# Blazor WebAssembly by Example

A project-based guide to building web apps with .NET, Blazor WebAssembly, and C#

Toi B. Wright

Foreword by Scott Hanselman, Partner Program Manager at Microsoft

# Blazor WebAssembly by Example

A project-based guide to building web apps with .NET, Blazor WebAssembly, and C#

**Toi B. Wright**

Packt

BIRMINGHAM—MUMBAI

# Blazor WebAssembly by Example



**Associate Group Product Manager**: Pavan Ramchandani

**Senior Editor**: Keagan Carneiro

**Content Development Editor**: Adrija Mitra

**Technical Editor**: Saurabh Kadave

**Copy Editor**: Safis Editing

**Project Coordinator**: Manthan Patel

**Proofreader**: Safis Editing

**Indexer**: Tejal Daruwale Soni

**Production Designer**: Vijay Kamble

First published: June 2021

Production reference: 1040621

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-751-1

www.packt.com

*To my boys, for their never-ending patience and understanding. To my readers, for their ceaseless curiosity about new technologies.*

*– Toi B. Wright*

# Foreword

Hi, friends! I've known Toi Wright for more than 15 years. I first met her at the Microsoft MVP Summit in Redmond back in 2005, if you can believe that. She's a brilliant technologist, community leader, and tech organizer, and we see each other every year at the annual Microsoft MVP Summit. I've had the opportunity to travel from my home in Portland to beautiful Dallas to speak to her in person at the *Dallas ASP.NET User Group*, of which she's the founder and organizer.

Toi has been very active in the ASP.NET community for many years. She has written courseware for Microsoft on ASP.NET and this is her second book on the topic. She's a respected programmer, architect, and communicator.

In this book, she walks you step by step through the creation of a number of standalone projects that are built on the Blazor WebAssembly framework. You'll learn how to leverage your experience with the .NET ecosystem to complete many different types of projects. Blazor takes .NET and your .NET skills to the web in a new way, and this book is the key to enabling that.

In *Blazor WebAssembly by Example*, Toi has shared her extensive knowledge and years of experience as a web developer and created an easy-to-follow guide for you to quickly learn how to use the Blazor WebAssembly framework. Through her words, step-by-step instructions, copious screenshots, and code samples, you will get started with running C# in your browser instead of JavaScript. Everything she'll show you, including .NET and Blazor itself, is all open source and based on open standards! I'm so glad that we both have a partner in Toi Wright to guide us in this powerful new web framework!

*Scott Hanselman – hanselman.com*

*Partner Program Manager at Microsoft*

# Contributors

## About the author

**Toi B. Wright** has been obsessed with ASP.NET for almost 20 years. She is the founder and president of the *Dallas ASP.NET User Group*. She has been a Microsoft MVP in ASP.NET for 16 years and is also an ASPInsider. She is an experienced full-stack software developer, book author, courseware author, speaker, and community leader with over 25 years of experience. She has a B.S. in computer science and engineering from the **Massachusetts Institute of Technology** (**MIT**) and an MBA from **Carnegie Mellon University** (**CMU**).

You can find her on Twitter at `@misstoi`.

*I would like to thank my husband and my two sons for their continued support, patience, and encouragement throughout the protracted process of writing this book.*

# About the reviewer

**Jürgen Gutsch** is a .NET-addicted web developer. He has been working with .NET and ASP.NET since the early versions in 2002. Before that, he wrote server-side web applications using classic ASP. He is also an active person in the .NET developer community. Jürgen writes for the dotnetPro Magazine, one of the most popular German-speaking developer magazines. He also publishes articles in English on his blog, *ASP.NET Hacker,* and contributes to several open source projects. Jürgen has been a Microsoft MVP since 2015.

The best way to contact him is using Twitter: `@sharpcms`.

He works as a developer, consultant, and trainer for the digital agency YOO Inc., located in Basel, Switzerland. YOO Inc. serves national as well as international clients and specializes in creating custom digital solutions for distinct business needs.

# Table of Contents

# 5

# Building a Weather App as a Progressive Web App (PWA)



# 6

# Building a Shopping Cart Using Application State

# 7
# Building a Kanban Board Using Events



# 8
# Building a Task Manager Using ASP.NET Web API

# Preface

Blazor WebAssembly is a framework, built on the popular and robust ASP.NET framework, that allows you to build single-page web applications that use C# on the client instead of JavaScript. Blazor WebAssembly does not rely on plugins or add-ons. It only requires that the browser support WebAssembly – and all modern browsers support it.

In this book, you will complete practical projects that will teach you the fundamentals of the Blazor WebAssembly framework. Each chapter includes a standalone project with detailed step-by-step instructions. Each project is designed to highlight one or more important concepts concerning Blazor WebAssembly. By the end of the book, you will have experience with building both simple standalone web applications and hosted web applications with SQL Server backends.

## Who this book is for

This book is for experienced web developers who are tired of constantly learning the latest new JavaScript framework and want to leverage their experience with .NET and C# to build web applications that can run anywhere.

This book is for anyone who wants to learn Blazor WebAssembly quickly by emphasizing the practical over the theoretical. It uses complete, step-by-step sample projects that are easy to follow to teach you the concepts required to develop web apps using the Blazor WebAssembly framework.

You do not need to be a professional developer to benefit from the projects in this book, but you do need some experience with C# and HTML.

## What this book covers

*Chapter 1, Introduction to Blazor WebAssembly*, provides an introduction to the Blazor WebAssembly framework. It explains the benefits of using the Blazor framework and describes the differences between the two hosting models: Blazor Server and Blazor WebAssembly. After highlighting the advantages of using the Blazor WebAssembly framework, the goals and support options for WebAssembly are discussed. Finally, it guides you through the process of setting up your computer to complete the projects in this book. By the end of this chapter, you will be able to proceed to any of the other chapters in this book.

*Chapter 2, Building Your First Blazor WebAssembly Application*, provides an introduction to Razor components through the creation of a simple project. This chapter is divided into three sections. The first section explains Razor components, routing, and Razor syntax. The second section walks you step by step through the process of creating your first Blazor WebAssembly application by using the Blazor WebAssembly App project template provided by Microsoft. The final section walks you step by step through the process of creating your own custom Blazor WebAssembly project template. The projects in *Chapters 3-7* will use this custom project template. By the end of this chapter, you will be able to create an empty Blazor WebAssembly project.

*Chapter 3, Building a Modal Dialog Using Templated Components*, provides an introduction to templated components through the creation of a modal dialog component. This chapter is divided into three sections. The first section explains `RenderFragment` parameters, `EventCallback` parameters, and CSS isolation. The second section walks you step by step through the process of creating a modal dialog component. The final section walks you step by step through the process of creating your own Razor class library and moving the modal dialog component to it. By the end of this chapter, you will be able to create a modal dialog component and share it with multiple projects through a Razor class library.

*Chapter 4, Building a Local Storage Service Using JavaScript Interoperability (JS Interop)*, provides an introduction to using JavaScript with Blazor WebAssembly through the creation of a local storage service. This chapter is divided into two sections. The first section explains the reasons that you still need to occasionally use JavaScript and how to invoke a JavaScript function from .NET. For completeness, it also covers how to invoke a .NET method from JavaScript. Finally, it introduces the Web Storage API that is used by the project. In the last section, it walks you step by step through the process of creating and testing a service that writes and reads to the local storage of the browser. By the end of this chapter, you will be able to create a local storage service by using JS Interop to invoke JavaScript functions from a Blazor WebAssembly application.

*Chapter 5, Building a Weather App as a Progressive Web App (PWA)*, provides an introduction to progressive web apps through the creation of a simple weather web app. This chapter is divided into two sections. The first section explains what a PWA is and how to create one. It covers both manifest files and service workers. Also, it describes how to use the CacheStorage API, the Geolocation API, and the OpenWeather One Call API, which are required by the project in this chapter. The second section walks you step by step through the process of creating a 5-day weather forecast app and converting it into a PWA by adding a logo, a manifest file, and a service worker. Finally, it shows you how to install and uninstall the PWA. By the end of this chapter, you will be able to convert a Blazor WebAssembly app into a PWA by adding a logo, a manifest file, and a service worker.

*Chapter 6, Building a Shopping Cart Using Application State*, provides an introduction to application state through the creation of a shopping cart web app. This chapter is divided into two sections. The first section explains application state and Dependency Injection (DI). The last section walks you step by step through the process of creating a shopping cart application. To maintain state in your application, you will create a service that you will register in the DI container and inject into your components. By the end of this chapter, you will be able to use DI to maintain application state within a Blazor WebAssembly app.

*Chapter 7, Building a Kanban Board Using Events*, provides an introduction to event handling through the creation of a Kanban board web app. This chapter is divided into two sections. The first section discusses event handling, arbitrary parameters, and attribute splatting. The last section walks you step by step through the process of creating a Kanban board application that uses the `DragEventArgs` class to enable you to drag and drop tasks between the dropzones. By the end of this chapter, you will be able to handle events in your Blazor WebAssembly app and will be comfortable using both attribute splatting and arbitrary parameters.

*Chapter 8, Building a Task Manager Using ASP.NET Web API*, provides an introduction to hosted Blazor WebAssembly applications through the creation of a task manager web app. This is the first chapter to use SQL Server. It is divided into two sections. The first section describes the components of a hosted Blazor WebAssembly application. It also explains how to use the HttpClient service and the various JSON helper methods to manipulate data. The last section walks you step by step through the process of creating a task manager application that stores its data in a SQL Server database. You will create an API controller with actions, using Entity Framework. By the end of this chapter, you will be able to create a hosted Blazor WebAssembly app that uses the ASP.NET Web API to update data in a SQL Server database.

*Chapter 9, Building an Expense Tracker Using the EditForm Component*, provides an introduction to the `EditForm` component through the creation of an expense tracker web app. This chapter uses SQL Server. It is divided into two sections. The first section introduces the `EditForm` component, the built-in input components, and the built-in validation components. The last section walks you step by step through the process of creating an expense tracker application that uses the `EditForm` component and some of the built-in components to add and edit the expenses that are stored in a SQL Server database. By the end of this chapter, you will be able to use the `EditForm` component in conjunction with the built-in components to input and validate data that is stored in a SQL Server database.

# To get the most out of this book

We recommend that you read the first two chapters of the book to understand how to set up your computer and how to use the empty Blazor WebAssembly project template. After that, you can complete the remaining chapters in any order. The projects in each chapter become more complex as you proceed through the book. The final two chapters require a SQL Server database in order to complete the project.

| Software/hardware covered in the book | OS requirements |
|---|---|
| Visual Studio 2019 Community Edition | Windows, macOS, or Linux |
| SQL Server 2019 Express Edition | Windows, macOS, or Linux |

**If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.**

This book assumes that you are an experienced web developer. You should have some experience with C# and HTML. Also, all of the projects use Bootstrap 4 as the CSS framework. If you have never used Bootstrap 4, we recommend that you familiarize yourself with it before proceeding, at `https://getbootstrap.com/docs/4.6/getting-started/introduction`.

There are some projects that use JavaScript and CSS, and two projects that use Entity Framework, but all the code is provided in the book.

# Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

# Code in Action

Code in Action videos for this book can be viewed at (`https://bit.ly/3f1rJ0R`).

# Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800567511_ColorImages.pdf`.

# Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Add the `DeleteProduct` method to the `@code` block."

A block of code is set as follows:

```
private void DeleteProduct(Product product)
{
    cart.Remove(product);
    total -= product.Price;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class CartService : ICartService
{
    public IList<Product> Cart { get; private set; }
    public int Total { get; set; }

    public event Action OnChange;
}
```

Any command-line input or output is written as follows:

```
Add-Migration Init
Update-Database
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "From the **Build** menu, select the **Build Solution** option."

> **Tips or important notes**
> Appear like this.

# Get in touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

# Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

# 1
# Introduction to Blazor WebAssembly

**Blazor WebAssembly** is Microsoft's new **single-page application (SPA)** framework for building web applications on **.NET Framework**. It enables developers to run C# code on the client. Therefore, instead of being forced to use **JavaScript** on the browser, we can now use C# on the browser.

In this chapter, we will prepare you to develop web applications using Blazor WebAssembly. We will discuss the two different Blazor hosting models and present the advantages of using Blazor WebAssembly over **Blazor Server**. Finally, we will guide you through the process of setting up your computer to complete the projects in this book.

In this chapter, we will cover the following topics:

- Benefits of using the Blazor framework
- Differences between the two hosting models
- What is **WebAssembly**?
- Setting up your PC

# Benefits of using the Blazor framework

Using the Blazor framework has several benefits. For starters, it is a free and open source framework built on Microsoft's robust .NET Framework. Also, it is an SPA framework that uses Razor syntax and can be developed using Microsoft's exceptional tooling.

## .NET Framework

Blazor is built on **.**NET Framework. Since **Blazor** is built on .NET Framework, anyone familiar with .NET Framework can quickly become productive using the Blazor framework. The Blazor framework leverages the robust ecosystem of .NET libraries and NuGet packages from .NET Framework. Also, since both client and server code are written in C#, they can share code and libraries, such as the application logic used for data validation.

Blazor is open source. Since Blazor is a feature of the ASP.NET framework, all of the source code for Blazor is available on GitHub as part of the `dotnet/aspnetcore` repository that is owned by the **.NET Foundation**. .NET Foundation is an independent, non-profit organization established to support the innovative, commercially friendly, open source ecosystem around the .NET platform. The .NET platform has a strong community of over 100,000 contributions from more than 3,700 companies.

Blazor is free. Since .NET Framework is free, this means that Blazor is also free. There are no fees or licensing costs associated with using Blazor, including for commercial uses.

## SPA framework

The **Blazor** framework is an SPA framework. As the name implies, an SPA is a web app that consists of a single page. The application dynamically rewrites the single page instead of loading an entirely new page in response to each UI update. The goal is faster transitions that make the web app feel more like a native app.

When a page is rendered, Blazor creates a render tree that is a graph of the components on the page. It is similar to the **Document Object Model** (**DOM**) created by the browser. However, it is a virtual DOM. Updates to the UI are applied to the virtual DOM and only the differences between the DOM and the virtual DOM are updated by the browser.

## Razor syntax

The name of the Blazor framework has an interesting origin story. The term *Blazor* is a combination of the word *browser* and the word *razor*. **Razor** is the **ASP.NET** view engine used to create dynamic web pages with C#. Razor is a syntax for combining HTML markup with C# code that was designed for developer productivity. It allows the developer to use both HTML markup and C# in the same file.

Blazor web apps are built using **Razor Components**. Razor Components are reusable UI elements that contain C# code, markup, and other Razor Components. Razor Components are quite literally the building blocks of the Blazor framework. For more information on Razor Components, refer to *Chapter 2*, *Building Your First Blazor WebAssembly Application*.

> **Important note**
>
> Razor Pages and MVC also use the Razor syntax. Unlike Razor Pages and MVC, which render the whole page, Razor Components only render the DOM changes. One way to easily distinguish between them is that Razor components use the `RAZOR` file extension, while Razor Pages use the `CSHTML` file extension.

## Awesome tooling

You can use either **Microsoft Visual Studio** or **Microsoft Visual Studio Code** to develop Blazor WebAssembly applications. Microsoft Visual Studio is an **integrated development environment** (**IDE**), while Microsoft Visual Code is a lightweight, yet powerful, editor. They are both incredible tools for building enterprise applications. Also, they are both available for free and there are versions that run on Windows, Linux, and macOS.

There are many benefits associated with using the Blazor framework to develop web apps. Since it is built on the mature .NET Framework, it enables developers to use the skills, such as C#, and the tools, such as Visual Studio, that they have already mastered. Also, since it is an SPA framework, Blazor web apps feel like native apps.

# Hosting models

**Blazor** has two different hosting models. The first hosting model that Microsoft released is the **Blazor Server** model. In this hosting model, the web app is executed on the server. The second hosting model that Microsoft released, and the topic of this book, is the **Blazor WebAssembly** model. In this hosting model, the web app is executed on the browser.

Each hosting model has its own advantages and disadvantages. However, they both use the same underlying architecture. Therefore, it is possible to write and test your code independent of the hosting model. The major differences between the two hosting models concern latency, security, data access, and offline support.

## Blazor Server

As we just mentioned, the Blazor Server hosting model was the first hosting model released by Microsoft. It was released as part of the .NET Core 3 release in September 2019.

The following diagram illustrates the Blazor Server hosting model:

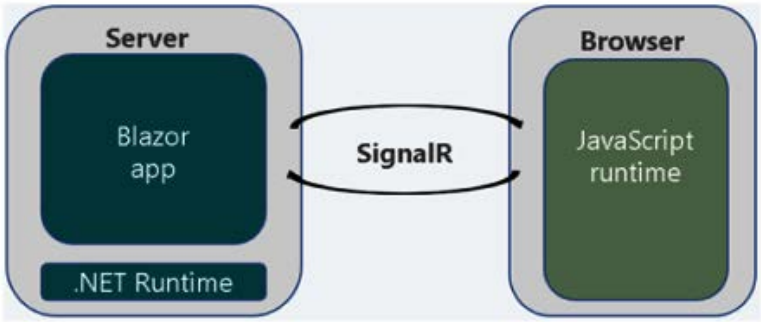



Figure 1.1 – Blazor Server

In this hosting model, the web app is executed on the server and only updates to the UI are sent to the client's browser. The browser is treated as a thin client and all of the processing occurs on the server. When using Blazor Server, UI updates, event handling, and JavaScript calls are all handled over an ASP.NET Core **SignalR** connection.

> **Important note**
>
> **SignalR** is a software library that allows the web server to push real-time notifications to the browser. Blazor Server uses it to send UI updates to the browser.

### Advantages of Blazor Server

There are a few advantages of using **Blazor Server** versus using **Blazor WebAssembly**. However, the key advantage is that everything happens on the server. Since the web app runs on the server, it has access to everything on the server. As a result, security and data access are simplified. Also, since everything happens on the server, the assemblies (DLLs) that contain the web app's code remain on the server.

Another advantage of using Blazer Server is that it can run on thin clients and older browsers, such as Internet Explorer, that do not support WebAssembly.

Finally, the initial load time for the first use of a web app that is using Blazor Server can be much less than that of a web app that is using Blazor WebAssembly because there are fewer files to download.

### Disadvantages of Blazor Server

The Blazor Server hosting model has a number of disadvantages versus Blazor WebAssembly due to the fact that the browser must maintain a constant connection to the server. Since there is no offline support, every single user interaction requires a network roundtrip. As a result of all of these roundtrips, Blazor Server web apps have higher latency than Blazor WebAssembly web apps and can feel sluggish.

> **Tip**
> Latency is the time between the UI action and the time when the UI is updated.

Another disadvantage of using Blazor Sever is that it relies on SignalR for every single UI update. Microsoft's support for SignalR has been improving, but it can be challenging to scale.

Finally, a Blazor Server web app must be served from an **ASP.NET Core** server.

## Blazor WebAssembly

The **Blazor WebAssembly** hosting model is the most recent hosting model released by Microsoft, and the topic of this book. **Blazor WebAssembly 3.2.0** was released in May 2020. **Blazor WebAssembly in .NET 5** was released as part of the **.NET 5.0** release in November 2020 and it is not a **long-term support** (**LTS**) release. This book will be using Blazor WebAssembly in .NET 5 for all of the projects.

> **Tip**
> LTS releases are supported by Microsoft for at least 3 years after their initial release. Blazor WebAssembly in .NET 5 is not an LTS release. If you are starting a new project with Blazor WebAssembly, you should use the most recent release..

The following diagram illustrates the Blazor WebAssembly hosting model:

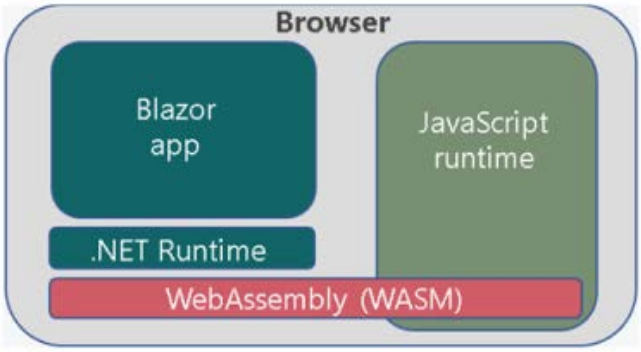



Figure 1.2 – Blazor WebAssembly

In this hosting model, the web app is executed on the browser. In order for both the web app and the .NET runtime to run on the browser, the browser must support **WebAssembly**. WebAssembly is a web standard supported by all modern browsers, including mobile browsers. While Blazor WebAssembly itself does not require a server, the web app may require one for data access and authentication.

In the past, the only way to run C# code on the browser was to use a plugin, such as **Silverlight**. Silverlight was a free browser plugin provided by Microsoft. It was very popular until Apple decided to disallow the use of a browser plugin on iOS. As a result of Apple's decision, Silverlight was abandoned by Microsoft. Blazor does not rely on plugins or recompiling the code into other languages. Instead, it is based on open web standards and is supported by all modern browsers, including mobile browsers.

## Advantages of Blazor WebAssembly

Blazor WebAssembly has many advantages. First of all, since it runs on the browser, it relies on client resources instead of server resources. Therefore, unlike Blazor Server, there is no latency due to each UI interaction requiring a roundtrip to the server.

Blazor WebAssembly can be used to create a **Progressive Web App** (**PWA**). A PWA is a web app that looks and feels like a native application. They provide offline functionality, background activity, native API layers, and push notifications. They can even be listed in the various app stores. By configuring your Blazor WebAssembly app as a PWA, your app can reach anyone, anywhere, on any device with a single code base. For more information on creating a PWA, refer to *Chapter 5*, *Building a Weather App as a Progressive Web App (PWA)*.

Finally, a **Blazor WebAssembly** web app does not rely on an **ASP.NET Core** server. In fact, it is possible to deploy a **Blazor WebAssembly** web app without a server.

### Disadvantages of Blazor WebAssembly

To be fair, there are some disadvantages to using Blazor WebAssembly that should be considered. For starters, when using Blazor WebAssembly, the .NET runtime, the `dotnet.wasm` file and your assemblies all need to be downloaded to the browser for your web app to work. Therefore, a Blazor WebAssembly application usually takes longer to initially load than a Blazor Server application. However, there are strategies, such as deferring the loading of some of the assemblies until they are needed, designed to speed up the load time of the application.

When debugging a Blazor Server application, you can use the standard .NET debugger. However, to debug a Blazor WebAssembly application, you need to use the browser's debugger. To enable the browser's debugger, you need to launch the browser with remote debugging enabled and then use *Alt+Shift+D* to initiate a proxy component that sits between the browser and the editor. Unfortunately, due to the complexities concerning debugging on the browser, there are certain scenarios, such as hitting breakpoints before the debug proxy is running and breaking on unhandled exceptions, that the debugger currently can't handle. Microsoft is actively working on improving the debugging experience.

Another disadvantage of Blazor WebAssembly web apps is that they are only as powerful as the browser that they run on. Therefore, thin clients are not supported. Blazor WebAssembly can only run on a browser that supports WebAssembly. Luckily, due to a significant amount of coordination between the **World Wide Web Consortium** (**W3C**) and engineers from Apple, Google, Microsoft and Mozilla, all modern browsers support WebAssembly.

The Blazor framework provides two different hosting models, Blazor Server and Blazor WebAssembly. A Blazor Server web app runs on the server and uses **SignalR** to serve the HTML to the browser. Conversely, a Blazor WebAssembly web app runs directly in the browser. They each have their advantages and disadvantages. However, if you want to create responsive, native-like web apps that can work offline, you need to use Blazor WebAssembly.

## What is WebAssembly?

WebAssembly is a binary instruction format that allows code written in C# to run on the browser at near-native speed. To run .NET binaries in a web browser, it uses a version of the .NET runtime that has been compiled to WebAssembly. You can think of it as executing natively compiled code in a browser.

WebAssembly is an open standard developed by a W3C Community Group. It was originally announced in 2015, and the first browser that supported it was released in 2017.

## WebAssembly goals

When WebAssembly was originally being developed, there were four main design goals for the project:

- Fast and efficient
- Safe
- Open
- Don't break the web

WebAssembly is fast and efficient. It is designed to allow developers to write code in any language that can then be compiled to run in the browser. Since the code is compiled, it is fast and performs at near-native speed.

WebAssembly is safe. It does not allow direct interaction with the browser's DOM. Instead, it runs in its own memory-safe, sandboxed execution environment. You must use JavaScript interop to interact with the DOM. The project in *Chapter 4*, *Building a Local Storage Service using JavaScript Interoperability (JS interop)*, will teach you how to use JavaScript interop.

WebAssembly is open. Although it is a low-level assembly language, it can be edited and debugged by hand.

WebAssembly didn't break the web. It is a web standard that is designed to work with other web technologies. Also, WebAssembly modules can access the same Web APIs that are accessible from JavaScript.

## WebAssembly support

As mentioned earlier, WebAssembly runs on all modern browsers, including mobile browsers. As you can see from the following table, all current versions of the most popular browsers are compatible with WebAssembly:

| **Browser** | **Version** |
|---|---|
| Microsoft Edge | Current |
| Mozilla Firefox | Current |
| Google Chrome, including Android | Current |
| Safari, including iOS | Current |
| Opera, including Android | Current |
| Microsoft Internet Explorer | Not Supported |

Figure 1.3 – WebAssembly browser compatibility

> **Important note**
>
> Microsoft Internet Explorer does not currently support WebAssembly and will never support WebAssembly. So, if your web app must be able to run on Microsoft Internet Explorer, do not use Blazor WebAssembly.

WebAssembly is a web standard that allows developers to run code written in any language in the browser. It is supported by all modern browsers.

# Setting up your PC

For the projects in this book, we use Visual Studio 2019, .NET 5.0, and SQL Server 2019.

All of the projects are built using **Visual Studio 2019 Community Edition** version 16.9.5 with the ASP.NET and Web Development workload. If you need to install Visual Studio 2019, follow the directions in the *Installing Visual Studio Community Edition* section later in this chapter.

> **Tip**
>
> Although we are using Visual Studio 2019 Community Edition, any edition of Visual Studio 2019 can be used to complete the projects in this book. **Microsoft Visual Studio Code** can also be used.

Blazor WebAssembly in .NET 5 requires .NET 5.0. To check the version of .NET that is running on your computer, open the Command Prompt and enter the following command:

```
dotnet --version
```

If your computer is not running .NET 5.0, follow the directions in the *Installing .NET 5.0* section later in this chapter.

The final two projects in this book use **SQL Server 2019 Express Edition** as the backend database. If you need to install SQL Server 2019 Express Edition, follow the directions in the *Installing SQL Server Express* section later in this chapter.

> **Tip**
>
> Although we are using SQL Server 2019 Express Edition, any year or edition of SQL Server can be used to complete the projects in this book.

# Installing Visual Studio Community Edition

Visual Studio Community Edition is the free edition of Visual Studio. To install Visual Studio Community Edition, perform the following steps:

1. Download the **Visual Studio installer** from `https://visualstudio.microsoft.com`.
2. Once the download is complete, run the installer to complete the installation. The first step in the installation process is for the **Visual Studio installer** to check the system for existing versions of Visual Studio.
3. Once the installer has finished checking for installed versions, it will open the following installation dialog:

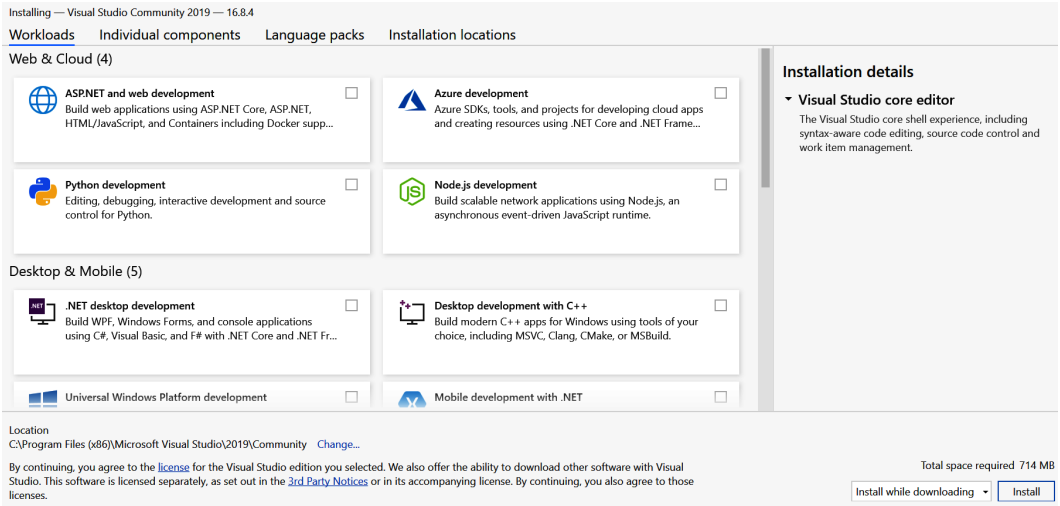



Figure 1.4 – The Visual Studio installer

4. Select the **ASP.NET and web development** workload and click the **Install** button to complete the installation.

## Installing .NET 5.0

To install **.NET 5.0**, perform the following steps:

1. Download the installer from `https://dotnet.microsoft.com/download/dotnet/5.0`.
2. Once the download completes, run the installer to complete the installation of .NET 5.0 on your computer.
3. Open the Command Prompt and enter the following command to verify that your computer is now running .NET 5.0:

```
dotnet --version
```

The following screenshot is from a computer that is running .NET 5.0:

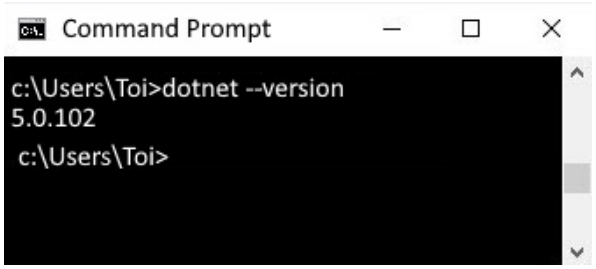



Figure 1.5 – .NET version

## Installing SQL Server Express

**SQL Server Express** is the free edition of SQL Server. To install **SQL Server Express**, do the following:

1. Download the **SQL Server installer** from `https://www.microsoft.com/en-us/sql-server/sql-server-downloads`.
2. After the download completes, run the SQL Server installer.

3. Select the **Basic** installation type:

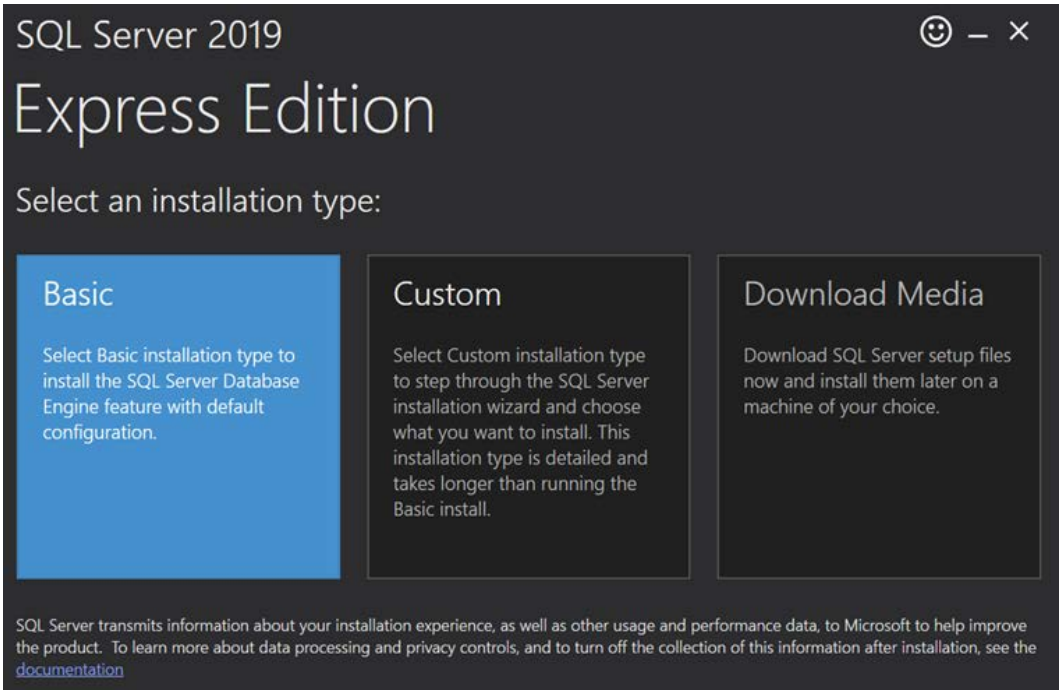



Figure 1.6 – The SQL Server installer

4. Click the **Accept** button to accept the Microsoft SQL Server License Terms.
5. Click the **Install** button to complete the installation.

The following screenshot shows the dialog that appears after SQL Server Express has been successfully installed:

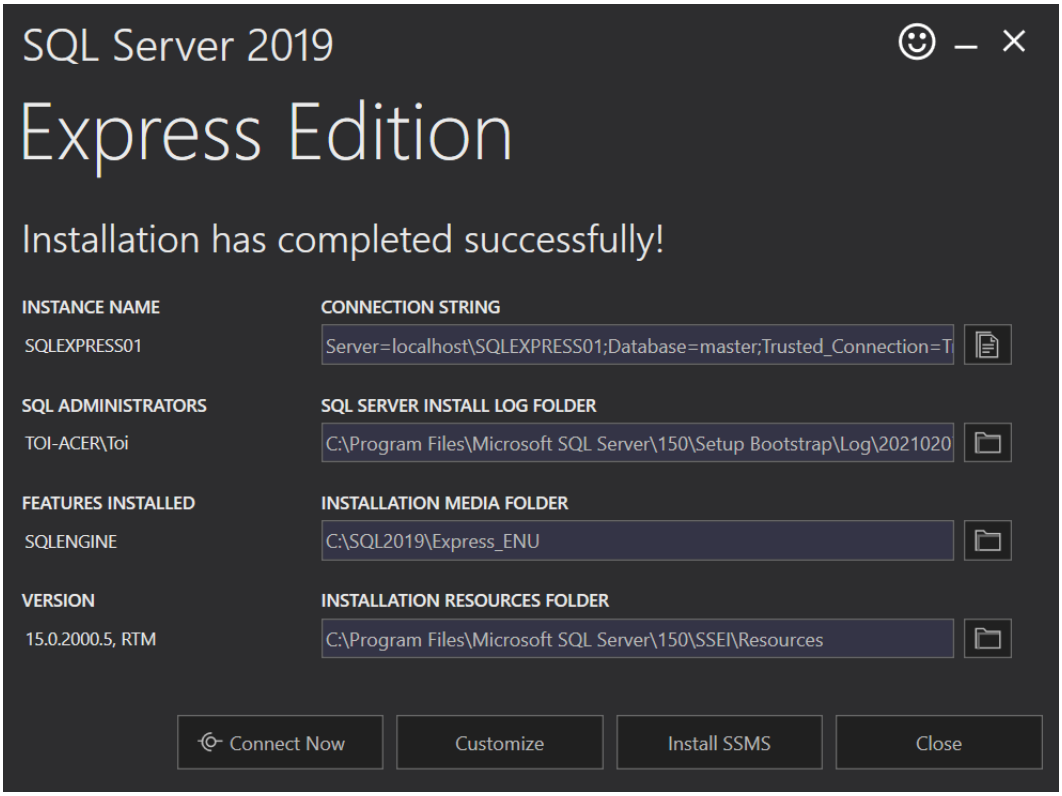



Figure 1.7 – SQL Server Express Edition

To complete all the projects in this book, you will require a code editor, such as Visual Studio 2019, .Net 5.0, or SQL Server. In this chapter, we showed you how to install Visual Studio 2019 Community Edition, .NET 5.0, and SQL Server 2019 Express Edition.

# Summary

After completing this chapter, you should understand the benefits of using Blazor WebAssembly versus other web development frameworks and be prepared to complete the projects in this book.

In this chapter, we introduced the **Blazor** framework. The Blazor framework is built on .NET Framework and allows developers to use C# on both the frontend and backend of a web app.

After that, we compared Blazor Server with Blazor WebAssembly. Blazor WebAssembly has many advantages over Blazor Server due to the fact that it runs on the browser while Blazor Server runs on the server. A Blazor WebAssembly web app can run offline and feels much more like a native application because all of the code is run directly on the browser. Finally, a Blazor WebAssembly app can be easily converted into a PWA.

In the last part of the chapter, we explained how to set up your computer with Visual Studio 2019 Community Edition, .NET 5.0, and SQL Server 2019 Express, all of which are required to complete the projects in this book.

Now that your computer is set up to create a Blazor WebAssembly web app, it is time to get started. In the next chapter, you will create your first Blazor WebAssembly web app.

# Questions

The following questions are provided for your consideration:

1. Does using Blazor WebAssembly mean that you never need to write JavaScript ever again?
2. Does Blazor WebAssembly require that any plugins be installed on the browser?
3. How much does it cost to get started developing with Blazor WebAssembly?

# Further reading

The following resources provide more information concerning the topics in this chapter:

- For more information on **Blazor**, refer to `https://blazor.net`.
- For more information on **the .NET Foundation**, refer to `https://dotnetfoundation.org`.

- For more information on the **ASP.NET** repository on **GitHub**, refer to `https://github.com/dotnet/aspnetcore`.
- For more information on **SignalR**, refer to `https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction`.
- For more information on **PWAs**, refer to `https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps`.
- For general information on **WebAssembly**, refer to `https://webassembly.org`.
- For more information on the **W3C WebAssembly Core Specification**, refer to `https://www.w3.org/TR/wasm-core`.
- For more information on browser compatibility with **WebAssembly**, refer to `https://caniuse.com/?search=wasm`.

# 2

# Building Your First Blazor WebAssembly Application

Razor components are the building blocks of Blazor WebAssembly applications. A Razor component is a chunk of user interface that can be shared, nested, and reused. Razor components are ordinary C# classes and can be placed anywhere in a project.

In this chapter, we will learn about Razor components. We will learn how to use them, how to apply parameters, and about their life cycle and their structure. We will learn how to use the `@page` directive to define routing. We will also learn how to use Razor syntax to combine C# code with HTML markup.

The Blazor WebAssembly project in this chapter will be created by using the **Blazor WebAssembly App** project template provided by Microsoft. After we create the project, we will examine it to further familiarize ourselves with Razor components. We will learn how to use components, how to add parameters, how to apply routing, how to use Razor syntax, and how to separate the Razor markup and code into separate files. Finally, we will configure our own custom project template that creates an empty Blazor WebAssembly project.

In this chapter, we will cover the following topics:

- Razor components
- Routing
- Razor syntax
- Using the Blazor App project template
- Creating an empty Blazor WebAssembly project template

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free Community edition of Visual Studio 2019, refer to *Chapter 1*, *Introduction to Blazor WebAssembly*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter02`.

The code in action video is available here: `https://bit.ly/3bEZrrg`.

# Razor components

Blazor WebAssembly is a component-driven framework. Razor components are the fundamental building blocks of a Blazor WebAssembly application. They are classes that are implemented using a combination of C#, HTML, and Razor markup. When the web app loads, the classes get downloaded into the browser as normal .NET assemblies (DLLs).

> **Important note**
>
> In this book, the terms *Razor component* and *component* are used interchangeably.

## Using components

HTML element syntax is used to add a component to another component. The markup looks like an HTML tag where the name of the tag is the component type.

The following markup in the `Pages\Index.razor` file of the `Demo` project that we will create in this chapter will render a `SurveyPrompt` instance:

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

The preceding `SurveyPrompt` element includes an attribute parameter named `Title`.

## Parameters

Component parameters are used to make components dynamic. Parameters are public properties of the component that are decorated with either the `Parameter` attribute or the `CascadingParameter` attribute. Parameters can be simple types, complex types, functions, `RenderFragments`, or event callbacks.

The following code for a component named `HelloWorld` includes a parameter named `Text`:

**HelloWorld.razor**

```
<h1>Hello @Text!</h1>
@code {
    [Parameter] public string Text { get; set; }
}
```

To use the `HelloWorld` component, include the following HTML syntax in another component:

```
<HelloWorld Text="World" />
```

In the preceding example, the `Text` attribute of the `HelloWorld` component is the source of the `Text` parameter. This screenshot shows the results of using the component as indicated:


Figure 2.1 – HelloWorld component

A component can also receive parameters from its route or a query string. You will learn more about the different types of parameters later in this chapter.

## Naming components

The name of a Razor component must be in title case. Therefore, `helloWorld` would not be a valid name for a Razor component since the *h* is not capitalized. Also, Razor components use the `RAZOR` extension rather than the `CSHTML` extension that is used by Razor Pages.

> **Important note**
> Razor components must start with a capital letter.

## Component life cycle

Razor components inherit from the `ComponentBase` class. The `ComponentBase` class includes both asynchronous and synchronous methods used to manage the life cycle of a component. In this book, we will be using the asynchronous versions of the methods since they execute without blocking other operations. This is the order in which the methods in the life cycle of a component are invoked:

1. `SetParameterAsync`: This method sets the parameters that are supplied by the component's parent in the render tree.
2. `OnInitializedAsync`: This method is invoked after the component is first rendered.
3. `OnParametersSetAsync`: This method is invoked after the component initializes *and* each time the component re-renders.
4. `OnAfterRenderAsync`: This method is invoked after the component has finished rendering. This method is for working with JavaScript since JavaScript requires the **Document Object Model** (**DOM**) elements to be rendered before they can do any work.

## Component structure

The following diagram shows code from the `Counter` component of the `Demo` project that we will create in this chapter:

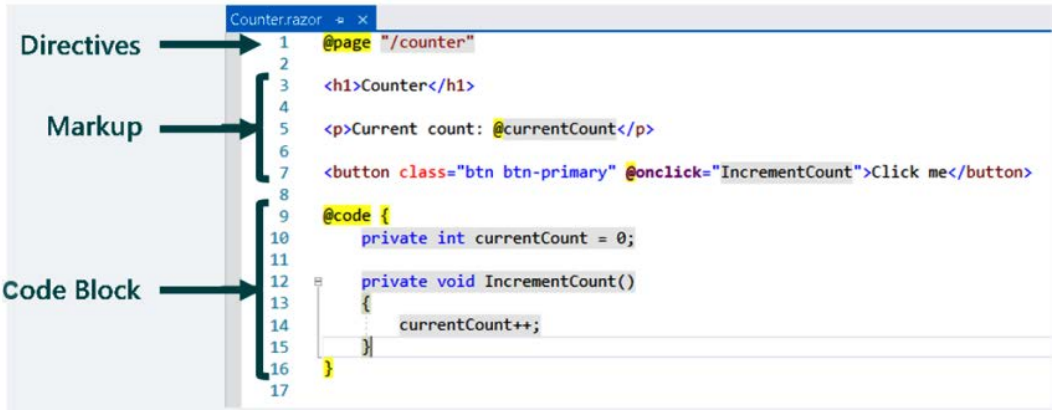



Figure 2.2 – Component structure

The code in the preceding example is divided into three sections:

- **Directives**
- **Markup**
- **Code Block**

Each of the sections has a different purpose.

## Directives

Directives are used to add special functionality, such as routing, layout, and dependency injection. They are defined within Razor and you cannot define your own directives.

In the preceding example, there is only one directive used – the `@page` directive. The `@page` directive is used for routing. In this example, the following URL will route the user to the `Counter` component:

```
/counter
```

A typical page can include many directives at the top of the page. Also, many pages have more than one `@page` directive.

Most of the directives in Razor can be used in a Blazor WebAssembly application. These are the Razor directives that are used in Blazor, in alphabetical order:

- `@attribute`: This directive adds a class-level attribute to the component. The following example adds the `[Authorize]` attribute:

  ```
  @attribute [Authorize]
  ```

- `@code`: This directive adds class members to the component. In the example, it is used to distinguish the code block.
- `@implements`: This directive implements the specified class.
- `@inherits`: This directive provides full control of the class that the view inherits.
- `@inject`: This directive is used for dependency injection. It enables the component to inject a service from the dependency injection container into the view. The following example injects `HttpClient` defined in the `Program.cs` file into the component:

  ```
  @inject HttpClient Http
  ```

- `@layout`: This directive is used to specify a layout for the Razor component.

- `@namespace`: This directive sets the component's namespace. You only need to use this directive if you do not want to use the default namespace for the component. The default namespace is based on the location of the component.
- `@page`: This directive is used for routing.
- `@typeparam`: This directive sets a type parameter for the component.
- `@using`: This directive controls the components that are in scope.

### Markup

This is HTML with Razor syntax. The Razor syntax can be used to render text and allows C# to be used as part of the markup. We will cover more about Razor syntax later in this chapter.

### Code block

The code block contains the logic for the page. It begins with the `@code` directive. By convention, the `@code` directive is at the bottom of the page. It is the only file-level directive that is not placed at the top of the page.

The code block is where we add C# fields, properties, and methods to the component. Later in this chapter, we will move the code block to a separate code-behind file.

Razor components are the building blocks of a Blazor WebAssembly application. They are easy to use since they are simply a combination of HTML markup and C# code. In the next section, we will see how routing is used to navigate between each of the components.

## Routing in Blazor WebAssembly

In Blazor WebAssembly, routing is handled on the client, not on the server. As you navigate in the browser, Blazor intercepts that navigation and renders the component with the matching route.

The URLs are resolved relative to the base path that is specified in the `wwwroot/index.html` file. It is specified in the `head` element using the following syntax:

```
<base href="/" />
```

Unlike other frameworks that you may have used, the route is not inferred from the location of its file. For example, in the `Demo` project, the `Counter` component is in the `/Pages/Counter` folder, yet it uses the following route:

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

## Route parameters

The `Router` component uses route parameters to populate the parameters of the corresponding component. The parameters of both the component and the route must have the same name, but they are not case sensitive.

Since optional route parameters are not supported, you may need to provide more than one `@page` directive to a component to simulate optional parameters. The following example shows how to include multiple `@page` parameters:

**RoutingExample.razor**

```
@page "/routing"
@page "/routing/{text}"
<h1>Blazor WebAssembly is @Text!</h1>
@code {
    [Parameter] public string Text { get; set; }
    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

In the preceding code, the first `@page` directive allows navigation to the component without a parameter and the second `@page` directive allows a route parameter. If a value for `text` is provided, it is assigned to the `Text` property of the component. If the `Text` property of the component is `null`, it is set to `fantastic`.

The following URL will route the user to the `RoutingExample` component:

```
/routing
```

The following URL will also route the user to the `RoutingExample` component, but this time the `Text` parameter will be set by the route:

```
/routing/amazing
```

This screenshot shows the results of using the indicated route:


Figure 2.3 – RoutingExample component

> **Important note**
> Route parameters are not case sensitive.

## Catch-all route parameters

Catch-all route parameters are used to capture paths across multiple folder boundaries. This type of route parameter is a `string` type and can only be placed at the end of the URL.

This is a sample component that uses a catch-all route parameter:

**CatchAll.razor**

```
@page "/{*path}"
<h1>Catch All</h1>
Route: @Path
@code {
    [Parameter] public string Path { get; set; }
}
```

For the `/error/type/3` URL, the preceding code will set the value of the `Path` parameter to `error/type/3`:

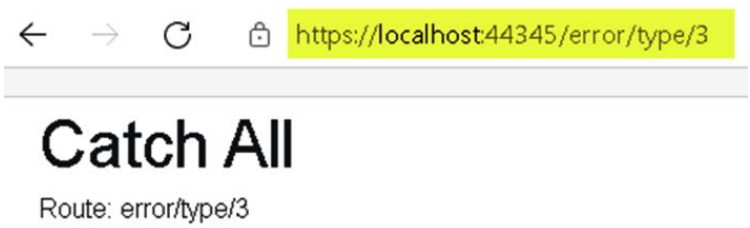



Figure 2.4 – Catch-all route parameter example

## Route constraints

Route constraints are used to enforce the datatype of a route parameter. To define a constraint, add a colon followed by the constraint type to the parameter. In the following example, the route is expecting a route parameter named `Increment` with the type of `int`:

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

The following route constraints are supported:

| Constraint | .NET Type | Sample Valid Values |
|---|---|---|
| bool | System.Boolean | TRUE, false |
| datetime | System.DateTime | 2020-12-31, 2020-12-31 5:26pm |
| decimal | System.Decimal | 42.99, -1,000.01 |
| double | System.Double | 1.234, -1,001.01e8 |
| float | System.Single | 1.234, -1,001.01e8 |
| guid | System.Guid | CD2C1638-1638-72D5-1638-DEADBEEF1638, {CD2C1638-1638-72D5-1638-DEADBEEF1638} |
| int | System.Int32 | 123456789, -123456789 |

Figure 2.5 – Supported route constraints

The following types are not currently supported as constraints:

- Regular expressions
- Enums
- Custom constraints

Routing is handled on the client. We can use both route parameters and catch-all route parameters to enable routing. Route constraints are used to ensure that a route parameter is of the required datatype. Razor components use Razor syntax to seamlessly merge HTML with C# code, which is what we will see in the next section.

# Razor syntax

Razor syntax is made up of HTML, Razor markup, and C#. Rendering HTML from a Razor component is the same as rendering HTML from an HTML file. The HTML in a Razor component is rendered by the server unchanged. Razor syntax uses both inline expressions and control structures.

## Inline expressions

Inline expressions start with an `@` symbol followed by a variable or function name. This is an example of an inline expression:

```
<h1>Blazor is @Text!</h1>
```

## Control structures

Control structures also start with an `@` symbol. The content within the curly brackets is evaluated and rendered to the output. This is an example of an `if` statement from the `FetchData` component in the `Demo` project:

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

Each code statement within a Razor code block must end with a semicolon. C# code is case sensitive and strings must be enclosed in quotation marks.

### Conditionals

The following types of conditionals are included in Razor syntax:

- `if` statements
- `switch` statements

This is an example of an `if` statement:

```
@if (DateTime.Now.DayOfWeek.ToString() != "Friday")
{
    <p>Today is not Friday.</p>
}
else if (DateTime.Now.Day != 13)
{
    <p>Today is not the 13th.</p>
}
else
{
    <p>Today is Friday the 13th.</p>
}
```

The preceding code uses an if statement to check it the current day of the week is Friday and/or the current day of the month is the 13th.

This is an example of a `switch` statement:

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 42:
        <p>Your number is 42!</p>
        break;
    default:
        <p>Your number was not 1 or 42.</p>
        break;
}
@code {
    private int value = 2;
}
```

The preceding `switch` statement compares the `value` variable to `1` and `42`.

## Loops

The following types of loops are included in Razor syntax:

- `for` loops
- `foreach` loops
- `while` loops
- `do while` loops

Each of the following examples uses an array of the `WeatherForecast` type. `WeatherForecast` includes a `Summary` property and is defined in the `Demo` project.

This is an example of a `for` loop:

```
@for (var i = 0; i < forecasts.Count(); i++)
{
   <div> forecasts[i].Summary</div>
};
@code {
    private WeatherForecast[] forecasts;
}
```

This is an example of a `foreach` loop:

```
@foreach (var forecast in forecasts)
{
    <div>@forecast.Summary</div>
};
@code {
    private WeatherForecast[] forecasts;
}
```

This is an example of a `while` loop:

```
@while (i < forecasts.Count())
{
    <div>@forecasts[i].Summary</div>
    i++;
};
@code {
    private WeatherForecast[] forecasts;
    private int i = 0;
}
```

This is an example of a `do while` loop:

```
@do
{
    <div>@forecasts[i].Summary</div>
    i++;
} while (i < forecasts.Count());
```

```
@code {
    private WeatherForecast[] forecasts;
    private int i = 0;
}
```

Razor syntax is easy to learn if you already know C#. It includes both inline expressions and control structures such as conditionals and loops.

# Project overview

The Blazor WebAssembly application that we are going to build in this chapter is a simple three-page application. Each page will be used to demonstrate one or more features of Razor components.

This is a screenshot of the completed `Demo` project:

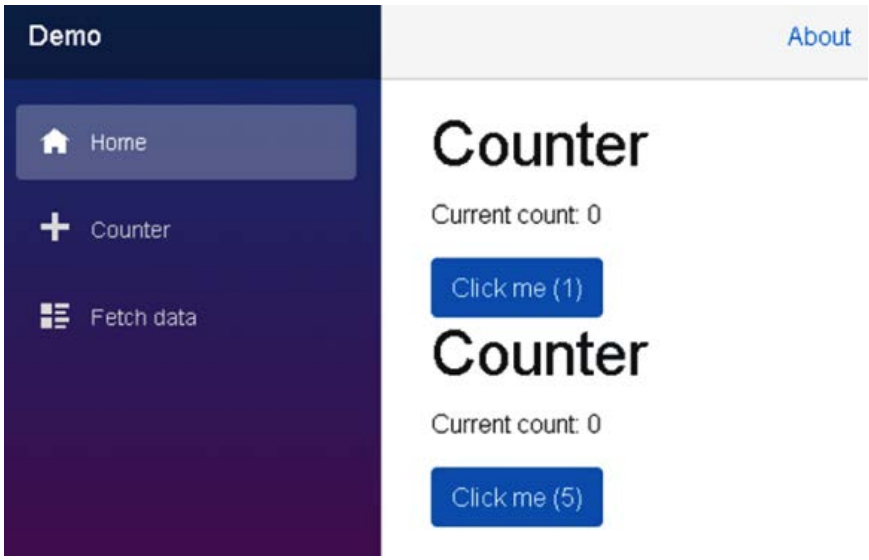



Figure 2.6 – Home page of the Demo project

After we have completed the `Demo` project, we will convert it into an empty Blazor WebAssembly project. The empty Blazor WebAssembly project will be used as the basis for a custom Blazor WebAssembly App project template.

# Creating the Demo Blazor WebAssembly project

The `Demo` project that we are creating is based on one of the many sample projects that are provided by the **Blazor WebAssembly App** project template. After we have used the template to create the project, we will examine the files in the sample project and update some of the files to demonstrate how to use Razor components. Finally, we will separate the code block of one of the components into a separate file to demonstrate how to use the code-behind technique to separate the markup from the code.

## Creating the Demo project

Visual Studio comes with quite a few project templates. We are going to use the **Blazor WebAssembly App** project template to create our first Blazor WebAssembly project. Since this project template can be used to create many different types of Blazor projects, it is important to follow the instructions carefully:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt*+*S*) textbox, enter `Blazor` and hit the *Enter* key.

   The following screenshot shows the **Blazor WebAssembly App** project template that we will be using:

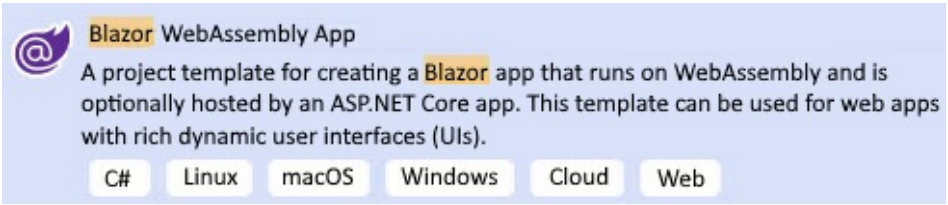



Figure 2.7 – **Blazor WebAssembly App** project template

4. Select the **Blazor WebAssembly App** project template and click the **Next** button.

5. Enter `Demo` in the **Project name** textbox and click the **Next** button.

   This is a screenshot of the dialog used to configure our new project:

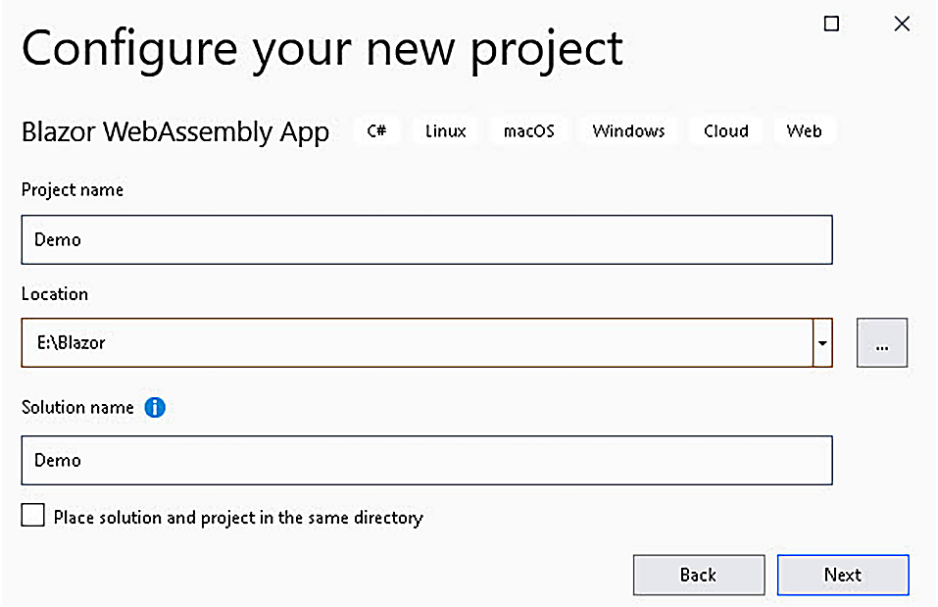



Figure 2.8 – The Configure your new project dialog

> **Tip**
>
> In the preceding example, we placed the `Demo` project into the `E:\Blazor` folder. However, the location of this project is not important.

6. Select **.NET 5.0** as the version of the .NET Framework to use.

   This is a screenshot of the dialog used to create our new Blazor WebAssembly app:

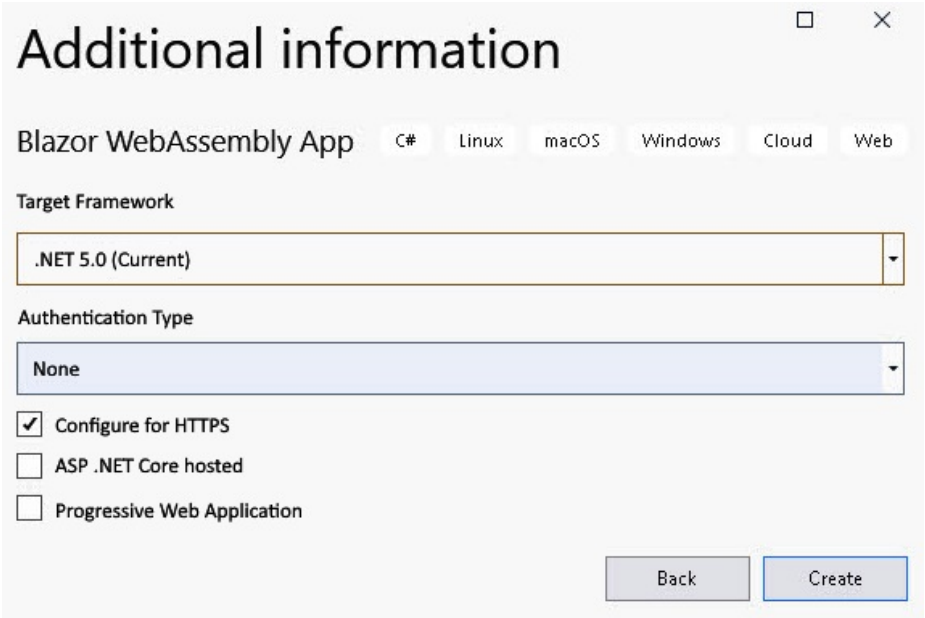



Figure 2.9 – Additional information for the Blazor WebAssembly App dialog

7. Click the **Create** button.

You have created the `Demo` Blazor WebAssembly project.

## Running the Demo project

Once the project has been created, you need to run it to get an understanding of what it does. The `Demo` project contains three pages: **Home**, **Counter**, and **Fetch data**:

1. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run the `Demo` project.

   This is a screenshot of the **Home** page from the `Demo` project:

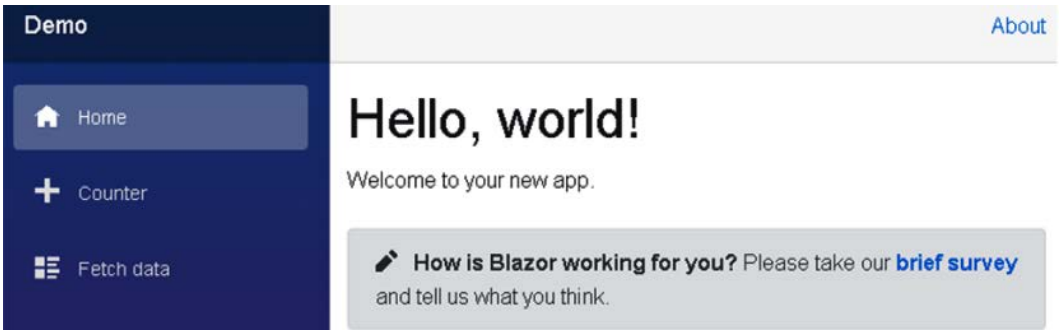



Figure 2.10 – The Home page

The **Home** page is split into two sections. The navigation menu is on the left side of the page and the body is on the right side of the page. The body of the **Home** page consists of some static text and a link to a survey.

2. Click the **Counter** option on the navigation menu to navigate to the **Counter** page.

   This is a screenshot of the **Counter** page from the `Demo` project:

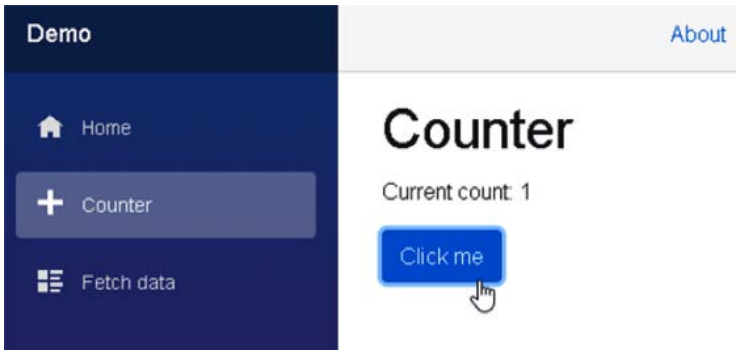



Figure 2.11 – The Counter page

The body of the **Counter** page includes a counter and a **Click me** button. Each time the button on the **Counter** page is clicked, the counter is incremented without a page refresh.

> **Important note**
> Since this is a **Single-Page Application** (**SPA**), only the section of the page that needs to be updated is updated.

3. Click the **Fetch data** option on the navigation menu to navigate to the **Fetch data** page.

This is a screenshot of the **Fetch data** page from the `Demo` project:

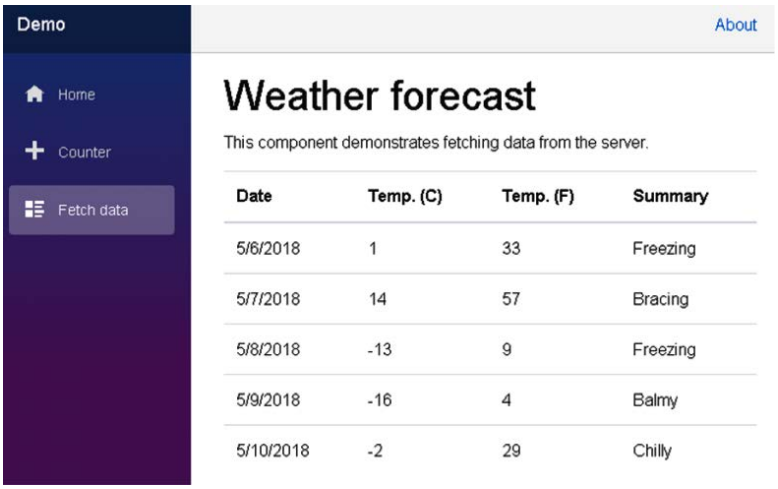



Figure 2.12 – The Fetch data page

The body of the **Fetch data** page includes a table that shows the weather forecast for a few days in 2018. As you will see, the data displayed in the table is static data from the `wwwroot\sample-data\weather.json` file.

## Examining the Demo project's structure

Now let's return to Visual Studio to examine the files in the `Demo` project.

The following figure shows the project's structure:

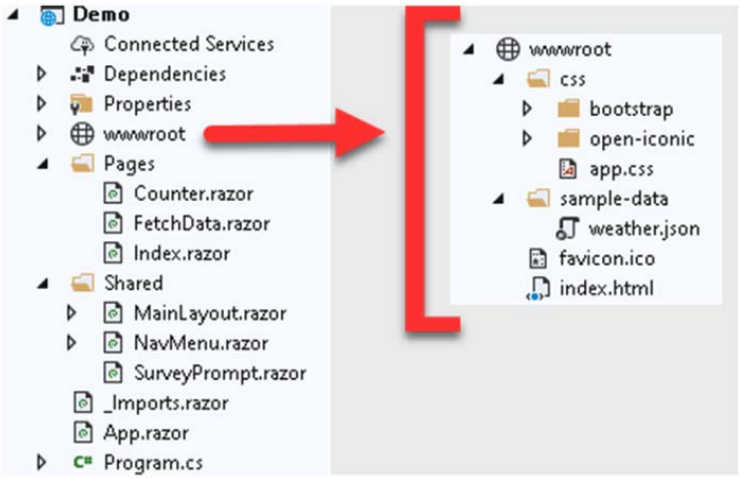



Figure 2.13 – Project structure

The project includes quite a few files with some of them divided into their own folders. Let's examine them.

## The wwwroot folder

The `wwwroot` folder is the application's web root. Only the files in this folder are web-addressable. The `wwwroot` folder contains a collection of **Cascading Style Sheets** (**CSS**) files, a sample data file, an icon file, and `index.html`. Later in this book, in addition to these types of files, we will use this folder for public static resources such as images and JavaScript files.

The `index.html` file is the root page of the web application. Whenever a page is initially requested, the contents of the `index.html` page are rendered and returned in the response. The `head` element of the `index.html` file includes links to each of the CSS files in the `css` folder and specifies the base path to use for the web app. The `body` element of the `index.html` file includes two `div` elements and a reference to the `blazor.webassembly.js` file.

This is the code in the `body` element of the `index.html` file:

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

The highlighted `div` element in the preceding code loads the `App` component.

The `blazor-error-ui` `div` element is for displaying unhandled exceptions. The styling for this `div` element is in the `wwwroot\css\app.css` file. The `blazor.webassembly.js` file is the script that downloads the .NET runtime, your application's assemblies, and your application's dependencies. It also initializes the runtime to run the web app.

## The App component

The `App` component is defined in the `App.razor` file:

**App.razor**

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

The `App` component is the root component of a Blazor WebAssembly application. It uses the `Router` component to set up the routing for the web app. In the preceding code, if the route is found, the `RouteView` component receives `RouteData` and renders the specified component using the indicated `DefaultLayout`. If the route is not found, the `NotFound` template is used and `LayoutView` is rendered using the indicated `Layout`.

As you can see, in the `Demo` project, both the `Found` template and the `NotFound` template are using the same layout. They are both using the `MainLayout` component. However, they do not need to use the same layout component.

## The Shared folder

The `Shared` folder in the `Demo` project includes the shared user interface Razor components, including the `MainLayout` component. Each of these components may be used one or more times by other Razor components.

## The Pages folder

The `Pages` folder includes the routable Razor components used by the project. The routable components are `Counter`, `FetchData`, and `Index`. Each of these components includes an `@page` directive that is used to route the user to the page.

### The _Imports.razor file

This file includes Razor directives such as the `@using` directive for namespaces. Your project can include multiple `_Imports.razor` files. Each one is applied to its current folder and subfolders. Any `@using` directives in the `_Imports.razor` file are only applied to Razor (`RAZOR`) files. They are not applied to C# (`CS`) files. This distinction is important when using the code-behind technique.

### The Program.cs file

The `Program.cs` file is the entry point for the application.

## Examining the shared Razor components

The shared Razor components are in the `Shared` folder. There are three shared Razor components in the `Demo` project:

- The `MainLayout` component
- The `NavMenu` component
- The `SurveyPrompt` component

### The MainLayout component

The `MainLayout` component is used to define the page layout for the web app:

**Pages/MainLayout.razor**

```
@inherits LayoutComponentBase
<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>
    <div class="main">
        <div class="top-row px-4">
            <a href="http://blazor.net"
               target="_blank"
               class="ml-md-auto">About</a>
        </div>
        <div class="content px-4">
            @Body
```

```
            </div>
        </div>
    </div>
```

The `MainLayout` component inherits from the `LayoutComponentBase` class. `LayoutComponentBase` represents a layout and has only one property, which is the `Body` property. The `Body` property gets the content to be rendered inside the layout.

The following diagram illustrates the layout of a page as defined by the `MainLayout` component:

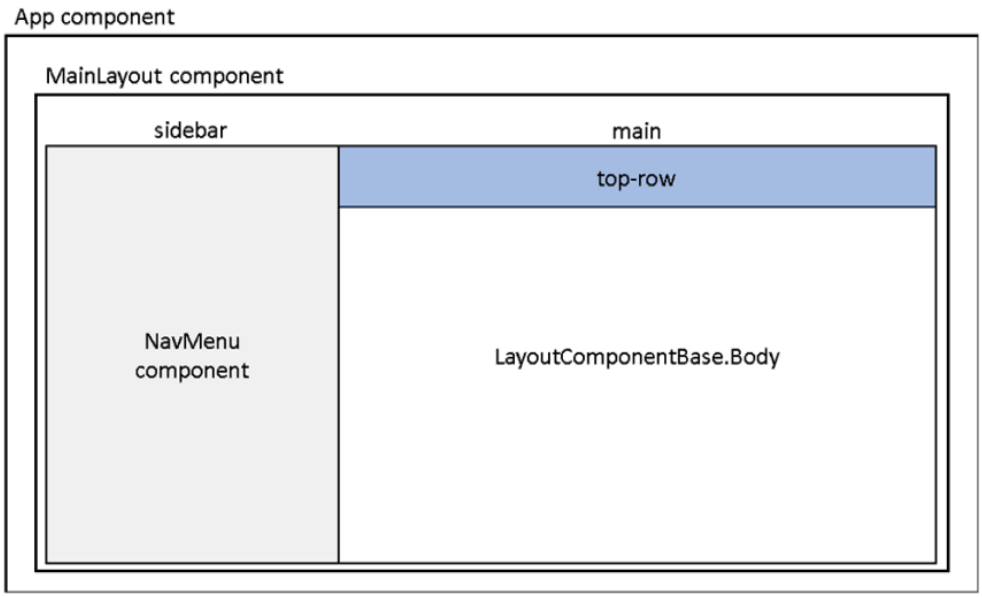



Figure 2.14 – Screen layout

**Tip**

The **Blazor WebAssembly App** project template uses **Bootstrap 4** to style its pages. If you are unfamiliar with Bootstrap 4, you should refer to `https://getbootstrap.com` to familiarize yourself with its syntax.

## The NavMenu component

The `NavMenu` component defines the navigation menu for the `Demo` project. It uses multiple `NavLink` components to define the various menu options. This is the section of the `NavMenu` component that references the `NavLink` components used for the project's navigation:

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href=""
                     Match="NavLinkMatch.All">
                <span class="oi oi-home"
                      aria-hidden="true"></span> Home
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="counter">
                <span class="oi oi-plus"
                      aria-hidden="true"></span> Counter
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="fetchdata">
                <span class="oi oi-list-rich"
                      aria-hidden="true"></span> Fetch data
            </NavLink>
        </li>
    </ul>
</div>
```

The `NavLink` component is defined in the `Microsoft.AspNetCore.Components.Routing` namespace. It behaves like an `a` element, except it has added functionality that highlights the current URL. This is the HTML that is rendered by `NavLink` for the `Counter` component when the `Counter` component is selected:

```
<a href="counter" class="nav-link active">
    <span class="oi oi-plus" aria-hidden="true"></span>
    Counter
</a>
```

The style used for the `nav-link` class is from Bootstrap. The style used for the `active` class is defined in the `wwwroot\css\app.css` file:

```
.sidebar .nav-item a.active {
    background-color: rgba(255,255,255,0.25);
    color: white;
}
```

### The SurveyPrompt component

The `SurveyPrompt` component creates a link to a brief survey on Blazor.

## Examining the routable Razor components

The routable Razor components are in the `Pages` folder. There are three routable Razor components in the `Demo` project:

- The `Index` component
- The `Counter` component
- The `FetchData` component

## The Index component

The `Home` page of the `Demo` project uses the `Index` component that is defined in the `Pages\Index.razor` file:

**Pages\Index.razor**

```
@page "/"
<h1>Hello, world!</h1>
Welcome to your new app.
<SurveyPrompt Title="How is Blazor working for you?" />
```

The preceding code includes an `@page` directive that references the root of the web app and some markup. The markup includes a `SurveyPrompt` component.

## The Counter component

The `Counter` component is more complex than the `Index` component. Similar to the `Index` component, it contains an `@page` directive that is used for routing and some markup. However, it also contains a C# code block:

**Pages\Counter.razor**

```
@page "/counter"
<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">
    Click me
</button>
<a href="counter" class="nav-link active">
    <span class="oi oi-plus" aria-hidden="true"></span>
    Counter
</a>
@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
```

```
            currentCount++;
        }
    }
```

In the preceding code block, a private `currentCount` variable is used to hold the number of clicks. Each time the `Counter` button is clicked, the `Counter` component's registered `@onclick` handler is called. In this case, it is the `IncrementCount` method.

The `IncrementCount` method increments the value of the `currentCount` variable and the `Counter` component regenerates its render tree. Blazor compares the new render tree against the previous one and applies any modifications to the browser's DOM. This results in the displayed count being updated.

## The FetchData component

The `FetchData` component is by far the most complex component in the `Demo` project.

These are the directives in the `Pages\FetchData.razor` file:

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

The `@page` directive is used for routing and the `@inject` directive is used for dependency injection. In this component, `HttpClient` that is defined in the `Program.cs` file is being injected into the view. For more information on dependency injection, refer to *Chapter 6*, *Building a Shopping Cart Using Application State*.

The following markup demonstrates the use of a very important pattern that you will often use when developing a Blazor WebAssembly application. Because the application runs on the browser, all data access must be asynchronous. That means that when the page first loads, the data will be null. For that reason, you need to test for the null case before attempting to process the data.

This is the markup in the `Pages\FetchData.razor` file:

```
<h1>Weather forecast</h1>
<p>This component demonstrates fetching data from the server.</
p>
@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 preceding markup includes an `if` statement and a `foreach` loop. While the value of `forecasts` is `null`, a **Loading** message is displayed. If you do not handle the case when the value of `forecasts` is `null`, the framework will throw an exception. Once the value of `forecasts` is no longer `null`, all of the items in the array are presented in a table.

> **Important note**
>
> The value of `forecasts` will be `null` the first time that the page is rendered.

As previously mentioned, Blazor components have a well-defined life cycle. The `OnInitializedAsync` method is invoked when the component is rendered. After the `OnInitializedAsync` method completes, the component is re-rendered.

This is the code block in the `Pages\FetchData.razor` file:

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

First, the preceding code block declares a parameter to contain an array of the type `WeatherForecast`. Next, it uses the `OnInitializedAsync` asynchronous method to populate the array. In order to populate the array, the `GetFromJsonAsync` method of the `HttpClient` service is used. For more information on `HttpClient`, refer to *Chapter 8*, *Building a Task Manager Using the ASP.NET Web API*.

## Using a component

Razor components are used by including them in the markup of another component. We will add a `Counter` component to the `Home` page. We do this as follows:

1. Return to Visual Studio.
2. Open the `Pages\Index.razor` file.
3. Delete all of the markup. Be sure you do not remove the `@page` directive at the top of the file.

4. Add the following markup below the `@page` directive:

```
<Counter />
```

5. From the **Build** menu, select the **Rebuild Solution** option.
6. Return to the browser and navigate to the **Home** page. If the `Demo` project is not still running, from the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run it.
7. Use *Ctrl* + *R* to refresh the browser.

> **Tip**
> Whenever you update your C# code, you need to refresh the browser for the browser to load the updated DLL.

8. Click the **Click me** button to test the `Counter` component.

## Adding a parameter to a component

Most components require parameters. To add a parameter to a component, use the `Parameter` attribute. We will add a parameter to specify the increment used by the `IncrementCount` method. We do this as follows:

1. Return to Visual Studio.
2. Open the `Pages\Counter.razor` file.
3. Add the following code to the top of the code block to define the new parameter:

```
[Parameter] public int? Increment { get; set; }
private int increment = 1;
```

4. Update the `IncrementCount` method to the following:

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

5. Add the following `OnParametersSet` method to set the value of `increment` to the value of the `Increment` parameter:

```
protected override void OnParametersSet()
{
    if (Increment.HasValue)
        increment = Increment.Value;
}
```

6. Add the highlighted text to the markup of the **Click me** button to display the current value of the `increment` variable:

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

## Using a parameter with an attribute

We will add another instance of the `Counter` component to the `Home` page that uses the new parameter. We do this as follows:

1. Open the `Pages\Index.razor` file.
2. Add the following markup to the bottom of the `Index.razor` file:

```
<Counter Increment="5"/>
```

   As you add the markup, **IntelliSense** is provided for the new `Increment` parameter:


Figure 2.15 – IntelliSense

3. From the **Build** menu, select the **Build Solution** option.
4. Return to the browser.
5. Use *Ctrl* + *R* to refresh the browser.
6. Navigate to the **Home** page.

The **Home** page now contains two instances of the `Counter` component. If you click the first **Click me** button, the first counter will be incremented by 1; if you click the second **Click me** button, the second counter will be incremented by 5:


Figure 2.16 – The Home page

7. Click each of the **Click me** buttons to verify they both work as intended.

## Adding a route parameter

Components can have multiple `@page` directives. We will add an `@page` directive to the `Counter` component that uses a parameter. We do this as follows:

1. Return to Visual Studio.
2. Open the `Pages/Counter.razor` file.
3. Add the following `@page` directive to the top of the file:

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

   The `Counter` component now includes two `@page` directives.

4. From the **Build** menu, select the **Build Solution** option.
5. Return to the browser.
6. Navigate to the **Counter** page.
7. Update the URL to the following:

```
/counter/4
```

> **Important note**
> Since the page is automatically reloaded when you change the URL, you do not need to refresh the browser in order to reload the page.

8. Click the **Click me** button.

   The counter should now increment by `4`.

9. Update the URL to an invalid route:

```
/counter/a
```

   Since this is not a valid route, you will be directed to the `NotFound` content defined in the `App` component:

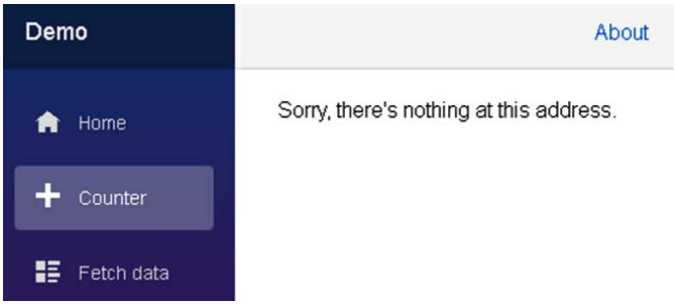



Figure 2.17 – Page not found

> **Tip**
> If you need to navigate to a URL in code, you should use `NavigationManager`. `NavigationManager` provides a `NavigateTo` method that is used to navigate the user to the specified URI without forcing a page load.

## Using partial classes to separate markup from code

Many developers prefer to separate their markup from their C# fields, properties, and methods. Since Razor components are regular C# classes, they support partial classes. The `partial` keyword is used to create a partial class. We will use a partial class to move the code block from the `RAZOR` file to a `CS` file. We do this as follows:

1. Return to **Visual Studio**.
2. Right-click the **Pages** folder and select **Add, Class** from the menu.
3. Name the new class `Counter.razor.cs`.

4. Update the `Counter` class to be a partial class by using the `partial` keyword:

```
public partial class Counter{}
```

5. Open the `Pages/Counter.razor` file.
6. Copy all of the code in the code block to the partial `Counter` class in the `Counter.razor.cs` file.
7. Delete the code block from the `Counter.razor` file.
8. Add the following `using` statement to the `Counter.razor.cs` file:

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

9. From the **Build** menu, select the **Build Solution** option.
10. Return to the browser.
11. Use *Ctrl* + *R* to refresh the browser.
12. Navigate to the **Counter** page.
13. Click the **Click me** button to verify that it still works.
14. Close the browser.

Using partial classes gives you the flexibility to move the code in the code block to a separate file, allowing you to use the code-behind technique.

We have created a `Demo` project by using the **Blazor WebAssembly App** project template provided by Microsoft. We added a parameter to the `Counter` component and moved the code in the code block of the `Counter` component to a separate file.

# Creating a custom Blazor WebAssembly project template

As you have seen, the `Demo` Blazor WebAssembly project created by the **Blazor WebAssembly App** project template includes quite a few files. In later chapters, we will want to start with an empty Blazor project. So, we will create our own project template that creates an empty Blazor WebAssembly project.

## Creating an empty Blazor project

We need to create an empty Blazor WebAssembly project to base our new project template on. We do this as follows:

1. Return to Visual Studio.
2. Delete the `wwwroot\sample-data` folder.
3. Delete all of the components in the `Pages` folder, except for the `Index` component.
4. Open the `Index.razor` file.
5. Delete all of the markup from the `Index` component. Make sure that you do not delete the `@page` directive at the top of the page.
6. Delete the `Shared\SurveyPrompt.razor` file.
7. Open the `Shared\MainLayout.razor` file.
8. Remove the `About` link from the top row of the layout by removing the following markup:

```
<a href="http://blazor.net" target="_blank"
   class="ml-md-auto">
        About
</a>
```

9. Open the `Shared\NavMenu.razor` file.
10. Remove the `li` elements for the `Counter` and `Fetch data` pages.
11. From the **Build** menu, select the **Build Solution** option.
12. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run the `Demo` project.

    The `Demo` project is now empty. It only contains a blank `Home` page.

## Creating a project template

The **Export Template Wizard** is used to create custom project templates. We will use the empty project that we just created as the basis for a custom project template. We do this as follows:

1. Return to Visual Studio.
2. From the **Project** menu, select the **Export Template** option to open the **Export Template Wizard window**.
3. Select **Project template** on the **Choose Template Type** dialog and click the **Next** button:

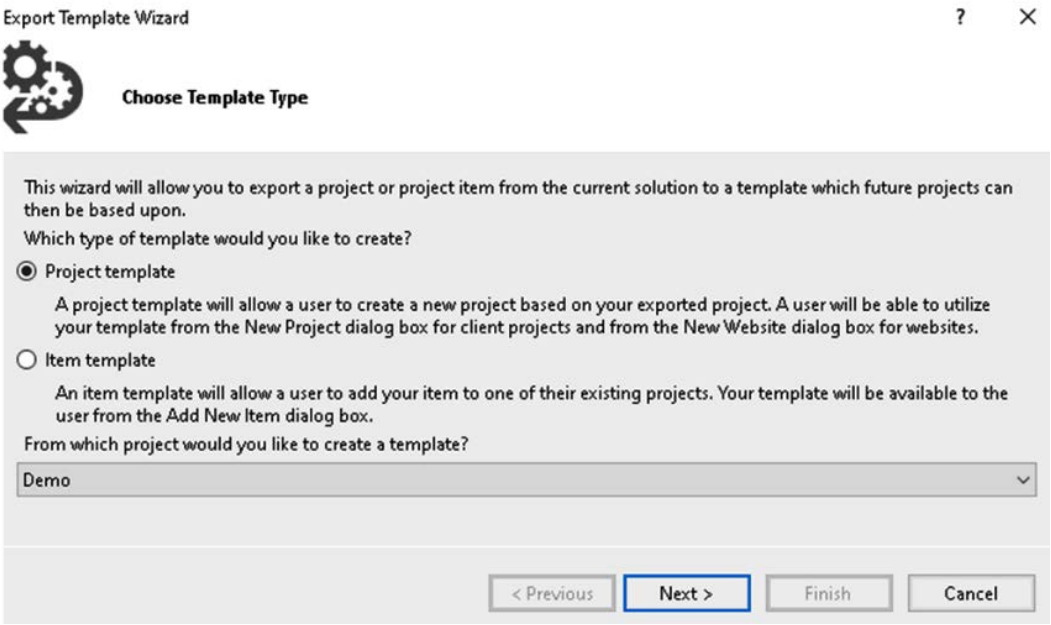



Figure 2.18 – The Choose Template Type dialog

4. Complete the **Select Template Options** dialog as shown in the following screenshot and click the **Finish** button:

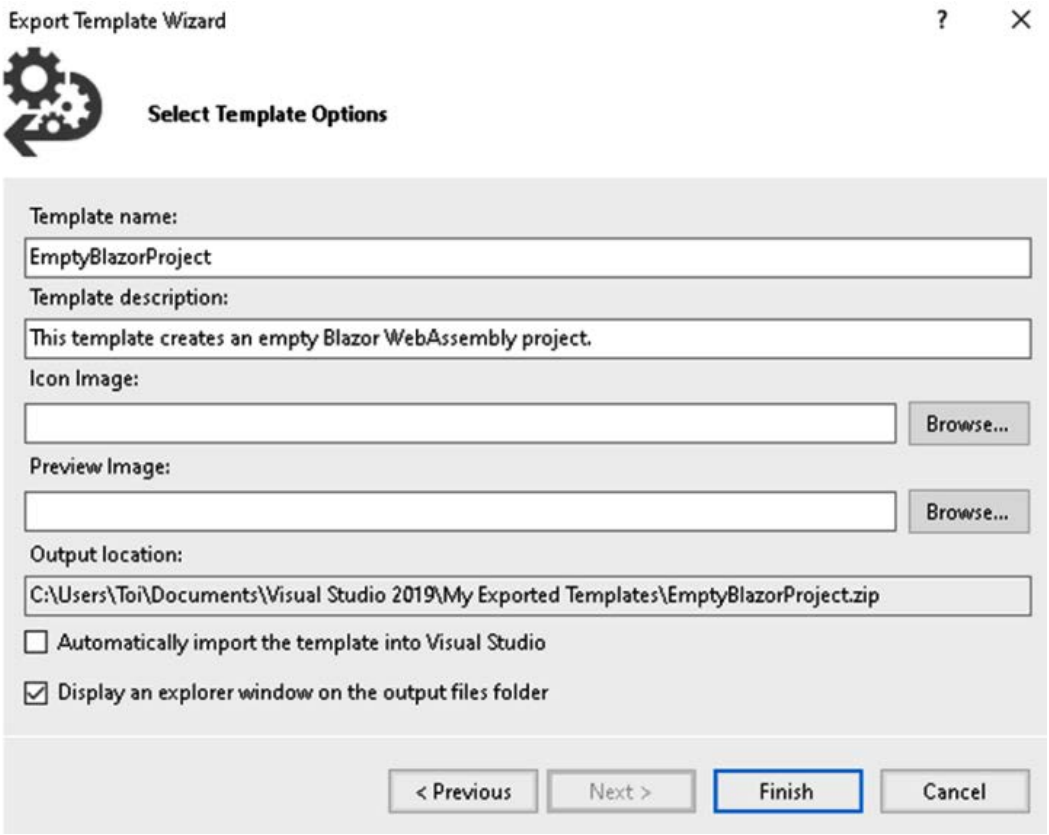



Figure 2.19 – The Select Template Options dialog

After you click the **Finish** button, your new project template will be saved to the folder indicated in the **Output location** field on the **Select Template Options** dialog and the folder will automatically open. The files that comprise your new project template are compressed into a file called `EmptyBlazorProject.zip`.

## Updating a custom project template

We need to make a few updates to our custom project template before it is ready to use. First, we will declare a template parameter for the project's name, and then we will update the metadata. We do this as follows:

1. Extract all of the files from the `EmptyBlazorProject.zip` file.

   The `EmptyBlazorProject.zip` file contains all of the files from the empty `Demo` project as well as a `MyTemplate.vstemplate` file that contains all of the metadata for the project template.

2. Open the `Shared/NavMenu.razor` file and replace the word `Demo` with `$projectname$`:

   ```
   <a class="navbar-brand" href="">$projectname$</a>
   ```

   The `$projectname$` parameter will be replaced by the name of the project that is provided by the user when the project is created.

   Open the `_Imports.razor` file and replace the word `Demo` with `$projectname$`:

   ```
   @using $projectname$
   @using $projectname$.Shared
   ```

3. Open the `MyTemplate.vstemplate` file.
4. Update the value of the `Name` element to `Empty Blazor WebAssembly App`:

   ```
   <Name>Empty Blazor WebAssembly App</Name>
   ```

5. Add the following elements after the `Description` element:

   ```
       <LanguageTag>C#</LanguageTag>
       <ProjectTypeTag>Web</ProjectTypeTag>
   ```

6. Replace the `Icon` element with the following `Icon Package` element:

   ```
   <Icon Package="{AAB75614-2F8F-4DA6-B0A6-763C6DBB2969}"
   ID="13"/>
   ```

7. Change the `ReplaceParameters` attribute to `true` for `NavMenu.razor` `ProjectItem`:

```
<ProjectItem ReplaceParameters="true"
             TargetFileName="NavMenu.razor">
    NavMenu.razor
</ProjectItem>
```

8. Change the `ReplaceParameters` attribute to `true` for `_Imports.razor` `ProjectItem`:

```
<ProjectItem ReplaceParameters="true"
             TargetFileName="_Imports.razor">
    _Imports.razor
</ProjectItem>
```

9. Save all of the updated files.
10. Update the `EmtpyBlazorProject.zip` file with the updated files.
11. Copy `EmtpyBlazorProject.zip` from the `Visual Studio 2019\MyExportedTemplates` folder to the `Visual Studio 2019\Templates\ProjectTemplates` folder.

## Using a custom project template

We can use a custom project template the same way that we use any of the built-in project templates. We do this as follows:

1. From the **File** menu, select the **New, Project** option.
2. Enter `Blazor` in the **Search for templates** textbox to locate your new template:

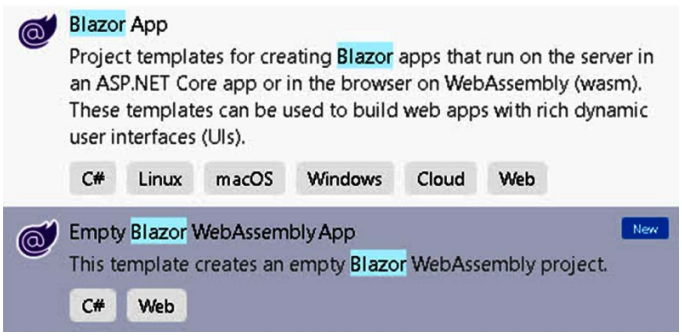



Figure 2.20 – Empty Blazor WebAssembly App template

3. Select the **Empty Blazor WebAssembly App** template and click the **Next** button.
4. Update the project name to `Sample` and click the **Create** button.
5. From the **Build** menu, select the **Build Solution** option.
6. From the **Debug** menu, select **Start Without Debugging** (*Ctrl+F5*).

We have created a new `Sample` project by using our custom project template. The only page in the `Sample` project is the `Home` page.

We created an empty project by deleting some of the components and code from the `Demo` project that we created in the previous section. Then, we used the **Export Template Wizard** to create a custom project template based on the empty project. After we updated some of the files in the custom project template, we copied them into the `ProjectTemplates` folder. Finally, we used the custom project template to create the `Sample` project.

# Summary

You should now be able to create a Blazor WebAssembly application.

In this chapter, we introduced Razor components, routing, and Razor syntax.

After that, we used the **Blazor WebAssembly App** project template provided by Microsoft to create the `Demo` Blazor WebAssembly project. We added a parameter to the `Counter` component and examined how routing works.

In the last part of the chapter, we created an empty Blazor WebAssembly project on which to base our own custom project template. We created a custom project template using the **Export Template Wizard**. After we finished configuring our custom project template, we used it to create an empty Blazor WebAssembly project.

We will use the **Empty Blazor WebAssembly App** project template to create the project in the next chapter of this book.

# Questions

The following questions are provided for your consideration:

1. Can Razor components include JavaScript?
2. What types of loops are supported by Razor syntax?
3. Can the Blazor App project template be used to create both Blazor WebAssembly applications and Blazor Server applications?
4. What are the advantages of using a custom project template?
5. How would you create your own custom item template to automatically create a code-behind page for each new component?

# Further reading

The following resources provide more information concerning the topics in this chapter:

- For more information on Bootstrap, refer to `https://getbootstrap.com`.
- For more information on Razor syntax, refer to `https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor`.
- For more information on creating custom project templates, refer to `https://docs.microsoft.com/en-us/visualstudio/ide/creating-project-and-item-templates`.
- For more information on template parameters, refer to `https://docs.microsoft.com/en-us/visualstudio/ide/template-parameters`.s

# 3
# Building a Modal Dialog Using Templated Components

A modal dialog is a dialog that appears on top of all other content in a window and requires user interaction to close it. Templated components are components that accept one or more UI templates as parameters. The UI templates can contain any Razor markup.

In this chapter, we will learn about `RenderFragment` parameters, `EventCallback` parameters, and CSS isolation. `RenderFragment` parameters are used when a parent component needs to share information with a child component, and conversely, `EventCallback` parameters are used when a child component needs to share information with its parent component. We will also learn how to apply styles to only a single component by using CSS isolation.

In this chapter, we will create a modal dialog component. The component will be a templated component that can render different HTML based on the contents of its parameters. It will use event callbacks to return events back to the calling component. It will use CSS isolation to add the formatting that will make it behave like a modal dialog. Finally, we will move the component to a **Razor class library** so that it can be shared with other projects.

In this chapter, we will cover the following topics:

- `RenderFragment` parameters
- `EventCallback` parameters
- CSS isolation
- Creating a Razor class library
- Creating the modal dialog project

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free Community Edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need the **Empty Blazor WebAssembly App project template** that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter03`.

The code in action video is available here: `https://bit.ly/33X2Zkc`.

# RenderFragment parameters

A `RenderFragment` parameter is a segment of UI content. A `RenderFragment` parameter is used to communicate UI content from the parent to the child. The UI content can include plain text, HTML markup, Razor markup, or another component.

The following code is for the `Alert` component. The UI content of the `Alert` component is displayed when the value of its `Show` property is `true`:

**Alert.razor**

```
@if (Show)
{
    <div class="dialog-container">
        <div class="dialog">
            <div>
                @ChildContent
            </div>
            <div>
                <button @onclick="OnOk">
                    OK
                </button>
            </div>
        </div>
    </div>
}
@code {
  [Parameter] public bool Show { get; set; }
  [Parameter] public EventCallback OnOk { get; set; }
  [Parameter] public RenderFragment ChildContent { get;
    set; }
}
```

The preceding code, for the `Alert` component, includes three different types of parameters: Boolean, `RenderFragment`, and `EventCallback`:

- The first parameter is the `Show` property. It is of type Boolean, which is a simple type. For more information on using simple types as parameters, see *Chapter 2, Building Your First Blazor WebAssembly Application*.
- The second parameter is the `OnOk` property. It is of type `EventCallback`. We will learn more about `EventCallback` parameters in the next section.
- The last parameter is the `ChildContent` property. It is of type `RenderFragment`.

The following markup uses the `Alert` component to display the current day of the week in a dialog when the **Show Alert** button is clicked. The Razor markup between the opening tag and the closing tag of the `Alert` element is bound to the `ChildContent` property of the `Alert` component:

**Index.razor**

```
@page "/"

<Alert Show="showAlert" OnOk="@(() => showAlert = false)">
    <h1>Alert</h1>
    <p>Today is @DateTime.Now.DayOfWeek.</p>
</Alert>

<button @onclick="@(() => showAlert = true)">
    Show Alert
</button>

@code {
    private bool showAlert = false;
}
```

The following screenshot shows the dialog that is displayed when the **Show Alert** button is clicked:


Figure 3.1 – Sample alert

The name of the `RenderFragment` parameter must be `ChildContent` if you want to use the content of the element without explicitly specifying the parameter's name. For example, the following markup results in the same output as the preceding markup that did not explicitly specify `ChildContent`:

```
<Alert Show="showAlert" OnOk="@(() => showAlert = false)">
    <ChildContent>
```

```
        <h1>Alert</h1>
        <p>Today is @DateTime.Now.DayOfWeek.</p>
    </ChildContent>
</Alert>

<button @onclick="@(() => showAlert = true)">
    Show Alert
</button>

@code {
    private bool showAlert = false;
}
```

The `ChildContent` element is highlighted in the preceding markup.

> **Important note**
>
> By convention, the name of the `RenderFragment` parameter used to capture the content of a parent element must be `ChildContent`.

It is possible to include multiple `RenderFragment` parameters in a component by explicitly specifying each parameter's name in the markup. We will use multiple `RenderFragment` parameters to complete the project in this chapter.

A `RenderFragment` parameter enables a parent component to communicate the UI content to be used by its child component, while an `EventCallback` parameter is used to communicate from the child component back to the parent component. In the next section, we will see how to use `EventCallback` parameters.

# EventCallback parameters

An event callback is a method that you pass to another method to be called when a particular event occurs. For example, when the button on the `Alert` component is clicked, the `@onclick` event uses the `OnOk` parameter to determine the method that should be called. The method that the `OnOK` parameter references is defined in the parent component.

`EventCallback` parameters are used to share information from the child component to the parent component. They share information with their parent and notify their parent when something, such as a button click, has occurred. The parent component simply specifies the method to call when the event is triggered.

This is an example of an `EventCallback` parameter:

```
[Parameter] public EventCallback OnOk { get; set; }
```

The following example uses a **lambda expression** for the `OnOk` method. When the `OnOk` method is called, the value of the `showAlert` property is set to `false`:

```
<Alert Show="showAlert" OnOk="@(() => showAlert = false)">
        <h1>Alert</h1>
        <p>Today is @DateTime.Now.DayOfWeek.</p>
</Alert>
@code {
    private bool showAlert = false;
}
```

A lambda expression allows you to create an anonymous function. You do not need to use an anonymous function. The following example shows how to use a method for the `OnOk` method instead of an anonymous function:

```
<Alert Show="showAlert" OnOk="OkClickHandler)">
        <h1>Alert</h1>
        <p>Today is @DateTime.Now.DayOfWeek.</p>
</Alert>
@code {
    private bool showAlert = false;

    private void OkClickHandler()
    {
        showAlert = false;
    }
}
```

When writing the `Alert` component, you might be tempted to update the `Show` parameter directly from the `OnOk` event on the component. You must not do so because if you update the values directly in the component and the component has to be re-rendered, any state changes will be lost. If you need to maintain state in the component, you should add a private field to the component.

> **Important note**
> Components should never write to their own parameters.

The `Alert` component displays text on the page, but it does not yet work like a modal dialog. In order to make it work like a modal dialog, we need to update the style sheets that are used by the component. We can do that by using CSS isolation. In the next section, we will see how to use CSS isolation.

# CSS isolation

The location of the **cascading style sheets** (**CSS**) used to style our Blazor WebAssembly apps is usually the `wwwroot` folder. Usually, the styles defined in those CSS files are applied to all of the components in the web app. However, there are times when you want more control over the styles that are applied to a particular component. To achieve that, we use CSS isolation. With CSS isolation, the styles in the designated CSS file will override the global styles.

## Enabling CSS isolation

In order to add a `CSS` file that is isolated to a certain component, create a `CSS` file in the same folder as the component with the same name as the component, but with a `CSS` file extension. For example, the `CSS` file for the `Alert.razor` component would be called `Alert.razor.css`.

The following markup is for an updated version of the `Alert` component. In this version, we have added the two highlighted classes:

**Alert.razor**

```
@if (Show)
{
    <div class="dialog-container">
        <div class="dialog">
            <div>
                @ChildContent
            </div>
            <div>
                <button @onclick="OnOk">
                    OK
                </button>
            </div>
        </div>
```

```
    </div>
}
@code {
  [Parameter] public bool Show { get; set; }
  [Parameter] public EventCallback OnOk { get; set; }
  [Parameter] public RenderFragment ChildContent { get;
    set; }
}
```

The following `CSS` file defines the styles used by the new classes:

**Alert.razor.css**

```
.dialog-container {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    background-color: rgba(0,0,0,0.6);
    z-index: 2000;
}

.dialog {
    background-color: white;
    margin: auto;
    width: 15rem;
    padding: .5rem
}
```

The preceding `CSS` includes styles for both the `dialog-container` class and the `dialog` class:

- `dialog-container`: This class sets the background color of the element to black with 60% opacity and places it on top of the other elements by setting its z-index to 2,000.
- `dialog`: This class sets the background color of the element white, centers it horizontally within its parent, and sets its width to 15 REM.

The following screenshot shows the `Alert` component using the preceding `CSS` file:


Figure 3.2 – Alert component

## Supporting child components

By default, when using CSS isolation, the CSS styles only apply to the current component. If you want the CSS styles to apply to a child component of the current component, you need to use the `::deep` combinator within your style. This combinator selects the elements that are descendants of the element's identifier.

For example, the following style will be applied to any `H1` headings with the current component, as well as any `H1` headings within the child components of the current component:

```
::deep h1 {
    color: red;
}
```

CSS isolation is useful if you don't want your component to use the global styles or you want to share your component via a Razor class library and you need to avoid styling conflicts.

# Project overview

In this chapter, we will build a modal dialog component. You will be able to customize both the `Title` and the `Body` of the modal dialog using Razor markup.

This is a screenshot of the modal dialog:

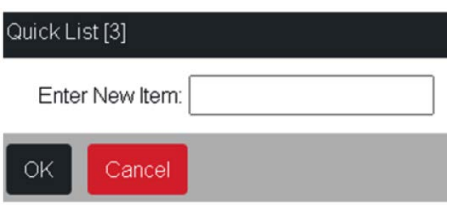



Figure 3.3 – Modal dialog

Finally, after we have completed the modal dialog component, we will move it to a Razor class library so that it can be shared with other projects.

The build time for this project is approximately 90 minutes.

# Creating the modal dialog project

The `ModalDialog` project will be created by using the **Empty Blazor WebAssembly App project template**. We will add a `Dialog` component that includes multiple sections, and use CSS isolation to apply styles that make it behave like a modal dialog. We will use `EventCallback` parameters to communicate from the component back to the parent when a button is clicked. We will use `RenderFragment` parameters to allow Razor markup to be communicated from the parent to the component. Finally, we will create a Razor class library and move our `Dialog` component to it so that it can be shared with other projects.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt* + *S*) textbox, enter `Blazor` and hit the *Enter* key.

   The following screenshot shows the **Empty Blazor WebAssembly App project template** that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*:

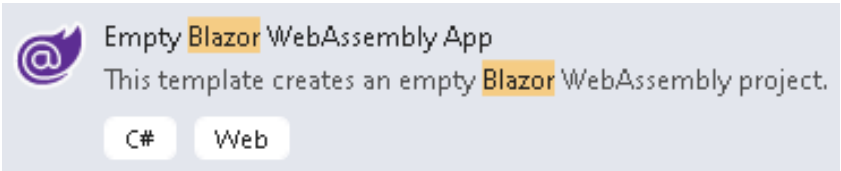



Figure 3.4 – Empty Blazor WebAssembly App project template

4. Select the **Empty Blazor WebAssembly App project template** and click the **Next** button.

5. Enter `ModalDialog` in the **Project name** textbox and click the **Create** button:

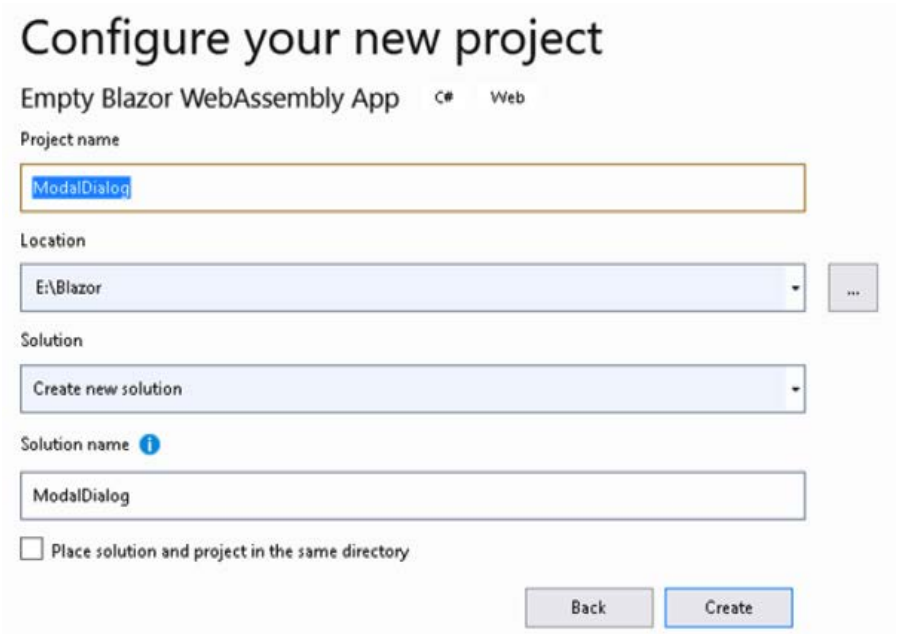



Figure 3.5 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `ModalDialog` project into the `E:/Blazor` folder. However, the location of this project is not important.

We have created the `ModalDialog` Blazor WebAssembly project.

## Adding the Dialog component

The `Dialog` component will be shared. Therefore, we will add it to the `Shared` folder. We do this as follows:

1. Right-click the `Shared` folder and select **Add, Razor Component** from the menu.
2. Name the new component `Dialog`.
3. Click the **Add** button.

4. Replace the markup with the following markup:

```
@if (Show)
{
    <div class="dialog-container">
        <div class="dialog">
            <div class="dialog-title">Title</div>
            <div class="dialog-body">Body</div>
            <div class="dialog-buttons">
                <button class="btn btn-dark mr-2">
                    OK
                </button>
                <button class="btn btn-danger">
                    Cancel
                </button>
            </div>
        </div>
    </div>
}

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

The `Show` property is used to show and hide the contents of the component. We have added a `Dialog` component, but it will not behave like a modal dialog box until the appropriate styles have been added to the project.

## Adding a CSS

The preceding markup includes five classes that we will use to style the `Dialog` component to make it behave like a modal dialog:

- `dialog-container`: This class is used to set the background color of the element to black with 60% opacity and place it on top of the other elements by setting its z-index to 2,000.
- `dialog`: This class is used to set the background color of the element to white, center it horizontally within its parent, and set its width to 25 REM.

- `dialog-title`: This class is used to set the background color to dark gray, set the text to white, and add some padding.
- `dialog-body`: This class is used to add some padding.
- `dialog-buttons`: This class is used to set the background color to silver and add some padding.

> **Tip**
>
> The rest of the classes used by the `Dialog` component, such as `btn`, are defined in the `wwwroot\css\bootstrap\bootstrap.min.css` file, which contains the styles provided by the Bootstrap framework.

We need to create a `CSS` file to define how to style each of these classes. We do this as follows:

1. Right-click the `Shared` folder and select **Add, New Item** from the menu.
2. Enter `css` in the **Search** box.
3. Select **Style Sheet**.
4. Name the style sheet `Dialog.razor.css`.
5. Click the **Add** button.
6. Enter the following styles:

**Dialog.razor.css**

```
.dialog-container {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    background-color: rgba(0,0,0,0.6);
    z-index: 2000;
}

.dialog {
    background-color: white;
    margin: auto;
```

```
    width: 25rem;
}

.dialog-title {
    background-color: #343a40;
    color: white;
    padding: .5rem;
}

.dialog-body {
    padding: 2rem;
}

.dialog-buttons {
    background-color: silver;
    padding: .5rem;
}
```

The styles in the `Dialog.razor.cs` file will only be used by the `Dialog` component due to CSS isolation.

## Testing the Dialog component

In order to test the `Dialog` component, we need to add it to another component. We will add it to the `Index` component that is used as the **Home** page of the application. We do this as follows:

1. Open the `Pages\Index.razor` file.
2. Add the following markup to the `Index.razor` file:

```
<Dialog Show="showDialog"></Dialog>

<button @onclick="OpenDialog">Show Dialog</button>

@code {
    private bool showDialog = false;

    private void OpenDialog()
```

```
        {
            showDialog = true;
        }
    }
```

Make sure you do not remove the `@page` directive at the top of the file.

3. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.
4. Click the **Show Dialog** button:

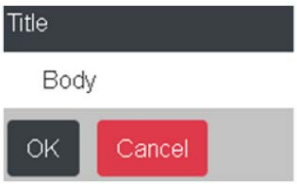



Figure 3.6 – Simple modal dialog

5. Click the **OK** button.

Nothing happens when you click the **OK** button because we have not yet added an `@onclick` event.

## Adding EventCallback parameters

We need to add `@onclick` events for both the **OK** button and the **Cancel** button. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `Shared\Dialog.razor` file.
3. Add `@onclick` events to each of the buttons:

```
<button class="btn btn-dark mr-2" @onclick="OnOk">
    OK
</button>
<button class="btn btn-danger" @onclick="OnCancel">
    Cancel
</button>
```

4. Add the following parameters to the code block:

```
[Parameter]
public EventCallback<MouseEventArgs> OnOk { get; set; }
[Parameter]
public EventCallback<MouseEventArgs> OnCancel { get; set;
}
```

> **Tip**
> The `Parameter` decorator does not need to be on the same line as the property that it applies to.

5. Open the `Pages\Index.razor` file.
6. Update the markup for the `Dialog` element by adding the highlighted markup:

```
<Dialog Show="showDialog"
        OnCancel="DialogCancelHandler"
        OnOk="DialogOkHandler">
</Dialog>
```

7. Add the following methods to the code block:

```
private void DialogCancelHandler(MouseEventArgs e)
{
    showDialog = false;
}

private void DialogOkHandler(MouseEventArgs e)
{
    showDialog = false;
}
```

8. From the **Build** menu, select the **Build Solution** option.
9. Return to the browser.
10. Use *Ctrl* + *R* to refresh the browser.
11. Click the **Show Dialog** button.
12. Click the **OK** button.

The dialog box will close when you click the **OK** button. Now let's update the `Dialog` component to allow us to customize the `Title` and `Body` properties of the modal dialog that it creates.

## Adding RenderFragment parameters

We will use `RenderFragment` parameters for both the `Title` and `Body` properties of the `Dialog` component. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `Shared\Dialog.razor` file.
3. Update the markup for `dialog-title` to the following:

```
<div class="dialog-title">@Title</div>
```

4. Update the markup for `dialog-body` to the following:

```
<div class="dialog-body">@Body</div>
```

5. Add the following parameters to the code block:

```
[Parameter]
public RenderFragment Title { get; set; }
[Parameter]
public RenderFragment Body { get; set; }
```

6. Open the `Pages\Index.razor` file.
7. Update the markup for the `Dialog` element to the following:

```
<Dialog Show="showDialog"
        OnCancel="DialogCancelHandler"
        OnOk="DialogOkHandler">
    <Title>Quick List [@(Items.Count + 1)]</Title>
    <Body>
        Enter New Item: <input @bind="NewItem" />
    </Body>
</Dialog>
```

The preceding markup will change the title of the dialog to `Quick List` and provide a textbox for the user to enter items for a list.

8. Add the following markup under the `Dialog` element:

```
<ol>
    @foreach (var item in Items)
    {
        <li>@item</li>
    }
</ol>
```

The preceding code will display the items in the `Items` list in an ordered list.

9. Add the following variables to the top of the code block:

```
private string NewItem;
private List<string> Items = new List<string>();
```

10. Update `DialogCancelHandler` to the following:

```
private void DialogCancelHandler(MouseEventArgs e)
{
    NewItem = "";
    showDialog = false;
}
```

The preceding code will clear the textbox and hide `Dialog`.

11. Update `DialogOkHandler` to the following:

```
private void DialogOkHandler(MouseEventArgs e)
{
    if (!string.IsNullOrEmpty(NewItem))
    {
        Items.Add(NewItem);
        NewItem = "";
    };
    showDialog = false;
}
```

The preceding code will add `NewItem` to the `Items` list, clear the textbox, and hide `Dialog`.

12. From the **Build** menu, select the **Build Solution** option.
13. Return to the browser.

14. Use *Ctrl* + *R* to refresh the browser.
15. Click the **Show Dialog** button.
16. Enter some text in the **Enter New Item** field.
17. Click the **OK** button.
18. Repeat.

    Each time that you click the **OK** button, the text in the **Enter New Item** field will be added to the list. The following screenshot shows a list where three items have already been added and a fourth item is about to be added using the modal dialog:

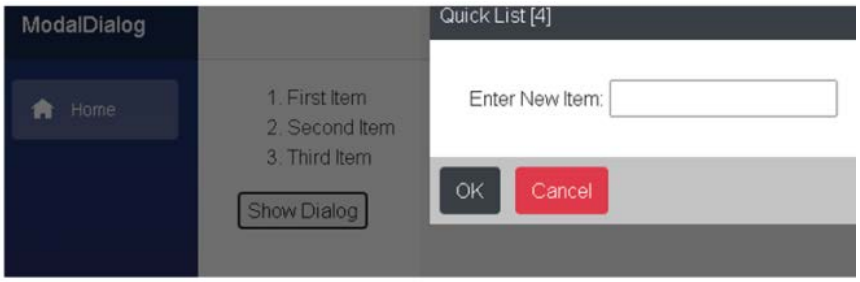



Figure 3.7 – Sample Quick List

19. Close the browser.

To share this new component with other projects, we need to add it to a **Razor class library**.

## Creating a Razor class library

We can share components across projects by using a Razor class library. To create a Razor class library, we will use the **Razor Class Library** project template. We do this as follows:

1. Right-click the solution and select the **Add, New Project** option from the menu.
2. Enter `Razor Class Library` in the **Search for templates** textbox to locate the **Razor Class Library** project template:

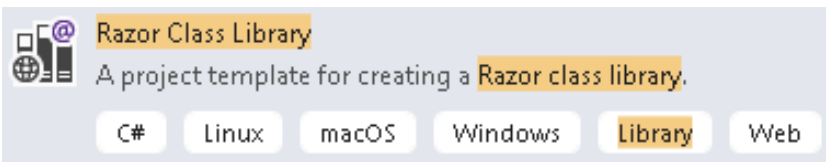



Figure 3.8 – Razor Class Library project template

3. Select the **Razor Class Library** project template.

4. Click the **Next** button.
5. Name the project `MyComponents` and click the **Create** button.
6. Accept the defaults and click the **Create** button.
7. Right-click the `ModalDialog` project and select the **Add, Project Reference** option from the menu.
8. Check the `MyComponents` checkbox and click the **OK** button.

We have created a sample Razor class library.

## Testing the Razor class library

The sample Razor class library that we just created includes one component called `Component1`. Before we continue, we should test whether the new Razor class library is working properly. We do this as follows:

1. Open the `ModalDialog\Pages\ Index.razor` file.
2. Add the following `@using` statement right below the `@page` directive:

```
@using MyComponents;
```

> **Tip**
> If you will be using this project on multiple pages, you should add the `@using` statement directly to the `ModalDialog\_Imports.razor` file.

3. Add the following markup below the `@using` statement:

```
<Component1 />
```

4. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.

   The following screenshot shows how the `Component1` component should render:


Figure 3.9 – Component1

> **Important note**
>
> If the `Component1` component is missing its styling, it is because the CSS file is cached. Use the following key combination, *Ctrl* + *Shift* + *R*, to empty the cache and reload the page.

5. Return to **Visual Studio**.
6. Delete the `Component1` element.

We have finished testing the sample Razor class library.

## Adding a component to the Razor class library

In order to share the `Dialog` component, we need to move it to the Razor class library that we just created and tested. We do this as follows:

1. Right-click the `ModalDialog\Shared\Dialog.razor` file and select the **Copy** option from the menu.
2. Right-click the `MyComponents` project and select the **Paste** option from the menu.
3. Right-click the `MyComponents\Dialog.razor` file and select the **Rename** option from the menu.
4. Rename the file to `BweDialog.razor`.

   In this case, `Bwe` stands for **Blazor WebAssembly by Example**.

> **Tip**
>
> When naming components in a Razor class library, you should give them unique names to avoid ambiguous reference errors. Most organizations prefix all of their shared components with the same text. For example, a company named **One Stop Designs** (**OSD**) might prefix all of their shared components with `Osd`.

5. Open the `ModalDialog\Pages\Index.razor` file.
6. Rename the `Dialog` element to `BweDialog`.
7. From the **Build** menu, select the **Build Solution** option.
8. Return to the browser.
9. Use *Ctrl* + *R* to refresh the browser.
10. Click the **Show Dialog** button.

11. Enter some text in the **Enter New Item** field.
12. Click the **OK** button.
13. Repeat.

The `BweDialog` component is now being used from the `MyComponents` project. Since it is now included in the Razor class library, you can easily share it with other projects.

# Summary

You should now be able to create a modal dialog and share it with multiple projects by using a Razor class library.

In this chapter, we introduced `RenderFragment` parameters, `EventCallback` parameters, and CSS isolation.

After that, we used the **Empty Blazor App** project template to create a new project. We added a `Dialog` component that acts like a modal dialog. The `Dialog` component uses both `RenderFragment` parameters and `EventCallback` parameters to share information between it and its parent. Also, it used CSS isolation for its styles.

In the last part of the chapter, we created a Razor custom library and moved the `Dialog` component to the new library.

So far, in this book, we have avoided using JavaScript. Unfortunately, there are still some functions that we can only accomplish with JavaScript. We will see how to use JavaScript interop to use JavaScript in Blazor WebAssembly in the next chapter of this book.

# Questions

The following questions are provided for your consideration:

1. How can you replace a table with a templated component?
2. How would you add default values for the `Title` property and the `Body` property of the `Dialog` component?
3. If you want all of the components in a Razor class library to use the same CSS file, can you move the styles to a shared CSS file?
4. Can you distribute your `Dialog` component using a NuGet package?

# Further reading

The following resources provide more information concerning the topics in this chapter:

- For more information on CSS, refer to `https://www.w3schools.com/css/default.asp`.
- For more information on lambda expressions, refer to `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions`.
- For more information on ASP.NET Core Razor components class libraries, refer to `https://docs.microsoft.com/en-us/aspnet/core/blazor/components/class-libraries`.
- For more information on NuGet, refer to `https://www.nuget.org`.

# 4

# Building a Local Storage Service Using JavaScript Interoperability (JS Interop)

The Blazor WebAssembly framework makes it possible for us to run C# code on the browser. However, there are some scenarios that it cannot handle, and in those cases, we need to use JavaScript functions to fill in the gaps.

In this chapter, we will learn how to use JavaScript with Blazor WebAssembly. We will learn how to invoke a JavaScript function from Blazor with and without a return value. Conversely, we will learn how to invoke .NET methods from JavaScript. We will accomplish both of these scenarios by using **JavaScript interop** (**JS interop**). Finally, we will learn how to store data on the browser by using **localStorage**.

The project that we will create in this chapter will be a local storage service that will read and write to the browser's localStorage. In order to access the browser's localStorage, we will need to use JavaScript. JS interop is used to invoke JavaScript from .NET.

In this chapter, we will cover the following topics:

- Why use JavaScript?
- Exploring JS interop
- Understanding local storage
- Invoking a JavaScript function from Blazor
- Invoking a .NET method from JavaScript
- Creating the local storage service

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free Community Edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter04`.

The code in action video is available here: `https://bit.ly/3tXVMeg`.

# Why use JavaScript?

With Blazor WebAssembly, you can create complete applications without directly using JavaScript. However, you may need to use JavaScript because there are some scenarios that you cannot accomplish without it. Without JavaScript, you can't manipulate the DOM or call any of the JavaScript APIs that we rely on for web development.

This is a sample of the things that you do not have access to directly from the Blazor WebAssembly framework:

- **DOM manipulation**
- The **Media Capture and Streams API**
- The **WebGL API** (2D and 3D graphics for the web)

- The **Web Storage API** (localStorage and sessionStorage)
- The **Geolocation API**
- JavaScript pop-up boxes (*alert*, *confirm*, *prompt*)
- The online status of the browser
- The browser's history
- **Chart.js**
- Other third-party JavaScript libraries

The preceding list is not at all comprehensive since there are hundreds of JavaScript libraries that are currently available. However, the key point to remember is that you cannot manipulate the DOM without using JavaScript. Therefore, we will probably always need to use some JavaScript in our web apps. Luckily, by using JS interop, this is easy to do.

# Exploring JS interop

To invoke a JavaScript function from .NET, we use the `IJSRuntime` abstraction. This abstraction represents an instance of a JavaScript runtime that the framework can call into. To use `IJSRuntime`, we must first inject it into our component using dependency injection. For more information on dependency injection, refer to *Chapter 6, Building a Shopping Cart Using Application State*.

The `@inject` directive is used to inject a dependency into a component. The following code injects `IJSRuntime` into the current component:

```
@inject IJSRuntime js
```

The `IJSRuntime` abstraction has two methods that we can use to invoke JavaScript functions:

- `InvokeVoidAsync`
- `InvokeAsync`

Both of these methods are asynchronous. The difference between these two methods is that one of them returns a value and the other does not. We can downcast an instance of `IJSRuntime` to an instance of `IJSInProcessRuntime` to run the method synchronously. Finally, we can invoke a .NET method from JavaScript by decorating the method with `JsInvokable`.

## InvokeVoidAsync

The `InvokeVoidAsync` method is used to invoke a JavaScript function that does not return a value. It invokes the specified JavaScript function asynchronously.

This is the `InvokeVoidAsync` method of `IJsRuntime`:

```
InvokeVoidAsync(string identifier, params object[] args);
```

The first argument is the identifier for the JavaScript method that is being called, and the second argument is an array of JSON-serializable arguments. The second argument is optional.

In JavaScript, the `Document` object represents the root node of the HTML document. The `title` property of the `Document` object is used to specify the text that appears in the browser's title bar. Assume that we want to update the browser's title as we navigate between the components in our Blazor WebAssembly app. To do that, we need to use JavaScript to update the `title` property.

The following JavaScript code defines a method called `setDocumentTitle`, which sets the `title` property of the `Document` object to the value provided by the `title` argument:

**bweInterop.js**

```
var bweInterop = {};

bweInterop.setDocumentTitle = function (title) {
    document.title = title;
}
```

> **Tip**
>
> In this book, we will be using the `bweInterop` namespace for our JavaScript code to both structure our code and minimize the risk of naming conflicts.

Before we can access the preceding JavaScript code, we need to add a reference to it from the `wwwroot/index.html` file. The following highlighted code is a reference to the JavaScript file. It assumes that it has been placed into a folder called `scripts`:

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

The new script tag should be added before the script tag that references the `_framework/blazor.webassembly.js` file in the `body` element of the `wwwroot/index.html` file.

The following `Document` component uses the `setDocumentTitle` JavaScript function to update the browser's title bar:

**Document.razor**

```
@inject IJSRuntime js

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

    protected override async Task OnAfterRenderAsync(bool
      firstRender)
    {
        if (firstRender)
        {
            await js.InvokeVoidAsync(
                "bweInterop.setDocumentTitle",
                Title);
        }
    }
}
```

In the preceding code, `IJSRuntime` is injected into the component. Then, the `OnAfterRenderAsync` method uses the `InvokeVoidAsync` method of `IJSRuntime` to invoke the `setDocumentTitle` JavaScript function the first time that the component is rendered.

The following markup uses the `Document` component to update the browser's title bar to `Home - My App`:

```
<Document Title="Home - My App" />
```

The following screenshot shows the updated document title:

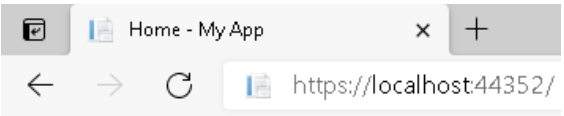



Figure 4.1 – Updated document title

The `InvokeVoidSync` method is used to call JavaScript functions that do not return a value. If we need to return a value, we need to use the `InvokeAsync` method instead.

## InvokeAsync

The `InvokeAsync` method is used when we want to call a JavaScript function that returns a value. It invokes the specified JavaScript function asynchronously.

This is the `InvokeAsync` method of `IJSRuntime`:

```
ValueTask<TValue> InvokeAsync<TValue>(string identifier,
                                      params object[] args);
```

Just like the `InvokeVoidAsync` method, the first argument is the identifier for the JavaScript method, and the second argument is an array of JSON-serializable arguments. The second argument is optional. The `InvokeAsync` method returns a `ValueTask` of the `TValue` type. `TValue` is a JSON-deserialized instance of the JavaScript's return value.

In JavaScript, the `Window` object represents the browser's window. If we need to determine the width and height of the current window, we can use the `innerWidth` and `innerHeight` properties of the `Window` object.

The following JavaScript code defines a method called `getWindowSize` that returns the width and height of the `Window` object:

**bweInterop.js**

```
var bweInterop = {};

bweInterop.getWindowSize = function () {
    var size = {
        width: window.innerWidth,
        height: window.innerHeight
    }

```

```
    return size;
}
```

This is the definition of the `WindowSize` class that is used to store the size of the window in .NET:

```
public class WindowSize
{
    public int? Width { get; set; }
    public int? Height { get; set; }
}
```

The following `Index` component invokes the `GetWindowSize` method from the `bweInterop.js` file:

**Index.razor**

```
@page "/"

@inject IJSRuntime js

@if (windowSize.Width != null)
{
    <h2>
        Window Size: @windowSize.Width x @windowSize.Height
    </h2>
}
<button @onclick="GetWindowSize">Get Window Size</button>

@code{
    private WindowSize windowSize = new WindowSize();

    private async Task GetWindowSize()
    {
        windowSize = await js.InvokeAsync<WindowSize>(
            "bweInterop.getWindowSize");
    }
}
```

In the preceding code, `IJSRuntime` is injected into the component. When the **Get Window Size** button is clicked the `GetWindowSize` method uses the `InvokeAsync` method of `IJSRuntime` to invoke the `getWindowSize` JavaScript function. The `GetWindowSize` JavaScript function returns the width and height of the window to the `windowSize` property. Finally, the component regenerates its render tree and applies any changes to the browser's DOM.

This is a screenshot of the page after the **Get Window Size** button has been clicked:


Figure 4.2 – Window size example

## Invoking JavaScript from .NET synchronously

So far in this chapter, we have only looked at invoking JavaScript functions asynchronously. But we can also invoke JavaScript functions synchronously. We do that by downcasting `IJSRuntime` to `IJSInProcessRuntime`. `IJSInProcessRuntime` represents an instance of a JavaScript runtime to which calls may be dispatched.

`IJSInProcessRuntime` allows our .NET code to invoke JS interop calls synchronously. This can be advantageous because these calls have less overhead than their asynchronous counterparts. The methods are similar to the asynchronous methods:

- `InvokeVoid`
- `Invoke`

The following code is the synchronous version of the `Document` component from earlier in this chapter. It uses `IJSInProcessRuntime` to invoke the JavaScript function synchronously:

**DocumentSync.razor**

```
@inject IJSRuntime js

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

    protected override void OnAfterRender(bool firstRender)
```

```
    {
        if (firstRender)
        {
            ((IJSInProcessRuntime)js).InvokeVoid(
                "bweInterop.setDocumentTitle",
                Title);
        }
    }
}
```

In the preceding code, the `IJsRuntime` instance has been downcast to an `IJSInProcessRuntime` instance. The `InvokeVoid` method of the `IJSInProcessRuntime` instance is used to invoke the `setDocumentTitle` JavaScript method.

The following markup uses the `DocumentSync` component:

```
<DocumentSync Title="Home - My App" />
```

## Invoking .NET from JavaScript

We can invoke a public .NET method from JavaScript by decorating the method with the `JSInvokable` attribute.

The following method is decorated with the `JSInvokable` attribute to enable it to be invoked from JavaScript:

```
[JSInvokable]
public void GetWindowSize(WindowSize newWindowSize)
{
    windowSize = newWindowSize;
    StateHasChanged();
}
```

In the preceding code, the `windowSize` property is updated each time the `GetWindowSize` method is invoked from JavaScript. The component's `StateHasChanged` method is called to notify the component that its state has changed and that the component should be re-rendered.

> **Tip**
>
> The `StateHasChanged` method of a component is only called automatically for `EventCallback` methods. In other cases, it must be called manually to notify the UI that it may need to be re-rendered.

To invoke a .NET method from JavaScript, you must create a `DotNetObjectReferenece` class for JavaScript to use in order to locate the .NET method. The `DotNetObjectReferenece` class wraps a JS interop argument, indicating that the value should not be serialized as JSON, but instead should be passed as a reference.

> **Important note**
>
> To avoid memory leaks and allow garbage collection on a component that creates a `DotNetObjectReference` class, we must diligently dispose of each instance of `DotNetObjectReference`.

The following code creates a `DotNetObjectReference` instance that wraps the `Resize` component. The reference is then passed to the JavaScript method:

```
private DotNetObjectReference<Resize> objRef;

protected async override Task OnAfterRenderAsync(bool
firstRender)
{
    if (firstRender)
    {
        objRef = DotNetObjectReference.Create(this);
        await js.InvokeVoidAsync(
            "bweInterop.registerResizeHandler",
             objRef);
    }
}
```

You can invoke a method in a .NET component from JavaScript using a reference to the component created with `DotNetObjectReference`. In the following JavaScript, the `registerResizeHandler` function creates `resizeHandler` that is called at initialization, and every time the window is resized.

You can use either the `invokeMethod` or `invokeMethodAsync` function to invoke .NET instance methods from JavaScript. The following example uses the `invokeMethodAsync` function to invoke the `GetWindowSize` method that is decorated with the `JSInvokable` attribute:

**bweInterop.js**

```
bweInterop.registerResizeHandler = function (dotNetObjectRef) {
    function resizeHandler() {
        dotNetObjectRef.invokeMethodAsync('GetWindowSize',
            {
                width: window.innerWidth,
                height: window.innerHeight
            });
    };

    resizeHandler();
    window.addEventListener("resize", resizeHandler);
}
```

This is the complete .NET code for the `Resize` component:

**Resize.razor**

```
@page "/resize"

@inject IJSRuntime js
@implements IDisposable

@if (windowSize.Width != null)
{
    <h2>
        Window Size: @windowSize.Width x @windowSize.Height
    </h2>
}

@code {
```

```
    private DotNetObjectReference<Resize> objRef;
    private WindowSize windowSize = new WindowSize();

    protected async override Task OnAfterRenderAsync(bool
      firstRender)
    {
        if (firstRender)
        {
            objRef = DotNetObjectReference.Create(this);
            await js.InvokeVoidAsync(
                "bweInterop.registerResizeHandler",
                    objRef);
        }
    }

    [JSInvokable]
    public void GetWindowSize(WindowSize newWindowSize)
    {
        windowSize = newWindowSize;
        StateHasChanged();
    }

    public void Dispose()
    {
        objRef?.Dispose();
    }
}
```

The preceding code for the `Resize` component displays the current width and height of the browser. As you resize the browser, the displayed values are automatically updated. Also, the `DotNetObjectReference` object is disposed of when the component is disposed of.

The `IJSRuntime` abstraction provides us with a way to invoke JavaScript functions from .NET and to invoke .NET methods from JavaScript. We will be using the JavaScript's **Web Storage API** to complete the project in this chapter. But before we can use it, we need to understand how it works.

# Understanding local storage

The Web Storage API for JavaScript provides mechanisms for browsers to store key/value pairs. For each web browser, the size of data that can be stored in web storage is at least 5 MB per origin. The localStorage is defined in the Web Storage API for JavaScript. We need to use JS interop to access localStorage on the browser.

The browser's localStorage is scoped to a particular URL. If the user reloads the page or closes and re-opens the browser, the contents of localStorage are retained. If the user opens multiple tabs, each tab shares the same localStorage. The data in localStorage is retained until it is explicitly cleared since it does not have an expiration date.

> **Tip**
> Data in a localStorage object that is created when using an InPrivate window or Incognito window is cleared when the last tab is closed.

These are the methods of localStorage:

- `key`: This method returns the name of the indicated key based on its position in localStorage.
- `getItem`: This method returns the value for the indicated key from localStorage.
- `setItem`: This method takes a key and value pair and adds them to localStorage.
- `removeItem`: This method removes the indicated key from localStorage.
- `clear`: This method clears localStorage.

> **Tip**
> sessionStorage is also defined in the Web Storage API. Unlike localStorage, which shares its value between multiple browser tabs, sesssionStorage is scoped to an individual browser tab. Therefore, if the user reloads the page, the data persists, but if the user closes the tab (or the browser), the data is cleared.

To view the content of the browser's localStorage, open **Developer tools (F12)** and select the **Application** tab. Look for **Local Storage** in the **Storage** section of the menu on the left. The following screenshot shows the **Application** tab of the **DevTools** dialog:

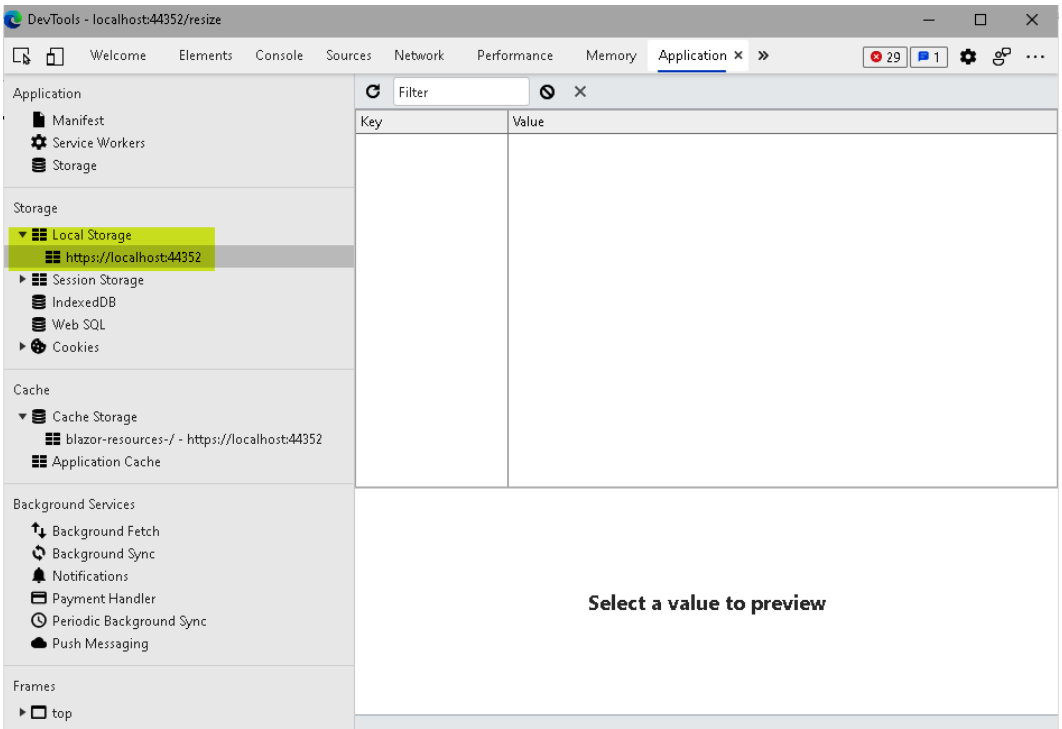



Figure 4.3 – Local Storage

By using the Web Storage API, it is easy to store data in the browser and to retreive it. Now, let's get a quick overview of the project that we are going to build in this chapter.

# Project overview

In this chapter, we will build a local storage service. The service will both write to and read from the browser's localStorage. We will use JS interop to accomplish this. Finally, we will create a component to test our service:

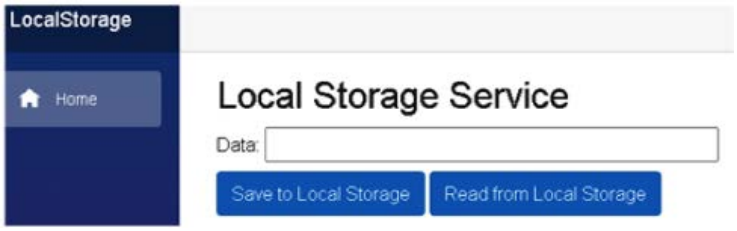



Figure 4.4 – Local Storage Service test page

The build time for this project is approximately 60 minutes.

# Creating the local storage service

The `LocalStorage` project will be created by using the **Empty Blazor WebAssembly App** project template. First, we will add a JavaScript file with the JavaScript functions that our service will need to use to update the browser's localStorage. Next, we will create the interface and class with the .NET methods that will invoke the JavaScript functions. Finally, we will test our service.

## Creating the local storage service project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt*+*S*) textbox, enter `Blazor` and hit the *Enter* key.

   The following screenshot shows the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*:

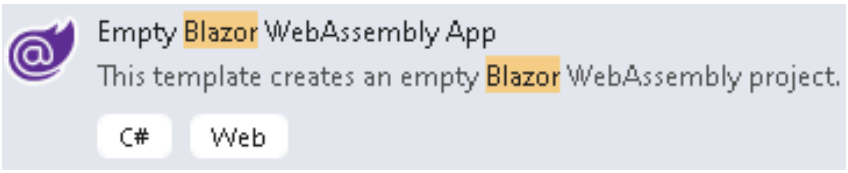



Figure 4.5 – Empty Blazor WebAssembly App project template

4. Select the **Empty Blazor WebAssembly App** project template and click the **Next** button.

5. Enter `LocalStorage` in the **Project name** textbox and then click the **Create** button:

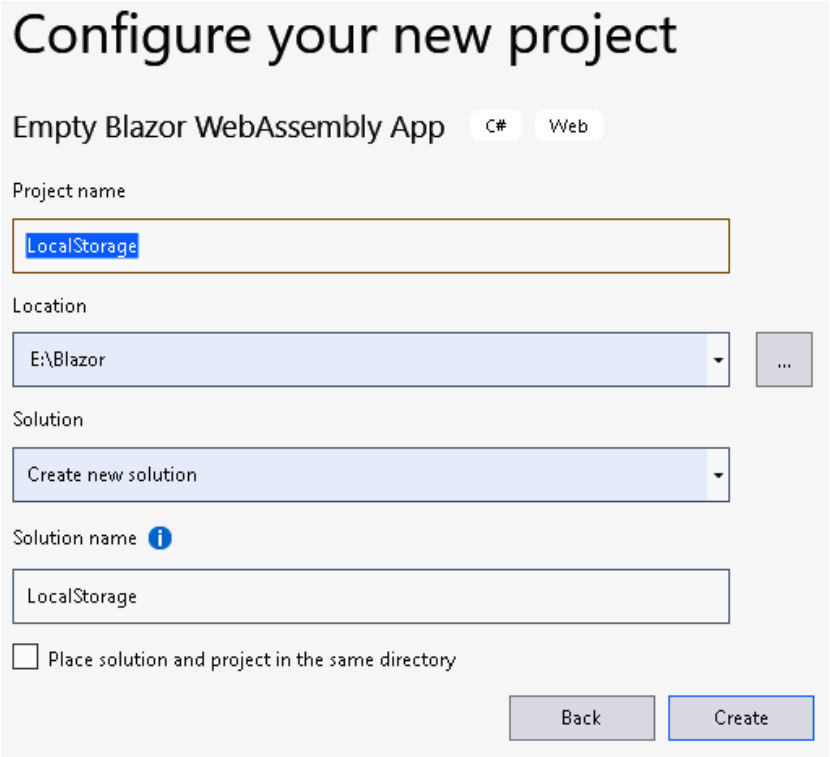



Figure 4.6 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `LocalStorage` project into the `E:/Blazor` folder. However, the location of this project is not important.

We have now created the `LocalStorage` Blazor WebAssembly project.

## Writing JavaScript to access localStorage

We need to write the JavaScript functions that will read to and write from the browser's localStorage. We do this as follows:

1. Right-click the `wwwroot` folder and select the **Add, New Folder** option from the menu.
2. Name the new folder `scripts`.
3. Right-click the `scripts` folder and select the **Add, New Item** option from the menu.
4. Enter `javascript` in the **Search** box.
5. Select **JavaScript File**.
6. Name the file `bweInterop.js`.
7. Click the **Add** button.
8. Enter the following JavaScript:

```
var bweInterop = {};

bweInterop.setLocalStorage = function (key, data) {
    localStorage.setItem(key, data);
}

bweInterop.getLocalStorage = function (key) {
    return localStorage.getItem(key);
}
```

9. Open the `wwwroot\index.html` file.
10. Add the following reference within the `body` element:

```
<script src="scripts/bweInterop.js"></script>
```

11. Make sure you add it before the reference to `_framework/blazor.webassembly.js`.

## Adding the ILocalStorageService interface

We need to create an interface for our service. We do this as follows:

1. Right-click the `LocalStorage` project and select the **Add, New Folder** option from the menu.
2. Name the new folder `Services`.
3. Right-click the `Services` folder and then select the **Add, New Item** option from the menu.
4. Enter `interface` in the **Search** box.
5. Select **Interface**.
6. Name the file `ILocalStorageService`.
7. Click the **Add** button.
8. Update `ILocalStorageService` with the following highlighted code:

```
interface ILocalStorageService
{
    Task SetItemAsync<T>(string key, T item);
    Task<T> GetItemAsync<T>(string key);
}
```

## Creating the LocalStorageService class

We need to create a new class based on the interface we just created. We do this as follows:

1. Right-click the `Services` folder and select the **Add, Class** option from the menu.
2. Name the new class `LocalStorageService`.
3. Update the code to the `LocalStorageService` class to inherit from `ILocalStorageSerivce`:

```
public class LocalStorageService : ILocalStorageService
{
}
```

4. Right-click `ILocalStorageService` and select the **Implement interface** option from the menu.

5. Add the following code to the `LocalStorageService` class:

```
private IJSRuntime js;
public LocalStorageService(IJSRuntime JsRuntime)
{
    js = JsRuntime;
}
```

The preceding code defines the constructor for the `LocalStorageService` class.

6. Add the following `using` statement:

```
using Microsoft.JSInterop;
```

7. Update the `SetItemAsync` method to the following:

```
public async Task SetItemAsync<T>(string key, T item)
{
    await js.InvokeVoidAsync(
        "bweInterop.setLocalStorage",
        key,
        JsonSerializer.Serialize(item));
}
```

The `SetItemAsync` method invokes the `bweInterop.setLocalStorage` JavaScript function with a key and a serialized version of the item to be stored in localStorage.

8. Add the following `using` statement:

```
using System.Text.Json;
```

9. Update the `GetItemAsync` method to the following:

```
public async Task<T> GetItemAsync<T>(string key)
{
    var json = await js.InvokeAsync<string>(
        "bweInterop.getLocalStorage",
        key);

    return string.IsNullOrEmpty(json)
            ? default
```

```
                : JsonSerializer.Deserialize<T>(json);
    }
```

The `GetItemAsync` method invokes the `bweInterop.getLocalStorage` JavaScript function with a key. If `bweInterop.getLocalStorage` returns a value, that value is deserialized and returned.

We have completed our service. Now we need to test it.

## Writing to localStorage

We need to test writing to the browser's localStorage using our local storage service. We do this as follows:

1. Open the `Pages\Index.razor` file.
2. Add the following markup:

```
@using LocalStorage.Services
@inject IJSRuntime js

<h2>Local Storage Service</h2>

<div>
    Data:
    <input type="text"
           @bind-value="data"
           size="50" />
</div>
<div class="pt-2">
    <button class="btn btn-primary"
            @onclick="SaveToLocalStorageAsync">
        Save to Local Storage
    </button>
</div>
@code {
}
```

The preceding markup adds a textbox for the data to be saved to the browser's localStorage and a button that is used to call the `SaveToLocalStorageAsync` method.

3. Add the following code to the code block:

```
private string data;
private LocalStorageService localStorage;

protected override void OnInitialized()
{
    localStorage = new LocalStorageService(js);
}

async Task SaveToLocalStorageAsync()
{
    await localStorage.SetItemAsync<string>(
        "localStorageData",
        data);
}
```

The preceding code initializes the component and defines the `SaveToLocalStorageAsync` method. The `SaveToLocalStorageAsync` method uses `localStorageData` as the key when saving the data to localStorage.

From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project:


Figure 4.7 – Local Storage Service test page

4. Enter the word `Test` into the **Data** textbox.
5. Click the **Save to Local Storage** button.
6. Click *F12* to open the Developer Tools.
7. Select the **Application** tab.
8. Open **Local Storage**.

The following screenshot shows the value of `localStorageData`:

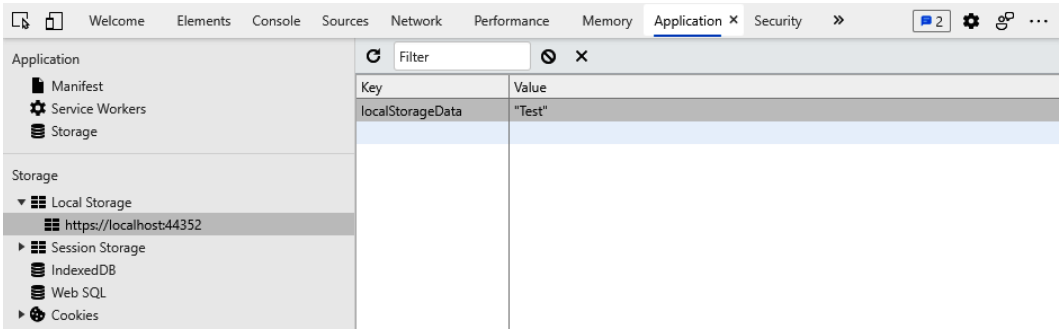



Figure 4.8 – Local storage

We have used the **Local Storage API** to save data to the browser's localStorage. Next, we need to learn how to read from the browser's localStorage.

## Reading from localStorage

We need to test reading from the browser's localStorage using our local storage service. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `Pages\Index.razor` file.
3. Add the following button beneath the existing button:

```
<button class="btn btn-primary"
        @onclick="ReadFromLocalStorageAsync">
    Read from Local Storage
</button>
```

   The preceding markup adds a button used to call the `ReadFromLocalStorageAsync` method.

4. Add the following method to the code block:

```
async Task ReadFromLocalStorageAsync()
{
    data = await localStorage.GetItemAsync<string>(
        "localStorageData");
}
```

The preceding code defines the `ReadFromLocalStorageAsync` method. `ReadFromLocalStorageAsync` uses the `localStorageData` key when accessing the browser's localStorage.

5. From the **Build** menu, select the **Build Solution** option.
6. Return to the browser.
7. Use *Ctrl* + *R* to refresh the browser.
8. Click the **Read from Local Storage** button.

We have now completed the testing of our local storage service.

# Summary

You should now be able to create a local storage service by using JS interop to invoke JavaScript functions from your Blazor WebAssembly application.

In this chapter, we explained why you still need to use JavaScript and how to use the **IJSRuntime** abstraction to invoke JavaScript functions from .NET, both synchronously and asynchronously. Conversely, we explained how to invoke .NET methods from JavaScript. Finally, we explained how to store data in the browser by using localStorage.

After that, we used the **Empty Blazor App** project template to create a new project. We added a couple of JavaScript functions to read and write localStorage. Then, we added a class to invoke those JavaScript functions.

In the last part of the chapter, we tested our local storage service.

One of the biggest benefits of using Blazor WebAssembly is that all of the code runs on the browser. This means that a web app built using Blazor WebAssembly can run offline. In the next chapter, we will leverage this benefit to create a progressive web app.

# Questions

The following questions are provided for your consideration:

1. Can `IJSRuntime` be used to render a UI?
2. How would you add our local storage service to a Razor class library?
3. In what scenarios would you use sessionStorage rather than localStorage?
4. Is localStorage secure?
5. How do you verify that the browser supports localStorage and that it is available for use?

# Further reading

The following resources provide more information regarding the topics covered in this chapter:

- For more information on using JavaScript, refer to `https://www.w3schools.com/js`.
- For more detailed information on JavaScript, refer to `https://developer.mozilla.org/en-US/docs/Web/javascript`.
- For the JavaScript reference, refer to `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference`.
- For more information on localStorage, refer to `https://www.w3.org/TR/webstorage/#the-localstorage-attribute`.
- For more information on the **Storage Living Standard**, refer to `https://storage.spec.whatwg.org`.
- For more information on **Microsoft Edge (Chromium) Developer Tools**, refer to `https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium`.

# 5

# Building a Weather App as a Progressive Web App (PWA)

As web developers, we develop amazing web apps of all kinds, but until recently there has been a divide between what a web app can do versus what a native app can do. A new class of apps called **Progressive Web Apps** (**PWAs**) is helping to bridge that divide by enabling native-like capabilities, reliability, and installability in our web apps. A PWA is a web application that takes advantage of native app features while retaining all of the features of a web app.

In this chapter, we will learn what defines a PWA, as well as how to create a PWA by adding a **manifest file** and a **service worker** to an existing web application.

The project that we create in this chapter will be a local 5-day weather forecast application that can be installed and run as a native application on Windows, Macs, iPhones, Android phones, and so on and can be distributed through the various app stores. We will use JavaScript's **Geolocation API** to obtain the location of the device and use the **OpenWeather One Call API** to fetch the weather forecast for that location. We will convert the application into a PWA by adding a manifest file and a service worker. The service worker will use the **CacheStorage API** to cache information so that the PWA can work offline.

In this chapter, we will cover the following topics:

- Understanding PWAs
- Working with manifest files
- Working with service workers
- Using the CacheStorage API
- Using the Geolocation API
- Using the OpenWeather One Call API
- Creating a PWA

# Technical requirements

To complete this project, you need to have **Visual Studio 2019** installed on your PC. For instructions on how to install the free Community Edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*.

We will be using an external weather API to access the weather forecast data for our project. The API that we will be using is the OpenWeather One Call API for getting current, forecasted, and historical weather data. This is a free API that is provided by **OpenWeather** (`https://openweathermap.org`). In order to get started with this API, you need to create an account and obtain an API key. If you do not want to create an account, you can use the `weather.json` file that we have provided in the GitHub repository for this chapter.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter05`.

The code in action video is available here: `https://bit.ly/3u2CrbX`.

# Understanding PWAs

A PWA is a web app that uses modern web capabilities to deliver an app-like experience to users. They look and feel like a native application because they run in their own app window instead of the browser's window, and they can be launched from the **Start** menu or taskbar. PWAs offer an offline experience and load instantly due to their use of caching. They can receive push notifications and are automatically updated in the background. Finally, although they do not require a listing in an app store for distribution, they can be distributed through the app stores.

Many large companies such as Pinterest, Starbucks, Trivago, and Twitter have embraced PWAs. Companies are drawn to PWAs because they can develop them once and use them everywhere.

A PWA feels like a native application due to a combination of technologies. In order to convert a web app into a PWA, it must use **HyperText Transfer Protocol Secure** (**HTTPS**) and include both a manifest file and a service worker.

## HTTPS

To be converted into a PWA, the web app must use HTTPS and must be served over a secure network. This should not be a problem since most browsers will no longer serve pages over HTTP. Therefore, even if you are not planning to convert a Blazor WebAssembly app into a PWA, you should always be using HTTPS.

> **Tip**
>
> A **Secure Sockets Layer** (**SSL**) certificate is required to enable HTTPS. A great source for free SSL certificates is **Let's Encrypt** (`https://letsencrypt.org`). It is a free, automated, and open **Certificate Authority** (**CA**).

## Manifest files

A manifest file is a simple **JavaScript Object Notation** (**JSON**) document that contains an application's name, defaults, and startup parameters for when a web application is launched. It describes how an application looks and feels.

This is an example of a simple manifest file:

```
{
  "name": "My Sample PWA",
  "display": "standalone",
  "background_color": "#ffffff",
```

```
  "theme_color": "#03173d",
  "icons": [
    {
      "src": "icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ]
}
```

A manifest file must include the name of the application and at least one icon. We will look more closely at manifest files in the next section.

## Service workers

A service worker is a JavaScript file that defines the offline experience for the PWA. It intercepts and controls how a web browser handles its network requests and asset caching.

This is the content of the `service-worker.js` file that is included in the **Blazor WebAssembly PWA** project template provided by Microsoft:

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

It is only one line of code and—as you can see—it does not actually do anything, but it currently counts as a service worker and is all that is technically needed to convert an application into a PWA. We will take a closer look at more robust service workers later in this chapter.

A PWA is a web app that can be installed on a device like a native application. If a web app uses HTTPS and includes a manifest file and a service worker, it can be converted into a PWA. Let's take a closer look at manifest files.

# Working with manifest files

A manifest file provides information about an app in JSON format. It is usually in the root folder of an application. The following code snippet shows how to add a manifest file named `manifest.json` to the `index.html` file:

```
<link href="manifest.json" rel="manifest" />
```

Here is a sample manifest file that includes many possible fields:

```
{
  "dir": "ltr",
  "lang": "en",
  "name": " 5-Day Weather Forecast",
  "short_name": "Weather",
  "scope": "/",
  "display": "standalone",
  "start_url": "./",
  "background_color": "transparent",
  "theme_color": "transparent",
  "description": "This is a 5-day weather forecast.",
  "orientation": "any",
  "related_applications": [],
  "prefer_related_applications": false,
  "icons": [
    {
      "src": "icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "url": "https://bweweather.azurewebsites.net",
  "screenshots": []
}
```

As mentioned earlier, a manifest file must include the name of the application and at least one icon. Beyond that, everything else is optional, although is it highly recommended that you include `description`, `short_name`, and `start_url` as a minimum.

These are the keys in the `manifest.json` file:

- `dir`: The base direction of the `name`, `short_name`, and `description`. It is either `ltr`, `rtl`, or `auto`.
- `lang`: The primary language of the `name`, `short_name`, and `description`.
- `name`: The name of the app. The maximum length is `45` characters.
- `short_name`: The short name of the app. The maximum length is `12` characters.

- `scope`: The navigation scope of the app.
- `display`: The way the app is displayed, set to `fullscreen`, `standalone`, `minimal-UI`, or `browser`.
- `start_url`: The **Uniform Resource Locator** (**URL**) of the app.
- `background_color`: The color used for the app's background during installation on the splash screen.
- `theme_color`: The default theme color.
- `description`: A short description of the app.
- `orientation`: The default screen orientation, set to `any`, `landscape`, or `portrait`.
- `related_applications`: Any related apps that the developer wishes to highlight. These are usually native apps.
- `prefer_related_applications`: A value notifying the user agent that the related application is preferred over a web app.
- `icons`: One or more images used by the app.
- `url`: The address of the app.
- `screenshots`: An array of images of the app in action.

The largest section of the `manifest.json` file is often the list of images. The reason for this is that many devices prefer images of different sizes.

> **Tip**
> An easy way to quickly generate many different sizes of the same image is to use the **Image Generator** tool on the **PWABuilder** website, found at `https://www.pwabuilder.com/generate`.

A manifest file controls how the PWA appears to the user and is required in order to convert a web app into a PWA. A service worker is also required to convert a web app into a PWA. Let's take a closer look at service workers.

# Working with service workers

Service workers provide the magic behind PWAs. They are used for caching, background syncing, and push notifications. A service worker is a JavaScript file that intercepts and modifies navigation and resource requests. It gives us full control over which resources are cached and how our PWA behaves in different situations.

A service worker is simply a script that your browser runs in the background. It is separate from the app and has no **Document Object Model** (**DOM**) access. It runs on a different thread than the thread used by the main JavaScript that powers your app, so it is not blocking. It is designed to be fully asynchronous.

## Service worker life cycle

When working with service workers, it is very important to understand their life cycle because offline support can add a significant amount of complexity to the web app. There are three steps in the life cycle of a service worker—*install*, *activate*, and *fetch*, as illustrated in the following diagram:

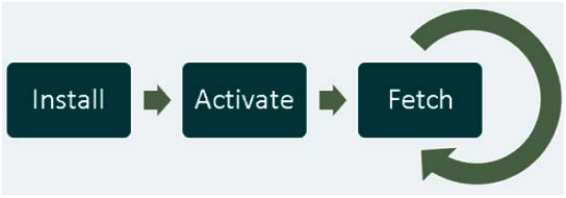



Figure 5.1 – Service worker life cycle

### Install

During the install step, the service worker usually caches some of the static assets of the website, such as a **You are offline** splash screen. If the files are cached successfully, the service worker is installed. However, if any of the files fail to download and cache, the service worker is not installed and does not move to the activate step.

If the service worker does not successfully install, it will try to install the next time the web app is run. Therefore, the developer can be assured that if the service worker has been successfully installed, the cache contains all of the static assets that were designated to be cached. After the install step is successfully completed, the activate step is initiated.

### Activate

During the activate step, the service worker handles the management of the old caches. Since a previous install may have created a cache, this is our opportunity to delete it. After the activate step is successfully completed, the service worker is ready to begin processing the fetch events.

### Fetch

During the fetch step, the service worker controls all of the pages that fall under its scope. It will handle the fetch events that occur when a network request is made from the PWA. The service worker will continue to fetch until it is terminated.

## Updating a service worker

In order to update the service worker that is running for our website, we need to update the service worker's `JavaScript` file. Each time a user navigates to our site, the browser downloads the current service worker and compares it with the installed service worker. If they are different, it will attempt to replace the old service worker.

However, this does not happen immediately. The new service worker has to wait until the old service worker is no longer in control before it can be activated. The old service worker will remain in control until all of the open pages are closed. When the new service worker takes control, its activate event will fire.

Cache management is handled during the `activate` callback. The reason we manage the cache during the `activate` callback is that if you were to wipe out any old caches in the install step, the old service worker (which has control of all the current pages) would suddenly stop being able to serve files from that cache.

The following screenshot shows a service worker that is **waiting to activate**:

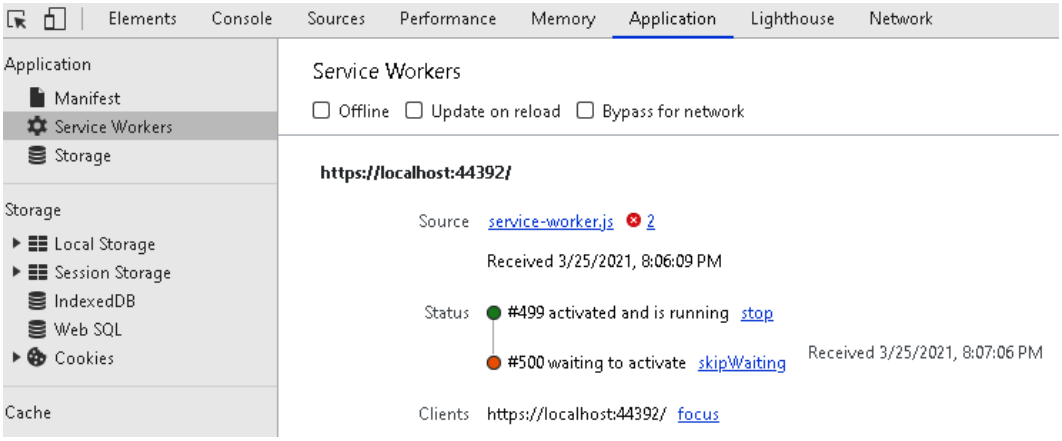



Figure 5.2 – Service worker waiting to activate

> **Tip**
> The service worker will not be activated until the user has navigated away from the app in all tabs. Reloading the tab will not suffice. However, you can activate a service worker that is **waiting to activate** by clicking the **skipWaiting** link.

## Types of service workers

There are many different types of service workers, from the ridiculously simple to the more complex. The following diagram shows some of the different types of service workers, ordered from simple to complex:

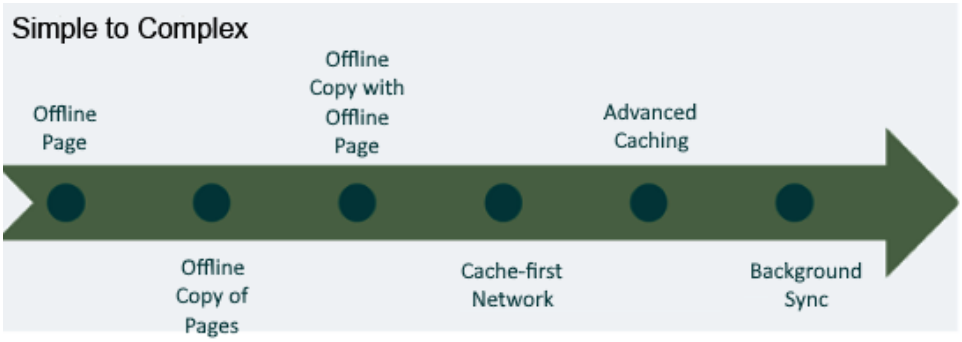



Figure 5.3 – Types of service workers

## Offline page

This is the simplest type of functioning service worker to create. All we need in order to create this type of service worker is a **HyperText Markup Language** (**HTML**) page that indicates an application is offline. Whenever an application is unable to connect to a network, we simply display that HTML page.

## Offline copy of pages

With this type of service worker, we store a copy of each page in the cache as our visitors view them. When an application is offline, it serves the pages from the cache. This approach may only work for applications with a limited number of pages because, if a page that a user wants to view has not yet been viewed by that user, it will not yet be in the cache and the app will fail.

## Offline copy with offline page

This type of service worker is an improved version of the offline copy of pages service worker. It combines the two previous types of service workers. With this type of service worker, we store a copy of each page in the cache as our visitors view them. When an application is offline, it serves the pages from the cache. If a page that a user wants to view is not in the cache, we display the HTML page that indicates the application is offline.

## Cache-first network

This type of service worker always uses the cache first. If the requested page is in the cache, it serves that page before it requests the page from the server and updates the cache with the new page. Using this service worker, we always serve the version of the page that is in the cache before requesting the page from the server, thus users are served the same data whether they are online or offline.

> **Tip**
> A cache-first network service worker type is preferred by Microsoft.

## Advanced caching

This type of service worker is a combination of each of the preceding types. With this type of service worker, we designate different files and routes to be cached using different rules. For example, some data such as stock prices should never be cached, while other data that does not change very often should be cached.

### Background sync

This is the most complex type of service worker. It allows a user to continue to use an application to add and edit data when they are offline. Then, when they are back online, the application will sync their data with the network.

This is not a complete list of all of the different types of service workers that are available. However, it should give you an idea of the power and flexibility of service workers and the importance of caching. All of the service workers on our list rely on the CacheStorage API for caching.

## Using the CacheStorage API

The **CacheStorage API** is used to cache `request/response` object pairs where the `request` objects are the keys and the `response` objects are the values. It was designed to be used by service workers to provide offline functionality. A `caches` object is an instance of **CacheStorage**. It is a global object that is located in the `window` object.

We can use the following code to test if it is available on the browser:

```
  const hasCaches = 'caches' in self;
```

A `caches` object is used to maintain a list of caches for a particular web app. Caches cannot be shared with other web apps and they are isolated from the browser's HTTP cache. They are entirely managed through the JavaScript that we write.

These are some of the methods of **CacheStorage**:

- `delete(cacheName)`: This method deletes the indicated cache and returns `true`. If the indicated cache is not found, it returns `false`.
- `has(cacheName)`: This method returns `true` if the indicated cache exists, and `false` otherwise.
- `keys`: This method returns a string array of the names of all of the caches.
- `open(cacheName)`: This method opens the indicated cache. If it does not exist, it is created and then opened.

When we open an instance of **CacheStorage**, a `Cache` object is returned. These are some of the methods of a `Cache` object:

- `add(request)`: This method takes a request and adds the resulting response to the cache.
- `addAll(requests)`: This method takes an array of requests and adds all of the resulting responses to the cache.
- `delete(request)`: This method returns `true` if it is able to find and delete the indicated request, and `false` otherwise.
- `keys()`: This methods returns an array of keys.
- `match(request)`: This method returns a response associated with the matching request.
- `put(request, response)`: This method adds the request and response pair to the cache.

> **Tip**
> A `Cache` object does not get updated unless we explicitly request it to be updated. Also, these objects do not expire. We need to delete them as they become obsolete.

Service workers use the CacheStorage API to allow the PWA to continue to function when it is offline. Next, we will explain how to use the Geolocation API.

# Using the Geolocation API

The Geolocation API for JavaScript provides a mechanism for us to obtain the location of a user. Using the Geolocation API, we can obtain the coordinates of a device that the browser is running on.

The Geolocation API is accessed through a `navigator.geolocation` object. When we make a call to the `navigator.geolocation` object, the user's browser asks the user for permission to access their location. If they accept, the browser uses the device's positioning hardware, such as the **Global Positioning System** (**GPS**) on a smart phone, to determine its location.

Before we attempt to use the `navigator.geolocation` object, we should verify that it is supported by the browser. The following code tests for the presence of geolocation support on the browser:

```
if (navigator.geolocation) {
    var position = await getPositionAsync();
} else {
    throw Error("Geolocation is not supported.");
};
```

For the project in this chapter, we will be using the `getCurrentPosition` method to retrieve the device's location. This method takes two callback functions. The `success` callback function returns a `GeolocationPosition` object, while the `error` callback function returns a `GeolocationPositionError` object. If the user denies us access to their position, it will be reported in the `GeolocationPositionError` object.

These are the properties of the `GeolocationPosition` object:

- `coords.latitude`: This property returns a double that represents the latitude of the device.
- `coords.longitude`: This property returns a double that represents the longitude of the device.
- `coords.accuracy`: This property returns a double that represents the accuracy of the latitude and the longitude, expressed in meters.
- `coords.altitude`: This property returns a double that represents the altitude of the device.
- `coords.altitudeAccuracy`: This property returns a double that represents the accuracy of the altitude, expressed in meters.
- `coords.heading`: This property returns a double that represents the direction in which the device is facing, expressed in degrees.
- `coords.speed`: This property returns a double that represents the speed of the device, expressed in meters per second.
- `timestamp`: This property returns the date and time of the response.

The `GeolocationPosition` object always returns the `coords.latitude`, `coords.longitude`, `coords.accuracy`, and `timestamp` properties. The other properties are only returned if they are available.

By using JavaScript's Geolocation API, we can determine the latitude and longitude of a device. We need this information in order to use the OpenWeather One Call API to request a local weather forecast for our project.

# Using the OpenWeather One Call API

The data source for the project in this chapter is a free API provided by **OpenWeather**. It is called the OpenWeather One Call API (`https://openweathermap.org/api/one-call-api`). This API is able to return current, forecast, and historical weather data. We will be using it to access the local forecast for the next 5 days. This is the format of an API call using the OpenWeather One Call API:

```
https://api.openweathermap.org/data/2.5/
onecall?lat={lat}&lon={lon}&appid={API key}
```

These are the parameters for the OpenWeather One Call API:

- `lat`: Latitude. This parameter is required.
- `lon`: Longitude. This parameter is required.
- `appid`: API key. This parameter is required. It is on the **Accounts** page under the **API key** tab.
- `units`: Units of measurement. This is set as **Standard**, **Metric**, or **Imperial**.
- `exclude`: Excluded data. This is used to simplify data that is returned. Since we will only be using the daily forecast, we will exclude current, minutely, and hourly data, and alerts for our project. This is a comma-delimited list.
- `lang`: Language of the output.

This is a fragment of the response from the OpenWeather One Call API:

**weather.json fragment**

```
{
  "dt": 1616436000,
  "sunrise": 1616416088,
  "sunset": 1616460020,
```

```
    "temp": {
      "day": 58.5,
      "min": 54.75,
      "max": 62.6,
      "night": 61.29,
      "eve": 61.25,
      "morn": 54.75
    },
    "feels_like": {
      "day": 49.69,
      "night": 51.91,
      "eve": 50.67,
      "morn": 47.03
    },
    "pressure": 1011,
    "humidity": 85,
    "dew_point": 54.01,
    "wind_speed": 17.83,
    "wind_deg": 168,
    "weather": [
      {
        "id": 502,
        "main": "Rain",
        "description": "heavy intensity rain",
        "icon": "10d"
      }
    ],
    "clouds": 98,
    "pop": 1,
    "rain": 27.91,
    "uvi": 2.34
  },
```

In the preceding JSON fragment, we have highlighted the fields that we are using in this chapter's project.

The OpenWeather One Call API is a simple API that we will be using to obtain the daily forecast for a given location. Now, let's get a quick overview of the project that we are going to build in this chapter.

# Project overview

In this chapter, we will build a Blazor WebAssembly app to display a local 5-day weather forecast and then convert it into a PWA.

The web app we will build uses JavaScript's Geolocation API to determine the current latitude and longitude of the device. It uses the OpenWeather One Call API to obtain the local weather forecast and uses a variety of Razor components to display the weather forecast to the user. After we have completed the web app, we will convert it into a PWA by adding a logo, a manifest file, and a service worker. Finally, we will install, run, and uninstall the PWA.

This is a screenshot of the completed application:

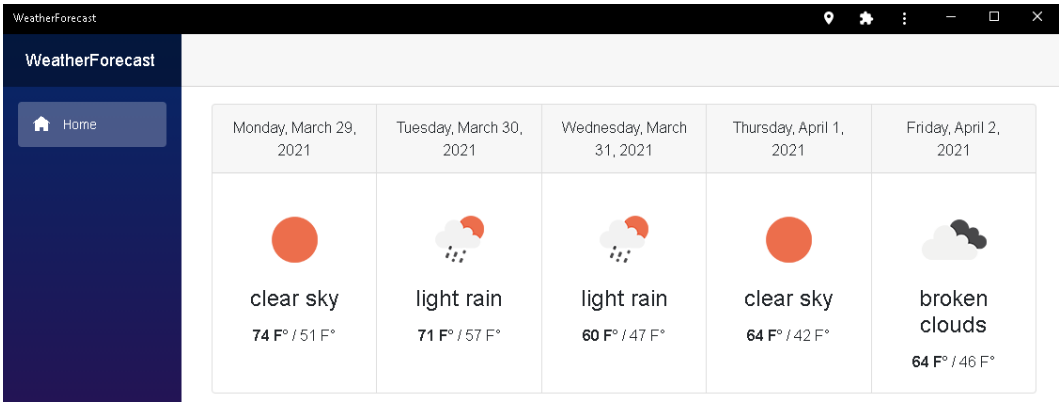



Figure 5.4 – WeatherForecast application

The build time for this project is approximately 120 minutes.

# Creating a PWA

A `WeatherForecast` project will be created by using the **Empty Blazor WebAssembly App** project template. First, we will use JS interop with the Geolocation API to obtain the coordinates of the device. We will then use the OpenWeather One Call API to obtain a weather forecast for those coordinates. Next, we will create a couple of Razor components to display the forecast.

In order to convert the web app into a PWA, we will add a logo, a manifest file, and an offline page service worker. After testing the service worker, we will install, run, and uninstall the PWA.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We will do this by following these steps:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt* + *S*) textbox, enter `blazor` and hit the *Enter* key.

   The following screenshot shows the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*:

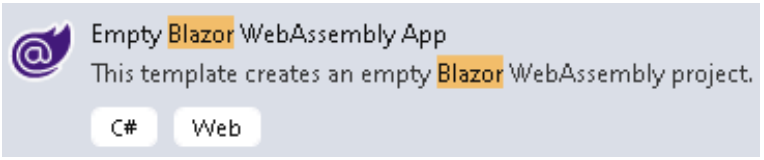



Figure 5.5 – Empty Blazor WebAssembly App project template

4. Select the **Empty Blazor WebAssembly App** project template and click the **Next** button.

5. Enter `WeatherForecast` in the **Project name** textbox and click the **Create** button, as illustrated in the following screenshot:

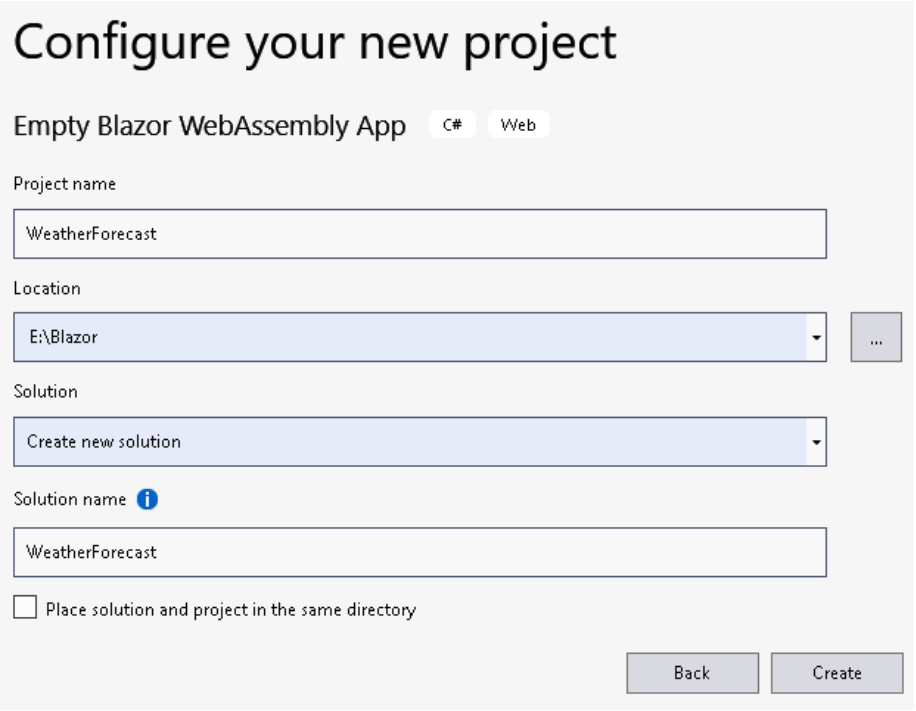



Figure 5.6 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `WeatherForecast` project into the `E:/Blazor` folder. However, the location of this project is not important.

We have now created a `WeatherForecast` Blazor WebAssembly project.

## Adding a JavaScript function

We now need to add a class to contain our current latitude and longitude. We will do this by following these steps:

1. Right-click the `wwwroot` folder and select the **Add, New Folder** option from the menu.
2. Name the new folder `scripts`.

3. Right-click the `scripts` folder and select the **Add, New Item** option from the menu.
4. Enter `javascript` in the **Search** box.
5. Select **JavaScript File**.
6. Name the file `bweInterop.js`.

> **Tip**
> In this book, we will be using the `bweInterop` namespace for our JavaScript code to both structure our code and minimize the risk of naming conflicts.

7. Click the **Add** button.
8. Enter the following JavaScript:

```
var bweInterop = {};

bweInterop.getPosition = async function () {
    function getPositionAsync() {
        return new Promise((success, error) => {
            navigator.geolocation.
getCurrentPosition(success, error);
        });
    }

    if (navigator.geolocation) {
        var position = await getPositionAsync();
        var coords = {
            latitude: position.coords.latitude,
            longitude: position.coords.longitude
        };
        return coords;
    } else {
        throw Error("Geolocation is not supported by
                    this browser.");
    };
}
```

The preceding JavaScript code uses the Geolocation API to return the latitude and longitude of the device. If it is not allowed or it is not supported, an error is thrown.

9. Open the `wwwroot\index.html` file.
10. Add the following reference toward the bottom of the `body` element:

```
<script src="scripts/bweInterop.js"></script>
```

You should add it right before the reference to `_framework/blazor.webassembly.js`.

We have created a JavaScript function that uses the Geolocation API to return our current latitude and longitude. Next, we need to invoke it from our web app.

## Using the Geolocation API

We need to invoke our `bweInterop.getPosition` function from our web app. We will do this by following these steps:

1. Right-click the `WeatherForecast` project and select the **Add, New Folder** option from the menu.
2. Name the new folder `Models`.
3. Right-click the `Models` folder and select the **Add, Class** option from the menu.
4. Name the new class `Position`.
5. Add the following highlighted properties to the `Position` class:

```
public class Position
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

This is the class that we will use to store our coordinates.

6. Open the `Pages\Index.razor` file.

7. Add the following markup:

```
@using WeatherForecast.Models
@inject IJSRuntime js

@if (pos == null)
{
    <p><em>@message</em></p>
}
else
{
    <h2>Latitude: @pos.Latitude, Longitude:
       @pos.Longitude </h2>
}

@code {
    string message = "Loading...";
    Position pos;
}
```

The preceding markup displays a message if the `pos` property is `null`. Otherwise, it displays the latitude and longitude from the `pos` property.

8. Add the following `OnInitializedAsync` method to the `@code` block:

```
protected override async Task OnInitializedAsync()
{
    try
    {
        await GetPosition();
    }
    catch (Exception)
    {
        message = "Geolocation is not supported.";
    };
}
```

The preceding code attempts to get our coordinates when the page initializes.

9. Add the following `GetPosition` method to the `@code` block:

```
private async Task GetPosition()
{
    pos = await js.InvokeAsync<Position>(
        "bweInterop.getPosition");
}
```

The preceding code uses JS interop to invoke the JavaScript function that we wrote that uses the Geolocation API to return our coordinates. For more information on JS interop, refer to *Chapter 4, Building a Local Storage Service Using JavaScript Interoperability (JS Interop)*.

10. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.

The following screenshot is an example of the dialog that will ask you for permission to access your location:

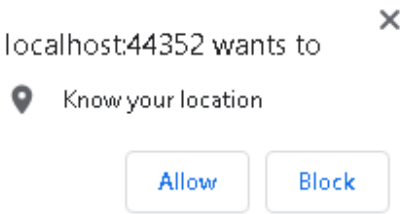



Figure 5.7 – Geolocation permission dialog

11. Click the **Allow** button to allow the app to have access to your location.

The following screenshot is of the updated **Home** page:


Figure 5.8 – Home page displaying coordinates

You can disable the app's ability to access your location by using the **Location access allowed** dialog that is shown in the following screenshot:

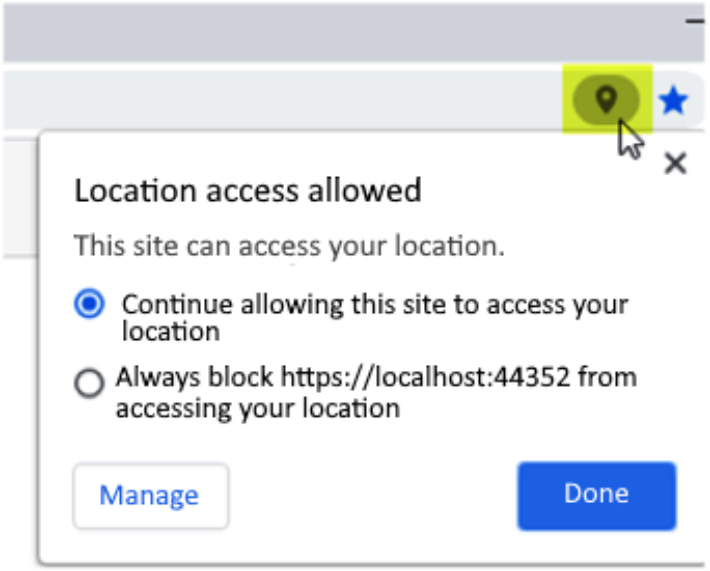



Figure 5.9 – Location access allowed dialog

The **Location access allowed** dialog is accessed via the highlighted button on the browser's toolbar. You may want to toggle the permissions to see how that impacts the app.

> **Tip**
> To change a URL's permission to access your location, select **Settings** from the browser's menu. Then, select **Site Settings** from the **Privacy and security** area. Finally, search for the URL that you are using, select it, and change the value of the **Location** field to one of the following: **Ask** (default), **Allow**, or **Block**.

We have used the Geolocation API to display our latitude and longitude on the **Home** page. Next, we need to provide those coordinates to the OpenWeather One Call API.

## Adding a Forecast class

We need to add a `Forecast` class to capture the results from the OpenWeather One Call API. We will do this by following these steps:

1. Return to **Visual Studio**.
2. Right-click the `Models` folder and select the **Add, Class** option from the menu.
3. Name the new class `OpenWeather`.
4. Add the following classes:

```
public class OpenWeather
{
    public Daily[] Daily { get; set; }
}

public class Daily
{
    public long Dt { get; set; }
    public Temp Temp { get; set; }
    public Weather[] Weather { get; set; }
}

public class Temp
{
    public double Min { get; set; }
    public double Max { get; set; }
}

public class Weather
{
    public string Description { get; set; }
    public string Icon { get; set; }
}
```

The preceding classes will be used with the OpenWeather One Call API.

## Adding a DailyForecast component

We need a component to display each day's forecast. We will do this by following these steps:

1. Right-click the `Shared` folder and select the **Add, Razor Component** option from the menu.
2. Name the new component `DailyForecast`.
3. Replace the existing markup with the following markup:

```
<div class="card text-center">
    <div class="card-header">
        @Date
    </div>
    <div class="card-body">
        <img src="@IconUrl" />
        <h4 class="card-title">@Description</h4>
        <b>@((int)HighTemp) F&deg;</b> /
        @((int)LowTemp) F&deg;
    </div>
</div>
@code {
}
```

   This component uses the `Card` component from **Bootstrap** to display the daily forecast. For more information on the `Card` component, see `https://getbootstrap.com/docs/5.0/components/card`.

4. Add the following code to the `@code` block:

```
[Parameter] public long Seconds { get; set; }
[Parameter] public double HighTemp { get; set; }
[Parameter] public double LowTemp { get; set; }
[Parameter] public string Description { get; set; }
[Parameter] public string Icon { get; set; }

private string Date;
private string IconUrl;

protected override void OnInitialized()
```

```
{
    Date = DateTimeOffset
        .FromUnixTimeSeconds(Seconds)
        .LocalDateTime
        .ToLongDateString();
    IconUrl = String.Format(
        "https://openweathermap.org/img/wn/{0}@2x.png",
        Icon);
}
```

The preceding code defines the parameters that are used to display the daily weather forecast. The `OnInitialized` method is used to format the `Date` and `IconUrl` fields.

We have added a `Razor` component to display each day's weather forecast using the `Code` component from Bootstrap.

## Using the OpenWeather One Call API

We need to fetch the weather forecast using the OpenWeather One Call API. We will do this by following these steps:

1. Open the `Pages\Index.razor` file.
2. Add the following `using` statement:

```
@using System.Text
```

3. Add the following `@inject` directive:

```
@inject HttpClient Http
```

4. Add the following property to the `@code` block:

```
OpenWeather forecast;
```

5. Add the `GetForecast` method to the `@code` block, as follows:

```
private async Task GetForecast()
{
    string APIKey = "{Your_API_Key}";

    StringBuilder url = new StringBuilder();
```

```
        url.Append("https://api.openweathermap.org");
        url.Append("/data/2.5/onecall?");
        url.Append("lat=");
        url.Append(pos.Latitude);
        url.Append("&lon=");
        url.Append(pos.Longitude);
        url.Append("&exclude=");
        url.Append("current,minutely,hourly,alerts");
        url.Append("&units=imperial");
        url.Append("&appid=");
        url.Append(APIKey);

        forecast = await Http
            .GetFromJsonAsync<OpenWeather>
            (url.ToString());
    }
```

The preceding method uses the OpenWeather One Call API with the coordinates obtained by the `GetPosition` method.

6. Update the `OnInitializedAsync` method to call the `GetForecast` method and update the error message, like this:

```
    try
    {
        await GetPosition();
        await GetForecast();
    }
    catch (Exception)
    {
        message = "Error encountered";
    };
```

The preceding code uses the `GetForecast` method to populate the `forecast` object.

> **Important note**
>
> You need to set the value of the `APIKey` string to the API key that you obtained from **OpenWeather**.

We have populated the `forecast` object. Next, we need to display it.

## Displaying the forecast

We need to add a collection of daily forecasts to the **Home** page. We will do this by following these steps:

1. Return to the `Pages\Index.razor` file.
2. Replace the `@if` statement with the following markup:

```
@if (forecast == null)
{
    <p><em>@message</em></p>
}
else
{
    <div class="card-group">
        @foreach (var item in forecast.Daily.Take(5))
        {
            <DailyForecast
               Seconds="@item.Dt"
               LowTemp="@item.Temp.Min"
               HighTemp="@item.Temp.Max"
               Description="@item.Weather[0].Description"
               Icon="@item.Weather[0].Icon" />
        }
    </div>
}
```

   The preceding markup loops through the forecast object five times. It uses the `DailyForecast` component to display the daily forecast.

3. From the **Build** menu, select the **Build Solution** option.
4. Return to the browser.
5. Use *Ctrl* + *R* to refresh the browser.
6. Close the browser.

We have completed our `WeatherForecast` application. Now, we need to convert it into a PWA. In order to do that, we need to add a logo, a manifest file, and a service worker.

## Adding the logo

We need to add an image to be used as a logo for the app. We will do this by following these steps:

1. Right-click the `wwwroot` folder and select the **Add, New Folder** option from the menu.
2. Name the new folder `images`.
3. Copy the `Sun-512.png` image from the GitHub repository to the `images` folder.

At least one image must be included in the manifest file for the PWA to be installed. Now, we can add a manifest file.

## Adding a manifest file

To convert the web app into a PWA, we need to add a manifest file. We will do this by following these steps:

1. Right-click the `wwwroot` folder and select the **Add, New Item** option from the menu.
2. Enter `json` in the **Search** box.
3. Select **JSON File**.
4. Name the file `manifest.json`.
5. Click the **Add** button.
6. Enter the following JSON code:

```
{
  "lang": "en",
  "name": "5-Day Weather Forecast",
  "short_name": "Weather",
  "display": "standalone",
  "start_url": "./",
  "background_color": "#ffa500",
  "theme_color": "transparent",
  "description": "This is a simple 5-day weather
     forecast application.",
  "orientation": "any",
  "icons": [
```

```
      {
        "src": "images/Sun-512.png",
        "type": "image/png",
        "sizes": "512x512"
      }
    ]
  }
```

7. Open the `wwwroot\index.html` file.
8. Add the following markup to the bottom of the `head` element:

```
<link href="manifest.json" rel="manifest" />
```

9. Add the following markup below the preceding markup:

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

> **Tip**
> For iOS Safari users, you must include the preceding link tag to instruct it to use the indicated icon or it will generate an icon by taking a screenshot of the page's content.

We have added a manifest file to our web app to control how it looks and behaves when it is installed. Next, we need to add a service worker.

## Adding a simple service worker

To finish converting the web app into a PWA, we need to add a service worker. We will do this by following these steps:

1. Right-click the `wwwroot` folder and select the **Add, New Item** option from the menu.
2. Enter `html` in the **Search** box.
3. Select **HTML Page**.
4. Name the file `offline.html`.
5. Click the **Add** button.

6. Add the following markup to the `body` element:

```
<h1>You are offline.</h1>
```

7. Right-click the `wwwroot` folder and select the **Add, New Item** option from the menu.
8. Enter `java` in the **Search** box.
9. Select **JavaScript File**.
10. Name the file `service-worker.js`.
11. Click the **Add** button.
12. Add the following constants:

```
const OFFLINE_VERSION = 1;
const CACHE_PREFIX = 'offline';
const CACHE_NAME = `${CACHE_PREFIX}${OFFLINE_VERSION}`;
const OFFLINE_URL = 'offline.html';
```

The preceding code sets the name of the current cache and the name of the file we will be using to indicate that we are offline.

13. Add the following event listeners:

```
self.addEventListener('install',
    event => event.waitUntil(onInstall(event)));
self.addEventListener('activate',
    event => event.waitUntil(onActivate(event)));
self.addEventListener('fetch',
    event => event.respondWith(onFetch(event)));
```

The preceding code designates the functions to be used for each of the following steps: install, activate, and fetch.

14. Add the following `onInstall` function:

```
async function onInstall(event) {
    console.info('Service worker: Install');
    const cache = await caches.open(CACHE_NAME);
    await cache.add(new Request(OFFLINE_URL));
}
```

The preceding function opens the indicated cache. If the cache does not yet exist, it creates the cache and then opens it. After the cache is open, it adds the indicated request/response pair to the cache.

15. Add the following `onActivate` function:

```
async function onActivate(event) {
    console.info('Service worker: Activate');
    const cacheKeys = await caches.keys();
    await Promise.all(cacheKeys
        .filter(key => key.startsWith(CACHE_PREFIX)
            && key !== CACHE_NAME)
        .map(key => caches.delete(key)));
}
```

The preceding code fetches the names of all of the caches. All of the caches that do not match the name of the indicated cache are deleted.

> **Tip**
> It is your responsibility to purge obsolete caches. Each browser has a limit as to the amount of storage that a web app can use. If you violate that limit, all of your caches may be deleted by the browser.

16. Add the following `onFetch` function:

```
async function onFetch(event) {
    if (event.request.method === 'GET') {
        try {
            return await fetch(event.request);
        } catch (error) {
            const cache = await
               caches.open(CACHE_NAME);
            return await cache.match(OFFLINE_URL);
        };
    };
}
```

In the preceding code, if the fetch fails, the cache is opened and the previously cached offline page is served.

17. Open the `wwwroot\index.html` file.
18. Add the following markup to the bottom of the `body` element:

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

We have added an offline page service worker that will display the `offline.html` page when the PWA is offline.

## Testing the service worker

We need to test that the service worker is allowing us to work offline. We will do this by following these steps:

1. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.
2. Click *F12* to open the **Developer Tools** interface.
3. Select the **Application** tab.

4. Select the **Manifest** option from the menu on the left to view the **App Manifest**.

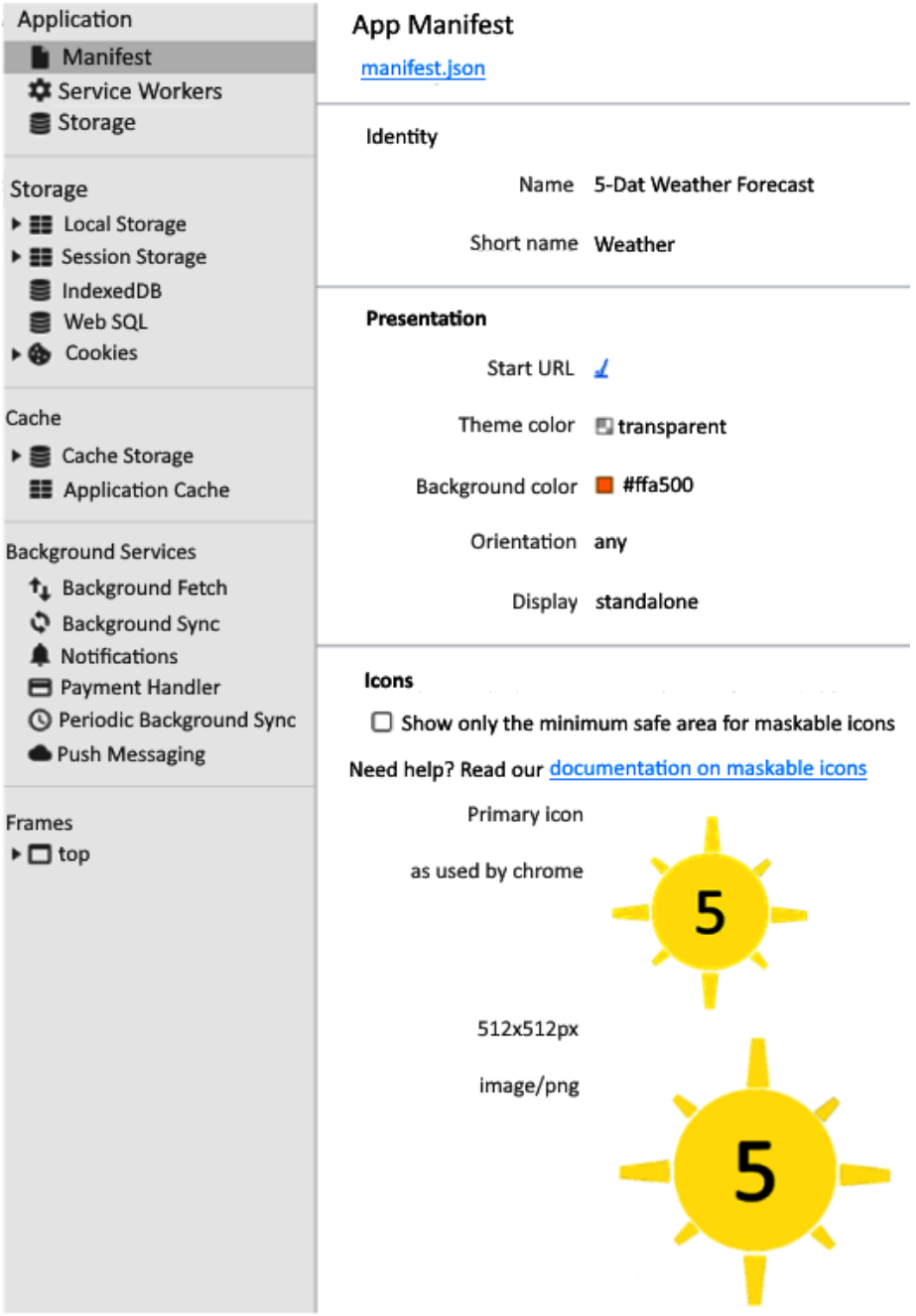



Figure 5.10 – App Manifest details

5. Select the **Service Workers** option from the menu on the left to view the service worker that is installed for the current client, as illustrated in the following screenshot:

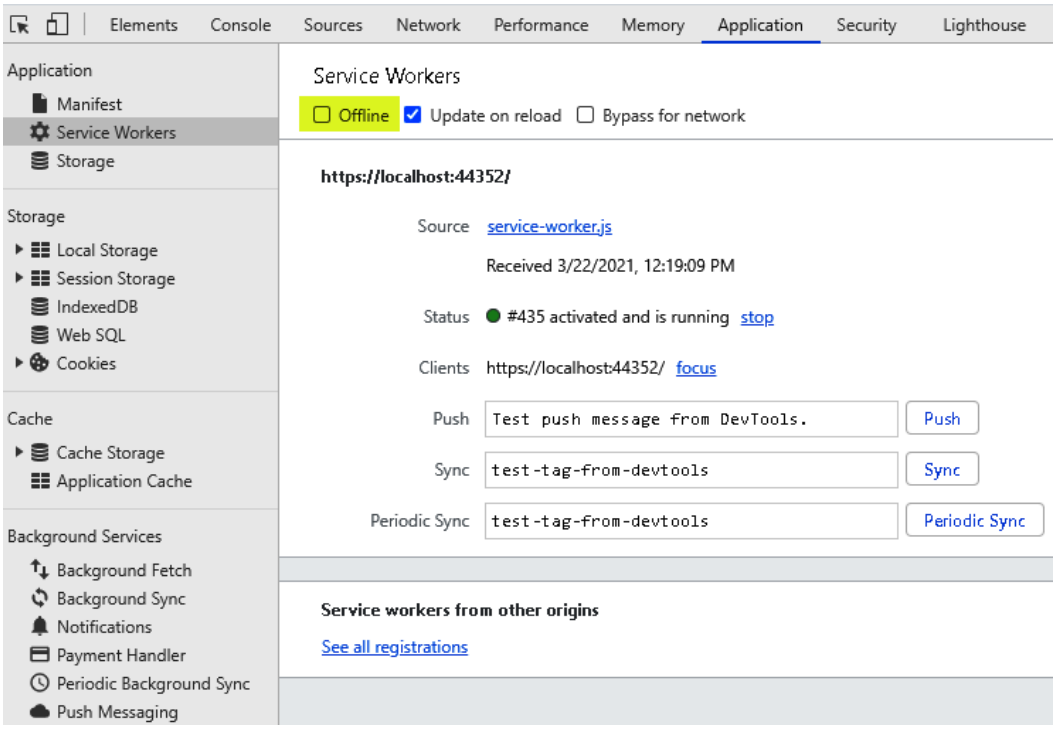



Figure 5.11 – Service Workers dialog

> **Tip**
> Click on the **See all registrations** link to see all of the service workers that are installed on your device.

6. Select the **Cache Storage** option from the menu on the left to view the caches.

7. Click on the `offline1` cache to view its contents, as illustrated in the following screenshot:

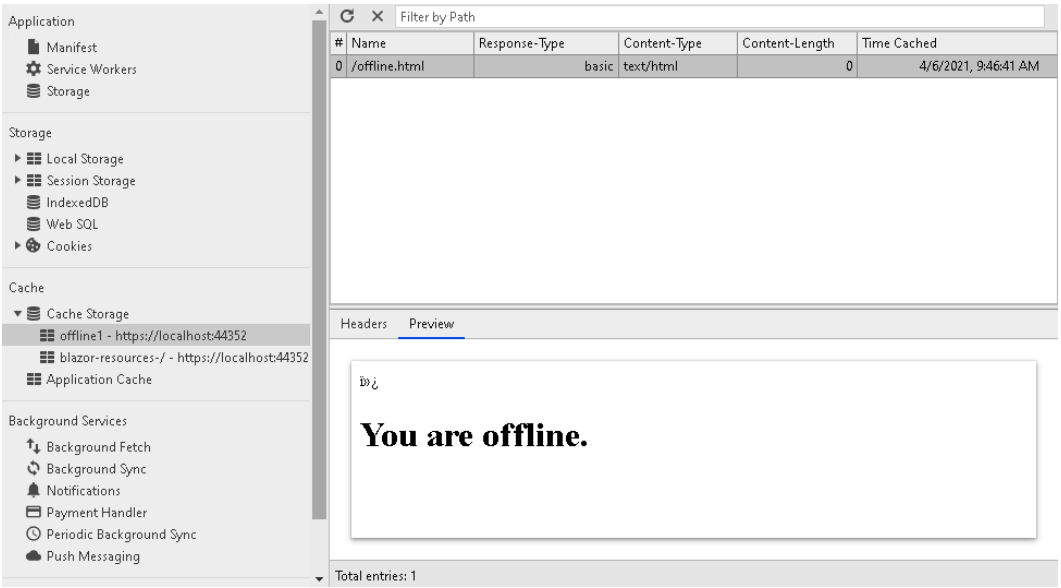



Figure 5.12 – Cache Storage option

8. Select the **Service Workers** option from the menu on the left.
9. Check the **Offline** checkbox on the **Service Workers** dialog.
10. Refresh the browser, and you should see the following screen:

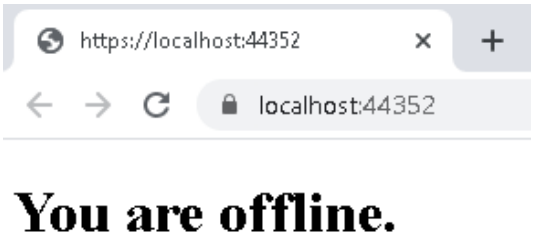



Figure 5.13 – Offline page

The page that is displayed is from the browser's cache.

11. Uncheck the **Offline** checkbox on the **Service Workers** dialog.
12. Refresh the browser.

We have tested that the service worker enables our web app to work offline. Now, we can install the PWA.

## Installing the PWA

We need to test the PWA by installing it. We will do this by following these steps:

1. Select the **Install 5-Day Weather Forecast** menu option from the browser's menu:

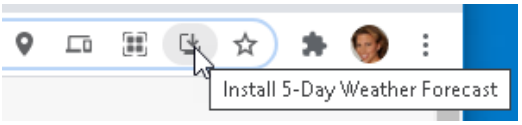



Figure 5.14 – Install 5-Day Weather Forecast option

> **Tip**
> On Chromium-based browsers, the **Install** button is on the URL bar. However, for other types of browsers, you will need to install the PWA from either the **Menu** button or the **Share** button.

2. Click the **Install** button on the dialog:

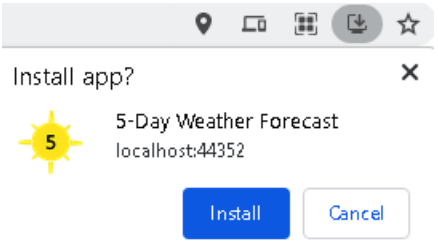



Figure 5.15 – Install PWA dialog

Once installed, the PWA appears without an address bar. It appears on our taskbar and we can run it from our **Start** menu. The following screenshot shows the PWA after it has been installed:

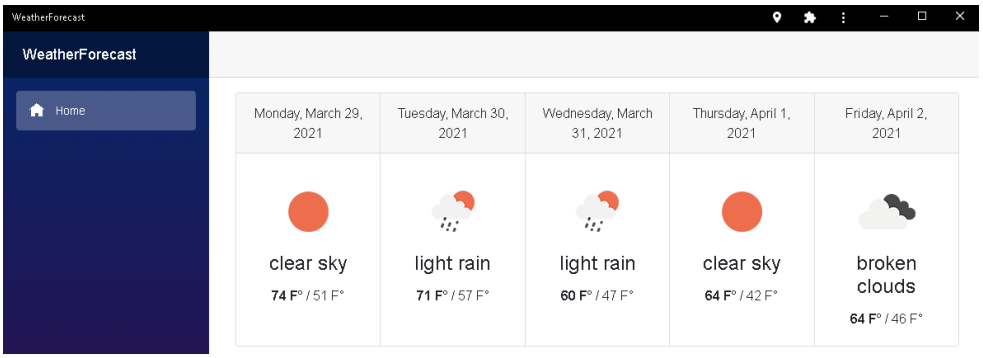



Figure 5.16 – Installed PWA

3. Close the browser.
4. Click the *Windows* key and open the **5-Day Weather Forecast** app:

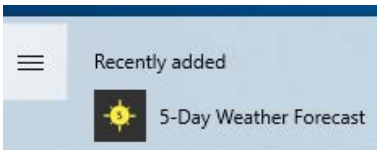



Figure 5.17 – The PWA on the Start menu

The application opens, and its icon appears on the taskbar. We can pin it to the taskbar if we want.

We have successfully installed and run the PWA. It is just as easy to uninstall a PWA as it is to install one.

## Uninstalling the PWA

We need to uninstall the PWA. We will do this by following these steps:

1. Select the **Customize and control 5-Day Weather Forecast** option from the PWA's menu:


Figure 5.18 – Customize and control 5-Day Weather Forecast option

2. Select the **Uninstall 5-Day Weather Forecast…** option:

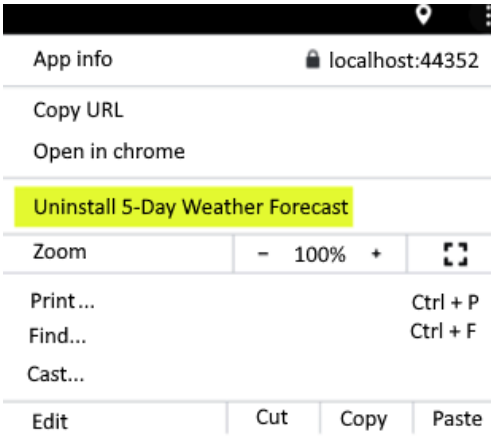



Figure 5.19 – Customize and control PWA dialog

3. Click the **Remove** button:

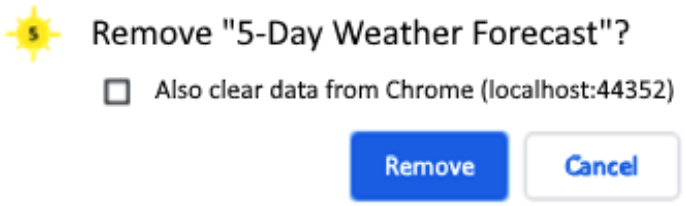



Figure 5.20 – Remove PWA dialog

We have uninstalled the PWA.

# Summary

You should now be able to convert a Blazor WebAssembly app into a PWA by adding a manifest file and a service worker.

In this chapter, we introduced PWAs. We explained how to convert a web app into a PWA by adding a manifest file and a service worker. We explained how to work with manifest files and service workers. We went into some detail explaining the different types of service workers and explained how to use the CacheStorage API to cache request/response pairs. Finally, we demonstrated how to use both the Geolocation API and the OpenWeather One Call API.

After that, we used the **Empty Blazor App** project template to create a new project. We added a JavaScript function that uses the Geolocation API to obtain our coordinates. We added some models to capture the coordinates and used JS interop to invoke the JavaScript function. We used the OpenWeather One Call API to obtain the local 5-day weather forecast and we created a couple of Razor components to display it.

In the last part of the chapter, we converted the Blazor WebAssembly app into a PWA by adding an image, a manifest file, and an offline page service worker. Finally, we installed, ran, and uninstalled the PWA. We can apply our new skills to convert our existing web apps into PWAs that combine the benefits of a web app with the look and feel of a native app.

In the next chapter, we will use **dependency injection** (**DI**) to build a shopping-cart application.

# Questions

The following questions are provided for your consideration:

1. Are service workers asynchronous or synchronous?
2. Can `localStorage` be used inside a service worker for data storage?
3. Can service workers manipulate the DOM?
4. Are PWAs secure?
5. Are PWAs platform-specific?
6. What are the differences between a PWA and a native app?

# Further reading

The following resources provide more information concerning the topics in this chapter:

- For more information on the Geolocation API specification, refer to `https://w3c.github.io/geolocation-api`.
- For more information on using the Geolocation API, refer to `https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API`.
- For more information on the OpenWeather API, refer to `https://openweathermap.org/api`.
- For more information on the **Web Application Manifest** specification, refer to `https://www.w3.org/TR/appmanifest/`.
- For more information on the **Service Worker** specification, refer to `https://w3c.github.io/ServiceWorker`.
- For more information on using the CacheStorage API**,** refer to `https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage`.
- For more examples of service workers, refer to the *Workbox* website at `https://developers.google.com/web/tools/workbox`.
- For more information on Microsoft's **PWABuilder**, refer to `https://www.pwabuilder.com`.

# 6
# Building a Shopping Cart Using Application State

Sometimes, we need our applications to maintain state between different pages. We can accomplish this by using **dependency injection** (**DI**). DI is used to access services that are configured in a central location.

In this chapter, we will create a shopping cart. As you add and delete items from the shopping cart, the application will maintain a list of the items in the shopping cart. The contents of the shopping cart will be retained when a user navigates to another page and then returns to the page with the shopping cart. Also, the shopping cart's total will be displayed on all of the pages.

In this chapter, we will cover the following topics:

- Application state
- Dependency injection
- Creating the shopping cart project

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free Community Edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter06`.

The code in action video is available here: `https://bit.ly/3fxwYob`.

# Application state

In a Blazor WebAssembly app, the browser's memory is used to hold the application's state. This means that when the user navigates between pages, the state is lost, unless we preserve it. We will be using the **AppState** pattern to preserve the application's state.

In the **AppState** pattern, a service is added to a DI container to coordinate the state between related components. The service contains all of the states that need to be maintained. Because the service is managed by the DI container, it can outlive individual components and retain the state of the application as the UI is changing.

The service can be a simple class or a complex class. One service can be used to manage the state of multiple components across the entire application. A benefit of the **AppState** pattern is that it leads to a greater separation between presentation and business logic.

> **Important note**
>
> The application state that is held in the browser's memory is lost when the user reloads the page.

For the project in this chapter, we will use a DI service instance to preserve the application's state.

# Understanding DI

DI is a technique in which an object accesses services that have been configured in a central location. The central location is the DI container. When using DI, each consuming class does not need to create its own instance of the injected class that it has a dependency on. It is provided by the framework and is called a service. In a Blazor WebAssembly application, the services are defined in the `Program.Main` method of the `program.cs` file.

We have already used DI in this book with the following services:

- HttpClient
- IJSRuntime
- NavigationManager

## DI container

When a Blazor WebAssembly application starts, it configures a DI container. The DI container is responsible for building the instances of the service and lives until the user closes the tab in their browser that is running the web app. In the following example, the `CartService` implementation is registered for `IcartService`:

```
  builder.Services.AddSingleton<ICartService, CartService>();
```

After a service has been added to a DI container, we use the `@inject` directive to inject the service into any classes that depend on it. The `@inject` directive takes two parameters: type and property:

- **Type**: This is the type of service.
- **Property**: This is the name of the property that is receiving the service.

The following example shows how to use the `@inject` directive:

```
 @inject ICounterService counterService
```

Dependencies are injected after the component instance has been created, but before the `OnInitialized` or `OnInitializedAsync` life cycle events are executed. This means that you cannot use the injected class in the component's constructor, but you can use it in either the `OnInitialized` or `OnInitializedAsync` method.

## Service lifetime

The lifetime of a service that is injected using DI can be any of the following values:

- Singleton
- Scoped
- Transient

### Singleton

If the service lifetime is defined as `Singleton`, this means that a single instance of the class will be created and that instance will be shared throughout the application. Any components that use the service will receive an instance of the same service.

In a Blazor WebAssembly application, this is true for the lifetime of the current application that is running in the current tab of the browser. This is the service lifetime that we will use to manage the application's state in this chapter's project.

### Scoped

If the service lifetime of the service is defined as `Scoped`, this means that a new instance of the class will be created for each scope. Since a Blazor WebAssembly application does not have a concept of DI scopes, these services are treated like `Singleton` services.

In our project template, we are using a `Scoped` service to create the `HttpClient` instance that we are using for data access. This is because Microsoft's project templates use the scoped service lifetime for their services for symmetry with server-side Blazor.

### Transient

If the service lifetime of the service is defined as `Transient`, this means that a new instance of the class will be created every time an instance of the service is requested. When using transient services, the DI container simply acts as a factory that creates unique instances of the class. Once the instance is created and injected into the dependent component, the container has no further interest in it.

We can use DI to inject the same instance of a service into multiple components. It is used by the **AppState** pattern to allow the application to maintain state between components.

Now, let's get a quick overview of the project that we are going to build in this chapter.

## Project overview

In this chapter, we will build a Blazor WebAssembly app that includes a shopping cart. We will be able to add and remove different products from the shopping cart. The cart's total will be displayed on each of the pages in the app.

The following is a screenshot of the completed application:

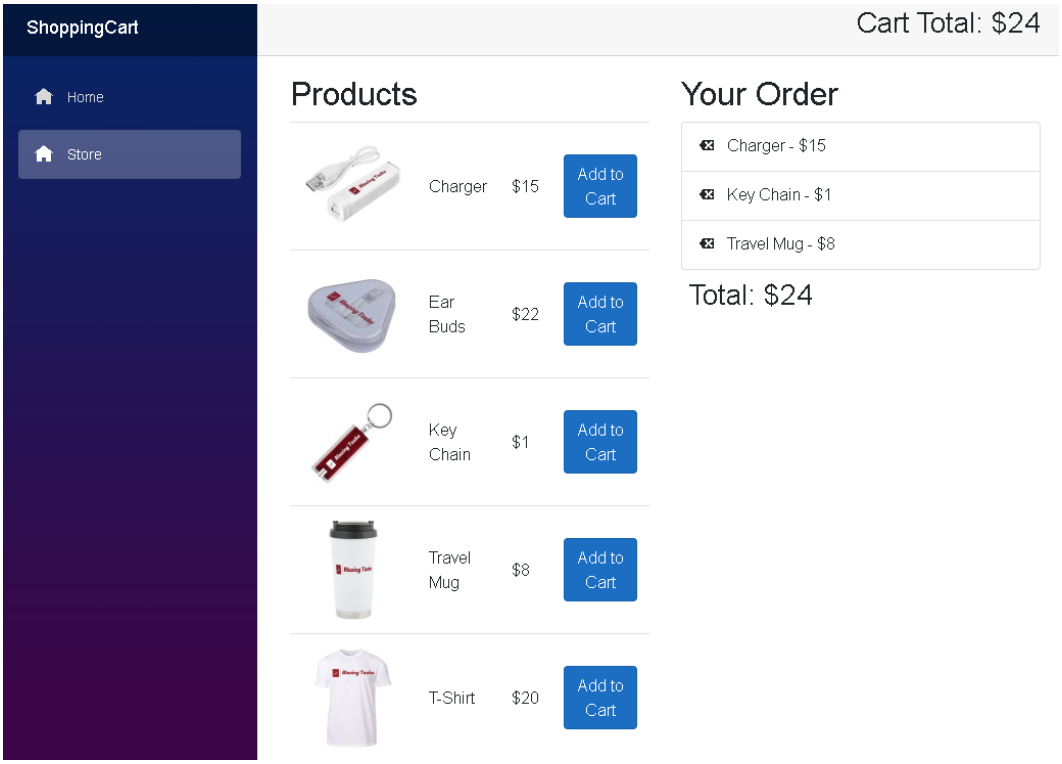



Figure 6.1 – ShoppingCart app

The build time for this project is approximately 60 minutes.

# Creating the shopping cart project

The `ShoppingCart` project will be created by using the **Empty Blazor WebAssembly App** project template. First, we will add logic to add and remove products from the shopping cart. Then, we will demonstrate that the cart's state is lost when we navigate between pages. To maintain the cart's state, we will register a service in the DI container that uses the **AppState** pattern. Finally, we will demonstrate that by injecting the new service into the relevant components, the cart's state is not lost.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt* + *S*) textbox, enter `blazor` and then hit the *Enter* key.

   The following screenshot shows the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*:

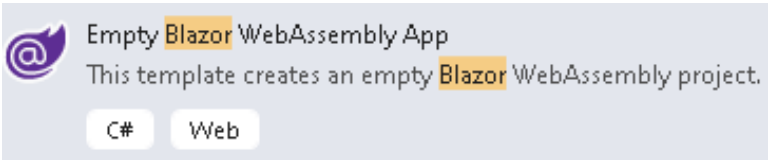



Figure 6.2 – Empty Blazor WebAssembly App project template

4. Select the **Empty Blazor WebAssembly App** project template and then click the **Next** button.
5. Enter `ShoppingCart` in the **Project name** textbox and then click the **Create** button:

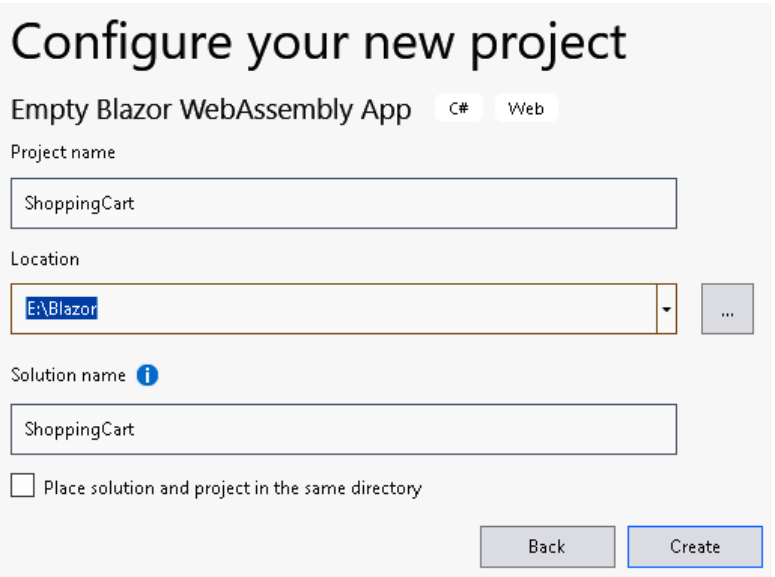



Figure 6.3 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `ShoppingCart` project into the `E:/Blazor` folder. However, the project's location is not important.

6. Open the `Pages\Index.razor` page.
7. Add the following markup:

```
<div class="jumbotron">
    <h1 class="display-4">Welcome to Blazing Tasks!</h1>
    <p class="lead">
        Your one stop shop for all your tasks.
    </p>
</div>
```

We have now created the `ShoppingCart` Blazor WebAssembly project.

## Adding the Product class

We need to add the products that are for sale. We do this as follows:

1. Right-click the `ShoppingCart` project and select the **Add, New Folder** option from the menu.
2. Name the new folder `Models`.
3. Right-click the `Models` folder and select the **Add, Class** option from the menu.
4. Name the new class `Product`.
5. Click the **Add** button.
6. Add the following properties to the `Product` class:

```
public int ProductId { get; set; }
public string ProductName { get; set; }
public int Price { get; set; }
public string Image { get; set; }
```

7. Right-click the `wwwroot` folder and select the **Add, New Folder** option from the menu.
8. Name the new folder `sample-data`.

9. Right-click the `sample-data` folder and select the **Add, New Item** option from the menu.
10. Enter `json` in the **Search** box.
11. Select **JSON File**.
12. Name the file `products.json`.
13. Click the **Add** button.
14. Update the file to the following:

**products.json**

```
[
  {
    "productId": 1,
    "productName": "Charger",
    "price": 15,
    "image": "charger.jpg"
  },
  {
    "productId": 2,
    "productName": "Ear Buds",
    "price": 22,
    "image": "earbuds.jpg"
  },
  {
    "productId": 3,
    "productName": "Key Chain",
    "price": 1,
    "image": "keychain.jpg"
  },
  {
    "productId": 4,
    "productName": "Travel Mug",
    "price": 8,
    "image": "travelmug.jpg"
  },
  {
```

```
    "productId": 5,
    "productName": "T-Shirt",
    "price": 20,
    "image": "tshirt.jpg"
  }
]
```

> **Important note**
> You can copy the `products.json` file from the GitHub repository.

15. Right-click the `wwwroot` folder and select the **Add, New Folder** option from the menu.
16. Name the new folder `images`.
17. Copy the following images from the GitHub repository to the `images` folder: `Charger.jpg`, `Earbuds.jpg`, `KeyChain.jpg`, `TravelMug.jpg`, and `Tshirt.jpg`.

We have added a collection of products to our web app. Next, we need to add a store.

## Adding the Store page

To add a store, we need to add a `Store` component to our web app. We do this as follows:

1. Open the `Shared\NavMenu.razor` page.
2. Add the following markup before the closing `ul` tag:

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="store">
        <span class="oi oi-home" aria-hidden="true">
        </span>
        Store
    </NavLink>
</li>
```

   The preceding markup adds a menu option for the **Store** page.
3. Right-click the `Pages` folder and select the **Add, Razor Component** option from the menu.

4. Name the new component `Store`.
5. Click the **Add** button.
6. Replace the markup with the following:

```
@page "/store"

@using ShoppingCart.Models
@inject HttpClient Http

@if (products == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <div class="row">

    </div>
}
@code {
    public IList<Product> products;
    public IList<Product> cart = new List<Product>();
    private int total;
}
```

The preceding code adds some directives and some properties.

7. Add the following markup in the `div` element:

```
<div class="col-xl-4 col-lg-6">
    <h2>Products</h2>
    <table class="table">
        @foreach (Product item in products)
        {
            <tr>
                <td>
                    <img src="images/@item.Image" />
                </td>
```

```
                    <td class="align-middle">
                        @item.ProductName
                    </td>
                    <td class="align-middle">
                        $@item.Price
                    </td>
                    <td class="align-middle">
                        <button class="btn btn-primary"
                        @onclick="@(() =>
                          AddProduct(item))">
                            Add to Cart
                        </button>
                    </td>
                </tr>
            }
        </table>
    </div>
```

The preceding markup adds a table that displays all of the products that are for sale.

8. Add the following markup below the preceding `div` element:

```
    <div class="col-xl-4 col-lg-6">
        @if (cart.Any())
        {
            <h2>Your Cart</h2>
            <ul class="list-group">
                @foreach (Product item in cart)
                {
                    <li class="list-group-item p-2">
                        <button class="btn btn-sm"
                                @onclick="@(()
                                 =>DeleteProduct(item))">
                            <span class="oi oi-delete">
                            </span>
                        </button>
                        @item.ProductName - $@item.Price
                    </li>
```

```
            }
        </ul>
        <div class="p-2">
            <h3>Total: $@total</h3>
        </div>
    }
</div>
```

The preceding markup displays all of the items in our list.

9. Add the following code to the `@code` block:

```
protected override async Task OnInitializedAsync()
{
    products = await Http.GetFromJsonAsync<Product[]>
            ("sample-data/products.json");
}
```

The preceding code reads the `products` from the `products.json` file using **HttpClient**.

10. Add the `AddProduct` method to the `@code` block:

```
private void AddProduct(Product product)
{
    cart.Add(product);
    total += product.Price;
}
```

The preceding code adds the indicated product to the cart and increments the total by the product's price.

11. Add the `DeleteProduct` method to the `@code` block:

```
private void DeleteProduct(Product product)
{
    cart.Remove(product);
    total -= product.Price;
}
```

The preceding code removes the indicated product from the cart and decrements the total by the product's price.

We have added a **Store** page to the web app. Now we need to test it.

## Demonstrating that application state is lost

We need to test the **Store** page. We do this as follows:

1. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.
2. Select the **Store** option on the navigation menu.
3. Add a few items to the cart.
4. Select the **Home** option on the navigation menu.
5. Return to the **Store** page by selecting the **Store** option on the navigation menu.
6. Confirm that the cart is now empty.

When we navigate between the pages in our web app, the state is lost. We can maintain the state by using the **AppState** pattern.

## Creating the ICartService interface

We need to create an `ICartService` interface. We do this as follows:

1. Return to **Visual Studio**.
2. Right-click the `ShoppingCart` project and select the **Add, New Folder** option from the menu.
3. Name the new folder `Services`.
4. Right-click the `Services` folder and select the **Add, New Item** option from the menu.
5. Enter `interface` in the **Search** box.
6. Select **Interface**.
7. Name the file `ICartService`.
8. Click the **Add** button.
9. Enter the following code:

```
IList<Product> Cart{ get; }
int Total { get; set; }
```

```
event Action OnChange;

void AddProduct(Product product);
void DeleteProduct(Product product);
```

10. Add the following `using` statement:

```
using ShoppingCart.Models;
```

We have created the `ICartService` interface. Now we need to create a class that inherits from it.

## Creating the CartService class

We need to create the `CartService` class. We do this as follows:

1. Right-click the `Services` folder and select the **Add, Class** option from the menu.
2. Name the class `CartService`.
3. Click the **Add** button.
4. Update the class to the following:

```
public class CartService : ICartService
{
    public IList<Product> Cart { get; private set; }
    public int Total { get; set; }

    public event Action OnChange;
}
```

The `CartService` class inherits from the `ICartService` interface.

5. Add the following `using` statement:

```
using ShoppingCart.Models;
```

6. Add the following constructor:

```
public CartService() { Cart = new List<Product>(); }
```

7. Add the `NotifyStateChanged` method to the class:

```
private void NotifyStateChanged() => OnChange?.Invoke();
```

In the preceding code, the `OnChange` event is invoked when the `NotifyStateChanged` method is called.

8. Add the `AddProduct` method to the class:

```
public void AddProduct(Product product)
{
    Cart.Add(product);
    Total += product.Price;
    NotifyStateChanged();
}
```

The preceding code adds the indicated product to the list of products and increments the total. It also calls the `NotifyStateChanged` method.

9. Add the `DeleteProduct` method to the class:

```
public void DeleteProduct(Product product)
{
    Cart.Remove(product);
    Total -= product.Price;
    NotifyStateChanged();
}
```

The preceding code removes the indicated product from the list of products and decrements the total. It also calls the `NotifyStateChanged` method.

We have completed the `CartService` class. Now we need to register `CartService` in the DI container.

## Registering CartService in the DI container

We need to register `CartService` in the DI container before we can inject it into our **Store** page. We do this as follows:

1. Open the `Program.cs` file.
2. Add the following code after the code that registers `HttpClient`:

```
builder.Services.AddScoped<ICartService, CartService>();
```

3. Add the following `using` statement:

```
using ShoppingCart.Services;
```

We have registered `CartService`. Now we need to update the **Store** page to use it.

## Injecting CartService

We need to update the **Store** page. We do this as follows:

1. Open the `Pages\Store.razor` page.
2. Add the following `@using` directive:

```
@using ShoppingCart.Services
```

3. Add the following `@inject` directive:

```
@inject ICartService cartService
```

4. Update the **Add to Cart** button to the following:

```
<button class="btn btn-primary"
        @onclick="@(() =>
          cartService.AddProduct(item))">
    Add to Cart
</button>
```

   The preceding markup uses `cartService` to add products to the cart.

5. Update the `cart div` element to the following:

```
@if (cartService.Cart.Any())
{
    <h2>Your Cart</h2>
    <ul class="list-group">
        @foreach (Product item in cartService.Cart)
        {
            <li class="list-group-item p-2">
                <button class="btn btn-sm"
                        @onclick="@(() =>cartService.
DeleteProduct(item))">
                    <span class="oi oi-delete"></span>
                </button>
                @item.ProductName - $@item.Price
            </li>
        }
```

```
        </ul>
        <div class="p-2">
            <h3>Total: $@cartService.Total</h3>
        </div>
    }
```

The preceding markup uses `CartService` to iterate through the products in the cart and to delete products from the cart.

6. Delete the `cart` property, the `AddProduct` method, and the `DeleteProduct` method from the `@code` block.
7. From the **Build** menu, select the **Build Solution** option.
8. Return to the browser.
9. Use *Ctrl* + *R* to refresh the browser.
10. Add a few items to the cart.
11. Select the **Home** option on the navigation menu.
12. Return to the **Store** page by selecting the **Store** option on the navigation menu.
13. Confirm that the cart is not empty.

We have confirmed that `CartService` is working. Now we need to add the cart total to all of the pages.

## Adding the cart total to all of the pages

To view the cart total on all of the pages, we need to add the cart total to a component that is used on all of the pages. Since the `MainLayout` component is used by all of the pages, we will add the cart total to it. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `Shared\MainLayout.razor` page.
3. Add the following `@using` directive:

```
@using ShoppingCart.Services
```

4. Add the following `@inject` directive:

```
@inject ICartService cartService
```

5. Add the following markup to `top-row div`:

```
<h3>Cart Total: $@cartService.Total</h3>
```

6. From the **Build** menu, select the **Build Solution** option.
7. Return to the browser.
8. Use *Ctrl* + *R* to refresh the browser.
9. Add a few items to the cart.
10. Confirm that the **Cart Total** field at the top of the page does not update.

The cart total at the top of the page is not being updated as we add new items to the cart. We need to deal with this.

## Using the OnChange method

We need to notify the component when it needs to be updated. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `Shared\MainLayout.razor` page.
3. Add the following `@implements` directive:

```
@implements IDisposable
```

4. Add the following `@code` block:

```
@code{
    protected override void OnInitialized()
    {
        cartService.OnChange += StateHasChanged;
    }

    public void Dispose()
    {
        cartService.OnChange -= StateHasChanged;
    }
}
```

In the preceding code, the component's `StateHasChanged` method is subscribed to the `cartService.OnChange` method in the `OnInitialized` method, and unsubscribed in the `Dispose` method.

5. From the **Build** menu, select the **Build Solution** option.
6. Return to the browser.
7. Use *Ctrl* + *R* to refresh the browser.
8. Add some items to the cart.
9. Confirm that the **Cart Total** field at the top of the page updates.

We have updated the component to call the `StateHasChanged` method whenever the `OnChange` method of `CartService` is invoked.

> **Tip**
> Do not forget to unsubscribe from the event when you dispose of the component.

You must unsubscribe from the event to prevent the `StateHasChanged` method from being invoked each time the `cartService.OnChange` event is raised. Otherwise, your application will experience resource leaks.

# Summary

You should now be able to use DI to apply the **AppState** pattern to a Blazor WebAssembly app.

In this chapter, we introduced application state and DI. After that, we used the **Empty Blazor WebAssembly App** project template to create a new project. We added a shopping cart to the project and demonstrated that application state is lost when we navigate between pages. To maintain the application's state, we registered the `CartService` service in the DI container. Finally, we demonstrated that by using the **AppState** pattern, we can maintain the shopping cart's state.

We can apply our new skills with DI to maintain the application state for any Blazor WebAssembly app.

In the next chapter, we will build a Kanban board using events.

# Questions

The following questions are provided for your consideration:

1. Can localStorage be used to maintain the cart's state when the page is reloaded?
2. Why don't we need to call the `StateHasChanged` method in the `Store` component?

# Further reading

The following resources provide more information concerning the topics covered in this chapter:

- For more information on DI, refer to `https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0`.
- For more information on events, refer to `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events`.

# 7

# Building a Kanban Board Using Events

As developers, we strive to make our applications as dynamic as possible. For that, we use **events**. Events are messages sent by an object to indicate that an action has occurred. Razor components can handle many different types of events.

In this chapter, we will learn how to handle different types of events in a Blazor WebAssembly app. We will also learn how to use both **arbitrary parameters** and **attribute splatting** to simplify how we assign attributes to components.

The project that we create in this chapter will be a Kanban board that uses the drag-and-drop events. Kanban boards visually depict work at various stages of a process. Our Kanban board will include three dropzones. Finally, we will use arbitrary parameters and attribute splatting to create an object to add new tasks to our Kanban board.

In this chapter, we will cover the following topics:

- Event handling
- Arbitrary parameters
- Attribute splatting
- Creating the Kanban board project

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free Community edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter07`.

The code in action video is available here: `https://bit.ly/3bHfPHt`

# Event handling

Razor components handle events by using an HTML element attribute named `@on{EVENT}` where `EVENT` is the name of the event.

The following code calls the `OnClickHandler` method when the button is clicked:

```
<button class="btn btn-success" @onclick="OnClickHandler">
    Click Me
</button>
@code {
    private void OnClickHandler()
    {
        // ...
    }
}
```

Since event handlers automatically trigger a UI render, we do not need to call `StateHasChanged` when processing them. Event handlers can be used to call both synchronous and asynchronous methods. Also, they can reference any arguments that are associated with the event.

The following code asynchronously calls the `OnChangeHandler` method when the checkbox is changed:

```
<input type="checkbox" @onchange="OnChangedHandler" />

@code {
    private async Task OnChangedHandler(ChangeEventArgs e)
    {
        newvalue = e.Value.ToString();
        // await ...
    }
}
```

In the preceding code, the `ChangeEventArgs` class is used to supply information about the change event. The event arguments are optional and should only be included if they are used by the method.

All of the `EventArgs` classes that are supported by the ASP.NET Core framework are supported by the Blazor WebAssembly framework. This is a list of the supported `EventArgs` classes:

- `ClipboardEventArgs`
- `DragEventArgs`
- `ErrorEventArgs`
- `EventArgs`
- `FocusEventArgs`
- `ChangeEventArgs`
- `KeyboardEventArgs`
- `MouseEventArgs`
- `PointerEventArgs`
- `WheelEventArgs`
- `ProgressEventArgs`
- `TouchEventArgs`

## Lambda expressions

When we need to include arguments with a method, we can use a **lambda expression**. Lambda expressions are used to create anonymous functions. They use the `=>` operator to separate the parameters from the body of the expression.

This is an example of an `@onclick` event that uses a lambda expression:

```
<button class="btn btn-info"
        @onclick="@(e => Console.WriteLine("Blazor
          Rocks!"))">
    Who Rocks?
</button>
```

In the preceding code, the `@onclick` event provides a string to the `Console.WriteLine` method.

> **Tip**
> If you use a loop variable directly in a lambda expression, the same variable will be used by all of the lambda expressions. Therefore, you should capture the loop variable's value in a local variable before using it in a lambda expression.

## Preventing default actions

Occasionally, we need to prevent the default action associated with an event. We can do that by using the `@on{EVENT}:preventDefault` directive, where `EVENT` is the name of the event.

For example, when dragging an element, the default behavior is that the user is not allowed to drop it into another element. In the Kanban board project, we will need to drop items into various dropzones. Therefore, we will need to prevent that default behavior.

The following code prevents the `ondragover` default behavior from occurring in order to allow us to drop elements into the `div` element:

```
<div class="dropzone"
     dropzone="true"
     ondragover="event.preventDefault();"
</div>
```

The Blazor WebAssembly framework makes it easy for us to access events by using the `@on{EVENT}` attribute. When working with components, we usually need to supply multiple attributes. Using attribute splatting, we can avoid assigning the attributes directly in the HTML markup.

# Attribute splatting

When a child component has many parameters, it can be tedious to assign each of the values in HTML. To avoid having to do that, we can use attribute splatting.

With attribute splatting, the attributes are captured in a dictionary and then passed to the component as a unit. One attribute is added per dictionary entry. The dictionary must implement `IEnumerable<KeyValuePair<string, object>>` or `IReadOnlyDictionary<string, object>` with string keys. We reference the dictionary using the `@attributes` directive.

This is the code for a component called `BweButton` that has quite a few parameters:

**BweButton.razor**

```
<button class="@Class" disabled="@Disabled" title="@Title" @
onclick="@ClickEvent">
    @ChildContent
</button>

@code {
    [Parameter] public string Class { get; set; }
    [Parameter] public bool Disabled { get; set; }
    [Parameter] public string Title { get; set; }
    [Parameter]
    public EventCallback ClickEvent { get; set; }
    [Parameter]
    public RenderFragment ChildContent { get; set; }
}
```

This is sample markup to render a `BweButton` component without using attribute splatting:

```
<BweButton Class="btn btn-danger"
           Disabled="false"
           Title="This is a button"
           ClickEvent="OnClickHandler">
    Submit
</BweButton>
```

This is the button that is rendered by the preceding markup:

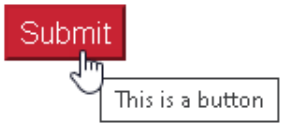



Figure 7.1 – Rendered BweButton

> **Tip**
>
> Some HTML attributes, such as `disabled` and `translate`, do not require any values. For these types of attributes, if the value is set to `false`, the attribute will not be included in the HTML output generated by the Blazor WebAssembly framework

By using attribute splatting, we can simplify the preceding markup to the following:

```
<BweButton @attributes="InputAttributes"
           ClickEvent="OnClickHandler">
    Submit
</BweButton >
```

This is the definition of `InputAttributes` used by the preceding markup:

```
public Dictionary<string, object> InputAttributes { get; set; }
    = new Dictionary<string, object>()
    {
        { "Class", "btn btn-danger" },
        { "Disabled", false},
        { "Title", "This is a button" }
    };
```

The preceding code defines `InputAttributes` that are passed to `BweButton`.

The real power of attribute splatting is realized when it is combined with arbitrary parameters.

# Arbitrary parameters

In the proceeding example, we used explicitly defined parameters to assign the button's attributes. A much more efficient way of assigning values to attributes is to use arbitrary parameters. An arbitrary parameter is a parameter that is not explicitly defined by the component. The `Parameter` attribute has a `CaptureUnmatchedValues` property that is used to capture any arbitrary parameters.

This is a new version of `BweButton` that uses arbitrary parameters:

```
<button @attributes="InputAttributes" >
    @ChildContent
</button>

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> InputAttributes {
      get; set; }
    [Parameter]
    public RenderFragment ChildContent { get; set; }
}
```

The preceding code includes a parameter named `InputAttributes` that has its `CaptureUnmatchedValues` property set to `true`.

> **Tip**
> A component can only have one parameter with `CaptureUnmatchedValues` set to `true`.

This is the updated markup used to render the new version of `BweButton`:

```
<BweButton @attributes="InputAttributes"
           @onclick="ButtonClicked"
           class="btn btn-info">
    Submit
</BweButton>
```

This is the definition of `InputAttributes` used by the preceding markup:

```
public Dictionary<string, object> InputAttributes { get; set; }
=
    new Dictionary<string, object>()
    {
        { "class", "btn btn-danger" },
        { "title", "This is another button" },
        { "name", "btnSubmit" },
        { "type", "button" },
          { "myAttribute", "123"}
    };
```

Although none of the attributes in the dictionary have been explicitly defined in the new version of `BweButton`, `BweButton` is still rendered.

This is the button that is rendered by the preceding markup:


Figure 7.2 – Rendered BweButton using arbitrary parameters

The reason the button is now teal is due to the position of the `@attributes` directive in the button's markup. When attributes are splatted onto an element, they are processed from left to right. Therefore, if there are duplicate attributes assigned, the one that appears later in the order will be the one that is used.

Arbitrary parameters are used to allow previously undefined attributes to be rendered by the component. This is useful with components that support a large variety of customizations, such as a component that includes an `input` element.

Now let's get a quick overview of the project that we are going to build in this chapter.

# Project overview

The Blazor WebAssembly application that we are going to build in this chapter is a Kanban board. The Kanban board will have three dropzones: **High Priority**, **Medium Priority**, and **Low Priority**. We will be able to drag and drop tasks between the dropzones and add additional tasks.

This is a screenshot of the completed application:

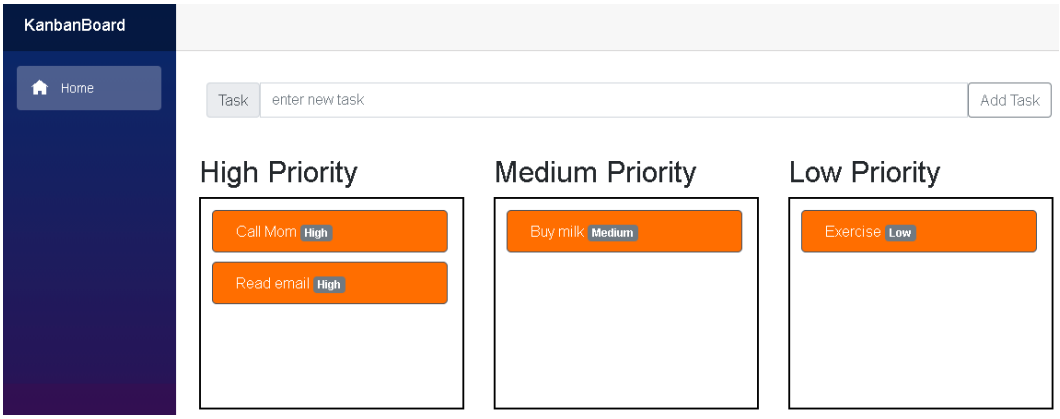



Figure 7.3 – Kanban Board app

The build time for this project is approximately 60 minutes.

# Creating the Kanban board project

The `KanbanBoard` project will be created by using the **Empty Blazor WebAssembly App** project template. First, we will add the `TaskItem` class. Then, we will add a `Dropzone` component. We will add three of the `Dropzone` components to the **Home** page to create the Kanban board. Finally, we will add the ability to add new tasks to the Kanban board.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt* + *S*) textbox, enter `Blazor` and hit the *Enter* key.

The following screenshot shows the **Empty Blazor WebAssembly App** project template that we created in *Chapter 2, Building Your First Blazor WebAssembly Application*:

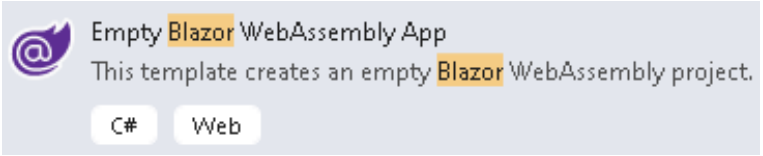



Figure 7.4 – Empty Blazor WebAssembly App project template

4. Select the **Empty Blazor WebAssembly App** project template and click the **Next** button.
5. Enter `KanbanBoard` in the **Project name** textbox and then click the **Create** button:

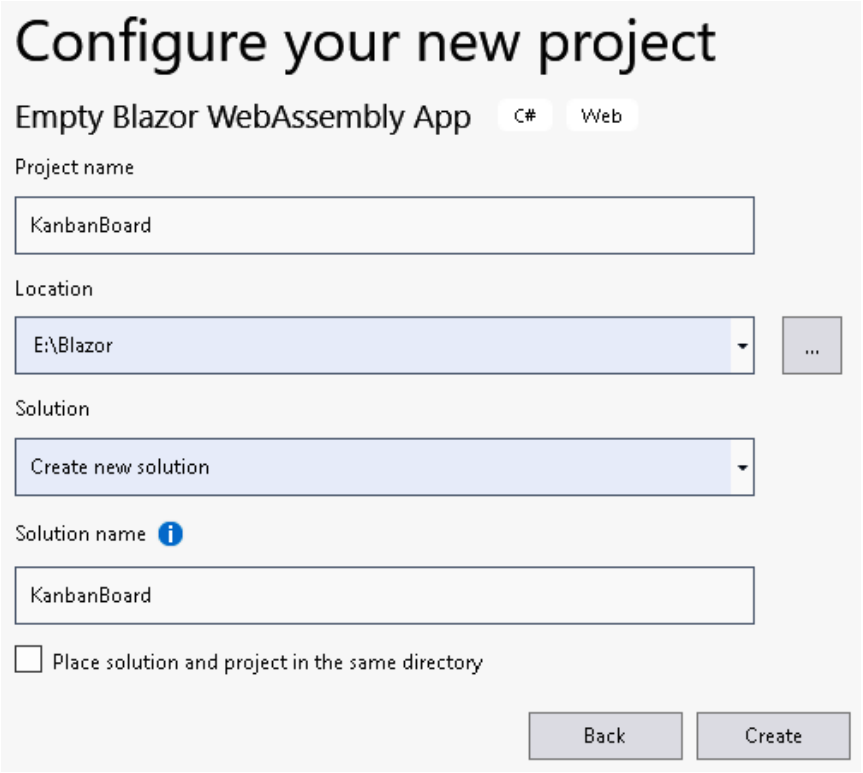



Figure 7.5 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `KanbanBoard` project into the `E:/Blazor` folder. However, the location of this project is not important.

We have now created the `KanbanBoard` Blazor WebAssembly project.

## Adding the classes

We need to add a `TaskPriority` enum and a `TaskItem` class. We do this as follows:

1. Right-click the `KanbanBoard` project and select the **Add, New Folder** option from the menu.
2. Name the new folder `Models`.
3. Right-click the `Models` folder and select the **Add, Class** option from the menu.
4. Name the new class `TaskPriority`.
5. Click the **Add** button.
6. Replace the class with the following enum:

```
public enum TaskPriority
{
    High,
    Medium,
    Low
}
```

7. Right-click the `Models` folder and select the **Add, Class** option from the menu.
8. Name the new class `TaskItem`.
9. Click the **Add** button.
10. Add the following properties to the `TaskItem` class:

```
public string TaskName { get; set; }
public TaskPriority Priority { get; set; }
```

We have added the `TaskPriority` enum and the `TaskItem` class to represent the tasks on the Kanban board. Next, we need to create the dropzones.

## Creating the Dropzone component

We need to add a `Dropzone` component. We do this as follows:

1. Right-click the `Shared` folder and select the **Add, Razor Component** option from the menu.
2. Name the new component `Dropzone`.
3. Click the **Add** button.
4. Remove the `h3` element.
5. Add the following `@using` directive:

```
@using KanbanBoard.Models
```

6. Add the following markup:

```
<div class="priority">
    <h2>@Priority.ToString() Priority</h2>
    <div class="dropzone"
         ondragover="event.preventDefault();"
         @ondrop="OnDropHandler">
        @foreach (var item in TaskItems
            .Where(q => q.Priority == Priority))
        {
        }
    </div>
</div>
```

The preceding markup labels `dropzone` by its priority and allows elements to be dropped into the element by preventing the default value of the `ondragover` event. The `OnDropHandler` method is called when an element is dropped into `dropzone`. Finally, it loops through all of the `TaskItems` class of the indicated `Priority`.

7. Add the following markup within the `@foreach` loop:

```
<div class="draggable"
     draggable="true"
     @ondragstart="@(() => OnDragStartHandler(item))">
    @item.TaskName
    <span class="badge badge-secondary">
```

```
        @item.Priority</span>
    </div>
```

The preceding markup makes the `div` element draggable by setting the `draggable` element to `true`. The `OnDragStartHandler` method is called when the element is dragged.

8. Add the following parameters to the `@code` block:

```
    [Parameter]
    public List<TaskItem> TaskItems { get; set; }
    [Parameter]
    public TaskPriority Priority { get; set; }
    [Parameter]
    public EventCallback<TaskPriority> OnDrop { get; set; }
    [Parameter]
    public EventCallback<TaskItem> OnStartDrag { get; set; }
```

9. Add the following `OnDropHandler` method:

```
    private void OnDropHandler()
    {
        OnDrop.InvokeAsync(Priority);
    }
```

The preceding code invokes the `OnDrop` method.

10. Add the following `OnDragStartHandler` method:

```
    private void OnDragStartHandler(TaskItem task)
    {
        OnStartDrag.InvokeAsync(task);
    }
```

The preceding code invokes the `OnStartDrag` method.

We have added a `Dropzone` component. Now we need to add some styling to the component.

## Adding a style sheet

We will add a style sheet to the `Dropzone` component using CSS isolation. We do this as follows:

1. Right-click the `Shared` folder and select the **Add, New Item** option from the menu.
2. Enter `css` in the **Search** box.
3. Select **Style Sheet**.
4. Name the style sheet `Dropzone.razor.css`.
5. Click the **Add** button.
6. Enter the following styles:

**Dropzone.razor.css**

```
.draggable {
    margin-bottom: 10px;
    padding: 10px 25px;
    border: 1px solid #424d5c;
    cursor: grab;
    background: #ff6a00;
    color: #ffffff;
    border-radius: 5px;
    width: 16rem;
}

    .draggable:active {
        cursor: grabbing;
    }

.dropzone {
    padding: .75rem;
    border: 2px solid black;
    min-height: 20rem;
}

.priority {
```

```
    min-width: 20rem;
    padding-right: 2rem;
}
```

We have finished styling the `Dropzone` component. Now we can put the Kanban board together.

## Creating the Kanban board

We need to add three dropzones to create our Kanban board, one dropzone for each of the three types of task. We do this as follows:

1. Open the `Pages\Index.razor` page.
2. Add the following `@using` directive:

```
@using KanbanBoard.Models
```

3. Add the following markup:

```
<div class="row p-2">
    <Dropzone Priority="TaskPriority.High"
              TaskItems="TaskItems"
              OnDrop="OnDrop"
              OnStartDrag="OnStartDrag" />

    <Dropzone Priority="TaskPriority.Medium"
              TaskItems="TaskItems"
              OnDrop="OnDrop"
              OnStartDrag="OnStartDrag" />

    <Dropzone Priority="TaskPriority.Low"
              TaskItems="TaskItems"
              OnDrop="OnDrop"
              OnStartDrag="OnStartDrag" />
</div>
```

4. Add the following `@code` block:

```
@code {
    public TaskItem CurrentItem;
    List<TaskItem> TaskItems = new List<TaskItem>();

    protected override void OnInitialized()
    {
        TaskItems.Add(new TaskItem
        {
            TaskName = "Call Mom",
            Priority = TaskPriority.High
        });
       TaskItems.Add(new TaskItem
       {
           TaskName = "Buy milk",
           Priority = TaskPriority.Medium
        });
        TaskItems.Add(new TaskItem
        {
            TaskName = "Exercise",
            Priority = TaskPriority.Low
        });
    }
}
```

The preceding code initializes the `TaskItems` object with three tasks.

5. Add the `OnStartDrag` method to the `@code` block:

```
private void OnStartDrag(TaskItem item)
{
    CurrentItem = item;
}
```

The preceding code sets the value of `CurrentItem` to the item that is currently being dragged. We will use this value when the item is dropped.

6. Add the `OnDrop` method to the `@code` block:

```
private void OnDrop(TaskPriority priority)
{
    CurrentItem.Priority = priority;
}
```

   The preceding code sets `Priority` of `CurrentItem` to the priority associated with the dropzone that it is dropped into.

7. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.
8. Drag and drop the tasks to change their priorities.

We have created a very simple Kanban board with three items. Let's add the ability to add more items.

## Creating the NewTask component

We need to add a `NewTask` component. We do this as follows:

1. Return to **Visual Studio**.
2. Right-click the `Shared` folder and select the **Add, Razor Component** option from the menu.
3. Name the new component `NewTask`.
4. Click the **Add** button.
5. Remove the `h3` element.
6. Add the following markup:

```
<div class="row p-3" style="max-width:950px">
    <div class="input-group mb-3">
        <label class="input-group-text"
          for="inputTask">
            Task
        </label>
        <input type="text"
               id="inputTask"
               class="form-control"
               @bind-value="@taskName"
```

```
                @attributes="InputParameters" />
        <button type="button"
                class="btn btn-outline-secondary"
                @onclick="OnClickHandler">
            Add Task
        </button>
    </div>
</div>
```

The preceding markup includes a label, a textbox, and a button.

This is a screenshot of the `NewTask` component:


Figure 7.6 – NewTask component

7. Add the following code to the `@code` block:

```
private string taskName;

[Parameter]
public EventCallback<string> OnSubmit { get; set; }
[Parameter(CaptureUnmatchedValues = true)]
public Dictionary<string, object>
    InputParameters{ get; set; }
```

8. Add the `OnClickHandler` method to the `@code` block:

```
private async Task OnClickHandler()
{
    if (!string.IsNullOrWhiteSpace(taskName))
    {
        await OnSubmit.InvokeAsync(taskName);
        taskName = null;
    }
}
```

The preceding code invokes the `OnSubmit` method and sets the `taskName` object to `null`.

We have now created the `NewTask` component. Next, we need to start using it.

## Using the NewTask component

We need to add the `NewTask` component to the **Home** page. We do this as follows:

1. Open the `Pages\Index.razor` page.
2. Add the following markup below the `@using` directive:

```
<NewTask OnSubmit="AddTask"
         @attributes="InputAttributes" />
```

3. Add the following code to the `@code` block:

```
public Dictionary<string, object> InputAttributes = new
Dictionary<string, object>()
{
    { "maxlength", "25" },
    { "placeholder", "enter new task" },
    { "title", "This textbox is used to enter your
      tasks." }
};
```

4. Add the `AddTask` method to the `@code` block:

```
private void AddTask(string taskName)
{
    var taskItem = new TaskItem()
    {
        TaskName = taskName,
        Priority = TaskPriority.High
    };
    TaskItems.Add(taskItem);
}
```

The preceding code sets the priority of the new item to `High` and adds it to the `TaskItems` object.

5. From the **Build** menu, select the **Build Solution** option.
6. Return to the browser.
7. Use *Ctrl* + *R* to refresh the browser.
8. Add a few new tasks.
9. Drag and drop the tasks to change their priorities.

We have added the ability to add new tasks to the Kanban board.

# Summary

You should now be able to handle events in your Blazor WebAssembly app. Also, you should be comfortable with using attribute splatting and arbitrary parameters.

In this chapter, we introduced event handling, attribute splatting, and arbitrary parameters. After that, we used the **Empty Blazor WebAssembly App** project template to create a new project. We added a `Dropzone` component to the application and used it to create a Kanban board. Finally, we added the ability to add tasks to the Kanban board while demonstrating both attribute splatting and arbitrary parameters.

Now that you know how to handle different types of events in your Blazor WebAssembly app, you can create more responsive applications. And, since you can use a dictionary to pass both explicitly declared attributes and implicit attributes to a component, you can create components faster since you do not need to explicitly define each parameter.

In the next chapter, we will use SQL Server to build a task manager using the ASP.NET Web API.

# Questions

The following questions are provided for your consideration:

1. How could you update the Kanban board to allow the user to delete a task?
2. Why would you want to include an attribute in the dictionary used for attribute splatting that is not defined on the component either explicitly or implicitly?
3. What is the base class of the `DragEventArgs` class?

# Further reading

The following resources provide more information concerning the topics covered in this chapter:

- For more information on ASP.NET Core Blazor event handling, refer to `https://docs.microsoft.com/en-us/aspnet/core/blazor/components/event-handling`.
- For more information on **Document Object Model** (**DOM**) events, refer to `https://developer.mozilla.org/en-US/docs/Web/Events`.
- For more information on the `DragEventArgs` class, refer to `https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.web.drageventargs`.

# 8

# Building a Task Manager Using ASP.NET Web API

Most websites are not islands standing alone. They need a server. They rely on a server for both data access and security, among other services.

In this chapter, we will learn how to create a hosted Blazor WebAssembly app. We will learn how to use the `HttpClient` service to call web APIs, and we will also learn how to use **JSON helper methods** to make requests in order to read, add, edit, and delete data.

The project that we create in this chapter will be a task manager. We will use a multi-project architecture to separate the Blazor WebAssembly app from the **ASP.NET Web API** endpoints. The hosted Blazor WebAssembly app will use JSON helper methods to read, add, edit, and delete tasks that are stored on SQL Server. An ASP.NET core project will provide the ASP.NET Web API endpoints.

In this chapter, we will cover the following topics:

- Understanding hosted applications
- Using the `HttpClient` service

- Using JSON helper methods
- Creating the TaskManager project

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free community edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need access to a version of SQL Server. For instructions on how to install the free edition of SQL Server 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter08`.

The code in action video is available here: `https://bit.ly/340Im6M`.

# Understanding hosted applications

When we create a new Blazor WebAssembly project by using Microsoft's **Blazor WebAssembly App** project template, we have the option to create a hosted Blazor WebAssembly app by checking the **ASP.NET Core hosted** checkbox.

The following screenshot highlights the **ASP.NET Core hosted** checkbox:

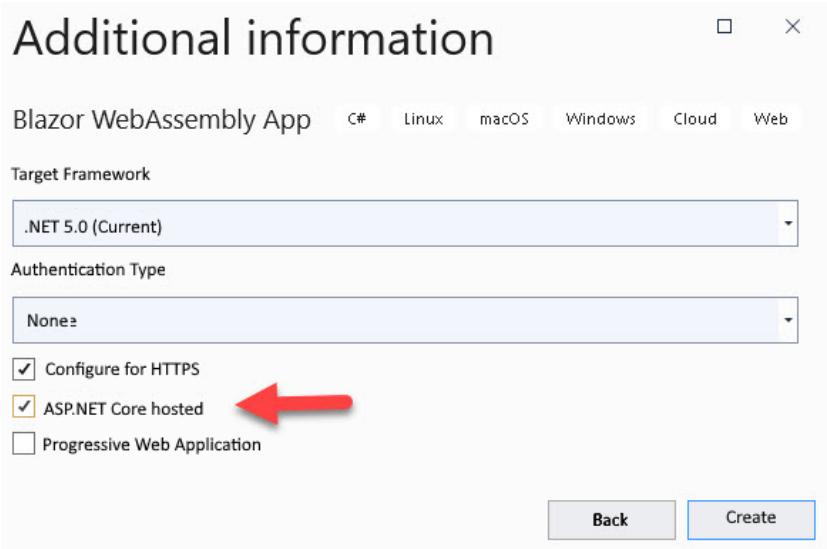



Figure 8.1 – Blazor WebAssembly App project template

The hosted Blazor WebAssembly app, created by the **Blazor WebAssembly App** project template, includes the following three projects:

- Client project
- Server project
- Shared project

## Client project

The client project is a client-side Blazor WebAssembly project. It is almost identical to the standalone Blazor WebAssembly app we created in *Chapter 2, Building Your First Blazor WebAssembly Application*, of this book. The only big difference is in how the data is accessed. In the client project, the sample data is accessed from the server project using Web API endpoints instead of a static file.

## Server project

The server project is an ASP.NET Core project. This project is responsible for serving the application. In addition to hosting the client app, the server project provides the Web API endpoints.

> **Tip**
> In this scenario, the server project must be set as the start up project in the solution.

## Shared project

The shared project is also an ASP.NET Core project. It contains application logic that is shared between the other two projects. In the past, we had to write validation code on both the client and the server. We had to write JavaScript validation code for the client and C# validation code for the server. Not surprisingly, sometimes the two validation models did not match. The shared project solves that problem since all of the validation code is kept in one location and one language.

By using a multi-project solution, we can create a more robust application. The shared project defines the classes, and the client project uses the `HttpClient` service to make requests for data from the server project.

# Using the HttpClient service

HTTP is not just for serving web pages – it can also be used for serving data. These are the HTTP methods that we will be using in this chapter:

- `GET`: This method is used to request one or more resources.
- `POST`: This method is used to create a new resource.
- `PUT`: This method is used to update a specified resource.
- `DELETE`: This method is used to delete a specified resource.

The `HttpClient` service is a preconfigured service for making HTTP requests from a Blazor WebAssembly app. It is configured in the `Program.cs` file. The following code is used to configure it:

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

The `HttpClient` service is added to a page using **dependency injection** (**DI**). To use the `HttpClient` service in a component, you must inject it by either using the `@inject` directive or the `Inject` attribute. For more information on DI, see *Chapter 6, Building a Shopping Cart Using Application State*.

The following code shows the two different ways to inject the `HttpClient` service into a component:

```
@inject HttpClient Http
[Inject] public HttpClient Http { get; set; }
```

After we have injected an `HttpClient` service into a component, we can use the JSON helper methods to send requests to a Web API.

# Using JSON helper methods

There are three JSON helper methods. There is one for reading data, one for adding data, and one for updating data. Since there is not one for deleting data, we will use the `HttpClient.DeleteAsync` method to delete data:

| JSON Helper Method | HTTP Method | Action |
|---|---|---|
| `GetFromJsonAsync` | `GET` | *Read* |
| `PostAsJsonAsync` | `POST` | *Create* |
| `PutAsJsonAsync` | `PUT` | *Update* |
| `HttpClient.DeleteAsync` | `DELETE` | *Delete* |

Figure 8.2 – Relationship between the HTTP methods and the JSON helper methods

The preceding table indicates the relationship between the JSON helper methods and the HTTP methods.

> **Tip**
>
> You can also use the `HttpClient` service and JSON helper methods to call external web API endpoints. By way of an example, see *Chapter 5, Building a Weather App as a Progressive Web App (PWA)*.

## GetFromJsonAsync

The `GetFromJsonAsync` method is used to read data. It does the following:

- Sends an `HTTP GET` request to the indicated URI.
- Deserializes the JSON response body to create the indicated object.

The following code returns a collection of `TaskItem` objects:

```
string requestUri = "TaskItems";
tasks = await
    Http.GetFromJsonAsync<IList<TaskItem>>(requestUri);
```

In the proceeding code, the type of object returned is `IList<TaskItem>`. We can also use the `GetFromJsonAsync` method to get an individual object. The following code returns a single `TaskItem` object where `Id` is the unique identifier of the object:

```
string requestUri = $"TaskItems/{Id}";
tasks = await
    Http.GetFromJsonAsync<TaskItem>(requestUri);
```

In the preceding code, the type of object returned is `TaskItem`.

## PostAsJsonAsync

The `PostAsJsonAsync` method is used to add data. It does the following:

- Sends an `HTTP POST` request to the indicated URI. The request includes the JSON-encoded content used to create the new data.
- Returns an `HttpResponseMessage` instance that includes both a status code and data.

The following code creates a new `TaskItem` object:

```
string requestUri = "TaskItems";
var response = await
    Http.PostAsJsonAsync(requestUri, newTaskItem);
if (response.IsSuccessStatusCode)
{
    var task = await
        response.Content.ReadFromJsonAsync<TaskItem>();
};
```

In the preceding code, `task` is deserialized from the response if the HTTP response is successful.

## PutAsJsonAsync

The `PutAsJsonAsync` method is used to update data. It does the following:

- Sends an `HTTP PUT` request to the indicated URI. The request includes the JSON-encoded content used to update the data.
- Returns an `HttpResponseMessage` instance that includes both a status code and data.

The following code updates an existing `TaskItem` object:

```
string requestUri = $"TaskItems/{task.TaskItemId}";
var response = await
   Http.PutAsJsonAsync<TaskItem>(requestUri, updatedTaskItem);
if (response.IsSuccessStatusCode)
{
    var task = await
        response.Content.ReadFromJsonAsync<TaskItem>();
};
```

In the preceding code, `task` is deserialized from the response if the HTTP response is successful.

## HttpClient.DeleteAsync

The `HttpClient.DeleteAsync` method is used to delete data. It does the following:

- Sends an `HTTP DELETE` request to the indicated URI.
- Returns an `HttpResponseMessage` instance that includes both a status code and data.

The following code deletes an existing `TaskItem` object:

```
string requestUri = $"TaskItems/{taskItem.TaskItemId}";
var response = await Http.DeleteAsync(requestUri);
if (!response.IsSuccessStatusCode)
{
    // handle error
};
```

The JSON helper methods make it easy to consume web APIs. We use them to read, create, and update data. We use `HttpClient.DeleteAsync` to delete data.

Now, let's get a quick overview of the project that we are going to build in this chapter.

# Project overview

In this chapter, we will build a Blazor WebAssembly app to manage tasks. We will be able to view, add, edit, and delete tasks. The tasks will be stored in a SQL Server database.

This is a screenshot of the completed application:

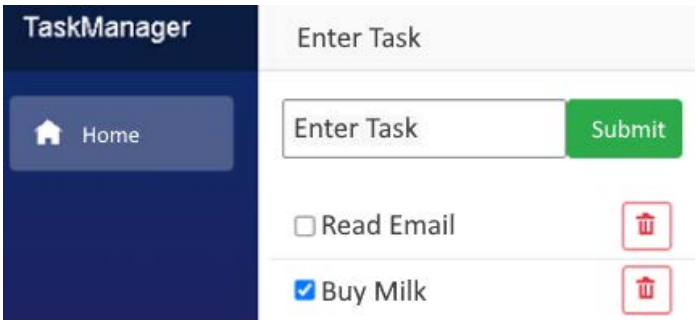



Figure 8.3 – TaskManager project

The build time for this project is approximately 75 minutes.

# Creating the TaskManager project

The `TaskManager` project will be created by using Microsoft's **Blazor WebAssembly App** project template to create a hosted Blazor WebAssembly app. First, we will examine the demo project created by the project template. Then, we will add a `TaskItem` class and a `TaskItemsController` class. We will use Entity Framework migrations to create a database in SQL Server. Finally, we will demonstrate how to read data, update data, delete data, and add data using the `HttpClient` service.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt*+*S*) textbox, enter `Blazor` and then hit the *Enter* key.

The following screenshot shows the **Blazor WebAssembly App** project template that we will be using:

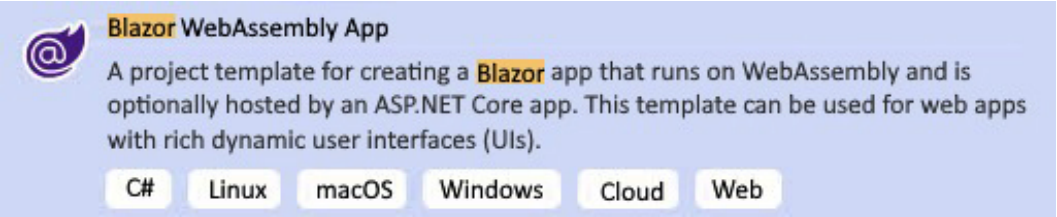



Figure 8.4 – Blazor WebAssembly App project template

4. Select the **Blazor WebAssembly App** project template and click the **Next** button.
5. Enter `TaskManager` in the **Project name** textbox and then click the **Next** button.

   This is a screenshot of the dialog used to configure our new project:

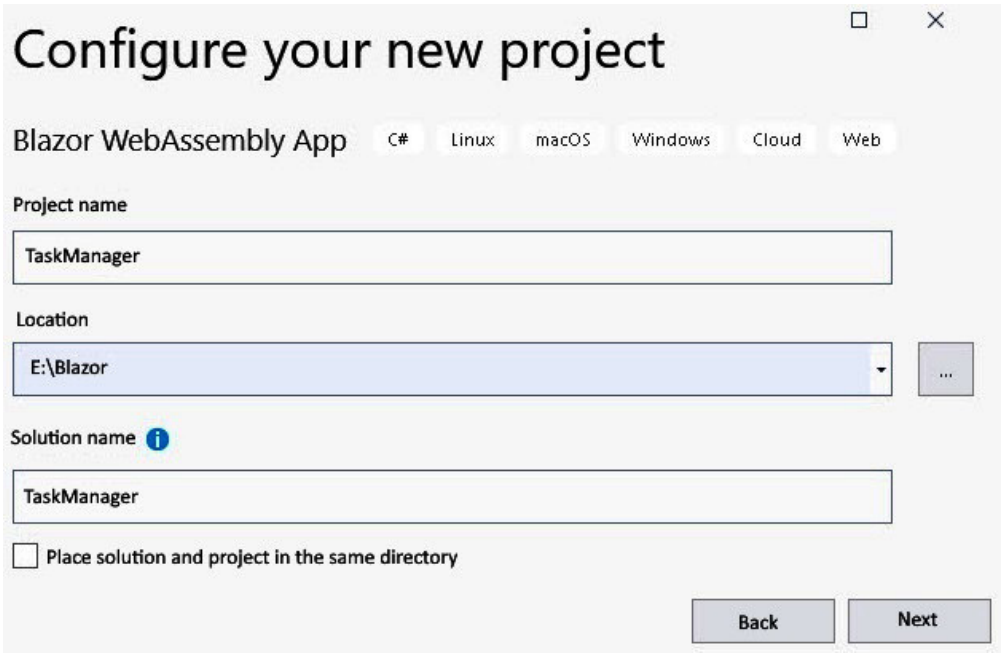



Figure 8.5 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `TaskManager` project into the `E:/Blazor` folder. However, the location of this project is not important.

6. Select **.NET 5.0** as the **Target Framework**.
7. Check the **ASP.NET Core Hosted** checkbox.
8. Click the **Create** button.
9. Right-click the `TaskManager.Server` project and select the **Set as Startup Project** option from the menu.

You have created the TaskManager project. The following screenshot shows the three projects that comprise the TaskManager project:

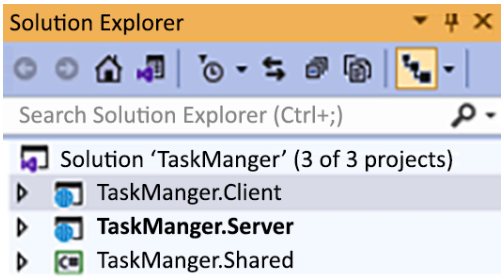



Figure 8.6 – Solution Explorer

Now we are all set to examine the hosted Blazor WebAssembly app.

## Examining the hosted Blazor WebAssembly app

The hosted Blazor WebAssembly app includes a demo project. Let's run it to get an idea of what it does. We do this as follows:

1. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run the project.
2. View the **Home** page.
3. View the **Counter** page.
4. View the **Fetch data** page.
5. Close the browser.

Before we can get started working on our **TaskManager** project, we need to remove the demo project.

## Emptying the solution

To empty the solution, we need to delete some components, update a couple of components, and delete both a controller and a class. We do this as follows:

1. Return to **Visual Studio**.
2. Delete all of the components in the `TaskManager.Client.Pages` folder, except for `Index`.
3. Delete the `TaskManager.Client.Shared\SurveyPrompt.razor` file.
4. Open the `TaskManager.Client.Shared\MainLayout.razor` file.
5. Remove the `About` link from the top row of the layout by removing the following markup:

```
<a href="http://blazor.net" target="_blank"
   class="ml-md-auto">
        About
</a>
```

6. Open the `TaskManager.Client.Shared\NavMenu.razor` file.
7. Remove the `li` elements for the `Counter` and `Fetch data` pages.
8. From the `TaskManager.Server` project, delete the `Controllers\WeatherForecastConroller.cs` file.
9. From the `TaskManager.Shared` project, delete the `WeatherForecast.cs` file.
10. From the **Build** menu, select the **Build Solution** option.

We have prepared the solution by deleting the demo project. Now, we can start adding our `TaskManager`-specific content. First, we will add a class to contain the tasks.

## Adding the TaskItem class

We need to add the `TaskItem` class. We do this as follows:

1. Right-click the `TaskManager.Shared` project and select the **Add, Class** option from the menu.
2. Name the new class `TaskItem`.
3. Click the **Add** button.

4. Make the class public by adding the `public` modifier:

```
public class TaskItem
```

5. Add the following properties to the `TaskItem` class:

```
public int TaskItemId { get; set; }
public string TaskName { get; set; }
public bool IsComplete { get; set; }
```

6. From the **Build** menu, select the **Build Solution** option.

We have added the `TaskItem` class. Next, we need to add an **API controller** for the `TaskItem` class.

## Adding the TaskItem API controller

We need to add a `TaskItemsController` class. We do this as follows:

1. Right-click the `TaskManager.Server.Contollers` folder and select the **Add, Controller** option from the menu.
2. Select the **API Controller with actions, using Entity Framework** option:

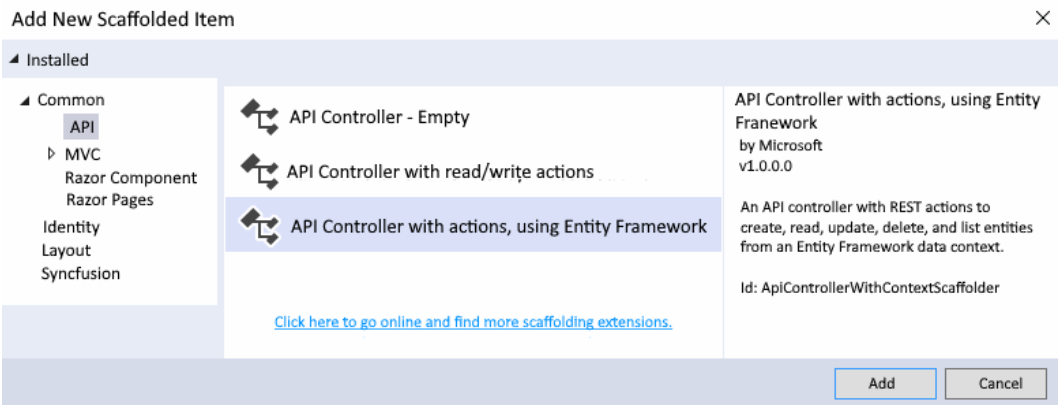



Figure 8.7 – Add New Scaffolded Item dialog

3. Click the **Add** button.
4. Set the **Model** class to `TaskItem (TaskManager.Shared)`.
5. Click the **Add data context** button to open the **Add Data Context** dialog:

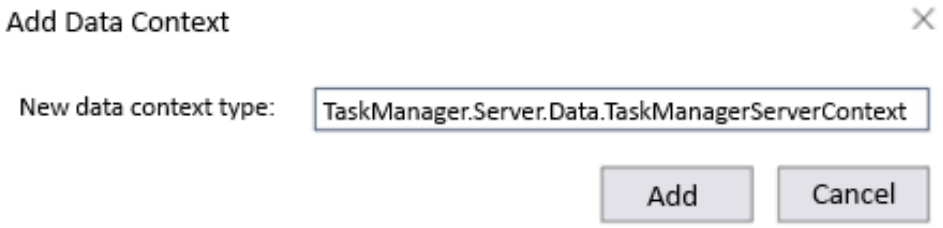



Figure 8.8 – Add Data Context dialog

6. Click the **Add** button to accept the default values.
7. Click the **Add** button on the **Add API Controller with actions, using Entity Framework** dialog:

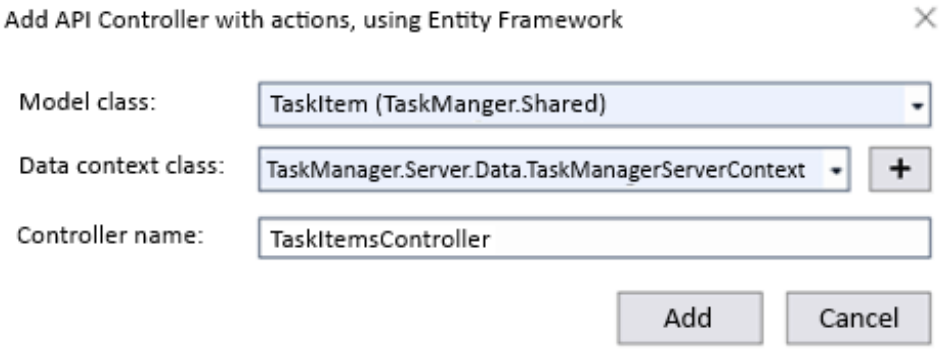



Figure 8.9 – Add API Controller with actions, using Entity Framework dialog

8. Update the route to the following:

```
[Route("[controller]")]
```

We have created the `TaskItemsController` class. Now we need to set up SQL Server.

## Setting up SQL Server

We need to create a new database on SQL Server and add a table to contain the tasks. We do this as follows:

1. Open the `TaskManager.Server\appsettings.json` file.
2. Update the connection string to point to your instance of SQL Server and change the name of the database to `TaskManager`:

```
  "ConnectionStrings": {
    "TaskManagerServerContext": "Server=TOI-WORK\\
  SQLEXPRESS2019; Database=TaskManager; Trusted_
  Connection=True; MultipleActiveResultSets=true"
  }
```

   The preceding code assumes that our server is named `TOI-WORK\\SQLEXPRESS2019` and the name of the database is `TaskManager`.

> **Important note**
>
> Although this example is using SQL Server Express 2019, it does not matter what version of SQL Server you use.

3. From the **Tools** menu, select the **NuGet Package Manager, Package Manager Console** option.
4. In **Package Manager Console**, change **Default project** to `TaskManager.Server`.
5. Execute the following commands in **Package Manager Console**:

```
Add-Migration Init
Update-Database
```

   The preceding commands use **Entity Framework** migrations to update SQL Server.

6. From the **View** menu, select **SQL Server Object Explorer**.
7. If you do not see the SQL Server instance that you are using for this project, click the **Add SQL Server** button to connect it:

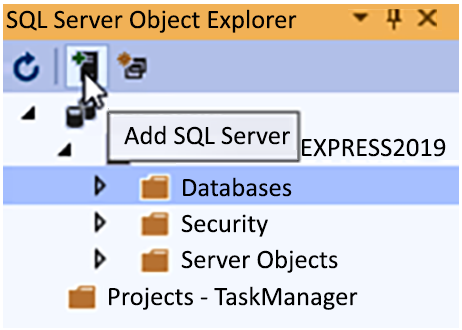



Figure 8.10 – SQL Server Object Explorer

8. Navigate to `TaskManager`, `Tables`, `dbo.TaskItem`:

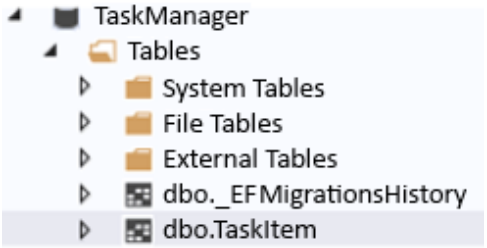



Figure 8.11 – TaskManager database

9. Right-click `dbo.TaskItem` and select the **View Data** option.
10. Pin the tab.
11. Enter a couple of tasks by completing the `TaskName` field and setting the `IsComplete` field to `False`:

| | TaskItemId | TaskName | IsComplete |
|---|---|---|---|
| | 1 | Buy Milk | False |
| ▶ | 2 | Read Email | *False* |
| * | NULL | NULL | NULL |

Figure 8.12 – Sample data

12. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run the project.
13. Add `/taskitems` to the address bar and then click *Enter*.

The following screenshot shows the JSON that is returned by `TaskItemsController`:

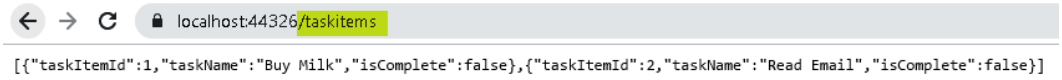



Figure 8.13 – JSON returned by the TaskItem API controller

14. Close the browser.

We have demonstrated that `TaskItemsController` works. Now we can start working on our client project.

## Displaying the tasks

We need to fetch the list of tasks and display them to the user. We do this as follows:

1. Return to **Visual Studio**.
2. Right-click the `TaskManager.Client.Pages` folder and select the **Add, Class** option from the menu.
3. Name the new class `Index.razor.cs`.
4. Click the **Add** button.
5. Add the `partial` modifier to the class:

```
public partial class Index
```

6. Add the following code to the `Index` class:

```
[Inject] public HttpClient Http { get; set; }
private IList<TaskItem> tasks;
private string error;

protected override async Task OnInitializedAsync()
{
    try
    {
        string requestUri = "TaskItems";
        tasks = await
            Http.GetFromJsonAsync<IList<TaskItem>>
           (requestUri);
```

```
        }
        catch (Exception)
        {
            error = "Error Encountered";
        };
    }
```

The preceding code uses the `GetFromJsonAsync` method to return the collection of `TaskItem` objects.

7. Add the following `using` statements:

```
using Microsoft.AspNetCore.Components;
using System.Net.Http;
using System.Net.Http.Json;
using TaskManager.Shared;
```

8. Open the `TaskManager.Client.Pages\Index.razor` page.
9. Update the markup to the following:

```
@page "/"

@if (tasks == null)
{
    <p><em>Loading...</em></p>
}
else
{
    @foreach (var taskItem in tasks)
    {

    }
}
```

The preceding markup displays the loading message if `tasks` is `null`. Otherwise, it loops through the collection of `TaskItem` objects in `tasks`.

10. Add the following markup to the `@foreach` loop:

```
<div class="d-flex col col-lg-3 border-bottom"
        @key="taskItem">
```

```
        <div class="p-2 flex-fill">
            <input type="checkbox"
                   checked="@taskItem.IsComplete" />
            <span>@taskItem.TaskName</span>
        </div>
        <div class="p-1">
            <button class="btn btn-outline-danger btn-sm"
                    title="Delete task">
                <span class="oi oi-trash"></span>
            </button>
        </div>
    </div>
```

The preceding markup displays a checkbox, the `TaskName` field, and a delete button for each `TaskItem` class.

11. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl+F5*) option to run the project.

    The following is a screenshot of the **Home** page:


Figure 8.14 – List of tasks

We have added a list of tasks to the **Home** page, but nothing happens when we click the checkbox or the delete button. Next, we need to allow the user to mark a task as complete.

## Completing the tasks

We will allow the user to mark a task as complete by clicking the checkbox next to the name of the task. We do this as follows:

1. Return to **Visual Studio**.
2. Right-click the `Pages` folder and select the **Add, New Item** option from the menu.
3. Enter `css` in the **Search** box.
4. Select **Style Sheet**.
5. Name the file `Index.razor.css`.
6. Click the **Add** button.

7. Enter the following style:

```
.completed-task {
    text-decoration: line-through;
}
```

8. Open the `Index.razor` file.
9. Update the `span` element used to display the task's name to the following:

```
<span class="@((taskItem.IsComplete? "completed-task" :
"")) ">
    @taskItem.TaskName
</span>
```

The preceding markup will set the class of the `span` element to `completed-task` when the task is completed.

10. Add the following markup to the checkbox:

```
@onchange="@(()=>CheckboxChecked(taskItem))"
```

11. Open the `TaskManager.Client.Pages\Index.razor.cs` file.
12. Add the following `CheckboxChecked` method:

```
private async Task CheckboxChecked(TaskItem task)
{
    task.IsComplete = !task.IsComplete;

    string requestUri =
      $"TaskItems/{task.TaskItemId}";
    var response = await
        Http.PutAsJsonAsync<TaskItem>(requestUri,
          task);
    if (!response.IsSuccessStatusCode)
    {
        error = response.ReasonPhrase;
    };
}
```

The preceding code uses the `PutAsJsonAsync` method to update the indicated `TaskItem` class.

13. From the **Build** menu, select the **Build Solution** option.
14. Return to the browser.
15. Use *Ctrl+R* to refresh the browser.
16. Mark one of the tasks as complete by clicking the checkbox next to it.

    The following screenshot shows a task that has been completed:


Figure 8.15 – Completed task

17. Return to **Visual Studio**.
18. Select the **dbo.TaskItem [Data]** tab that we pinned earlier.
19. Click the **Refresh** (*Shift+Alt+R*) button to verify that the `IsComplete` field has been updated.

When a user checks the checkbox next to a task, the UI is updated and the SQL Server database is updated. Next, we need to add the ability to delete tasks.

## Deleting the tasks

We need to allow users to delete tasks. We do this as follows:

1. Open the `Index.razor` file.
2. Update the `button` element to the following by adding the highlighted code:

```
<button class="btn btn-outline-danger btn-sm"
        title="Delete task"
        @onclick="@(()=>DeleteTask(taskItem))">
    <span class="oi oi-trash"></span>
</button>
```

3. Open the `TaskManager.Client.Pages\Index.razor.cs` file.
4. Add the following `DeleteTask` method:

```
private async Task DeleteTask(TaskItem taskItem)
{
    tasks.Remove(taskItem);
    string requestUri =
        $"TaskItems/{taskItem.TaskItemId}";
```

```
        var response = await Http.DeleteAsync(requestUri);
        if (!response.IsSuccessStatusCode)
        {
            error = response.ReasonPhrase;
        };
    }
```

The preceding code uses the `Http.DeleteAsync` method to delete the indicated `TaskItem` class.

5. From the **Build** menu, select the **Build Solution** option.
6. Return to the browser.
7. Use *Ctrl+R* to refresh the browser.
8. Click the delete button to delete one of the tasks.
9. Return to **Visual Studio**.
10. Select the **dbo.TaskItem [Data]** tab.
11. Click the **Refresh** (*Shift+Alt+R*) button to verify that one of the tasks has been deleted.

We have added the ability to delete tasks. Now we need to add the ability to add new tasks.

## Adding new tasks

We need to add the ability to add new tasks. We do this as follows:

1. Open the `Index.razor` file.
2. Add the following markup before the `@foreach` loop:

```
<div class="d-flex col col-lg-3 mb-4">
    <input placeholder="Enter Task" @bind="newTask" />
    <button class="btn btn-success"
            @onclick="AddTask">Submit</button>
</div>
```

3. Open the `TaskManager.Client.Pages\Index.razor.cs` file.

4. Add the following variable:

```
private string newTask;
```

5. Add the following `AddTask` method:

```
private async Task AddTask()
{
    if (!string.IsNullOrWhiteSpace(newTask))
    {
        TaskItem newTaskItem = new TaskItem
        {
            TaskName = newTask,
            IsComplete = false
        };
        tasks.Add(newTaskItem);

        string requestUri = "TaskItems";
        var response = await
           Http.PostAsJsonAsync(requestUri,
             newTaskItem);
        if (response.IsSuccessStatusCode)
        {
            newTask = string.Empty;
            var task =
              await response.Content.ReadFromJsonAsync
                <TaskItem>();
        }
        else
        {
            error = response.ReasonPhrase;
        };
    };
}
```

The preceding code uses the `PostAsJsonAsync` method to create a new `TaskItem` class. The returned `TaskItem` class is deserialized into the `task` variable.

> **Tip**
> If you need the `Id` property of the new `TaskItem` class , you can obtain it from the `task` variable.

6. From the **Build** menu, select the **Build Solution** option.
7. Return to the browser.
8. Use *Ctrl+R* to refresh the browser.
9. Add a few new tasks.
10. Return to **Visual Studio**.
11. Select the **dbo.TaskItem [Data]** tab.
12. Click the **Refresh** (*Shift+Alt+R*) button to verify that the tasks have been added to the SQL Server database.

We have now added the ability to add new tasks.

# Summary

You should now be able to create a hosted Blazor WebAssembly app that uses the ASP.NET Web API to update data in a SQL Server database.

In this chapter, we introduced hosted Blazor WebAssembly apps, the `HttpClient` service, and the JSON helper methods used to read, create, and update data. We also demonstrated how to delete data using the `HttpClient.DeleteAsync` method.

After that, we used Microsoft's **Blazor WebAssembly App** project template to create a hosted Blazor WebAssembly app. We examined the demo project and then deleted it from the multi-project solution. We added both a `TaskItem` class and a `TaskItem` API controller. Next, we configured SQL Server by updating the connection string to the database and using Entity Framework migrations. Finally, we used the `HttpClient` service to read the list of tasks, update a task, delete a task, and add new tasks.

We can apply our new skills to create a hosted Blazor WebAssembly app that is part of a multi-project solution and use the ASP.NET Web API to read, create, update, and delete data.

In the next chapter, we will build an expense tracker using the `EditForm` component.

# Questions

The following questions are provided for your consideration:

1. What are the benefits of using a hosted Blazor WebAssembly project versus a standalone Blazor WebAssembly project?
2. Is it better to use an `@code` block for the code or to use a partial class for the code?

# Further reading

The following resources provide more information concerning the topics covered in this chapter:

- For more information on the `HttpClient` class, refer to `https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient`.
- For more information on calling a web API from Blazor WebAssembly, refer to `https://docs.microsoft.com/en-us/aspnet/core/blazor/call-web-api`.
- For more information on the extension methods that perform serialization and deserialization using `System.Text.Json`, refer to `https://docs.microsoft.com/en-us/dotnet/api/system.net.http.json`.
- For more information on Entity Framework, refer to `https://docs.microsoft.com/ef`.

# 9

# Building an Expense Tracker Using the EditForm Component

Most applications require some data input. The Blazor WebAssembly framework includes many built-in components for inputting and validating data.

In this chapter, we will learn how to use the `EditForm` component, the various built-in input components, and the built-in input validation components.

The project that we'll create in this chapter will be a travel expense tracker. We will use a multi-project architecture to separate the Blazor WebAssembly app from the **ASP.NET Web API** endpoints. The page used to add and edit expenses will use the `EditForm` component as well as many of the built-in input components. It will also use the built-in validation components. We will learn how to use the built-in components to add data input, validation, and submission to any Blazor WebAssembly app.

In this chapter, we will cover the following topics:

- The `EditForm` component
- Using the built-in input components
- Using the validation components
- Creating the `ExpenseTracker` project

# Technical requirements

To complete this project, you need to have Visual Studio 2019 installed on your PC. For instructions on how to install the free community edition of Visual Studio 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*. You will also need access to a version of SQL Server. For instructions on how to install the free edition of SQL Server 2019, refer to *Chapter 1, Introduction to Blazor WebAssembly*.

The source code for this chapter is available in the following GitHub repository: `https://github.com/PacktPublishing/Blazor-WebAssembly-by-Example/tree/main/Chapter09`.

The code in action video is available here: `https://bit.ly/2T5UfpR`.

# Overview of the EditForm component

In the previous chapters of this book, we used the standard HTML `form` element to collect user input. However, the Blazor WebAssembly framework provides an enhanced version of the standard HTML `form` element called the `EditForm` component.

The `EditForm` component not only manages forms, it also coordinates both validation and submission events. The following code shows an empty `EditForm` element:

```
<EditForm Model="expense" OnValidSubmit="HandleValidSubmit">
</EditForm>
```

In the preceding code, the `Model` property specifies the top-level model object for the form. The `OnValidSubmit` property specifies the callback that will be invoked when the form is submitted without any validation errors.

There are three different callbacks that are associated with form submission:

- `OnValidSubmit`
- `OnInvalidSubmit`
- `OnSubmit`

We can use the `OnValidSubmit` and `OnInvalidSubmit` callbacks together or separately. Or, we can use the `OnSubmit` callback by itself. If we use the `OnSubmit` callback, we are responsible for performing the form validation. Otherwise, the form validation is performed by the `EditForm` component.

> **Tip**
> If we set an `OnSubmit` callback, any callbacks set using `OnValidSubmit` or `OnInvalidSubmit` are ignored.

There are quite a few built-in input components that we can use in conjunction with the `EditForm` component.

# Using the built-in input components

The following table lists the built-in input components along with the HTML that they render:

| Input Component | HTML Rendered |
|---|---|
| `InputCheckbox` | `<input type="checkbox">` |
| `InputDate<TValue>` | `<input type="date">` |
| `InputFile` | `<input type="file">` |
| `InputNumber<TValue>` | `<input type="number">` |
| `InputRadio` | `<input type="radio">` |
| `InputRadioGroup` | `<input type="radio">` |
| `InputSelect<TValue>` | `<select>` |
| `InputText` | `<input>` |
| `InputTextArea` | `<textarea>` |

Figure 9.1 – Built-in input components

All of the built-in input components are able to receive and validate user inputs when placed within an `EditForm` element. They inherit from the abstract `InputBase` class. We can add additional input components by creating components that also inherit from the `InputBase` class.

The input data is validated both when the form is submitted and when the data is changed.

# Using the validation components

Input validation is an important aspect of every application since it prevents users from entering invalid data. The Blazor WebAssembly framework uses data annotations for input validation. There are over 30 built-in data annotation attributes. This is a list of the ones that we will be using in this project:

- `Required`: This attribute specifies that a value is required. It is the most commonly used attribute.
- `Display`: This attribute specifies the string to display in error messages.
- `MaxLength`: This attribute specifies the maximum string length allowed.
- `Range`: This attribute specifies the numeric range constraints of the value.

The following code demonstrates the use of a few data annotations:

```
[Required]
public DateTime? Date { get; set; }
[Required]
[Range(0, 500, ErrorMessage = "The {0} field must be < {2}.")]
public decimal? Amount { get; set; }
```

There are two built-in validation components:

- `ValidationSummary`
- `ValidationMessage`

The `ValidationSummary` component summarizes the validation messages, while the `ValidationMessage` component shows the validation messages for individual components. An `EditForm` component can include both types of validation components. However, in order to use either type of validation component, we must add `DataAnnotationsValidator` to the `EditForm` component.

The following screenshot shows the results of both a `ValidationSummary` component and individual `ValidationMesssage` components:

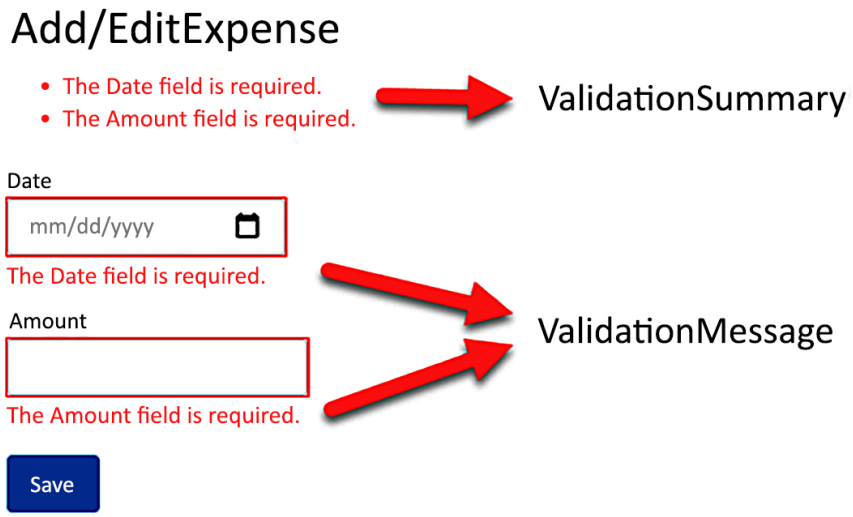



Figure 9.2 – Validation components

Now let's get a quick overview of the project that we are going to build in this chapter.

# Project overview

In this chapter, we will build a project to track travel expenses. We will be able to view, add, and edit expenses. The expenses will be stored in a SQL Server database.

This is a screenshot of the **Add/Edit Expense** page from the completed application.

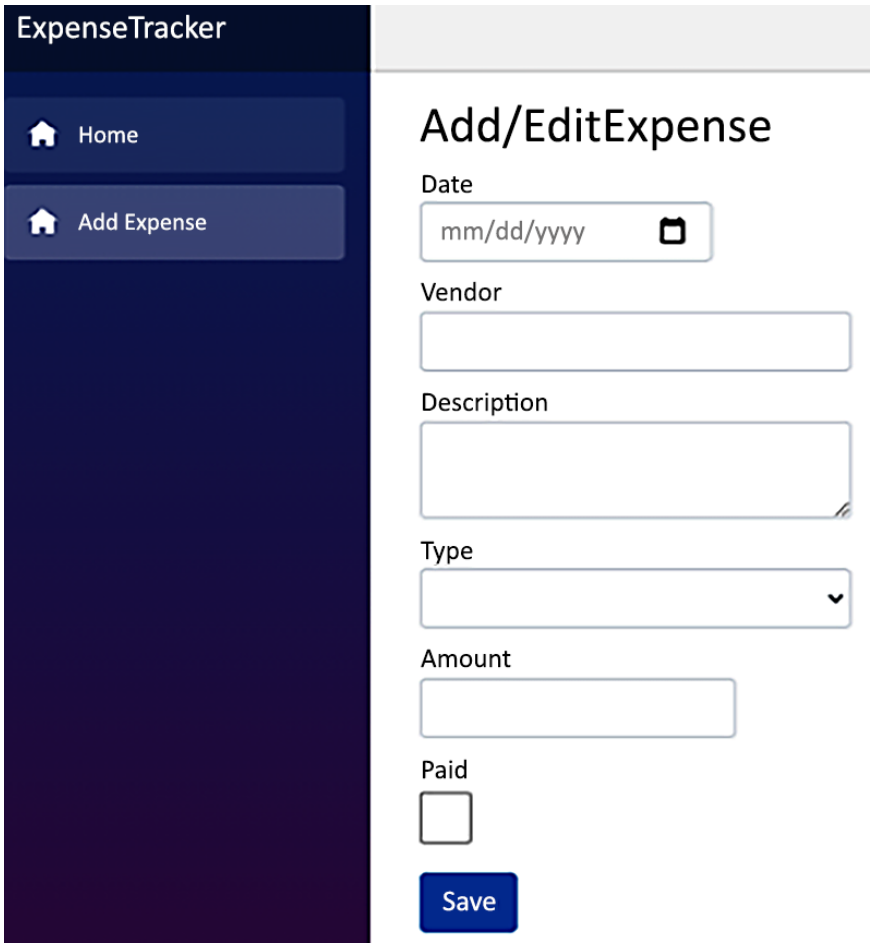



Figure 9.3 – Add/Edit Expense page

The build time for this project is approximately 90 minutes.

# Creating the ExpenseTracker project

The `ExpenseTracker` project will be created by using **Microsoft's Blazor WebAssembly App** project template to create a hosted Blazor WebAssembly app. First, we will remove the demo project. Then, we will add the classes and API controllers needed for our project. We will add a table to the **Home** page to display the current list of expenses. Finally, we will use the `EditForm` component in conjunction with many of the built-in input components to add and edit the expenses.

## Getting started with the project

We need to create a new Blazor WebAssembly app. We do this as follows:

1. Open **Visual Studio 2019**.
2. Click the **Create a new project** button.
3. In the **Search for templates** (*Alt* + *S*) textbox, enter `Blazor` and hit the *Enter* key.

   The following screenshot shows the **Blazor WebAssembly App** project template that we will be using.

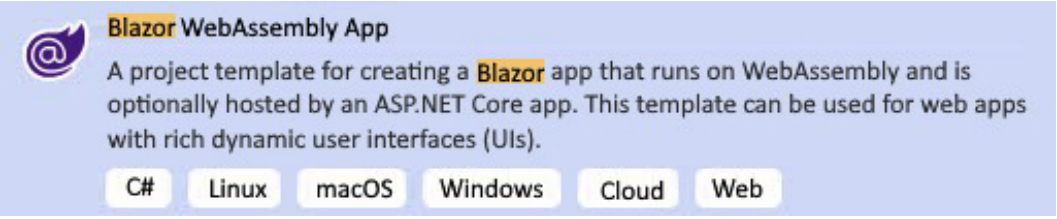



Figure 9.4 – Blazor WebAssembly App project template

4. Select the **Blazor WebAssembly App** project template and then click the **Next** button.

5. Enter `ExpenseTracker` in the **Project name** textbox and click the **Next** button:

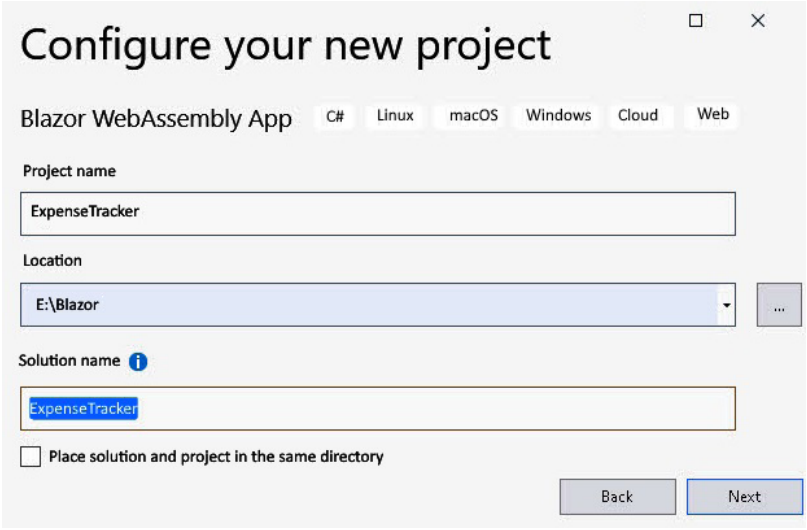



Figure 9.5 – Configure your new project dialog

> **Tip**
> In the preceding example, we placed the `ExpenseTracker` project into the `E:/Blazor` folder. However, the location of this project is not important.

6. Select **.NET 5.0** as the **Target Framework**.
7. Check the **ASP.NET Core Hosted** checkbox.

   When you check the **ASP.NET Core Hosted** checkbox, the project template will create a multi-project solution. For more information on hosted applications, refer to *Chapter 8, Building a Task Manager Using ASP.NET Web API*.
8. Click the **Create** button.
9. Right-click the `ExpenseTracker.Server` project and select the **Set as Startup Project** option from the menu.

You have created the `ExpenseTracker` Blazor WebAssembly project. The project that is created using the project template for hosted applications includes a demo project. Before we can get started, we need to remove the demo project.

## Removing the demo project

To remove the demo project, we need to delete some components, update a couple of components, and delete both a controller and a class. We do this as follows:

1. Delete all of the components in the `ExpenseTracker.Client.Pages` folder, except for `Index`.
2. Delete the `ExpenseTracker.Client.Shared\SurveyPrompt.razor` file.
3. Open the `ExpenseTracker.Client.Shared\MainLayout.razor` file.
4. Remove the `About` link from the top row of the layout by removing the following markup:

```
<a href="http://blazor.net" target="_blank"
   class="ml-md-auto">
        About
</a>
```

5. Open the `ExpenseTracker.Client.Shared\NavMenu.razor` file.
6. Remove the `li` elements for the `Counter` and `Fetch data` pages.
7. From the `ExpenseTracker.Server` project, delete the `Controllers\WeatherForecastConroller.cs` file.
8. From the `ExpenseTracker.Shared` project, delete the `WeatherForecast.cs` file.
9. From the **Build** menu, select the **Build Solution** option.

We have prepared the solution by removing the demo project. Now, we can start adding our `ExpenseTracker` specific content. First, we need to add a couple of classes.

## Adding the classes

We need to add both an `ExpenseType` class and an `Expense` class. We do this as follows:

1. Right-click the `ExpenseTracker.Shared` folder and select the **Add, Class** option from the menu.
2. Name the new class `ExpenseType`.
3. Click the **Add** button.
4. Make the class public by adding the `public` modifier:

```
public class ExpenseType
```

5. Add the following properties to the `ExpenseType` class:

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

6. Right-click the `ExpenseTracker.Shared` folder and select the **Add, Class** option from the menu.
7. Name the new class `Expense`.
8. Click the **Add** button.
9. Make the class public by adding the `public` modifier:

```
public class Expense
```

10. Add the following `using` statement:

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

11. Add the following properties to the `Expense` class:

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

[Required]
public DateTime? Date { get; set; }

[Required]
[MaxLength(100)]
public string Vendor { get; set; }

public string Description { get; set; }

[Required]
[Display(Name = "Expense Type")]
public int? ExpenseTypeId { get; set; }

[Required]
[Range(0, 500, ErrorMessage = "The {0} field must be <=
{2}.")]
public decimal? Amount { get; set; }

public bool Paid { get; set; }
```

In the preceding code, we have used data annotations to add some simple data validation. `Date`, `Vendor`, `ExpenseTypeId`, and `Amount` are all required. The maximum length of `Vendor` is 100 characters. The display name for `ExpenseTypeId` is `Expense Type`. `Amount` of the expense is capped at `500`.

12. From the **Build** menu, select the **Build Solution** option.

We have added both the `ExpenseType` class and the `Expense` class. Now we need to configure the Web API endpoints.

## Adding the API controllers

We need to add an API controller for each of the new classes. We do this as follows:

1. Right-click the `ExpenseTracker.Server.Contollers` folder and select the **Add, Controller** option from the menu.
2. Select the **API Controller with actions, using Entity Framework** option.
3. Click the **Add** button.
4. Set **Model class** to `ExpenseType (ExpenseTracker.Shared)`.
5. Click the **Add data context** button to open the **Add Data Context** dialog:


Figure 9.6 – Add Data Context dialog

6. Click the **Add** button to accept the default values.

7. Click the **Add** button:

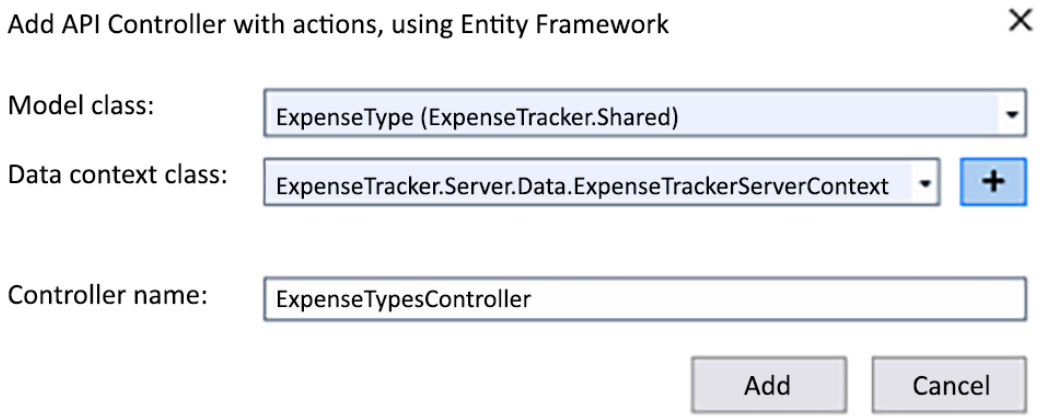



Figure 9.7 – Add API Controller with actions, using Entity Framework dialog

8. Update the route to the following:

```
[Route("[controller]")]
```

9. Right-click the `ExpenseTracker.Server.Contollers` folder and select the **Add, Controller** option from the menu.
10. Select the **API Controller with actions, using Entity Framework** option.
11. Click the **Add** button.
12. Set **Model class** to `Expense (ExpenseTracker.Shared)`.
13. Click the **Add** button.
14. Update the route to the following:

```
[Route("[controller]")]
```

We have added two new controllers to provide the API endpoints that our application will use. Next, we need to create the SQL Server database.

## Creating the SQL Server database

We need to create the SQL Server database and add two tables to it. We do this as follows:

1. Open the `ExpenseTracker.Server\appsettings.json` file.
2. Update the connection string to point to your instance of SQL Server and change the name of the database to `ExpenseTracker`:

```
  "ConnectionStrings": {
    "ExpenseTrackerServerContext": "Server=TOI-WORK\\
  SQLEXPRESS2019; Database=ExpenseTracker; Trusted_
  Connection=True; MultipleActiveResultSets=true"
  }
```

   The preceding code assumes that our server is named `TOI-WORK\\SQLEXPRESS2019` and the name of the database is `ExpenseTracker`.

> **Important note**
>
> Although the example is using SQL Server Express 2019, it does not matter what version of SQL Server you use.

3. Open the `ExpenseTracker.Server.Data\ExpenseTrackerServerContext.cs` file.
4. Add the following `OnModelCreating` method:

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<ExpenseType>().HasData(
    new ExpenseType { Type = "Airfare", Id = 1 },
    new ExpenseType { Type = "Lodging", Id = 2 },
    new ExpenseType { Type = "Meal", Id = 3 },
    new ExpenseType { Type = "Other", Id = 4 }
    );
}
```

   The preceding code will seed the `ExpenseType` table.

5. From the **Tools** menu, select the **NuGet Package Manager, Package Manager Console** option.
6. In the **Package Manager Console**, change **Default project** to `ExpensesManager.Server`.
7. Execute the following commands in the **Package Manager Console**:

```
Add-Migration Init
Update-Database
```

   The preceding commands use **Entity Framework** migrations to update SQL Server.

8. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.
9. Add `/expensetypes` to the address bar and click *Enter*.

   The following screenshot shows the JSON that is returned by `ExpenseTypesController`.

[{“id:1,“type”:“Airfare”},{“id:2,“type”:“Lodging”},{“id:3,“type”:“Heal”},{“id:4,“type”:“Other”}]

Figure 9.8 – JSON returned by the ExpenseTypes API controller

10. Close the browser.

We have created a new database on SQL Server, added two tables, and populated one of the tables with seed data. After we finished setting up SQL Server, we tested that `ExpenseTypesController` works. Finally, we are ready to create a component to display the expenses.

## Viewing the expenses

We need to add a table to display the list of expenses. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `ExpenseTracker.Client.Pages\Index.razor` page.
3. Update the markup to the following:

```
@page "/"

@using ExpenseTracker.Shared
@inject HttpClient Http

```

```
<h2>Expenses</h2>
@if (expenses == null)
{
    <p><em>Loading...</em></p>
}
else if (expenses.Count == 0)
{
    <div>None Found</div>
}
else
{

}

@code{
    IList<Expense> expenses;
}
```

The preceding code defines `expenses` as an `IList<Expense>` and checks to see if it is `null` or empty.

4. Add the following `OnInitializedAsync` method to the `@code` block.

```
protected override async Task OnInitializedAsync()
{
    expenses = await Http.GetFromJsonAsync
            <IList<Expense>>("Expenses");
}
```

The preceding code populates the `expenses` object.

5. Add the following `table` to the `else` statement:

```
<table class="table">
</table>
```

6. Add the following `thead` element to `table`:

```
<thead>
    <tr>
        <th></th>
```

```
            <th>#</th>
            <th>Date</th>
            <th>Vendor</th>
            <th class="text-right">Amount</th>
        </tr>
    </thead>
```

7. Add the following `tbody` element to `table`:

```
    <tbody>
        @foreach (var item in expenses)
        {
        <tr class="@(item.Paid ? "" : "table-danger")">
            <td>
                <a href="/expense/@item.Id">Edit</a>
            </td>
            <td>@item.Id</td>
            <td>@item.Date.Value.ToShortDateString()</td>
            <td>@item.Vendor</td>
            <td class="text-right">@item.Amount</td>
        </tr>
        }
    </tbody>
```

The preceding code loops through each of the `Expense` objects in the `expenses` object and displays them as rows in a table. If the expense is not yet paid, the row is highlighted in red by using the `table-danger` class.

8. From the **Debug** menu, select the **Start Without Debugging** (*Ctrl* + *F5*) option to run the project.

The following screenshot shows the empty **Home** page:

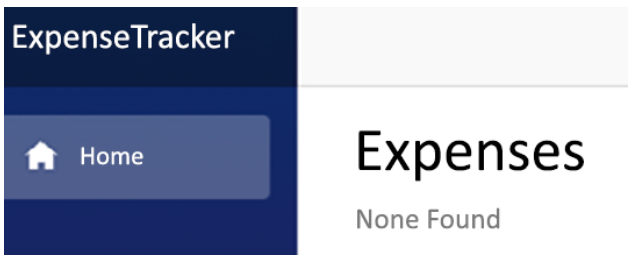



Figure 9.9 – Empty Home page

We have added the ability to display the expenses in a table. Next, we need to add the ability to add expenses.

## Adding the ExpenseEdit component

We need to add a component to enable us to add and edit expenses. We do this as follows:

1. Return to **Visual Studio**.
2. Open the `ExpenseTracker.Client.Shared\NavMenu.razor` page.
3. Add the following markup to the `ul` element:

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="expense">
        <span class="oi oi-home"
              aria-hidden="true"></span>
        Add Expense
    </NavLink>
</li>
```

4. Right-click the `ExpenseTracker.Client.Pages` folder and select the **Add, Razor Component** option from the menu.
5. Name the new component `ExpenseEdit`.
6. Click the **Add** button.
7. Update the markup to the following:

```
@page "/expense"
@page "/expense/{id:int}"

@using ExpenseTracker.Shared
@inject HttpClient Http
@inject NavigationManager Nav

<h3>Add/Edit Expense</h3>

@if (!ready)
{
    <p><em>Loading...</em></p>
}
```

```
else
{
    <EditForm Model="expense"
              OnValidSubmit="HandleValidSubmit">

    </EditForm>

    <div>@error</div>
}
```

The preceding code displays `EditForm` if the component is ready.

8. Add the following `@code` block:

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

    private bool ready;
    private string error;
    private Expense;
    private IList<ExpenseType> types;
}
```

9. Add the following `OnInitializedAsync` method to the `@code` block:

```
protected override async Task OnInitializedAsync()
{
    types = await Http.GetFromJsonAsync
        <IList<ExpenseType>>("ExpenseTypes");

    if (id == 0)
    {
        expense = new Expense();
    }
    else
    {
        expense = await Http.GetFromJsonAsync
            <Expense>($"Expenses/{id}");
    }
```

```
        ready = true;
    }
```

The preceding code initializes both the `types` object and the `expense` object. Once they have both been initialized, the value of `ready` is set to `true`.

10. Add the following `HandleValidSubmit` method to the `@code` block:

```
    private async Task HandleValidSubmit()
    {
        HttpResponseMessage response;
        if (expense.Id == 0)
        {
            response = await Http.PostAsJsonAsync
                ("Expenses", expense);
        }
        else
        {
            string requestUri = $"Expenses/{expense.Id}";
            response = await Http.PutAsJsonAsync
                (requestUri, expense);
        };

        if (response.IsSuccessStatusCode)
        {
            Nav.NavigateTo("/");
        }
        else
        {
            error = response.ReasonPhrase;
        };
    }
```

The preceding code adds new expenses by using the `PostAsJsonAsync` method and updates existing expenses by using the `PutAsJsonAsync` method. If the relevant method is successful, the user is returned to the **Home** page. Otherwise, an error message is displayed.

We have completed the code for this component, but `EditForm` is still empty. We need to add some markup to `EditForm`.

## Adding the input components

We need to add input components to the `EditForm` element. We do this as follows:

1. Add the following markup to `EditForm` to input `Date`:

```
<div>
    <label>
        Date
        <InputDate @bind-Value="expense.Date"
                   class="form-control" />
    </label>
</div>
```

2. Add the following markup to `EditForm` to input `Vendor`:

```
<div>
    <label class="d-block">
        Vendor
        <InputText @bind-Value="expense.Vendor"
                   class="form-control" />
    </label>
</div>
```

3. Add the following markup to `EditForm` to input `Description`:

```
<div>
    <label class="d-block">
        Description
        <InputTextArea @bind-Value=
                         "expense.Description"
                       class="form-control" />
    </label>
</div>
```

4. Add the following markup to `EditForm` to input `ExpenseTypeId`:

```
<div>
    <label class="d-block">
        Type
        <InputSelect @bind-Value=
                       "expense.ExpenseTypeId"
```

```
                        class="form-control">
                <option value=""></option>
                @foreach (var item in types)
                {
                    <option value="@item.Id">
                        @item.Type
                    </option>
                }
            </InputSelect>
        </label>
    </div>
```

5. Add the following markup to `EditForm` to input `Amount`:

```
    <div>
        <label>
            Amount
            <InputNumber @bind-Value="expense.Amount"
                         class="form-control" />
        </label>
    </div>
```

6. Add the following markup to `EditForm` to input `Paid`:

```
    <div>
        <label>
            Paid
            <InputCheckbox @bind-Value="expense.Paid"
                           class="form-control" />
        </label>
    </div>
```

7. Add the following markup for the `Submit` button:

```
    <div class="pt-2 pb-2">
        <button type="submit"
                class="btn btn-primary mr-auto">
            Save
        </button>
    </div>
```

8. Add the following markup to add the validation summary:

```
<DataAnnotationsValidator />
<ValidationSummary />
```

9. From the **Build** menu, select the **Build Solution** option.
10. Return to the browser.
11. Use *Ctrl* + *R* to refresh the browser.
12. Select the **Add Expense** option from the menu.
13. Click the **Save** button.

The following screenshot shows the validation errors:

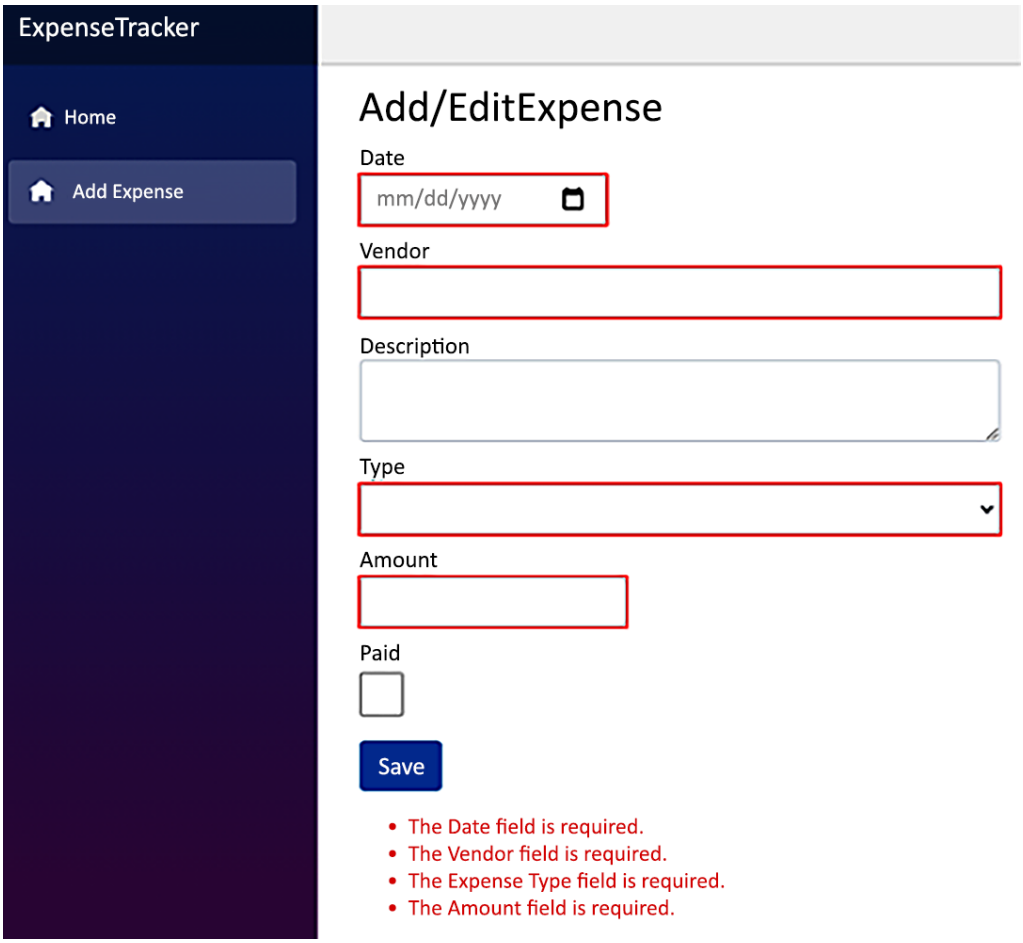



Figure 9.10 – Data validation for the ExpenseEdit component

14. Add an expense.
15. Click the **Save** button.

We have completed the expense tracker project.

# Summary

You should now be able to use the `EditForm` component in conjunction with the built-in input components to input data. You should also be comfortable with the built-in validation components.

In this chapter, we introduced the built-in `EditForm` component, various input components, and validation components. After that, we used the **Blazor WebAssembly App** project template to create a multi-project solution. We added a couple of classes and API controllers. Next, we configured SQL Server by updating the connection string to the database and using Entity Framework migrations. We updated the **Home** page to display the list of expenses. Finally, we added a new page that includes an `EditForm` component and many of the built-in input components in order to input, validate, and submit the expenses.

We can apply our new skills to add data input, validation, and submission to any Blazor WebAssembly app.

The next step is to start building your own web apps. To stay up to date and learn more about Blazor WebAssembly, visit `https://blazor.net`, and read the *ASP.NET Blog* at `https://devblogs.microsoft.com/aspnet`.

We hope you enjoyed the book and wish you every success!

# Questions

The following questions are provided for your consideration:

1. What is the advantage of using the built-in input components?
2. What additional input components would you like to add to the list of built-in input components that are already provided by the framework?

# Further reading

The following resources provide more information concerning the topics in this chapter:

- For more information on ASP.NET Core component forms, refer to `https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms`.
- For more information on data annotations, refer to `https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations`.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

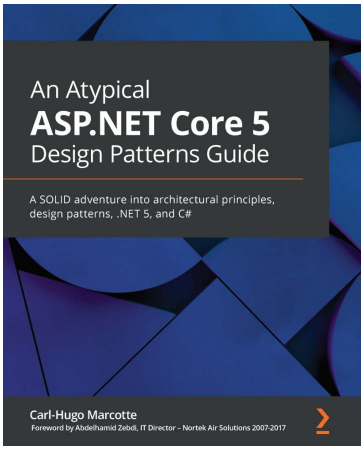
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

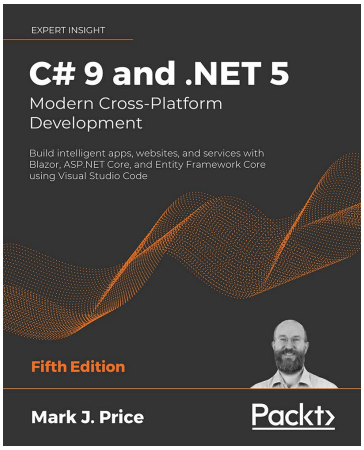



**An Atypical ASP.NET Core 5 Design Patterns Guide**

Carl-Hugo Marcotte

ISBN: 978-1-78934-609-1

- Apply the SOLID principles for building flexible and maintainable software
- Get to grips with .NET 5 dependency injection
- Work with GoF design patterns such as strategy, decorator, and composite
- Explore the MVC patterns for designing web APIs and web applications using Razor
- Discover layering techniques and tenets of clean architecture
- Become familiar with CQRS and vertical slice architecture as an alternative to layering
- Understand microservices, what they are, and what they are not
- Build ASP.NET UI from server-side to client-side Blazor




**C# 9 and .NET 5 – Modern Cross-Platform Development - Fifth Edition**

Mark J. Price

ISBN: 978-1-80056-810-5

- Build your own types with object-oriented programming
- Query and manipulate data using LINQ
- Build websites and services using ASP.NET Core 5
- Create intelligent apps using machine learning
- Use Entity Framework Core and work with relational databases
- Discover Windows app development using the Universal Windows Platform and XAML
- Build rich web experiences using the Blazor framework
- Build mobile applications for iOS and Android using Xamarin.Forms

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Hello!

I am Toi, author of Blazor WebAssembly by Example. I hope you enjoyed reading my book and that you found the book's projects a practical way for you to learn the fundamentals about the Blazor WebAssembly framework.

It would really help me (and other potential readers!) if you would leave a review on Amazon to share your thoughts on my book.

Your review will help me understand what has worked well in this book, as well as what could be improved upon for future editions. So, I would really appreciate the feedback.

Best Wishes,

Toi B. Wright

# Index

## Symbols



## A



## B



## C

## D



## E



## G

## H



## I



## J



## K



## L

# M



# O



# P



# R

# S

# T



# U



# V



# W