in .NET apps, but Blazor also utilizes a similar concept to ASP.NET Core Razor Pages. Blazor application pages or features can be found under the Pages project path, while routing is handled by the page's @page directive.

```
@page "/myRoute"

<!-- component markup -->

@code {
// component logic
}
```

All Blazor projects include an Index, Counter, and Fetch Data example pages. These items are consistent throughout all project types, except for Fetch Data, as the **key difference between each project type is where the application is hosted in relation to the data it consumes.**

Let's begin by examining the Counter page and its component markup and logic.

Counter

The Counter page is a simple component decorated with the page directive. This component demonstrates the basic composition of a Blazor component including routing, data binding, and event binding/handling. Each portion of component composition is highlighted in **Figure 7**.

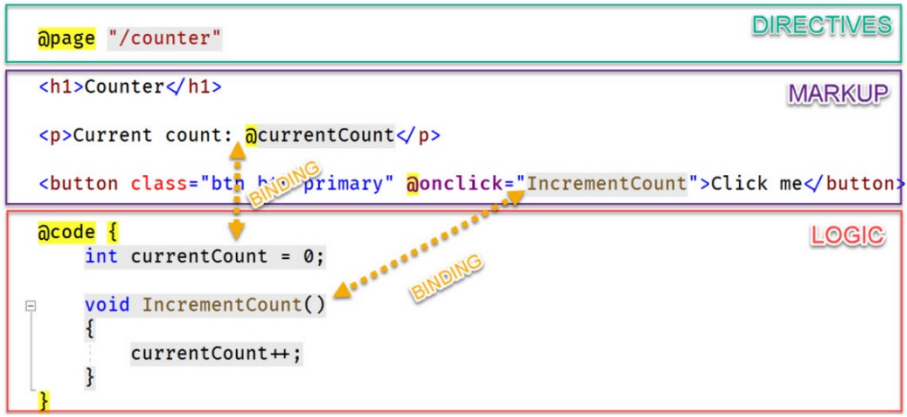


Figure 7: The composition of the counter component has directives, markup, and logic connected by data binding.

The counter component uses a basic HTML button to increment a counter field which is displayed within a paragraph tag. Because Blazor operates as a single page application all the interactions in the component happen on the client. Updates to the browser's DOM are handled by the Blazor framework though data binding. We can see the rendering of the counter component in **Figure 8**.

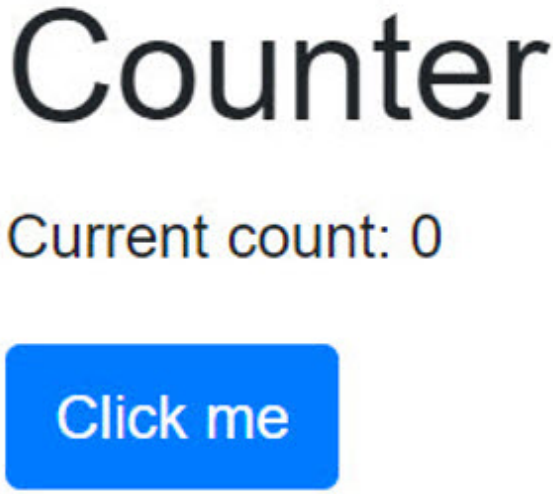


Figure 8: The counter component rendered in the browser.

Moving on to Fetch Data page, we'll see how Blazor is capable of handling local data sources.

Fetch Data

In the WebAssembly project type, the Fetch Data page is a component that utilizes data from a local static file. The Fetch Data component demonstrates dependency injection and basic Razor template concepts. This version of Fetch Data is very similar to the example found in the Full-Stack template except for the location in which the data is loaded from.

At the top of the component following the routing directive dependency injection is declared. The **@inject** directive instructs Blazor to resolve an instance of **HttpClient** to the variable **Http**. The HttpClient is then used by the components logic to fetch data using GetJsonAsync which binds data from the JSON request to an array of **WeatherForecast** objects:

```
@page "/fetchdata"
@inject HttpClient Http

<h1>Weather forecast</h1>
// omitted for brevity

@code {
  WeatherForecast[] forecasts;

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

Displaying the **WeatherForecast** data is done by iterating over the **forecasts** collection and binding the values to an HTML table:

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

No server is used or needed for these basic examples. If we plan to develop our application with hosting and web services, then the Full-Stack or Server-Side templates may be a better starting point.

Blazor Full-Stack

The Blazor **Full-Stack** template encompasses the same project structure as the Client-Side template with a few additions. Just like the Client-Side template there is no HTML rendered by the server and all files are delivered to the client as static files including .NET binaries. The difference however is added ASP.NET Core hosting and Web API and a Shared project for common application logic.

The template includes three projects: A Client-Side Blazor application **Blazor.Client**, an ASP.NET Core server application **Blazor.Server**, and a shared .NET Standard project for common application logic **Blazor.Shared**.

Blazor.Server

The server application is responsible for serving the application and providing web API endpionts for data. In **Startup.cs** we'll find the **MimeType** settings which configure the server to allow ***.wasm** and ***.dll** files to be served. In addition, compression is enabled to help reduce the size of binary files as they are transferred to the client. Compression is enabled through the **AddResponseCompression** middleware.

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

The Startup process also is where the Blazor application middleware is initialized by **app.UseClientSideBlazorFiles<Client.Startup>()**. This identifies the **Blazor.Client** application which is being served.

```
// This method gets called by the runtime. Use this method
to configure the HTTP request pipeline.

app.UseClientSideBlazorFiles<Client.Startup>();
```

The application comes with a simple example of a Web API controller and action. In the **WeatherForecastController**, a **WeatherForecasts** action generates a random set of weather forecasts. In the Full-Stack template, the **WeatherForecasts** web API replaces the static **weather.json** file found in the Client-Side template.

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    …
    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
     …
      return Enumerable.Range(1, 5)
```

```
        .Select(index => new WeatherForecast…)
        .ToArray();
    }
}
```

Blazor.Client

Nearly all the Client application is identical to that of the Client-Side template. However, the **FetchData** example differs slightly by requesting data from the **WeatherForcasts** web API endpoint in the GetJsonAsync method call.

```
@code {
WeatherForecast[] forecasts;

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

Blazor.Shared

Since the project includes a server and client solution that both use .NET it's possible to share code between both applications. This is a scenario unique to Blazor since the client is running .NET client side instead of JavaScript. The example given in the project template utilizes the same **WeatherForecast** class on both the server and client application.

```
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);
}
```

The WeatherForcast class is just a basic idea of shared code, however other shared application code may include validation, converters, and business logic which is decoupled from system, IO, or web concepts. Expanding on this idea further a theoretical application may share libraries between other .NET frameworks such as: Xamarin, Windows Desktop, or other .NET based web applications.

Server Rendering

In other .NET web technologies like ASP.NET MVC (and Core MVC), the Razor templates are rendered by the server and sent to the client as HTML. Some JavaScript frameworks like Angular and React share rendering responsibilities on both client and server in a process called ahead-of-time Compilation (AOT) or Isomorphic Rendering. Even though the Blazor client application is hosted on a .NET server, **all views are rendered client-side** in both the Client-Side and Full-Stack project templates. Currently, server pre-rendering can be enabled by heavily customizing the Full-Stack template.

Blazor Server App

The Blazor Server-Side project template takes a significantly different approach to how a Blazor application is delivered and interacts with the browser. When using the server-side configuration Blazor utilizes the browser as a "thin-client" by deploying a SignalR JavaScript application to the client. On the server, Blazor implements a SignalR hub communicating with the client via web sockets. In the server-side hosting model, Blazor is executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over the SignalR connection. In this configuration there is **no need for WebAssembly** and Blazor is executed on the ASP.NET Core runtime at the server. In **Figure 9,** we can see a block diagram of how the Blazor Server is implemented. All UI updates are sent as diffs, bidirectionally as binary packets over web sockets. To the user, the application is indistinguishable from any other web application.

> *In this configuration there is no need for WebAssembly and Blazor is executed on the ASP.NET Core runtime at the server.*

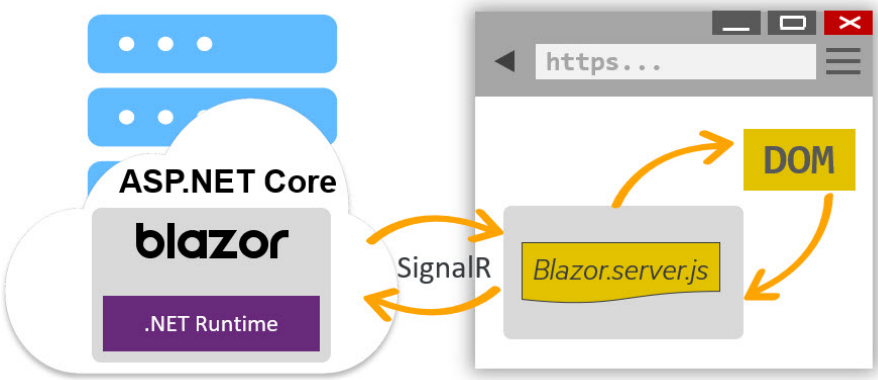


Figure 9: This diagram shows Blazor running server-side and interacting with the Browser using a SignalR connection.

Despite the drastic differences in how Blazor operates server-side, the actual application model stays relatively the same. In the server-side project template there are few differences in the example code provided by the template. The template includes a single project much like the Blazor WebAssembly app.

In a Blazor Server app the _host.cshtml file is the client entry point to the application. Unlike the Blazor WebAssembly application, the _Host.cshtml file is using Razor Pages to bootstrap the application. Therefore, we see the Razor syntax used on the page in the form of a Tag Helper, **<component ...>**. The component Tag Helper is invoking a Razor Component's rendering from within a Razor Page. This is where the root component of our application is invoked from **App.razor**. When configured for server-side operation the JavaScript file, **blazor.server.js** replaces **blazor.webassembly.js**. blazor.server.js invokes the SignalR client and establishes web socket communication with the server.

```
<body>
    <app>
        <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>
    …
    <script src="_framework/blazor.server.js"></script>
</body>
```

Because the entire application runs server side, in this project type the FetchData example utilizes data from a local service. The @inject directive in this example resolves an instance of WeatherForecastService in place of HttpClient as seen in the Full-Stack project template. The WeatherForecastService in the example is a simple class that generates random data, however in a real-world scenario the service could be an Entity Framework database context, repository, or other sources of data.

```
@using BlazorBookExamples.Data
@inject WeatherForecastService ForecastService
<h1>Weather forecast</h1>
// omitted for brevity

@code {
    WeatherForecast[] forecasts;

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

The WeatherForecastService and other services are added to the dependency injection container in the ConfigureServices method found in Startup.cs.

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

Server

The server project provided by the template is a simple ASP.NET Core host. In the Startup.cs of this project, the UseEndpoints middleware is invoked and maps the Blazor SignalR hub to the correct path.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
```

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

Pros and Cons

Since only a small amount of JavaScript is required to bootstrap the client and no .NET assemblies are transferred to the client the Server-Side Blazor application is efficient with network traffic. Even during operation network traffic is light because communication between the browser "thin-client" and the server is a small binary packet. However, because web sockets are used, the client and server must always be connected. This means that Blazor Server-Side apps cannot work in offline mode.

Blazor Project Types, Conclusion

Each project type includes a similar set of examples with the Counter and Fetch Data components. The Fetch Data example varies from project to project to showcase specific features of that project type.

With the Blazor WebAssembly, Full-Stack, and Blazor Server app project templates developers can choose the starting point that best fits their application's requirements. The Blazor WebAssembly template focuses on running static files completely in the browser, while the Full-Stack template includes ASP.NET Core hosting and Web API. Using the Blazor Server template utilizes the Blazor framework on the server and relies on SignalR in place of WebAssembly thus trading performance for a dependency on always being connected.

BLAZOR, COMPONENTS BASICS

In the previous chapter we covered the various project types used to create a new project. In this chapter we'll get better acquainted with how components are written.

As we briefly discussed in the previous chapter **Introduction to Blazor**, the Blazor framework uses the Razor Component model. We'll learn more about Razor Components in this chapter as we build a component to display a weather forecast. Through this process we'll learn the fundamentals of component construction including **directives**, **parameters**, **child content/templates**, **routing** and **event handling**. For simplicity we'll use the resources given to us by the new **Blazor Server** app template.

Directives

Compiler directives are used to give Razor instructions that typically change the way a component is parsed or enables different functionality. Let's create a new page to display a weekly weather forecast. In the Blazor framework pages are components which are decorated with the **@page** directive. The @page directive is followed by a string value which defines the component's page route giving it a URL which can be navigated to in the browser, ex: @page "/my-url". Even though the @page directive enables routing, it does not change our ability to use the component as a basic building block for other components or "pages".

Let's begin by adding a new Razor Component to our application called WeeklyForecast. In the project choose the Pages folder and right click, choose Add > New Item, then select Razor Component as shown in **Figure 10**. Name the component **WeeklyForecast.razor**, then click create to finish.

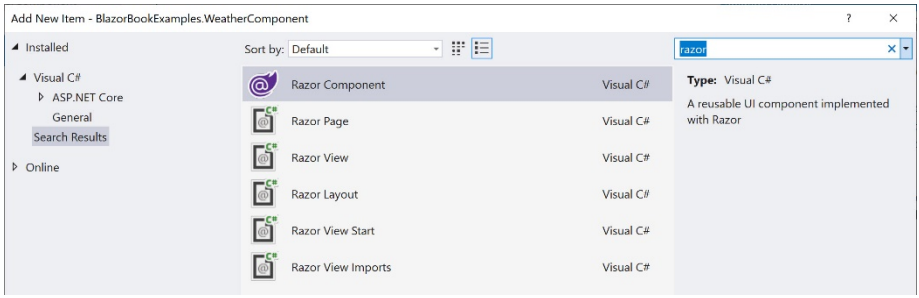


Figure 10: Creating a new Razor Component with the Add New Item dialog.

Our component is created with a very basic boilerplate. We'll leave this boilerplate in place and add our @page directive with the value *"/weeklyforecast"* to the very top of the file. It is best practice to place all directives at the top of the file unless there is a special use case where it should to be placed elsewhere. For example, the @code block is a directive which is normally located at the bottom of the component, however moving it will not cause an error.

```
@page "/weeklyforecast"
<h3>WeeklyForecast</h3>

@code {

}
```

We can run our application and view the changes in the browser. Once the app is running append */weeklyforecast* to navigate to the newly created page. The results should resemble **Figure 11**.

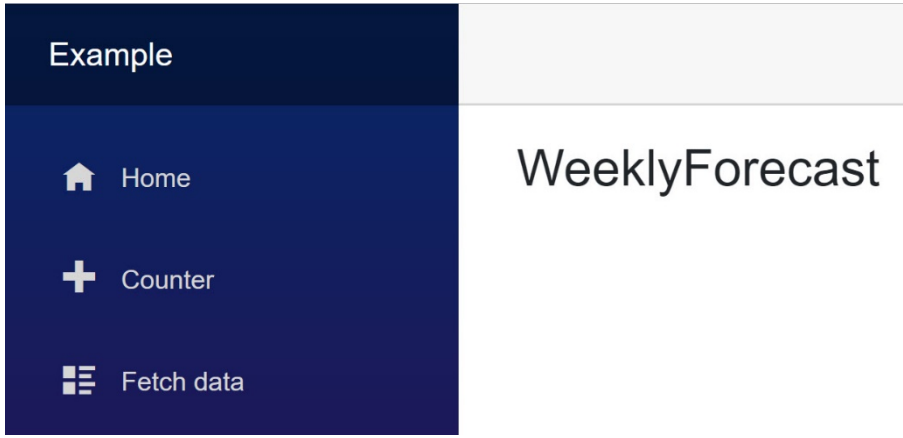


Figure 11: The WeeklyForecast component using the @page directive to specify routing.

Let's use this page to build a prototype of what our final product should look like. This will help us decide where components can be abstracted into reusable bits. We'll display weather information to the user using HTML and CSS classes from Bootstrap and Iconic.

Inside the WeeklyForecast page, we'll append our first prototype UI element, a day of weather forecast data. We'll use hard coded values, which will eventually be replaced with data. The markup is contained within a div element which uses the Bootstrap **card** class to give a nice appearance. Inside the card's body, we're using a span element with the Open Iconic classes **io io-rain**, this will render an icon which represents the weather status

```
@page "/weeklyforecast"

<h3>WeeklyForecast</h3>
<div class="card bg-light" style="width: 18rem;">
    <div class="card-body text-center">
        <span class="h1 oi oi-rain"></span>
        <h1 class="card-title">17 C&deg;</h1>
        <p class="card-text">
               Rainy weather expected Monday
         </p>
    </div>
</div>
…
```

We'll save these changes and reload the WeeklyForecast page which now shows the weather card. The rendered page should look like **Figure 12**.

Figure 12: A prototype of the weekly forecast page displaying a single weather forecast.

Our page is taking shape; however, we're building a weekly forecast and we currently have a single day. Let's modify the page so it can display five days of forecast data. Since we'll be repeating the cards, we'll wrap the display in a flex-box container. Since we're using Bootstrap, we can use the convenience class **d-flex.** Inside the *d-flex* container we need to repeat our weather display. We'll rely on Razor to help us repeat the display with a **foreach** loop. To create a simple loop without any real data

we'll use the **Enumerable.Range** method to generate a sequence for us. With multiple cards being displayed, let's remove the static width style attribute *style="width: 18rem;"* and let the element expand as needed. To create some whitespace between each card the m-2, or margin 2 CSS class is added.

```
…
<div class="d-flex">
    @foreach (var item in Enumerable.Range(1, 5))
    {
        <div class="card bg-light m-2">
            <div class="card-body text-center">
                <span class="h1 oi oi-rain"></span>
                <h1 class="card-title">17 C&deg;</h1>
                <p class="card-text">
                       Rainy weather expected Monday
                   </p>
            </div>
        </div>
    }
</div>
…
```

The result is a five-day forecast that repeats across the page as shown in **Figure 13**.

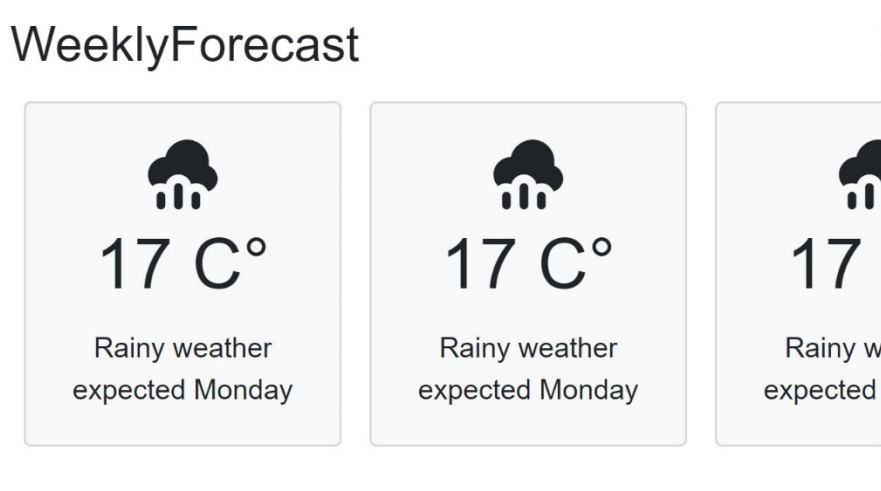


Figure 13: The weekly weather page shows a repeated weather forecast card with five static elements across the page.

By now you can probably see a pattern emerge where the individual day of weather is repeated. This repeated section could easily be a reusable component that encapsulates a day of weather. Let's abstract our component out of the WeeklyForecast page into its own component file.

In the project's /Shared folder we'll create a new component called WeatherDay. We'll remove the boilerplate content from the newly created component and move the HTML from inside of the *foreach* loop of the WeeklyForecast component to the WeatherDay component. This quick copy/past job is all we need to create a basic Razor Component. With the markup moved to the WeatherDay component, we can make use of it by name in WeeklyForecast. We'll add a WeatherDay component to body of the *foreach* loop.

/Shared/ WeatherDay.razor

```
<div class="card bg-light m-2">
    <div class="card-body text-center">
        <span class="h1 oi oi-rain"></span>
        <h1 class="card-title">17 C&deg;</h1>
        <p class="card-text">
               Rainy weather expected Monday
        </p>
    </div>
</div>
```

/Pages/WeeklyForecast.razor

```
…
<div class="d-flex">
    @foreach (var item in Enumerable.Range(1, 5))
    {
        <WeatherDay></WeatherDay>
    }
</div>
…
```

When we re-run the page, it should be identical to the previous run as seen in **Figure 13**. There were no changes to the result as we simply moved the responsibility for rendering the card from WeeklyForecast to WeatherDay.

Parameters

Let's replace some of the static values with component parameters. The easy way to understand component parameters in the Razor Component model is to see them as public properties that are decorated with a special [Parameter] attribute. We'll

discuss additional parameter attributes as we go however the standard [Parameter] attribute will fit most scenarios.

Now that our component's HTML is isolated inside the WeatherDay component we can make our component more dynamic by allowing it to accept data using parameters. For our weather component we need to show the weather **summary**, i.e. is it Rainy, Sunny, or Cloudy with a corresponding icon. We'll also display the **temperature** and **day of the week**. Let's create a few parameters to get started and implement more detailed features as we work through each challenge. We'll begin with an @code block and add our three properties and decorate them using the [Parameter] attribute.

```
@code {

    [Parameter]
    public string Summary { get; set; }

    [Parameter]
    public int TemperatureC { get; set; }

    [Parameter]
    public DayOfWeek DayOfWeek { get; set; }

}
```

One-Way Data Binding

To display the parameter's value, we'll use one-way databinding. This will simply write the value inline where the value appears in the markup. In Razor we simply prefix the properties name with an @ symbol, ex: @Summary. For each parameter we'll update the markup by replacing the static text with a data bound value. When Razor generates the final output, all values are automatically converted to a string. There is no need to manually covert a value like *TemperatureC* even though it is an *int*, however we do have the option of customizing the formatting if we wish.

```
<div class="card bg-light m-2">
    <div class="card-body text-center">
        <span class="h1 oi oi-rain"></span>
        <h1 class="card-title">@TemperatureC C&deg;</h1>
        <p class="card-text">
                @Summary weather expected @DayOfWeek
        </p>
    </div>
</div>
```

Now that our component has parameters that are data bound, we can return to the WeeklyForecast page and update our component instance to make use of the new feature. We'll locate the WeatherDay component on the page and set some static values for our parameters.

```
@foreach (var item in Enumerable.Range(1, 5))
{
    <WeatherDay
        TemperatureC="20"
        Summary="Cloudy"
        DayOfWeek="DayOfWeek.Friday">
    </WeatherDay>
}
```

Let's continue by adding some dynamic data to our WeeklyForecast page. We'll inject a WeatherForecastService on the page and use it to generate data. The WeatherForecastService is included in the new project template and is located under the /Data folder in the project. Open the WeatherForecastService.cs file and examine how it works. We can see there is a collection of Summaries that are used to fill the Summary property. The method, GetForecastAsync generates 5 random WeatherForecast values based on a start date.

Let's simplify the Summaries list to only include the values: **Cloudy**, **Rainy**, and **Sunny**, this will correspond to three icons available in our project.

```
private static readonly string[] Summaries = new[]
{
 //"Freezing", "Bracing", "Chilly", "Cool", "Mild",
 //"Warm", "Balmy", "Hot", "Sweltering", "Scorching"
 "Cloudy", "Rainy", "Sunny"
};
```

In the WeatherDay we'll use a read-only property to determine which icon to show based on the Summary field. A quick ternary statement converts the Summary string to the corresponding icon name.

```
string IconCssClass =>
        Summary == "Cloudy" ? "cloud" :
        Summary == "Rainy" ? "rain" :
        "sun";
```

Now we can use the IconCssClass property to modify the CSS value of the component's icon.

```
<span class="h1 oi oi-@IconCssClass"></span>
```

Injecting a Service

The WeatherDay component can now be bound to dynamic data. We'll inject the WeatherForecastService on the page using the @inject directive. The dependency is available because the WeatherForecastService was already registered in Startup.cs by the template. We can see how dependency injection (**DI**) configured by looking in the **ConfigureServices** method. We'll discuss the various DI methods in the later chapter dedicated to this topic.

```
services.AddSingleton<WeatherForecastService>();
```

On the WeeklyForecast page, beneath the @page directive we add @inject and specify both the Type (**T**) that will be resolved and the variable name an instance of that type will be assigned, ex: **@inject T myTInstance.** An @using directive will bring the WeatherForecastService into scope.

```
@page "/weeklyforecast"
@using Data;
@inject WeatherForecastService WeatherService
```

OnInitializedAsync Lifecycle Method

With the WeatherForecastService available we can now call the *GetForecastAsync* method which generates an array of WeatherForecast objects. The data should to be loaded when the component is first initialized. Razor Components handle initialization through the **OnInitializedAsync** and **OnInitialized** lifecycle methods. The component lifecycle methods are automatically inherited from the ComponentBase class that makes up all Razor Components.

It's best practice to use asynchronous code whenever possible as it leads to an overall better user experience. In addition, our WeatherForecastService provides async functionality, so we'll be using OnInitializedAsync to fetch data as the component initializes.

Let's add a field to capture the data that will be displayed on the page. We'll create an empty array of WeatherForecast called *forecasts*. Following the forecasts field, we'll override the OnInitializedAsync method and populate the forecast field by awaiting the call to GetForecastAsync.

```
WeatherForecast[] forecasts;

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

Now that we're using data from our WeatherService we'll need to replace the static loop in our view with one that uses the *forecasts* array. Inside the new loop we reference the current item and apply the corresponding property value to the parameter of the component. *The .NET compiler is smart, so we'll only need to use the @ symbol on string values to identify string literals from property values.*

```
@*foreach (var item in Enumerable.Range(1, 5))*@
@foreach (var forecast in forecasts)
{
     <WeatherDay
         TemperatureC="forecast.TemperatureC"
         Summary="@forecast.Summary"
         DayOfWeek="forecast.Date.DayOfWeek">
     </WeatherDay>
}
```

Since we're no longer working with static data, we should consider the possibility of null values. The component will render before OnInitializedAsync executes and after, therefore if the component renders while the *forecasts* array is null, a NullReferenceException will be thrown when entering the foreach loop. To safeguard against this, we use an if/then statement to provide a display when no data is present and prevent the exception.

```
@if (forecasts == null)
{
    <span>No Data</span>
}
else
{
    foreach (var forecast in forecasts)
    {
        <WeatherDay TemperatureC="forecast.TemperatureC"
               Summary="@forecast.Summary"
               DayOfWeek="forecast.Date.DayOfWeek">
        </WeatherDay>
    }
}
```

We can now render data dynamically by visiting the *weeklyforecast* route as shown in **Figure 14.**

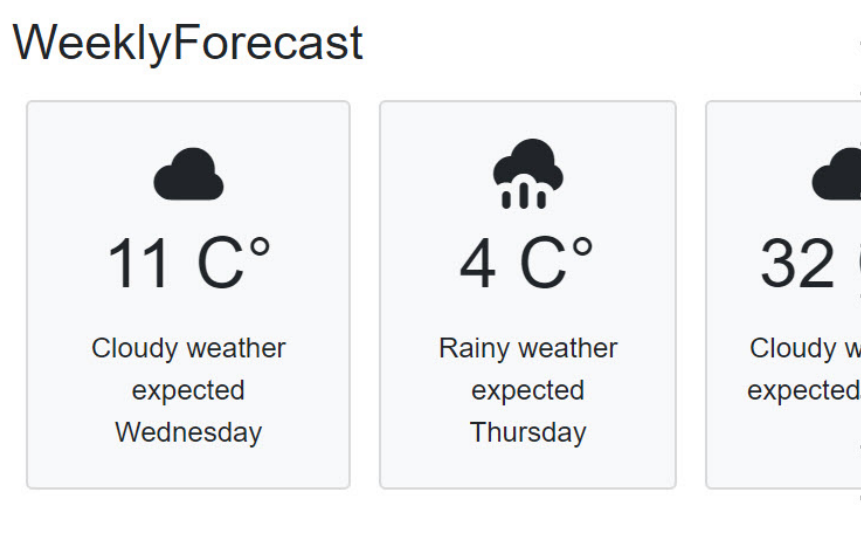


Figure 14: The weekly weather page shows a repeated weather forecast card with five static elements across the page.

Our WeeklyForecast and WeatherDay components are starting to come together. We're able to dynamically populate components based on a collection of data which is loaded as the component initializes. The data flows from the WeeklyForecast directly to each WeatherDay component through one-way data binding. This is a common way displaying data and works well when we have a consistent UI we're creating.

Let's expand on our WeatherDay component and allow its UI to be extended further by providing a template region.

Child Content/Templates

Templated components are components that accept one or more UI templates as parameters. Templates can be used customize a portion of the components rendered output. We'll continue with our WeatherDay component as an example of how to implement a template.

To make our WeatherDay component more flexible we'll be adding a template region shown as a red block in **Figure 15**. This area will allow developers to insert any HTML, Razor, or Component as child content.

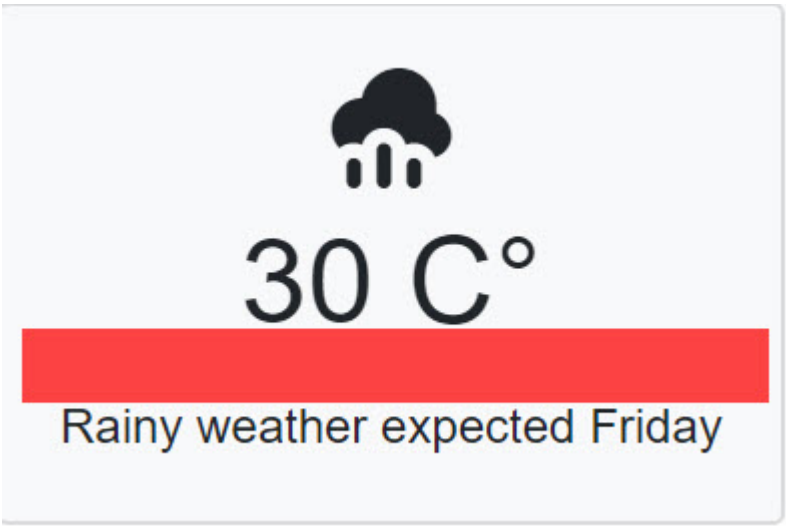


Figure 15: A red box outlines where the template region of the component will be created.

To add a template to our WeatherDay component we'll make use of the **RenderFragment** class. A RenderFragment represents a segment of UI content, implemented as a delegate that writes the content to a RenderTreeBuilder (the Blazor virtual DOM). On the surface one might assume the child content is written as raw HTML however, the RenderFragment conforms to the component architecture of Blazor.

> *On the surface one might assume the child content is written as raw HTML however, the RenderFragment conforms to the component architecture of Blazor.*

Let's add a RenderFragment to our WeatherDay component. The RenderFragment is added exactly like any other component parameter using a property and [Parameter] attribute. For this example, we'll call the property *CustomMessage*.

```
@code {

    [Parameter]
    public RenderFragment CustomMessage { get; set; }

    [Parameter]
    public string Summary { get; set; }
…

}
```

To make use of the parameter, we reference the RenderFragment when we would like it to appear in our component markup. In between the *h1* and *p* elements of our component the *@CustomMessage* is placed, this is where the template will be rendered.

```
<div class="card bg-light m-2">
    <div class="card-body text-center">
        <span class="h1 oi oi-@IconCssClass"></span>
        <h1 class="card-title">@TemperatureC C&deg;</h1>
        @CustomMessage
        <p class="card-text">
              @Summary weather expected @DayOfWeek</p>
    </div>
</div>
```

When using the WeatherDay component the template is applied by specifying the template by name using an HTML tag, *<CustomMessage>*. Inside this tag we can use Razor markup, HTML, other components, or any combination of these things. Let's apply this to our example by adding a simple *if* statement to check for a *Rainy* forecast and then display a special alert on those items. Inside the *CustomMessage* we'll add some Razor code and basic element styled with a Bootstrap *alert alert-danger* CSS class.

```
<WeatherDay TemperatureC="forecast.TemperatureC"
              Summary="@forecast.Summary"
              DayOfWeek="forecast.Date.DayOfWeek">
    <CustomMessage>
        @if (forecast.Summary == "Rainy")
        {
            <div class="alert alert-danger">
                Tornado Warning!
            </div>
        }
    </CustomMessage>
```

```
</WeatherDay>
```

When we run the application and view the *weekyforecast* page, it now displays a "Tornado Warning!" alert box inside of the WeatherDay component as seen in **Figure 16**.

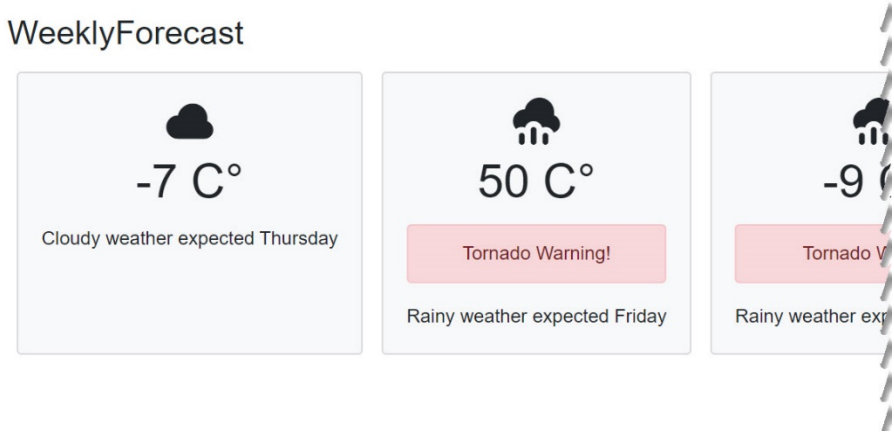


Figure 16: The weekly forecast is rendered with a Tornado warning displayed in the template region of the WeatherDay component.

Thus far, the WeeklyForecast and WeatherDay components have most of the features we commonly find in component architecture. However, one important aspect of UI development hasn't been discussed yet, **interactivity**. Next, we'll look at how to handle events with Blazor by giving users the ability to select an item shown in the weekly forecast.

Event Handling

When we think about events and delegates in C#, it's likely that we'll consider the: **event** and **delegate** keywords, or the **Action** and **Func** delegate-types. These are technically valid options when working with standard C# assemblies. Working with the UI in a Blazor application using Razor Components introduces a different architecture pattern and a special delegate called the **EventCallback**.

The EventCallback is a delegate-type used to expose events across components. A parent component can assign a callback method to a child component's EventCallback.

When using the EventCallback in this manner, the parent and child components are automatically re-rendered when events are raised. Using event, delegate, Action, or Func, may result in the UI not updating as expected. For this reason, the EventCallback should always be used in Razor Components.

> *The EventCallback is a delegate-type used to expose events across components.*

To demonstrate how EventCallback is used we'll continue with the WeeklyForecast and WeatherDay components. We'll add the ability for a WeatherDay to be selected from the weekly forecast when the users click on a given day.

We'll begin by adding a new parameter to our WeatherDay component named OnSelected. The OnSelected parameter should be an EventCallback of DayOfWeek, this means the event will return a type of DayOfWeek as an argument. The DayOfWeek is a value we can use to identify which item triggered OnSelected.

```
[Parameter]
public EventCallback<DayOfWeek> OnSelected { get; set; }
```

Beneath the OnSelected parameter an event handler is created, this handler is responsible for raising the OnSelected event delegate on the parent component. To invoke OnSelected we call InvokeAsync and pass the current DayOfWeek as an argument.

```
void HandleOnSelected()
{
    OnSelected.InvokeAsync(this.DayOfWeek);
}
```

Continuing with the WeatherDay logic a property to Selected property is added to indicate if the component is in a selected state. We'll use this property not only to bind the selected value to the component, but to also set the corresponding CSS for the UI. Using a private property SelectedCss we determine which CSS class to apply based on the value of the Selected property. If the item is selected, the *bg-primary text-white* class is used to highlight the component and invert the text color, otherwise the value will default to *bg-light*.

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

```
private string SelectedCss => Selected ?
                "bg-primary text-white" : "bg-light";
```

In our component markup we can now trigger the HandleOnSelected event delegate when the user clicks the UI. On the outermost *div* element we'll attach an *onclick* event handler and assign it to the HandleOnSelected method. We'll also modify the *div* to use the SelectedCss value effectively toggling the UI's appearance.

```
<div class="card m-2 @SelectedCss"
                @onclick="HandleOnSelected">
    <div class="card-body text-center">
        …
    </div>
</div>
```

The WeatherDay component can now be selected and we can apply the new feature with a few modifications to the WeatherForecast class and WeeklyForecast component. Since the WeatherForecast provides the data and state for our view, we'll add a Selected property which we can then bind to in our components.

```
public class WeatherForecast
{
       public bool Selected { get; set; }
     …
```

In the WeeklyForecast component an event handler is added for the OnSelected event. The method HandleItemSelected will accept the argument DayOfWeek which we can then use to set identify which item was selected. In the method a quick loop over each forecast is used to clear the selected value. Once cleared we can use a LINQ statement to find the matching DayOfWeek and toggle set the selected value to true.

```
void HandleItemSelected(DayOfWeek selectedValue)
{
    // Clear selections
    foreach (var item in forecasts)
        item.Selected = false;

    forecasts.First(f =>
    f.Date.DayOfWeek == selectedValue).Selected = true;
}
```

In the WeeklyForecast the markup receives a few updates to tie everything together. Where the WeatherDay component is used, the new properties are applied. The OnSelected method is given the HandleItemSelected delegate and the Selected value is bound to the *forecast* item's Selected property.

```
<WeatherDay TemperatureC="forecast.TemperatureC"
            Summary="@forecast.Summary"
            DayOfWeek="forecast.Date.DayOfWeek"
            OnSelected="HandleItemSelected"
            Selected="forecast.Selected">
```

With the updates complete we can now run the application and navigate to the *weeklyforecast* page and select items from the view as shown in **Figure 17**.

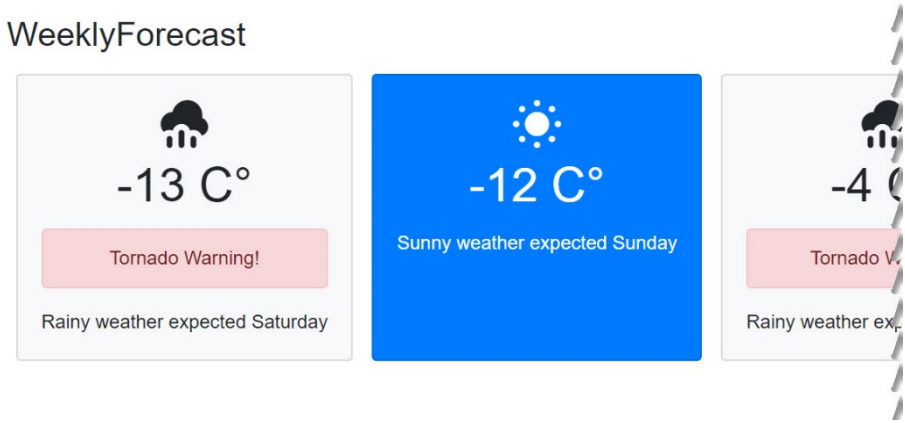


Figure 17: The second item in the list is selected when clicked by the user.

We could explain upon this concept further by adding any custom logic to the HandleItemSelected method. Imagine what features you might add when selecting an item, some examples include: displaying the hourly weather forecast, listing closed businesses or suggesting activities for the given forecast. This concept can be applied to many business applications where data includes parent-child relationships.

Blazor Component Basics, Conclusion

Throughout this chapter we learned the fundamentals of component construction such as **directives**, **parameters**, **child content/templates**, **routing** and **event handling**. Each example guided us through the process of creating a component while introducing topics that we will revisit in greater detail throughout the book. The intention of this chapter was to provide information that will immediately make us productive when using the Blazor framework.

CODE BEHIND APPROACH

In the previous chapter we learned about component architecture. In this chapter we'll separate our component markup from logic using partial classes.

In this chapter we'll learn when it's helpful to use a code-behind approach in your Blazor development, how we benefit and what to expect when re-factoring existing components.

When working with Blazor the default component architecture is to have all of the markup and logic mixed within a single file (.razor). When components are simple this approach works well, however as the complexity becomes greater managing the entirety of a component becomes difficult. Using a "code-behind" approach allows for markup and logic to be separated into their own files. In this article we'll investigate when it's helpful to use a code-behind, how we benefit and what to expect when re-factoring existing components.

Single File Approach

Let's begin with a quick review of the single file approach. We'll use Blazor's default template's FetchData component as an example which is shown in **Figure 18**. The FetchData component displays an HTML table of weather forecast data by looping over data provided by a service. The component uses directives, HTML/Razor, and a @code block; these are all typical items that make up a component.

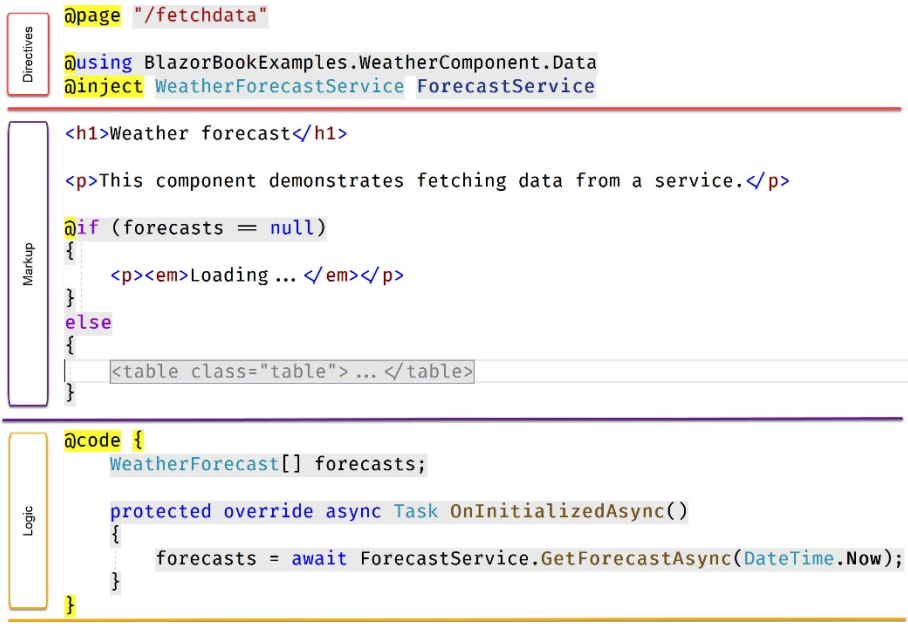


Figure 18: The FetchData component with a single file approach.

The markup for the component is quite verbose. Weighing in at 30+ lines the markup itself extends the length of the file, causing us to scroll down to find the components logic represented by a @code block. In **Figure 19** we can see there is nearly 40 lines of markup before we reach the component's logic.

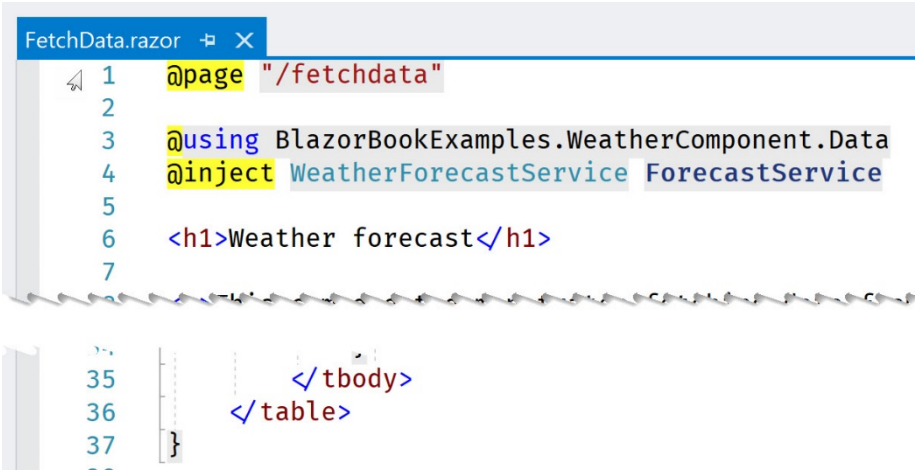


Figure 19: The markup section of the FetchData component spans 37 lines.

Keep in mind that this component is still actually quite simple. As the complexity becomes greater each section will grow. This means we will eventually have more directives, using statements, markup, and logic in our code block. While it may be convenient to have an all-inclusive file, it will eventually become tedious to maintain due to constant scrolling between markup and logic.

Let's re-factor the FetchData component by using a code-behind file see how it improves the overall developer experience.

Code-Behind Approach

Code-behind is a common term for the technique of using a separate code file to that represents all the logic for a corresponding page, view, or component. Creating a code-behind in Blazor requires a few steps, but thankfully it's supported by the framework, so setup is quite simple. To complete our code-behind we'll need to create a class and then link the code-behind to our markup. Once we have the structure in place, we can move over our existing logic.

Creating and Linking a Code-

Behind

First, we'll need a class file that will represent our code-behind. This is a standard .NET partial class; however, naming is very important. When we create our code-behind file we need to consider both the file name and class name as they can greatly affect our developer experience.

Let's start with the file name. When we add the file name, we should follow a common convention of [componentName].razor.cs. The .razor.cs extension is understood by Visual Studio and will be properly nested within the file explorer window as seen in **Figure 20** Since the component already occupies a class name, we'll need to declare the new class as a partial class using the *partial* keyword.

```
public partial class FetchData { }
```

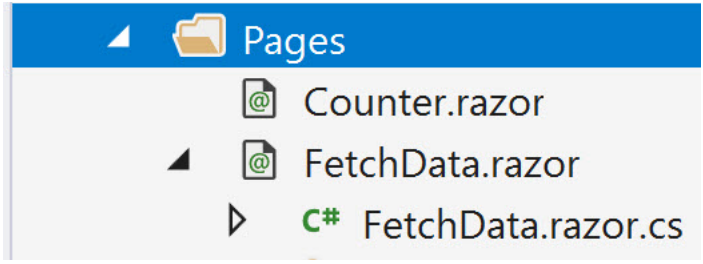


Figure 20: Visual Studio nests files with the same name and suffix with the .razor.cs pattern.

We can now migrate all the code from the @code block to our code-behind, and a few directives as well.

Migrating @code to Code-Behind

Most of the logic from our @code block can be copied directly as-is to the code-behind without change, however there are some small updates that will need to be made.

As we move code to our code-behind, additional using statements may be required if the code-behind doesn't automatically pick up a namespace. This is where standard tooling comes in handy with [ctrl]+[.] making short work of any missing namespaces.

Simply highlight any item with a red underscore and press the [ctrl]+[.] key combination, if it's a missing namespace, then accept the fix and the change will be applied.

```
// added by [ctrl] + [.]
using BlazorBookExamples.WeatherComponent.Data;
using Microsoft.AspNetCore.Components;
```

With all the code migrated, we still need to tackle one final step, dependency injection (DI).

Dependency Injection in Blazor Code-Behind

Dependency Injection in Razor markup is handled by the @inject directive. While the directive is available in code-behind classes, it is written much differently and at first glance may not be obvious. To use dependency injection in a code-behind file we'll use the [Inject] attribute and apply it to a property of the type we would like to inject, ex: [Inject] MyType MyProp { get; set; }.

```
[Inject]
WeatherForecastService ForecastService { get; set; }
```

With the [Inject] attribute added the component can now resolve the service and we no longer need the directive in the Razor markup. We can proceed with cleaning up our markup by removing the @inject and @using directives as well as the @code block.

Code-Behind Final Thoughts

When we started the component's directives, markup and logic were all contained in a single file. While a single file approach provides benefits in its simplicity, and is great for small components, it simply doesn't scale well. Once our component requires complex markup or logic it's easy to get lost in the large file and/or confused by two different ways of expressing code. In addition, we lose access to some of Visual Studio's most valuable productivity tooling.

The process of converting an existing component to use a code-behind is quite easy. A few code changes are required with regards to using statements and

dependency injection. For the most part Visual Studio even guides us through making the proper corrections to achieve a successful compilation.

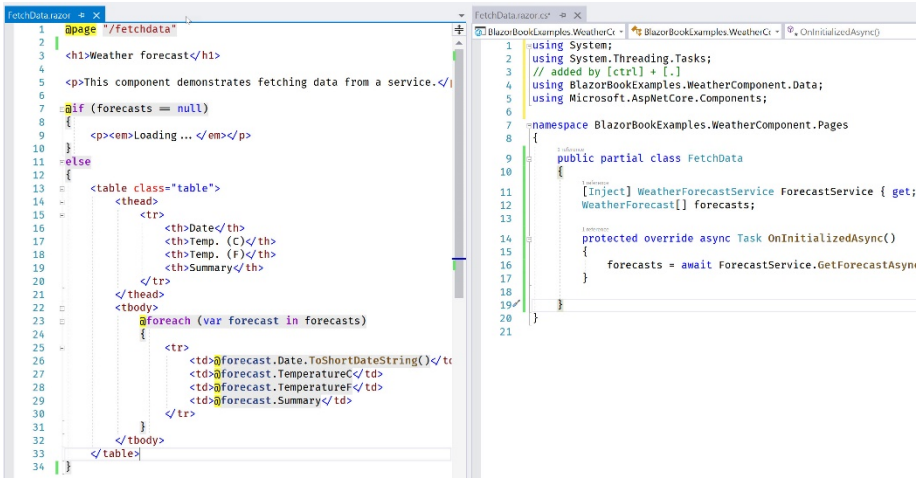


Figure 21: A side-by-side view of the FetchData component, markup on the left and logic on the right.

When the transition is complete, we benefit from clearly defined roles for markup and logic with a lot less scrolling. If we need to see the component in its entirety, we can now snap the two windows side-by-side and focus on the task at hand as shown in **Figure 21**.

DATA BINDING

The previous chapters introduced us to Razor syntax. In this chapter we'll get an understanding of how to work with data binding, learn how binding is defined, and when binding events are triggered.

Data binding is the flow of data between one or more components and variables. One-way binding is useful when we need to represent data in our UI. It is particularly useful in displaying information for read-only scenarios. Two-way data binding is typically used to trigger several components to update their state based on other components in the system. Therefore, two-way data binding is generally used with input components where one input's value is directly impacted by the value of another.

To get a better understanding of how data binding is handled in Blazor, we'll setup a simple example. We'll create a conversion tool which uses two HTML inputs to perform a conversion to or from inches and centimeters. We'll build the example in a series of steps with each step revealing a core concept of data binding. **Figure 22** shows the completed example that we will be discussing in the next steps.

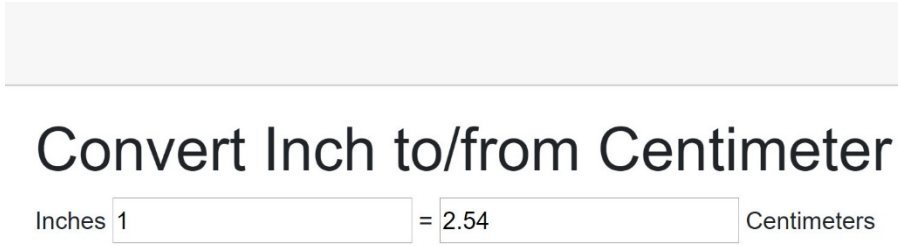


Figure 22: In the example, either input may be used to convert a value to inches or centimeters. Both inputs are kept in sync through two-way data binding, ex: 1 in = 2.54 cm.

Let's create a new component for this set of examples. Under the Pages folder add a component called *Converter* and set the page route to match, ex: *@page "/converter"*. The first test is to see the default behaviors of the @ directive when used on an HTML element. We'll begin by add an HTML *input* to the component and setting its value to *1* and type to *number*.

```
<input value="1" type="number" />
```

Running the application at this point will produce a rather uninteresting page. The page has an input that defaults to 1 and allows us to set any numeric value. While not terribly functional it provides context to the next example.

One-Way Binding

Let's use one-way data binding to set the numeric values of a component. We'll add a second HTML element, a paragraph *p* tag that displays the text "The value of inches is: 1".

```
<input value="1" type="number" />
<p>The value of inches is: 1</p>
```

After the markup a `@code` block is added where we'll write some logic for the component. Inside the code block add a field of the type *double* with the name *inches*. We'll set the default value of the inches field to 1.

```
@code {
```

```
        double inches = 1; // default value
}
```

With the *inches* field available, let's return to the markup section and use it to replace the value of *1* on both the input and the paragraph text.

```
<input value="@inches"  type="number" />

<p>The value of inches is: @inches</p>

@code {
    double inches = 1; // default value
}
```

Take a moment to imagine how this component will operate, what will the initial state look like and what will happen when we enter a new number in the input. At first glance we can determine that initially the input will hold the value 1 and the text will read "The value of inches is 1". What might come as a surprise is what happens when a new number is entered in the input box. When the input value is changed, only the input's **value** is changed, and not the value of *inches.* When we write value="@inches", value is set to the number currently held in inches, just as if we set it manually to 1 like we did in the previous example. Another way of reasoning about it is to say we're **not** setting value as a reference to the *inches* field. This read-only output is considered one-way data binding.

When we write value="@inches", value is set to the number currently held in inches, just as if we set it manually to 1 like we did in the previous example. Another way of reasoning about it is to say we're ***not*** *setting value as a reference to the inches field.*

Binding Events

To successfully update both the *inches* field and the text output when the user changes the input's value, we will need to use an event. By handling the input's *onchange* event we can run some business logic to update the inches field. We'll start by creating an event handler called *UpdateValue.* The event handler will take the ChangeEventArgs which contains the value set by the user. We can use the event arguments to then set the inches field with the new value. Since inches is a double value type, we'll need to add some safeguards and parse the event argument properly.

```
void UpdateValue(ChangeEventArgs e)
{
    double val = 0; // Failing to parse will set to 0
    double.TryParse(e.Value.ToString(), out val);
    inches = val;
}
```

In the markup we can now assign the UpdateValue delegate to the onchange event of the input.

```
<input value="@inches"
        @onchange="UpdateValue"
        type="number" />
```

Everything is in place to keep the input value, inches field, and text display syncronized. When the user changes the input value the UpdateValue method is invoked causing the backing field to update. Because on change is an EventCallback it triggers the UI to re-render the text output resulting in the expected behavior.

Two-way Binding

Using the input's value property and manually handling the onchange event is quite useful, however there is an easier way. Let's continue with the example using two-way binding by making use of the *@bind* directive. What's interesting is our implementation thus far is very close to how @bind works internally. When @bind is called on an input by default, the *value* property is set and updated. In addition, the onchange event is the default event which triggers data binding. Blazor is also doing extra work to ensure that types are converted automatically.

On the input component in our example we can replace the *value* property with *@bind*. Since @bind will also handle the onchange event and perform any necessary type conversions we can also remove the *@onchange* directive and *UpdateValue* handler. The result is all three values, the inches field, the input value, and the display of inches, this can be seen in **Figure 23:**

```
<input type="number" @bind="inches" />

<p>The value of inches is: @inches</p>

@code {
    double inches = 1; // default value
}
```

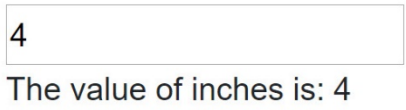


Figure 23: When the value of the input changes, the backing field updates and "The value of inches" is re-rendered.

With a simple two-way binding example in place, let's expand further by adding a second input for centimeters. The centimeters input is nearly identical to the inches input. We'll use a field named *centimeters* to hold a value initialized at *2.54* and add a number input bound to the *centimeters* field.

```
<input type="number" @bind="inches" />

<input type="number" @bind="centimeters" />

@code {
    double inches = 1; // default value
    double centimeters = 2.54; // default value
}
```

Each input is still independently bound to their respective field. We'll need to add some business logic to perform the actual conversion. Since @bind internalizes the event handler, we'll need to do work by leveraging C# properties.

Let's add a property called Inches and then change the binding from the inches field, to the Inches property. The property Inches will simply return the backing field when we **get** the value. When **set** the value of Inches the conversion is applied by multiplying the value by 2.54 and both *centimeters* and *inches* are updated. When the values are updated by the set method data binding will ensure the changes are reflected in the UI.

```
double inches = 1; // default value
public double Inches
{
    get => inches;
    set
    {
        centimeters = value * 2.54;
        inches = value;
    }
}
```

We repeat the same logic for centimeters by adding a property and applying a conversion when setting inches field. Additional HTML elements such as labels are added to complete the UI.

```
@page "/converter"

<h1>Convert Inch to/from Centimeter</h1>

<label>Inches</label>
<input type="number" @bind="Inches" />

<span> = </span>

<input type="number" @bind="Centimeters" />
<label>Centimeters</label>

@code {

    double inches = 1; // default value
    double centimeters = 2.54; // default value

    public double Inches
    {
        get => inches;
        set
        {
            centimeters = value * 2.54;
            inches = value;
        }
    }

    public double Centimeters
    {
        get => centimeters;
        set
        {
            inches = value / 2.54;
            centimeters = value;
        }
    }
}
```

We have completed the conversion tool which was shown in **Figure 22**. The inputs are fully data bound with the values all syncronized when the user changes either value.

Advanced Two-Way Binding

The conversion component is complete, it now updates when either input is **changed**. We've reached our goal but there's still room for improvement. The component currently requires an onchange event to trigger before data binding takes place. This means updates occur when the element loses focus as we click or tab away from the input. Instead of the default behavior of onchange, let's update the example so it binds oninput, which occurs immediately as we type. To change the default event bindings to make use of the oninput event, we'll use the *@bind:event* directive and specify the event name "oninput".

```
<input type="number" @bind="Inches"
                     @bind:event="oninput" />

<input type="number" @bind="Centimeters"
                     @bind:event="oninput" />
```

The UI will now update when as the user types in the input or changes the value in any way.

The binding directive is quite flexible and can be configured to specific properties as well. For example, we can set the default property and the corresponding event using the syntax: *@bind-propertyName @bind-propertyName:event="eventName"*. The following example has the same effect as before but shows how to bind to the *value* property explicitly.

```
<input type="number" @bind-value="Inches"
                     @bind-value:event="oninput" />

<input type="number" @bind-value="Centimeters"
                     @bind-value:event="oninput" />
```

Binding Conclusion

Data in Blazor is not just convenient but also flexible. Whether we are using one-way or two-way binding, there is a complete control over bind attribute and the event which triggers an update. This simplicity of data binding reduces the need manually written boilerplate code. When two-way binding is used in conjunction with C# property getters and setters, we can fine tune how the values are handled when they're bound.

BLAZOR RENDERTREE EXPLAINED

Blazor is built upon a DOM abstraction called a RenderTree. In this chapter we'll learn about what exactly a DOM abstraction is, what the RenderTree is used for, and why Blazor developers should know about it.

Abstracting the Document Object Model (DOM) sound intimidating and complex, however with modern web application it has become the normal. The primary reason is that updating what has rendered in the browser is a computationally intensive task and DOM abstractions are used to intermediate between the application and browser to reduce how much of the screen is re-rendered. In order to truly understand the impact Blazor's RenderTree has on the application we need to first review the basics.

Let's begin with a quick definition of the Document Object Model. A Document Object Model or **the DOM is a platform and language agnostic interface** that treats an XML or HTML document as a tree structure. In the **DOM's tree structure**, each node is an object that makes up part of the document. This means the DOM is a document structure with a logical tree.

> *... **the DOM is a platform and language agnostic interface** that treats an XML or HTML document as a tree structure.*

When a web application is loaded into the browser a JavaScript DOM is created. This tree of objects acts as the interface between JavaScript and the actual document in the browser. When we build dynamic web applications or single page applications (SPAs) with JavaScript we use the DOMs API surface. When we use the DOM for creating, updating and deleting HTML elements, modifying CSS and other attributes, this is known as DOM manipulation. In addition to manipulating the DOM, we can also use it and create and respond to events.

In the following code sample, we have a basic web page with two elements, an *h1* and *p*. When the document is loaded by the browser a DOM is created representing the elements from the HTML. We can see in **figure 24** a representation of what the DOM looks like as nodes in a tree.

```
<!DOCTYPE html>
<html>
<body>
   <h1>Hello World</h1>
   <p id="beta">This is a sample document.</p>
</body>
</html>
```

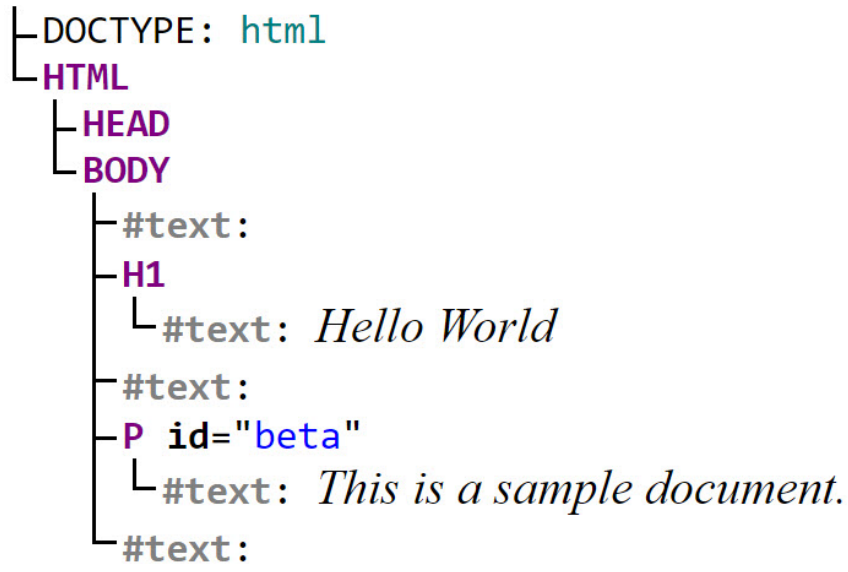


Figure 24: An HTML document is loaded as a tree of nodes; each object represents an element in the DOM.

Using JavaScript we can traverse the DOM explicitly by referencing the objects in the tree. Starting with the root node *document* we can traverse the objects children until we reach a desired object or property. For example, we can get the second child off the body branch by calling *document.body.children[1]* and then retrieve the *innerText* value as a property.

```
document.body.children[1].innerText
"This is a sample document."
```

An easier way to retrieve the same element is to use a function that will search the DOM for a specific query. Several convenience methods exist to query the DOM by various selectors. For example, we can retrieve the *p* element by its id *beta* using the getElementById function.

```
document.getElementById("beta").innerText
"This is a sample document."
```

Throughout the history of the web, frameworks have made working with the DOM easier. jQuery is a framework that has an extensive API built around DOM manipulation. In the following example we'll once again retrieve the text from the *p* element. Using jQuery's *$* method we can easily find the element by the id attribute and access the *text*.

```
//jQuery
$("#beta").text()
"This is a sample document."
```

jQuery's strength is convenience methods which reduce the amount of code required to find and manipulate objects. However, the biggest drawback to this approach is **inefficient handling of updates** due to directly changing elements in the DOM. Since direct DOM manipulation is a computationally expensive task, it should be performed with a bit of caution.

> *Since direct DOM manipulation is a computationally expensive task, it should be performed with a bit of caution.*

It's common practice in most applications to perform several operations which update the DOM. Using a typical JavaScript or jQuery approach we may remove a node from the tree and replace it with some new content. When elements are updated in this way, elements and their children can often be removed and replaced when no

change needed. In the following example, several similar elements are removed using a wildcard selector, *n-elements.* The elements are then replaced, even if they only needed modification. As we can see in **figure 25**, many elements are removed and replaced while only two required updates.

```
// 1 = initial state
$("n-elements").remove() // 2-3
$("blue").append(modifiedElement1) // 4
$("green").append(modifiedElement2) // 4
$("orange").append(modifiedElement3) // 4
```

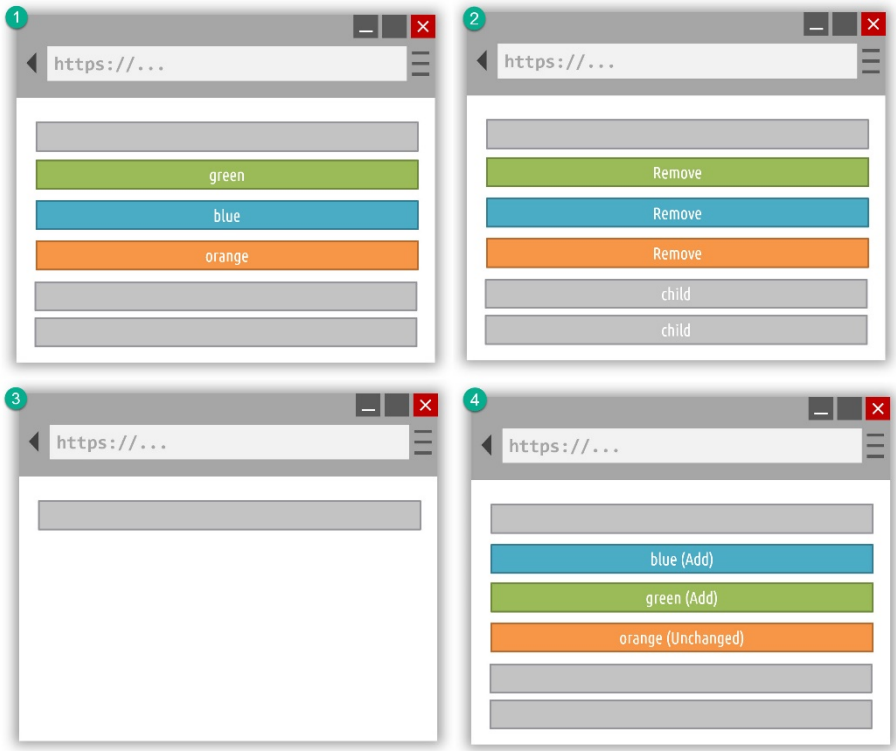


Figure 25: 1) The initial state; 2) Like elements are selected for removal 3) Elements and their children are removed from the DOM 4) All elements are replaced with only some receiving changes.

In a Blazor application we are not responsible for making changes to the DOM. Instead, Blazor uses an abstraction layer between the DOM and the application code we write. Blazor's DOM abstraction is called the RenderTree and is a lightweight copy

of the DOM's state. The RenderTree can updated more efficiently than the DOM and reconciles multiple changes into a single DOM update. To maximize effectiveness the RenderTree uses a diffing algorithm to to ensure it only updates the necessary elements in the browser's DOM.

If we multiple updates on elements in a Blazor application within the same scope of work, the DOM will only receive the changes produced by the final difference produced. When we perform work a new copy of the RenderTree is created from the changes either through code or data binding. When the component is ready to re-render the current state is compared to the new state and a diff is produced. Only the difference values are applied to the DOM during the update.

Let's look take a closer look at how the RenderTree can potentially reduce DOM updates. In **figure 26** we begin with the initial state with three elements that will receive updates, green, blue, and orange.

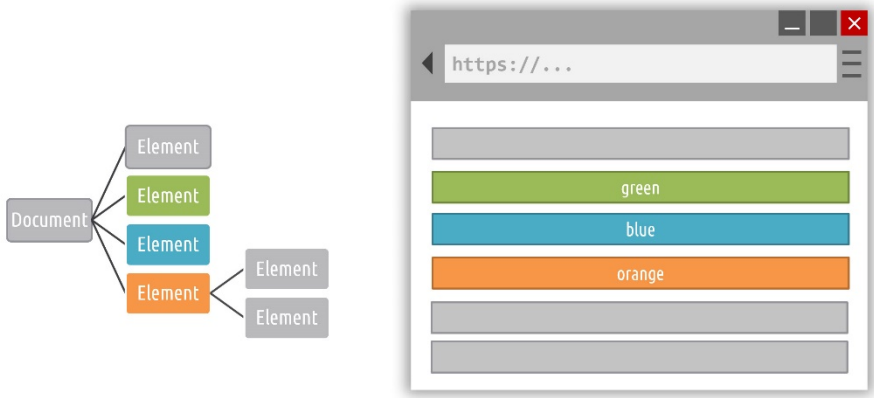


Figure 26: The initial state of the RenderTree (left) and DOM (right). The elements with the values, green, blue, and orange will be affected by code.

In **figure 27** we can see work being done over several steps within the same cycle. The items are removed and replaced, with the result swapping only the value of *green* and *blue*. Once the lifecycle is complete the differences are reconciled.

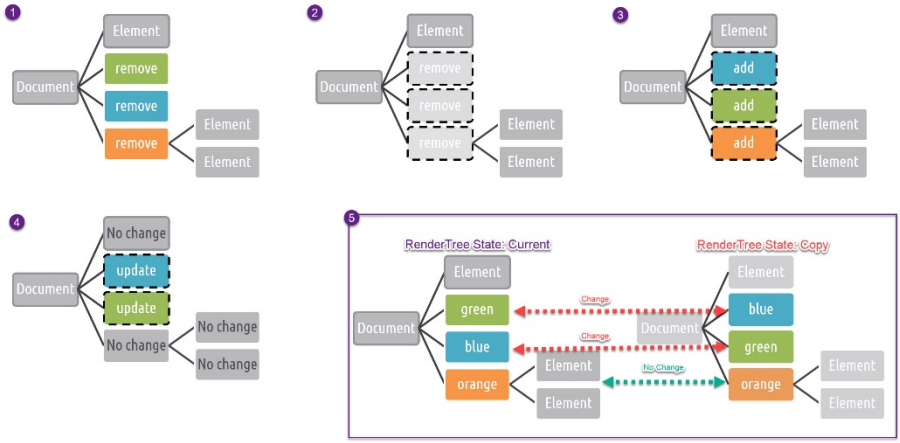


Figure 27: 1) our current RenderTree 2-4) some elements removed, replaced, and updated 5) The current state and new state are compared to find the difference.

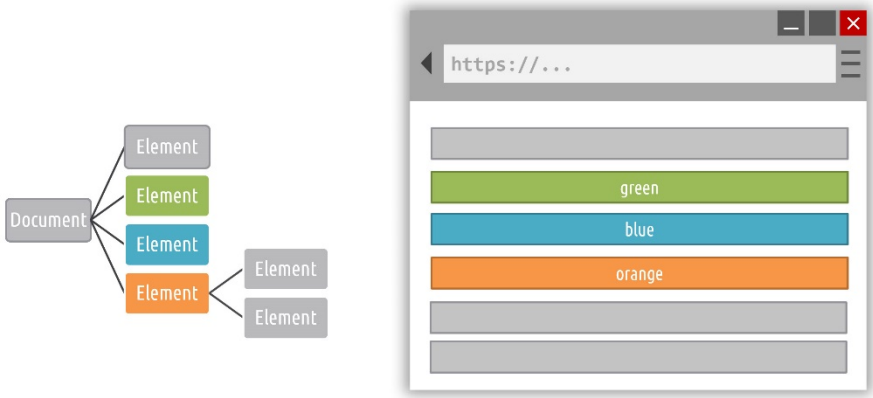


Figure 28: The RenderTree difference is used to update only elements that changed during the operation.

Creating a RenderTree

In a Blazor application, Razor Components (.razor) are actually processed quite differently from traditional Razor Pages or Views, (.cshtml) markup. Razor in the context of MVC or Razor Pages is one-way process which is rendered server side as

HTML. A component in Blazor takes a different approach – its markup is used to generate a C# class that builds the RenderTree. Let's take a closer look at the process to see how the RenderTree is created.

When a Razor Component is created a .razor file is added to our project, the contents are used to generate a C# class. The generated class inherits from the ComponentBase class which includes the component's BuildRenderTree method as shown in **Figure 29**. BuildRenderTree is a method receives a RenderTreeBuilder object and appends the component to the tree by translating our markup into RenderTree objects.

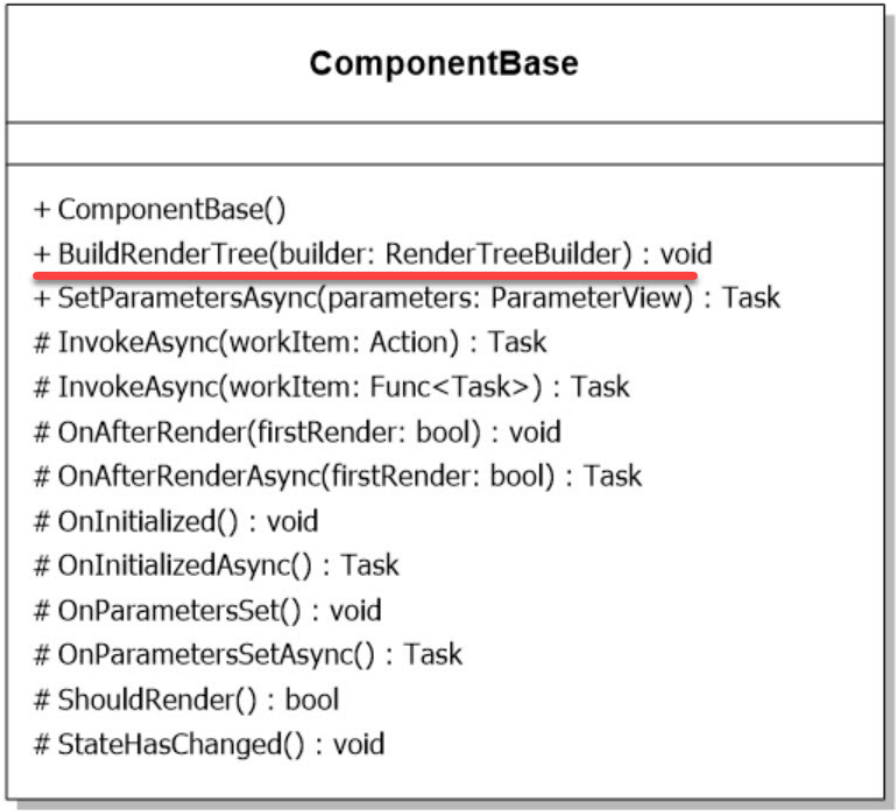


Figure 29: The ComponentBase class diagram with the BuildRenderTree method highlighted.

Using the Counter component example included in the .NET template we can see how the component's code becomes a generated class. In the counter component there are significant items we can identify in the resulting RenderTree including:

1. *page* routing directive
2. *h1* is a basic HTML element
3. *p*, *currentCount* is a mix of static content and data-bound field
4. *button* with an *onclick* event handler of *IncrementCount*
5. *code* block with C# code

```
@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++;
    }
}
```

The code in the counter example is used to generate a class with detailed BuildRenderTree method that describes the objects in the tree. If we examine the generated class, we can see how the significant items were translated into pure C# code:

1. page directive becomes an attribute tag on the class
    a. *Counter* is a public class that inherits *ComponentBase*
2. *AddMarkupContent* defines HTML content like the *h1* element
3. Mixed elements such as *p*, *currentCount* become separate nodes in the tree defined by their specific content types, OpenElement and AddContent
4. *button* includes attribute objects for CSS and the *onclick* event handler
5. *code* within the code block is evaluated as C# code

```
[Route("/counter")]
public class Counter : ComponentBase
{
```

```
    private int currentCount = 0;

    protected override void BuildRenderTree(RenderTreeBuilder builder)
    {
        builder.AddMarkupContent(0, "<h1>Counter</h1>\r\n\r\n");
        builder.OpenElement(1, "p");
        builder.AddContent(2, "Current count: ");
        builder.AddContent(3, this.currentCount);
        builder.CloseElement();
        builder.AddMarkupContent(4, "\r\n\r\n");
        builder.OpenElement(5, "button");
        builder.AddAttribute(6, "class", "btn btn-primary");
        builder.AddAttribute<MouseEventArgs>(7, "onclick",
               EventCallback.Factory.Create<MouseEventArgs>(this,
               new Action(this, Counter.IncrementCount)));
        builder.AddContent(8, "Click me");
        builder.CloseElement();
    }

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

We can see how the markup and code turn into a very structured piece of logic. Every part of the component is represented in the RenderTree so it can be efficiently communicated to the DOM.

Included with each item in the render tree is a **sequence number**, ex: AddContent(num, value). Sequence numbers are included to assist the diffing algorithm and boost efficiency. Having a raw integer gives the system an immediate indicator to determine if a change has happened by evaluating order, presence or absence of and item's sequence number. For example, if we compare a sequence of objects 1,2,3 with 1,3 then it can be determined that 2 is removed from the DOM.

The RenderTree is a powerful utility that is abstracted away from us by clever tooling. As we can see by the previous examples, our components are just standard C# classes. These classes can be built by hand using the ComponentBase class and manually writing the RenderTreeBuilder method. While possible, this would not be advised and is considered bad practice. Manually written RenderTree's can be problematic if the sequence number is not a **static** linear number. The diffing algorithm needs complete predictability, otherwise the component may re-render unnecessarily voiding it's efficiently.

> *Manually written RenderTree's can be problematic if the sequence number is not a **static** linear number.*

Optimizing Component Rendering

When we work with list of elements or components in Blazor we should consider how the list of items will behave and the intentions of how the components will be used. Ultimately Blazor's diffing algorithm must decide how the elements or components can be retained and how RenderTree objects should map to them. The diffing algorithm can generally be overlooked, but there are cases where you may want to control the process.

1. A **list** rendered (for example, in a **@foreach block**) which contains a unique identifier.
2. A **list with child elements** that may change with inserted, deleted, or re-ordered entries
3. In cases when **re-rendering** leads to a **visible behavior differences**, such as lost element focus.

The RenderTree mapping process can be controlled with the **@key** directive attribute. By adding a @key we instruct the diffing algorithm to preserve elements or components related to the key's value. Let's look at an example where @key is needed and meets the criteria listed above (rules 1-3).

An unordered list *ul* is created. Within each list item *li* is an *h1* displaying the *Value* of the class *Color*. Also, within each list item is an *input* which displays a *checkbox* element. To simulate work that we might do in a list such as: sorting, inserting, or removing items, a button is added to reverse the list. The button uses an in-line function *items = items.Reverse()* to reverse the array of items when the button is clicked.

```
<ul class="list-group">
    @foreach (var item in items)
    {
        <li class="list-group-item">
            <h1>@item.Value</h1>
            <input type="checkbox" />
        </li>
    }
</ul>

<button @onclick="_ => items = items.Reverse()">Reverse</button>
```

```
@code {
    // class Color {int Id, string Value}
    IEnumerable<Color> items = new Color[] {
       new Item {Id = 0, Value = "Green" },
       new Item {Id = 1, Value = "Blue" },
       new Item {Id = 2, Value = "Orange" },
       new Item {Id = 3, Value = "Purple" }
    };

}
```

When we run the application the list renders with a checkbox for each item. If we select the checkbox in the "Green" list item then reverse the list, then the selected checkbox will remain at the top of the list and is now occupying the "Purple" list item. This is because the diffing algorithm only updated the text in each h1 element. The initial state, and reversed state is shown in **Figure 30**, note the position of the checkbox remains unchanged.

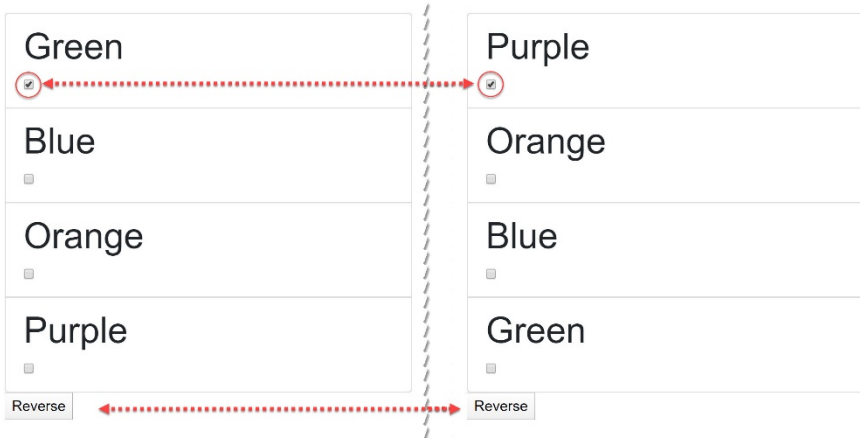


Figure 30: A rendering error is visible as the checkbox fails to move when the array is reversed, and the DOM loses context of the element's relationship.

We can use the @key directive to provide additional data for the RenderTree. The @key will identify how each list item is related to its children. With this extra information the diffing algorithm can preserve the element structure. In our example we'll assign the item's Id to the @key and run the application again.

```
@foreach (var item in items)
{
    <li @key="item.Id" class="list-group-item">
        <h1>@item.Value</h1>
        <input type="checkbox" />
```

```
    </li>
}
```

With the @key directive applied the RenderTree will create, move, or delete items in the list and their associated child elements. If we select the checkbox in the "Green" list item then reverse the list, then the selected checkbox will also move because the RenderTree is moving the entire *li* group of elements within the list, this can be seen in **Figure 31**.

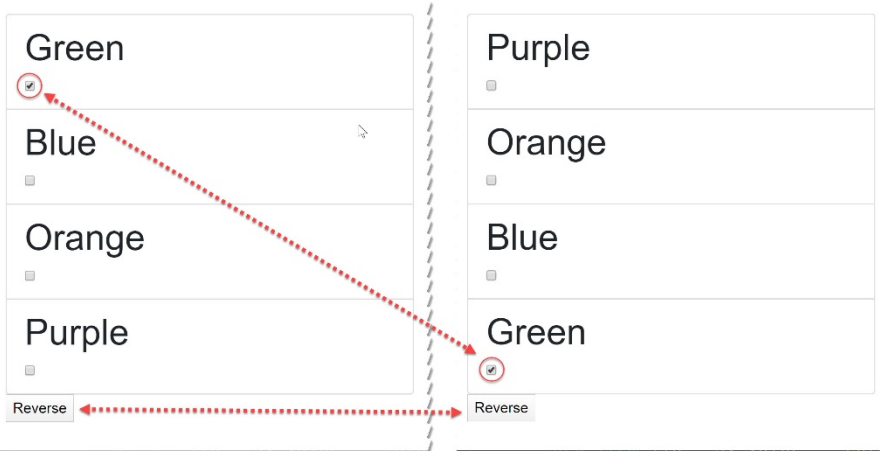


Figure 31: Using the key attribute the elements retain their relationship and the checkbox remains with the appropriate container as the DOM updates.

For this example, we had an ideal scenario that met the criteria for needing @key. We were able to fix the visual errors caused by re-rendering the list of items. However, use cases aren't always this extreme, so it's important to take careful consideration and understand the implications of applying @key.

When @key isn't used, Blazor preserves child element and component instances as much as possible. The advantage to using @key is control over how model instances are mapped to the preserved component instances, instead of the diffing algorithm selecting the mapping. Using @key comes with a slight diffing performance cost, however if elements are preserved by the RenderTree it can result in a net benefit.

Conclusion

While the RenderTree is abstracted away through the Razor syntax in .razor files, it's important to understand how it impacts the way we write code. As we saw through

example, understanding the RenderTree and how it works is essential when writing components that manage a hierarchy. The @key attribute is essential when working with collections and hierarchy so the RenderTree can be optimized and avoid rendering errors.

The RenderTree is also something that should be considered as we explore the next chapter on JavaScript interop as JavaScript can affect the DOM.

JAVASCRIPT INTEROPERABILITY

The previous chapter we learned how components render. In this chapter we'll see how JavaScript can communicate with Blazor applications.

Blazor features a JavaScript Interoperability (interop) API. Through the JavaScript interop a Blazor app can invoke JavaScript functions from C#. In reverse, C# methods can be invoked from JavaScript code. This means we can have round-trip callbacks allowing both platforms to work together.

The JavaScript interop is a necessity for Blazor development. Since Blazor via WebAssembly does not have complete access to the DOM, many DOM APIs we often need are supported through accessing JavaScript directly. Use cases for the JavaScript includecalling DOM APIs such as GeoLocation, MatchMedia, Web Storage (aka localStorage), browser cookies, and direct DOM manipulation. While it is possible to directly access and manipulate the DOM through the interop, it's important to consider how Blazor's RenderTree may be impacted.

The JavaScript interop has four essential pieces to the API:

- **IJSRuntime** – The JavaScript runtime abstraction layer which provides the InvokeAsync method.
- **InvokeAsync** – Used to invoke a JavaScript method and return a value.
- **InvokeVoidAsync** – An extension method which calls a JavaScript method with no return value.

- and **JSInvokable** – an attribute used to identify a .NET method which can be invoked from JavaScript.

Let's get a better understanding of when to use the JavaScript interop and what methods are needed. We'll use some examples to interact with the interop and see how JavaScript is necessary for a Blazor application.

Invoking JavaScript from .NET

In this example we'll learn how to invoke JavaScript from .NET. The task is to build a feature for our application that allows our application to change between a light and dark theme, as seen in **figure 32**.

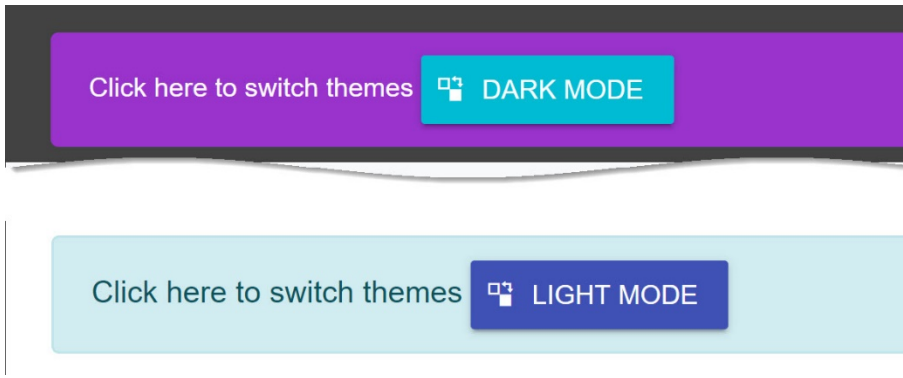


Figure 32: A toggle button allows users to switch between light and dark application themes.

Since Blazor applications are made up of standard web technologies, the best way to implement this feature is to change the application's CSS. To completely theme our application, we'll need to toggle between two unique CSS files that contain all of the theme's values: light.css and dark.css. We'll assume files have been provided for us, shown in **Figure 33**. The CSS itself isn't our concern, we only need to understand that when either theme is applied the application will use a light or dark pallet.

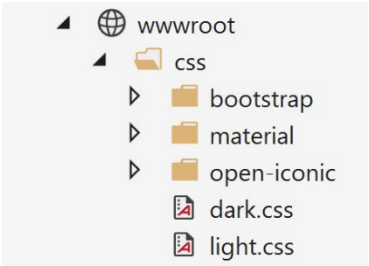


Figure 33: The project includes a light and dark theme as separate css files located in the wwwroot directory.

Blazor applications use either a _Hosts.cshtml or index.html file that hosts the client SPA page. Inside of the page's **app** element is where the dynamic portion of the application exists. Components and code cannot affect elements outside of the app element, this includes the page's **head** elements.

```
<head>
    <!-- Blazor has no scope here -->
    <meta charset="utf-8" />
    <base href="~/" />
    ...
    <link href="css/light.css" rel="stylesheet" />
    <!-- /Blazor has no scope here -->
</head>
<body>
    <app>
        <!-- Blazor (RenderTree) exist here -->
        <component type="typeof(App)" render-mode="ServerPrerendered" />
        <!-- /Blazor (RenderTree) exist here -->
    </app>
</body>
```

For our application to enable the theme to toggle between light and dark we'll need to replace the **link** element within the head. Since Blazor has no scope to this element we'll rely on the JavaScript interop to perform the task of modifying the head. Since our JavaScript will operate on elements outside of Blazor's scope, there's no chance of interfering with the RenderTree.

Before we write our interop code, we'll create a component which will eventually toggle the application theme. An example of the theme chooser component can be seen in **Figure 32**. A new ThemeChooser component is created that prompts the user with a button. Pressing the button will toggle the isDark Boolean flag, thus changing a text value from "Light" to "Dark". The text value for the theme is represented by the property SetThemeName. When the component is displayed it will read "Click here

to switch themes [Dark Mode]". At this point the component works in a limited capacity as it can only toggle a simple text message and not the actual theme of the application.

```
<div class="alert alert-info my-2">
    Click here to switch themes
    <TelerikButton OnClick="SwapTheme" Primary="true"
    Icon="@IconName.InvertColors">@SetThemeName Mode</TelerikButton>
</div>

@code {

    bool isDark;
    string SetThemeName => isDark ? "Dark" : "Light";

    async Task SwapTheme()
    {
        isDark = !isDark;
    }
}
```

Next we'll add a JavaScript file **themeChooser.js** to our application that has the code required to change the application's link element. The themeChooser.js file is located in our application's **wwwroot** folder as shown in **Figure 34**.

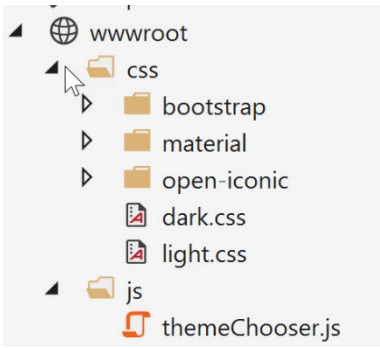


Figure 34: The themeChooser.js file provides interop functionality is saved in the wwwroot folder.

The JavaScript function must be available on the global scope of window for our Blazor application to access it. This is done by adding the namespace **themeChooser** to window, then defining the function **setTheme** under the themeChooser namespace. The naming here is important as we'll need to call these functions from our Blazor code later and Intellisense is not able to correct our mistakes.

```
window.themeChooser = {
    setTheme: function (themeName) { … theme swap code }
```

In the function setTheme a new link element is built using the parameter **themeName**. The themeName will be passed in from our C# code when the function is called. The themeName is joined with the CSS file's path and .css extension and set to the links href attribute. Finally, the head is located in the DOM and the original link is removed and replaced with the new link. The browser will recognize the change and automatically update the page with the new CSS resource.

```
window.themeChooser = {
    setTheme: function (themeName) {

        // Build the new css link
        let newLink = document.createElement("link");
        newLink.setAttribute("id", "theme");
        newLink.setAttribute("rel", "stylesheet");
        newLink.setAttribute("type", "text/css");
        newLink.setAttribute("href", `css/${themeName}.css`);

        // Remove and replace the theme
        let head = document.getElementsByTagName("head")[0];
        head.querySelector("#theme").remove();
        head.appendChild(newLink);
    }
}
```

Next, we'll add the themeChooser.js file as a static resource to in the application. In the applications host file, we'll simply make a reference using a script tag.

```
<head>
    …
    <link id="theme" href="css/light.css" rel="stylesheet" />
    <script src="js/themeChooser.js"></script>
</head>
```

Now the application has everything it needs to use the themeChooser.js code. Now we can return to the Blazor component add the code to invoke the setTheme function using the JavaScript interop.

To call into JavaScript from .NET the IJSRuntime abstraction is used. The IJSRuntime is instantiated by dependency injection and is included by the default Blazor services configuration. From this instance we can call the InvokeAsync<T> and InvokeVoidAsync methods. In this case we'll be using the InvokeVoidAsync method because we don't expect a return value from our setTheme function. The SwapTheme

method is updated with a call to **await js.InvokeVoidAsync**. The first parameter specifies the JavaScript namespace and function name. The theme name is used as the second parameter and is passed through to the setTheme function in JavaScript. We're only using a simple string parameter here, but any JSON serializable object can be used.

```
@inject IJSRuntime js

<div class="alert alert-info my-2">
    Click here to switch themes
    <TelerikButton OnClick="SwapTheme" Primary="true"
    Icon="@IconName.InvertColors">@SetThemeName Mode</TelerikButton>
</div>

@code {

    bool isDark;
    string SetThemeName => isDark ? "Dark" : "Light";

    async Task SwapTheme()
    {
        isDark = !isDark;
        await js.InvokeVoidAsync("themeChooser.setTheme", SetThemeName);
    }
}
```

This completes the feature and the application can now toggle between light and dark themes at the click of a button as shown in **Figure 35**.

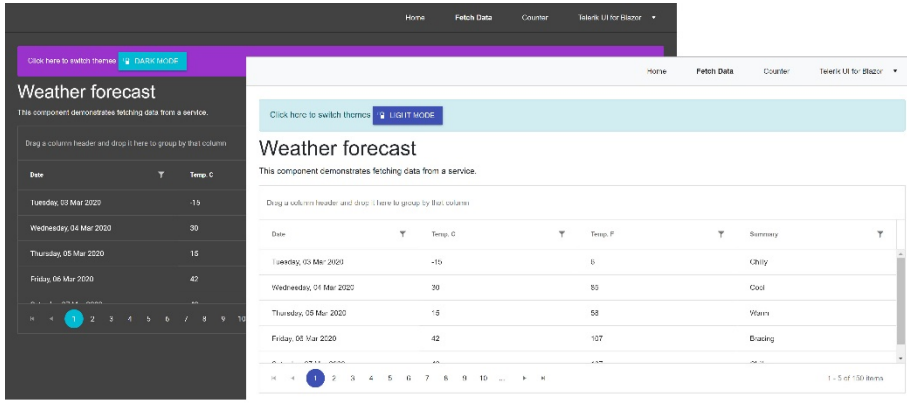


Figure 35: The theme chooser component switches the application's theme from light to dark using JavaScript interop.

In this example we saw how to leverage the JavaScript interop to execute JavaScript functions from within C#. We were able to pass argument to JavaScript and use the values to modify the theme of the application. In the next example we'll see how to preform round-trip actions using the JavaScript interop by passing values from JavaScript to C# and executing C# code from JavaScript.

Invoking .NET from JavaScript

In this example we'll learn how to pass values from JavaScript to .NET and invoke .NET from C#. The task is to build a feature for our application that allows our application to use the web geolocation API, as seen in **Figure 38**. Geolocation is a good candidate for using the JavaScript interop because Blazor does not have an API for geolocation. To access the geolocation API we will need to invoke JavaScript functions to exercise the API and perform callbacks to .NET with the results.

Before we get started let's briefly discuss how web geolocation works. The Geolocation interface is a W3c specification which most modern web browsers implement through the Navigator.geolocation read-only property. This property returns a **Geolocation** object that gives access to the location of the device through a set of functions. For the following example we'll use the **getCurrentPosition** function to return a **GeolocationCoordinates** object which represents the **latitude** and **longitude** of the user's device. This allows our app to offer customized results based on the user's location. Throughout the example we'll add classes to represent these objects in .NET, this will help serialize and deserialize values as they are passed from JavaScript to C# and back.

Let's begin with a simple example that we can use to query the browser and determine if the Geolocation API is supported. We'll check for the feature by checking if **navigator.geolocation** exists and return the result as a Boolean value. We'll start by adding the namespace **blazorGeolocation** to the window. Inside blazorGeolocation we'll add a new function, hasGeolocationFeature which performs the check and returns a true/false value. This simple snippet of JavaScript is our first bit of interop code that will interact with the geolocation API.

```
window.blazorGeolocation = {
    hasGeolocaitonFeature: function () {
        return navigator.geolocation ? true : false;
    }
};
```

We'll save the file as blazorGeolocation.js and reference it in the **head** of the application's _Host.cshtml or index.html file.

```
<head>
    …
    <script src="~/js/geoLocation.js"></script>
</head>
```

The next requirement is to create a .NET counterpart which can both invoke the **hasGeolocationFeature** function and return the result. We create a new class called Geolocation which represents the geolocation API in .NET. The Geolocation class will have a constructor that requires an **IJSRuntime** instance which we'll use internally.

```
public class Geolocation
{
    private readonly IJSRuntime js;
    public Geolocation(IJSRuntime js)
    {
        this.js = js;
    }
}
```

We covered IJSRuntime in the previous example where we used the InvokeVoidAsync to trigger a JavaScript method from C#. This time we'll use the IJSRuntime method InvokeAsync<T>, which also triggers a JavaScript function, but expects a return result of T. Our T in this instance is the Boolean value which we'll get from calling hasGeoloactionFeature in JavaScript. We'll create a method on our Geolocation class in C# to handle the interop call for us and return a ValueTask<bool>. ValueTask will ensure that when HasGeolocationFeature is called it will return either bool, or Task<bool>.

```
public class Geolocation
{
    …
    public async ValueTask<bool> HasGeolocationFeature() =>
        await js.InvokeAsync<bool>
               ("blazorGeolocation.hasGeolocaitonFeature");

}
```

Now we can test our interop from a component. We will need to inject the IJSRuntime in the component so it can be passed into the Geolocation object. Next, we'll create a Boolean field **hasGeolocation** to hold the result from when our interop is called. To ensure the page is completely loaded and JavaScript APIs are available we'll leverage the **OnAfterRenderAsync** component lifecycle method. In

OnAfterRenderAsync a new Geolocation class is created and HasGeolocationFeature is called and the result is returned to hasGeolocation. To complete the operation, we call **StateHasChanged** to re-render the user interface and display the message "Browser has Geolocation".

```
@inject IJSRuntime js
@using GeoLocation
@page "/"

<h1>Hello, world!</h1>

@if (hasGeoLocation)
{
    <p>Browser has Geolocation</p>
}

@code {
    bool hasGeoLocation;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            var geo = new Geolocation(js);
            hasGeoLocation = await geo.HasGeolocationFeature();
            StateHasChanged();
        }
    }
}
```

When the application loads the browser will see that the code is accessing the geolocation feature and immediately prompt the user for permission as seen in **Figure 36.** At this time the page will also indicate that geolocation is available as shown in **Figure 37**.

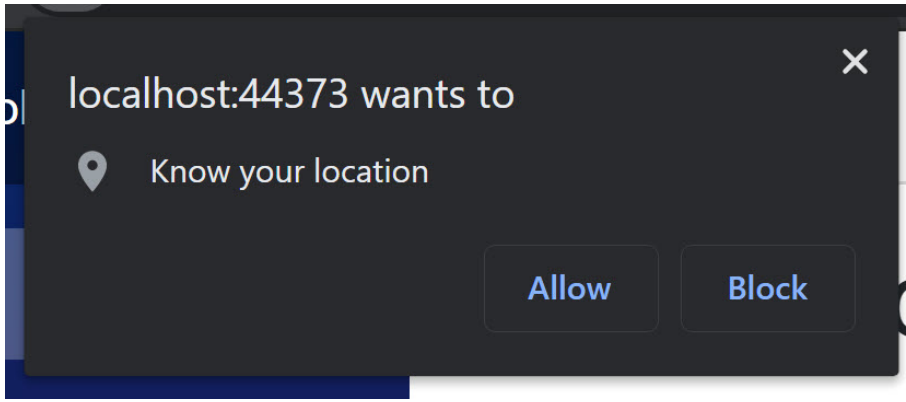


Figure 36: When any reference is made to geolocation from JavaScript for the first time, the browser will prompt the user to allow permission to use their location.

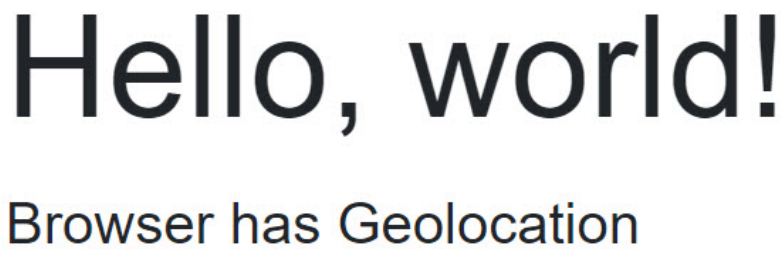


Figure 37: When the page loads the "Browser has Geolocation" message is displayed indicating JavaScript has returned a value.

So far, the interop is returning a simple Boolean type that indicates if the Geolocation feature is available. In the next steps we'll add a method to get a complex type from JavaScript that contains the user's coordinates. In order to send and receive complex types between C# and JavaScript the **objects must be JSON serializable**. We'll be using the Geolocation APIs **getCurrentPosition** function which expects a callback function which will return a **Position** object.

```
Position {
       coords: {
               latitude: 38.2099893,
               longitude: -85.21792610000001,
               … },
```

```
      timestamp: 1583430690849
}
```

The Position object also contains a JavaScript timestamp which we'll need to convert to a .NET friendly value so it can be serialized correctly in C#. To do this we'll create a helper function which takes a Position and maps the timestamp to a JavaScript date, this will guarantee that it is serialized as a Date when sent to .NET.

```
window.blazorGeolocation = {
    toSerializeable: function (e) {
        return {
            "coords": {
                "latitude": e.coords.latitude,
                "longitude": e.coords.longitude
            },
            "timestamp": new Date(e.timestamp)
        };
    },
    hasGeolocaitonFeature: function () {
        return navigator.geolocation ? true : false;
    }
};
```

In order to ensure the values are serializable in C# we'll need a few classes to receive the values from the JavaScript callback. It's important to name these objects and their properties so that they correspond to their JavaScript counterparts.

```
getCurrentPosition(PositionCallback successCallback,
                   optional PositionErrorCallback errorCallback,
                   optional PositionOptions options);

PositionCallback = void (Position position);

PositionErrorCallback = void (PositionError positionError);
```

Exact specifications for the Geolocation interface can be found at the w3c website. https://www.w3.org/TR/geolocation-API/

```
public class PositionOptions
{
    public bool EnableHighAccuracy { get; set; } = false;
    public int Timeout { get; set; }
    public int MaximumAge { get; set; } = 0;
}
```

```
public class Coords
{
    public double Latitude { get; set; }
```

```
    public double Longitude { get; set; }
}
```

```
public class Position
{
    public Coords Coords { get; set; }
    public DateTime Timestamp { get; set; }
}
```

In addition to a successful callback we can also add support for a PositionError through an enumerator.

```
public enum PositionError
{
    PERMISSION_DENIED = 1,
    POSITION_UNAVAILABLE,
    TIMEOUT
}
```

With the .NET side of the Geolocation objects complete, we can begin building a .NET API to adapt to the JavaScript implementation of getCurrentPosition function. We'll start with the callback functions that JavaScript will call when the getCurrentPosition function returns a success or failure. Two Actions are created **OnGetPosition** and **OnGetPositionError** which will handle the callback in .NET. To allow JavaScript to invoke these callbacks, corresponding functions **RaiseOnGetPosition** and **RaizeOnGetPostionError** are created with the **JSInvokable** attribute applied. JSInvokable is a special attribute used to allow functions to be invoked by the JavaScript interop. This pattern will help adapt the JavaScript API by handing off to OnGetPosition and OnGetPositionError when called.

```
public class GeoLocation
    {
       …
        private Action<Position> OnGetPosition;

        [JSInvokable]
        public void RaiseOnGetPosition(Position p) =>
               OnGetPosition?.Invoke(p);

        private Action<PositionError> OnGetPositionError;

        [JSInvokable]
        public void RaiseOnGetPositionError(PositionError err) =>
               OnGetPositionError?.Invoke(err);
       …
```

Next, we'll add a .NET method GetCurrentPosition which is nearly identical to its JavaScript counterpart. The GetCurrentPostion method will take an onSuccess and onError action as well as the PostionOptions we may want to pass to the JavaScript API. Internally the method sets the OnGetPosition and OnGetPositionError actions. To finish, the method will call a JavaScript method **blazorGeolocation.getCurrentPosition** and pass in a **DotNetObjectReference** and the PositionOptions. DotNetObjectReference is a special wrapper which passes an object reference to JavaScript instead of a JSON serializable value.

```
        …
        public async ValueTask GetCurrentPosition(
               Action<Position> onSuccess,
               Action<PositionError> onError,
               PositionOptions options = null)
        {
            OnGetPosition = onSuccess;
            OnGetPositionError = onError;
            await js.InvokeVoidAsync
               ("blazorGeolocation.getCurrentPosition",
                 DotNetObjectReference.Create(this), options);
        }

}
```

With the .NET API complete, we can wrap up the JavaScript side by finishing the blazorGeolocation.getCurrentPosition function. We already outlined this function call from C# in the GetCurrentPosition method as a function that takes a DotNetObjectReference and PositionOptions. In JavaScript the getCurrentPosition function is defined with an internal onSuccess callback that delegates back the .NET through the DotNetObjectReference, **geolocationRef**. Using the geolocationRef invokeMethodAsync function calls to RaiseOnGetPosition in .NET passing it the serialized result. The same pattern is followed for the onError function, invokeMethodAsync is used to call back to .NET invoking RaiseOnGetPositionError and supplying the error code. The last expression in the getCurrentPosition function completes the process by calling the directly to the Geolocation API.

```
window.blazorGeolocation = {
       …
    getCurrentPosition: function (geolocationRef, options) {

        function onSuccess(result) {
            return geolocationRef.invokeMethodAsync('RaiseOnGetPosition',
                       blazorGeolocation.toSerializeable(result));
        };

        function onError(er) {
```

```
            return geolocationRef
               .invokeMethodAsync('RaiseOnGetPositionError', er.code);
        };

        navigator.geolocation
                 .getCurrentPosition(onSuccess, onError, options);
    },
};
```

With the interop code complete, we can modify our component and add the new functionality. Two new fields are added to hold a Position and PositionError value. An OnSuccess method is added to handle a successful callback. This method sets the Position value when called and re-renders the component using StateHasChanged. An OnError method is also added to set the error value when an error occurs. In the OnAfterRenderAsync method a new PositionOptions is created and next GetCurrentPosition is called with the OnSuccess, OnError callbacks and options.

```
@inject IJSRuntime js
@using GeoLocation
@page "/"

<h1>Hello, world!</h1>

@if (hasGeoLocation)
{
    <p>Browser has GeoLocation</p>
}

@if (position != null)
{
    <p>Current position: @position.Coords.Latitude Lat,
       @position.Coords.Longitude Long</p>
    <p>Positioin reported at: @position.Timestamp</p>
}

@switch (error)
{
    case PositionError.PERMISSION_DENIED:
        <p>GeoLocation Permisson Denied</p>
        break;
    case PositionError.POSITION_UNAVAILABLE:
        <p>GeoLocation Position Unavailable</p>
        break;
    case PositionError.TIMEOUT:
        <p>GeoLocation Timeout</p>
        break;
    default:
        break;
}
```

```
@code {
    Position position;
    PositionError error;
    bool hasGeoLocation;

    void OnSuccess(Position p)
    {
        position = p;
        StateHasChanged();
    }

    void OnError(PositionError e)
    {
        error = e;
        StateHasChanged();
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            var geo = new GeoLocation(js);
            hasGeoLocation = await geo.HasGeolocationFeature();
            StateHasChanged();

            var options = new PositionOptions() {
               EnableHighAccuracy = false,
               Timeout = 200,
               MaximumAge = 0 };
            await geo.GetCurrentPosition(OnSuccess, OnError, options);
        }
    }

}
```

When the component renders it will invoke the Geolocation API through the interop and set the internal callback methods using OnSuccess and OnError. In JavaScript the Geolocation API receives a reference to the .NET object which it uses to marshal its own callbacks by invoking RaiseOnGetPosition and passing back the Position. This process triggers the component to re-render. Razor markup is used to display the position as shown in **Figure 38**.

Figure 38: The application displays the user's position as geolocation coordinates obtained from the JavaScript interop.

Conclusion

As we saw through example, it's possible to have full round trip communication with JavaScript from Blazor. This is a useful tool when accessing elements outside of the RenderTree or utilizing browser APIs that aren't available in Blazor. The JavaScript interop is capable of passing simple and complex values that are JSON serializable making communication JavaScript just as straight forward as working with HttpClient.

FRONT-END TOOLING

The previous chapter we learned how work with JavaScript. In this chapter we'll discuss static web assets and related tooling.

As with any web application, a Blazor application requires front-end web dependencies. In the current climate of front-end web development most dependencies are resolved and built using JavaScript tooling such as **npm** and **Webpack**. These tools are powerful but can be cumbersome to use, increase the learning curve of the platform, and operate in the context of JavaScript. Most front-end dependencies don't need these complex tools, nor are we required to directly use them to accomplish what we need. The .NET ecosystem and Visual Studio have tools available to **manage front-end dependencies** perform tasks like **compiling Sass into CSS**.

Managing Dependencies with LibMan

Library Manager (**LibMan**) is a lightweight, front-end dependency acquisition tool. LibMan downloads popular libraries and frameworks from online resources such as: **CDNJS** and **unpkg**. Dependencies acquired with LibMan are fetched and placed in the desired location within the Blazor project.

Let's see how LibMan can be used to pull in the Bootstrap framework's Sass (.scss) source as a dependency. This will give us an idea of how LibMan is used to manage dependencies at a granular level. To add Bootstrap from Visual Studio, we'll right click

on our Blazor project, then choose *add* > *Client-Side Library* from the sub-menu as shown in **Figure 39**.

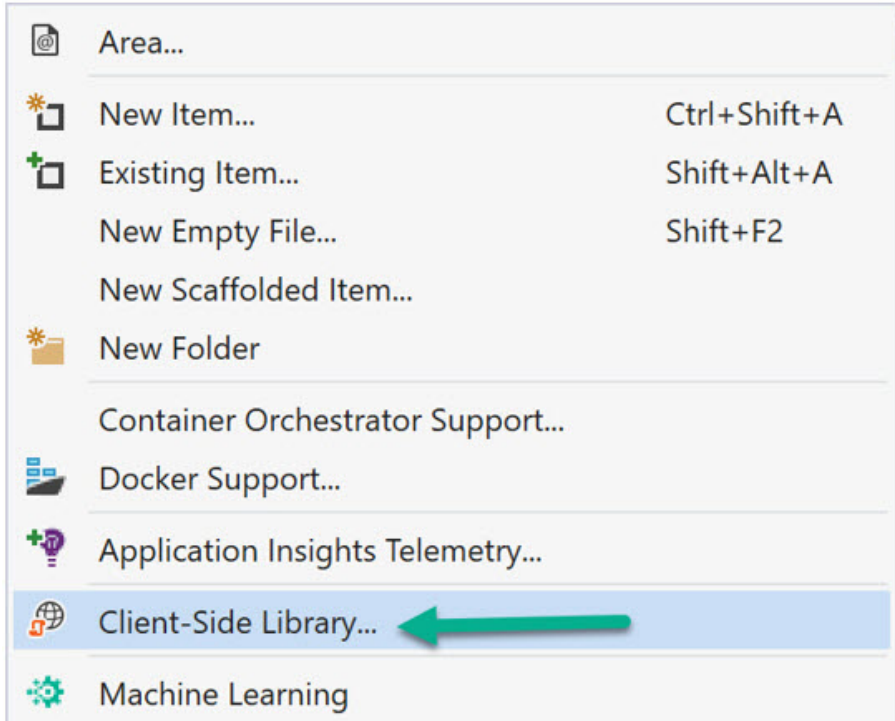


Figure 39: The LibMan dialog can be opened through the project context menu.

The Add Client-Side Library dialog will appear where we can configure how LibMan fetches our dependency. From this window shown in **Figure 40**, we'll choose the package source, unpkg. The unpkg source is useful in this scenario because it contains the dependency's complete repository including source code. To fetch Bootstrap from unpkg, we'll specify the library name and version, *bootstrap@latest* will give us the most current version of the dependency. Next, we'll choose specific files from the library, in this case we'll be targeting the *scss* folder and its contents. Because we're choosing only the scss folder we'll fetch the important to our needs thus avoiding arbitrary files being added to our project. Finally, we give a target location for LibMan to deposit our files, in this case we'll use the *Themes/Bootstrap* folder. Clicking install will initialize LibMan by generating a *libman.json* file and fetching the

dependencies.

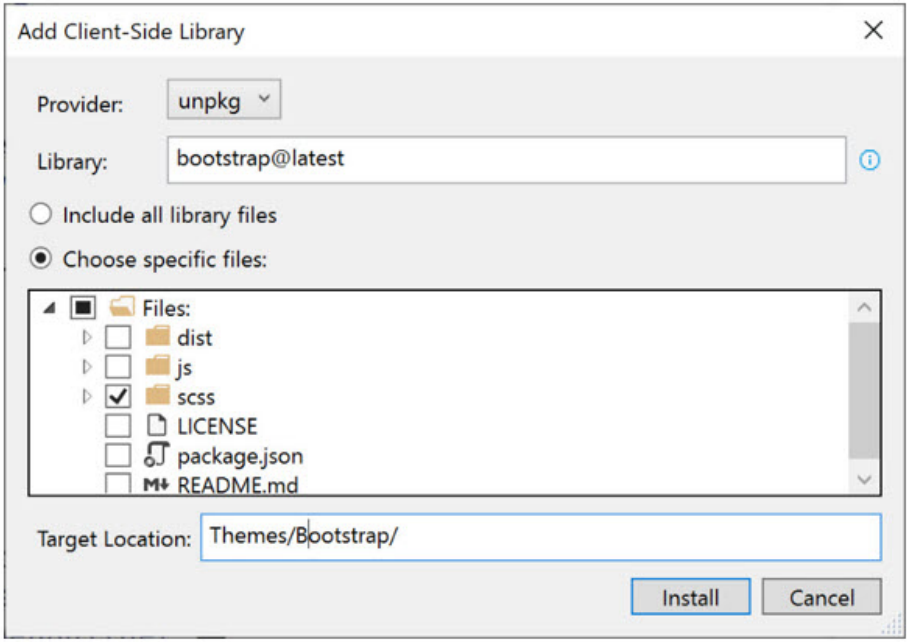


Figure 40: The LibMan dialog configures and initializes a libman.json file describing what front-end dependences should be fetched.

Even though we used LibMan in the context of Visual Studio for the example, LibMan can execute from the CLI as well.

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

Compiling with BuildWebCompiler

With the Bootstrap scss dependency installed we need a way to compile the source code into a CSS file. Once again, we will use existing .NET infrastructure instead of JavaScript tooling. For scss compilation we'll make use of **WebCompiler**, a simple tool for compiling web resources like scss, TypeScript and more. To add WebCompiler to our project we'll add the NuGet package **BuildWebCompiler** as shown in **Figure 41**.

Figure 41: The BuildWebCompiler NuGet package used to compile web resources, as shown in the NuGet explorer window.

In addition to the NuGet package an optional Visual Studio plugin is available which adds context menus for web resources. The plugin offers shortcuts for compiling assets and installing the BuildWebCompiler package.

With our tooling in place we define a **compilerconfig.json** file and specify the *outputFile*, and *inputFile* for the Bootstrap library. We configure WebCompiler to find the bootstrap.scss source file and output the compiled asset to the *wwwroot* folder in our application where it can be served to the client.

```
[
  {
    "outputFile":
    "wwwroot/css/bootstrap/bootstrap.css",
    "inputFile":
    "Themes/Bootstrap/scss/bootstrap.scss"
  }
]
```

When the config file is saved the compilation will initialize and output our file, this process will also be triggered when the project is built from either Visual Studio or the CLI.

Tools like LibMan and WebCompiler are get straight to the point and operate with very little configuration or overhead. While Blazor is the new kid on the block web

development is not new to the .NET ecosystem. Existing tools like these are finding new use with Blazor and a new ecosystem is finding its place.

RAZOR CLASS LIBRARIES

The previous chapter we discussed non-.NET static web assets. In this chapter we'll focus on creating and consuming .NET dependencies through Razor Class Libraries.

Razor Class Libraries are packages that include any combination of Components, Pages, Interop code and their supporting static assets like JavaScript, CSS, images, and fonts. The idea is to have fully functioning components that can be delivered out-of-the-box by either .dll files or NuGet packages. Having a reliable mechanism for sharing assets creates a productive environment for building Blazor applications.

To get a firm understanding of Razor Class Libraries (RCL) we will create a new RCL and then consume it in a Blazor application. This process will show the perspective of how RCLs are created and how they are used.

Creating a new Razor Class Library

To create a new RCL we'll leverage the .NET Razor Class Library template. We'll start with a fresh application and add an RCL to the project. The new project dialog is

opened by right-clicking on the project and selecting Add > New Project as shown in **Figure 42**.

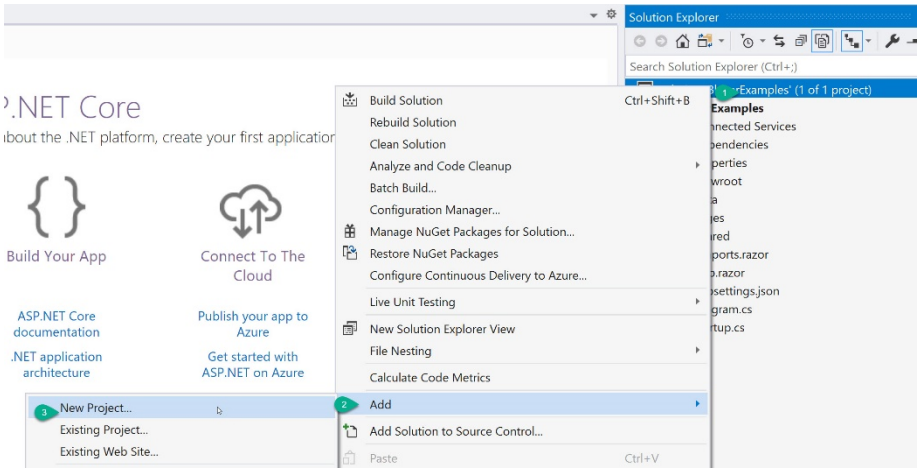


Figure 42: Adding a new project to an existing solution through the "add" context menu.

In the "Add a new project" dialog we'll select the Razor Class Library template from the list of project templates as seen in **Figure 43**. The template is selected and we continue to the next configuration screen shown in **Figure 44**. This next step is important as this step will distinguish between a Razor Component or Razor Pages project type. When the Support page views check box is selected, the template will generate a Razor Pages/Views project. Since we intend to support the Blazor framework the checkbox is left **unchecked** just as represented in **Figure 44**.

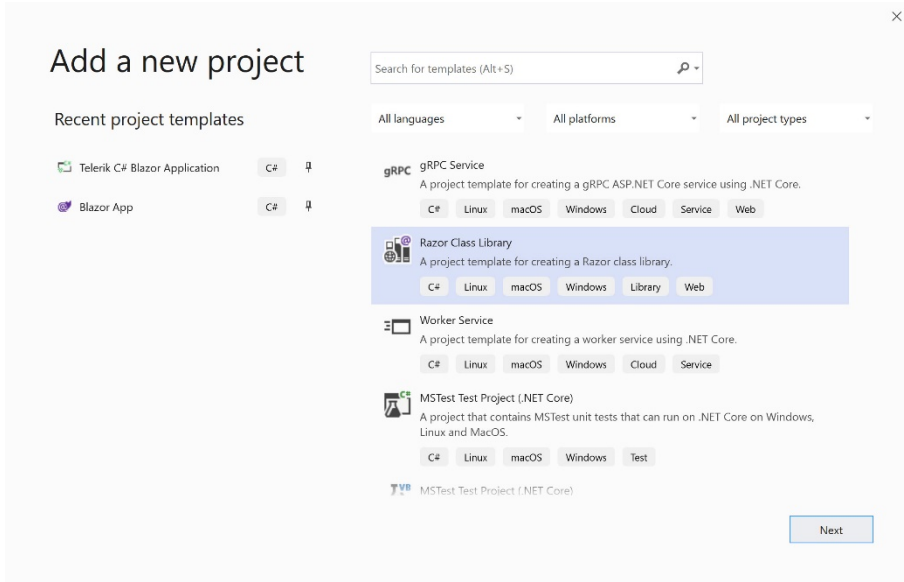


Figure 43: The Razor Class Library selected in the "Add a new project" dialog.

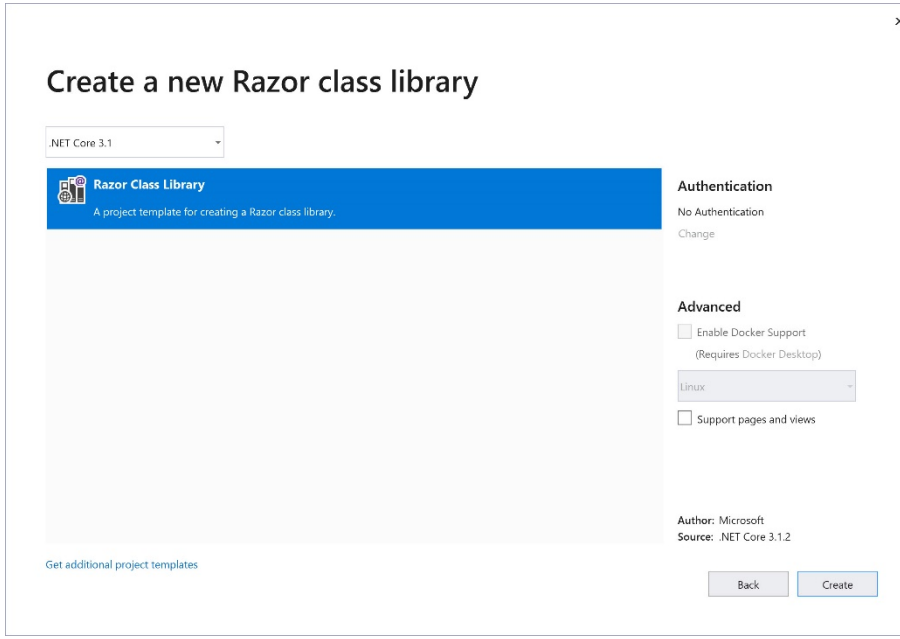


Figure 44: The Razor Class Library with the "Support pages and views" option disabled (unchecked).

The new project is added to our solution with a set of examples that are included by default. In the project we have a wwwroot which matches the convention used Blazor projects. The wwwroot folder in RCL is where we'll store our static web assets so the end user can consume them. The files in wwwroot will be placed in a special path when added to an application, we'll cover this in the next section when we consume the RCL in an application.

Inside the wwwroot folder an image, background.png is found. Along with the image is a CSS file, styles.css. Together these files make up the visual assets for the component example, Component1 which is also included in the template. Also in the wwwroot folder is a JavaScript file, **exampleJsInterop.js**. The exampleJsInterop is used with the corresponding **ExampleJsInterop** class in the root of the project. These files can be seen in **Figure 44** where the project tree is shown.

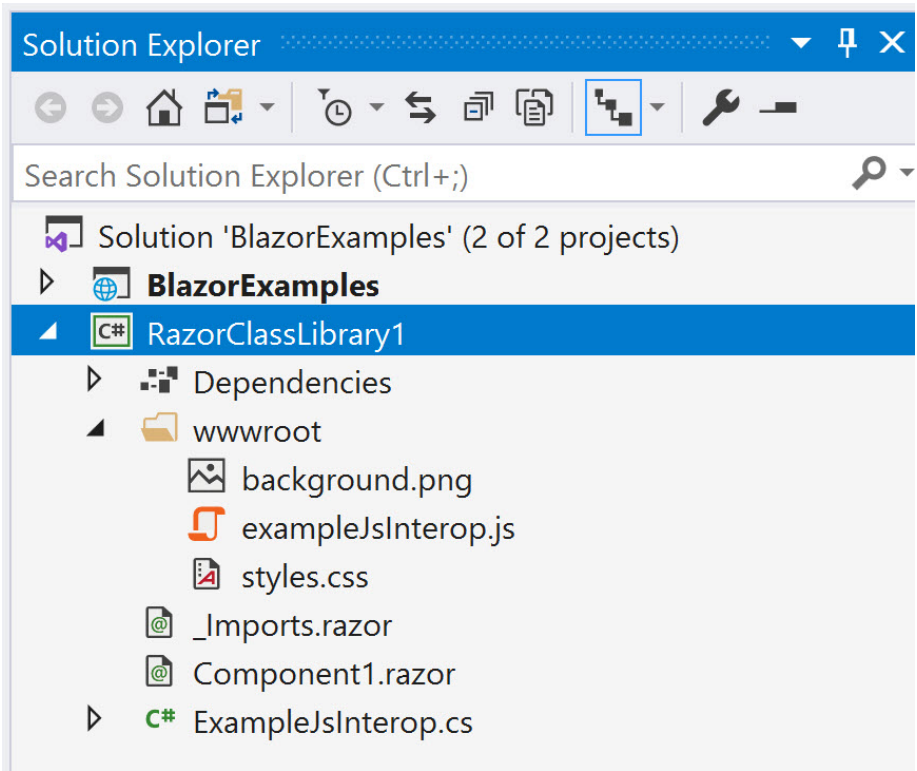


Figure 44: The files included with the Razor Class Library template. wwwroot contains the static assets that support Component1 and ExampleJsInterop.

The default project has a sample component, Component1. The component can be found in the root of the project as Component1.cs. The component itself displays a simple div element styled like a banner.

```
<div class="my-component">
    This Blazor component is defined in the
      <strong>RazorClassLibrary1</strong> package.
</div>
```

While the component in this example is in the root, any number of folders and components may be added to customize the project. Components added to the project will follow the same rules as they do in a typical Blazor application where the path and file name correspond to the namespace and class. This means we can easily move components from a Blazor application to an RCL with minimal or no code changes.

With the RCL project added to our solution we can now consume the components, code, and assets in the library.

Consuming a Razor Class Library

Razor Class Libraries can be added as DLL files, NuGet packages or through projects. A few steps are necessary to complete the process depending on what types of assets are contained in the RCL. Throughout this example we will continue with the RCL project that was added using the project template. We'll learn how to add the component, static assets and JavaScript interop from the RCL.

To add the RCL to a Blazor application from a project we'll need a project reference. From the Blazor application project we'll right-click on project's **Dependencies** node and choose **Add References**, as shown in **Figure 45**.

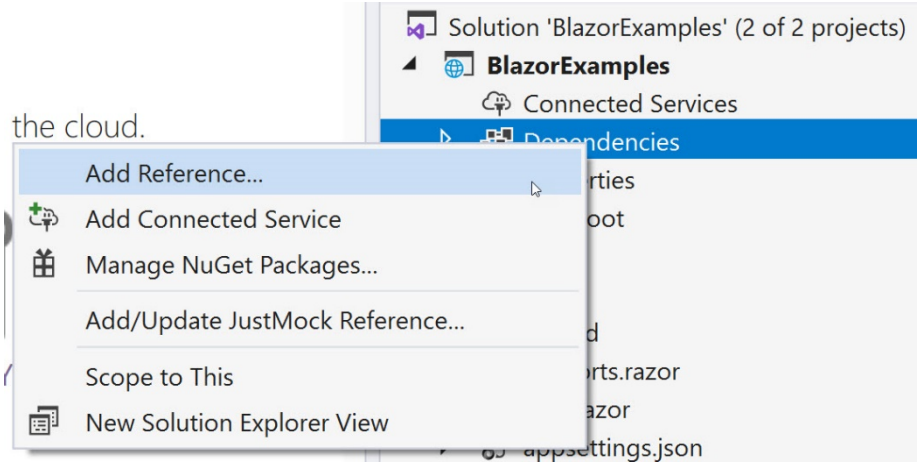


Figure 45: The Add Reference dialog is found by right-clicking on the Dependencies node in a Blazor application project.

As seen in **Figure 46**, using the Reference Manager dialog the RCL project **RazorClassLibrary1**, is selected and added to the Blazor application.

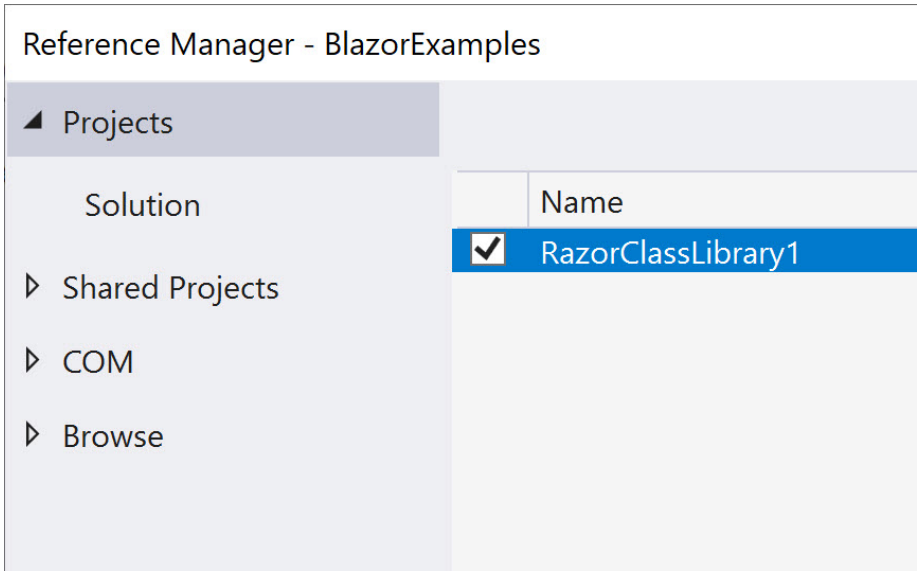


Figure 46: The Razor Class Library selected in Reference Manager dialog.

The project reference resolves any .NET dependencies our project needs from the RCL, this is the same as if we were to add a NuGet package to the project. With the .NET dependencies taken care of, we still need to manually register the static assets contained in the RCL. With Blazor and RCLs static assets are saved in the wwwroot folder of the project, however with RCLs these assets resolve to a different path. When referencing static assets such as CSS and JavaScript from a RCL the wwwroot folder path resolves to **_content/<namespace>/**, with this pattern we can easily reference these assets in the **head** our application's _Host.cshtml or index.html file. Let's reference the CSS and JavaScript files from the RCL's in the Blazor application.

```
    <link href="_content/RazorClassLibrary1/styles.css" rel="stylesheet"/>
    <script src="_content/RazorClassLibrary1/exampleJsInterop.js">
    </script>
</head>
```

With all of the references in place we can use the component from the RCL in our Blazor application. We'll open index.razor and add a using statement bringing the RCLs component into scope. Now we can add the Component1 component directly to our page.

```
@page "/"
@using RazorClassLibrary1
<h1>Hello, @message!</h1>

Welcome to your new app.

<Component1></Component1>
```

We can now run the Blazor application to see the results. When the application runs Component1 is displayed. The CSS style defined in the _content path is applied as shown in **Figure 47**.

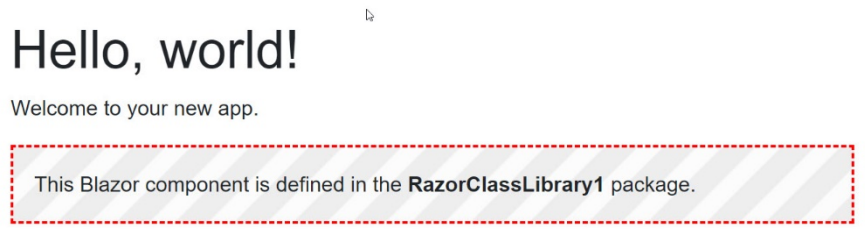


Figure 47: Component1 from the referenced Razor Class Library rendered in a running application.

As previously mentioned RCLs can contain more than just components. In our scenario, a JavaScript interop was also included in the referenced library. The JavaScript file was already referenced in an earlier step, now we just need to exercise the C# API. The example from the RCL displays an input box using the DOM's built in prompt. Let's modify the index page again, this time we'll call the included JavaScript interop and access the prompt.

At the top of the component the inject directive is used to get a reference to the IJSRuntime instance. In the code block a message field is defined to capture the result from the prompt's output. The message is set to "world" by default, so when the page initializes the display will read "Hello, world". Next, the method ShowPrompt is created, which will be invoked by a button click event. Inside the ShowPrompt method we call ExampleJsInterop.Prompt and pass the IJSRuntime instance and prompt message of "Say hello:".

```
@inject IJSRuntime js
@page "/"
@using RazorClassLibrary1

…

@code {
    string message = "world";
    async Task ShowPrompt()
    {
        message = await ExampleJsInterop.Prompt(js, "Say hello:");
    }
}
```

We'll complete the process by updating the components markup. The original "Hello, World" message is replaced with "Hello, @message" so that the message field displays as part of the heading. Finally, a button is added with the **onclick** event bound to our ShowPrompt method. This completes the feature and we'll run the application.

```
<h1>Hello, @message!</h1>

Welcome to your new app.

<Component1></Component1>
<button @onclick="ShowPrompt">Show Prompt</button>
```

When the application runs the initial heading will read "Hello, world". Below the heading the Component1 component from the referenced RCL is displayed. When the [Show Prompt] button is clicked, the browser's built-in prompt is displayed using JavaScript from the referenced RCL. In **Figure 48** we can see the application displaying

the initial “Hello, world” value and the browser prompt displayed with the message “Say hello:”. Entering a message in the prompt and clicking ok will return the message back to the application and the page is updated with the result. **Figure 49** shows the result “Hello, Blazor” after entering “Blazor” into the prompt and accepting the prompt.

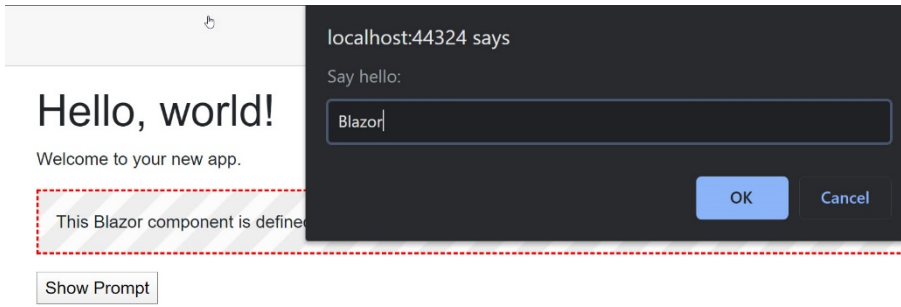


Figure 48: The RazorClassLibrary1’s Prompt interop executes and displays the browser’s built-in prompt.

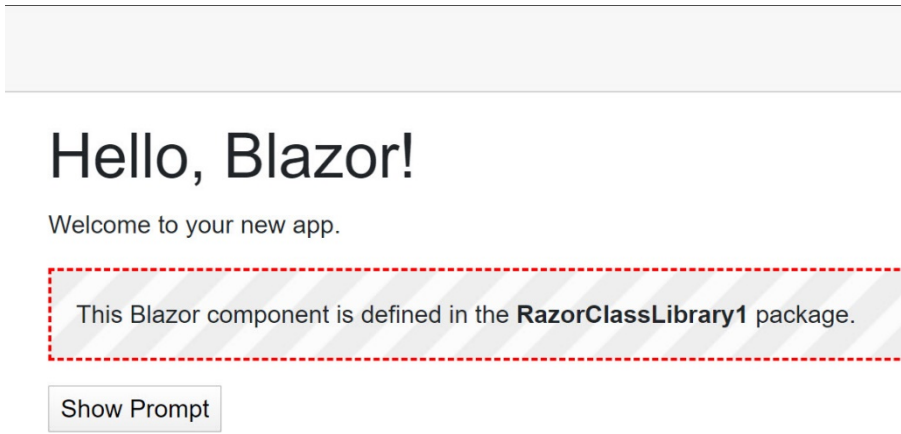


Figure 49: The final state after of the page shows the message “Hello, Blazor” in the heading.

As we can see with the template example, RCLs make sharing reusable bits of UI and code easy. Using RCLs we can create common libraries that are used across multiple projects. In addition, RCLs can be bundled as NuGet packages and distributed through NuGet.org or hosted on private NuGet servers.

Now that we understand what Razor Class Libraries are, let's see them at scale. Next, we'll look at the Telerik UI for Blazor, a library with many rich UI components and tools for building large scale Line of Business applications.

INTRODUCING TELERIK UI FOR BLAZOR

Progress Software and the Telerik brand has a long history of supporting the .NET community with products like Telerik UI for ASP.NET AJAX, UI for ASP.NET MVC, and UI for ASP.NET Core. This support continues with the release of Telerik UI for Blazor!

Telerik UI for Blazor is a completely original product and does **not** wrap existing jQuery/JavaScript products in C# and pretend it's something new. The wrapper programming model is a leaky abstraction that would "bleed back" into the .NET API layer and possibly interfere with the RenderTree. The Telerik UI for Blazor started from scratch, writing components with .NET whenever possible and only relying on JavaScript interop when necessary. The native .NET approach is a long-term investment that allows seamless integration with the Blazor framework.

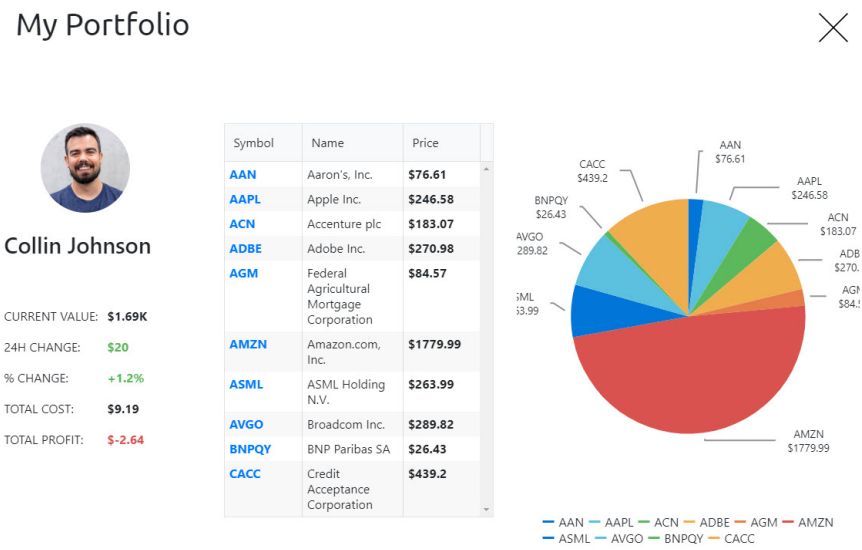


Figure 50: A Telerik UI for Blazor application with a data grid and chart component. Try it live https://demos.telerik.com/blazor-financial-portfolio

*Telerik UI for Blazor is a completely original product and does **not** wrap existing jQuery/JavaScript products in C# and pretend it's something new.*

The Telerik UI for Blazor is a full product suite that includes a Razor Class Library with 30+ UI components including what you see in **Figure 50**; data grid, scheduler, charts, window, drop downs and much more. In addition, the components support premium features like data binding, accessibility and globalization/localization. Telerik UI for Blazor isn't just UI components either. The product includes a professional support, a themes & theme builder, templates, and document processing utilities for working with office documents.

Installing

Installing Telerik UI for Blazor requires just a few simple steps. First, we'll need a 30-day free trial from Telerik.com. After creating an account, the browser will automatically download a product installer. The installer will add to our system the Telerik UI for Blazor NuGet package source, binaries, Visual Studio integration and demos. Once the installer is complete, we can choose to manually upgrade an existing project, or start a new project with one of the many templates included in the product. Since we're already on the topic of installation, we'll continue discussing the upgrade path before exploring the new-project templates.

We'll use what we learned from the previous chapter on Razor Class Libraries (RCL) to add the Telerik UI for Blazor to an existing application. As with any RCL we need to add a reference to the library. In this example we'll use a private NuGet feed to acquire the Telerik UI for Blazor NuGet package. We'll open the NuGet package manager in Visual Stuido, as shown in **Figure 52.** Then select the package source, and search for Telerik to display the NuGet package. *The private feed should already be available as it was added by the product's installer, if needed the package source can easily be added manually by following the guides in the product documentation.*

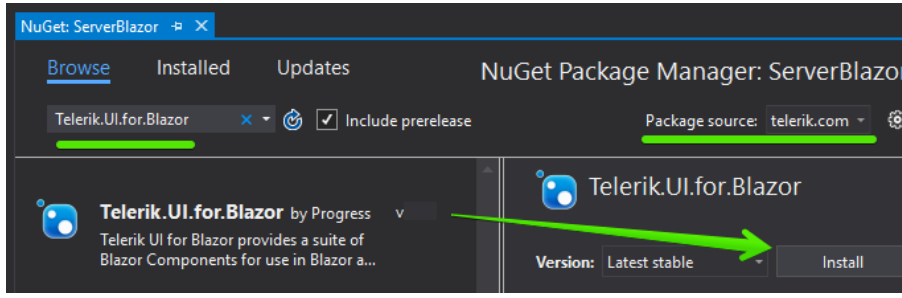


Figure 52: The Telerik UI for Blazor Razor Class Library installed from the NuGet package manger.

With the NuGet reference added we'll need to reference the static assets from the library. In our project's _Host.cshtml or index.html file we'll add the library's CSS and JavaScript interop files.

```
<head>
  <link rel="stylesheet" href=
 "_content/telerik.ui.for.blazor.trial/css/kendo-theme-default/all.css" />

  <script src="_content/telerik.ui.for.blazor.trial/js/telerik-blazor.js"
   defer></script>

  <!-- For commercial licenses use
```

```
  ref="_content/telerik.ui.for.blazor /-resource-"
  -->
</head>
```

In the head section we can also change the application theme using the CSS file path. Where the path name is /css/kendo-theme-**default**, we have the option to choose **default, bootstrap, or material,** shown in Figure 53. With this single change all the components in the app can use the Google Material Design theme.

```
<link rel="stylesheet" href=
"_content/Telerik.UI.for.Blazor/css/kendo-theme-material/all.css" />
```

Telerik UI for Blazor also requires some services through dependency injection. The services are easily registered with the AddTelerikBlazor method called from our project's entry point, Startup.cs for Blazor server, or Program.cs for WebAssembly.

```
// Blazor Server only, Startup.cs
services.AddTelerikBlazor();

// Blazor WebAssembly only, Program.cs
builder.Services.AddTelerikBlazor();
```

Since there are many components in the library and we're likely to use them throughout the project we'll add them to the global scope. Adding the Telerik.Blazor and Telerik.Blazor.Component to _Imports.razor will ensure we have access to the library anywhere the application.

```
@using Telerik.Blazor
@using Telerik.Blazor.Components
```

Finally, we'll add the TelerikRootComponent to our application's MainLayout. Telerik UI for Blazor has advanced features like modal windows, tool tips, and animations that need to register at the root element of the application and TelerikRootComponent will facilitate this for us.

```
@inherits LayoutComponentBase

<TelerikRootComponent>

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

    <div class="main">
        @Body
```

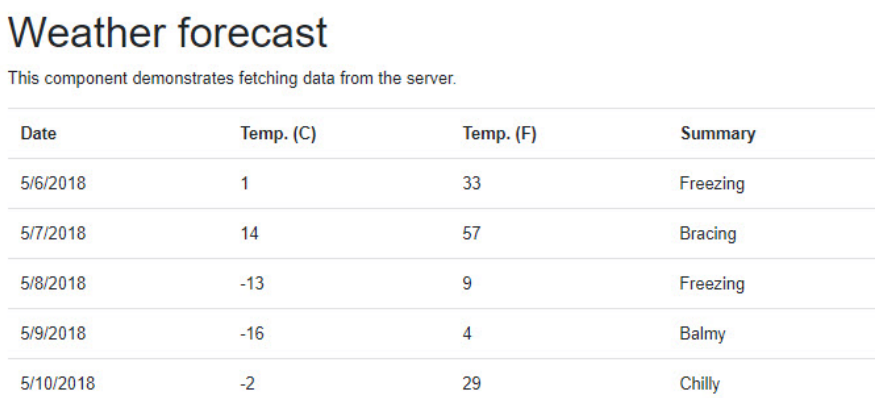
```
    </div>

</TelerikRootComponent>
```

The installation is complete, and we are now ready to begin using the Telerik UI for Blazor in our application. To get familiar with the Telerik Grid we'll modify the existing FetchData component and replace the static HTML with a fully featured component.

Blazor Data Grid

The Telerik UI for Blazor Data Grid has a long list of professional features. The data grid is capable of data binding, multi/selection, context-aware filtering, sorting, paging, themes, row/column templates, and multiple edit modes. Let's see these features in action by replacing the hand coded table in the FetchData example, shown in **Figure 54**, with the Telerik Data Grid.



Weather forecast

This component demonstrates fetching data from the server.

Date	Temp. (C)	Temp. (F)	Summary
5/6/2018	1	33	Freezing
5/7/2018	14	57	Bracing
5/8/2018	-13	9	Freezing
5/9/2018	-16	4	Balmy
5/10/2018	-2	29	Chilly

Figure 54: The unaltered HTML data table included in the FetchData example.

Locate the code for FetchData under the /Pages folder. Replace the entire table element with a TelerikGrid component. We can also eliminate the null check here as the Telerik Grid will do this operation internally.

```
<!-- all code below can be removed -->
@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
```

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

The TelerikGrid component binds the Data property to forecasts which is an array of the WeatherForecast object. The grid also has the Pageable, FilterMode, Groupable, Reorderable, and Sortable properties enabled. Inside of the TelerikGrid component, we define child components for each field we would like displayed in the grid. Because this is all C# code, we can set the Field property with C#'s nameof operator giving us type safety. Column headers are customizable through the Title property, when not set the header will automatically get the name of the property. In addition, templates can be used to display custom formats, images, and even other UI components. Here a template is used to format the Date field.

```
<TelerikGrid Data=forecasts Sortable=true Pageable=true Groupable=true
             FilterMode=GridFilterMode.FilterRow Reorderable=true >
    <GridColumns>
        <GridColumn Field="@nameof(WeatherForecast.Date)">
            <Template>
                @( (context as WeatherForecast).Date.ToShortDateString() )
            </Template>
        </GridColumn>
        <GridColumn Field="@nameof(WeatherForecast.TemperatureF)"
                    Title="Temp (F)"/>
        <GridColumn Field="@nameof(WeatherForecast.TemperatureC)"
                    Title="Temp (C)"/>
        <GridColumn Field="@nameof(WeatherForecast.Summary)"/>
    </GridColumns>
</TelerikGrid>
```

With the HTML replaced with a Telerik Grid component we can now re-run the application and see the changes as shown in **Figure 55**. Now we have a fully featured grid that is accessibility compliant.

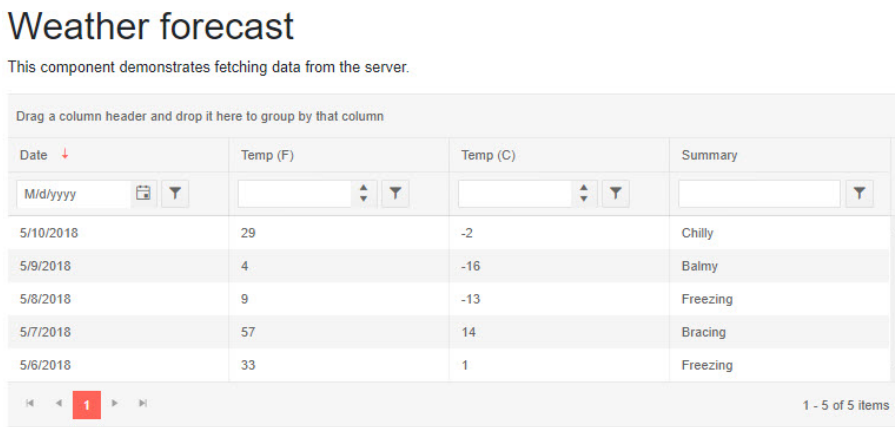


Figure 55: The Telerik UI for Blazor Data Grid, a fully featured data grid component.

Adding components to a project is relatively easy, with just a few steps we were able to transform a standard HTML component into something much more interactive and feature rich.

Telerik UI for Blazor also includes project templates to make new projects even easier. With Visual Studio integration we can simplify the process.

Visual Studio Integration

Telerik UI for Blazor currently has four templates to help kickstart new projects. When the installer runs the templates are added to Visual Studio. We can find the templates in the new project dialog by clicking File > new Telerik Project. In the dialog there are two empty templates, one for Server, and one for WebAssembly (Client App). These are great for advanced users that want a project with all the Telerik assets already referenced, but no pesky examples to remove before getting started. For users that would like some guidance, there are templates for Server, and WebAssembly

(Client App) that include working examples of the most popular user experiences: grid, menu, chart, and form. The Create New Project dialog can be seen in **Figure 56**.

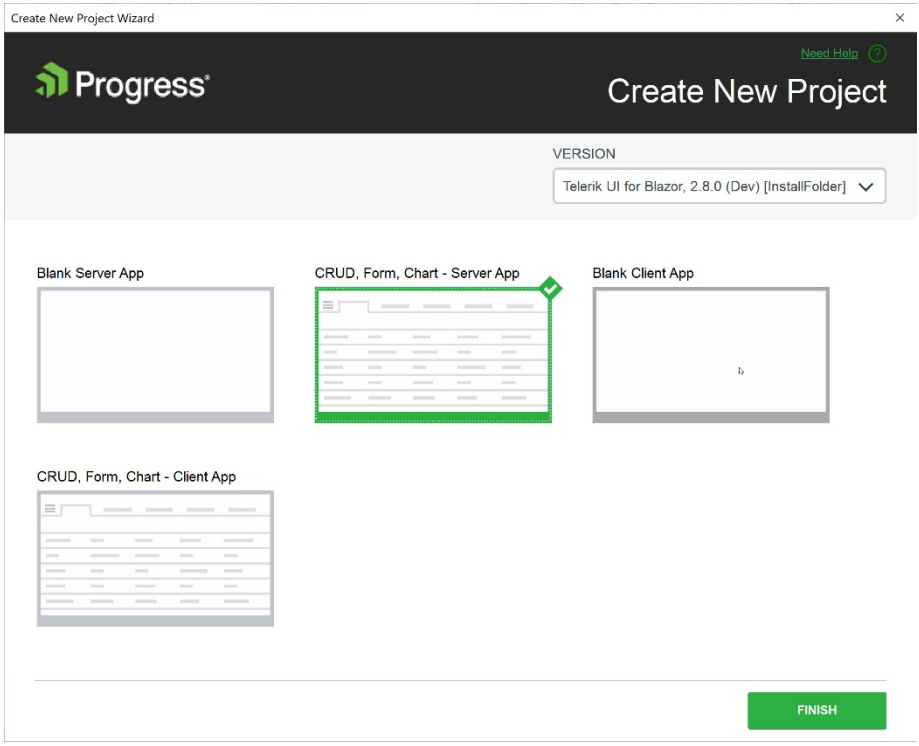


Figure 56: Telerik UI for Blazor's new project dialog has for templates for getting started.

Let's choose the **CRUD, Form, Chart – Server App** template and take a quick tour.

When using this template all the Telerik UI for Blazor assets are referenced, so there are no additional setup steps involved. We can immediately run the application and see each of the demo pages showcasing the most popular UI components & scenarios.

On the **Home** page is an overview of what is contained in the application and links to various resources. In **Figure 57-1**, we can see a Telerik Menu component utilizing the built in NavLink component that's integrated with Blazor. From the menu we can navigate to other example pages in the project or explore resources from the Telerik UI for Blazor. The next item in **Figure 57-2** is a Telerik Window component that includes helpful information about the examples as well as demonstrates the usage of

the Window component. **Figure 57-3** lists the examples along with the. razor file we can find the corresponding code in, these items are also interactive. If we scan down to button in **Figure 57-4,** we can trigger an Animation component which shows how to create a toast notification in our applications.

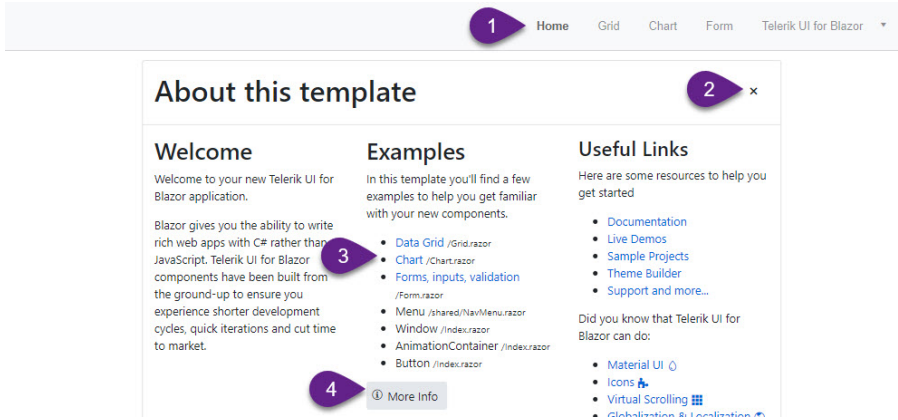


Figure 57: 1) The Telerik Menu component 2) A Telerik Window example 3) an interactive list of examples and their file locations 4) a Telerik Animation example.

If we click through to the Grid example, we'll find a fully configured Telerik Grid component. This example includes Create Update and Delete (CUD) operations and provides a starting point for any data entry application, shown in **Figure 58**.

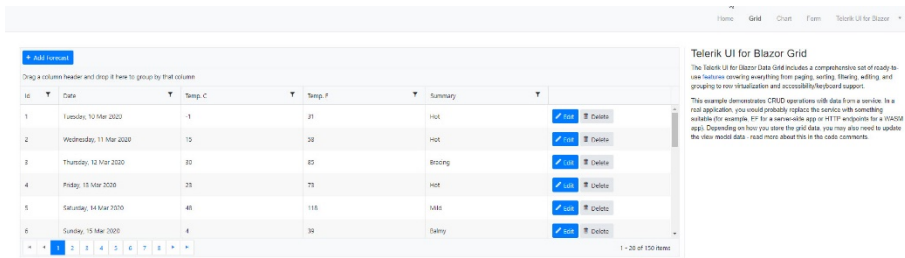


Figure 58: The Telerik Grid with a full editing experience ready to adapt to our own application scenario.

Next on the menu is the Chart example shown in **Figure 59**. The Telerik chart is fully configured with legend, labels, title and multiple series of bound data. The example is a perfect starting point for our next dashboard style application.

Figure 59: The Telerik Chart component with two Line Series components.

In the last example shown in **Figure 60**, is the ubiquitous Form. The form here shows many of the input components included in the Telerik UI for Blazor. All of the input components work seamlessly with Blazor's built in form's Validation.

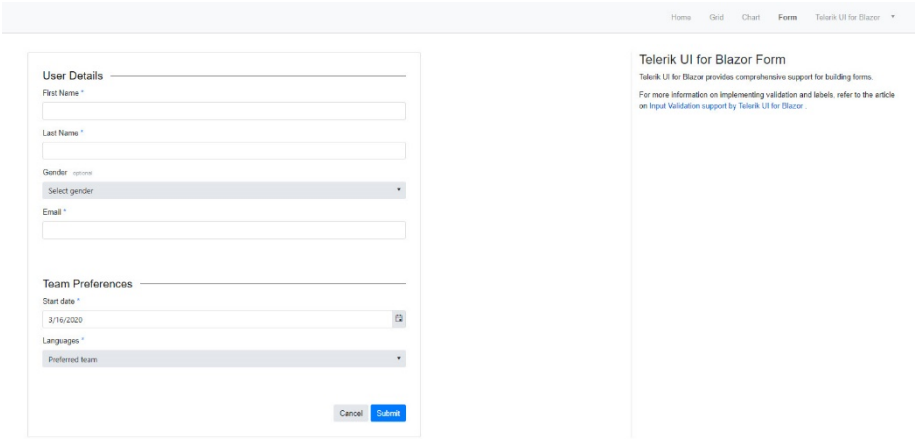


Figure 60: A form example with multiple Telerik input components and form validation.

Try Telerik UI for Blazor

Telerik UI for Blazor is meant to get developers up and running quickly. With the fully featured native Blazor UI components and project templates themes we can be far more productive in less time. Telerik UI for Blazor is available for free for 30 days. The

trial includes full access to the complete product and all its features. Visit the Telerik website www.telerik.com/blazor-ui and click Try Now to get started. The Telerik website also has interactive demos, documentation, and getting started videos.

CONCLUSION

Throughout the book we examined Blazor the new Single Page Application Framework form Microsoft. WebAssembly has enabled the development of non-JavaScript applications. WebAssembly to brings .NET to the browser using the Mono run-time enabling a new generation of .NET apps running the browser. In addition to full a client-side runtime with WebAssembly, Blazor also offers a Server-side model using SignalR. With these two models we have a choice of how to deliver applications and tailor the experience based on requirements.

We saw how easy the Razor Component model is and how to build various user interfaces. The component model has a low learning curve and adds flexibility through templates. The Razor syntax is a familiar experience for .NET developers and builds a solid foundation for component development. With Razor Components data binding syntax, we quickly tackle even the toughest workflows. We saw examples leveraging the full control over value binding and custom event syntax usage. Through these techniques we're able to define what values bind and when they are bound with little to no code.

Modern web applications have a lot of interactive elements, this has led to new ideas around keeping the DOM updated with new information. Rendering the DOM is an expensive task and Blazor aims for efficiency with the RenderTree. The RenderTree's intelligent diffing algorithm ensure Blazor apps render fast only updating the minimal changes necessary. We learned how to fully leverage this feature using the @key directive effectively hinting to the framework when components emit a hierarchy of related elements.

Blazor manages to bridge C# and JavaScript development through the JavaScript interop layer. This is the perfect tool for migrating existing JavaScript code and utilizing features out of Blazor's reach. Through the examples we saw how to invoke

JavaScript from .NET and pass values to the JavaScript environment. We also looked at an example of two-way communication with callbacks using the Geolocation API.

As developers move away from JavaScript .NET has the capability to transition some front-end JavaScript and CSS dependencies to .NET tooling. Using LibMan we saw how to use alternative package services to import the Bootstrap frameworks CSS source code (scss). With BuildWebCompiler we were able to compile the assets without the need for npm or Webpack.

Creating and consuming .NET dependencies is possible through Razor Component Libraries. Using the RCL template new project are simple to create and follow a similar structure to Blazor applications. Consuming libraries is as easy as referencing an RCL project or installing a NuGet package. RCLs can contain many components, and static assets as we saw with the Telerik UI for Blazor native Blazor component library.

With each chapter and example, we explored different features of Blazor. Each idea was presented in a way to get started and be productive right away. This book is just the beginning, Blazor is a deep topic with a variety of features we didn't have a chance to cover. If you're interested in continuing your journey with Blazor visit the Telerik Blog at www.Telerik.com/Blogs and follow the author Ed Charbeneau (@EdCharbeneau) on Twitter.

GLOSSARY

WebAssembly (wasm) - is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation.

Razor Syntax - is a template markup syntax, based on the C# programming language.

Razor Page - a .cshtml page used with the ASP.NET Razor Pages framework that contains markup and logic. Compatible with Blazor, but does not produce a RenderTree.

Razor View – a .cshtml view in the ASP.NET MVC framework. Produces an HTML document from Razor code, but does not produce a RenderTree.

Razor Component (.razor) - the component model for Blazor which encapsulates Razor Syntax and C# code and produces a RenderTree.

Blazor Component - See: Razor Component.

Blazor WebAssembly – a Blazor application that runs in the browser using WebAssembly. This application does not require a server.

Blazor Server – a Blazor application that runs on the server using a SignalR web socket connection to communicate with the client's web browser.

Blazor Client-Side – See: Blazor WebAssembly.

Blazor Server-Side – See: Blazor Server.

Dependency Injection (DI) - is a software design pattern that allows loosely coupled code. The concept involves a container which registers and instantiates object into the applications as a global value.

MSBuild - Microsoft Build Engine, better known as MSBuild, is a free and open-source build tool set for managed code as well as native C++ code and was part of .NET Framework.

Interop – See: interoperability

Interoperability - the ability of computer systems or software to exchange and make use of information.

JavaScript Interop – an interop layer that allows Blazor applications to invoke JavaScript functions or vice versa.

Mono run-time – a cross platform .NET Runtime designed for portability. This runtime powers Blazor, Xamarin and Unity applications.

npm – Node Package Manager is a package manager for the JavaScript programming language.

Webpack - is an open-source JavaScript module bundler, used primarily for JavaScript, but it can transform front-end assets like HTML, CSS, and images.

Razor Class Library (RCL) – a Razor Class Library is a specialized .NET library for Razor Components and their supporting assets. RCLs can also be used for Razor Views and Razor Pages.

BuildWebCompiler NuGet package used to compile web resources

Library Manager (LibMan) - is a lightweight, front-end dependency acquisition tool.

Code behind - a software pattern where the markup or view code is backed by a separate file containing the logic. The logic file in this pattern is referred to as a code behind.

single-page web app (SPA) - A single-page application is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages.

Geolocation - Geolocation is the identification or estimation of the real-world geographic location of an object.

IJSRuntime - Represents an instance of a JavaScript runtime to which calls may be dispatched.

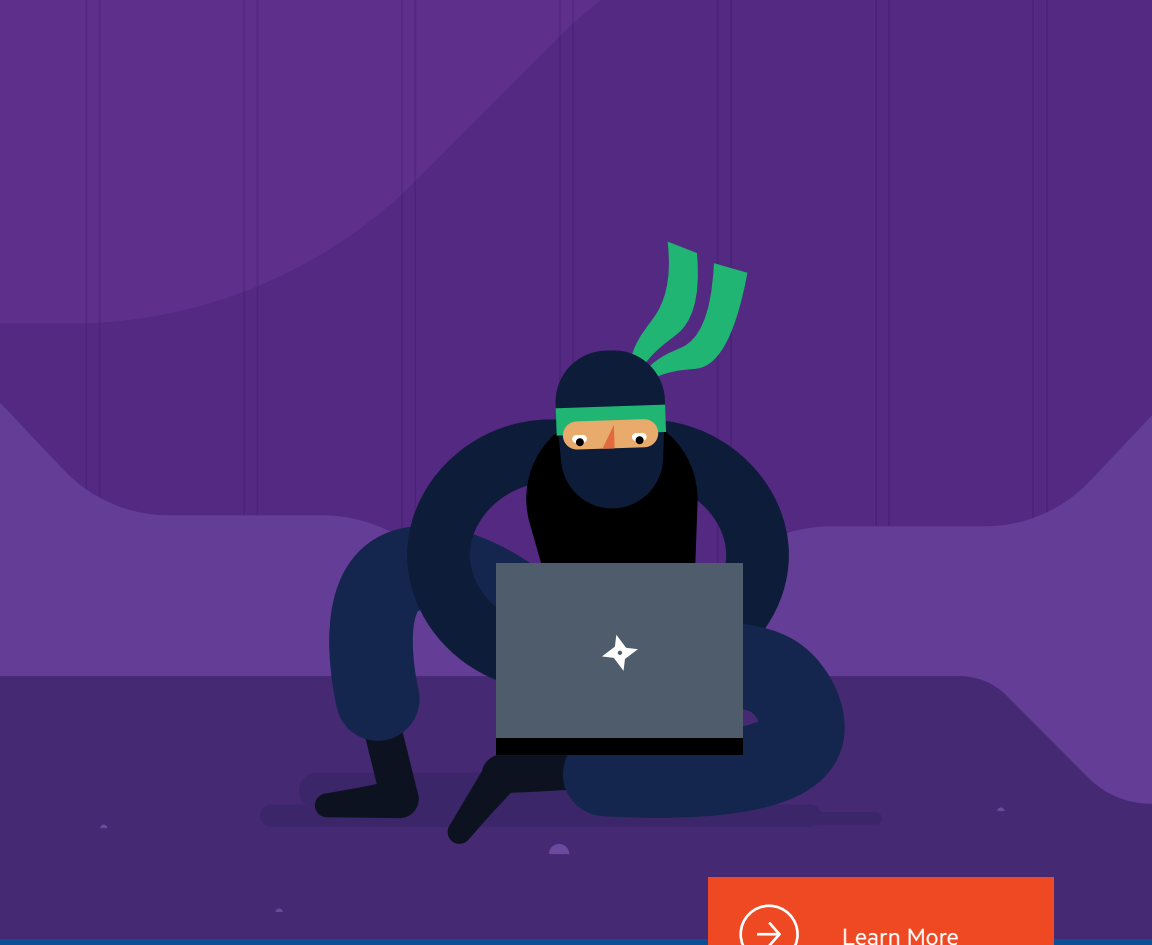
Document Object Model (DOM) - is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure

RenderTree - a copy of the DOM where changes can be quickly made as nodes in this tree can be created, updated and deleted without consequence of re-rendering the page.

Data Binding - is a general technique that binds data sources from the provider and consumer together and synchronizes them.

Directive - a language construct that specifies how a compiler should process its input.

Microsoft MVP - Microsoft Most Valuable Professional (MVP) award is given by Microsoft to "technology experts who passionately share their knowledge with the community."



Learn More

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying strategic business applications. We enable customers and partners to deliver modern, high-impact digital experiences with a fraction of the effort, time and cost. Progress offers powerful tools for easily building adaptive user experiences across any type of device or touchpoint, the flexibility of a cloud-native app dev platform to deliver modern apps, leading data connectivity technology, web content management, business rules, secure file transfer, network monitoring, plus award-winning machine learning that enables cognitive capabilities to be a part of any application. Over 1,700 independent software vendors, 100,000 enterprise customers, and two million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.

Progress®