EXPERT INSIGHT

Web Development with Blazor

A practical guide to building interactive UIs with C# 14 and .NET 10

Fourth Edition

Foreword by
Daniel Roth
Principal Product Manager
ASP.NET Core and Blazor
Microsoft

Jimmy Engström

packt

Web Development with Blazor

Fourth Edition

A practical guide to building interactive UIs with C# 14 and .NET 10

Jimmy Engström

‹packt›

Web Development with Blazor

Fourth Edition

Portfolio Director: Ashwin Nair
Portfolio Manager: Nitin Nainani
Project Manager: Ruvika Rao
Content Engineer: Adrija Mitra
Technical Editor: Rohit Singh
Indexer: Manju Arasan
Proofreader: Adrija Mitra
Production Designer: Prashant Ghare
Growth Lead: Anamika Singh

First Published: June 2021
Second Edition: March 2023
Third Edition: April 2024
Fourth Edition: June 2026
Production reference: 1080626

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80611-289-0
www.packtpub.com

I dedicate this book to my mom and dad, who gave me my first computer and encouraged my curiosity without knowing it would shape the rest of my life. Without your support and encouragement, I would never have started this journey.

To my brother, who patiently taught me how to code and helped turn that curiosity into a passion that would eventually become my career. Thank you for taking the time to share your knowledge and inspire me to keep learning.

To my sister, who helped me with my English homework growing up. Without that help, my life would probably have looked very different. Knowing English has allowed me to travel the world, speak internationally, and make friends from all over the globe.

And most of all, to my wife, Jessica. Thank you for your endless support, encouragement, patience, and for always believing in me, even during the late nights, deadlines, and stressful moments. I love you. None of this would have been possible without you.

Thank you to everyone who helped review this book and make it better.

– Jimmy Engström

Since my parents don't know English very well, I would like to add this in Swedish as well.

Jag dedikerar den här boken till min mamma och pappa, som gav mig min första dator och uppmuntrade min nyfikenhet utan att veta att det skulle forma resten av mitt liv. Utan ert stöd och er uppmuntran hade jag aldrig påbörjat den här resan.

Till min bror, som tålmodigt lärde mig att koda och hjälpte mig att förvandla den nyfikenheten till en passion som till slut blev min karriär. Tack för att du tog dig tiden att dela med dig av din kunskap och inspirera mig att fortsätta lära mig.

Till min syster, som hjälpte mig med engelskaläxorna när jag växte upp. Utan den hjälpen hade mitt liv förmodligen sett väldigt annorlunda ut. Tack vare engelskan har jag fått möjlighet att resa världen över, tala internationellt och få vänner från hela världen.

Och framför allt till min fru Jessica. Tack för ditt oändliga stöd, ditt tålamod, din uppmuntran och för att du alltid trott på mig, även under sena kvällar, deadlines och stressiga stunder. Jag älskar dig. Det här hade aldrig varit möjligt utan dig.

Tack också till alla som hjälpt till att granska boken och gjort den bättre.

– Jimmy Engström

Foreword

When we first set out to build Blazor, the goal was simple to describe and challenging to achieve: enable developers to build rich, interactive web applications using the language, tools, and platform they already know and love, without sacrificing the capabilities of the web. In the years since, Blazor has grown from an ambitious experiment into a core part of modern web development with .NET. Just as importantly, a vibrant community of developers, educators, and contributors has grown alongside it. This book, now in its fourth edition, is one of the best examples of that community at work.

I've had the pleasure of knowing Jimmy for many years, and there are few people who combine deep technical expertise with genuine enthusiasm the way he does. Jimmy understands deeply how Blazor works and has extensive real-world experience using Blazor for production applications. He's also a talented teacher and educator. He takes concepts that can seem complex at first and presents them through practical, approachable examples that help developers build confidence as they learn. Whether you're encountering Blazor for the first time or looking to deepen your existing knowledge, that ability to teach through real-world experience makes a tremendous difference.

What makes this fourth edition especially valuable is how closely it follows the evolution of Blazor and modern web development. Today's Blazor applications can render on the server, run in the browser through WebAssembly, or combine multiple rendering approaches within a single application. Developers have more flexibility than ever to choose the right model for each scenario. Keeping pace with that evolution is no small task, and Jimmy has done the hard work of ensuring that readers are learning the current state of the platform and the patterns that matter today.

The broader .NET ecosystem has evolved as well. Modern web applications rarely exist in isolation. They interact with APIs, databases, caches, cloud services, and increasingly complex distributed systems. At the same time, developers are expected to move quickly while maintaining reliability and visibility into how their applications behave. Tools such as Aspire help address these challenges by making it easier to develop, run, and observe multi-service applications. Throughout this edition, Jimmy incorporates modern practices and tooling that reflect how real applications are built, helping readers move beyond individual components and toward complete, production-ready solutions.

Whether you're building your first Blazor application or looking to sharpen your skills with the latest capabilities, you're in excellent hands. Jimmy will guide you from the fundamentals through the modern techniques, tools, and patterns that make Blazor such a productive and enjoyable platform for building web applications.

Read it, build with it, experiment, and most importantly, enjoy the process. Welcome to Blazor. I can't wait to see what you build!

Daniel Roth

Principal Product Manager, ASP.NET Core and Blazor

Microsoft

Contributors

About the author

Jimmy Engström has been programming since he was 7 years old and got his first computer. He loves staying on the cutting edge of technology and trying new things. When he first discovered Blazor, he immediately saw its potential and adopted it while it was still in beta. He has been running Blazor in production since Microsoft launched it.

His passion for the .NET industry and community has taken him around the world, speaking about development and sharing knowledge with other developers. Microsoft has recognized this passion by awarding him the Microsoft **Most Valuable Professional** (**MVP**) award 12 years in a row.

Jimmy was named Educator of the Year 2025 for his contributions to the developer community. He is also an instructor on *Dometrain*, where he creates video courses focused on .NET and Blazor development.

In addition to writing books, Jimmy is an international speaker, content creator, and host of the Coding After Work podcast and YouTube channel, where he talks with developers and technology leaders from around the world.

About the reviewers

Stacy Cashmore has been developing solutions since the mid-1990s in various companies and industries, ranging from facilitating contract jobbing to allowing consumers to close a mortgage without the help of a financial adviser, with lots in between.

She has a passion for sharing knowledge, using storytelling to share her experiences to help teams grow in the ways they develop software and work together, and performing live coding demonstrations to inspire others to try new technologies.

In 2022, Stacy published her first book, aimed at helping developers get started building dynamic applications using C#, Azure Functions, and Azure Static Web Apps. In February 2026, she released the second edition, updated for .NET 9 and with added infrastructure-as-code content to help readers apply what they've learned in real-world scenarios.

For her efforts in the community, Stacy has been awarded the Microsoft MVP for Developer Technologies since 2020.

Mabrouk Mahdhi is a Principal Software Engineer and a recognized Microsoft MVP in .NET technologies. He specializes in modern .NET development, with deep expertise in Blazor, ASP.NET Core, and .NET MAUI, building scalable web applications, enterprise APIs, and high-performance, data-driven systems.

He is the founder and organizer of Tunisia Dev Days, one of the largest .NET and software engineering conferences in North Africa, bringing together developers, architects, and technology leaders to discuss innovation, AI, cloud computing, and modern software engineering practices.

Moien Tajik is a Principal Software Engineer at AIHR in the Netherlands, with experience leading engineering teams across both B2C and B2B domains. Previously a Technical Fellow at Alibaba Travels Company, he has deep expertise across the .NET ecosystem, from backend to frontend, and has designed and scaled systems with different architectures, including microservices, modular monoliths, and cloud and on-premise deployments.

He brings a strong foundation in software architecture, distributed systems, and technical leadership, with a track record of delivering robust, scalable solutions. Moien is passionate about technology, product innovation, AI, and engineering excellence.

Eder Bragança Pereira is a computer scientist and a software engineer focused on JavaScript. He is passionate about technology and works on building web applications, while also exploring game development as an area of growing interest.

Table of Contents

Preface

Building modern web applications has traditionally required developers to combine multiple technologies and frameworks. With Blazor and ASP.NET Core, developers can now build interactive, modern web applications using C# and .NET across both the server and the browser.

This book will guide you through the most common scenarios you will encounter when building applications with Blazor. You will learn how Blazor fits into the modern ASP.NET Core ecosystem and how to use features such as **server-side rendering** (**SSR**), streaming rendering, enhanced navigation, interactive server rendering, WebAssembly, and hybrid applications.

As you progress through the book, you will learn how to create Blazor applications, work with Razor components, build reusable UI components, validate forms, manage state, integrate APIs, and structure production-ready applications. The focus throughout the book is practical development, helping you understand not only how the features work individually, but also how they work together in real-world applications.

By the end of this book, you will have the knowledge and confidence needed to build, deploy, and maintain modern, production-ready Blazor applications.

Who this book is for

The book is for web developers and software developers who want to explore Blazor to learn how to build dynamic web UIs. This book assumes familiarity with C# programming and web development concepts.

What this book covers

Chapter 1, *Hello Blazor*, will teach you where Blazor comes from and how the different hosting models fit together. You will learn the difference between Blazor Server, Blazor WebAssembly, Blazor Hybrid, and server-side rendering.

Chapter 2, *Creating Your First Blazor App*, will help you set up your development environment and create your first Blazor app. You will also learn the project structure so we have a solid starting point for the rest of the book.

Chapter 3, *Exploring Render Modes*, will teach you how Blazor renders components using static SSR, streaming SSR, Interactive Server, Interactive WebAssembly, and Auto mode. You will learn when each render mode makes sense.

Chapter 4, *Uncovering Aspire*, will teach you how Aspire helps us run and connect the different parts of our application. You will also see how the dashboard, logs, traces, and metrics make development easier.

Chapter 5, *Managing State – Part 1*, will teach you how to store and access data for the blog application. You will set up PostgreSQL with Aspire and use Entity Framework to work with the data.

Chapter 6, *Understanding Basic Blazor Components*, will teach you how Blazor components work. You will learn the basics of Razor syntax, dependency injection, lifecycle events, and parameters.

Chapter 7, *Creating Advanced Blazor Components*, will teach you how to build more reusable components. You will learn about binding, `EventCallback`, `RenderFragment`, and built-in Blazor components.

Chapter 8, *Building Forms with Validation*, will teach you how to create forms and validate user input in Blazor. We will use this to build the admin interface for managing blog posts.

Chapter 9, *Creating an API*, will teach you how to create a web API using Minimal APIs. This allows the WebAssembly parts of the app to communicate with the server.

Chapter 10, *Adding Authentication and Authorization*, will teach you how to secure the blog application. You will add authentication, protect pages and API calls, and work with roles.

Chapter 11, *Sharing Code and Resources*, will teach you how to share components, CSS, images, and other static files between projects. This helps us keep things consistent across multiple Blazor applications.

Chapter 12, *JavaScript Interop*, will teach you how Blazor can talk to JavaScript, and how JavaScript can talk back to .NET. You will also learn when JavaScript is still needed in a Blazor app.

Chapter 13, *Managing State – Part 2*, will teach you more ways to keep state in a Blazor application. You will learn how to keep data around when users navigate, reload the page, or come back later.

Chapter 14, *Debugging the Code*, will teach you how to debug Blazor applications. You will look at debugging Blazor Server, Blazor WebAssembly, browser debugging, and Hot Reload.

Chapter 15, *Exploring Tracing and Metrics*, will teach you how to understand what your app is doing when it runs. You will learn how logs, metrics, traces, OpenTelemetry, and Aspire can help us find problems.

Chapter 16, *Testing*, will teach you how to test Blazor components with bUnit. You will learn how to write tests that verify component behavior and markup.

Chapter 17, *Deploy to Production*, will teach you what to think about when deploying a Blazor application. You will learn the main options and decisions without tying the chapter to one specific provider.

Chapter 18, *Moving from, or Combining, an Existing Site*, will teach you how Blazor can work together with existing applications. You will learn how to combine Blazor with Angular, React, MVC, Razor Pages, and web components.

Chapter 19, *Going Deeper into WebAssembly*, will teach you about Blazor WebAssembly-specific topics. You will look at performance, loading, progressive web apps, native dependencies, and common problems.

Chapter 20, *Examining Source Generators*, will teach you what source generators are and why they can be useful. You will also see how Blazor uses generated code behind the scenes.

Chapter 21, *Visiting .NET MAUI*, will teach you how Blazor fits into cross-platform app development. You will learn what .NET MAUI is, how Blazor Hybrid works, and how we can reuse our Blazor knowledge to build apps for Android, iOS, macOS, and Windows.

Chapter 22, *Where to Go from Here*, will wrap up the book by sharing lessons learned from running Blazor in production. You will learn about some real-world issues, such as memory, concurrency, errors, and browser support, and where to go next in the Blazor community.

To get the most out of this book

I recommend reading the first few chapters to ensure you are up to speed with the basic concepts of Blazor in general. The project we are creating is adapted for real-world use, but some parts are left out, such as proper error handling. You should, however, get a good grasp of the building blocks of Blazor.

The book focuses on using Visual Studio 2026. That said, feel free to use whichever version you are comfortable with that supports Blazor.

Software covered in this book	**OS requirements**
Visual Studio 2026, .NET 10	Windows 10 or later, macOS, 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.

I would love for you to share your progress while reading this book or in Blazor development in general. Tweet me at `@EngstromJimmy`.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

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://packt.link/gbp/9781806112890`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "`EditForm` will create an `EditContext` instance as a cascading value so that all the components you put inside of `EditForm` will access the same `EditContext`."

A block of code is set as follows:

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

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

```
var builder = DistributedApplication.CreateBuilder(args);
var postgres = builder.AddPostgres("postgres")
    .WithLifetime(ContainerLifetime.Persistent)
    .WithDataVolume(isReadOnly: false)
    .WithPgAdmin();
```

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

```
dotnet new blazor -o BlazorApp
cd Data
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "When the installation is done, click **Close and restart**".

> **Note**
>
> Warnings or important notes appear like this.

> **Tip**
>
> Tips and tricks 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 or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

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 reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

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 `http://authors.packt.com/`.

Share your thoughts

Once you've read *Web Development with Blazor, Fourth Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

`https://packt.link/r/1806112892`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the *How to Unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:

How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

Part 1

Getting Started with Blazor

In this first part of the book, we'll build a solid understanding of what Blazor is, where it comes from, and how the different hosting models work. We'll create our first Blazor application, explore the new render modes, and look at how Aspire can make local development easier. By the end of this part, we'll have the foundation we need to start building a real Blazor application.

This part includes the following chapters:

- *Chapter 1, Hello Blazor*
- *Chapter 2, Creating Your First Blazor App*
- *Chapter 3, Exploring Render Modes*
- *Chapter 4, Uncovering Aspire*

1

Hello Blazor

Thank you for picking up your copy of *Web Development with Blazor*. This book aims to get you started as quickly and smoothly as possible, chapter by chapter, without requiring you to read the entire book from cover to cover before getting your Blazor on.

This book will begin by guiding you through the most common scenarios you'll encounter when starting your journey with Blazor, and will also dive into a few more advanced scenarios later on. This book aims to show you what Blazor is—Blazor Server, Blazor WebAssembly, Blazor Hybrid (via .NET MAUI), and, on top of that, **Server-Side Rendering** (**SSR**)—and how it all works practically to help you avoid traps.

This is the book's fourth edition. Much has happened since the first edition, including the release of .NET 6 and .NET 7. In the second edition, I updated the content to reflect the changes and the new functionality we gained. The third edition was a massive update, with a lot of changes based on .NET 8, including new render modes.

The updates in .NET 9 and .NET 10 are not as significant as those in .NET 8, though we did get some really nice features, which we will touch on throughout the book.

I give Blazor presentations all over the world, and I often get some common questions. Without going into too much detail, they are often related to download size or startup time for Blazor WebAssembly, and to continuous connection for Blazor Server. Starting in .NET 8 and in later versions, we can leverage a new mode, **Server-Side Rendering** (**SSR**), that solves all of these problems in one swift blow. Okay, maybe not all problems, but we are well on our way to solving them. A common belief is that Blazor runs on WebAssembly, but WebAssembly is just one way to run Blazor. Many books, workshops, and blog posts on Blazor focus heavily on WebAssembly. There are a few differences between the various ways of running Blazor; I will point them out as we go along.

This first chapter will explore the origins of Blazor, the technologies that made Blazor possible, and the various ways of running Blazor. We will also discuss which type (Blazor WebAssembly, Blazor Server, or Blazor Hybrid) is best suited for you.

In this chapter, we will cover the following topics:

- How Blazor was created
- Why Blazor?
- Preceding Blazor
- Introducing WebAssembly
- Introducing .NET

Technical requirements

It is recommended that you have some knowledge of .NET before you start, as this book is aimed at .NET developers who want to utilize their skills to make interactive web applications. However, it's more than possible that you will pick up a few .NET tricks if you are new to the world of .NET.

How Blazor was created

Blazor is an open-source web UI framework. That's a lot of buzzwords in the same sentence, but simply put, it means that you can create interactive web applications using HTML, CSS, and C# with full support for bindings, events, forms and validation, dependency injection, debugging, and much more. We will take a look at these in this book.

In 2017, *Steve Sanderson* (well-known for creating the Knockout JavaScript framework and who used to work for the ASP.NET team at Microsoft) was about to do a session called *Web Apps can't really do *that*, can they?* at the developer conference NDC Oslo.

But Steve wanted to show a cool demo, so he thought, *would it be possible to run C# in WebAssembly?* He found an old, inactive project on GitHub called *DotNetAnywhere* (available at

`https://github.com/chrisdunelm/DotNetAnywhere`), which was written in C, and he utilized tools to compile the C code into WebAssembly.

As a result, he got a simple console app running in the browser. This would have been a fantastic demo for most people, but Steve wanted to take it further. He thought, *is it possible to create a simple web framework on top of this?* He went on to see if he could also get the tooling to work.

When it was time for his session, he had a working sample that could create a new project, create a to-do list with great tooling support, and run the project in the browser.

Damian Edwards (the .NET team) and *David Fowler* (the .NET team) were also at the NDC conference. Steve showed them what he was about to present, and they later described the moment as one when their heads exploded and their jaws dropped.

And that's how the prototype of Blazor came into existence.

The name Blazor comes from a combination of Browser and Razor (which is the technology used to combine code and HTML). Adding an *L* made the name sound better, but other than that, it has no real meaning or acronym.

There are a few different flavors of Blazor, including Blazor Server, Blazor WebAssembly, Blazor Hybrid (using .NET MAUI), and Server-Side Rendering.

The different versions have some pros and cons, all of which I will cover in the upcoming sections and chapters.

When I saw Blazor for the first time, it got me thinking: I wanted to put it to the test. But to do that, we have to go back to 1985. When I was seven years old, I got my first computer, a Sinclair ZX Spectrum. I remember that I sat down and wrote the following:

```
10 PRINT "Jimmy"
20 GOTO 10
```

That was *my* code; I made the computer write my name on the screen over and over!

That was when I decided that I wanted to become a developer to make computers do things.

After becoming a developer, I wanted to revisit my childhood and decided I wanted to build a ZX Spectrum emulator. In many ways, the emulator has become my test project instead of a simple *Hello World* when I encounter new technology. I've had it running on a Gadgeteer, Xbox, Xbox 360, and even a HoloLens (to name a few platforms/devices).

But is it possible to run my emulator in Blazor?

It took me only a couple of hours to get the emulator working with Blazor WebAssembly by leveraging my already built .NET Standard DLL; I only had to write the code specific to this implementation, such as the keyboard and graphics. This is one of the reasons Blazor (both Server and WebAssembly) is so powerful: it can run libraries that have already been made. Not only can you leverage your knowledge of C#, but you can also take advantage of the large ecosystem and .NET community.

Tip

I once got a question from a person I was mentoring regarding software development: *"What is the thing you are most proud of?"* My answer was the ZX Spectrum emulator. It has really challenged me and the technologies I run it on, and I am constantly learning from it. It is also one of my favorite projects to work on, as I keep finding ways to optimize and improve the emulator. You can find the emulator here: `https://zxbox.com`.

Building interactive web applications used to only be possible with JavaScript. Now, we know we can use Blazor.

Why Blazor?

Not that long ago, I was asked by a random person on Facebook whether I work with Blazor.

I said, *"Yes, yes, I do."*

He then continued with a long remark, telling me that Blazor would never beat Angular, React, or Vue.

I see these kinds of remarks quite often, and it's essential to understand that beating other **SPA** (short for **Single-Page Application**) frameworks has never been the goal. This is not *Highlander*, and there can be more than one.

Learning web development has previously been pretty tough. Not only do we need to know ASP.NET for the server, but we also need to learn an SPA framework like React, Angular, or Vue.

But it doesn't end there. We also need to learn NPM, Bower, and Parcel as well as JavaScript or TypeScript.

We need to understand transpiling and build that into our development pipeline. This is, of course, just the tip of the iceberg; depending on the technology, we need to explore other rabbit holes too.

Blazor is an excellent choice for .NET developers to write interactive web applications without needing to learn (or keep up with) everything we just mentioned. We can leverage our existing C# knowledge and the packages we use, and share code between the server and client.

I usually say: "Blazor removes all the things I hate about web development." I guess the saying should be, "Blazor *can* remove all the things I hate about web development." With Blazor, it is still possible to do JavaScript interop and use JavaScript frameworks or other SPA frameworks from within Blazor, but we don't have to.

Blazor has opened a door, allowing me to feel productive and confident that I am creating a great user experience for my users by leveraging my existing C# knowledge.

Preceding Blazor

You probably didn't get this book to read about **JavaScript**, but it helps to remember that we came from a pre-Blazor time. I recall that time – the dark times. Many of the concepts used in Blazor are not that far from those used in many JavaScript frameworks, so I will start with a brief overview of where we came from.

As developers, we have many different platforms we can develop for, including desktop, mobile, games, the cloud (or server-side), AI, and even IoT. All these platforms have a lot of different languages to choose from, but there is, of course, one more platform: the apps that run inside the browser.

I have been a web developer for a long time, and I've seen code move from the server to run within the browser. It has changed the way we develop our apps. Frameworks such as Angular, React, Aurelia, and Vue have revolutionized the web by enabling the update of small parts on the fly, rather than reloading the entire page. This new on-the-fly update method has enabled pages to load more quickly, reducing perceived load time (not necessarily the entire page load).

However, for many developers, this is an entirely new skill set, namely switching between a server (most likely C#, if you are reading this book) and a frontend developed in JavaScript. Data objects are written in C# on the backend, serialized to JSON, sent via an API, and deserialized into another object written in JavaScript on the frontend. The validation logic needs to be duplicated to enable fast client-side validation and to perform an additional check on the server.

JavaScript used to work differently across browsers, which jQuery addressed by providing a common API that was translated into something the browser could understand. Now, the differences between web browsers are much less significant, rendering jQuery obsolete in many cases.

JavaScript differs slightly from other languages, as it is neither object-oriented nor typed, for example. In 2010, Anders Hejlsberg (known for being the original language designer of C#, Delphi, and Turbo Pascal) started working on **TypeScript**. This object-oriented language can be compiled/transpiled into JavaScript.

You can use TypeScript with Angular, React, Aurelia, and Vue, but in the end, it is JavaScript that will run the actual code. Simply put, to create interactive web applications today using JavaScript/TypeScript, you need to switch between languages and choose and keep up with different frameworks.

In this book, we will look at this in another way. Even though we will talk about JavaScript, our primary focus will be on developing interactive web applications mainly using C#.

Now, we know a bit about the history of JavaScript. JavaScript is no longer the only language that can run within a browser, thanks to WebAssembly, which we will cover in the next section.

Introducing WebAssembly

In this section, we will look at how **WebAssembly** works. One way of running Blazor is by using WebAssembly, but for now, let's focus on what WebAssembly is.

WebAssembly is a binary instruction format that is compiled and, therefore, smaller. It is designed for native speeds, which means that when it comes to speed, it is closer to C++ than it is to JavaScript. When loading JavaScript, the JavaScript files (or inline JavaScript) are downloaded, parsed, optimized, and JIT-compiled; most of those steps are not needed for WebAssembly.

WebAssembly has a very strict security model that protects users from buggy or malicious code. It runs within a sandbox and cannot escape that sandbox without going through the appropriate APIs. Suppose you want to communicate outside WebAssembly, for example, by changing the **Document Object Model** (**DOM**) or downloading a file from the web. In that case, you will need to do that with JavaScript interop (more on that later; don't worry – Blazor will solve this for us).

Let's look at some code to get a bit more familiar with WebAssembly. In this section, we will create an app that sums two numbers and returns the result, written in C (to be honest, this is about the level of C I'm comfortable with).

We can compile C into WebAssembly, but it requires the installation of some tooling, so we will not do this all the way. The point here is just to give us a feeling of how WebAssembly works under the hood.

Consider this code:

```
int main() {
  return 1+2;
}
```

The result of this will be the number 3.

WebAssembly is a stack machine language, which means that it uses a stack to perform its operations.

Consider this code:

```
1+2
```

Most compilers will optimize the code and return 3, since it is always the same.

But let's assume that all the instructions should be executed. This is the way WebAssembly would do things:

1. It will start by pushing 1 onto the stack (`instruction: i32.const 1`), followed by pushing 2 onto the stack (`instruction: i32.const 2`). At this point, the stack contains 1 and 2.
2. Then, we must execute the add instruction (`i32.add`), which will pop (get) the two top values (1 and 2) from the stack, add them up, and push the new value onto the stack (3).

This demo shows that we can build WebAssembly from C code. Even though we never need to go to this level to understand WebAssembly (Blazor handles all of that for us), we will use C code and other libraries compiled into WebAssembly later in the book (*Chapter 19*, *Going Deeper into WebAssembly*).

Tip

Other languages

WebAssembly was originally designed as a compilation target for languages such as C, C++, and Rust. However, many other languages and runtimes can also target or run on top of WebAssembly. Here is a great collection of some of these languages: `https://github.com/appcypher/awesome-wasm-langs`.

> WebAssembly is highly performant, often achieving near-native speeds. This performance has made it possible for technologies such as Unity to compile applications for browser-based execution using WebAssembly.
>
> Here are a couple of examples of games running on top of WebAssembly:
>
> - **Angry Bots (Unity)**: `https://beta.unity3d.com/jonas/AngryBots/`
> - **Doom**: `https://wasm.continuation-labs.com/d3demo/`
>
> This is a great list of different WebAssembly projects: `https://github.com/mbasso/awesome-wasm`.

This section touched the surface of how WebAssembly works; in most cases, you won't need to know much more. We will dive into how Blazor uses this technology in *Chapter 3*, *Exploring Render Modes*.

To write Blazor apps, we can leverage the power of .NET 10, which we'll look at next.

Introducing .NET

.NET is a platform developed by Microsoft for building different types of applications, including web, mobile, and desktop applications. The .NET team has been working hard on tightening everything up for us developers for years. They have been making everything simpler, smaller, cross-platform, and open source – not to mention easier to utilize your existing knowledge of .NET development.

.NET Core was a step toward a more unified .NET. It allowed Microsoft to re-envision the whole .NET platform, build it in a completely new way, and make it run on even more platforms.

There were three different types of .NET runtimes:

- .NET Framework (full .NET)
- .NET Core
- Mono/Xamarin

Different runtimes had different capabilities and performances. This also meant that creating a .NET Core app, for example, required different tooling and frameworks to be installed.

.NET 5 was the start of our journey toward one single .NET. With this unified toolchain, the experience of creating, running, and so on became the same across all the different project types. *Framework* and *Core* were dropped from the name. .NET 5 is still modular in a similar

way to what we are used to, so we do not have to worry that merging all the different .NET versions is going to result in a bloated .NET.

Thanks to the .NET platform, you will be able to reach all the platforms we talked about at the beginning of this chapter (web, desktop, mobile, games, the cloud (or server side), AI, and even IoT) using only C# and the same tooling.

Blazor has been around for a while now. In .NET Core 3, the first version of Blazor Server was released, and at Microsoft Build in 2020, Microsoft released Blazor WebAssembly.

In .NET 5, we got a lot of new components for Blazor—pre-rendering and CSS isolation, to name a couple of things. Don't worry; we will go through all these things throughout the book.

In .NET 6, we got even more functionality, like Hot Reload, co-located JavaScript, new components, and much more, all of which we will explore throughout the book.

In .NET 7, we got yet more enhancements for Blazor developers. We got performance improvements and `get`/`set`/`after` modifiers, among other things.

In 2023, Microsoft released .NET 8, bringing a major shift to Blazor. During development, this new way of developing Blazor apps was called *Blazor United*, which is a name they have now updated to simply *Blazor*. This is the new way of creating Blazor applications, and it is an awesome way. But let's save something for later chapters as well.

.NET 8 brought us performance improvements, native AOT, better source generators, and so much more.

.NET 9 was more of a service release, focusing on ensuring everything ran smoothly rather than introducing new features. With that said, we did get a couple of new features, like Authentication State Serialization.

.NET 10 is not a huge release either, but it contains important improvements that make our daily development so much better. Microsoft has focused on making Hot Reload more reliable and much faster. There are also improvements in handling the state. Those two are my favorite features. But there is more, much more, and we will take a look at it throughout the book.

Looking at the enhancements and number of features, I can only conclude that Microsoft believes in Blazor, and so do I.

Now that you know about some of the surrounding technologies, in the next section, it's time to introduce the main character of this book: Blazor.

Summary

In this chapter, we discussed how Blazor was created and its underlying technologies, such as WebAssembly. We revisited the journey Microsoft has taken with .NET, which has enabled us to reach where we stand today. We also talked about the big shift that happened in .NET with the release of .NET 8, where we got so many more options along with the new render modes. At this point, you might feel a bit lost. What is this Blazor Server and WebAssembly stuff?

Don't worry, we will go through all of this in *Chapter 3*, *Exploring Render Modes*.

In the upcoming chapters, I will walk you through various scenarios to equip you with the knowledge to handle everything from upgrading an old or existing site and creating a new server-side site to creating a new WebAssembly site.

In the next chapter, we'll get our hands dirty by configuring our development environment and creating and examining our first Blazor app.

Further reading

- As a .NET developer, you might be interested in the Uno Platform, which makes it possible to create a UI in XAML and deploy it to many different platforms, including WebAssembly: `https://platform.uno/`
- If you want to see how the ZX Spectrum emulator is built, you can download the source code here: `https://github.com/EngstromJimmy/ZXSpectrum`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/8VuShAAtK`

2

Creating Your First Blazor App

In this chapter, we will set up our development environment so that we can start developing Blazor apps. We will create our first Blazor app and go through the project structure. We will also briefly take a look at using the command line to create a project.

By the end of this chapter, you will have a working development environment and will have created a Blazor app that can run a mix of streaming server-side rendering, Blazor Server, and Blazor WebAssembly.

As such, we will cover the following:

- Setting up your development environment
- Creating our first Blazor application
- Introducing the .NET CLI
- Figuring out the project structure

Technical requirements

We will create a new project (a blog engine) and continue working on this project throughout the book.

You can find the source code for this chapter at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter02`.

Setting up your development environment

In this book, the focus will be on Windows development, and any screenshots will be from Visual Studio (unless stated otherwise). But since .NET 10 is cross-platform, we will go through how to set up your development environment on Windows, macOS, and Linux.

The official downloads for all supported platforms are available at `https://visualstudio.microsoft.com/`.

We can download Visual Studio or Visual Studio Code from the webpage.

Windows

On Windows, we have many different options for developing Blazor applications. Visual Studio 2026 is the most powerful tool we can use.

There are three different editions, which are as follows:

- Community 2026
- Professional 2026
- Enterprise 2026

The Community edition is free to use, while the Professional and Enterprise editions are paid. The Community edition does have some limitations, and we can compare the different editions here `https://visualstudio.microsoft.com/vs/compare/`.

For this book, we can use any of these versions. Follow these steps to install Visual Studio on Windows:

1. Download Visual Studio 2026 from `https://visualstudio.microsoft.com/`. Choose the version that is right for you.
2. Install Visual Studio and, during the installation, make sure to select **ASP.NET and web development**. To the right is a list of all the components that will be installed. Check **.NET 10.0 WebAssembly Build Tools** as shown in *Figure 2.1*:

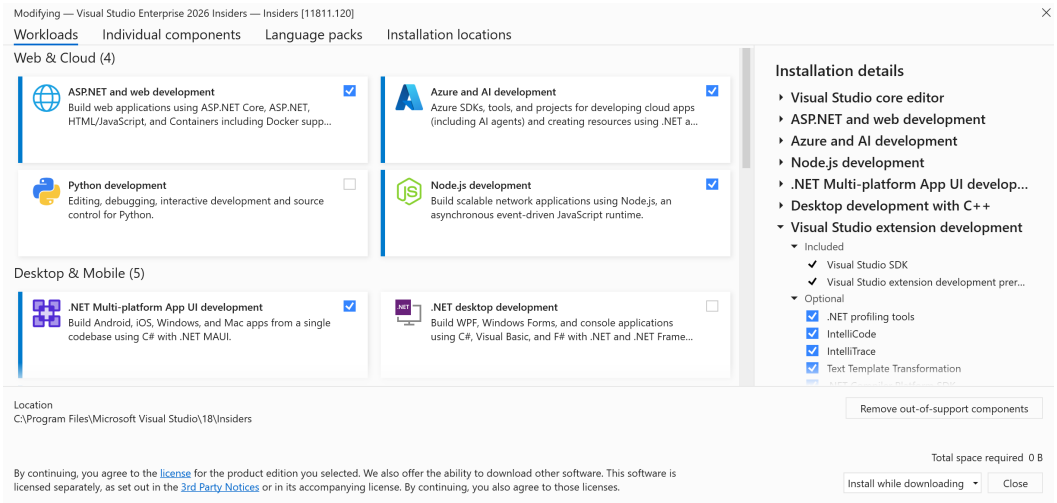


Figure 2.1: Visual Studio 2026 installation on Windows

We can also use Visual Studio Code to develop Blazor on Windows, but we won't discuss the installation process for Windows. That's because if you are on Windows, Visual Studio 2026 is a much better experience.

macOS and Linux (or Windows)

Visual Studio Code is a lightweight, cross-platform editor that runs on macOS, Linux, and Windows. It is the recommended option for Blazor development on non-Windows systems.

You can download the different versions of Visual Studio Code from `https://code.visualstudio.com/Download`.

Once installed, we also need to add two extensions:

1. Open Visual Studio Code and open the extension panel by pressing *Shift* + *Command* + *X* on macOS or *Ctrl* + *Shift* + *X* on Linux and Windows.
2. Search for **C# Dev Kit for Visual Studio Code** and click **Install**. You might need a Microsoft account to install C# Dev Kit.
3. Search for **JavaScript Debugger (Nightly)** and click **Install**.

There are other IDEs that are cross-platform as well, like JetBrains Rider, for example, which some developers may prefer.

Installing Docker

Later in this book, we will work with Aspire, which can automatically set up and manage local development dependencies when starting a project. To take full advantage of Aspire, certain tools must already be installed on your system.

The simplest way to support this workflow is by using Docker. While alternatives such as Podman or WSL can also be used, this book assumes Docker is installed.

For Windows, follow these steps to install Docker:

1. Download Docker Desktop from `https://www.docker.com/get-started`.
2. Run the installer you have downloaded.
3. Select the default boxes as seen in *Figure 2.2* and press **OK**.

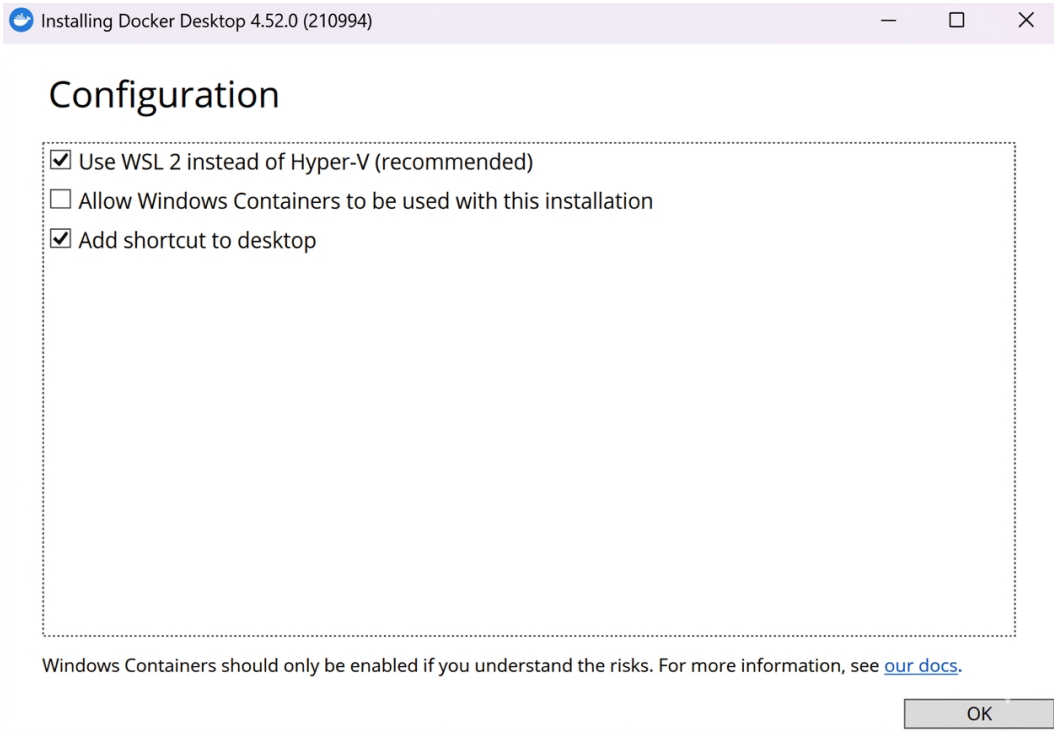


Figure 2.2: Docker install window

4. When the installation is done, click **Close and restart**.

The installation process for macOS and Linux is different but outside the scope of this book.

Now that everything is set up, let's create our first app.

Creating our first Blazor application

Throughout the book, we will create a blog engine. There won't be a lot of business logic that you'll have to learn, the app will be simple to understand, but we will touch on many of the technologies and areas you will encounter when building a Blazor app.

> **Note**
>
> I had an opportunity to discuss the project with Steve Sanderson (creator of Blazor) and Dan Roth (program manager at ASP.NET). We concluded that this would showcase the most important features of Blazor.

The project will allow visitors to read blog posts and post comments. It will also feature an admin site where we can write blog posts that will be password-protected.

We will create an app that leverages Blazor Server, Blazor WebAssembly, and streaming server-side rendering. Don't worry, we will go through the render mode in the next chapter, *Chapter 3*, *Exploring Render Modes*.

> **Note**
>
> In this book, we will use Visual Studio 2026 going forward, but other platforms have similar ways to create projects.

Exploring the templates

In .NET 8, Microsoft reduced the number of templates we have access to. We will explore them further in *Chapter 6*, *Understanding Basic Blazor Components*. This chapter will give you a quick overview.

In .NET 7, we had different templates depending on whether we wanted sample code, but in .NET 10, we only have two web templates and also have one Blazor Hybrid template (.NET MAUI). We will return to this in *Chapter 21*, *Visiting .NET MAUI*.

Blazor Web App template

The **Blazor Web App** template gives us a Blazor app. Once we have selected this template, we get options for how we want our app to run. We can configure our app with or without sample code. We can choose whether our app should support interactive components and what type of interactivity we want to include. We can also choose whether to specify the rendering mode per component or for the entire app. So, right away, we don't need to choose between Blazor Server and Blazor WebAssembly; we can mix and match.

If we add sample pages, we gain a couple of components to see what a Blazor app looks like, along with some basic setup and menu structure. It also contains code for adding Bootstrap, isolated CSS, and other such things (see *Chapter 11*, *Sharing Code and Resources*).

This is the template we will use in the book to better understand how things fit together.

Depending on the choice we make here, the template will generate different results. For example, if we choose the **Auto** or **Interactive WebAssembly** render mode, the template will create two separate projects (more on this in *Chapter 3*, *Exploring Render Modes*).

Blazor WebAssembly Standalone App template

The **Blazor WebAssembly Standalone App** template gives us (as the name implies) a Blazor WebAssembly standalone app. Here, we can choose whether to include sample pages as well. It includes a couple of components to show what a Blazor app looks like, along with basic setup and menu structure. It also includes code for adding Bootstrap, isolated CSS, and related features (see *Chapter 11*, *Sharing Code and Resources*).

So why do we have this one? Well, the Blazor Web App depends on server-rendering technologies in one way or another. If you want to run your app from a file share, GitHub Pages, or Azure Static Web Apps (to name a few), this is the template for you. The very clear downside of Blazor WebAssembly is that it's not great with SEO, but we will come back to that in later chapters.

There are many opinions on choosing templates. For me, the choice is simple: if we have a server, we should choose the Blazor Web App template, which gives us the most options. But if we want to run on GitHub Pages, for example, well, then the WebAssembly Standalone template is the only option.

Creating a Blazor web application

To start, we will create a Blazor web application and play around with it:

1. Start Visual Studio 2026 and you will see the following screen:

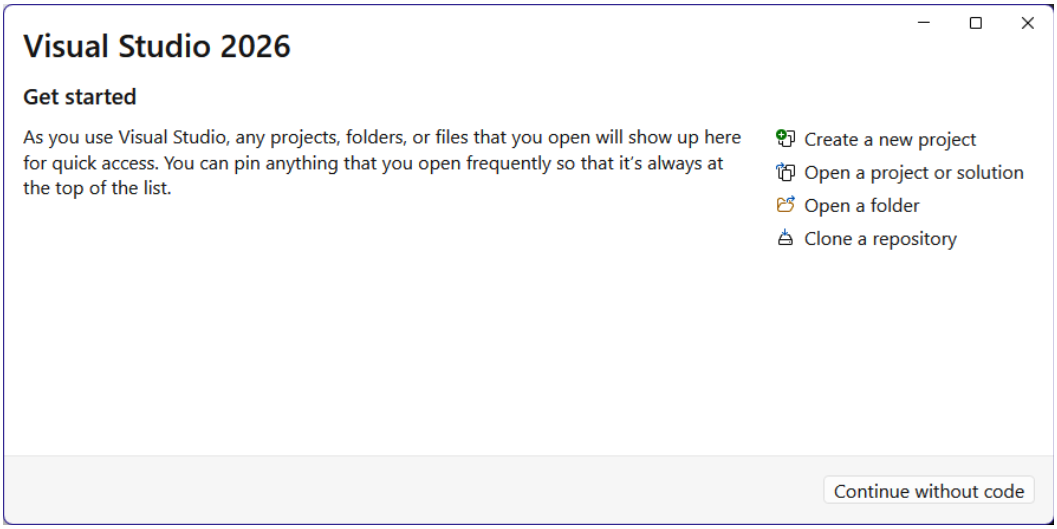


Figure 2.3: Visual Studio startup screen

2. Click **Create a new project**, and in the search bar, type `blazor`.

3. You will get a list of different templates – this may be a mix of different .NET templates. Now we need to select the template for our project. Select **Blazor Web App** from the search results and click **Next**:

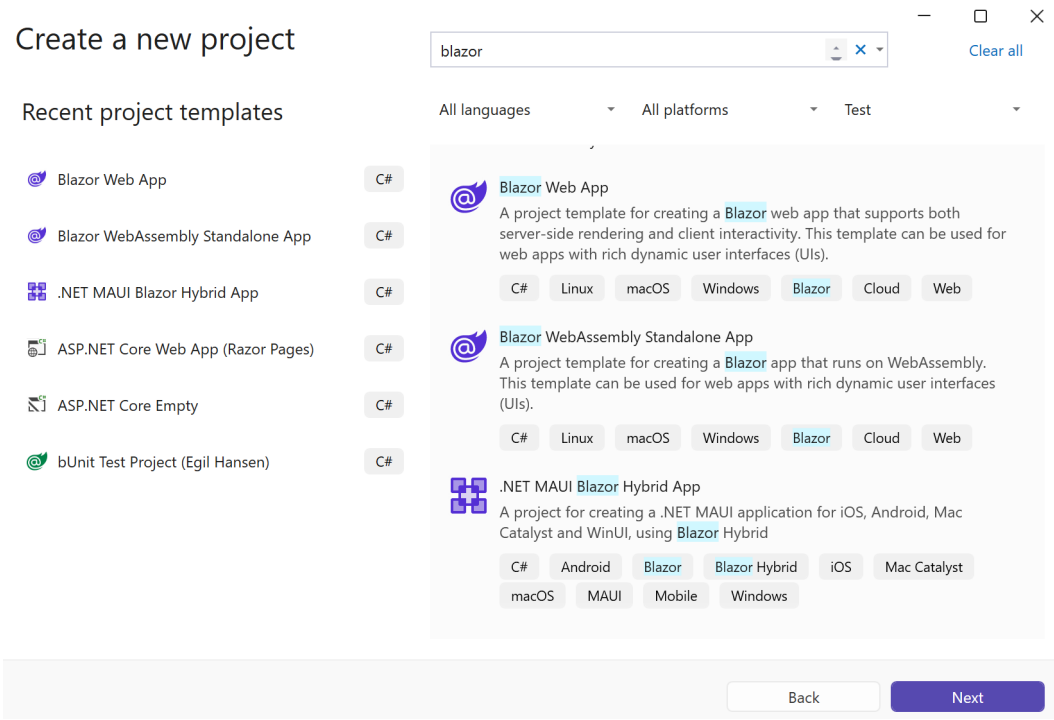


Figure 2.4: The Visual Studio "Create a new project" screen

4. Now it's time to name the project (this is the hardest part of any project, but fear not, I have done that already!). Name the application `BlazorWebApp`. Change the solution name to `MyBlog` and click **Next**.

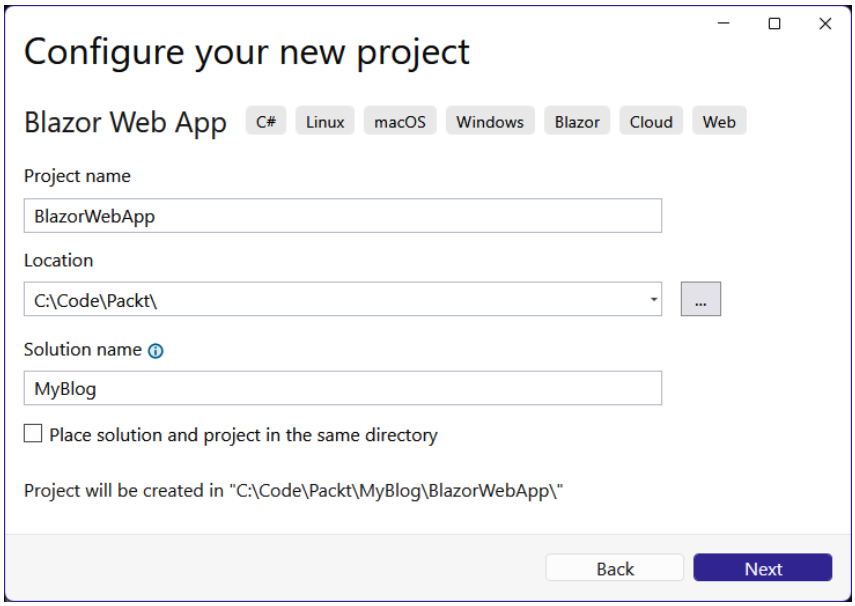


Figure 2.5: The Visual Studio 'Configure your new project' screen

5. Next, choose what kind of Blazor app we should create:
    - Select **.NET 10.0 (Long Term Support)** from the dropdown menu
    - Select **Authentication type** as **None**
    - Check **Configure for HTTPS**
    - Select **Interactive render mode** as **Auto (Server and WebAssembly)**
    - Select **Interactivity location** as **Global**
    - Check **Include sample pages**
    - Check **Use dev.localhost TLD in the application URL**
    - Check **Enlist in Aspire Orchestration** and select the latest version

Then click **Create**:

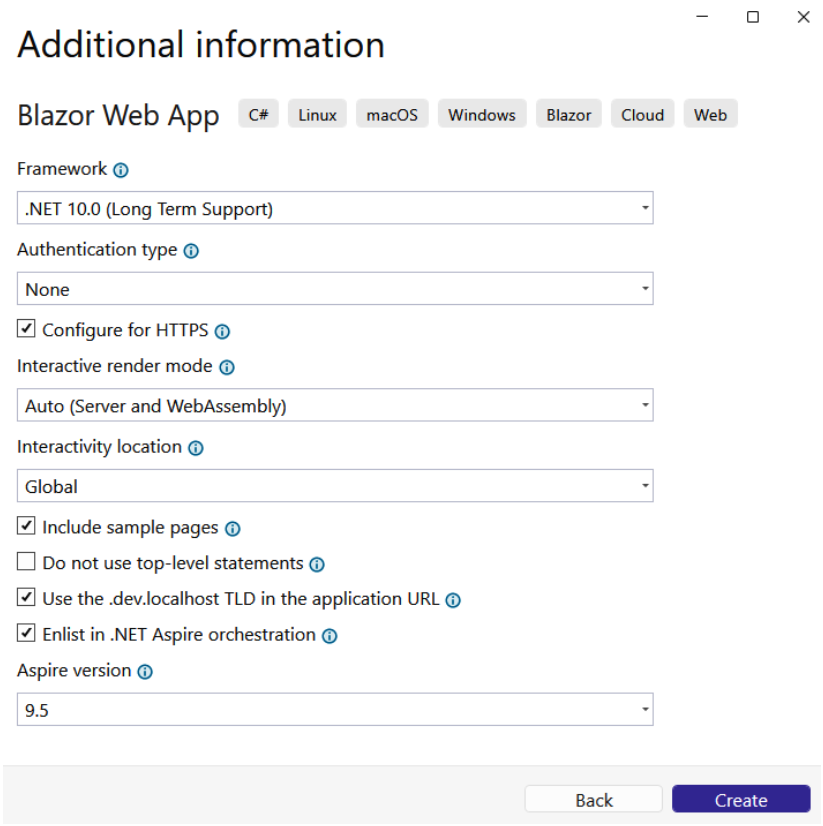


Figure 2.6: Visual Studio screen for creating a new Blazor app

6. Now run the app by pressing *F5* (we can also find it under **Debug | Start debugging**).

Congratulations! You have just created your first Blazor web app. The site should look something like *Figure 2.7*:

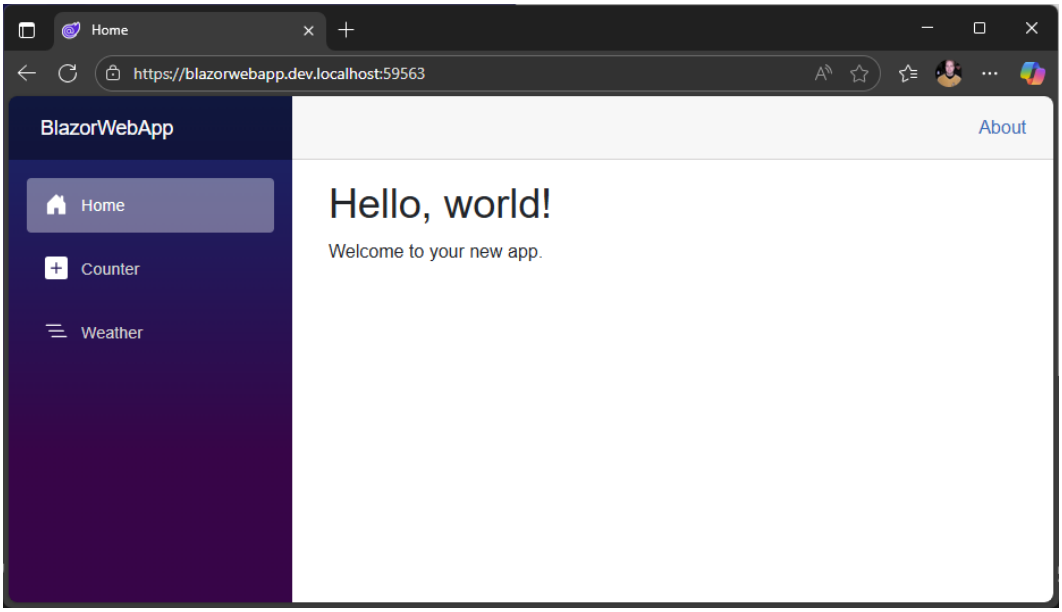


Figure 2.7: A new Blazor web app

This launches Aspire, which orchestrates and starts your Blazor project. Visual Studio will then open two browser windows: one for the Aspire dashboard and one for your Blazor web app.

You may notice that the URL is something like: `https://blazorwebapp.dev.localhost:59563/`. The `dev.localhost` part is because we selected **Use dev.localhost TLD in the application URL** checkbox when creating our project.

But what about Aspire? We will come back to that in *Chapter 4*, *Uncovering Aspire*.

Explore the site a bit, navigate to **Counter** and **Weather** to get a taste of the load times and see what the sample application does.

Visual Studio is not the only way to create a project; next, we will take a look at the command line.

Introducing the .NET CLI

Many other frameworks have a CLI to create projects, add references, and so on, and .NET has one too.

The **.NET Command Line Interface**, or **.NET CLI**, accessed through `dotnet`, is a powerful, cross-platform tool for working with .NET projects. Developers familiar with .NET Core may already know it, but since .NET 5, the CLI is no longer limited to .NET Core scenarios.

The CLI can perform many of the same tasks as Visual Studio, such as creating projects and managing NuGet packages. In the following example, we create a Blazor application using the dotnet command. This project is for demonstration purposes only and is not used later in the book.

To create a Blazor application, run the following command:

```
dotnet new blazor -o BlazorWebApp
```

The .NET CLI works on Windows, Linux, and macOS. While we do not explore it in depth in this book, it is worth knowing that it provides extensive functionality for project creation and management.

> **Note**
>
> The .NET CLI is designed to support a fully command-line-driven workflow. If you prefer working from the command line, see the official documentation at `https://docs.microsoft.com/en-us/dotnet/core/tools/`.

Let's go back to the Blazor template, which has added numerous files for us. In the next section, we will look at what Visual Studio has generated for us.

Figuring out the project structure

Visual Studio will generate two Blazor projects (and two Aspire projects, which we will ignore for now). The Blazor projects are: `BlazorWebApp`, which is the server project, and `BlazorWebApp.Client`, which is where we put our WebAssembly components.

I know we haven't really talked about the different render modes yet, but it is sometimes easier to see it in action first before going down the rabbit hole of what it all really means.

Now, it's time to look at the different files and how they may differ in different projects. Let's take a look at the code in the two projects we just created (in the *Creating our first Blazor app* section) as we go through them.

Program.cs (BlazorWebApp project)

`Program.cs` is the first class that gets called, so let's start looking at that one.

The `Program.cs` file looks like this:

```
using BlazorWebApp.Client.Pages;
using BlazorWebApp.Components;
```

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

builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents()
    .AddInteractiveWebAssemblyComponents();

var app = builder.Build();

app.MapDefaultEndpoints();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseWebAssemblyDebugging();
}
else
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days. You may want to change this
    // for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
app.UseStatusCodePagesWithReExecute("/not-found",
    createScopeForStatusCodePages: true);
app.UseHttpsRedirection();

app.UseAntiforgery();

app.MapStaticAssets();
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(
        BlazorWebApp.Client._Imports).Assembly);

app.Run();
```

In .NET 6, Microsoft removed the `Startup.cs` file and put all the startup code in `Program.cs`. It also uses top-level statements, which makes the code a bit less bloated.

Top-level statements allow us to write code directly in `Program.cs` without needing to wrap it in a `Main` method or a `Program` class.

There are a few things worthy of mentioning here. The first method we call is `AddServiceDefaults`, which comes from Aspire and configures several services for us by default. We will revisit this later. The `Program` class continues adding all the dependencies we need in our application. In this case, we add `RazorComponent`, which enables us to run Razor components. Then, we add `InteractiveServerComponents`, giving us access to all the objects we need to run Blazor Server. Since we selected the auto render mode, we also get access to Blazor `WebAssembly` by adding `InteractiveWebAssemblyComponents.`

The remaining startup code in `Program.cs` also configures **HTTP Strict Transport Security** (**HSTS**), forcing your application to use HTTPS, and makes sure that your users don't use any untrusted resources or certificates. This configuration also ensures that the site redirects HTTP requests to HTTPS for security.

`MapStaticFiles` enables downloading of static files such as CSS or images.

`UseAntiforgery` is a method that adds anti-forgery middleware to the application pipeline, providing a layer of security against **Cross-Site Request Forgery** (**CSRF** or **XSRF**) attacks. These types of attacks occur when a malicious web app influences the interaction between a client browser and a web app that trusts that browser, often leading to unwanted actions being performed without the user's consent.

The different `app.Use*` methods add request delegates to the request pipeline or middleware pipeline. Each request delegate (`ExceptionHandler`, `HttpsRedirection`, `StatusCodePagesWithReExecute`, and so on) is called consecutively from the top to the bottom in the order they are added in `Program.cs` and back again. This is why the exception handler is the first one to be added.

If there is an exception in any of the request delegates that follow, the exception handler will still be able to handle it (since the request travels back through the pipeline), as shown in *Figure 2.8*:

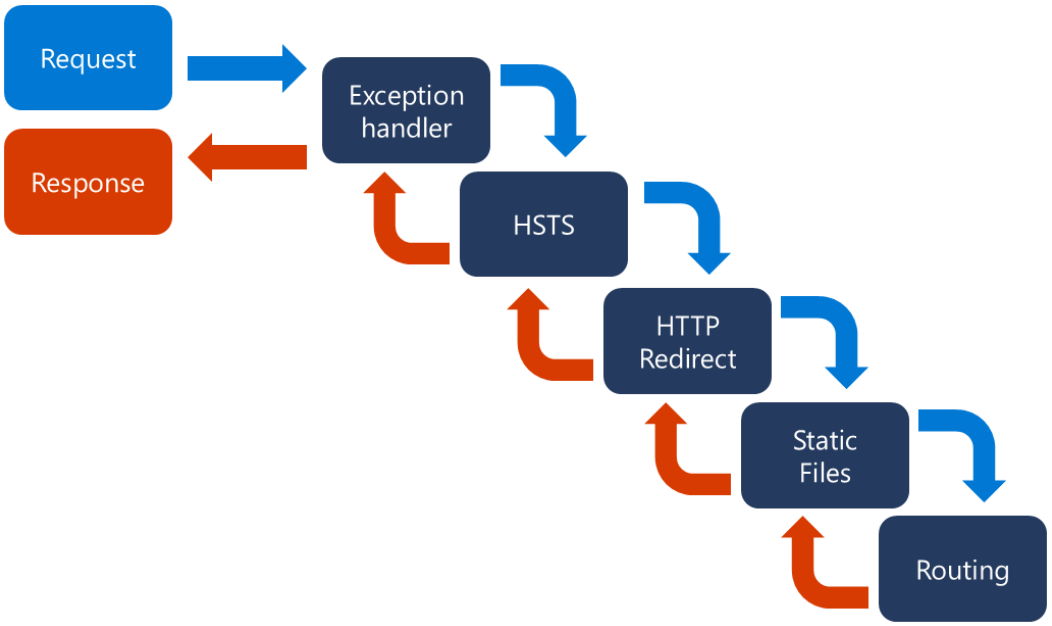


Figure 2.8: The request middleware pipeline

If any of these request delegates handle the request, in the case of a static file, for example, there is no need to involve routing, and the remaining request delegates will not get called. Sometimes, it is essential to add request delegates in the correct order; for example, we want to run authentication early in the pipeline to ensure that users can't access things they shouldn't.

> **Note**
>
> There is more information about the middleware here if you want to dig even deeper: `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0`.

At the end of the `Program` class, we map Razor components to the `App` component. We add the different render modes and additional assemblies – in this case, the `BlazorWebApp.Client` project, which is our WebAssembly project. This way, our server project can identify all the resources available in the client project.

Program.cs (BlazorWebApp.Client)

The `Program.cs` file located in the BlazorWebApp.Client project (the WebAssembly project) doesn't contain much.

It looks like this:

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

var builder = WebAssemblyHostBuilder.CreateDefault(args);

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

It simply sets up a host builder and uses the default configuration.

It is worth noting here that we are not setting up Aspire in this `Program.cs` file.

App (BlazorWebApp)

The next thing that happens is that the `App` component runs.

It looks like this:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0" />
    <base href="/" />
    <ResourcePreloader />
    <link rel="stylesheet"
          href="@Assets["lib/bootstrap/dist/css/bootstrap.min.css"]" />
    <link rel="stylesheet" href="@Assets["app.css"]" />
    <link rel="stylesheet" href="@Assets["BlazorWebApp.styles.css"]" />
    <ImportMap />
    <link rel="icon" type="image/png" href="favicon.png" />
    <HeadOutlet @rendermode="InteractiveAuto" />
</head>

<body>
    <Routes @rendermode="InteractiveAuto" />
    <ReconnectModal />
    <script src="@Assets["_framework/blazor.web.js"]"></script>
</body>
```

```
</html>
```

Let's go through it and see what we can learn. It starts with `html`, `doctype`, and a `head` tag. The `head` tag contains meta tags, **stylesheets** (**CSS**), and a base tag. The `base` tag is so the application finds the appropriate files. If we are, for example, hosting our application in a subfolder (like on GitHub Pages), we need to modify the `base` tag to reflect that.

The `HeadOutlet` component allows us to add elements like page titles directly from within our code (we will revisit this in *Chapter 7*, *Creating Advanced Blazor Components*).

The `Routes` component handles all routing, which we will examine next.

Plus, we have the JavaScript that makes all of this possible.

In .NET 9 and .NET 10, we have a couple of new additions in this file.

First, we have `@Assets`, which adds a hash to all files, allowing them to be cached for a longer period. When the content of the files changes, so does the hash.

We also have `ImportMap`, which ensures that the URL for JavaScript files is redirected to a hashed version. `ResourcePreloader` will preload resources, allowing us to load our site even faster. This means that before we start up WebAssembly, the download of resources can start. The start time of a Blazor (WebAssembly) site has improved significantly in .NET 10, thanks to improvements in hashing, compression, and preloading, among other enhancements.

We will go deeper into these things in the next chapter, *Chapter 3*, *Exploring Render Modes*.

Routes

The `Routes` component is the one that handles all the routing. It looks like this:

```
<Router AppAssembly="typeof(Program).Assembly"
        NotFoundPage="typeof(Pages.NotFound)">
    <Found Context="routeData">
        <RouteView RouteData="routeData"
                   DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

This file handles routing, selecting which component to render on the page based on the current URL (using the `@page` directive). It shows an error message if the route can't be found.

In *Chapter 10*, *Adding Authentication and Authorization*, we will make changes to this file when we implement authentication.

The `Routes` component also includes a default layout. We can override the layout per component, but usually, you'll have one layout page for your site. In this case, the default layout is called `MainLayout`.

It also specifies a `NotFound` component, which can be found in the BlazorWebApp project in the `BlazorWebApp.Client\Pages\NotFound.razor` file.

`FocusOnNavigate` will set the focus on a specific element once we have navigated or loaded a new component or route – in this case, `h1`.

Main layout

`MainLayout`, which we can find in the `Layout` folder, contains the default layout for all components when viewed as a page. The `MainLayout` contains a couple of `div` tags, one for the sidebar and one for the main content:

```
@inherits LayoutComponentBase

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

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

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

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

```
    <span class="dismiss">✖ </span>
</div>
```

The only things you need in this document are `@inherits LayoutComponentBase` and `@Body`; the rest is just layout using Bootstrap.

> **Note**
>
> **Bootstrap** is one of the most popular CSS frameworks for developing responsive and mobile-first websites. We can find a reference to Bootstrap in the `App.Razor` file. It was created by and for *Twitter/X*. You can read more about Bootstrap here: `https://getbootstrap.com/`.

The `@inherits` directive inherits from `LayoutComponentBase`, which contains all the code to use a layout. `@Body` is where the component will be rendered (when viewed as a page).

The `@Body` is a `RenderFragment`, which we will return to in *Chapter 7*, *Creating Advanced Blazor Components*.

The `MainLayout` also contains the `default` error UI for Blazor.

NavMenu

At the top of the layout, you can see `<NavMenu>`, a Razor component. It is located in the `Layout` folder and looks like this:

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

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

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

```
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="counter">
                <span class="bi bi-plus-square-fill-nav-menu"
                      aria-hidden="true"></span> Counter
            </NavLink>
        </div>

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

It contains the left-side menu and is a standard Bootstrap menu. It also has three menu items and logic for a hamburger menu (if viewed on a phone).

You will find another component, `NavLink`, which is built into the framework. It will render an anchor tag, but will also check the current route. If you are currently on the same route or URL as the `NavLink`, it will automatically add a CSS class called `active` to the tag.

We will run into a couple more built-in components that will help us along the way. There are also some pages in the template, but we will leave them for now and go through them in *Chapter 6*, *Understanding Basic Blazor Components*.

CSS

In the `Layout` folder, there are three CSS files: `NavMenu.razor.css`, `MainLayout.razor.css`, and `ReconnectModal.razor.css`. These files are CSS styles that affect only the specific component (the first part of the name). We will return to a concept called isolated CSS in *Chapter 11*, *Sharing Code and Resources*.

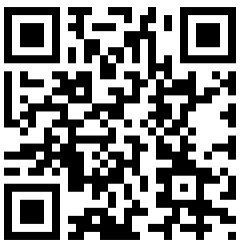
Summary

In this chapter, we set up the development environment and created our first Blazor app. We also learned about the order in which classes, components, and layouts are called, which makes it easier to follow the code.

In the next chapter, it is time to take a look at the different render modes.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

3

Exploring Render Modes

In this chapter, we'll look at render modes. Before .NET 8, this used to be very simple: we had two, **Blazor Server** and **Blazor WebAssembly**. But more importantly, we had one or the other, never both. With .NET 8, things changed. We got more combinations, which means greater complexity, but also greater power and flexibility.

At the same time, we're seeing a trend across frontend frameworks. Many of them are moving back to server rendering in some form. For example, React has Next.js, and Angular has Angular Universal or **Server-Side Rendering** (**SSR**). This is mostly about improving things like SEO and page load time. Instead of waiting for a full client app to boot up, the user gets a pre-rendered page from the server, and then the client app takes over.

Blazor is following the same idea. With the new render modes introduced in .NET 8, we can choose how our components are rendered: on the server, on the client, or a mix of both. This gives us better control and can improve both performance and user experience.

In this chapter, we will begin by breaking down the different ways Blazor can render applications, starting with server-side rendering and exploring static and streaming SSR using a dedicated test project. Then we will look into the interactive render modes, including Blazor Server, Blazor WebAssembly, and the Auto mode, comparing how they behave, their trade-offs, and their real-world use cases. By the end of this chapter, you will have a solid understanding of how each render mode works and how to choose the right approach for performance, interactivity, and scalability.

In this chapter, we will cover the following topics:

- Running the WebAppTest project
- Understanding Server-Side Rendering (SSR)
- Delving into the Blazor Server / Interactive Server

- Exploring Blazor WebAssembly / InteractiveWebAssembly
- Understanding Auto / Interactive Auto
- Understanding where to put interactivity

Technical requirements

You can find the source code for this chapter at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter03`.

Running the WebAppTest project

I have created a test project that enables us to switch between render modes. This way, we can see the render modes in action and make an informed decision.

From this chapter's GitHub folder, you can find the `WebAppTest` project. This is a test project, so don't concentrate on the code – unlike our project, switching between render modes is not a normal thing to do.

Follow these steps to run the project:

1. Clone/download the repo: `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter03`.
2. Open the `WebAppTest.sln` solution.
3. Press *F5* to run the project. We will use it as a reference as we continue through this chapter. Don't worry if you don't have a computer nearby; we will also cover the expected outcome.

As I mentioned earlier, this is a test project – when the weather data is loading, there is a 5-second delay so we can actually see what happens when the page loads.

Note

I have heard many comments about Blazor being WebAssembly; however, WebAssembly is just one way to run Blazor. As we will explore, there are many ways of rendering Blazor. This is where Blazor becomes interesting. It's not just that multiple rendering approaches exist; other frameworks can do that as well. The difference is that in Blazor, the same component model and the same C# code can move between these modes. That means we can choose the right rendering strategy for each scenario without rebuilding the application or switching stacks.

Now, we will start from the beginning and test each render mode throughout this chapter. Let's begin!

Understanding Server-Side Rendering (SSR)

Server-Side Rendering (**SSR**) is the most straightforward approach for rendering a Blazor app. The server renders your components and sends the finished HTML to the browser. You can think of it as a one-time event: the browser requests a page, the server sends it back, and no Blazor runs on the client, so no interactivity occurs after that.

Blazor supports two kinds of SSR: **static** and **streaming**. Static SSR renders the whole page before sending it. Streaming SSR can send the page in chunks, so parts of the page appear as they become ready. This can make the app feel faster, especially when loading certain data takes longer.

Although SSR renders everything on the server, Blazor still provides some nice extras to enhance the experience. One of those is **enhanced navigation**. Instead of performing a full-page reload on every navigation, Blazor only updates the parts that have changed. It can also keep things like scroll position and form values, so it feels much closer to a **Single Page Application** (**SPA**) than a classic server-rendered app.

Static SSR

Static SSR is, well, static. The page is rendered once on the server and sent to the browser as plain HTML. There is no Blazor running on the client and there is no interactivity.

Pre-rendering was turned on by default in .NET 8. Static SSR is basically that pre-rendering step. With static SSR, the server:

1. Gets a request from the browser
2. Renders the components and performs any required data work (such as database calls)
3. Sends one complete HTML page back

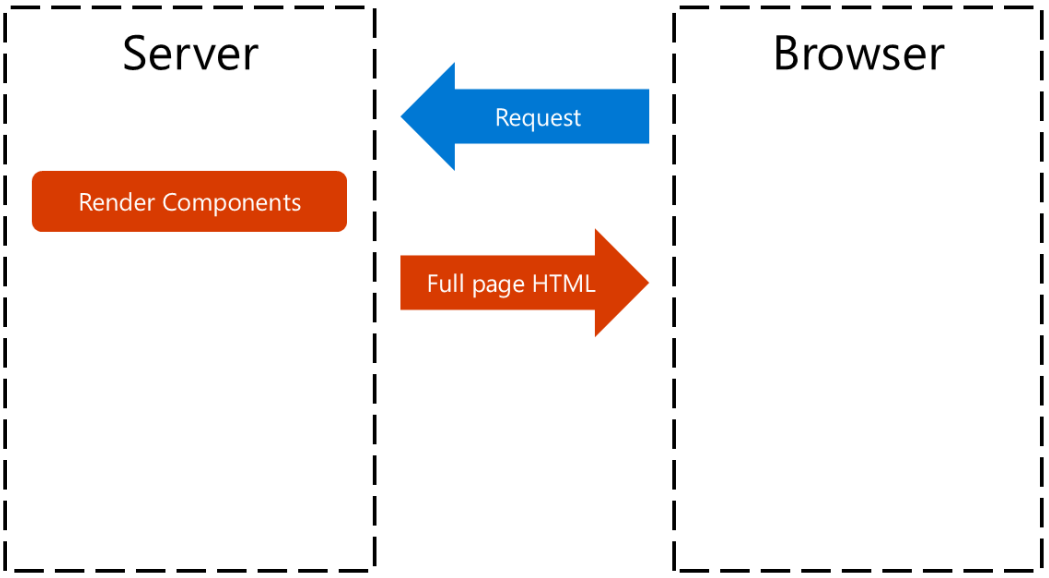


Figure 3.1: The flow of static SSR

With this method, nothing is streamed in parts. We see the page only when everything is ready.

Even though the page is static, Blazor still provides useful helpers. As mentioned earlier, enhanced navigation makes the app feel a bit like an SPA. Instead of reloading the whole page, Blazor only replaces the parts that have changed and can keep things like scroll position and form values.

There is still no client-side Blazor running, but we can post forms back to the server. That provides us with a bit of interactivity, similar to classic Razor Pages or **Model-View-Controller** (**MVC**), but with a more pleasant navigation experience. That said, not all pages need rich interactivity. For many pages, simple server rendering is more than enough.

Static SSR also helps with **SEO** (short for **search engine optimization**). One of the big problems with modern client-side frameworks (not just Blazor) is that they render most content in the browser. If the HTML is almost empty on initial load, search engines struggle to read it. That is why server-side rendering has found its way into Angular (Angular Universal/ SSR) and React (Next.js).

So, this render mode solves three big problems:

- **SEO**: Search engines get real HTML with content
- **Faster initial load**: The browser gets ready-to-show HTML, no app startup needed
- **No extra client cost for simple pages**: Pages that don't need interactivity don't need to load the Blazor app in the browser

For many apps, a significant number of pages fall into the "static is enough" category, and static SSR is a great fit.

Testing static SSR

Let's see static SSR in action:

1. Start the `WebAppTest` project if it is not already running by pressing *F5*.
   This time, we won't go through the project in detail – the code is not important; what matters is the user experience.
2. Click **Change render mode**, and then click **SSR**.
3. Go to the **Counter** page, and start clicking the **Click me** button.
   Nothing is happening. No reload, no increments. This is because we don't have any interactivity.
4. Now, click the **Weather** link.
   At first, nothing happens. You will likely notice that it takes some time for the data to appear. This is because there is a delay when getting data (to simulate a slow database or API). The page waits for 5 seconds, then the data starts to load.

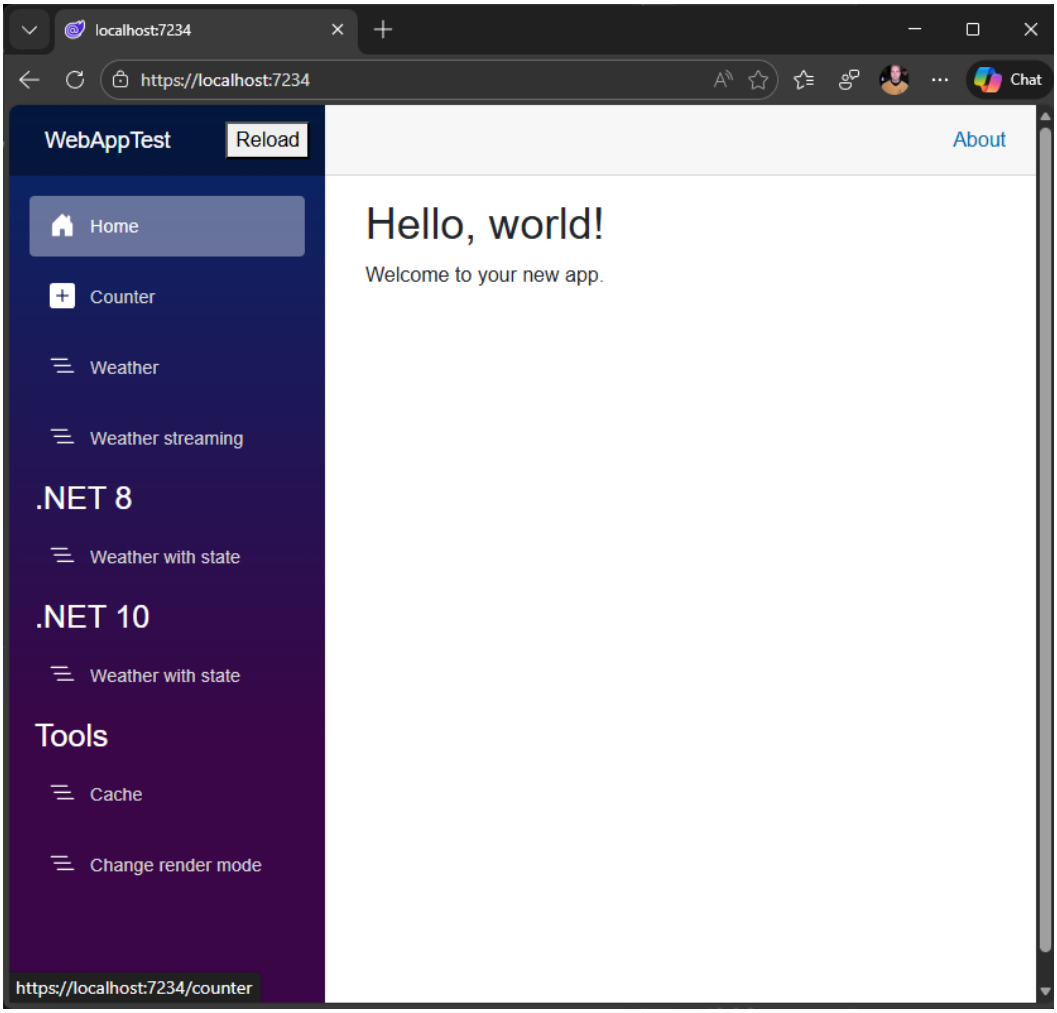


Figure 3.2: Test application

I have taken this example to the limit; 5 seconds is a really long time, but it is to show the huge difference in the user experience between static SSR and what we will look at next, which is Streaming SSR.

Streaming SSR

Streaming SSR isn't a render mode on its own; it's static SSR with an attribute.

With static SSR, the server waits until everything is ready before it sends the HTML to the browser. If one part of the page is slow (for example, it calls an API or a database), the whole page is delayed. The user just waits.

Streaming rendering changes that.

With streaming SSR, Blazor sends the page in chunks. It can send the layout and fast-rendering components first, and then fill in the slower parts later when they are ready. So, instead of showing a blank screen, the user sees content almost right away.

We turn this on per component by adding the following `StreamRendering` attribute:

```
@attribute [StreamRendering]
```

This doesn't introduce a new render mode. Instead, it enables streaming behavior for the component when using SSR, allowing its content to be sent to the browser as soon as it's ready rather than waiting for the entire page to finish rendering.

For example, the page might first render the title and a summary, then load the details once they're available. The result is a much faster and smoother user experience.

Here's a diagram to help explain how that works:

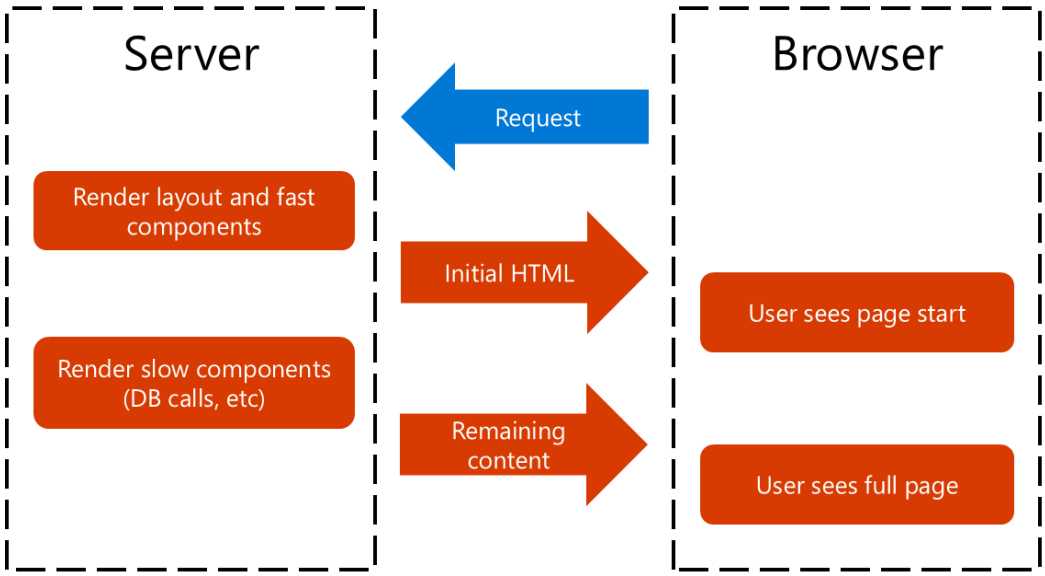


Figure 3.3: The flow of streaming SSR

Streaming also works with all the interactive modes, including Blazor Server, Blazor WebAssembly, and InteractiveAuto. However, it's worth noting that streaming is not limited to interactive scenarios. It works with pure SSR as well, which means we can use it on non-interactive, SEO-friendly pages to improve perceived load time.

When combined with interactive modes, we can stream the initial HTML from the server and then enable interactivity once the chosen render mode kicks in. This gives us the best of both worlds:

- Fast initial load (content on the page) thanks to SSR
- Rich interactivity once the app becomes interactive

This is a big step up from the old *blank page while the app boots* problem that we've seen with pure SPA apps.

Testing streaming SSR

The test application covers us as well. Continuing with the previous example, let's have a look at the **Weather** page again:

1. Click the **Weather** link, and click **Reload**. It will take 5 seconds for anything to happen. The app feels sluggish.
2. Now, click the **Weather streaming** link, and click **Reload**.
   Notice how we get new content on the screen almost instantly. Five seconds later, the data updates. The page re-renders whenever new data is available, so we're getting a piece-by-piece loading of the page. It's the same five seconds, but it no longer feels sluggish and slow. It now feels so much faster. Granted, five seconds is still a long time, but that has nothing to do with Blazor; this is all our fault because we have a delay in the data generation.

Enhanced navigation

Before we move on, we need to talk about something that isn't a render mode but still plays a big part in how SSR feels: enhanced navigation. As mentioned before, this plays a big role in SSR. When you run Blazor in pure SSR, the browser will normally do a full-page reload on every navigation, just like a classic server-rendered site. That works fine, but it doesn't feel great. It's a bit old web. The screen goes blank, everything reloads, and any scroll position or form values are lost along the way.

Enhanced navigation fixes that.

With enhanced navigation turned on (which it is by default), Blazor only replaces the parts of the page that have actually changed. It hooks into regular link clicks and form submissions and handles them in a smarter way than a normal full-page refresh. The result is that even static SSR can feel a lot closer to a modern SPA:

- Navigation doesn't blink the whole page
- Scroll position is kept
- Form values can be kept
- Only the content that changed is fetched

There's no Blazor running on the client, no SignalR, no WebAssembly, but the experience is noticeably smoother. You're still doing server rendering, but without the clunky page reloads we're used to from the old days. This is especially nice on pages that don't need interactivity. You get the best parts of server rendering (fast initial load, good SEO, simple hosting), but navigation feels much more app-like.

It's worth pointing out that enhanced navigation is not something we'll go into in depth in this book. You won't need to configure it for the scenarios we're building, but it's good to know what it is when you see it mentioned in docs or samples.

In short, enhanced navigation gives SSR a nice coat of polish so it behaves more like a modern web app, while still keeping the simplicity of server rendering.

This is pretty cool, but the book's subtitle includes "*interactive*," so where is the interactivity? Funny you should ask; that is what we will cover in the next section.

Delving into the Blazor Server / Interactive Server

Now, let's talk about the **Interactive Server** (or **Blazor Server**, as it used to be called).

Blazor Server runs your components on the server and uses SignalR to communicate with the browser. Instead of sending full pages, it maintains a live connection and only sends UI updates:

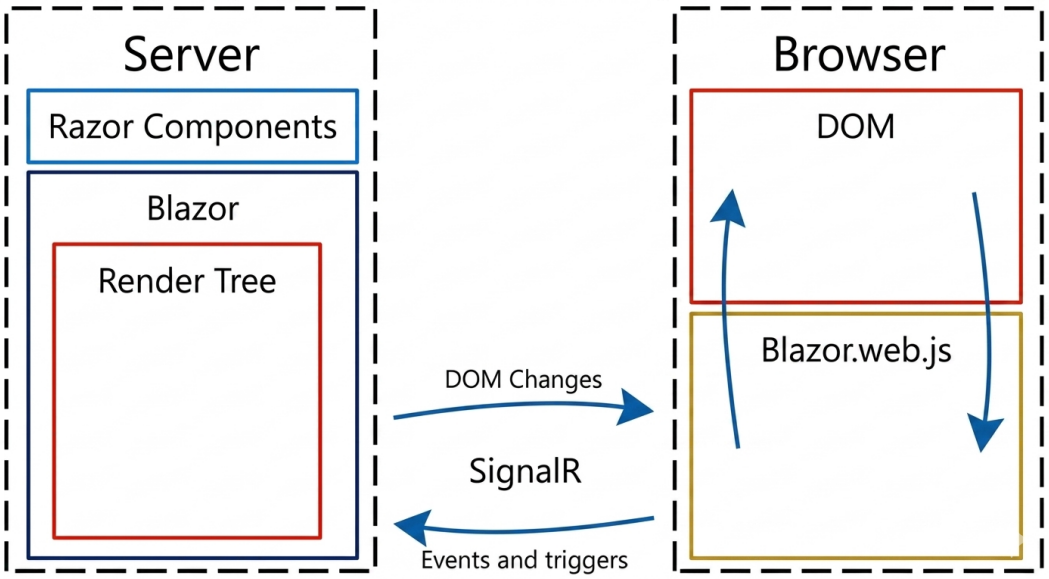


Figure 3.4: Overview of Blazor Server

SignalR is a real-time communication library that selects the best available connection method. It tries to use WebSockets first, and if that's not possible, it falls back to other protocols. This connection enables Blazor Server to keep the app running on the server while updating the UI in the browser as needed.

Each Blazor component runs on the server and is written using Razor, a mix of HTML and C#. Components can be nested and reused just like in other frameworks.

When something changes, for example, when a button is clicked or a timer fires, Blazor re-renders the component and builds a render tree, a lightweight representation of the UI. It then compares this to the previous render to figure out exactly what has changed.

Those changes are sent over SignalR to the browser using a compact binary format. On the client side, a bit of JavaScript takes care of applying those updates to the actual DOM.

If you have worked with traditional ASP.NET, you will notice this is very different. We're not reloading the whole page or sending full HTML responses. Blazor Server just sends the small pieces that have changed.

Benefits of Blazor Server

There are quite a few upsides to this render mode:

- **Tiny download size**: Only the bare minimum is sent to the client, which means the page loads quickly.

- **Full server access**: We can call services and databases directly without needing an API.
- **Secure**: The code stays on the server, so it can't be decompiled or viewed by the client.
- **Centralized logic**: Everything runs on the server, so we don't have to worry about syncing logic between the client and server.

Downsides of Blazor Server

Of course, there are trade-offs:

- **Requires a constant connection**: If the SignalR connection drops, the app stops working. In .NET 10, this has improved somewhat, but it is still mostly true.
- **No offline support**: This rules out PWA functionality, where we could have our app installed and use it offline.
- **Every interaction is a round trip**: Although only the changes are sent, each click still goes back to the server.
- **More load on the server**: Each user has a connection and state stored in memory.
- **Scaling is harder**: You'll need to plan for load balancing and memory usage, especially as the number of users grows.
- **Needs ASP.NET Core hosting**: You can't host this on a static site or without server-side support.

If you want to scale it out, you can offload the SignalR connections to Azure SignalR Service, which handles all the real-time connections for you.

Testing the render mode

Let's test this out:

1. Start the `WebAppTest` project if it is not already running by pressing *F5*.
2. Click Change render mode, then click Server Prerendered.
3. Go to the **Counter** page and start clicking on the Click me button.
   The change is instant; the counter counts up. It makes calls over SignalR to the server, which re-renders the content, compares it to the render tree, and sends the changes over the wire.
4. Now click the **Reload** button, and you might be able to see the render mode change from **Prerendered** to **Server** (where "Server" means SignalR).
5. Click Weather, and the page loads fast, but the data arrives after quite some time. This is because of the 5-second delay.
6. Click the **Reload** button.

> The page is first prerendered on the server while it waits for the data, and the HTML is sent to the browser. Once the app becomes interactive, it switches over to SignalR. At that point, the content is rendered again. This happens because there is no handoff between prerendering and interactive server mode, even though both run on the server.

We will explore ways to address this later in the book. In a real-world scenario, our APIs hopefully don't take 5 seconds to complete, so don't worry about performance.

Real-world usage

At my former workplace, we had a large existing ASP.NET MVC app with both a customer portal and an internal CRM. We wanted to modernize it, and Blazor Server was a perfect fit since we already had the server-side infrastructure in place.

We started by migrating one component at a time. In most cases, it proved faster to rebuild the component in Blazor than to continually add features to the old MVC version. The UX also received a nice upgrade – pages loaded faster, and we could refresh parts of the UI without affecting the rest.

One fun side effect we didn't expect: users thought nothing had happened when they clicked **Save**, just because it was so fast. So, we had to go back and add visual feedback, like a toast or a checkmark, to show that things worked. A good problem to have!

The Interactive Server is really easy to get started with, especially if you are coming from a WebForms/MVC/Razor Pages world. It does have its challenges when it comes to scaling. If you know roughly how many users your system will have, this can be a great option. It is especially well-suited for internal tools or applications where you control the user base.

For scenarios with larger or more unpredictable traffic, we may need to rethink where the work is done and how the app scales. This is where Blazor WebAssembly comes into play, which happens to be the next render mode we will look at.

Exploring Blazor WebAssembly / Interactive WebAssembly

Blazor WebAssembly lets us run .NET code directly in the browser. Instead of relying on a server to render everything, the browser runs the app locally using WebAssembly.

It will, by default, prerender the content, send that to the web browser, and then start to load WebAssembly:

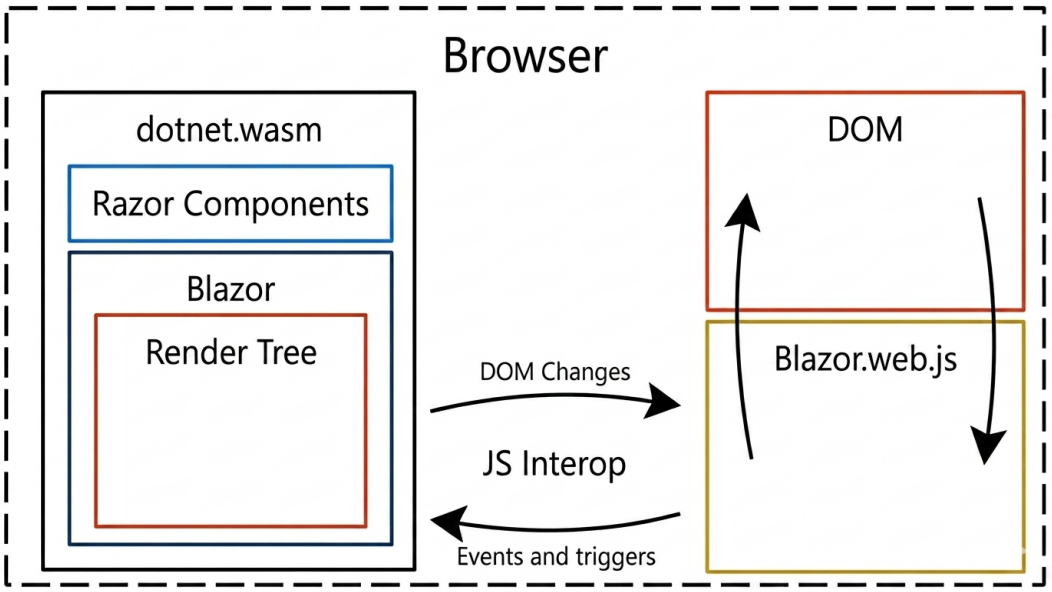


Figure 3.5: Overview of Blazor WebAssembly

Blazor uses a version of the **Mono runtime** that has been compiled into WebAssembly, which is what makes it possible to run .NET DLLs in the browser. When the app starts, the runtime is downloaded along with your application's assemblies. Then the app runs directly inside the browser without needing a constant server connection.

Thanks to caching, compression, and preloading, the previous challenges with Blazor being slow to start up are no longer an issue in production. However, note that when running in development (Debug), the startup can still feel slow because the app is unoptimized, so don't be surprised if your local experience does feel a bit sluggish – it will run faster when published.

> **Note**
>
> In older versions of Blazor WebAssembly (like .NET 9 and earlier), the app relied on a file called `blazor.boot.json` to describe what needed to be downloaded. That file is no longer used in .NET 10.

Just like with Blazor Server, components still build up a render tree to track changes. But instead of sending those changes over SignalR, Blazor WebAssembly applies them directly in the browser using JavaScript interop.

Benefits of Blazor WebAssembly

Running everything in the browser comes with some real advantages:

- **Works offline / PWA support**: The app can keep running even without a network connection.
- **No round trips**: Since the logic runs in the browser, everything feels more responsive.
- **Backend agnostic**: You don't need a .NET-compatible server. You can host it on a static file server or **CDN** (**content delivery network**).
- **No load on your server**: The browser does the work, which helps if you have many users and limited backend resources.

Downsides of Blazor WebAssembly

Still, there are a few things to be aware of:

- **Larger initial download**: The runtime and your app's DLLs need to be downloaded before the app starts.
- **Slower startup**: Especially noticeable on mobile or slow connections.
- **No direct server access**: You'll need to expose APIs for any backend work, such as database access.
- **Code is public**: Just like JavaScript, everything is running in the browser, which means it can be decompiled.

Testing the render mode

Let's have a look at WebAssembly in action:

1. Start the `WebAppTest` project if it is not already running by pressing *F5*.
2. Click **Change render** mode, and then click **Wasm Prerendered**.
3. Go to the **Counter** page, click **Reload**, and start clicking on the **Click me** button.
   You may notice that while it says **Render mode: Prerendered**, the button won't work. It is currently in SSR mode, and WebAssembly is loading in the background.

   Once WebAssembly has been loaded, the button becomes active. Keep in mind that we are running in Debug with unoptimized code.
4. Click on **Weather**, then reload. You will see the same behavior as with the Interactive Server: a 5-second delay on the prerendered content, followed by an empty page and the data being regenerated. This is for the same reason as in the Blazor Server example: there is no handover between prerendered and client-side WebAssembly.

> We will come back to this later in the book. In .NET 10, this is improved with better support for preserving state during the transition, so the app doesn't need to reload and re-render everything.

Real-world notes

WebAssembly can feel magical the first time you see your .NET code running inside the browser with no server behind it. That said, startup time can be an issue, especially for larger apps. We'll cover how to improve that using trimming, lazy loading, and AOT compilation in *Chapter 19*, *Going Deeper into WebAssembly*.

You might be wondering: should we choose WebAssembly or Server? We don't have to. This is where the Auto render modes come in.

Understanding Auto / Interactive Auto

Interactive Server loads fast and becomes interactive, but adds extra load to the server and requires a constant connection. Meanwhile, Interactive WebAssembly requires a larger initial download. But what if we could get the best of both worlds?

If we choose the **Auto render mode**, Blazor will first connect using SignalR, allowing for a fast load and enabling you to be up and running with interactivity quickly. In the background, WebAssembly will be downloaded and cached. The next time our user visits our site, Blazor will load the cached WebAssembly files instead. This will remove that whole initial download.

Blazor will only use the Interactive Server if the download takes longer than 100 ms, which is usually hard to emulate when running with a debugger.

In our app, we intentionally added a 5-second delay to make the differences between render modes more noticeable. I find this useful when creating videos, courses, and presentations, but it doesn't reflect a real application. To get a more realistic experience, we can remove the delay:

1. In the `WebAppTest.Client` project, open the `Services/WeatherService.cs` file. Near the top, you'll find the `AddDelay` property.
2. Change the `AddDelay` property to `false`:

```
public bool AddDelay { get; set; } = false;
```

3. Now run the app again and try the same scenarios as before.

Everything suddenly feels much faster. And this is still Debug, with unoptimized DLLs. The differences between render modes are still there, but they're no longer exaggerated by an artificial delay. This is much closer to what you'll see in a real application.

Now that we know more about the different render modes, let's take a look at where we can add interactivity.

Understanding where to put interactivity

You might remember that in the *Creating a Blazor web application* section in *Chapter 2*, *Creating Your First Blazor App*, we selected **Global** as the interactivity location. There is one other interactivity option available. Let's take a closer look at both.

Global

Global means the whole app uses the same render mode. You set it once, and all components follow that mode unless you specifically opt out. The nice part is that this is just an attribute on a component.

In our app, we've gone with global interactivity. If you open `Components/App.razor` in the `BlazorWebApp` project, you'll find this line:

```
<Routes @rendermode="InteractiveAuto" />
```

That `@rendermode` attribute sets the *global* render mode. If you remove it, your app will fall back to static SSR.

So how does this work?

It comes down to where the `@rendermode` is set. When it's applied high up, like on the root component (the Routes component), it becomes global. Every component in the app will use the same render mode.

Per component (or page)

If you want more control, you can skip setting the render mode globally and instead define it *per component*. This is also done using an attribute, either when you use the component or directly in the component itself:

```
@rendermode InteractiveServer
```

Note

You can't override the render mode deeper in the hierarchy. If it's set on the root component, that mode applies to everything below it. So, if you want to control interactivity per component, make sure you don't set it globally. We'll revisit this a few more times throughout the book.

What should you choose?

If your app needs interactivity, and most apps do, start with global interactivity. It's simple and covers everything. Later, if you find certain pages don't need to be interactive, you can opt them out by using:

```
@attribute [ExcludeFromInteractiveRouting]
```

That way, you still get fast static rendering where it makes sense without sacrificing interactivity across the rest of the app.

Mixing render modes adds complexity: you can't share state between a component rendered in the browser and one rendered on the server. Although it is possible, I recommend using global mode rather than per-component mode.

Summary

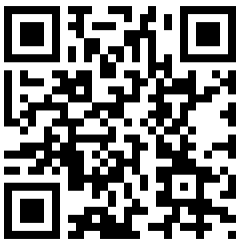
In this chapter, we explored how Blazor renders applications, starting with static SSR and streaming SSR, where the server sends HTML to the browser for fast load times and good SEO. We then looked at the interactive render modes: Interactive Server, which keeps logic on the server using SignalR; Interactive WebAssembly, which runs .NET directly in the browser; and Interactive Auto, which combines the two to get fast startup and better long-term performance.

We also learned how render modes affect user experience, performance, and scalability, and how choosing between global and per-component interactivity lets us balance simplicity with flexibility. Understanding these trade-offs is key to building Blazor apps that are both fast and interactive.

In the next chapter, we will explore Aspire.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

4

Uncovering Aspire

In this chapter, we will uncover one of Microsoft's latest additions: **Aspire**. Aspire helps us set up our project, run it locally, and even deploy it. This is what Microsoft considers the best way to develop web applications. I am personally not a fan of Docker or running virtual machines, but with Aspire, things become so much easier.

In this chapter, we'll look at how Aspire wires projects together, how services find and talk to each other, and how the built-in dashboard, logs, and traces help us understand what's going on in the system. The focus in this book is on the local developer experience. As such, we will cover the following topics in the chapter:

- What is Aspire?
- Why use Aspire?
- Project structure
- Unraveling the Aspire Community Toolkit
- Exploring the Aspire dashboard
- Understanding logs, traces, and metrics

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter03` folder on GitHub as the starting point.

You can find the source code for this chapter at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter04`.

What is Aspire?

This is one of those questions where the short answers don't really help. If you search online, you'll find a bunch of one-liners like:

"Aspire is an opinionated cloud stack for .NET."

"Aspire helps you describe your infrastructure."

"Aspire makes your app cloud-ready."

All of these are true, but none of them tell you what Aspire actually is.

Let's take a more practical approach.

Aspire is a way for us to define everything our app needs, using C#. If you have a small app, this might only be your Blazor client and your API. But for bigger systems, this could include background services, message queues, databases, blob storage, and whatever else the app depends on. There is also now an option to use Aspire with TypeScript instead of C#, so developers who use TypeScript can also take advantage of Aspire.

Normally, all of this stuff takes time to set up: ports to configure, connection strings to copy, secrets to add, services to install, and so on. If you work in a company with a large system, you know exactly what I mean.

I still remember starting at my current job. It took me three days just to get the app running. I had no idea what half the scripts did. Some services ran locally, some ran on other machines, and the instructions had changed so many times that no one really knew which version was correct anymore. On top of that, the order in which I started things mattered.

Many companies operate this way, and there is nothing wrong with it. But imagine cloning a repo, pressing *F5*, and everything just works. Aspire spins up all services for you, handles ports, handles references between services, and keeps everything in sync so you can start coding instead of troubleshooting setup steps.

That's what Aspire brings to the table.

It's worth mentioning that Aspire moves fast. By the time I save this file, there's probably a new update out with more improvements. And that's a good thing! Aspire is getting better almost every week.

Aspire is a fairly new Microsoft project, and the dashboard itself is built using Blazor. I have to admit, it's pretty fun to see Blazor show up in something like this.

Now that we have a rough idea of what Aspire is, let's look at why we would use it.

Why use Aspire?

Let's be honest, setting up a full project locally is often a mess. You've got:

- A frontend that needs the API running
- An API that needs a database
- A background worker that needs a queue
- Secrets that nobody documented
- Ports that change depending on who committed last

And when something breaks? Good luck figuring out if it's a config issue, a missing tool, or just the wrong version of Redis.

This is where Aspire makes a big difference.

With Aspire, your setup becomes part of the project. You define all the services and how they connect in code, in C#, and Aspire handles the rest. It wires everything together for you. It sets the right environment variables. It picks free ports. It knows what depends on what and starts things in the right order.

That means no more:

- Manual steps
- README files with 15 setup instructions
- "It works on my machine" problems

Here's what it looks like. This is the `AppHost` project and the `AppHost.cs` file:

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

builder.AddProject<Projects.BlazorWebApp>("blazorwebapp");

builder.Build().Run();
```

That's it. Aspire sees this and spins everything up. You can press *F5* and get to work. It even shows you all running services in a dashboard, with logs and health checks baked in. To be fair, though, in our example so far, it is only spinning up one single project, so it's not that fancy.

We won't go super deep into Aspire, but since it's now the preferred way to structure modern .NET apps, we'll absolutely be using it. For teams, this is a big win. New developers can clone the repo and get going right away. No one needs to spend a day helping someone install a bunch of software, start SQL Server, or figure out why `localhost:5001` isn't responding.

In *Chapter 2*, *Creating Your First Blazor App*, we installed Docker, which can handle spinning up any software we need. This can include PostgreSQL, which is what we will use here, as well as Azure services, RavenDB, or almost anything else that can run in a Docker container.

Aspire builds on top of this. It uses Docker's ability to run these services but removes the manual work of wiring everything together.

This is the power of Aspire. Aspire doesn't just save time; it makes your dev setup repeatable, clean, and *transparent*.

So, to answer the question "why use Aspire?" Because working on a .NET app shouldn't feel like trying to assemble IKEA furniture without the manual.

In the next section, let's take a look at the project structure.

Project structure

As you have seen from *Chapter 2*, *Creating Your First Blazor App*, when you create a new Aspire project using the official template, it doesn't just give you a single app; it gives you a whole solution setup that is meant to grow with your project.

Let's break down what you get:

- `BlazorWebApp.AppHost`: This is where everything gets wired up
- `BlazorWebApp.ServiceDefaults`: It contains the shared config, including logging, metrics, and tracing
- `BlazorWebApp`: Your Blazor project
- `BlazorWebApp.Client`: The Blazor WebAssembly project

Let's take a quick look at the Aspire-related projects.

AppHost project

This is the *orchestrator*. It doesn't run any logic of your app; its only job is to define the distributed app. It's the entry point that tells Aspire what services to start and how they relate to each other. Here, you'll write C# to say things like:

- *"I want to run the Web project."*
- *"The Web project needs access to Redis and SQL Server."*
- *"Here's what port to use, and what environment variables to pass."*

We'll go deeper into this, but just remember, this is where Aspire gets its instructions.

ServiceDefaults project

This is where you put the shared configuration, such as:

- Structured logging
- OpenTelemetry tracing and metrics
- Retry policies
- Health checks
- Service discovery settings

Instead of duplicating that config across every project, set it once in `ServiceDefaults` and reference it from the other projects. That way, everything behaves consistently.

BlazorWebApp and BlazorWebApp.Client

This is your actual app (not related to Aspire). It could be a Blazor Web App, an API, a minimal API, MVC, or whatever your frontend is built with. It doesn't know or care about the AppHost. It just runs like a normal .NET app. The only difference is that Aspire wires it up automatically with the right environment variables and connected services.

References and wiring

Then you wire everything together in the AppHost project using simple builder syntax. For example:

```
builder.AddProject<Projects.MyAspireApp_Web>("web")
       .WithReference(redis)
       .WithReference(sqlServer);
```

This sets up the following:

- Port bindings
- Environment variables
- Named connections between services
- DNS-like routing (e.g., your app can reach SQL Server at `sqlServer` instead of `localhost`)

The big picture

Aspire doesn't force any of this. You can move things around or leave parts out. But the default structure works well, especially for teams or apps that are growing.

I wanted to start playing around with Aspire, but it was important to me (and the team) that the projects would work just as before; we didn't want to make Aspire the only way to run our project. We added an Aspire project to our solution and allowed it to override the environment variables we were using to better work with Aspire. So, Aspire is not an all-or-nothing solution (even though it can be).

Aspire has a lot of things built in, but it also has a Community Toolkit where we can find additional assets. Let's look at that next.

Unraveling the Aspire Community Toolkit

Aspire is still pretty new, but the .NET community has already started building around it, and one of the most useful things out there right now is the **Aspire Community Toolkit**. It's a growing collection of community-created resources that extend Aspire with support for services not yet included in the official SDK. So, if you want to hook up your Aspire app to Kafka, MongoDB, Dapr, or anything else not built in, chances are the toolkit has something for you.

What is it?

The Aspire Community Toolkit is a collection of open-source extensions that follow the same style as the built-in Aspire methods. So instead of:

```
builder.AddContainer("my-random-kafka", ...)
```

You can write:

```
builder.AddKafka("kafka")
```

As you can see, this is much cleaner and easier to reuse across projects.

These extensions are available as NuGet packages, and they're designed to plug into the `DistributedApplicationBuilder` class just like any other Aspire resource.

Examples of community resources

As of writing, the toolkit includes extensions for things like:

- Kafka (`AddKafka()`)
- Dapr (`AddDapr()`)
- MinIO (S3-compatible storage)
- Vector (log aggregation)
- Prometheus + Grafana (metrics dashboards)

- MailDev (email testing)
- RabbitMQ (with custom settings)
- PostgreSQL (with simplified config)
- Redis Stack (with extra modules)

And more are being added all the time. You can browse the full list here:

`https://github.com/CommunityToolkit/Aspire`

But where Aspire really shines is in holding all of this together, thanks to the Aspire dashboard.

Exploring the Aspire dashboard

One of the most helpful things Aspire gives us is the dashboard.

It starts automatically when you run your app and gives you a full overview of everything that's running. There's no need to memorize ports, dig through logs, or switch between terminals.

You get the following:

- A list of all services in your app
- The consoles of the different services in one place (this alone is reason enough to use Aspire)
- Logs for each service
- Port info and URLs
- Health checks
- Traces (as we discussed earlier)
- Quick actions like *restart* or *open* in a browser

When you launch your Aspire app (usually by pressing *F5* or running the `AppHost` project), Aspire spins up a local dashboard, typically at something like: `https://localhost:12345`. The exact port changes depending on what's free, but you'll see it in the terminal or output window. Once the dashboard opens, you'll see something like this:

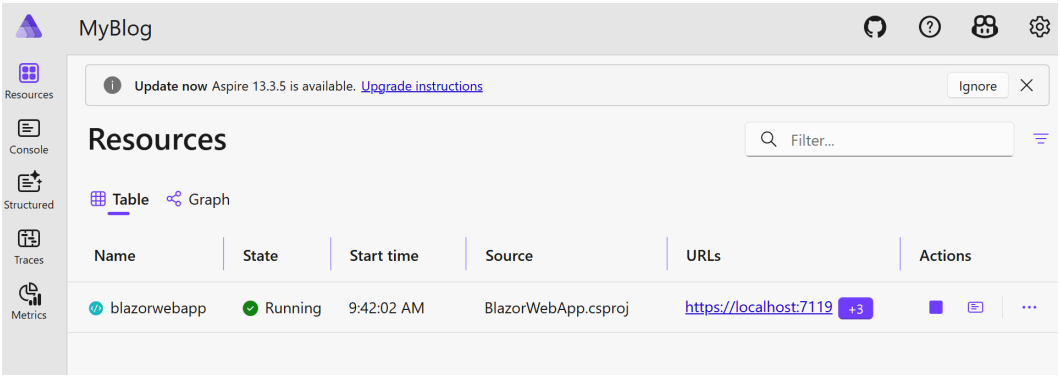


Figure 4.1: The Aspire dashboard

Before Aspire, running the project often meant juggling multiple command windows, one for each service, each showing its own logs and output. The dashboard pulls all of that into a single view. If we had multiple projects, like frontend and API, then we would have two command windows. The dashboard consolidates all of that into one.

We can track calls across services, examine timings, and more.

Even if we have a single resource, adding Aspire to it is worth it just because of the dashboard. The dashboard shows a lot of information. Let's take a look at what we can find.

Exploring logs, traces, and metrics

One of the nicest things about Aspire is that you don't have to set up a bunch of logging, tracing, or metrics pipelines manually. That stuff is built in from the start, and it works across all your services.

Aspire uses OpenTelemetry under the hood, which is the industry standard for distributed tracing and metrics. But don't worry, you don't need to become an OpenTelemetry expert to benefit from it. Let's take a closer look at what you get out of the box.

Structured logs

All your logs, from every service, show up in the Aspire dashboard.

Each service has its own tab where you can:

- View logs in real time
- See the log level (info, warn, error, etc.)
- Spot issues across services without switching terminals

And the logs are structured, meaning they aren't just strings. They're objects with properties like:

- Timestamp
- Log level
- Message
- Category
- Correlation ID (this is super useful for tracing across services)

You can filter logs by service or search for errors, all from your browser.

Traces

This is where things get really cool. If one service calls another (like your frontend calls your API, and your API calls a database), Aspire will automatically track that full request chain and show it in a timeline. You'll see:

- When the request started
- How long each step took
- Which services were involved
- Any failures or exceptions along the way

It's a bit like looking at a flight itinerary, but for requests through your system.

And again, you don't need to write any special code for this. If you use `AddServiceDefaults()` in your app, you're already opted in.

Metrics

Aspire also collects basic metrics, like:

- Number of requests
- Response time
- Exceptions
- Memory usage
- CPU usage (depending on platform)

These also appear in the dashboard, allowing you to quickly assess how each service is performing. You won't get Prometheus-level detail out of the box, but it's more than enough to see if something is off.

Want to go deeper? You can plug in Prometheus, Grafana, or other exporters later; Aspire doesn't get in the way.

Correlated logs

Because Aspire sets up distributed tracing, you also get correlated logs. This means you can trace a single request across multiple services by its trace ID.

Super handy when you're trying to debug things like:

- *"Why is this request taking 3 seconds?"*
- *"Where did it fail?"*
- *"Which service logged this error?"*

Clicking on a trace in the dashboard lets you see the whole journey.

As you can see, Aspire gives you production-level observability without needing a DevOps team to set it up. You can use it from day one, even in tiny side projects, with everything available right in the dashboard.

Summary

In this chapter, we provided a brief overview of the possibilities offered by Aspire. We will continue to use Aspire throughout the book; however, Aspire is a book all on its own. This chapter is not intended to make you master Aspire, but rather to help you (hopefully) like it and know how to use it.

You don't have to use all of Aspire to get value from it. Even if your app is small, it's worth trying out, especially for the dashboard and smooth developer experience. Aspire improves the development experience by saving time, simplifying onboarding, and maintaining consistency across environments.

In this book, we'll focus on its benefits for local development, but it can also be used as a foundation for provisioning, deploying, and managing applications. And since it's built by the .NET team (and moving fast), it's a safe bet that we'll see even more features and improvements over time. You can find more information about Aspire at their official site: `https://aspire.dev`.

Wow, this was an information-heavy chapter. In the next chapter, it is time to write some code again. We are about to look at state management.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/8VuShAAtK`

Part 2

Building a Blazor Application

In this part of the book, we'll start building the blog application and add the features most real applications need. We'll work with data, create components, build forms, add validation, create an API, secure the application, share code, use JavaScript interop, and manage state. By the end of this part, we'll have a working Blazor application with many of the building blocks we need for real-world development.

This part includes the following chapters:

- *Chapter 5, Managing State – Part 1*
- *Chapter 6, Understanding Basic Blazor Components*
- *Chapter 7, Creating Advanced Blazor Components*
- *Chapter 8, Building Forms with Validation*
- *Chapter 9, Creating an API*
- *Chapter 10, Adding Authentication and Authorization*
- *Chapter 11, Sharing Code and Resources*
- *Chapter 12, JavaScript Interop*
- *Chapter 13, Managing State – Part 2*

5

Managing State – Part 1

In this chapter, we will start looking at managing state. There is also a continuation of this chapter in *Chapter 13*, *Managing State – Part 2*.

There are many ways to manage state or persist data. As soon as we leave a component, the state is gone. If we click the counter button and then navigate away, Blazor won't know how many times we clicked it, so we have to start over. You can't imagine how many times I have clicked that counter button over the years. It is such a simple yet powerful demo of Blazor and was a part of Steve's original demo back in 2017.

To get started quickly, I have split state management into two chapters. In this chapter, we focus on data access, and we will return to more state management in the second part. Since the previous edition, Aspire has become a really great way to set up a development environment, so it makes sense to use it in this new edition.

We will set up a PostgreSQL database and use Entity Framework to access the data. We will also use a common pattern called the **repository pattern** and create a repository to access the database data.

By the end of this chapter, you will have learned how to set up a PostgreSQL database using Aspire and how to use a repository to access data.

We will cover the following main topics:

- Creating data classes
- Creating an interface and models
- Creating the repository
- Adding the repository and database

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter04` folder on GitHub as the starting point.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter05`.

Creating data classes

There are two data classes we need to create:

- The data classes for the database
- The data classes for the API

The API data classes describe our business domain, not how we store the data. In our case, these will be almost identical. These API-facing data classes are typically referred to as **Data Transfer Objects** (**DTOs**) and are shared between the server and the client. This is one of the powers of having the same language in both frontend and backend. In contrast, in a JS framework, we would have to create copies of the data objects: once in C# for the backend and again in TypeScript or JavaScript for the frontend. As we will see later in the book, sharing the same logic and, more importantly, the validation logic has its upsides.

We will place the database classes in the BlazorWebApp project because we only need to communicate with the database when running on the server. Then we need another project to share between the client and server. The `BlazorWebApp.Client` project is just that, a project that contains the code that will run on the client and also shares the code with the server. We can use the `BlazorWebApp.Client` project for the shared code.

Before we continue, I just want to mention that there are probably people who would separate this into many different projects. Heck, I did that in the previous editions. However, in this edition, I want to dial it back a bit. Instead of having many different projects, let's focus on Blazor, and what Blazor gives us out of the box. We will focus on one server project and one client project (which is shared between server and client). This is really all we need. In previous versions of Blazor, it also included a shared project, but Microsoft has removed that from the template since the client project is shared.

Now we need to create a class for our blog post. Let's start with the database classes. To do that, we will go back to Visual Studio:

1. Open the `MyBlog` solution in Visual Studio if it is not already open.

2. In the `BlazorWebApp` project, add the following NuGet package:

```
Aspire.Npgsql.EntityFrameworkCore.PostgreSQL
```

3. Right-click on the `BlazorWebApp` project and select **Add | New Folder**. Name the folder `Database`.
4. Right-click on the **Database** folder and select **Add | New Folder**. Name the folder `Entities`.
5. Right-click on the **Entities** folder and select **Add | Class**. Name the class `BlogPost.cs` and click **Add**.
6. Follow *Step 3* again and create the following files:
   - `Category.cs`
   - `Tag.cs`
   - `Comment.cs`

   In the end, you should have four classes in the `Entities` folder.
7. Open `Category.cs` and replace the content with the following code:

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

namespace BlazorWebApp.Database.Entities;

public class Category
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public string Name { get; set; } = string.Empty;

    // Navigation property for related BlogPosts
    public virtual ICollection<BlogPost> BlogPosts { get; set; }
}
```

The `Category` class contains `Id` and `Name` properties.

8. Open `Tag.cs` and replace the content with the following code:

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

namespace BlazorWebApp.Database.Entities;

public class Tag
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public string Name { get; set; } = string.Empty;

    // Navigation property for the many-to-many relationship
    // with BlogPosts
    public virtual ICollection<BlogPost> BlogPosts { get; set; }
}
```

The `Tag` class contains an `Id` property and a `Name` property.

9. Open `Comment.cs` and replace the content with the following code:

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

namespace BlazorWebApp.Database.Entities;

public class Comment
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    // Foreign key to BlogPost
    public int BlogPostId { get; set; }

    // Navigation property to BlogPost
    public virtual BlogPost BlogPost { get; set; }
```

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

    public string Text { get; set; } = string.Empty;

    public string Name { get; set; } = string.Empty;
}
```

The `Comment` class could be part of the `Blogpost` class, but to use the same classes for different database types, we add comments as a separate entity referencing the blog post.

10. Open `BlogPost.cs` and replace the content with the following code:

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

namespace BlazorWebApp.Database.Entities;

public class BlogPost
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public string Title { get; set; } = string.Empty;

    public string Text { get; set; } = string.Empty;

    public DateTime PublishDate { get; set; }

    [ForeignKey("CategoryId")]
    public Category? Category { get; set; }
    public int? CategoryId { get; set; }

    public List<Tag> Tags { get; set; } = new();
}
```

In this class, we define the content of our blog post. We need an `Id` property to identify the blog post, a title, some text (the article), and the publishing date. We also have a `Category` property in the class, which is of the `Category` type. In this case, a blog post can have only one category, and a blog post can contain zero or more tags. We define the `Tag` property with `List<Tag>`.

Great, now we need a data context, a way to access our database:

1. In the `BlazorWebApp` project, click the **Database** folder and press *Shift* + *F2*, name the file `BlogDbContext.cs`, then press *Enter*. This is the quickest way to create a new class in Visual Studio.
2. Open the newly created file and replace the content with:

```
using BlazorWebApp.Database.Entities;
using Microsoft.EntityFrameworkCore;

namespace BlazorWebApp.Database;

public class BlogDbContext(DbContextOptions<BlogDbContext> options) :
DbContext(options)
{
    public DbSet<BlogPost> BlogPosts { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Tag> Tags { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(
        ModelBuilder modelBuilder)
    {
        // Define the many-to-many relationship between
        // BlogPost and Tag
        modelBuilder.Entity<BlogPost>()
            .HasMany(b => b.Tags)
            .WithMany(t => t.BlogPosts)
            .UsingEntity(j => j.ToTable("BlogPostTags")); // Join
            // table for many-to-many

        // Other model configurations can be placed here
    }
}
```

This code is the definition of the database, the tables, and the relationships.

This code is also using primary constructors, which just might be the nicest feature in .NET in a long time.

Normally, at this point, we would create an Entity Framework migration to add tables or make changes to the tables, but since its inner workings are outside the scope of this book, we will choose another approach that creates the database without handling migrations.

> **Note**
>
> If you want to read more about Entity Framework and migrations, you can find more information here:`https://learn.microsoft.com/en-us/ef/`.

The database classes and context are done. Next, we will start working with repositories. But before we can implement a repository, we need two things: DTOs that describe the data we want to pass around and an interface that defines what the repository should be able to do.

Creating an interface and models

In this section, we will prepare the pieces our repositories need: DTOs and an interface. Later, we will create two repository implementations: one that talks directly to the database and one that uses HTTP calls to a Web API. Both will use the same interface, which means the rest of the app can ask for blog posts, categories, tags, and comments without caring where the data comes from. In *Chapter 9*, *Creating an API*, we will get back to creating the Web API.

Why are we creating two repositories?

We are creating a service with direct database access and a client that goes over the web and then uses direct database access. But we will use the same interface for both scenarios, making it possible to use one on the server and the other on the client.

I tend to access my data in only one way, always via a web API. This way, I don't have to handle two different ways of retrieving data, and I know that there is no other way to retrieve data other than using the web API. Also, I can make sure caching and stuff like that are in one place. But the point is to show that it is possible to mix and match and pick what is right for your scenario. By showing the two methods of creating repositories, this book is relevant for those who want to use WebAssembly as well as for those who want to use the server.

The first thing we need is a bunch of DTOs. If you are new to creating APIs, **DTOs** (**Data Transfer Objects**), in short, are classes that are responsible for transferring data. Imagine that we might want to change how we store our data; in this book, it's PostgreSQL, but we might

change that. In that case, our DTOs should remain the same, so that the API won't break. This is one of the reasons we separate DTOs from the database classes we created earlier. Another reason is that there might be columns in the database you want to keep in the database, rather than send to your users.

Let's add some DTOs:

1. Right-click on the `BlazorWebApp.Client` project, click **Add | New Folder**, then name the folder `Models`.
2. In the **Models** folder, create the following files (*Shift + F2*):
   - `BlogPost.cs`
   - `Category.cs`
   - `Comment.cs`
   - `Tag.cs`
3. Open `BlogPost.cs` and replace the content with:

```
namespace BlazorWebApp.Client.Models;
public class BlogPost
{
    public string? Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string Text { get; set; } = string.Empty;
    public DateTime PublishDate { get; set; }
    public Category? Category { get; set; }
    public List<Tag> Tags { get; set; } = new();
}
```

   As you can see, these files are a lot simpler than the ones we created for the database. They are designed to transfer data, not to describe how to store the data.

4. Open `Category.cs` and replace the content with:

```
namespace BlazorWebApp.Client.Models;
public class Category
{
    public string? Id { get; set; }
    public string Name { get; set; } = string.Empty;
}
```

5. Open `Comment.cs` and replace the content with:

```
namespace BlazorWebApp.Client.Models;

public class Comment
{
    public string? Id { get; set; }
    public required string BlogPostId { get; set; }
    public DateTime Date { get; set; }
    public string Text { get; set; } = string.Empty;
    public string Name { get; set; } = string.Empty;
}
```

6. Open `Tag.cs` and replace the content with:

```
namespace BlazorWebApp.Client.Models;

public class Tag
{
    public string? Id { get; set; }
    public string Name { get; set; } = string.Empty;
}
```

Awesome! We now have some DTOs.

Next, we will create an interface:

1. Right-click on the `BlazorWebApp.Client` project, select **Add | New Folder**, and name it `Interfaces`.
2. In the **Interfaces** folder, create a new file (*Shift + F2*) and name it `IBlogRepository.cs`.
3. Open `IBlogRepository.cs` and replace its content with the following:

```
using BlazorWebApp.Client.Models;

namespace BlazorWebApp.Client.Interfaces;

public interface IBlogRepository
{
    Task<int> GetBlogPostCountAsync();
    Task<List<BlogPost>> GetBlogPostsAsync(int numberofposts,
        int startindex);
```

```
    Task<List<Category>> GetCategoriesAsync();
    Task<List<Tag>> GetTagsAsync();
    Task<List<Comment>> GetCommentsAsync(string blogPostId);
    Task<BlogPost?> GetBlogPostAsync(string id);
    Task<Category?> GetCategoryAsync(string id);
    Task<Tag?> GetTagAsync(string id);
    Task<BlogPost?> SaveBlogPostAsync(BlogPost item);
    Task<Category?> SaveCategoryAsync(Category item);
    Task<Tag?> SaveTagAsync(Tag item);
    Task<Comment?> SaveCommentAsync(Comment item);
    Task DeleteBlogPostAsync(string id);
    Task DeleteCategoryAsync(string id);
    Task DeleteTagAsync(string id);
    Task DeleteCommentAsync(string id);
}
```

Alright, so here's the deal with this `IBlogRepository` thing: it's basically our cheat sheet for handling all the blog stuff, like posts, comments, tags, and categories. Need to grab some posts or zap one out of existence? This interface is your go-to. It's all about making our lives easier when we're coding the blog and keeping things tidy and straightforward.

Now, we have an interface for the repository with the methods we need to list blog posts, tags, and categories, as well as to save (create or update) and delete them. Next, let's implement the interface.

Creating the repository

Now that we have defined the interface, it's time to create the repository that implements it. The interface acts as a contract, describing what the repository should do. By implementing it, we ensure that our repository follows the same structure regardless of how the data is retrieved.

The idea is to create a class that stores our blog posts, tags, comments, and categories in a database. We will start by implementing the direct access implementation. This is the one we can use when accessing information directly from the database and not through a Web API. We will use the direct access implementation when running our components on the server and accessing the database, and our Web API will also use it to access the database, but we will come back to that in *Chapter 9*, *Creating an API*.

So, to implement the interface for direct database access, follow these steps:

1. Right-click on the `BlazorWebApp` project, select **Add | New Folder**, and name the folder `Repositories`.
2. Right-click on the **Repositories** folder, select **Add | Class**, and name the class `BlogRepositoryEntityFrameworkDirectAccess.cs`.
3. Open `BlogRepositoryEntityFrameworkDirectAccess.cs` and replace the code with the following:

```
using BlazorWebApp.Client.Interfaces;
using BlazorWebApp.Client.Models;
using BlazorWebApp.Database;
using Microsoft.EntityFrameworkCore;
namespace BlazorWebApp.Repositories;
public class BlogRepositoryEntityFrameworkDirectAccess(
    IDbContextFactory<BlogDbContext> factory) : IBlogRepository
{}
```

   This is the start of our repository with direct access. It references the `IBlogRepository` interface, and we will implement each method that the interface requires.

   The error list in Visual Studio should contain many errors since we haven't implemented the methods yet. We are inheriting from the `IBlogRepository` interface, so we know which methods to expose.

   The `BlogRepositoryEntityFrameworkDirectAccess` name is exaggerated; normally, we would probably name it something else, but to really show you what this version of the repository is for, I have chosen a descriptive name.

4. Next, it's time to implement the API by adding the methods to get blog posts. Add the following code:

```
public async Task<BlogPost?> GetBlogPostAsync(string id)
{
    using var context = factory.CreateDbContext();
    //Convert id to an int
    if (int.TryParse(id, out int intid))
    {
        var item = await context.BlogPosts.Include(p =>
            p.Category).Include(p => p.Tags).FirstOrDefaultAsync(
                p => p.Id == intid);
```

```
        if (item != null)
        {
            return ConvertBlogPostToDto(item);
        }
    }
    return null;

}

public async Task<List<BlogPost>> GetBlogPostsAsync(
    int numberofposts, int startindex)
{
    using var context = factory.CreateDbContext();
    return await context.BlogPosts.OrderByDescending(p =>
        p.PublishDate).Skip(startindex).Take(numberofposts)
            .Select(p => ConvertBlogPostToDto(p)).ToListAsync();
}

public async Task<int> GetBlogPostCountAsync()
{
    using var context = factory.CreateDbContext();
    return await context.BlogPosts.CountAsync();
}
```

The `GetBlogPostsAsync` method takes a couple of parameters that we will use later for paging. It will retrieve the blog posts from our database and return the posts we are requesting, skipping the correct number for our paging.

Last but not least, we also have a method that returns the current blog post count, which we will use for paging.

We also call a couple of helper methods, which we need to convert our database objects to DTOs and our DTOs to database objects. This is where you might use libraries like `AutoMapper` to map between these different object types. Personally, I usually map them manually and ensure the mappings are in one place, as we have done here.

5. Now let's add the helper methods:

```
private static Tag ConvertTagToDto(
    BlazorWebApp.Database.Entities.Tag item)
{
    return new Tag() { Id = item.Id.ToString(), Name = item.Name };
```

```
}

private static Comment ConvertCommentToDto(
    BlazorWebApp.Database.Entities.Comment item)
{
    return new Comment() { Id = item.Id.ToString(), Text =
        item.Text, Date = item.Date, BlogPostId =
            item.BlogPostId.ToString(), Name = item.Name };
}

private static Category? ConvertCategoryToDto(
    BlazorWebApp.Database.Entities.Category? item)
{
    if (item == null)
    {
        return null;
    }
    return new Category() { Id = item.Id.ToString(),
        Name = item.Name };
}

private static BlogPost ConvertBlogPostToDto(
    BlazorWebApp.Database.Entities.BlogPost item)
{
    Category? category = null;
    if (item.Category != null)
    {
        category = new Category() { Id =
            item.Category.Id.ToString(),
                Name = item.Category.Name };
    }
    return new BlogPost()
    {
        Id = item.Id.ToString(),
        Title = item.Title,
        Text = item.Text,
        PublishDate = item.PublishDate,
        Category = category,
        Tags = item.Tags.Select(t => new Tag() { Id =
            t.Id.ToString(), Name = t.Name }).ToList()
```

```
    };
}
```

They simply convert a DTO to an Entity Framework file, or vice versa. You may notice that we are converting a string to an `int`. This is because our interface is using strings as IDs. The point here is that if we use a string, we can use any type of storage, like RavenDB, JSON files, or, in this case, PostgreSQL. The type of ID really doesn't matter.

6. Now, we need to add the same `Category` methods. To do this, add the following code:

```
public async Task<List<Category>> GetCategoriesAsync()
{
    using var context = factory.CreateDbContext();
    return await context.Categories.Select(c =>
        ConvertCategoryToDto(c)!).ToListAsync();
}

public async Task<Category?> GetCategoryAsync(string id)
{
    using var context = factory.CreateDbContext();
    return ConvertCategoryToDto(await
        context.Categories.FirstOrDefaultAsync(
            c => c.Id == Convert.ToInt32(id)));

}
```

The `Category` methods don't support paging. Otherwise, they should look familiar, as they do almost the same as the blog post methods.

7. Now, it's time to do the same thing for tags. Add the following code:

```
public async Task<Tag?> GetTagAsync(string id)
{
    using var context = factory.CreateDbContext();
    var item = await context.Tags.FirstOrDefaultAsync(
        t => t.Id == Convert.ToInt32(id) );
    if (item != null)
    {
        return ConvertTagToDto(item);
    }
    return null;
```

```
}

public async Task<List<Tag>> GetTagsAsync()
{
    using var context = factory.CreateDbContext();
    return await context.Tags.Select(t =>
        ConvertTagToDto(t)).ToListAsync();
}
```

As we can see, the Tag code is essentially a copy of the categories code.

8. We also need a way to retrieve the comments for a blog post. We will not create a method to retrieve a single comment; we always retrieve all comments for a specific post. Add the following method:

```
public async Task<List<Comment>> GetCommentsAsync(string blogPostId)
{
    using var context = factory.CreateDbContext();
    int id = Convert.ToInt32(blogPostId);
    return await context.Comments.Where(c => c.BlogPostId ==
        id).Select(t => ConvertCommentToDto(t)).ToListAsync();
}
```

This method retrieves all comments for a blog post.

9. We also need a couple of methods for saving the data, so next up, we'll add methods for saving blog posts, categories, comments, and tags. We'll start with blog posts, so add the following code:

```
public async Task<BlogPost?> SaveBlogPostAsync(BlogPost item)
{
    using var context = factory.CreateDbContext();
    if (item.Id == null)
    {
        var newitem = new BlazorWebApp.Database.Entities.BlogPost()
        {
            Title = item.Title,
            Text = item.Text,
            PublishDate = item.PublishDate,
            CategoryId = item.Category == null ? null :
                int.Parse(item.Category.Id!),
```

```
        };

        foreach (var tag in item.Tags)
        {
            var t = await context.Tags.FirstOrDefaultAsync(
                t => t.Id == Convert.ToInt32(tag.Id));
            if (t != null)
            {
                newitem.Tags.Add(t);
            }
        }

        context.Add(newitem);
        await context.SaveChangesAsync();
        item.Id = newitem.Id.ToString();
        return item;
    }
    else
    {
        var existingitem = await context.BlogPosts.Include(p =>
            p.Category).Include(p => p.Tags).FirstOrDefaultAsync(
                p => p.Id == Convert.ToInt32(item.Id));
        if (existingitem != null)
        {
            existingitem.Title = item.Title;
            existingitem.Text = item.Text;
            existingitem.PublishDate = item.PublishDate;
            existingitem.CategoryId = item.Category == null ?
                null : int.Parse(item.Category.Id!);

            existingitem.Tags.Clear();
            foreach (var tag in item.Tags)
            {
                var t = await context.Tags.FirstOrDefaultAsync(
                    t => t.Id == Convert.ToInt32(tag.Id));
                if (t != null)
                {
                    existingitem.Tags.Add(t);
                }
            }
            await context.SaveChangesAsync();
```

```
            return item;
        }
    }
    return item;
}
```

10. Add the following code for categories:

```
public async Task<Category?> SaveCategoryAsync(Category item)
{
    using var context = factory.CreateDbContext();
    if (item.Id == null)
    {
        var newitem = new BlazorWebApp.Database.Entities.Category()
        {
            Name = item.Name,
        };
        context.Add(newitem);
        await context.SaveChangesAsync();
        item.Id = newitem.Id.ToString();
        return item;
    }
    else
    {
        var existingitem =
            await context.Categories.FirstOrDefaultAsync(
                p => p.Id == Convert.ToInt32(item.Id));
        if (existingitem != null)
        {
            existingitem.Name = item.Name;
            await context.SaveChangesAsync();
            return item;
        }
    }
    return item;
}
```

11. Add the following for comments:

```
public async Task<Comment?> SaveCommentAsync(Comment item)
{
```

```
    using var context = factory.CreateDbContext();
    if (item.Id == null)
    {
        var newitem = new BlazorWebApp.Database.Entities.Comment()
        {
            BlogPostId = int.Parse(item.BlogPostId),
            Text = item.Text,
            Date = item.Date,
            Name = item.Name
        };
        context.Add(newitem);
        await context.SaveChangesAsync();
        item.Id = newitem.Id.ToString();
        return item;
    }
    else
    {
        var existingitem =
            await context.Comments.FirstOrDefaultAsync(
                p => p.Id == Convert.ToInt32(item.Id));
        if (existingitem != null)
        {
            existingitem.Text = item.Text;
            await context.SaveChangesAsync();
            return item;
        }
    }
    return item;
}
```

12. Add the following for tags:

```
public async Task<Tag?> SaveTagAsync(Tag item)
{
    using var context = factory.CreateDbContext();
    if (item.Id == null)
    {
        var newitem = new BlazorWebApp.Database.Entities.Tag()
        {
            Name = item.Name,
        };
```

```
            context.Add(newitem);
            await context.SaveChangesAsync();
            item.Id = newitem.Id.ToString();
            return item;
        }
        else
        {
            var existingitem = await context.Tags.FirstOrDefaultAsync(
                p => p.Id == Convert.ToInt32(item.Id));
            if (existingitem != null)
            {
                existingitem.Name = item.Name;
                await context.SaveChangesAsync();
                return item;
            }
        }
        return item;
    }
```

13. We now have a method for saving and getting items. But sometimes, things don't go as planned, and we need a way to delete the items that we have created. Next up, we will add some delete methods. Add the following code:

```
public async Task DeleteBlogPostAsync(string id)
{
    await DeleteItemAsync<Database.Entities.BlogPost>(id);
}

public async Task DeleteCategoryAsync(string id)
{
    await DeleteItemAsync<Database.Entities.Category>(id);
}

public async Task DeleteTagAsync(string id)
{
    await DeleteItemAsync<Database.Entities.Tag>(id);
}

public async Task DeleteCommentAsync(string id)
{
    await DeleteItemAsync<Database.Entities.Comment>(id);
```

```
}

private async Task DeleteItemAsync<T>(string id) where T:class
{
    if (int.TryParse(id, out int intid))
    {
        using var context = factory.CreateDbContext();
        var item = await context.Set<T>().FindAsync(intid);
        if (item != null)
        {
            context.Set<T>().Remove(item);
            await context.SaveChangesAsync();
        }
    }
}
```

The code we just added calls the `DeleteItemAsync` method, which is a generic method that can delete the blog post, tag, category, etc. As you may see, we are converting the ID into an `int` type. This is because our ID in the database is an `int`. If you know that you will never support anything else but an `int`, it makes more sense to use an `int` in the interface as well, so you don't have to convert. In our case, this interface should work with Entity Framework, and also RavenDB, which uses strings as IDs.

With that, our repository is done! The next step is to add and configure the Blazor project to use our new storage.

Adding the repository and database

We have the DTOs, we have the Entity Framework classes, and we even have our repository. But before we configure our repository, we need a database server and a database!

Adding PostgreSQL

In our `Myblog.AppHost` project, we need to add a package:

1. Right-click on the `MyBlog.AppHost`, then select **Manage NuGet Packages...**.
2. Search for `Aspire.Hosting.PostgreSQL` and click **Install**.
3. In the `MyBlog.AppHost` project, we need to add a couple of lines.

This is the end result:

```
var builder = DistributedApplication.CreateBuilder(args);
var postgres = builder.AddPostgres("postgres")
    .WithLifetime(ContainerLifetime.Persistent)
    .WithDataVolume(isReadOnly: false)
    .WithPgAdmin();

var myBlogDatabase = postgres.AddDatabase("myBlogDb");

builder.AddProject<Projects.BlazorWebApp>("blazorwebapp")
    .WithReference(myBlogDatabase)
    .WaitFor(postgres);

builder.Build().Run();
```

This is where Aspire really shines. We are creating a Postgres instance, which will spin up in our Docker environment, and we are telling it to keep running even if we stop debugging. This will speed up development time.

`WithDataVolume` tells Aspire to store the data somewhere else, not inside the container; this will persist the state between debug sessions. Then we call `WithPgAdmin`, which will spin up an additional container that will host an admin interface for our database. This is a web interface for looking at our data. I knooooow (I hope some hear Monica voice from Friends here), it IS amazing!

In that Postgres server, we are creating a database called `myBlogDb`. No passwords, no config, no magic connection strings. Then we tell Aspire that the database should be available in the BlazorWebApp.

In a production scenario, you would set everything up and provide the config with those connection strings, but this is outside the scope of this book.

4. Run the project, and you might see an error message like this: **Container runtime unhealthyContainer runtime 'docker' was found but appears to be unhealthy. Ensure that Docker is running and that the Docker daemon is accessible. If Resource Saver mode is enabled, containers may not run.**
We see this because our Docker Desktop is not running. To fix this, start Docker Desktop. You might see a setup tutorial; follow the steps. If you don't already have an account, the tutorial will ask you to create one. When I did it, I had to update **Windows Subsystem for Linux** (**WSL**) as well, but the tutorial will guide you through it. When all that is done, the Docker engine should start up.

5. Now, run the project again and see if it works better.
6. When we started our app, Aspire downloaded a PostgreSQL image and started it inside Docker Desktop. If you open Docker Desktop and go to the **Containers** tab, you will see something like *Figure 5.1*:

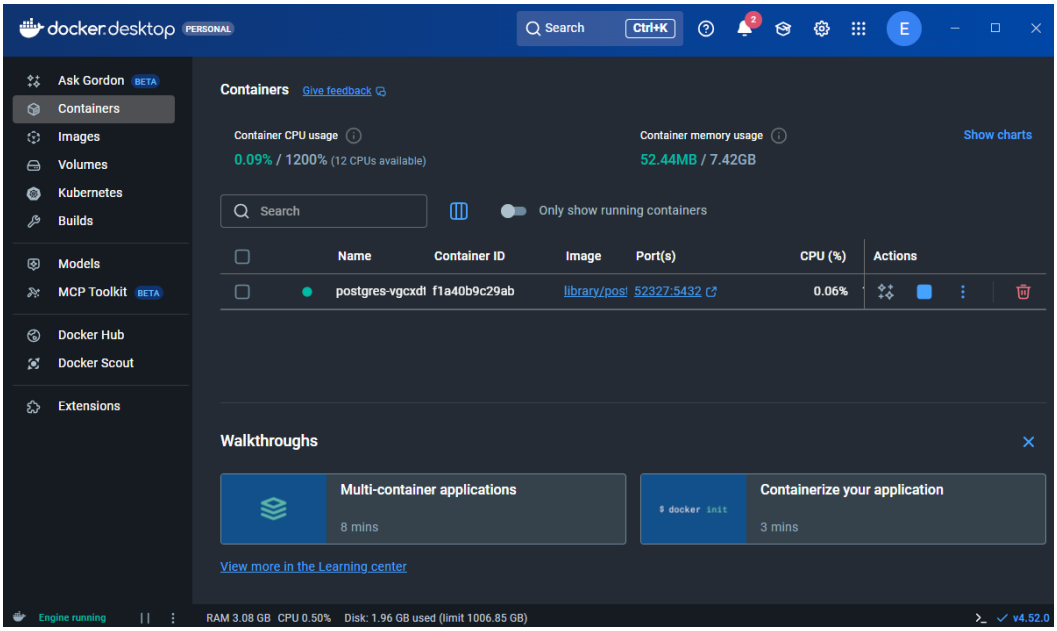


Figure 5.1: PostgreSQL container running in Docker

I know what you are thinking! I thought the same thing. It just can't be this easy! But it is... it really is! Imagine this being your first day at work: you just cloned the repo, and you press *F5*. Moments later, your machine is running exactly what you need, with the correct versions.

Now the server is up and running, and we have a database. Let's see if we can configure the Blazor project.

Configuring the database and repository

Now it's time to configure the `DbContext` and repository in our Blazor Server project:

1. In the `BlazorWebApp` project, in `Program.cs`, add the following namespaces:

```
using BlazorWebApp.Components;
using BlazorWebApp.Database;
```

2. Just before the line `var app = builder.Build();`, add the following code:

```
var connectionString =
    builder.Configuration.GetConnectionString("myBlogDb");

builder.Services.AddDbContextFactory<BlogDbContext>(options =>
    options.UseNpgsql(connectionString));
```

We are adding support for PostgreSQL and connecting it to the `BlogDbContext`. The connection string name must match the database name in the Aspire project. Aspire will automatically hook everything up for us.

3. A bit further down, in the `IsDevelopment` if statement, add the following:

```
if (app.Environment.IsDevelopment())
{
    app.UseWebAssemblyDebugging(); //<- Already there

    using (var scope = app.Services.CreateScope())
    {
        var context = scope.ServiceProvider.GetRequiredService
            <BlogDbContext>();
        await context.Database.EnsureCreatedAsync();
    }
}
```

This is instead of migrations; it ensures that when we spin up our database, we create the tables as well.

With that, we have everything we need, and the project should start. We are not reading or writing to the database, so the database is not yet in use.

Let's run it and see what happens by pressing *F5*. This will launch Aspire. Aspire will spin up our database, pgAdmin, and our Blazor application. Aspire should look something like *Figure 5.2*:

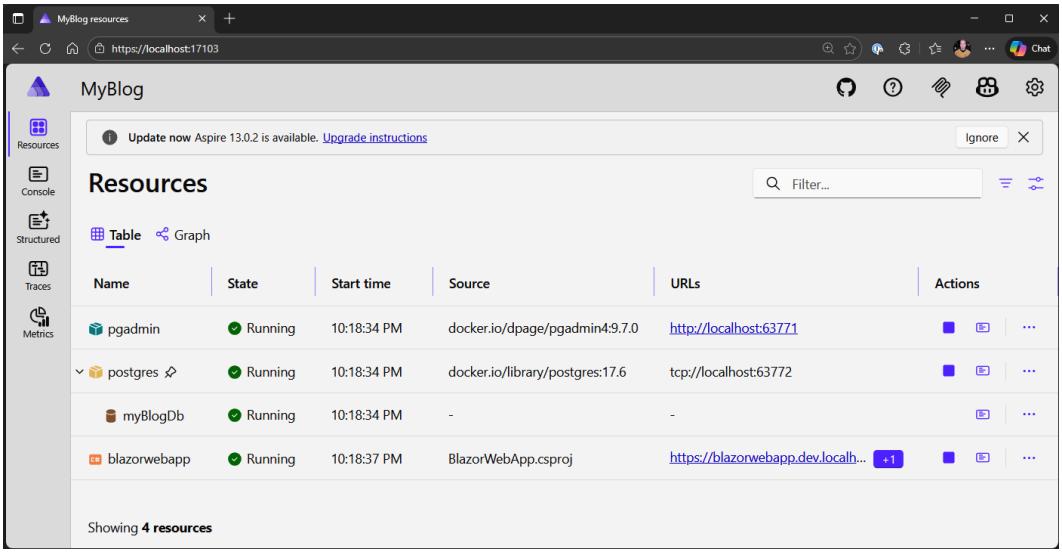


Figure 5.2: Aspire running all our resources

We can also click the **Graph** button, which will show us a visual representation of our resources, as in *Figure 5.3*:

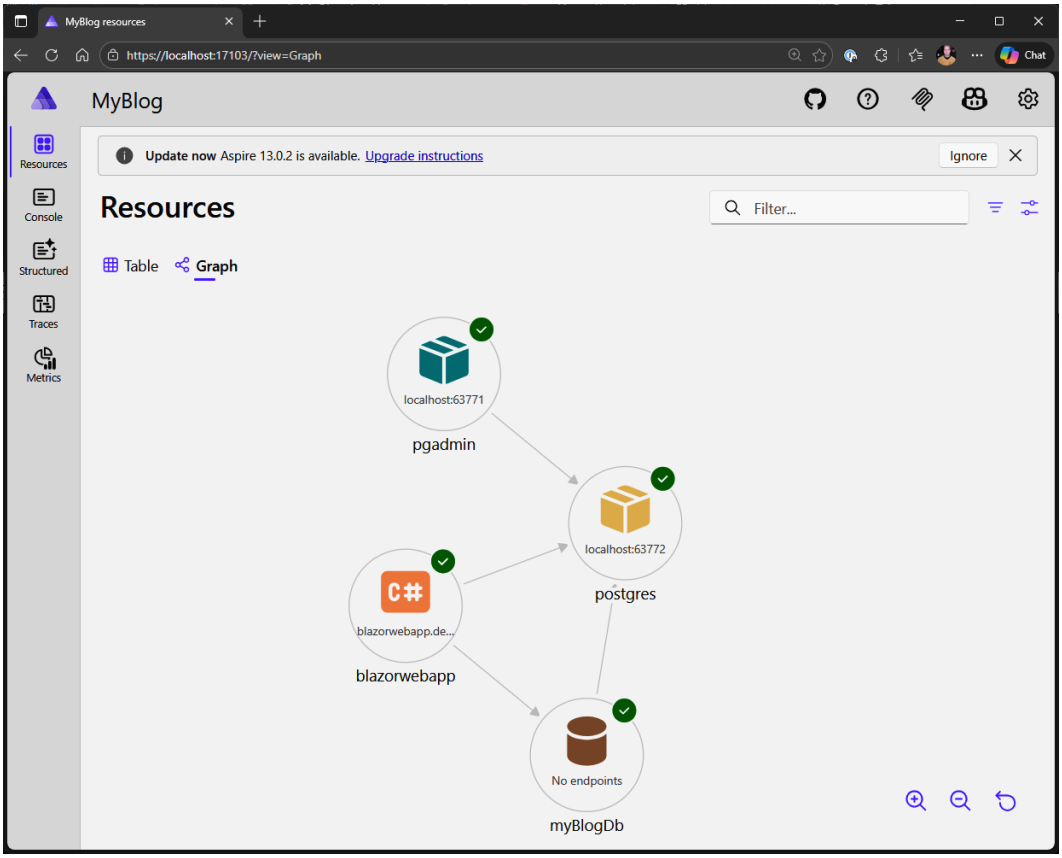


Figure 5.3: Aspire running all our resources

This is again the power of Aspire: we spin up everything we need for our application to run. We don't even have to install any additional software to view our database. That's what pgAdmin is for.

Exploring pgAdmin

pgAdmin is a free, open-source management tool for PostgreSQL. It gives you a graphical interface to interact with your PostgreSQL databases, so you don't have to run SQL commands manually all the time. You can use it to browse tables, run queries, manage users, and back up or restore databases. It's a handy tool when working with PostgreSQL.

Let's explore pgAdmin:

1. In the Aspire dashboard, look for `pgAdmin`, and click the URL in the **URLs** column. This will open a new window with pgAdmin. It takes a bit of time for pgAdmin to start, so if it hasn't started yet, give it a bit more time.
2. In the menu to the left, expand **Servers** | **Databases** | **myBlogDb** | **Schemas** | **Tables**. There, you should see the newly created tables based on Entity Framework, as shown in *Figure 5.4*:

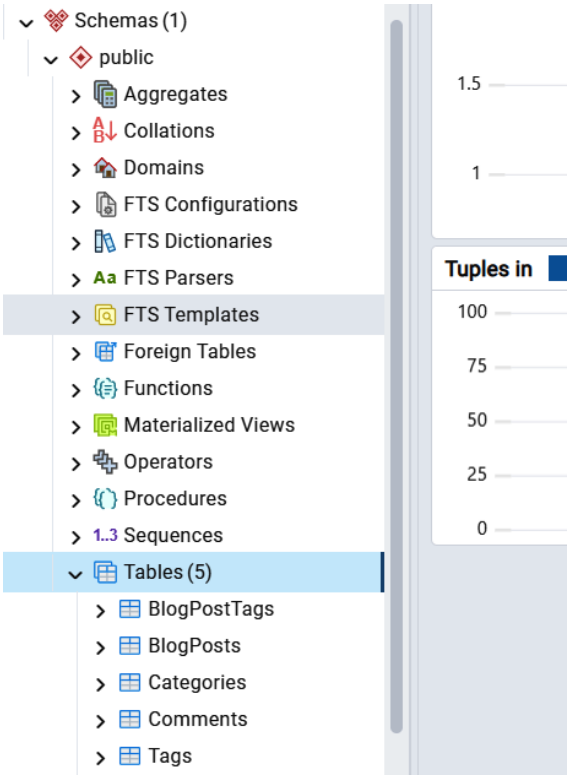


Figure 5.4: pgAdmin portal

We are all set; we have a database server, a database, tables, and a great start.

Summary

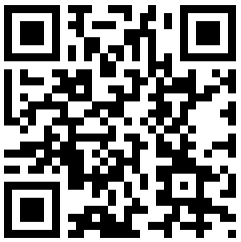
In this chapter, we took the first steps toward managing state in a Blazor application by focusing on data access and persistence. We set up a PostgreSQL database using Aspire, making it easy to spin up and manage our development environment with minimal configuration. Along the way, we defined database entities, created DTOs for client-server communication, and introduced a repository interface to abstract data access.

We then implemented a concrete repository using Entity Framework to handle common operations, including retrieving, saving, and deleting blog posts, categories, tags, and comments. This approach keeps our data access logic consistent and flexible, allowing us to swap storage implementations later without breaking the API.

We now have a running database, a configured Blazor application, and a clean data access layer ready for use by the UI. In the next chapter, we will shift our focus to Blazor components, explore the built-in components provided by the templates, and start building our own components using the repository created here.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

6

Understanding Basic Blazor Components

In this chapter, we will look at the components that come with the Blazor template and begin building our own components. Knowing the different techniques used for creating Blazor websites will help us when we start building our components. Blazor uses components for most things, so we will use the knowledge from this chapter throughout the book.

We will start this chapter with theory and end by creating a component that displays some blog posts using the API we created in *Chapter 5*, *Managing State – Part 1*. In between, we will look at how components access services through dependency injection, how render modes affect components, where different kinds of code should live, how lifecycle events control when code runs, and how parameters pass data into components.

As such, we will cover the following topics:

- Exploring components
- Learning Razor syntax
- Understanding dependency injection
- Figuring out where to put the code
- Exploring lifecycle events
- Understanding parameters
- Writing our first component

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter05` folder as the starting point.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter06`.

Exploring components

In Blazor, a **component** is a `.razor` file containing a small, isolated piece of functionality (code and markup), which can be used as a page. A component can host other components as well.

There are three different ways we can create a component:

- Using Razor syntax, with the code and HTML sharing the same file
- Using a code-behind file together with a `.razor` file
- Using only a code-behind file

In this chapter, we will go through the different options. To begin, we'll go through the components in the template we used to create the project; these all use the first option, `.razor` files, in which we have a mix of code and HTML in the same file.

The components in the template are as follows:

- **Counter**
- **Weather**

Let's take a look.

Counter

The `Counter` page shows a button and a counter; if we click the button, the counter increases.

We will now break the page apart, making it easier to understand. The page is located in the `BlazorWebApp.Client` project, inside the `Pages` folder.

At the top of the page is the `@page` directive, which makes it possible to route to the component directly, as we can see in this code:

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

If we start the `BlazorWebApp` project and add `/counter` to the end of the URL, we see that we can directly access the component by using its route. We can also make the route take parameters, but we will return to that later.

Depending on the template we choose, the page might also specify a render mode (we will return to this topic later, but it is worth mentioning here in case you have seen it in other templates):

```
@rendermode InteractiveAuto
```

This is how we can set the render mode for a specific component. This means that when we use this component, it will first render the page using Blazor Server (with SignalR) and, in the background, download the WebAssembly version so that the next time we load the page, it will then run the WebAssembly version instead. If we specify render mode globally as we have done in this book, the render mode cannot be set manually like this. It is set globally, as the name implies. We can override it by disabling interactivity, but we cannot change the interactivity render mode itself.

Next, let's explore the code. To add code to the page, we use the `@code` statement, and within that statement, we can add ordinary C# code, as shown:

```
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

In the preceding code block, we have a private `currentCount` variable set to `0`. Then, we have a method called `IncrementCount()`, which increments the `currentCount` variable by `1`.

We show the current value by using the `@` sign. In Razor, the `@` sign indicates that it is time for some code:

```
<p role="status">Current count: @currentCount</p>
```

As we can see, Razor is very smart because it understands when the code stops and the markup continues, so there is no need to add extra code to transition from code to markup (more on this in the next section).

As we can also see, we are mixing HTML attributes with `@currentCount`, and Razor understands the difference.

Next, we have a button that is the trigger for changing the value:

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

This is an HTML button with a `Bootstrap` class (to make it look a bit nicer). Here, `@onclick` binds the button's `onclick` event to the `IncrementCount()` method. If we were to use `onclick` without the `@`, it would refer to the JavaScript event and not work.

Looking at *Figure 6.1*, when we click the button, it will call the `IncrementCount()` method (depicted by **1**), the method increments the variable (depicted by **2**), and, due to changing the variable, the UI will automatically be updated (depicted by **3**):

```
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {
    private int currentCount = 0;

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


Figure 6.1: The flow of the Counter component

The `counter` component is implemented inside the `BlazorWebApp.Client` project, which is the WebAssembly project. In that project, we should include all the components we want to run in WebAssembly. The `BlazorWebApp` project then references the `BlazorWebApp.Client` project, so that it finds all the components and can run them as Blazor Server components if we want to.

Weather

The next component we will take a look at is the `Weather` component. It's located in the `BlazorWebApp.Client` project, in the `Pages/Weather.razor` folder.

The file initially looks like this:

```
@page "/weather"
```

Just as with the `counter` component, we first define a route. The HTML part of the `Weather` component looks like this:

```
<PageTitle>Weather</PageTitle>

<h1>Weather</h1>

<p>This component demonstrates showing data.</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>
}
```

If we don't have any forecasts, it will show **Loading...**, and as soon as we have some data, it will render a table showing the data.

Then the code section generates some mock data, which looks like this:

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

    protected override async Task OnInitializedAsync()
    {
        // Simulate asynchronous loading to demonstrate
        // streaming rendering
        await Task.Delay(500);

        var startDate = DateOnly.FromDateTime(DateTime.Now);
        var summaries = new[] { "Freezing", "Bracing", "Chilly", "Cool",
            "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching" };
        forecasts = Enumerable.Range(1, 5).Select(index =>
            new WeatherForecast
        {
            Date = startDate.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        }).ToArray();
    }

    private class WeatherForecast
    {
        public DateOnly Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    }
}
```

When the page is initialized, the `WeatherForecast` array is populated with random data.

This reminds me that someone (as a joke) added a bug report to one of Dan Roth's repos on GitHub, reporting, "*Weather forecast is unreliable.*"

The conversation continued, "*The weather forecast feature was completely accurate during my trip to London, but has proven to be misleading in California. Counter and overall purpleness are great.*" Dan Roth replied, "*Thanks for the feedback! I'll get in touch with the folks on the .NET Core framework team to make sure that System.Random does a better job of taking California weather patterns into account* 😉."

This is one of many reasons why I love the .NET community.

As we can see, using Razor syntax allows us to mix code with HTML seamlessly. The code checks whether there is any data – if so, it will render the table; if not, it will show a loading message. The component will update automatically once we have data. There are component libraries that can make this process a bit simpler, which we will look at in the next chapter, *Chapter 7*, *Creating Advanced Blazor Components*.

Now that we know how the sample template is implemented, it is time to dive deeper into Razor syntax.

Learning Razor syntax

One of the things I like about the Razor syntax is that it is easy to mix code and HTML tags. By having the code close to the markup, it is, in my opinion, easier to follow and understand. The syntax is very fluid; the Razor parser understands when the code stops and markup begins, which means we don't need to think about it that much. It is also not a new language; instead, we can leverage our existing C# and HTML knowledge to create our components. This section will contain a lot of theory to help us understand the Razor syntax.

To transition from HTML to code (C#), we use the `@` symbol. There are a handful of ways we can add code to our file, which we'll explore in the following sections:

- Razor code blocks
- Implicit Razor expressions
- Explicit Razor expressions
- Expression encoding
- Directives

Let's get started!

Razor code blocks

We have already seen some Razor component code blocks. A Razor component code block looks like this:

```
@code {
    //your code here
}
```

If we wish, we can skip the code keyword like so:

```
@{
    //your code here
}
```

Inside those curly braces, we can mix HTML and code like this:

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
    RenderName("Steve Sanderson");
    RenderName("Daniel Roth");
}
```

Notice how the `RenderName()` method transitions from code into the paragraph tags and back to code; this is an implicit transition.

If we want to output text without having an HTML tag, we can use the `text` tag instead of using the paragraph tags, as shown in the following example:

```
<text>Name: <strong>@name</strong></text>
```

This would render the same result as the previous code but without the paragraph tags, and the `text` tag would not be rendered.

Implicit Razor expressions

Implicit Razor expressions are when we add code inside HTML tags. We have already seen this in the `Weather` example:

```
<td>@forecast.Summary</td>
```

We start with a `<td>` tag, then use the `@` symbol to switch to C#, and switch back to HTML with the `end` tag. We can use the `await` keyword together with a method call, but, other than that, implicit Razor expressions cannot contain any spaces.

We cannot call a generic method using implicit expressions since `<>` would be interpreted as HTML. Hence, to solve this issue, we can use explicit expressions.

Explicit Razor expressions

We can use **explicit Razor expressions** if we want to use spaces in the code. To do this, write the code with the `@` symbol followed by parentheses `( )`. So, it would look like this: `@()`.

In this sample, we subtract `7` days from the current date:

```
<td>@(DateTime.Now - TimeSpan.FromDays(7))</td>
```

We can also use explicit Razor expressions to concatenate text; for example, we can concatenate text and code like this:

```
<td>Temp@(forecast.TemperatureC)</td>
```

The output would then be `<td>Temp42</td>`, assuming the temperature is 42.

Plus, using explicit expressions, we can easily call generic methods by using this syntax:

```
<td>@(MyGenericMethod<string>())</td>
```

The Razor engine knows whether we are using code or not. It also makes sure to encode strings to HTML when outputting them to the browser, a process called **expression encoding**.

Expression encoding

If we have HTML as a string, it will be escaped by default. Take this code, for example:

```
@("<span>Hello World</span>")
```

Since we are going to store our blog posts in a database, it becomes important to be able to render those blog posts as HTML.

But doing it this way, the rendered HTML would look like this:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

To output the actual HTML from a string (something we will do later, in *Chapter 7*, *Creating Advanced Blazor Components*), you can use this syntax:

```
@((MarkupString)"<span>Hello World</span>")
```

Using `MarkupString`, the output will be HTML, showing the HTML tag span. In some cases, one line of code isn't enough; then, we can use code blocks. These are the `@{}` mentioned at the beginning of this section.

Directives

There are a bunch of directives that change the way a component gets parsed or can enable functionality. These are reserved keywords that follow the `@` symbol. We will go through the most common and useful ones.

I find that it is pretty nice to have the layout and the code inside the same `.razor` file.

Note that we can use code-behind to write our code to get a bit more separation between the code and layout. Later in this chapter, we will look at how to use code-behind instead of Razor syntax for everything. For now, the following examples will look at how we would do the same directives using both Razor syntax and code-behind.

Adding an attribute

To add an attribute to our page, we can use the `attribute` directive:

```
@attribute [Authorize]
```

If we were using a code-behind file, we would use the following syntax instead:

```
[Authorize]
public partial class SomeClass {}
```

Adding an interface

To implement an interface (`IDisposable` in this case), we would use the following code:

```
@implements IDisposable
```

Then, we would implement the methods the interface needs in a `@code{}` section.

To do the same in a code-behind scenario, we would add the interface after the class name, as shown in the following example:

```
public partial class SomeClass : IDisposable {}
```

Inheriting

To inherit another class, we should use the following code:

```
@inherits TypeNameOfClassToInheritFrom
```

To do the same in a code-behind scenario, we would add the class we want to inherit from after the class name:

```
public class SomeClass : TypeNameOfClassToInheritFrom {}
```

Generics

We can define our component as a generic component. Generics allow us to define the data type so the component works with any data type.

To define a component as a generic component, we add the `@typeparam` directive. Then, we can use the type in the code of the component like this:

```
@typeparam TItem
@code
{
      [Parameter]
      public List<TItem> Data { get; set; }
}
```

Generics are super-powerful when creating reusable components; this makes our components reusable for different data types. We will return to generics in *Chapter 8*, *Building Forms with Validation*.

Changing the layout

If we want to have a specific layout for a page (not the default one specified in the `Routes.razor` file), we can use the `@layout` directive:

```
@layout AnotherLayout
```

This way, our component will use the specified layout (this only works for components with the `@page` directive).

Setting a namespace

By default, the component's namespace will be the name of the default namespace of our project, plus the folder structure. If we want our component to be in a specific namespace, we can use the following:

```
@namespace Another.NameSpace
```

Setting a route

We have already touched on the `@page` directive. If we want our component to be directly accessed using a URL, we can use the `@page` directive:

```
@page "/theurl"
```

The URL can contain parameters, subfolders, and much more, which we will discuss later in this chapter.

Adding a using statement

To add a namespace to our component, we can use the `@using` directive:

```
@using System.IO
```

If there are namespaces that we use in several of our components, then we can add them to the `Imports.razor` file instead. This way, they will be available in all the components we create.

Note

If you want to dive further into directives, you can find more information here: `https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-10.0#directives`.

Now we know more about how Razor syntax works. Don't worry; we will have plenty of time to practice it. There is one more directive that I haven't covered in this section, and that is `inject`. We first need to understand what **dependency injection** (**DI**) is and how it works, which we will see in the next section.

Understanding dependency injection

DI is a software pattern and technique used to implement **Inversion of Control** (**IoC**).

IoC is a generic term that means we can indicate that the class needs a class instance instead of letting our classes instantiate an object. We can say that our class wants either a specific class or a specific interface. The creation of the required instance happens elsewhere, and the IoC container is responsible for determining which implementation is created.

When it comes to DI, it is a form of IoC in which an object (class instance) is passed through constructors, parameters, or service lookups.

> **Note**
>
> Here is a great resource if you want to dive deeper into DI in .NET: `https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection`.

In Blazor, we can configure DI by providing a way to instantiate an object; this is a key architecture pattern that we should use. It could look something like this:

```
builder.Services.AddScoped<IBlogRepository,
    BlogApiEntityFrameworkDirectAccess>();
```

Here, we say that if any class wants `IBlogRepository`, the application should instantiate an object of the `BlogApiEntityFrameworkDirectAccess` type. In this case, we are using an interface; instead, we could just say:

```
builder.Services.AddScoped<BlogApiEntityFrameworkDirectAccess>();
```

In this case, when we request a `BlogApiEntityFrameworkDirectAccess` object, the DI container returns an instance of that type. If there is only one implementation of a component, there is no need to create an interface for it.

In the previous chapter, *Chapter 5, Managing State – Part 1*, we created an `IBlogRepository` interface. On the server, we want this interface to resolve to an instance of

`BlogApiEntityFrameworkDirectAccess`. Since the server has direct access to the database, it can use this implementation directly.

Let's add it to our `Program.cs`.

1. In the `BlazorWebApp` project, open `Program.cs`.
2. Just above `var app = builder.Build();` add:

```
builder.Services.AddScoped<IBlogRepository,
BlogRepositoryEntityFrameworkDirectAccess>();
```

So now, on the server, when we ask for an `IBlogRepository` interface, dependency injection will make sure to create a new object of the type `BlogRepositoryEntityFrameworkDirectAccess` and return it.

The dependency injection works differently depending on whether we are running on the server (Blazor Server / Interactive Server) or on the client (Blazor WebAssembly / Interactive WebAssembly). We need to specify how this should work on each of these platforms. Right now, we only have it set up in the server version.

When we implement the WebAssembly version, DI will return a different class instead.

There are many advantages to using DI. Our dependencies are loosely coupled, so we don't instantiate another class in our class. Instead, we ask for an instance, which makes it easier to write tests and change implementations depending on the platform.

Any external dependencies will be much more apparent since we must pass them into the class. We can also set the way we should instantiate the object in a central place. This is done by configuring the DI in `Program.cs`.

We can configure the creation of objects in different ways, such as the following:

- Singleton
- Scoped
- Transient

Let's review each option.

Singleton

When we use a **singleton**, the object is the same for all site users. The object is only created once.

To configure a singleton service, use the following:

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

We should use a singleton when we want to share our object with all the users of our site, but beware, because since the state is shared, it can lead to issues if the object stores data specific to an individual user or a session. Once this data is changed by one user, the change is reflected for all users who might be using the application simultaneously. It may also lead to data being shared unintentionally.

I was once asked in a podcast if I had ever done anything really bad during my career. This is the one. I used a singleton, which exposed customer data to another customer. Luckily, it was a timing issue, so only one customer was affected, and we rolled out a fix in minutes.

Please learn from my mistake and be aware when using singletons.

Scoped

When we use **scoped**, a new object is created once per connection, and since Blazor Server needs a connection to work, it will be the same object as long as the user has a connection.

Blazor WebAssembly does not have a server-side circuit or connection in the same way as Blazor Server. The application runs inside the user's browser, so there is only one user for that application instance.

Because of that, a scoped service in Blazor WebAssembly lives for as long as the WebAssembly application is running in the browser. In practice, that makes it behave much like a singleton within that browser tab. So even though we still register the service as scoped, the scope is different from Blazor Server. If we use scoped, it will work the same way as a singleton in Blazor WebAssembly, since we only have one user and everything runs in the browser.

It is worth noting that running Blazor WebAssembly in a new browser will start a new instance of the WebAssembly application, including new instances of objects. The recommendation is still to use scoped if the idea is to scope a service to the current user. This makes it easier to move code between Blazor Server and Blazor WebAssembly and gives a bit more context on how the service is supposed to be used.

To configure a scoped service, use the following:

```
services.AddScoped<IWeatherForecastService, WeatherForecastService>();
```

We should use scoped if we have data that belongs to the user, as we can keep the user's state by using scoped objects. More on that in *Chapter 13, Managing State – Part 2*.

Plus, if creating the user state is heavy or time-consuming, this could be a time-saver.

It's worth mentioning here that the new *per component* model will create a `SignalR` connection if there is any component currently running in `InteractiveServer` mode. If we navigate to a new page without any `InteractiveServer` components, the connection will eventually be disconnected. This means that the state will also be removed. So, when using the *per component* model, we need to make sure not to save any important information in a scoped variable unless we persist it in some other way as well.

Transient

When we use **transient**, a new object is created every time we ask for it.

To configure a transient service, use the following:

```
services.AddTransient<IWeatherForecastService, WeatherForecastService>();
```

We can use transient if we don't need to keep any state and don't mind the object being created every time we ask for it.

Now that we know how to configure a service, we need to start using the service by injecting it.

Injecting the service

There are three ways to inject a service.

The first way is to use the `@inject` directive in the Razor file:

```
@inject WeatherForecastService ForecastService
```

This will make sure we have access to `WeatherForecastService` in our component.

The second way is to create a property by adding the `Inject` attribute if we are using code-behind:

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

The third way is for when we want to inject a service into another service – then, we need to inject the services using the constructor:

```
public class MyService
{
    public MyService(WeatherForecastService
      weatherForecastService)
    {
```

```
    }
}
```

Now we know how DI works and why we should use it. You can also use primary constructors to add dependency injection into your classes. In .NET 7, using a scoped service meant that the data was accessible as long as the connection (or circuit) was active. But with .NET 8 and beyond, it changes just slightly depending on the render mode. Let's look at that next.

Changing the render mode

The biggest change for render modes came in .NET 8 – the ability to change render mode in the same app. In .NET 7, we had to choose one or the other, but now we can change it as we see fit. .NET 10 adds nothing new to this other than performance improvements, which is very nice, of course. Perhaps if a specific page is not interactive, we can use the new **SSR** (**Server-Side Rendering**). This is very similar to WebForms or MVC. The page gets rendered on the server. No additional interactivity will work, which means that features such as event handling, data binding, and component updates will not function after the page has been rendered.

We can set the render mode on each component, or we can do it when we use the component. When we create the project, we select which interactive render mode we want.

Let's take a look at the different options:

- **None**: No interactivity, only static rendered files, no SignalR, and no WebAssembly. Using this option, we can use both Static Server-Side Rendering and Streaming Server-Side Rendering.
- **Server**: This will give us access to interactivity using Blazor **Server**, not WebAssembly.
- **WebAssembly**: This will give us access to interactivity using Blazor **WebAssembly**, not Blazor Server.
- **Auto** (Server and WebAssembly): Gives us the ability to use both Server and WebAssembly.

We can also set the interactivity location to per page/component, which means that the default behavior of the site is static, and we need to specify on each component if we want to use interactivity. We can also set the interactivity location globally, which we do by setting the render mode on the `Routes` component like this:

```
<Routes @rendermode="@InteractiveAuto" />
```

To change the render mode per component, we can use the syntax above or set it per component like this:

```
@rendermode InteractiveAuto
```

The configuration for render modes depends on how you set up your application. If you are using the Blazor Web App Template, by default, all components are rendered using server prerendering. This means that the component is first rendered on the server and then pushed to the web browser. SignalR or WebAssembly starts up, and the component is rendered again, making additional calls to the database, for example.

When I started using Blazor, I turned off prerendering. Back then, the only available version was Blazor Server. But if we don't pre-render the content, we will have bad, or actually, no SEO. This is why many other frontend frameworks are moving towards SSR in different forms. Prerendering makes the page feel snappier, and with the addition of `StreamRendering`, there is no real disadvantage to having it turned on.

We can, however, disable prerendering by doing it like this:

```
<Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

We will have plenty of time to learn more about render modes during the course of the book. In .NET 7, there were more predefined templates, but in .NET 10, the same scenarios can be achieved by combining render mode and interactivity settings.

Coming from an old template

Recently, during a live stream on the Coding After Work channel, we updated one of our older sites from .NET 7 to .NET 10. The good news is that it only took about two hours, even with some back-and-forth with the viewers. In that time, we managed to move from a Blazor Server-only setup to the new Blazor Web App template, which also gives us the option to run parts of the app as WebAssembly if we want to.

Table 6.1 shows how the .NET 7 templates map to the .NET 10 templates and configuration options.

.NET 7 template	.NET 10 template	Interactive render mode	Interactivity location
Blazor Server App	Blazor Web App	Server	Global
Blazor WebAssembly App	Blazor WebAssembly Standalone App		
Blazor WebAssembly (ASP.NET Core Hosted)	Blazor Web App	WebAssembly	Global

Table 6.1: .NET 7 to .NET 10 templates

If you are new to Blazor, this table has no significance, but if you have worked with Blazor in .NET 7 and want to use a project template that you used in .NET 7, this is how to choose the new template.

In this chapter, we have mentioned code-behind a couple of times. In the next section, we will look at how we can use code-behind with Razor files and also how to skip the Razor files altogether.

Figuring out where to put the code

We have seen examples of writing code directly in the Razor file. I prefer doing that unless the code gets too long or too complicated. I always lean in favor of readability.

There are four ways we can write our components:

- In the Razor file
- In a partial class
- Inheriting a class
- Only code

Let's go through each item on this list in more detail.

In the Razor file

If we are writing a file that is not very complex, it would be nice not to switch files when writing components. As we already covered in this chapter, we can use the `@code` directive to add code directly to our Razor file.

If we want to move the code to a code-behind file, then it is only the directives that we need to change. For the rest of the code, we can just move it to the code-behind class, while the template (HTML markup and C#) stays in the `.razor` file. When I started with Blazor, writing code and markup in the same file felt strange, coming from an MVC world where the separation between code and markup is a big part of the way to use MVC. But I suggest you try it out when developing your web apps.

At work, we started using code-behind but switched to writing code in the `.razor` file instead, and we haven't looked back since. However, many developers prefer code-behind, separating code from the layout. For that, we can use a partial class.

In a partial class

We can create a partial class with the same filename as the Razor file and add `.cs`.

If you have downloaded the source code (or you can check the code on GitHub), you can look at `WeatherCodeBehind.razor.cs` in the `MyBlog/Examples` folder. I have moved all the code to the code-behind file; the result when compiling this will be the same as if we kept the code in the Razor file. It is just a matter of preference.

The code-behind looks like this:

```
namespace BlazorWebApp.Components.Pages;
public partial class WeatherWithCodeBehind
{
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        // Simulate asynchronous loading to demonstrate
        // streaming rendering
        await Task.Delay(500);

        var startDate = DateOnly.FromDateTime(DateTime.Now);
        var summaries = new[] { "Freezing", "Bracing", "Chilly", "Cool",
            "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching" };
        forecasts = Enumerable.Range(1, 5).Select(index =>
```

```
            new WeatherForecast
        {
            Date = startDate.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        }).ToArray();
    }

    private class WeatherForecast
    {
        public DateOnly Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    }
}
```

Since we are using a partial class, there is no need to connect the Razor file with the code-behind. If it has the same name and the same namespace, it will just work. We can mix where we put the code. You don't have to have all the code in the code-behind, even though I would probably pick one and stick with it. If you prefer using code-behind, this is the way you want to do it.

This is not the only way to use a code-behind file; we can also inherit from a code-behind file.

Inheriting a class

We can also create a completely different class (the common pattern is to call it the same thing as the Razor file and add `Model` at the end) and inherit it in our Razor file. For that to work, we need to inherit from `ComponentBase`. In the case of a partial class, the class already inherits from `ComponentBase`, since the Razor file does that.

Fields must be protected or public (not private) for the page to access the fields. I recommend using the partial class if we don't need to inherit from a base class.

This is a snippet of the code-behind class declaration:

```
public class WeatherWithInheritsModel:ComponentBase
```

We'll need to inherit from `ComponentBase` or from a class that inherits from `ComponentBase`.

In the Razor file, we will use the `@inherits` directive:

```
@inherits WeatherWithInheritsModel
```

The Razor file will now inherit from our code-behind class (this was the first way available to create code-behind classes)

Both the partial and inherit options are simple ways of moving the code to a code-behind file. Inheriting a model was the first available way, but as I mentioned, use partial classes instead if you prefer code-behind. Another option is to skip the Razor file and use only code entirely.

Only code

Visual Studio will use source generators to convert the Razor code into C#. We will dig deeper into source generators in *Chapter 20*, *Examining Source Generators*. The Razor file will generate code at compile time. We can skip the Razor step if we want to and write our layout completely in code.

This file (`CounterWithoutRazor.cs`) is available in the repo on GitHub.

The counter example would look like this:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Rendering;
using Microsoft.AspNetCore.Components.Web;
namespace BlazorWebApp.Component.Pages;

[Route("/CounterWithoutRazor")]
public class CounterWithoutRazor : ComponentBase
{
    protected override void BuildRenderTree
      (RenderTreeBuilder builder)
    {
        builder.AddMarkupContent(0, "<h1>Counter</h1>\r\n\r\n");
        builder.OpenElement(1, "p");
        builder.AddContent(2, "Current count: ");
        builder.AddContent(3, currentCount);
        builder.CloseElement();
        builder.AddMarkupContent(4, "\r\n\r\n");
        builder.OpenElement(5, "button");
        builder.AddAttribute(6, "class", "btn btn-primary");
        builder.AddAttribute(7, "onclick", EventCallback.Factory
            .Create<MouseEventArgs>(this, IncrementCount));
```

```
            builder.AddContent(8, "Click me");
            builder.CloseElement();
        }
        private int currentCount = 0;
        private void IncrementCount()
        {
            currentCount++;
        }
    }
```

The Razor file will first be converted into something roughly the same as the previous code, and then the code will be compiled. It adds the elements one by one, which, in the end, renders the HTML.

The numbers in the code are how Blazor keeps track of each element in the render tree. Some prefer to write the code as shown in the previous code block rather than using the Razor syntax; there are even efforts in the community to simplify the process of manually writing the `BuildRenderTree()` function. Some of Microsoft's built-in components are also built in this way.

I recommend never writing this manually, but I've kept it in the book because it shows how Razor files get compiled. Now that we know how to use code-behind, let's look at the lifecycle events of Blazor and when they are executed.

Exploring lifecycle events

We can use a couple of lifecycle events to run our code. In this section, we will go through them and see when we should use them. Most lifecycle events have two versions: synchronous and asynchronous.

OnInitialized and OnInitializedAsync

The first time the component is loaded, `OnInitialized()` is called, then `OnInitializedAsync()`. This is a great method to load any data, as the UI has not yet been rendered. If we are doing long-running tasks (such as getting data from a database), we should put that code in the `OnInitializedAsync()` method.

These methods will only run once when the component is created. If you want to update the UI when a parameter changes, see `OnParametersSet()` and `OnParametersSetAsync()`.

OnParametersSet and OnParametersSetAsync

`OnParametersSet()` and `OnParametersSetAsync()` are called when the component is initialized (after `OnInitialized()` and `OnInitializedAsync()`) and whenever the value of a parameter changes.

If we, for example, load data in the `OnInitialized()` method, but it uses a parameter, the data won't be reloaded if the parameter is changed, since `OnInitialized()` will only run once. We need to trigger a reload of the data in `OnParametersSet()` or `OnParametersSetAsync()`, or move the loading to that method.

OnAfterRender and OnAfterRenderAsync

After the component renders, the `OnAfterRender()` and `OnAfterRenderAsync()` methods are called. When the methods are called, all the elements are rendered, so if we want or need to call any JavaScript code, we have to do that from these methods (we will get an error if we try to make a JavaScript interop from any of the other lifecycle event methods). There is a limitation of prerendering. When the component prerenders, there is nothing connected to the web browser, and we will not be able to run any JavaScript. We also have access to a `firstRender` parameter, so we control when we want to run our code.

ShouldRender

`ShouldRender()` is called when our component is re-rendered; if it returns `false`, the component will not be rendered again. The component will always render once, even if this method returns `false`.

`ShouldRender()` does not have an asynchronous option.

Now we know when the different lifecycle events happen and in what order. A component can also have parameters, which allows us to reuse them with different data.

Understanding parameters

A `parameter` makes it possible to send a value to a component. To add a parameter to a component, we use the `[Parameter]` attribute on the `public` property:

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

The syntax for this is the same if we use a code-behind file. We can add a parameter to the route using the `@page` directive by specifying it in the route:

```
@page "/parameterdemo/{MyParameter}"
```

In this case, we have to have a parameter specified with the same name as the name inside the curly braces. To set the parameter in the `@page` directive, we go to `/parameterdemo/THEVALUE`.

There are cases where we want to specify another type instead of a string (string is the default). We can add the data type after the parameter name, like this:

```
@page "/parameterdemo/{MyParameter:int}"
```

This will match the route only if the data type is an integer. We can also pass parameters using cascading parameters. We can have more than one page directive per component if we want to handle more than one route.

Cascading parameters

If we want to pass a value to multiple components, we can use a cascading parameter.

Instead of using `[Parameter]`, we can use `[CascadingParameter]` like this:

```
[CascadingParameter]
public int MyParameter { get; set; }
```

To pass a value to the component, we surround it with a `CascadingValue` component like this:

```
<CascadingValue Value="MyProperty">
    <ComponentWithCascadingParameter/>
</CascadingValue>
@code {
    public string MyProperty { get; set; } = "Test Value";
}
```

`CascadingValue` is the value we pass to the component, and `CascadingParameter` is the property that receives the value.

As we can see, we don't pass any parameter values to the `ComponentWithCascadingParameter` component; the cascading value will match the parameter with the same data type. If we have

multiple parameters of the same type, we can specify the name of the parameter in the component with the cascading parameter, like this:

```
[CascadingParameter(Name = "MyCascadingParameter")]
```

We can also do so for the component that passes `CascadingValue`, like this:

```
<CascadingValue Value="MyProperty" Name="MyCascadingParameter">
    <ComponentWithCascadingParameter/>
</CascadingValue>
```

If we know that the value won't change, we can specify that by using the `IsFixed` property:

```
<CascadingValue Value="MyProperty" Name="MyCascadingParameter"
    IsFixed="True">
    <ComponentWithCascadingParameter/>
</CascadingValue>
```

This way, Blazor won't look for changes, which is more efficient if we know the values won't change. The cascading values/parameters cannot be updated upward but are updated only downward. This means that to update a cascading value, we need to implement it in another way, for example, using events. Updating it from inside the component won't change any components that are higher in the hierarchy.

In *Chapter 7*, *Creating Advanced Blazor Components*, we will look at events, which are one way to solve the problem of updating a cascading value.

Phew! This has been an information-heavy chapter, but now that we know the basics of Blazor components, it is time to build one!

Writing our first component

The first component we will build will display all the blog posts on the site. At this point, we haven't written any blog posts yet, but we'll fix that shortly so we can focus on building something real.

In *Chapter 5*, *Managing State – Part 1*, we created a database and a repository. Now it's time to put them to use.

We'll start by updating the existing `Home.razor` page to list blog posts:

1. In the `BlazorWebApp.Client` project, open `Pages/Home.razor`.
2. Replace the contents of that file with the following code:

```
@page "/"
@using BlazorWebApp.Client.Interfaces
@using BlazorWebApp.Client.Models
@inject IBlogRepository _repository
@code {
}
```

   If we start from the top, we can see a page directive. It will ensure that the component is shown when the route is `"/"`.

   Then, we have two `@using` directives, bringing in the namespaces so we can use them in the Razor file.

   Then, we inject our API (using DI) and name the instance `_repository`.

3. Add a variable that holds all our posts. To do this, in the `code` section, add the following:

```
protected List<BlogPost> posts = new();
```

4. Now, we need to load the data. To load posts, add the following in the `code` section:

```
protected override async Task OnInitializedAsync()
{
    posts = await _repository.GetBlogPostsAsync(10, 0);
    await base.OnInitializedAsync();
}
```

   Now, when the page loads, the posts will be loaded as well: `10` posts and page `0` (the first page).

5. Under the `@inject` row, add the following code:

```
<ul>
    @foreach (var p in posts)
    {
        <li>@p.Title</li>
    }
</ul>
```

Here we add an **unordered list** (**UL**); inside that, we loop over `blog posts` and show the title.

6. Now, we can run the application by pressing *F5* (**Debug** | **Start Debugging**). This will start up Aspire (along with everything we need) and our Blazor site.
7. We should see an empty page, since we don't have any blog posts right now.
8. But the page will also break. (If it doesn't, try reloading the page.)
What is up with that?

Well, right now we are running our site as InteractiveAuto, which means it will prerender (in our case, an empty page) and then switch over to WebAssembly. But we haven't added an API yet, and we don't have any DI set up for the WebAssembly project, so this is totally normal. I wanted to show you this because this is a common thing I help my customers with.

At this point in the book, we are not ready for WebAssembly, so let's change our site to use InteractiveServer instead.

9. In the `BlazorWebApp` project, open `Components/App.razor`. There we can find this line: `<Routes @rendermode="InteractiveAuto" />`. Change it to:

```
<Routes @rendermode="InteractiveServer" />
```

This is one of the amazing things about the Blazor Web App template; it is *that* easy to change the render mode globally.

10. Rerun the project (*F5*) and make sure it works (it should still be an empty page, but there should be no errors).
11. Let's add some data. From the Aspire portal, open **pgAdmin**.
12. Navigate to the left menu, then select **Servers** | **Postgres** | **Databases** | **myBlogDb**.

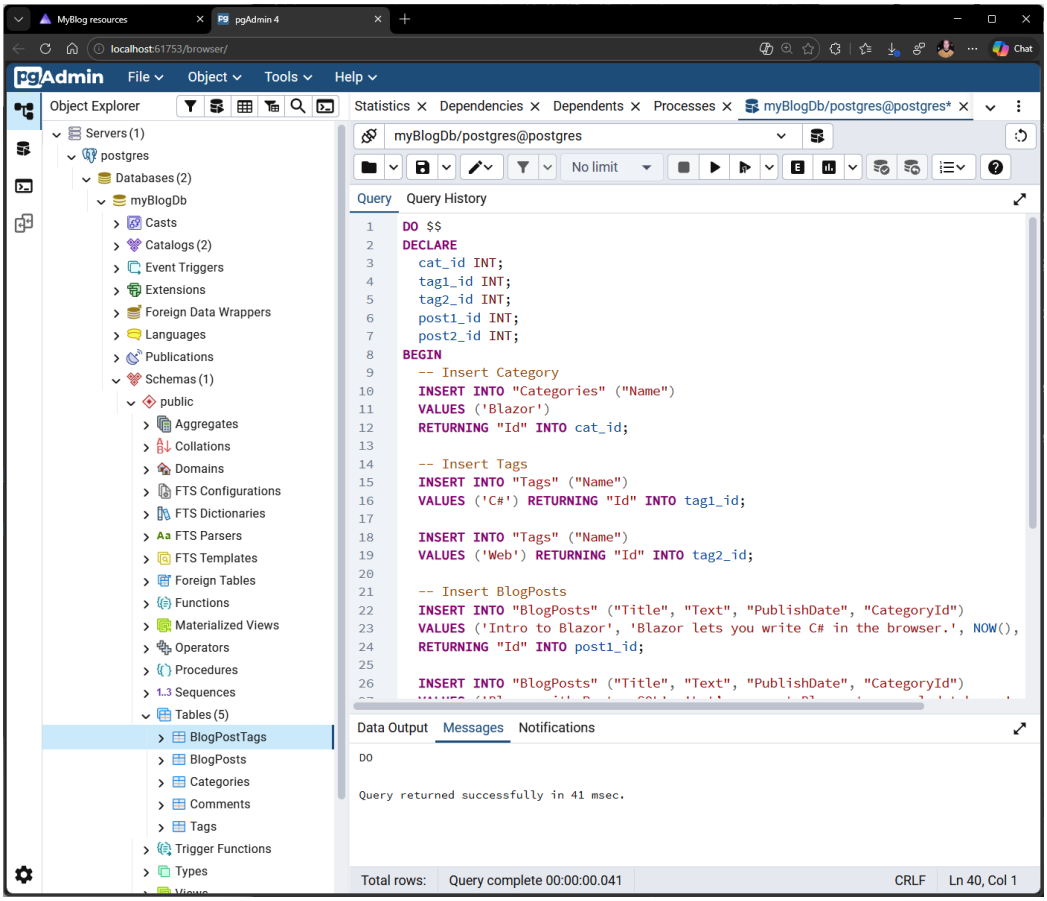


Figure 6.2: pgAdmin interface

13. Press *Alt* + *Shift* + *Q* to open the Query tool.

14. In the repo, you will find a `DatabaseScript.sql` file (so you don't have to type it in). Open the file and paste its contents into the Query tool editor, then run the query. This is what is in the file:

```
DO $$
DECLARE
  cat_id INT;
  tag1_id INT;
  tag2_id INT;
  post1_id INT;
  post2_id INT;
```

```
BEGIN
  -- Insert Category
  INSERT INTO "Categories" ("Name")
  VALUES ('Blazor')
  RETURNING "Id" INTO cat_id;

  -- Insert Tags
  INSERT INTO "Tags" ("Name")
  VALUES ('C#') RETURNING "Id" INTO tag1_id;

  INSERT INTO "Tags" ("Name")
  VALUES ('Web') RETURNING "Id" INTO tag2_id;

  -- Insert BlogPosts
  INSERT INTO "BlogPosts" ("Title", "Text",
    "PublishDate", "CategoryId")
  VALUES ('Intro to Blazor', 'Blazor lets you write
    C# in the browser.', NOW(), cat_id)
  RETURNING "Id" INTO post1_id;

  INSERT INTO "BlogPosts" ("Title", "Text",
    "PublishDate", "CategoryId")
  VALUES ('Blazor with PostgreSQL', 'Let's connect Blazor
    to a real database.', NOW(), cat_id)
  RETURNING "Id" INTO post2_id;

  -- Insert BlogPost-Tag relationships
  INSERT INTO "BlogPostTags" ("BlogPostsId", "TagsId") VALUES
    (post1_id, tag1_id), -- Post 1 -> C#
    (post1_id, tag2_id), -- Post 1 -> Web
    (post2_id, tag2_id); -- Post 2 -> Web

  -- Insert a Comment
  INSERT INTO "Comments" ("BlogPostId", "Date", "Text", "Name")
  VALUES (post1_id, NOW(), 'Great post!', 'Alice');
END $$;
```

15. Run the script (*F5*). Now we should have blog posts, tags, comments, and categories!
16. Reload the Blazor page, and you should now see a couple of blog posts!

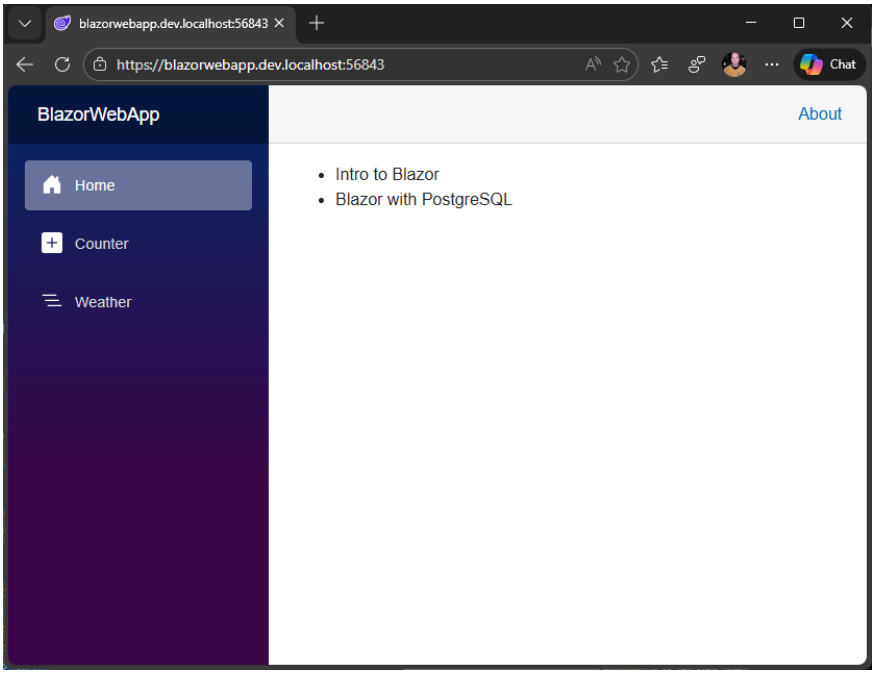


Figure 6.3: Blog posts are visible

Great job, we have created our first component!

We will return to this when we implement a web API for WebAssembly in *Chapter 9*, *Creating an API*.

Summary

In this chapter, we learned a lot about Razor syntax, something we will use throughout the rest of the book. We explored how components are structured, how code and markup can live together, and when it makes sense to move code into a code-behind file. We also looked at dependency injection, render modes, lifecycle events, and parameters, all core building blocks when working with Blazor.

The key takeaway from this chapter is that everything in Blazor is a component. Pages are components. Reusable UI parts are components. Even layouts are components. Once we understand how components work, we understand how Blazor works.

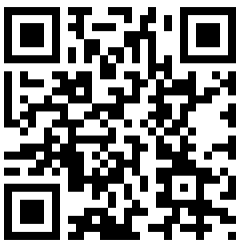
We also saw how render modes affect behavior and why it matters whether we run on the server or in WebAssembly. That knowledge will become more important as we continue building features and start mixing hosting models.

Finally, we built our first real component that talks to our repository and shows actual data from the database. That is a big step. We moved from theory to something that looks like a real application (well, maybe not yet).

In the next chapter, we will take this foundation and look at more advanced component scenarios, making our components more reusable and powerful.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

7

Creating Advanced Blazor Components

In the previous chapter, we learned all the basics of creating a component. Now it's time to learn how to take our components to the next level.

This chapter focuses on features that make our components reusable. Reusable components help us avoid writing the same code over and over again, and make it easier for teams to maintain consistent UI and behavior across an application.

To achieve this, we will explore several important features of Blazor components. We will look at binding, which allows components to share and synchronize data. We will also explore Actions and EventCallback, which enable components to communicate changes and trigger logic in other components. Another important feature is RenderFragment, which allows us to pass Razor content into components and makes it possible to build highly flexible UI components. Finally, we will explore several built-in components and framework features that add functionality beyond what plain HTML provides.

As such, we will cover the following topics:

- Exploring binding
- Diving into Actions and EventCallback
- Using RenderFragment
- Exploring the built-in components

Technical requirements

In this chapter, we will start building our components. For this, you'll need the code we developed in *Chapter 6*, *Understanding Basic Blazor Components*. You are good to go if you have

followed the instructions in the previous chapters. If not, then make sure you clone or download the repository. The starting point for this chapter can be found in the `Chapter06` folder.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter07`.

Exploring binding

When building applications, data is important, and we can use binding to show or change data. **Binding** is the process of connecting data in our component to the user interface so that changes in one are reflected in the other. By using binding, you can connect variables within a component (so that they update automatically) or set a component attribute. Perhaps the most fantastic thing is that by using binding, Blazor understands when it should update the UI and the variable (if the data changes in the UI). In Blazor, there are two different ways that we can bind values to components, as follows:

- One-way binding
- Two-way binding

By using bindings, we can send information between components and ensure we can update a value when we want to.

One-way binding

We have already touched on one-way binding in *Chapter 6*, *Understanding Basic Blazor Components*, even though we never called it *binding*. In the `Counter` component, updating the `currentCount` variable causes the UI to update. This is an example of one-way binding, where data flows from the component to the UI.

Let's look at the `Counter` component again and continue building on it in this section. In this section, we will combine parameters with one-way data flow and later see how full binding is achieved.

The `Counter.razor` example looks like this:

```
@page "/counter"

<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">
    Click me</button>
```

```
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The `Counter` component will display the current count and a button to increment it. This is one-way binding. Even though the button can change the value of `currentCount`, it only flows in one direction to the screen.

Since this part is designed to demonstrate the functionality and theory and is not part of the finished project we are building, you don't have to write or run this code. The source code for these components is available on GitHub.

Next, we can add a parameter to the `Counter` component. The code will then look like this:

```
@page "/counterwithparameter"

<h1>Counter</h1>
<p>Current count: @CurrentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">
    Click me</button>
@code {
    [Parameter]
    public int IncrementAmount { get; set; } = 1;
    [Parameter]
    public int CurrentCount { get; set; } = 0;
    private void IncrementCount()
    {
        CurrentCount += IncrementAmount;
    }
}
```

The code sample has two parameters, one for `CurrentCount` and one for `IncrementAmount`. By adding parameters to the components, we can change their behavior. This sample is, of course, a bit silly. The chances are you won't have any use for a component like this that just counts up when you press a button. But it illustrates the idea very well.

We can now take the component and add it to another component. This is how we can create a reusable component and change its behavior by changing the value of the parameters. We change its behavior like this:

```
@page "/parentcounter"
<CounterWithParameter IncrementAmount="@incrementamount"
    CurrentCount="@currentcount"/>
The current count is: @currentcount
@code {
    int incrementamount = 10;
    int currentcount = 0;
}
```

In this sample, we have two variables, `incrementamount` and `currentcount`, that we pass into our `CounterWithParameter` component.

If we were to run this, we would see a `Counter` component that counts in increments of `10`. However, the `currentcount` variable will not be updated, since it is only a one-way binding (one direction).

To help us with that, we can implement two-way binding so that our parent component is notified of any changes.

Two-way binding

Two-way binding binds values in both directions, and our `Counter` component will be able to notify our parent component of any changes. In the next chapter, *Chapter 8*, *Building Forms with Validation*, we will talk even more about two-way binding.

For now, to make our `CounterWithParameter` component bind in two directions, we need to add an `EventCallback<T>` struct. The name must consist of the parameter's name followed by `Changed`. This way, Blazor will update the value if it changes. In our case, we would need to name it `CurrentCountChanged`. The code would then look like this:

```
[Parameter]
public EventCallback<int> CurrentCountChanged { get; set; }
private async Task IncrementCount()
{
    CurrentCount += IncrementAmount;
    await CurrentCountChanged.InvokeAsync(CurrentCount);
}
```

By merely using that naming convention, Blazor knows that `CurrentCountChanged` is the event that will get triggered when a change to `CurrentCount` occurs. This is used by the parent component when it binds to `CurrentCount`. When `CurrentCountChanged` is invoked, Blazor sends the new value back to the parent and updates the variable used in the `@bind-CurrentCount` expression.

`EventCallback` cannot be `null`, so there is no reason to do a null check (more on that in the next section).

We also need to change how we listen for changes:

```
<CounterWithParameterAndEvent IncrementAmount="@incrementamount"
    @bind-CurrentCount="currentcount"/>
```

Here, we need to add `@bind-` before the `CurrentCount` binding.

You can also use the following syntax to set the name of the event:

```
<CounterWithParameterAndEvent IncrementAmount="@incrementamount"
    @bind-CurrentCount="currentcount"
    @bind-CurrentCount:event="CurrentCountChanged"/>
```

By using `:event`, we can tell Blazor exactly which event we want to use; in this case, the `CurrentCountChanged` event. The more common approach here is to use the convention and not specify our own event names.

In the next chapter, *Chapter 8*, *Building Forms with Validation*, we will continue to look at binding with input/form components.

We can, of course, also create events using `EventCallback`.

Diving into Actions and EventCallback

To communicate changes, we can use **EventCallback**, as shown in the *Two-way binding* section. Blazor provides two versions of this type: `EventCallback` and `EventCallback<T>`. The non-generic `EventCallback` is used when the event only signals that something happened, while `EventCallback<T>` is used when we want to pass a value back to the parent component. In most cases, you will use `EventCallback<T>` since it allows you to send updated data along with the event.

`EventCallback<T>` differs a bit from what we might be used to in .NET. `EventCallback<T>` is a class that is specially made for Blazor to have the event callback exposed as a parameter for a component.

In .NET, in general, you can add multiple listeners to an event (multi-cast), but with `EventCallback<T>`, you will only be able to add one listener (single-cast).

It is worth mentioning that you can use events the way you are used to from .NET in Blazor. However, you probably want to use `EventCallback<T>` because there are many upsides to using `EventCallback/EventCallback<T>` over traditional .NET events, as follows:

- .NET events are classes, and `EventCallback` is a struct. This means that in Blazor, we don't have to perform a null check before calling `EventCallback` because a struct cannot be null.
- `EventCallback` is asynchronous and can be awaited. When `EventCallback` has been called, Blazor will automatically execute `StateHasChanged` on the consuming component to ensure the component updates (if it needs to be updated).

Blazor components are typically designed to expose a single callback to their parent. While you can use `Action<T>`, it does not support the same patterns as .NET events (such as multiple subscribers). For this reason, `EventCallback<T>` is the recommended approach.

Some events include event arguments that provide additional information, such as mouse position or keyboard input. You can access these arguments if you need them, but you don't have to. If your logic doesn't depend on that extra information, you can ignore them and use a simpler handler. You can add them by specifying them in a method, or you can use a lambda expression like this:

```
<button @onclick="@((e)=>message=$"x:{e.ClientX} y:{e.ClientY}")">
    Click me</button>
```

When the button is clicked, it will set a variable called `message` to a string containing the mouse coordinates. The lambda has one parameter, `e`, of the `MouseEventArgs` type. However, you don't have to specify the type, and the compiler understands what type the parameter is.

Now that we have added actions and used `EventCallback` to communicate changes, we will see how we can execute `RenderFragment` in the next section.

Using RenderFragment

To make our components even more reusable, we can supply them with a piece of Razor syntax. In Blazor, you can specify a **RenderFragment**, which is a fragment of Razor syntax that you can execute and show.

There are two types of render elements: `RenderFragment` and `RenderFragment<T>`. `RenderFragment` is simply a Razor fragment without any input parameters, and `RenderFragment<T>` has an input parameter that you can use inside the Razor fragment code by using the `context` keyword. We won't go into depth about how to use this now, but later in this chapter, we will talk about a component (**Virtualize**) that uses `RenderFragment<T>`, and in the next chapter, *Chapter 8*, *Building Forms with Validation*, we will implement a component using `RenderFragment<T>`.

We can set `RenderFragment` as the default content inside the component tags and give it a default value. We will explore this next and build a component using these features.

We can also return a `RenderFragment` from a method and render it directly. Take this example:

```
@page "/RenderFragmentTest"

@for (int i = 0; i < 10; i++)
{
    @Render(i)
}

@code
{
    private RenderFragment Render(int number)
    {
         return @<p>This is a render fragment @number</p>;
    }
}
```

Now we have a component with a method that returns a `RenderFragment`.

The method could be static if we needed to use it in other components as well. When using a loop like this, performance often improves and memory usage is typically lower than when rendering a component for each item because render fragments avoid the overhead of component instances.

Note

Grid Component

If you want to dig deeper into render fragments, please check out `Blazm.Components`, which has a grid component that heavily uses `RenderFragment<T>`. It is not currently updated since the built-in QuickGrid has most of the functions now, but it is, in my opinion, still a good place to learn from.

You can find it on GitHub here: `https://github.com/EngstromJimmy/Blazm.Components`.

ChildContent

So far, we've seen how to return and render fragments. Next, let's look at how `RenderFragment` is commonly used in components.

By naming the render fragment `ChildContent`, Blazor will automatically use whatever is placed between the component tags as content. This only works if the component has a single render fragment. If you have more than one, you need to explicitly specify which fragment you are setting.

We can also give the `RenderFragment` a default value. This will be used if no content is provided between the component tags. We can set this directly in code by using the `@` syntax:

```
@<b>This is a default value</b>;
```

We will build a component using this pattern in the next section.

Building an alert component

To better understand how to use render fragments, let's build an alert component that will use render fragments. The built-in templates use Bootstrap, so we will do the same for this component. Bootstrap has many components that are easy to import to Blazor. When working on big projects with multiple developers, building components is an easy way to ensure everyone on the team writes code the same way.

Let's build a simple alert component based on Bootstrap:

1. Create a folder by right-clicking on **BlazorWebApp.Client** | **Add** | **New Folder**, and name the folder `ReusableComponents`.
2. Create a new Razor component and name it `Alert.razor`.

3. Replace the content with the following code in the `Alert.razor` file:

```
<div class="alert alert-primary" role="alert">
    A simple primary alert—check it out!
</div>
```

The code is taken from Bootstrap's web page (`http://getbootstrap.com`) and it shows an alert that looks like this:

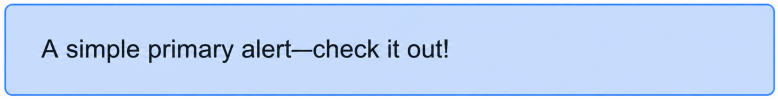


Figure 7.1: The default look of a Bootstrap alert component

There are two ways in which we could customize this `alert` component. The first option is to add a `string` parameter for the message. However, since this is a section on render fragments, we will explore the second option – yes, you guessed it, *render fragments*.

4. Add a code section with a `RenderFragment` property called `ChildContent` and replace the alert text with the new property:

```
<div class="alert alert-primary" role="alert">
    @ChildContent
</div>
@code{
    [Parameter]
    public RenderFragment ChildContent { get; set; } =
        @<b>This is a default value</b>;
}
```

Now we have a `RenderFragment` type and have set a default value, displaying the fragment between the `div` tags. We also want to add an `enum` for the different ways you can style the alert box.

5. In the `code` section, add an `enum` containing the different styles available:

```
public enum AlertStyle
{
    Primary,
    Secondary,
```

```
    Success,
    Danger,
    Warning,
    Info,
    Light,
    Dark
}
```

6. Add a parameter/property for the `enum` style:

```
[Parameter]
public AlertStyle Style { get; set; }
```

7. The final step is to update the `class` attribute for `div`. Change the `class` attribute to look like this:

```
<div class="@($"alert alert-{Style.ToString().ToLower()}")"
    role="alert">
```

8. Next, in the `BlazorWebApp.Client` project, in the `Pages` folder, create a new Razor component and name it `AlertTest.razor`. Replace the code with the following snippet:

```
@page "/alerttest"
@using BlazorWebApp.Client.ReusableComponents
<Alert Style="Alert.AlertStyle.Danger">
    This is a test
</Alert>
<Alert Style="Alert.AlertStyle.Success">
    <ChildContent>
        This is another test
    </ChildContent>
</Alert>
<Alert Style="Alert.AlertStyle.Success" />
```

The page shows three alert components:

- The first alert has the `Danger` style. We are not specifying what property to set for the `This is a test` text, but by convention, it uses the property called `ChildContent`.

- In the second alert, we have specified the `ChildContent` property. If you use more render fragments in your component, you must set them like this with full names.
- In the last alert, we didn't specify anything, which will give the property the default render fragment we specified in the component.

9. Run the project and navigate to `/AlertTest` to see the test page:

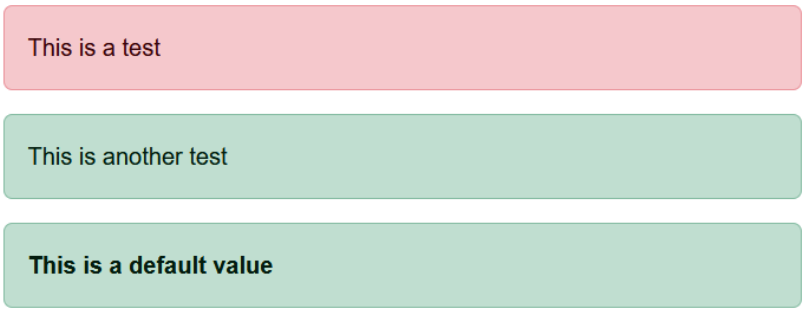


Figure 7.2: Screenshot of the test page

We have finished our first reusable component!

Designing components for real-world use

When I create reusable components, I tend to adapt them to the workplace. So I might not expose every single style that Bootstrap offers, but rather the ones I expect us to use. I also name them according to how they are supposed to be used. A button might be called **Primary**, but it might instead be called **Save**, which describes the use case.

Creating reusable components is how I prefer to build my Blazor sites, since I don't have to write the same code twice. This becomes even more apparent when working in a larger team. It makes it easier for all developers to produce the same code and achieve the same result, enabling higher code quality and fewer tests.

At work, when we upgraded to the latest Bootstrap version, a few CSS classes were deprecated and replaced by others. Thankfully, because we followed this approach of making reusable components, we only had to make changes in a handful of places. There were a couple of places where we still had an old codebase (not using components), and it became very apparent that creating components was worth the effort.

If you find yourself writing the same code twice, you might want to add that into a component. At my old job, we started using `Radzen`, an open-source component library (among other things). At my current job, we use `MudBlazor`. We use `Progress Telerik` on our stream. Using third-party components can speed up development, but these components are often built for

many different users. They can do a lot of things. This means that every single developer on our team now has access to all that power. With great power comes great responsibility.

In one of my presentations, I used that quote with a picture of Batman and text that said "Superman." I didn't get a single reaction. I have never failed with a joke like that. But joking aside, this means that all developers need to keep in mind how to use the components. Otherwise, the UI might look different depending on which developer uses the components. I put a lot of time into designing reusable components that help the team to be productive, hiding the parameters we don't use, and giving the components reasonable default values. So, even if you use third-party components, try to figure out what you are using and perhaps create an abstraction on top of the third-party components. If you don't know, the quote is from *Spider-Man*, or Uncle Ben, to be precise. But it did remind me of one of my favorite puns. Do you know why *Spider-Man* always has such witty comebacks? Because with great power comes great response-ability. I will show myself out.

Blazor has a bunch of built-in components. In the next section, we will dig deeper into what they are and how to use them.

Exploring the new built-in components

When Blazor first came out, there were a couple of things that were hard to do, and, in some cases, we needed to involve JavaScript to solve the challenge. In this section, we will look at some of the new components we got in .NET 5, all the way through .NET 10.

We will take a look at the following components or functions:

- Setting the focus of the UI
- Influencing the HTML head
- Component virtualization
- Error boundaries
- Sections

Let's get started.

Setting the focus of the UI

One of my first Blazor blog posts was about how to set the focus on a UI element, but now this is built into the framework. The previous solution involved JavaScript calls to change the focus on a UI element, but by using `ElementReference`, you can now set the focus on the element.

Let's build a component to test the behavior of this feature:

1. In the `BlazorWebApp.Client` project, in the `Pages` folder, add a new Razor component and name it `SetFocus.razor`.

2. Open `SetFocus.razor` and add a page directive:

```
@page "/setfocus"
```

3. Add an element reference:

```
@code {
    ElementReference textInput;
}
```

`ElementReference` is precisely what it sounds like: a reference to an element. In this case, it is an input textbox.

4. Add the textbox and a button:

```
<input @ref="textInput" />
<button @onclick="() => textInput.FocusAsync()">Set focus</button>
```

Using `@ref`, we specify a reference to any type of component or tag that we can use to access the input box. The `button onclick` method will execute the `FocusAsync()` method of the textbox and set the focus on it.

5. Press *F5* to run the project and then navigate to `/setfocus`.
6. Press the **Set focus** button and notice how the textbox gets its focus.

It could seem like a silly example since this only sets the focus, but it is a handy feature, and the `autofocus` HTML attribute won't work for Blazor. It would make more sense to call `FocusAsync` in the `OnAfterRender` method to get the focus change when we load the page, but that wouldn't make it as cool a demo.

In my blog post, I had another approach. My goal was to set the focus of an element without having to use code. In the upcoming chapter, *Chapter 8*, *Building Forms with Validation*, we will implement the `autofocus` feature from my blog post but use the new .NET features instead.

The release of .NET 5 solved many things we previously had to write with JavaScript; setting the focus is one example. In later versions, we could influence the HTML head.

Influencing the HTML head

Sometimes, we want to set our page's title or change the social network meta tags. The `head` tag is located in the `App` component, and that part of the page isn't reloaded or rerendered (only the components within the routes component are rerendered). In previous versions of Blazor, you had to write code for that yourself using JavaScript.

But .NET has a component called `HeadOutlet` that can solve that.

Using many of the techniques we have discussed so far, we will utilize the `HeadOutlet` component to create a page to view one of our blog posts:

1. In the `BlazorWebApp.Client` project, open `Home.razor`.
2. Change the `foreach` loop to look like this:

   ```
   <li><a href="/Post/@p.Id">@p.Title</a></li>
   ```

   We added a link to the title to look at one blog post. Notice how we can use the `@` symbol inside the `href` attribute to get the ID of the post.
3. In the `Pages` folder, add a `Razor` component, and name it `Post.razor`.
4. In the `code` section, add a parameter that will hold the ID of the post that comes from the URL:

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

5. Add a `page` directive to get the set, the URL, and the ID:

   ```
   @page "/post/{BlogPostId}"
   ```

   The `page` directive will set the URL for our blog post to `/post/`, followed by the ID of the post. We don't have to add a `using` statement to all our components. Instead, open `_Imports.razor` and add the following namespaces:

   ```
   @using BlazorWebApp.Client.Interfaces
   @using BlazorWebApp.Client.Models
   ```

   This will ensure that all our components have these namespaces by default.
6. Open `Post.razor` again and, just beneath the `page` directive, inject the API (the namespace is now supplied from `_Imports.razor`):

   ```
   @inject IBlogRepository _repository
   @inject NavigationManager _navman
   ```

Our API will now be injected into the component, and we can retrieve our blog post. We also have access to a navigation manager.

7. In the code section, add a property for our blog post:

```
public BlogPost? BlogPost { get; set; }
```

This will contain the blog post that we want to show on the page.

8. To load the blog post, add the following code:

```
protected override async Task OnParametersSetAsync()
{
    BlogPost = await _repository.GetBlogPostAsync(BlogPostId);
    await base.OnParametersSetAsync();
}
```

In this case, we are using the `OnParametersSetAsync()` method. This ensures that when the parameter is set or changed, we fetch the corresponding data from the database and update the content.

9. We must also show the post and add the necessary `meta` tags. To do that, add the following code just above the code section:

```
@if (BlogPost != null)
{
    <PageTitle>@BlogPost.Title</PageTitle>
    <HeadContent>
    <meta property="og:title"
      content="@BlogPost.Title" />
    <meta property="og:description" content="@(new
      string(BlogPost.Text.Take(100).ToArray()))" />
    <meta property="og:image" content=
      "@($"{_navman.BaseUri}/pathtoanimage.png")" />
    <meta property="og:url" content="@_navman.Uri" />
    <meta name="twitter:card" content="@(new
      string(BlogPost.Text.Take(100).ToArray()))" />
    </HeadContent>

    <h2>@BlogPost.Title</h2>
    @((MarkupString)BlogPost.Text)
```

```
}
```

When the page is first loaded, the `BlogPost` parameter can be null, so we first need to check whether we should show the content at all.

By adding the `Title` component, Blazor will set the title of our site to the title of our blog post in this instance.

According to the information I gathered on **search engine optimization** (**SEO**), the meta tags we have added are the bare minimum to use with Facebook and X (formerly known as Twitter). We don't have an image for each blog post, but we can have one that is site-wide (for all blog posts) if we would like. Just change `Pathtoanimage.png` to the name of the image and put the image in the `wwwroot` folder.

Then show an `H2` tag with the title and the text beneath that. You might remember `MarkupString` from *Chapter 6*, *Understanding Basic Blazor Components*. This will output the string from our blog post without changing the HTML (not escaping the HTML).

10. Run the project by pressing *F5* and navigate to a blog post to see the title change:

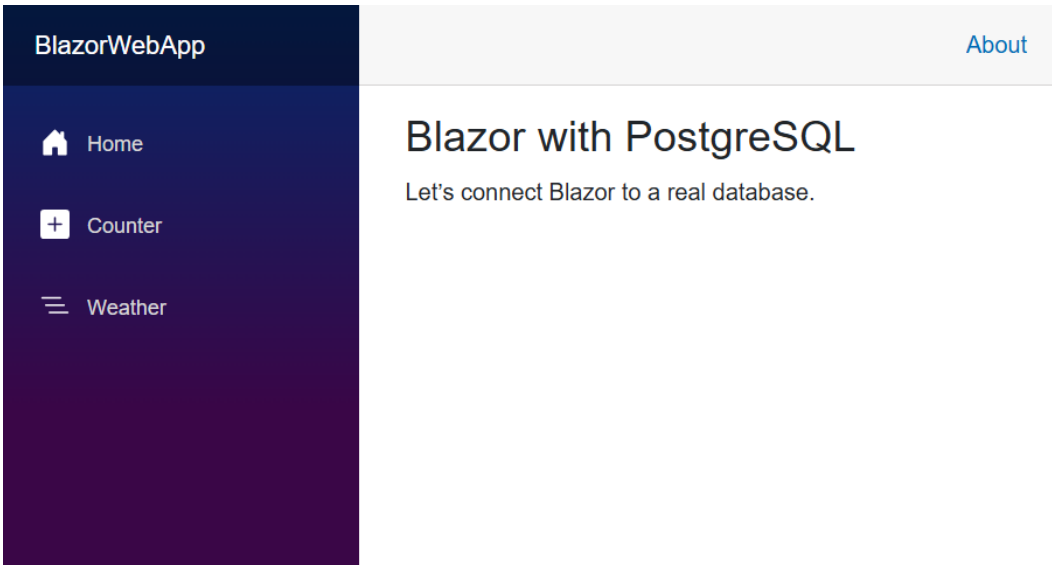


Figure 7.3: Blog post screenshot

Our blog is starting to take form. We have a list of blog posts and can view a single post; we are far from done, but we're well on our way.

Component virtualization

Virtualize is a component in Blazor that ensures it only renders the components or rows that can fit on the screen. If you have a large list of items, rendering all of them will have a significant impact on memory.

Many third-party component vendors offer grid components with the same virtualization function. The `Virtualize` component was, in my opinion, the most exciting thing in the .NET 5 release.

The `Virtualize` component will calculate how many items can fit on the screen (based on the size of the window and the height of an item). Blazor will add a `div` tag before and after the content list when scrolling the page, ensuring the scrollbar shows the correct position and scale (even though no items are rendered).

The `Virtualize` component works just like a `foreach` loop.

The following is the code we currently have in our `Home.razor` file:

```
<ul>
    @foreach (var p in posts)
    {
        <li><a href="/Post/@p.Id">@p.Title</a></li>
    }
</ul>
```

Right now, it shows all our blog posts in our database as a long list. Granted, we only have a few right now, but we might have many posts one day.

We can change the code (don't change the code just yet though) to use the new `Virtualize` component by updating it to the following:

```
<Virtualize Items="posts" Context="p">
    <li><a href="/Post/@p.Id">@p.Title</a></li>
</Virtualize>
```

Instead of the `foreach` loop, we use the `Virtualize` component and add a render fragment that shows how each item should be rendered. The `Virtualize` component uses `RenderFragment<T>`, which, by default, will send in an item of type `T` to the render fragment. In the case of the `Virtualize` component, the object will be one blog post (since items are `List<T>` of blog posts).

We access each post with the variable named `context`. However, we can use the `Context` property on the `Virtualize` component to specify another name, so instead of `context`, we are now using `p`. This way, we can keep the syntax just as we did with the `ul`.

Note that in this case, we're using `p`, which might not be as descriptive. In some cases, we want a more descriptive variable name than "context."

The `Virtualize` component is even more powerful than this, as we will see in the next feature that we implement, where we only load the post we need to fill the view:

1. In the `BlazorWebApp.Client` project, open `Home.razor`.
2. Delete the `OnInitializedAsync` method and `protected List<BlogPost> posts = new List<BlogPost>()`, as we don't need them anymore.
3. Change the loading of the post to `Virtualize`:

```
<ul>
    <Virtualize ItemsProvider="LoadPosts" Context="p">
        <li><a href="/Post/@p.Id">@p.Title</a></li>
    </Virtualize>
</ul>
```

   In this case, we are using the `ItemsProvider` delegate, which takes care of getting posts from our API.

   We pass in a method called `LoadPosts`, which we also need to add to the file.

4. Now, let's add the `LoadPosts` method by adding the following code:

```
public int totalBlogposts { get; set; }
private async ValueTask<ItemsProviderResult<BlogPost>>
    LoadPosts(ItemsProviderRequest request)
{
    if (totalBlogposts == 0)
    {
        totalBlogposts = await _repository.GetBlogPostCountAsync();
    }
    var numblogposts = Math.Min(request.Count,
        totalBlogposts - request.StartIndex);
    var blogposts= await _repository.GetBlogPostsAsync(
        numblogposts,request.StartIndex);
    return new ItemsProviderResult<BlogPost>(
```

```
            blogposts, totalBlogposts);
    }
```

We added a `totalBlogposts` property to store the number of posts we currently have in our database. The `LoadPost` method returns a `ValueTask` with `ItemsProviderResult<Blogpost>`. The method takes an `ItemsProviderRequest` parameter, which specifies the number of posts the Virtualize component wants and how many to skip.

If we don't know how many total posts we have, we need to retrieve that information from our API by calling the `GetBlogPostCountAsync` method. Then, we need to figure out how many posts we should get; either we get as many posts as we need or we get all the remaining posts (whichever value is the smallest).

Then, we call our repository to get the actual posts by calling `GetBlogPostsAsync` and return `ItemsProviderResult`.

All of this means that we now have an infinite scroll for all our blog posts. However, we only load the ones currently visible on the screen and ensure that we do not render more items in the HTML than are needed to fill the viewport.

Error boundaries

In .NET 6, we gained a very handy component to handle errors called **ErrorBoundary**.

We can surround the component with an `ErrorBoundary` component; if an error occurs, it will show an error message instead of the whole page failing:

```
<ErrorBoundary>
   <ComponentWithError />
</ErrorBoundary>
```

This component takes two render fragments. By specifying it as in the previous example, we only set the `ChildContent` render fragment. This is the default.

We can also supply a custom error message like this:

```
<ErrorBoundary>
    <ChildContent>
        <ComponentWithError />
    </ChildContent>
    <ErrorContent>
        <h1 style="color: red;">Oops… something broke</h1>
```

```
    </ErrorContent>
</ErrorBoundary>
```

In this sample, we specify `ChildContent`, which makes it possible for us to specify more than one property, as is the case with `ErrorContent`. This is a great component to extend and create your own functionality. You can get access to the exception by using the `context` parameter (as we did with `virtualize`):

```
<ErrorBoundary Context="ex">
     <ChildContent>
        <p>@(1/zero)</p>
    </ChildContent>
    <ErrorContent>
       An error occurred
       @ex.Message
    </ErrorContent>
</ErrorBoundary>
@code {
    int zero = 0;
}
```

This is a great way to handle errors in the UI.

Sections

Starting from .NET 8, .NET gives us the ability to add sections. You might remember a similar feature of `WebForms`.

We can use the `SectionOutlet` component to define an area in a layout component where we want to insert content. Then, inside our components, we can add a `SectionContent` component to add the content we want to appear in the outlet.

If we have more than one `SectionContent` referencing the `SectionOutlet` component, it will render the latest `SectionContent`. We can refer to a `SectionOutlet` by using a section name or a section ID. A section name is simply a string defined by us that we can use. The ID is an object, so we can get a nicer syntax to keep track of our sections.

We can add a section to the layout file and add content to that section from our components. It's a layout thing. Let's say we want to add contextual menus. For example, in this way, we could change a menu that is in a completely different component.

Let's look at some code (you should not follow along for this part).

First, we need to add the following namespace, preferably in the `_Imports.razor` file since this is one of the built-in components:

```
Microsoft.AspNetCore.Components.Sections;
```

In a layout component, we add an outlet like this:

```
<SectionOutlet SectionName="top-header"/>
```

Then, in our component, we can add a `SectionContent` component like this:

```
<SectionContent SectionName="top-header">
    <b>Test</b>
</SectionContent>
```

If we instead want to use the section ID, we can do it like this: In the layout file, let's assume it is called `MainLayout`:

```
<SectionOutlet SectionId="MainLayout.TopHeader"/>
```

Make the following changes in the code section of `MainLayout`:

```
@code
{
    public static SectionOutlet TopHeader = new();
}
```

Then, inside the component, we change it to this:

```
<SectionContent SectionId="Layout.MainLayout.TopHeader">
    <b>Using SectionId</b>
</SectionContent>
```

This is a great way to change the layout files. By doing this, we can create more advanced layouts that work with every page/component. We can move more of the layout to the layout file instead of putting it in each component. I love this feature. This will clean up so much code.

Summary

In this chapter, we looked at more advanced scenarios for building components. Building components is what Blazor is all about, making it easy to implement changes along the way because there is only one point where the change must be made.

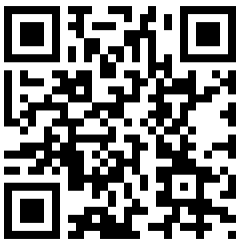
We also implemented our first reusable component, which will help maintain the same standard across the team and reduce duplicate code.

Plus, we used some Blazor features to load and display data.

In the next chapter, we will look at forms and validation to start building the administration part of our blog.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

8

Building Forms with Validation

In this chapter, we will learn how to create and validate forms, which is an excellent opportunity to build our admin interface to manage our blog posts, and also take a look at the new Enhanced Form Navigation. We will also build multiple reusable components and learn about new Blazor features. This chapter will be super fun, and we will use many of the things we have learned so far.

As such, we will cover the following topics:

- Exploring form elements
- Adding validation
- Custom validation class attributes
- Looking at bindings
- Building an admin interface

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter07` folder as a starting point.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter08`.

Exploring form elements

There are many HTML form elements, and we can use them all in Blazor. In the end, what Blazor outputs is HTML.

Blazor does have components that enhance functionality, so we can use them instead of HTML elements because they provide additional functionality.

Blazor offers the following input components:

- `EditForm`
- `InputBase<>`
- `InputCheckbox`
- `InputDate<TValue>`
- `InputNumber<TValue>`
- `InputSelect<TValue>`
- `InputText`
- `InputTextArea`
- `InputRadio`
- `InputRadioGroup`
- `InputFile`
- `ValidationMessage`
- `ValidationSummary`

Let's go through them all in the next sections.

EditForm

`EditForm` renders as a `form` tag, but it has a lot more functionality.

Unlike traditional form tags, where you need to create an action or method, Blazor's `EditForm` will create an `EditContext` instance as a cascading value so that all the components you put inside `EditForm` will access the same `EditContext`. `EditContext` tracks the metadata regarding the editing process, such as which fields have been edited, and keeps track of any validation messages.

You need to assign either a model (a class you wish to edit) or an `EditContext` instance to the `EditForm`. For most use cases, assigning a model is the way to go, but for more advanced scenarios, you might want to be able to trigger `EditContext.Validate()`, for example, to validate all the controls connected to `EditContext`. This is very rarely done, but it is good to know it is possible.

`EditForm` has the following events that you can use to handle form submissions:

- `OnValidSubmit` gets triggered when the data in the form validates correctly (we will come back to validation in just a bit)
- `OnInvalidSubmit` gets triggered if the form does not validate correctly
- `OnSubmit` gets triggered when the form is submitted, regardless of whether the form validates correctly or not. Use `OnSubmit` if you want to control the validation yourself.

Let's take a look at an example. Consider a class that holds a person. The class has a name and an age for that person, and looks like this:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

`EditForm` for this class would look like this (without any other elements for now):

```
<EditForm Model="personmodel" OnValidSubmit="@validSubmit">
    ...
    <button type="submit">Submit</button>
</EditForm>
@code {
    Person personmodel = new Person();
    private Task validSubmit()
    {
        //Do processing here
        return Task.CompletedTask;
    }
}
```

`EditForm` specifies a model (in this case, `personmodel`), and we are listening to the `OnValidSubmit` event.

The `Submit` button is a regular HTML button that is not a specific Blazor component.

This is what is needed to get a form up and running. We will use it later in this chapter.

InputBase<>

All the Blazor input classes derive from the `InputBase` class. It provides several features we can use for all of the `input` components, and we will go through the most important ones.

`InputBase` handles `AdditionalAttributes`, which means that if we add any other attributes to the tag, they will automatically get transferred to the output. This means that the components derived from this class can leverage any HTML attributes, since they will be part of the output.

`InputBase` has a `Value` property, which we can bind to, and a `ValueChanged` event callback.

We can also change `DisplayName` so that the automated validation messages reflect the correct name rather than the property's name, which is the default behavior.

Note that not all controls support the `DisplayName` property. Some properties are only used inside the component, and we will return to those in a bit.

InputCheckbox

The `InputCheckbox` component will render as `<input type="checkbox">`.

InputDate<TValue>

The `InputDate` component will render as `<input type="date">`. We can use `DateTime`, `DateOnly`, `TimeOnly`, and `DateTimeOffset` as values for the `InputDate` component.

There is no built-in way to format the date; it will use the web browser's current setting. This behavior is by design and is part of the HTML5 spec.

InputNumber<TValue>

The `InputNumber` component will render as `<input type="number">`. We can use `Int32`, `Int64`, `Single`, `Double`, and `Decimal` as values for the `InputNumber` component.

InputSelect<TValue>

The `InputSelect` component will render as `<select>`. We will create `InputSelect` later in this chapter, so we won't go into further detail here.

InputText

The `InputText` component will render as `<input type="text">`.

InputTextArea

The `InputTextArea` component will render as `<textarea>`. In this chapter, we will build our own version of this control.

InputRadio

The `InputRadio` component will render as `<input type="radio">`. Use this for individual options.

InputRadioGroup

The `InputRadioGroup` component is not an element itself but rather groups other `InputRadioInputs`. Use this to group options. We can add multiple InputRadio components inside of the `InputRadioGroup`.

InputFile

The `InputFile` component will render as `<Input type="file">`. This component will make it easier to get the file data. It will supply us with a stream for each file's content.

> **Note**
>
> We can dive into `InputFile` further by checking out the documentation here: `https://learn.microsoft.com/en-us/aspnet/core/blazor/file-uploads?view=aspnetcore-10.0`.

As we can see, there is a Blazor component for almost all HTML form controls, with added functionality such as validation, which we will see in the next section.

Adding validation

We have already touched on validation; the input components and `EditForm` include built-in functionality to handle it.

One way to add validation to our form is to use `DataAnnotations`. Using data annotations, we don't need to write any custom logic to ensure the data in the form is correct; instead, we can add attributes to the data model and let `DataAnnotationsValidator` handle the rest.

There are a bunch of `DataAnnotations` instances in .NET already that we can use; we can also build our own annotations.

Some of the built-in data annotations are as follows:

- `Required`: Makes the field required.
- `Email`: Checks that the entered value is an email address.
- `MinLength`: Checks that the number of characters is not fewer than the value specified.

- `MaxLength`: Checks that the number of characters is not exceeded.
- `Range`: Checks that the value is within a specific range.

There are many more annotations that can help us validate our data.

To test them out, let's add data annotations to our data classes:

1. In the `BlazorWebApp.Client` project, open `Models/BlogPost.cs`.
2. At the top of the file, add a `using` statement for `System.ComponentModel.DataAnnotations`:

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

3. Add the `Required` and `MinLength` attributes to the `Title` property:

```
[Required]
[MinLength(5)]
public string Title { get; set; } = string.Empty;
```

   The `Required` attribute will ensure we can't leave the title empty, and `MinLength` will make sure it has at least `5` characters.

   We assign a default value of `string.Empty` to avoid null values. However, the `[Required]` attribute will still trigger validation if the field is empty, since it validates both null and empty strings.

4. Add the `Required` attribute to the `Text` property:

```
[Required]
public string Text { get; set; } = string.Empty;
```

   The `Required` attribute will ensure the `Text` property cannot be empty, which makes sense – why would we create an empty blog post?

5. Open `Models/Category.cs`, and at the top of the file, add a `using` statement:

```
using System.ComponentModel.DataAnnotations
```

6. Then add the `Required` attribute to the `Name` property:

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

   The `Required` attribute will make sure we can't leave `Name` empty.

7. Open `Models/Tag.cs`, and at the top of the file, add a `using` statement:

```
using System.ComponentModel.DataAnnotations
```

8. Then add the `Required` attribute to the `Name` property:

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

   The `Required` attribute will make sure we can't leave the name empty.

9. Open `Models/Comment.cs`, and at the top of the file, add a `using` statement:

```
using System.ComponentModel.DataAnnotations
```

10. Then add the `Required` attribute to the `Name` and `Text` property:

```
[Required]
public string Text { get; set; } = string.Empty;
[Required]
public string Name { get; set; } = string.Empty;
```

Great, now our data models have validation built into them. We need to give our users feedback on what went wrong with the validation. We can do that by using the `ValidationMessage` or `ValidationSummary` components.

ValidationMessage

The `ValidationMessage` component can show individual error messages for a specific property. We want to use this component to show validation errors under a form element.

To add a `ValidationMessage` component, we have to specify the `For` property with the name of the property for which we want to show the validation errors:

```
<ValidationMessage For="@(() => personmodel.Name)"/>
```

ValidationSummary

The `ValidationSummary` component will show all the validation errors as a list for the entire `EditContext`.

```
<ValidationSummary/>
```

We don't need to supply any model or property to the `ValidationSummary` component since it gets access to the `EditContext` using the cascading value.

I prefer to show the error close to the problem so the user can see where the issue is. However, we can also show validation errors as a list using `ValidationSummary`.

To ensure our input controls match the Bootstrap theme (or any theme we use), we can create a **custom validation class**.

Custom validation class attributes

By simply using the `editform`, input components, and `DataAnnotationsValidator`, the framework will automatically add classes to the components when they are and aren't valid.

By default, these classes are `.valid` and `.invalid`. In .NET 5, we were given a way to customize these class names ourselves.

When using Bootstrap, the default class names are `.is-valid` and `.is-invalid`, and the list of class names must also include `.form-control` to get the proper styles.

The next component we build will help us get the proper Bootstrap styling on all our form components. We will create our own `FieldCssClassProvider` to customize what classes Blazor will use:

1. In the `BlazorWebApp.Client` project, inside the `ReusableComponents` folder, add a new class called `BootstrapFieldCssClassProvider.cs`.
2. Open the new class and add the following code:

```
using Microsoft.AspNetCore.Components.Forms;
namespace BlazorWebApp.Client.ReusableComponents ;
public class BootstrapFieldCssClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext
        editContext, in FieldIdentifier fieldIdentifier)
   {
```

```
            var isValid = !editContext.GetValidationMessages(
                fieldIdentifier).Any();
            var isModified = editContext.IsModified(fieldIdentifier);
            return (isModified, isValid) switch
            {
                (true, true) => "form-control modified is-valid",
                (true, false) => "form-control modified is-invalid",
                (false, true) => "form-control",
                (false, false) => "form-control"
            };
        }
    }
```

`BootstrapFieldCssClassProvider` needs an `EditContext` instance to work.

The code will check whether the form (or `EditContext`, to be specific) is valid and whether or not it has been modified. Based on that, it returns the correct CSS classes.

It also returns "form-control" for all elements; that way, we don't have to add it to every element in the form. We could validate an untouched form as valid or invalid, but we don't want it to show that the form is OK just because it hasn't been changed yet.

To deal with this, we need to get the `EditContext` instance from our `EditForm` and then set `FieldCssClassProvider` on `EditContext` as follows:

```
CurrentEditContext.SetFieldCssClassProvider(provider);
```

Rather than setting the `FieldCssClassProvider` directly in every form, we can wrap this setup in a small reusable component.

Earlier in this chapter, we saw that `EditForm` exposes its `EditContext` as a cascading value. We can use that to build a component that lives inside an `EditForm`, gets access to the current `EditContext`, and applies our custom CSS class provider automatically:

3. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, add a new class and name it `CustomCssClassProvider.cs`.
4. Open the new file and replace the content with the following code:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;
namespace BlazorWebApp.Client.ReusableComponents;
public class CustomCssClassProvider<ProviderType> : ComponentBase
```

```
    where ProviderType : FieldCssClassProvider, new()
{
    [CascadingParameter]
    EditContext? CurrentEditContext { get; set; }
    public ProviderType Provider { get; set; } = new();
    protected override void OnInitialized()
    {
        if (CurrentEditContext == null)
        {
            throw new InvalidOperationException($"{nameof
                (CustomCssClassProvider <ProviderType>)} requires a
                    cascading parameter of type {nameof(
                        EditContext)}. For example, you can use
                            {nameof(CustomCssClassProvider
                                <ProviderType>)} inside an
                                    EditForm.");
        }
        CurrentEditContext.SetFieldCssClassProvider
         (Provider);
    }
}
```

This generic component takes a `type` value, in this case the type of `Provider`.

We specified that `type` must inherit from `FieldCssClassProvider` and must have a constructor without parameters.

The component inherits from `ComponentBase`, which makes it possible to place the component inside a Blazor component.

In this case, we are writing our component with C# only, but it does not render anything.

We have a `Cascading` parameter that will be populated from `EditForm`. We throw an exception if `EditContext` is missing for some reason (for example, if we place the component outside `EditForm`).

Finally, we set `FieldCssClassProvider` on `EditContext`.

To use the component, we have to add the following code inside an `EditForm` (right now, we don't have one, but don't worry, we will create an `EditForm` soon):

```
<CustomCssClassProvider ProviderType="BootstrapFieldCssClassProvider"/>
```

We provide our `CustomCssClassProvider` component with the right `ProviderType`: `BootstrapFieldCssClassProvider`.

This is one way of implementing components to help us encapsulate functionality, but we could have written the code this way:

```
<EditForm Model="personmodel" @ref="CurrentEditForm">
…
</EditForm>
@code {
    public EditForm CurrentEditForm { get; set; }
    protected override Task OnInitializedAsync()
    {
        CurrentEditForm.EditContext.SetFieldCssClassProvider(new
            BootstrapFieldCssClassProvider())
        return base.OnInitializedAsync();
    }
}
```

But with the new `CustomCssClassProvider` component, we can write the same thing like this:

```
<EditForm Model="personmodel">
<CustomCssClassProvider ProviderType="BootstrapFieldCssClassProvider" />
</EditForm>
```

If we are using EditContext, we can always create a component like this, since it is a `cascading` parameter.

Now, we have a component that will make our form controls look like Bootstrap controls, and instead of adding specific code to every component, we can now add the `CustomCssClassProvider` component. Next, it's time to put that into practice and create a couple of forms by building our admin interface.

Looking at bindings

In this chapter, we are using **bindings** to bind data to our form controls. We briefly discussed bindings in *Chapter 7*, *Creating Advanced Blazor Components*, but it's time to dig deeper into bindings.

Binding to HTML elements

With HTML elements, we can use `@bind` to bind variables to the element. If we are binding to a textbox, we would do it like this:

```
<input type="text" @bind="Variable"/>
```

`@bind` and `@bind-value` both work and do the same thing. Note the lower `v` in value. The input element is an HTML element that will render as a normal HTML element with no extra features (except binding). Compare this to `InputText`, which works similarly but also provides additional features, such as validation and styles.

By default, the variable's value changes when we leave the textbox. But we can change that behavior by adding a `@bind:event` attribute like this:

```
<input type="text" @bind="Variable" @bind:event="oninput"/>
```

We can even take full control over what is happening by using the `@bind:get` and `@bind:set` attributes like this:

```
<input type="text" @bind:get="SomeText" @bind:set="SetAsync" />
```

These are doing the same thing as `@bind`, so we can't use them together with `@bind`. The `@bind:set` attribute has another nice feature: we can run asynchronous methods when we set a value.

There is also a way for us to run a method after the value is set by using `@bind:after` like this:

```
<input type="text" @bind="SomeText" @bind:after="AfterAsync" />
```

This gives us great flexibility when it comes to binding to HTML elements.

On top of that, we can also set the culture using `@bind:culture`. Both date and number fields use invariant culture and will use the appropriate browser formatting, but if we use a text field, we can change the behavior like this:

```
<input type="text" @bind="SomeNumber" @bind:culture="GBCulture" />
```

Where `GBCulture`, in this case, is a `CultureInfo` object. Lastly, we can set the format using `@bind:format`. This is only implemented for `DateTime` at this point:

```
<input type="text" @bind="SomeDate" @bind:format="MM/dd/yyyy" />
<input type="text" @bind="SomeDate" @bind:format="yyyy-MM-dd" />
```

We now know how we can bind to HTML elements. Next, we will take a look at binding to components.

Binding to components

When binding to components, `Get`, `Set`, and `After` will also work. `Culture`, `Event`, and `Format` will work on some components.

When binding to a component, we use `@bind-{ParameterName}`, so for the parameter `Value`, it would look like this:

```
<InputText @bind-Value="text" />
```

Note the capital **V** in `@bind-Value`. This must match the parameter name exactly (`Value`) on the component, including the casing.

In the background, `@bind-Value` will affect two other parameters, `ValueExpression` and `ValueChanged`. This means you will not be able to set them manually if you use `@bind-Value`. When we change the value, `ValueChanged` will be triggered, and we can listen to the event and make things happen when it changes.

We can also use `Get` and `Set` like this:

```
<InputText @bind-Value:get="text" @bind-Value:set="(value) =>
    {text=value; }" />
<InputText @bind-Value:get="text" @bind-Value:set="Set" />
<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />
```

We must always supply both `Get` and `Set`, and they cannot be combined with `@bind-Value`. These samples use `InputText`, a built-in Blazor component, but this concept works with any parameter on any component.

The same thing goes for `After`. It can be used with any component like this:

```
<InputText @bind-Value="text" @bind-Value:after="() => { }" />
<InputText @bind-Value="text" @bind-Value:after="After" />
<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />
```

We have access to some nice binding features, and they work when binding to components as well as to HTML elements.

Next, we will build an admin interface using bindings.

Building an admin interface

To build a simple admin interface for our blog, we need to be able to do the following:

- List categories
- Edit categories
- List tags
- Edit tags
- List blog posts
- Edit blog posts

If we look at the preceding list, we might notice that some of the things seem similar – perhaps we can build shared components for those. Categories and tags are very similar; they have names, and the name is the only thing we should be able to edit.

Let's make a component for that. The component is going to be responsible for listing, adding, deleting, and updating the object.

Since the object we are working with is either `Category` or `Tag`, we need to be able to call different APIs depending on the object, so our component needs to be generic:

1. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, add a new Razor component and call it `ItemList.razor`.
2. Open the newly created file and at the top of the file, add:

```
@typeparam ItemType
```

`@typeparam` is to make the component generic, and the variable holding the generic type is called `ItemType`.

3. Add a `code` section (if you don't have one already) and add the following lines of code:

```
@code{
[Parameter]
public List<ItemType> Items { get; set; } = new();
[Parameter, EditorRequired]
public required RenderFragment<ItemType> ItemTemplate { get; set; }
}
```

We have two parameters: a list where we can add all the items and an `ItemTemplate` instance that we can use to change how we want the item to be shown.

The `EditorRequired` attribute makes sure that we need to set this value when using the component. Otherwise, Visual Studio will show error messages until we fix it.

In this case, we are using `RenderFragment<T>`, which will give us access to the item inside the template (things will become clearer as soon as we implement it).

4. We also need a couple of events; add the following code to the `code` section:

```
[Parameter]
public EventCallback<ItemType> DeleteEvent { get; set; }
[Parameter]
public EventCallback<ItemType> SelectEvent { get; set; }
```

We added two events. The first is when we delete a tag or a category. We will send an event to the parent component, where we can add the code needed to delete the item. The second is when we select an item so that we can edit the item.

5. Now, it's time to add the UI; replace the top of the file below `@typeparam` to the code tag with:

```
@using System.Collections.Generic
<h3>List</h3>
<table>
    <Virtualize Items="@Items" Context="item">
        <tr>
            <td>
                <button class="btn btn-primary" @onclick="@(()=>
```

```
                    {SelectEvent.InvokeAsync(item); })">
                        Select</button>
            </td>
            <td>@ItemTemplate(item)</td>
            <td>
                <button class="btn btn-danger" @onclick="@(()=>
                    {DeleteEvent.InvokeAsync(item);})">
                        Delete</button>
            </td>
        </tr>
    </Virtualize>
</table>
```

If we look back at *Step 3*, we'll see that we used the variable for the lists and `RenderFragment`.

Then, we use the new `Virtualize` component to list our items. To be fair, we might not have that many categories or tags, but why not use it when we can?

We set the `Items` property to `"Items"` (which is the name of our list), and the `Context` parameter to `"item"`.

We can give the context whatever name we want; we're only going to use it inside the `Virtualize` render template.

We also added two buttons that simply invoke the `EventCallback` instances we added in *Step 4*. Between those buttons, we added `@ItemTemplate(item)`; we want Blazor to render the template, but we also send the current item in the loop.

That means we have access to the item's value inside our template.

Listing and editing categories

With our new component, it's now time to create a component for listing and editing our categories:

1. In the `BlazorWebApp.Client` project, open `_Imports.razor` and make sure the following namespaces are included:

```
@using BlazorWebApp.Client.ReusableComponents
```

2. Right-click the `Pages` folder, select **Add** | **New Folder**, and name the folder `Admin`.

3. In the `Pages/Admin` folder, add a new Razor component and name it `CategoryList.razor`.
4. At the top of the component, replace `<h3>CategoryList</h3>` with the following code:

```
@page "/admin/categories"
@inject IBlogRepository _repository
<h3>Categories</h3>
```

We started with the `@page` directive, telling Blazor that if we navigate to the `"admin/categories"` URL, we will get to the `CategoryList.razor` component. We will add a `using` statement and then inject our repository.

5. The next step is to add a form to edit the categories. Add the following code under the code from the previous step:

```
<EditForm OnValidSubmit="Save" Model="Item">
    <DataAnnotationsValidator />
    <CustomCssClassProvider ProviderType=
        "BootstrapFieldCssClassProvider" />
    <InputText @bind-Value="@Item.Name" />
    <ValidationMessage For="@(()=>Item.Name)" />
    <button class="btn btn-success" type="submit">Save</button>
</EditForm>
```

We added `EditForm`, which will execute the `Save` method if the form validates correctly.

For validation, we added `DataAnnotationsValidator`, which will validate the supplied data against the annotations we added to the `Tag` and `Category` classes.

Since we are using Bootstrap, we want our form controls to look the same, so we added `CustomCssClassProvider`, which we created earlier in this chapter.

We added `InputText` and connected it to a `Category` object called `Item` (which we will add in just a second).

Below that, we added `ValidationMessage`, which will show any errors for the `name` property, and then a **Submit** button.

6. Now, it's time to add our `ItemList` component; under the code we added in the previous step, add this code:

```
<ItemList Items="Items" DeleteEvent="@Delete"
    SelectEvent="@Select" ItemType="Category">
    <ItemTemplate>
        @{
            var item = context as Category;
            if (item != null)
            {
                @item.Name
            }
        }
    </ItemTemplate>
</ItemList>
```

We add our component, and we bind the `Items` property to a list of items (we will create that list in the next step).

We bind the `Select` and `Delete` events to methods, and we specify the type of the list in the `ItemType` property. Then, we have `ItemTemplate`, and since we are using `RenderFragment<T>`, we now have access to a variable called `context`.

We convert that variable to a category and print out the name of the category. This is the template for each item that will be shown on the list.

7. Finally, replace the existing `@code` section with the following code:

```
@code {
    private List<Category> Items { get; set; } = new();
    public Category Item { get; set; } = new();
    protected override async Task OnInitializedAsync()
    {
        Items = (await _repository.GetCategoriesAsync()) ?? new();
        await base.OnInitializedAsync();
    }
    private async Task Delete(Category category)
    {
        try
        {
            await _repository.DeleteCategoryAsync(category.Id!);
```

```
            Items.Remove(category);
        }
        catch { }
    }
    private async Task Save()
    {
        try
        {
            await _repository.SaveCategoryAsync(Item);
            if (!Items.Contains(Item))
            {
                Items.Add(Item);
            }
            Item = new Category();
        }
        catch { }
    }
    private Task Select(Category category)
    {
        try
        {
            Item = category;
        }
        catch { }
        return Task.CompletedTask;
    }
}
```

We added a list to hold all our categories and a variable that holds one item (the item currently being edited). We use `OnInitializedAsync` to load all the categories from the repository.

The `Delete` and `Save` methods call the repository's corresponding methods, and the Select method takes the provided item and assigns it to the item variable (ready to be edited).

We check whether we already have the item in the list before we add it to the list. Run the project and navigate to `/admin/categories`.

8. Try to add, edit, or delete a category, as shown in *Figure 8.1*:

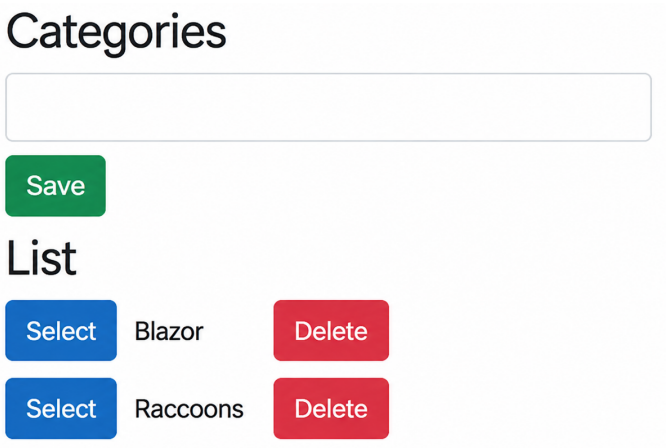


Figure 8.1: The Edit Category view

Now, we need a component for listing and editing tags as well – it is pretty much the same thing, but we need to use `Tag` instead of `Category`.

Listing and editing tags

We just created a component for listing and editing categories; now, we need to create a component for listing and editing tags:

1. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, add a new Razor component called `TagList.razor`.
2. At the top of the component, replace `<h3>TagList</h3>` with the following code:

```
@page "/admin/tags"
@inject IBlogRepository _repository
<h3>Tags</h3>
```

   We started with the `@page` directive telling Blazor that if we navigate to the `"admin/tags"` URL, we will get to the `TagList.Razor` component. Then, we inject our repository.

3. The next step is to add a form to edit the tags. Add the following code under the code from the previous step:

```
<EditForm OnValidSubmit="Save" Model="Item">
    <DataAnnotationsValidator />
    <CustomCssClassProvider
```

```
            ProviderType="BootstrapFieldCssClassProvider" />
        <InputText @bind-Value="@Item.Name" />
        <ValidationMessage For="@(()=>Item.Name)" />
        <button class="btn btn-success" type="submit">Save</button>
    </EditForm>
```

We added `EditForm`, which will execute the `Save` method if the form validates without a problem. For validation, we added `DataAnnotationsValidator`, which will validate the supplied data against the annotations we added to the `Tag` and `Category` classes.

Since we are using Bootstrap, we want our form controls to look the same, so we added `CustomCssClassProvider`, which we created earlier in this chapter.

We added `InputText` and connected it to a `Tag` object called `Item` (which we will add in a moment).

Below it, we add a `ValidationMessage` instance that will show any errors for the `name` property, and then a **Submit** button.

4. Now, it's time to add our `ItemList` component. Under the code we added in the previous step, add this code:

```
<ItemList Items="Items" DeleteEvent="@Delete"
    SelectEvent="@Select" ItemType="Tag">
    <ItemTemplate>
        @{
            var item = context as Tag;
            if (item != null)
            {
                @item.Name
            }
        }
    </ItemTemplate>
</ItemList>
```

We added our component and bound the `Items` property to a list of items (we will create that list in the next step). We bind the `Select` and `Delete` events to methods and specify the `List` type in the `ItemType` property.

Then we have `ItemTemplate`; since we are using `RenderFragment<T>`, we now have access to a variable called `context`. We convert that variable to a tag and print the tag's name.

This is the template for each item shown in the list.

5. Finally, we replace the @code section with the following code:

```
@code {
    private List<Tag> Items { get; set; } = new List<Tag>();
    public Tag Item { get; set; } = new Tag();
    protected async override Task OnInitializedAsync()
    {
        Items = (await _repository.GetTagsAsync()) ?? new();
        await base.OnInitializedAsync();
    }
    private async Task Delete(Tag tag)
    {
        try
        {
            await _repository.DeleteTagAsync(tag.Id!);
            Items.Remove(tag);
        }
        catch { }
    }
    private async Task Save()
    {
        try
        {
            await _repository.SaveTagAsync(Item);
            if (!Items.Contains(Item))
            {
                Items.Add(Item);
            }
            Item = new Tag();
        }
        catch { }
    }
    private Task Select(Tag tag)
    {
        try
        {
            Item = tag;
        }
        catch { }
```

```
            return Task.CompletedTask;
        }
    }
```

We added a list to hold all our tags and a variable that holds one item (the item currently being edited). We use `OnInitializedAsync` to load all the tags from the repository.

The `Delete` and `Save` methods call the API's corresponding methods, and the Select method takes the provided item and assigns it to the `Item` variable (ready to be edited).

We check whether the item is already in the list before adding it.

6. Run the project and navigate to `/admin/tags`.
7. Try to add, edit, and delete a tag, as shown in *Figure 8.2*:

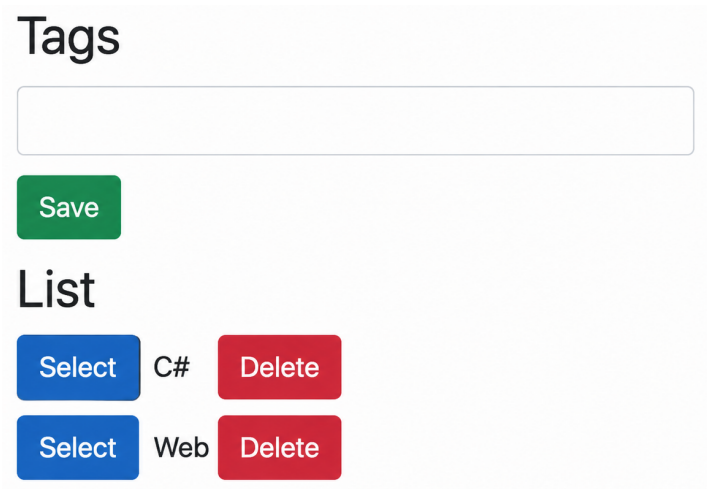


Figure 8.2: The Edit tag view

Now, we need ways to list and edit blog posts.

Listing and editing blog posts

Let's start with listing and editing blog posts:

1. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, add a new Razor component called `BlogPostList.razor`.

2. At the top of the `BlogPostList.razor` file, replace `<h3>BlogPostList</h3>` with the following code:

```
@page "/admin/blogposts"
@inject IBlogRepository _repository
<a href="/admin/blogposts/new">New blog post</a>
@if (posts?.Count == 0)
{
    <p>No blog posts found</p>
}
else if (posts == null)
{
    <p>Loading...</p>
}
else
{
    <ul>
        @foreach (var p in posts)
        {
            <li>
                @p.PublishDate
                <a href="/admin/blogposts/@p.Id">@p.Title</a>
            </li>
        }
    </ul>
}
```

We added a `page` directive, injected our repository, and listed the `blog posts` using a `foreach` loop. We also linked the posts to a URL with the `Id` instance of the blog.

3. Add the following code in the `code` section:

```
private List<BlogPost>? posts = null;

protected override async Task OnInitializedAsync()
{
    await Task.Delay(1000);
    var numberofposts = await _repository.GetBlogPostCountAsync();
    posts = await _repository.GetBlogPostsAsync(numberofposts, 0);
    await base.OnInitializedAsync();
}
```

We added the functionality to load posts from the database and a small delay so that we can see the `Loading...` message for just a brief moment. Now, there is only one thing left in the section: adding a page where we can edit the blog post.

A very popular way of writing blog posts is using Markdown; our blog engine will support that. Since Blazor supports any .NET Standard **dynamic link library** (**DLL**), we will add an existing library called `Markdig`. This is the same engine that Microsoft uses for their `docs` site.

We can extend `Markdig` with different extensions (as Microsoft has done), but in the following steps, let's keep this simple and add support only for Markdown without all the fancy extensions.

4. Under the `BlazorWebApp.Client` project, right-click the **Dependencies** node in Solution Explorer and select **Manage NuGet Packages**.
5. Search for `Markdig` and click **Install**.
6. Add a new class in the `ReusableComponents` folder called `InputTextAreaOnInput.cs`.
7. Open the new file and replace its contents with the following code:

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.AspNetCore.Components.Rendering;
namespace Microsoft.AspNetCore.Components.Forms;
    public class InputTextAreaOnInput : InputBase<string?>
    {
        protected override void BuildRenderTree(
            RenderTreeBuilder builder)
        {
            builder.OpenElement(0, "textarea");
            builder.AddMultipleAttributes(1, AdditionalAttributes);
            builder.AddAttribute(2, "class", CssClass);
            builder.AddAttribute(3, "value",
                BindConverter.FormatValue(CurrentValue));
            builder.AddAttribute(4, "oninput", EventCallback
                .Factory.CreateBinder <string?>(this, __value =>
                    CurrentValueAsString = __value,
                        CurrentValueAsString));
            builder.CloseElement();
        }
        protected override bool TryParseValueFromString(string?
            value, out string? result, [NotNullWhen(false)] out
```

```
                string? validationErrorMessage)
        {
            result = value;
            validationErrorMessage = null;
            return true;
        }
    }
```

The preceding code is taken from Microsoft's GitHub repository; it is how they implement the `InputTextArea` component.

In their build system, they can't handle `.razor` files, so that's why they implement the code this way.

I made one change in Microsoft's code, and that is `oninput`, which used to say `OnChange`. For most cases, `OnChange` will be just fine, which means when I leave the textbox, the value will be updated (and trigger validations). But in our case, we want the preview of the HTML to be updated in real time, which is why we had to implement our own.

One option could have been not to use the `InputTextArea` component and instead use the `TextArea` tag, but we would lose the validation highlighting. This is the way to go if we ever need to customize the behavior of an input control.

I recommend using `.razor` files over `.cs` files if you make many changes to the implementation.

8. Next, in the `Pages/Admin` folder, add a new Razor component called `BlogPostEdit.razor`.
9. At the top of the `BlogPostEdit.razor` file, replace `<h3>BlogPostEdit</h3>` with the following code:

```
@page "/admin/blogposts/new"
@page "/admin/blogposts/{Id}"
@inject IBlogRepository _repository
@inject NavigationManager _manager
@using Markdig;
@using Microsoft.AspNetCore.Components.Forms
```

We add two different `page` directives because we want to be able to create a new blog post and supply an ID to edit an existing one. If we do not supply an ID, the `Id` parameter will be null (or the default).

We inject our repository and `NavigationManager`, and add `using` statements.

10. We also need some variables. Add the following code in the `code` section:

```
[Parameter]
public string? Id { get; set; }
BlogPost Post { get; set; } = new();
List<Category> Categories { get; set; } = new();
List<Tag> Tags { get; set; } = new();
string? selectedCategory = null;
string? markDownAsHTML { get; set; }
```

We added a parameter for the blog post ID (if we want to edit one), a variable to hold the post we are editing, one to hold all the categories, and one to hold all the tags. We also added variables to hold the currently selected category and the Markdown converted to HTML.

11. Now, we need to add the form. Add the following code:

```
<EditForm Model="Post" OnValidSubmit="SavePost">
    <DataAnnotationsValidator />
    <CustomCssClassProvider
        ProviderType="BootstrapFieldCssClassProvider" />
    <InputText @bind-Value="Post.Title"/>
    <ValidationMessage For="()=>Post.Title"/>
    <InputDate @bind-Value="Post.PublishDate"/>
    <ValidationMessage For="()=>Post.PublishDate"/>
    <InputSelect @bind-Value="selectedCategory">
        <option value="0" disabled>None selected</option>
        @foreach (var category in Categories)
        {
            <option value="@category.Id">@category.Name </option>
        }
    </InputSelect>
    <ul>
        @foreach (var tag in Tags)
        {
        <li>
```

```
            @tag.Name
            @if (Post.Tags.Any(t => t.Id == tag.Id))
            {
                <button type="button" @onclick="@(() =>
                    {Post.Tags.Remove(Post.Tags.Single(
                        t=>t.Id==tag.Id)); })">Remove</button>
            }
            else
            {
                <button type="button" @onclick="@(()=> {
                    Post.Tags.Add(tag); })">Add</button>
            }
        </li>
        }
    </ul>
        <InputTextAreaOnInput @bind-Value="Post.Text"
            @onkeyup="UpdateHTML"/>
        <ValidationMessage For="()=>Post.Text"/>
        <button type="submit" class="btn btn-success">Save</button>
    </EditForm>
```

We added an `EditForm`, and when we submit the form (if it is valid), we execute the `SavePost` method.

We added `DataAnnotationsValidator`, which validates our model against the data annotations in the class.

We added `CustomCssClassProvider` so that we get the correct Bootstrap class names. Then, we added components for the `title`, `publish date`, `category`, `tags`, and, last but not least, the `text` (the blog post's content).

We added the text using the component we created in *Step 6* (the component that updates with each keystroke).

Finally, we hook up the `@onkeyup` event to update the preview with each keystroke.

12. We still need to add our `SavePost` method. Add the following code in the `code` section:

```
public async Task SavePost()
{
    if (!string.IsNullOrEmpty(selectedCategory) &&
        Categories.Count > 0)
```

```
    {
        var category = Categories.FirstOrDefault(c =>c.Id ==
            selectedCategory);
        if (category != null)
        {
            Post.Category = category;
        }
    }
    await _repository.SaveBlogPostAsync(Post);
    _manager.NavigateTo("/admin/blogposts");
}
```

13. Now, it's time to show the preview. Add the following code just below the `EditForm` close tag:

```
@((MarkupString)markDownAsHTML)
```

We use `MarkupString` to make sure Blazor outputs the HTML code without escaping the characters. You might remember that from *Chapter 6*, *Understanding Basic Blazor Components*.

14. Now, it is time to set up `Markdig`. Add the following code somewhere in the `code` section:

```
MarkdownPipeline pipeline = default!;
protected override Task OnInitializedAsync()
{
    pipeline = new MarkdownPipelineBuilder()
             .UseEmojiAndSmiley()
             .Build();
    return base.OnInitializedAsync();
}
```

To configure `Markdig`, we need to create a pipeline. As I mentioned earlier in the chapter, this is the engine Microsoft uses for its documentation site. It has many extensions available, including source code highlighting and emoticons.

We also added emoticons to the pipeline to make it a little more fun.

15. We must also add code to load the data (`blog posts`, `categories`, and `tags`). Add the following methods to the code section:

```
protected void UpdateHTML()
{
    markDownAsHTML = Markdig.Markdown.ToHtml(Post.Text, pipeline);
}
protected override async Task OnParametersSetAsync()
    {
        if (Id != null)
        {
            var p = await _repository.GetBlogPostAsync(Id);
            if (p != null)
            {
                Post = p;
                if (Post.Category != null)
                {
                    selectedCategory = Post.Category.Id;
                }
                UpdateHTML();
            }
        }
        Categories = (await _repository.GetCategoriesAsync());
        Tags = (await _repository.GetTagsAsync());
        await base.OnParametersSetAsync();
    }
```

16. Now, run the site. You might need to rerun the database script to populate your database table, especially if you are using the source on GitHub because each chapter has a new container. Navigate to `/admin/blogposts`, click on a **blog post** to edit it, and test the new Markdown support.
Figure 8.3 shows the **Edit** page with Markdown support:

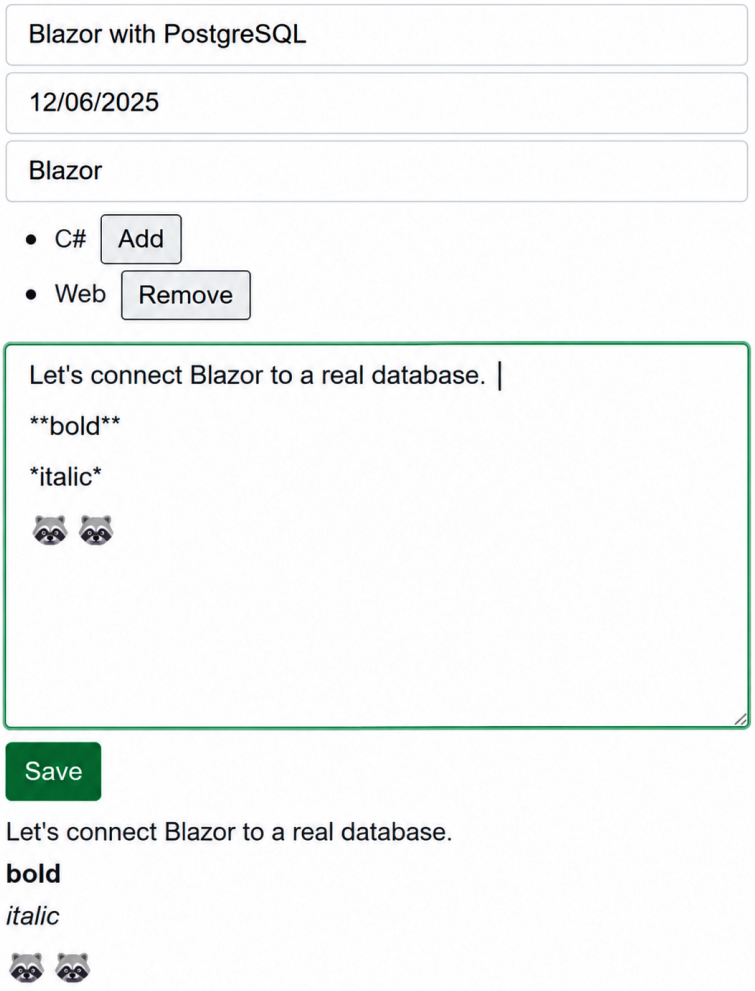


Figure 8.3: The Edit page with Markdown support

We still have one more thing to do: we need to ensure that the blog post page shows a converted HTML version of the Markdown.

17. Open `/Pages/Post.razor` and add the following `using` statement at the top of the file:

```
@using Markdig;
```

18. Add the following code to the `code` section:

```
MarkdownPipeline pipeline = default!;
protected override Task OnInitializedAsync()
{
    pipeline = new MarkdownPipelineBuilder()
             .UseEmojiAndSmiley()
             .Build();
    return base.OnInitializedAsync();
}
```

19. Finally, replace the following row:

```
@((MarkupString)BlogPost.Text)
```

With this:

```
@((MarkupString)Markdig.Markdown.ToHtml(BlogPost.Text, pipeline))
```

Great job! Now, we have an admin interface up and running, so we can start writing blog posts.

Looking at the code we wrote, none of the textboxes have labels. We could go through and add labels everywhere we use a textbox, but that quickly leads to duplicate code.

Instead, we can solve this by adding an abstraction layer on top of the built-in components (or any third-party components we use). This allows us to define things like labels, validation messages, and styling in one place and reuse them across the application.

Why use an abstraction layer for your components

An abstraction layer has saved me countless times. Granted, adding an abstraction layer does take some time and effort, but I promise you that you will get that time back.

Why should we do this? Well, for several reasons: if we are using Bootstrap, for example, and we want to upgrade Bootstrap to the latest version, there might be classes that have changed.

Using components makes it easy to change only those components. Plus, it makes it easier to change component vendors in the future if you have your own components encapsulating the third-party components.

But the real reason is that if we add a layer, we can set the team's programming style or language. Everything we build will have the same default values, the same access to properties, and the same UX. We can add functionality, but in most cases, it is more important to limit the functionality.

A third-party component has a lot of functionality; it should cater to many different use cases. But this also means that your team now has access to many different functionalities that can make the UX different for each developer who implements the functionality.

Let's add a couple of shared components to the project.

Adding shared components

The first shared component is a textbox with a label and a validation message built in.

If we take a look at our `CategoryList` component, the code looks like this:

```
<InputText @bind-Value="@Item.Name" />
<ValidationMessage For="@(()=>Item.Name)" />
```

And a label using Bootstrap looks something like this:

```
<label for="validationCustomCategoryName" class="form-label">
    Category name</label>
<div class="input-group has-validation">
<input type="text" class="form-control" id="validationCustomCategoryName">
<div class="invalid-feedback">
Please choose a category name.
</div>
</div>
```

Let's see if we can combine these; some features are already built-in. Since we only add a layer, we don't have to handle as much functionality. Rather, we need to send the parent component values to the encapsulated component.

Let's look at some code to see what's going on:

1. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, add a new Razor component and call it `BlogInputText.razor`.
2. In the `code` section, add the following code:

```
[Parameter]
public string Id { get; set; } = Guid.NewGuid().ToString();
[Parameter]
public string? Label { get; set; }
[CascadingParameter]
public required EditContext CurrentEditContext { get; set; }
[Parameter,EditorRequired]
```

```
public required string Value { get; set; }
[Parameter]
public EventCallback<string> ValueChanged { get; set; }
[Parameter]
public required Expression<Func<string>>
    ValueExpression { get; set; }
```

Let's take a look at what is happening.

First, we add a parameter to generate an ID we can use for the label tag in the next step. Then we add a string that can contain text for our label. If we do have some text, we render the label. If it is `null`, we don't render the label. I prefer not to have a `ShowLabel` property; if there is text, it should show the label. We also have the current edit context, which we will use to send to the next level of components.

In our form, we have an `EditForm`; the `EditForm` will send the `EditContext` to all the child components and will keep track of the state of the form. We want to grab that edit context and send it to all the components inside of this component as well.

Then, we have the trio of value parameters: `Value`, `ValueChanged`, and `ValueExpression`.

3. In the non-code part of the page, add the following (replacing the `H3` tag):

```
@using System.Linq.Expressions
<CascadingValue Value="CurrentEditContext">
    @if(!string.IsNullOrWhiteSpace(Label))
    {
        <label for="@Id" class="form-label">@Label</label>
    }
    <InputText id="@Id" Value="@Value" ValueChanged="ValueChanged"
        ValueExpression="ValueExpression" />
    <ValidationMessage For="@ValueExpression" />
</CascadingValue>
```

First, we grab the `CurrentEditContext` and pass it to a child component; this way, all children will have the same edit context as the parent `EditForm`. If we have any text in the `Label` parameter, we should show the label. Then we add the `InputText`, the built-in component. If we want to do this with a third-party library, we would do it in a similar way.

Next is where things get a bit more complicated: we could have simply used `@bind-Value`, which would notify Blazor that a change has occurred, but it would notify the `EditContext` that the Value parameter of our component has changed, not the model. So, instead of doing that, we set the `Value` and `ValueChanged` parameters to the values we send to the component. This way, the notification of a value change will directly indicate that the model has changed. The `ValueExpression` ensures that the `EditContext` is notified of the change and displays the corresponding validation message.

To be honest, in this example, it doesn't really matter, but if we were using a third-party component with validation built in, it might not work (depending on how they build the component). Using this method should ensure that validation will always work with third-party components. Finally, we add a `ValidationMessage`, which displays any validation errors for the field. By using the same `ValueExpression`, we ensure that both the input and the validation message are tied to the same property in the model, so the correct validation messages are shown.

Now, we need to use this component. Let's change the `TagList` first. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, open `TagList.razor`.

Right now, we have the following code:

```
<InputText @bind-Value="@Item.Name" />
<ValidationMessage For="@(()=>Item.Name)" />
```

Replace this code with the following:

```
<BlogInputText @bind-Value="@Item.Name" Label="Name" />
```

Now, isn't that kind of elegant? By using `@bind-Value`, .NET is automatically setting all the other parameters for this component.

4. Let's do the same with the `CategoryList`. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, open `CategoryList.razor`.
   Find the following code:

```
<InputText @bind-Value="@Item.Name" />
<ValidationMessage For="@(()=>Item.Name)" />
```

And replace it with the following code:

```
<BlogInputText @bind-Value="@Item.Name" Label="Name" />
```

This kind of change makes me genuinely happy – it simplifies usage, makes the UI easier to understand, and removes the need for duplicated code. Even though we know how to do this now, I want to add one more example that perhaps showcases the really nice benefits of working this way. Let's create a button component as well.

Creating a button component

This component will be a bit more to take in:

1. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, add a new component and name it `BlogButton.razor`.
2. Replace the content with this code:

```
@using Microsoft.AspNetCore.Components.Forms
<button
  type="@InternalButtonType"
  disabled="@Disabled"
  class="@InternalCssClass"
  title="@Title"
  @onclick="OnButtonClick">
  @ChildContent
</button>
```

   We are adding a normal HTML button, nothing fancy. We also add functionality to change the type (button or submit), whether it is disabled or not, which CSS class it should have, a method to run, and the title.

3. Add the following code to the `code` section:

```
[CascadingParameter]
public EditContext? EditContext { get; set; }

[Parameter]
public RenderFragment? ChildContent { get; set; }
```

   Here, we are doing the same thing as with the `BlogTextbox`, and we bring in the `EditContext`, which we will use in just a bit. We also have a `RenderFragment` for the content of the button.

Next, add the following code. This is where things get interesting. We will use the `EditContext` to automatically disable the button when the form is invalid:

```
private bool? _disabled = null;
private string? _disabledHelpText = "";
private string formerrors = "";

[Parameter]
    public new bool Disabled
    {
        get
        {
            if (_disabled == null || (_disabled != null &&
                !_disabled.Value))
            {
                if (EditContext != null)
                {
                    formerrors = "";
                    var validationmessages = EditContext?
                        .GetValidationMessages();
                    if (validationmessages != null)
                    {
                        foreach (var m in validationmessages)
                        {
                            formerrors += m + (MarkupString)" \r\n";
                        }
                    }
                    return (!EditContext.IsModified() ||
                        !(validationmessages.Count() == 0));
                }
                else
                {
                    return false;
                }
            }
            else
            {
                return _disabled!.Value;
            }

        }
```

```
        set => _disabled = value;
    }
```

First, we add a couple of private fields we will use in the component, then we add a property indicating whether the button is disabled. It will use `EditContext` to check whether there are any errors in the form, save those errors in a variable, and, if the form is OK, enable the button; if it is not, disable it.

There is a downside to this implementation: for the validation to trigger, we need to click somewhere else on the page to trigger the change in the field. So, now, the button will be disabled if the form is not OK.

4. Add the following code. This adds support for displaying help text on the button and allows us to show a different message when the button is disabled (for example, explaining why it can't be clicked):

```
private string? Title => Disabled && !string.IsNullOrWhiteSpace(
    DisabledHelpText) ? DisabledHelpText : HelpText;
[Parameter]
public string? DisabledHelpText { get { return _disabledHelpText +
    (MarkupString)"\r\n" + formerrors; } set { _disabledHelpText =
        value; } }

[Parameter]
public string? HelpText { get; set; }
```

This code will set a `Title` for the button, which will be displayed when you hover over it. We can also set a `HelpText` or a `DisabledHelpText`. If the function is disabled for any reason – it might not be related to the form; it might be a permission issue – we will display text explaining why the button is disabled. It will also display any form errors on the button, so it is easy to see which form element is the problem without having to scroll to it.

5. Sometimes, we might want to use the button, but without a form, to simply execute a method. To do this, add the following code:

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

    private string InternalButtonType =>
        OnClick.HasDelegate ? "button" : "submit";
```

```
    private async Task OnButtonClick(EventArgs args)
    {
        if (OnClick.HasDelegate)
        {
            await OnClick.InvokeAsync(args);
        }
    }
```

If we have a delegate for `OnClick`, we want the button to simply be a button. If we don't have a delegate, we assume the button is used inside an `EditForm`. When the button is clicked, the `OnButtonClick` method will run.

6. Now, we get to the really juicy part. Let's add an `enum`; we can add it in the code section:

```
public enum ButtonType
{
    Save,
    Cancel,
    Delete,
    Remove,
    Select
}
```

Notice that we are not using vocabulary like *Primary* or *Danger* – that's Bootstrap lingo. But what we want to know is what the button is used for, which is why we use names like *Save*, *Cancel*, etc.

7. Then, we add a parameter for `ButtonType` like this:

```
[Parameter] public ButtonType Type { get; set; }

private string InternalCssClass
{
    get
    {
        return Type switch
        {
            ButtonType.Save => "btn btn-success",
            ButtonType.Cancel => "btn btn-danger",
            ButtonType.Delete => "btn btn-danger",
            ButtonType.Remove => "btn btn-danger",
```

```
            ButtonType.Select => "btn btn-primary",
            _ => "btn btn-primary"
        };
    }
}
```

We add a parameter for `ButtonType` and an internal property that translates the *Save* use case, for example, into a Bootstrap CSS class.

Now our team doesn't have to bother remembering which Bootstrap class they should use. They know it's a button, and they know what the button is used for. The component takes care of the rest.

Let's test it out!

8. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, open `TagList.razor`. Replace the following line:

```
<button class="btn btn-success" type="submit">Save</button>
```

With this new line:

```
<BlogButton Type="BlogButton.ButtonType.Save">Save</BlogButton>
```

9. If you run the project now, you will see that the button is disabled if we haven't made any changes to the form and will become enabled if we add something to the textbox.
10. Let's do the same for `CategoryList`. In the `BlazorWebApp.Client` project, in the `Pages/Admin` folder, open `CategoryList.razor`. Replace the following line:

```
<button class="btn btn-success" type="submit">Save</button>
```

With this new line:

```
<BlogButton Type="BlogButton.ButtonType.Save">Save</BlogButton>
```

There are more places we can modify and add this button to, but let's not spend time on that for now. If you want, you can return to this and make sure we are using the new button and `InputText` everywhere.

At this point, we have one more component to build.

Locking the navigation

In .NET 7, we got a new component called `NavigationLock`. Right now, if we write a blog post and click somewhere in the menu, our changes will be lost. The same thing happens if we change the URL and press *Enter*. With `NavigationLock`, we can prevent that from happening.

`NavigationLock` can prevent us from leaving the page and navigating to another page inside our site. In that case, we can show a custom message using JavaScript. If we navigate to another site, it can trigger a warning, but we don't have control over the message shown. This functionality is built into the browser.

We will implement this in the same way we did with `FieldCssClassProvider`, as a reusable component. We want to check whether our `EditContext` has any changes made so we can trigger the navigation lock:

1. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, add a new Razor component and name it `BlogNavigationLock.razor`.
2. At the top of the component, add the following code:

```
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime
@implements IDisposable
```

   We inject an `IJSRuntime` interface to make JavaScript calls. We will return to JavaScript interop in *Chapter 12*, *JavaScript Interop*.

   We also implement the `IDisposable` interface.

3. In the code section, add the following code:

```
[CascadingParameter]
public required EditContext CurrentEditContext { get; set; }
public string InternalNavigationMessage { get; set; } =
    "You are about to lose changes, are you sure
        you want to navigate away?";
public bool CheckNavigation { get; set; } = true;
```

   We have a CascadingParameter, which gets the current `EditContext`, just as we did with `FieldCssClassProvider`.

We also added a string for the message displayed when we try to navigate away from the page.

4. When a change happens in `EditContext`, we need to update the component and make sure it locks the navigation. Add the following code:

```
protected override Task OnInitializedAsync()
{
    CurrentEditContext.OnFieldChanged += OnFieldChangedAsync;
    return base.OnInitializedAsync();
}
private async void OnFieldChangedAsync(object?
Sender,FieldChangedEventArgs args)
{
    await InvokeAsync(StateHasChanged);
}
void IDisposable.Dispose()
{
    CurrentEditContext.OnFieldChanged -= OnFieldChangedAsync;
}
```

We start listening for field changes, and when one occurs, we call the `StateHasChanged` method to update the component.

`InvokeAsync` is required because the call is initiated from another thread.

We also override the `Dispose` method and remove the event listener.

5. In the code section, add the following code:

```
private async Task OnBeforeInternalNavigation
    (LocationChangingContext context)
{
    if (CurrentEditContext.IsModified() && CheckNavigation)
    {
        var isConfirmed = await
            JSRuntime.InvokeAsync<bool>("confirm",
            InternalNavigationMessage);
        if (!isConfirmed)
        {
            context.PreventNavigation();
        }
```

```
    }
}
```

This method will make a JavaScript call whenever `EditContext` (or the model) changes, showing a confirmation dialog with the message we added. If we do not confirm, the navigation will be prevented.

6. Now, we can add the `NavigationLock` component. Just under the directives, add the following code:

```
<NavigationLock
    ConfirmExternalNavigation="@(CurrentEditContext.IsModified() &&
        CheckNavigation)"
    OnBeforeInternalNavigation="OnBeforeInternalNavigation" />
```

This `NavigationLock` component will prevent both external navigation (to another site) and internal navigation (to another page on our blog). It checks whether `EditContext` (model) has any changes and prevents external navigation. On internal navigation, it will execute the `OnBeforeInternalNavigation` method, which checks whether `EditContext` has been changed.

7. In `Pages/Admin/BlogPostEdit.razor`, add the new Razor component we created just below `CustomCssClassProvider`:

```
<BlogNavigationLock @ref="NavigationLock"/>
```

This will get `EditContext` from the cascading value and execute the code we just wrote.

8. In the `code` section, add the following:

```
BlogNavigationLock? NavigationLock { get; set; }
```

9. In the `SavePostAsync` method, just before navigating to `admin/blogposts`, add the following:

```
NavigationLock?.CurrentEditContext.MarkAsUnmodified();
```

When saving the object, the `EditContext` doesn't know that, so we tell it that the model is now unmodified and that navigation should not be stopped.

10. Now run the site, navigate to `Admin/BlogPosts`, and click a blog post.

11. Try to navigate to another site (it should work)
12. Try to navigate to another page (it should work)
13. Change the blog post.
14. Try navigating to another site (it should display a message box):

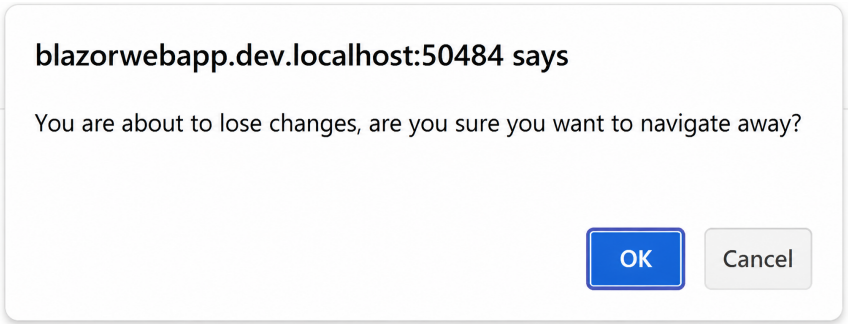


Figure 8.4: Navigation lock

Awesome! We have implemented another reusable component.

Summary

This chapter taught us how to create forms and make API calls to get and save data.

We built custom input controls and added Bootstrap styling. Most business apps use forms, and we can add logic close to the data by annotating data.

We also created multiple reusable components and used many of the things we discussed in previous chapters. We even touched on JavaScript interop, which we will go into more detail about in *Chapter 12*, *JavaScript Interop*.

The validation and input control functionality that Blazor offers will help us build amazing applications and give our users a great experience. You may notice that the admin pages are currently wide open. We need to secure our blog with login functionality, but we will come to that in *Chapter 10*, *Adding Authentication and Authorization*.

In the next chapter, we will create a web API to retrieve data when running components, such as `InteractiveAuto` or `InteractiveWebAssembly`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/cBPYbekR35`

9

Creating an API

When running Blazor using WebAssembly (InteractiveWebAssembly or InteractiveAuto), everything runs in the browser. That means we can't use the direct database calls we created earlier. Instead, we need to create an API that the client can communicate with to retrieve and update data on the server. For that to work, we need an API to access the data.

In this chapter, we will create a web API using Minimal API. When using Blazor Server, the API will be secured by the page (if we add an Authorize attribute), so we get that for free. But with WebAssembly, everything will be executed in the browser, so we need something that WebAssembly can communicate with to update the data on the server.

To do this, we will need to cover the following topics:

- Creating the service
- Creating the client

Technical requirements

Make sure you have read the previous chapters or use the `Chapter08` folder as a starting point.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter09`.

Creating the service

There are many ways to create a service, such as via REST.

For those who haven't worked with REST before, **REST** stands for **representational state transfer**. Simply put, it is a way for machines to talk to other devices using HTTP.

With REST, we use different HTTP verbs for different operations. They could look something like this:

URI	Verb	Action
`/BlogPosts`	Get	Gets a list of blog posts
`/BlogPosts`	Post	Creates a new blog post
`/BlogPosts/{id}`	Get	Gets a blog post with a specific ID
`/BlogPosts/{id}`	Put	Replaces a blog post
`/BlogPosts/{id}`	Patch	Partially updates a blog post
`/BlogPosts/{id}`	Delete	Deletes a blog post.

Table 9.1: REST calls

We will implement an API for **tags**, **categories**, and **blog posts**.

Since the API takes care of whether the `post` should be created, we'll cheat and only implement `Put` (replace) because we don't know whether we are creating or updating the data. This is a simplification for demo purposes.

We will implement the API in the **BlazorWebApp** project.

Learning about Minimal APIs

Before we jump into implementing the Minimal API, let's take a moment to learn about it. Back in November 2019, one of the members of the **Distributed Application Runtime** (**Dapr**) team wrote a couple of tutorials on how to build a distributed calculator using different languages.

They had examples using Go, Python, Node.js, and .NET Core. The code showed how much harder it was to write a distributed calculator in C# than in the other languages.

Microsoft asked various non-.NET developers what their perception of C# was. Their response wasn't great.

Then, Microsoft asked them to complete a tutorial using an early version of Minimal API.

After the tutorial, they were asked about their perception, and their response had shifted and was now more positive – it felt like home.

The goal of Minimal APIs was to reduce complexity and ceremony and embrace minimalism. I initially thought that "minimal" meant I wouldn't be able to do everything. But after digging

deeper, I realized that it's not about limiting functionality; it's about reducing boilerplate and making the code simpler.

From my point of view, Minimal APIs are a much nicer way to code APIs. At my former workplace, we switched to Minimal APIs because we thought the syntax was much nicer.

A very simple example of adding a Minimal API is just adding this line in `Program.cs`:

```
app.MapGet("/api/helloworld", () => "Hello world!");
```

Here we are saying that if we navigate to a URL `"/api/helloworld"`, we return a string with `"Hello World"`.

This is, of course, the simplest example possible, but it is possible to implement more complex things as well, as we will see in the next section.

Adding the API controllers

We have four data models: **BlogPost**, **Tag**, **Category**, and **Comment**.

Let's create four different files, one for each data model, to demonstrate that there are friendly ways to add more complex APIs using Minimal APIs. For a small project, it would probably make more sense to add everything in `Program.cs`.

Adding APIs for handling blog posts

Let's start by adding the API methods for handling blog posts.

Execute the following steps to create the API:

1. In the `BlazorWebApp` project, add a new folder called `Endpoints`.
2. In the `Endpoints` folder, create a class called `BlogPostEndpoints.cs`. The idea is to create an extension method we can use later in `Program.cs`. Add these `using` statements at the top of the file:

   ```
   using BlazorWebApp.Client.Models;
   using BlazorWebApp.Client.Interfaces;
   using Microsoft.AspNetCore.Authorization;
   using Microsoft.AspNetCore.Mvc;
   ```

3. Then replace the class with the following code:

   ```
   public static class BlogPostEndpoints
   {
       public static void MapBlogPostApi(this WebApplication app)
   ```

```
    {
        app.MapGet("/api/BlogPosts",
        async (IBlogRepository repository, [FromQuery]
            int numberofposts, [FromQuery] int startindex) =>
        {
            return Results.Ok(await repository
                .GetBlogPostsAsync(numberofposts, startindex));
        });
    }
}
```

As we are creating an extension method, we must ensure the class is static. The `MapBlogPostApi` method uses the `this` keyword, which makes the method available on any `WebApplication` class.

We set up the Minimal API by using `MapGet` and a path, which means that the method will run if we access that path with the correct parameters using a `Get` verb.

The method takes a couple of parameters. The first is of the `IBlogRepository` type, which will use dependency injection to get an instance of the class we need—in this case, `BlogRepositoryEntityFrameworkDirectAccess`, which will access the database using Entity Framework.

The other parameters will use the query string (since we are using the `query` attribute); in most cases, a Minimal API will figure these things out, but it's never wrong to nudge it in the right direction.

We have created a method that returns the data directly from the database (the same repository the Blazor Server project is using). We also need to make sure to call it from `Program.cs`.

4. In `Program.cs`, add the following namespace:

```
using BlazorWebApp.Endpoints;
```

5. Also, add the following code just above `app.Run();`:

```
app.MapBlogPostApi();
```

6. It's time to test the API. Start the project. Go to the following URL: `https://localhost:{REPLACEWITHYOURPORTNUMBER}/api/BlogPosts?`

`numberofposts=10&startindex=0` (the port number might be something else). We will get some JSON back with a list of our blog posts.
We are off to a good start! Now, we need to implement the rest of the API as well.

7. In the `Endpoints/BlogPostEndpoints.cs` file, in the `MapBlogPostApi` method, add this code to get the blog post count:

```
app.MapGet("/api/BlogPostCount",
async (IBlogRepository repository) =>
{
    return Results.Ok(await repository.GetBlogPostCountAsync());
});
```

We use the GET verb again, but this time with a different route (`/api/BlogPostCount`) that returns the total number of blog posts.

8. We also need to be able to retrieve a single blog post. Add the following code:

```
app.MapGet("/api/BlogPosts/{*id}",
async (IBlogRepository repository, string id) =>
{
    return Results.Ok(await repository.GetBlogPostAsync(id));
});
```

In this case, we are using the `Get` verb but with another URL containing the ID for `Post` that we want to get.

We are using a string as an ID. In some databases, such as RavenDB, IDs can contain slashes, for example, `CollectionName/IdOfThePost`.

To support IDs that contain slashes, we add * to the route parameter. This tells the routing system to treat everything after the route as part of the ID. Without it, the slash would be interpreted as a separator in the URL, and the endpoint would not match.

Next, we need an API that is protected, typically one that updates or deletes things.

9. Let's add an API that saves a blog post. Add the following code under the code we just added:

```
app.MapPut("/api/BlogPosts",
async (IBlogRepository repository, [FromBody] BlogPost item) =>
{
```

```
    return Results.Ok(await repository.SaveBlogPostAsync(item));
}).RequireAuthorization();
```

As I mentioned earlier in this chapter, we will only add one API for creating and updating blog posts, and we will use the Put verb (replace) to do that. We have added the RequireAuthorization method at the end, which will ensure that the user must be authenticated to call the method.

10. Next up, we add code for deleting `blog posts`. To do this, add the following code:

```
app.MapDelete("/api/BlogPosts/{*id}",
async (IBlogRepository repository, string id) =>
{
    await repository.DeleteBlogPostAsync(id);
    return Results.Ok();
}).RequireAuthorization();
```

In this case, we use the Delete verb, and just as with saving, we add the `RequireAuthorization` method at the end. This tells the API that the endpoint should only be accessible to authenticated users.

Next, we need to do this for `Categories` and `Tags` as well.

Adding APIs for handling categories

Let's start with `Categories`. Follow these steps:

1. In the `Endpoints` folder, add a new class called `CategoryEndpoints.cs`. Replace the code with the following:

```
namespace BlazorWebApp.Endpoints;

using BlazorWebApp.Client.Interfaces;
using BlazorWebApp.Client.Models;
using Microsoft.AspNetCore.Mvc;

public static class CategoryEndpoints
{
    public static void MapCategoryApi(this WebApplication app)
    {
        app.MapGet("/api/Categories",
        async (IBlogRepository repository) =>
        {
```

```
            return Results.Ok(await
                repository.GetCategoriesAsync());
        });
        app.MapGet("/api/Categories/{*id}",
        async (IBlogRepository repository, string id) =>
        {
            return Results.Ok(await
                repository.GetCategoryAsync(id));
        });

        app.MapPut("/api/Categories",
        async (IBlogRepository repository,
            [FromBody] Category item) =>
        {
            return Results.Ok(await
                repository.SaveCategoryAsync(item));
        }).RequireAuthorization();

        app.MapDelete("/api/Categories/{*id}",
        async (IBlogRepository repository, string id) =>
        {
            await repository.DeleteCategoryAsync(id);
            return Results.Ok();
        }).RequireAuthorization();
    }
}
```

2. In `Program.cs`, add the following code just above `app.Run();`:

```
app.MapCategoryApi();
```

These are all the methods needed to handle categories.

This is basically the same thing that we did with blog posts in the previous section.

Next, let's do the same thing with `Tags`.

Adding APIs for handling tags

Let's do the same things for `Tags` by following these steps:

1. In the `Endpoints` folder, add a new class called `TagEndpoints.cs`. Add the following code:

```
using BlazorWebApp.Client.Models;
using BlazorWebApp.Client.Interfaces;
using Microsoft.AspNetCore.Mvc;
namespace BlazorWebApp.Endpoints;

public static class TagEndpoints
{
    public static void MapTagApi(this WebApplication app)
    {
        app.MapGet("/api/Tags",
        async (IBlogRepository repository) =>
        {
            return Results.Ok(await repository.GetTagsAsync());
        });
        app.MapGet("/api/Tags/{*id}",
        async (IBlogRepository repository, string id) =>
        {
            return Results.Ok(await repository.GetTagAsync(id));
        });
        app.MapPut("/api/Tags",
        async (IBlogRepository repository, [FromBody] Tag item) =>
        {
            return Results.Ok(await repository.SaveTagAsync(item));
        }).RequireAuthorization();
        app.MapDelete("/api/Tags/{*id}",
        async (IBlogRepository repository, string id) =>
        {
            await repository.DeleteTagAsync(id);
            return Results.Ok();
        }).RequireAuthorization();
```

```
    }
}
```

2. In `Program.cs`, add the following code just above `app.Run();`:

```
app.MapTagApi();
```

Last but not least, let's do the same for `Comments`.

Adding APIs for handling comments

Let's do the same things for `Comments` by following these steps:

1. In the `Endpoints` folder, add a new class called `CommentEndpoints.cs`. Add the following code:

```
using BlazorWebApp.Client.Models;
using BlazorWebApp.Client.Interfaces;
using Microsoft.AspNetCore.Mvc;
namespace BlazorWebApp.Endpoints;

public static class CommentEndpoints
{
    public static void MapCommentApi(this WebApplication app)
    {
        app.MapGet("/api/Comments/{*blogPostId}",
            async (IBlogRepository repository, string blogPostId) =>
            {
                return Results.Ok(await
                    repository.GetCommentsAsync(blogPostId));
            }).RequireAuthorization();

        app.MapPut("/api/Comments",
            async (IBlogRepository repository,
                [FromBody] Comment item) =>
            {
                return Results.Ok(await
                    repository.SaveCommentAsync(item));
            }).RequireAuthorization();

        app.MapDelete("/api/Comments/{*id}",
            async (IBlogRepository repository, string id) =>
```

```
            {
                await repository.DeleteCommentAsync(id);
                return Results.Ok();
            });
    }
}
```

2. In `Program.cs`, add the following code just above `app.Run();`:

```
app.MapCommentApi();
```

Great! We have an API! Now, it's time to create the client that will access the API.

Creating the client

To access the API, we need to create a client. There are many ways of doing this, but we will do it the simplest way possible by writing the code ourselves.

The client will implement the same `IBlogRepository` interface. This way, we have the same code to call regardless of which implementation we are using: direct database access with `BlogRepositoryEntityFrameworkDirectAccess` or over the network using `BlogApiWebClient`, which we are going to create next.

Let's get started:

1. Under the `BlazorWebApp.Client` node, right-click **Dependencies** and select **Manage NuGet Packages**.
2. Search for `Microsoft.AspNetCore.Components.WebAssembly.Authentication` and click **Install**.
3. Also, search for `Microsoft.Extensions.Http` and click **Install**.
4. In the `BlazorWebApp.Client` project, in the root of the project, add a new class and name it `BlogApiWebClient.cs`.
5. Open the newly created file and add the following namespaces:

```
using BlazorWebApp.Client.Models;
using BlazorWebApp.Client.Interfaces;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using System.Net.Http.Json;
using System.Text.Json;
```

6. Add `IBlogRepository` to the class and make it public like this:

```
namespace BlazorWebApp.Client;
public class BlogApiWebClient(IHttpClientFactory factory) :
    IBlogRepository
{
}
```

Some API calls will be public (do not require authentication), but `HttpClient` will be configured to require a token.

Blazor can automatically handle token management when configured correctly; in this case, we call it `Api`. This, of course, assumes that everything is configured correctly.

7. Now, it's time to implement calls to the API. Let's begin with the `Get` calls for blog posts. Add the following code:

```
public async Task<BlogPost?> GetBlogPostAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<BlogPost>($"/api/BlogPosts/{id}");
}
public async Task<int> GetBlogPostCountAsync()
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<int>("/api/BlogPostCount");
}
public async Task<List<BlogPost>> GetBlogPostsAsync(
    int numberofposts, int startindex)
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<List<BlogPost>>(
            $"/api/BlogPosts?numberofposts={numberofposts}
                &startindex={startindex}")??[];
}
```

We use the `HttpClientFactory` we injected, then call `GetFromJsonAsync`, which automatically downloads the JSON and converts it into the class that we supply to the generic method.

8. Next, we add the API calls that need authentication, such as saving or deleting a blog post. Add the following code under the code from *step 7*:

```
public async Task<BlogPost?> SaveBlogPostAsync(BlogPost item)
{
    var httpclient = factory.CreateClient("Api");
    var response = await httpclient.
        PutAsJsonAsync<BlogPost>("/api/BlogPosts", item);
    return await response.Content.ReadFromJsonAsync<BlogPost>();
}
public async Task DeleteBlogPostAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    await httpclient.DeleteAsync($"/api/BlogPosts/{id}");
}
```

9. Now, we need to do the same for `Categories`. Add the following code to the `BlogApiWebClient` class:

```
public async Task<List<Category>> GetCategoriesAsync()
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<List<Category>>($"/api/Categories") ?? [];
}
public async Task<Category?> GetCategoryAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<Category>($"/api/Categories/{id}");
}
public async Task DeleteCategoryAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    await httpclient.DeleteAsync($"/api/Categories/{id}");
}
public async Task<Category?> SaveCategoryAsync(Category item)
```

```
{
    var httpclient = factory.CreateClient("Api");
    var response = await httpclient
        .PutAsJsonAsync<Category>("/api/Categories", item);
    return await response.Content.ReadFromJsonAsync<Category>();
}
```

10. Next up, we will do the same for `Tags`. Add the following code just under the code we just added:

```
public async Task<Tag?> GetTagAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<Tag>($"/api/Tags/{id}");
}
public async Task<List<Tag>> GetTagsAsync()
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient
        .GetFromJsonAsync<List<Tag>>($"/api/Tags") ?? [];
}
public async Task DeleteTagAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    await httpclient.DeleteAsync($"/api/Tags/{id}");
}
public async Task<Tag?> SaveTagAsync(Tag item)
{
    var httpclient = factory.CreateClient("Api");
    var response = await httpclient
        .PutAsJsonAsync<Tag>("/api/Tags", item);
    return await response.Content.ReadFromJsonAsync<Tag>();
}
```

11. Let's not forget about our comments! Add the following code just under the code we just added:

```
public async Task<List<Comment>> GetCommentsAsync(string blogPostId)
{
    var httpclient = factory.CreateClient("Api");
    return await httpclient.GetFromJsonAsync
        <List<Comment>>($"/api/Comments/{blogPostId}") ?? [];
}

public async Task DeleteCommentAsync(string id)
{
    var httpclient = factory.CreateClient("Api");
    await httpclient.DeleteAsync($"/api/Comments/{id}");
}
public async Task<Comment?> SaveCommentAsync(Comment item)
{
    var httpclient = factory.CreateClient("Api");
    var response = await httpclient
        .PutAsJsonAsync<Comment>("/api/Comments", item);
    return await response.Content.ReadFromJsonAsync<Comment>();
}
```

Great job! Our API client is now done!

Summary

In this chapter, we learned how to create an API using Minimal APIs and an API client. Both of these are important parts of most applications. This way, we can get blog posts from our database and show them when we are running on WebAssembly. It is worth mentioning that we can always run our applications using a web API; this is just to show that we can use different ways to access our data depending on what hosting model we are currently using.

In the next chapter, we will add the login functionality to our sites and call our API for the first time.

10

Adding Authentication and Authorization

In this chapter, we will learn how to add **authentication** and **authorization** to our blog, because we don't want just anyone to be able to create or edit blog posts.

Covering authentication and authorization could take a whole book, so we will keep things simple here. This chapter aims to get the built-in authentication and authorization functionalities working, using the built-in ASP.NET functionality. That means that there is not a lot of Blazor magic involved here; many learning resources already exist that we can take advantage of.

Almost every system today has some way to log in, whether it is an admin interface (like ours) or a member login portal. There are many different login providers, such as Google, X, and Microsoft. We could use any of these providers since we will just be building on existing architecture.

Some sites might already have a database for storing login credentials, but for our blog, we will use a service called Auth0 to manage our users. It is a very powerful way to add many different social providers (if we want to), and we don't have to manage the users ourselves. Managing user credentials is not for the faint-hearted, and by not handling them ourselves, we also avoid the risk of leaking sensitive data.

We can also check the option to add authentication when creating our project. The authentication works differently for Blazor Server, Blazor WebAssembly, and the API, which we will look at in more detail in this chapter.

We will cover the following topics in this chapter:

- Setting up authentication

- Securing our Blazor app
- Securing Blazor WebAssembly
- Adding roles

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter09` folder as a starting point.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter10`.

Setting up authentication

There are many built-in features for authentication. The easiest way to add authentication is to select an authentication option when creating a project. However, we can also add authentication later by scaffolding it if we want to.

When using the Blazor Web App Template, the server handles all the authentication. The client doesn't have to know anything about the authentication provider.

Setting up Auth0

First, we need to set up Auth0.

Auth0 is a service that can help us with handling authentication for our users. There are many different services like this, but Auth0 is a good fit for us. We can connect one or more social connectors, which will allow our users to log in with Facebook, X, Twitch, or whatever we add to our site.

Even though all of this can be achieved by writing code ourselves, an integration like this is a great way to add authentication quickly, and it also provides a very powerful solution. Also, authentication is complex, so don't write this yourself unless you are sure what you are doing.

Auth0 is free for up to 7,000 users (which our blog probably won't reach, especially not the admin interface). If we want to add EntraID, for example, it has a different configuration but works in the same way.

Auth0 also offers robust functionality for adding data to our users that we have access to. We will do that later in the chapter when we add roles to our users.

To set up Auth0 for our application, follow these steps:

1. Head over to `https://auth0.com` and create an account.
2. Click the **Create Application** button.

3. Now, it's time to name our application. Use `MyBlog` for example.

   Then, it's time to select what kind of application type we are using. Is it a native app, a single-page web application, a regular web application, or a machine-to-machine application?

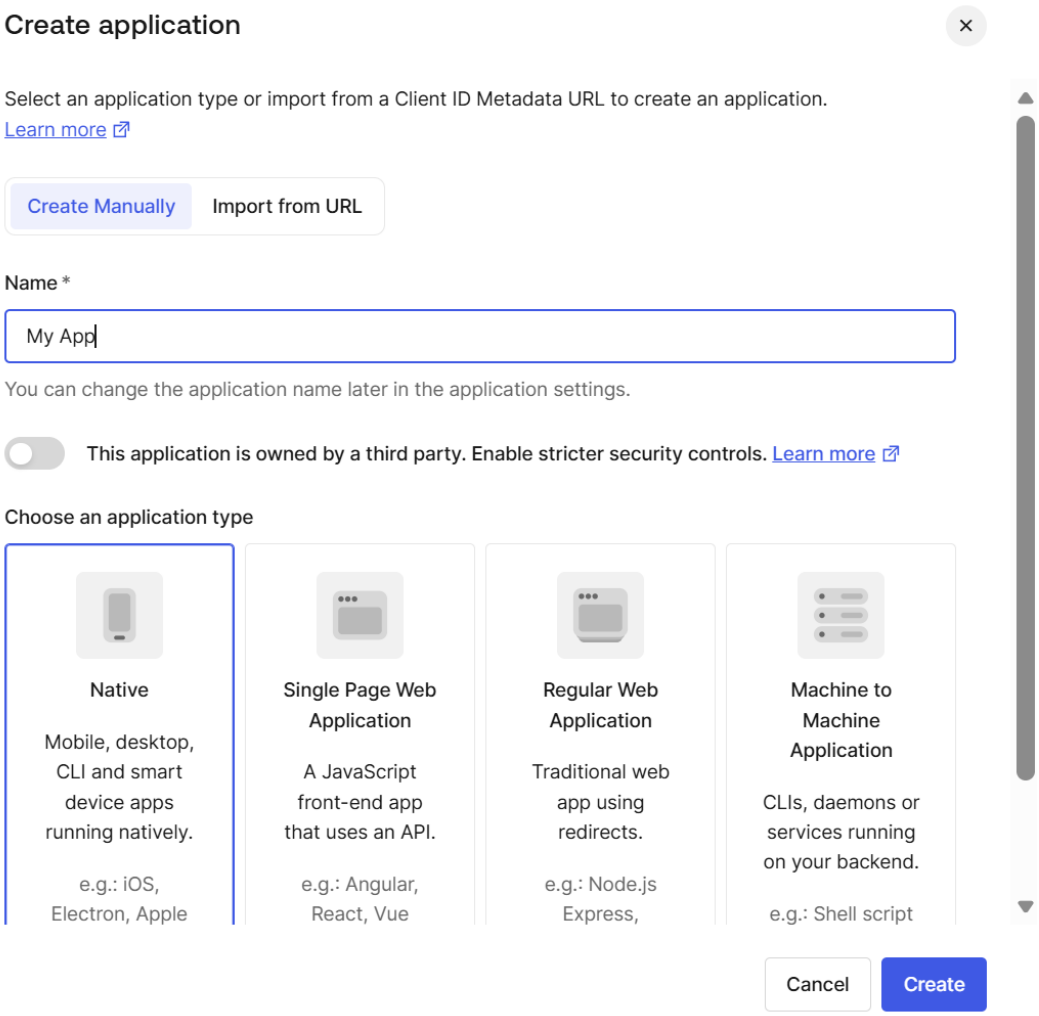


Figure 10.1: Auth0 application type selector

This depends on which hosting model we are going to run.

The beautiful thing with how we have the project set up right now is that the server is going to handle all the authentication and hand that over to WebAssembly (if we have

a component that is running in InteractiveAuto or InteractiveWebAssembly). But it won't limit the functionality, only what we need to configure when setting up our application.

If we intend to run only as Blazor Server (InteractiveServer), we should use a regular web application. But we might want to change to running everything in InteractiveWebAssembly, so let's not limit ourselves here.

Select **Single Page Application** – that way, we get the option to use our authentication in any hosting model.

4. Next, we will choose what technology we are using for our project. We have Apache, .NET, Django, Go, and many other choices, but we don't have a choice for Blazor specifically, at least not at the time of writing. Just skip this and click the **Setting** tab.
5. Now, we will set up our application. There are a couple of values we need to save and use later. You need to make sure that you write down the `Domain` and `Client ID`, as we will use those in a bit.
   If we scroll down, we can change the logo, but we will skip that.

   Leave `Application Login URI` empty

   The port numbers are random, and with Aspire, they can shift from one launch to another. By default, Aspire exposes our application on several ports. It uses `launchSettings.json` in addition to adding random ports.

   In the BlazorWebApp project, open `Properties/launchSettings.json`. Look for HTTPS and find the application URL. It should look like this: `https://blazorwebapp.dev.localhost:7119` or `https://localhost:7119` (the port number might be different, though). Make a note of them; we need them in the next step.
6. Next, in the `MyBlog.AppHost` project, open `AppHost.cs`. After `builder.AddProject<Projects.BlazorWebApp>("blazorwebapp")`, add the following:

```
    .WithEndpoint("https", endpoint =>
    {
        endpoint.Port = 7119;
        endpoint.IsProxied = false;
    })
    .WithEndpoint("http", endpoint =>
    {
        endpoint.Port = 5025;
```

```
            endpoint.IsProxied = false;
        })
```

Make sure to use the port numbers that you obtained in *step 5*.

This will make sure Aspire will expose our application on the same ports every time. Use the ports you have in your `launchSettings.json`.

7. Back in the Auth0 portal, make sure you add your application's port number:
   a. `Allowed callback URLs: https://localhost:PORTNUMBER/callback`
   b. `Allowed logout URLs: https://localhost:PORTNUMBER/`

   Allowed callback URLs are the URLs Auth0 will call after user authentication. **Allowed logout URLs** are where the user should be redirected after logout.

Now, press **Save** at the bottom of the page.

Configuring our Blazor app

We are done with configuring Auth0. Next, we will configure our Blazor app.

There are many ways to store secrets in .NET (a file that is not checked in, Azure Key Vault, etc.). You can use the one that you are most familiar with. We will keep it very simple and store secrets in our user secrets.

To configure our Blazor project, follow these steps:

1. In the `BlazorWebApp` project, right-click on the project, select **Manage User Secrets**, and add the following code to the root of the existing app settings object:

```
"Auth0": {
  "Authority": "Get this from the domain
       for your application at Auth0",
  "ClientId": "Get this from Auth0 setting"
}
```

   These are the values we made a note of in the previous section. Replace the values with your own values from `Auth0`.

   Since our site is an ASP.NET site with some added Blazor functionality, this means we can use a `NuGet` package to get some of the functionality out of the box.

2. In the `BlazorWebApp` project, add a reference to the `Auth0.AspNetCore.Authentication` NuGet package.

3. Open `Program.cs` and add the following code at the top of the file:

```
using BlazorWebApp;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
```

4. After `.AddInteractiveWebAssemblyComponents()`, add `.AddAuthenticationStateSerialization();` and it should now look like this:

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

This will automatically serialize the authentication state so that WebAssembly can pick it up.

5. Add the following code just before `var app = builder.Build();`:

```
builder.Services.AddCascadingAuthenticationState();
builder.Services
    .AddAuth0WebAppAuthentication(options =>
    {
        options.Domain =
            builder.Configuration["Auth0:Authority"]??"";
        options.ClientId =
            builder.Configuration["Auth0:ClientId"]??"";
    });
```

We added a call to `AddCascadingAuthenticationState`, which will make sure to always send an `AuthenticationState` object to all our components regardless of the hosting method, and configured Auth0 to get information from our `appsettings`.

The handling of authentication state was previously in the Routes file. It is worth noting that this has changed in Blazor development, so if you find any old examples, they might have cascading authentication state inside of Routes instead. This is the new way of using cascading authentication state.

6. Also, add the following code just after `app.UseAntiforgery();`. This code will allow us to secure our site:

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

7. Next, we will add a couple of small endpoints for login and logout. Since these are simple, we'll define them directly in `Program.cs`. We could have created extension methods for them, just as we did earlier with our blog APIs, but for these small endpoints, keeping them inline keeps things simpler. Add the following code just before `app.Run()`:

```
app.MapGet("account/login", async (
    string returnUrl, HttpContext context) =>
{
    var authenticationProperties =
        new LoginAuthenticationPropertiesBuilder()
         .WithRedirectUri(returnUrl)
         .Build();
    await context.ChallengeAsync(Auth0Constants
        .AuthenticationScheme, authenticationProperties);
});
```

When our site redirects to `authentication/login`, the Minimal API endpoint will initiate the login functionality.

8. We need to add a similar functionality for logout. Add the following code below the previous endpoint from *Step 7*:

```
app.MapGet("authentication/logout", async (HttpContext context) =>
{
    var authenticationProperties =
        new LogoutAuthenticationPropertiesBuilder()
         .WithRedirectUri("/")
         .Build();
    await context.SignOutAsync(Auth0Constants
        .AuthenticationScheme, authenticationProperties);
    await context.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);
});
```

The site needs to sign out twice: once for the Auth0 authentication scheme and once for the cookie authentication scheme. Auth0 handles the external login, while the cookie keeps the local session on our site. Both need to be cleared to fully log the user out.

With these endpoints in place, we now have login and logout functionality wired up through Auth0. Next, we'll update our Blazor application to make use of this and secure our pages.

Securing our Blazor app

Blazor uses `App.razor` for routing. To enable securing Blazor, we need to add a couple of components in the app component. We need to change the route view to `AuthorizeRouteView`, which can have different views depending on whether or not you are authenticated.

Let's get started:

1. In the `BlazorWebApp.Client` project, open `_Imports.razor` and add the namespace:

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

2. Open the `Routes.razor` component and replace everything inside the `Router` component with the following:

```
<Found Context="routeData">
    <AuthorizeRouteView RouteData="@routeData"
        DefaultLayout="@typeof(MainLayout)">
        <Authorizing>
            <p>Determining session state, please wait...</p>
        </Authorizing>
        <NotAuthorized>
            <h1>Sorry</h1>
            <p>You're not authorized to reach this page.
            You need to log in.</p>
        </NotAuthorized>
    </AuthorizeRouteView>
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
</Found>
```

In previous versions of Blazor, we had to surround the code with `<CascadingAuthenticationState>`, but with .NET 8 and beyond, that is handled automatically by adding the call to `AddCascadingAuthenticationState`.

Now, only two things remain: a page that we can secure and a display for the login link.

3. In the `Layout` folder, add a new Razor component called `LoginStatus.razor`. Replace the content with the following:

```
<AuthorizeView>
    <Authorized>
        <a href="authentication/logout">Log out</a>
    </Authorized>
    <NotAuthorized>
        <a href="account/login?returnUrl=/">Log in</a>
    </NotAuthorized>
</AuthorizeView>
```

`LoginStatus` is a component that shows a login link if we are not authenticated and a logout link if we are authenticated.

4. Open `Layout/MainLayout.razor` and replace the `about` link with the following:

```
<LoginStatus />
```

Now, our layout page will show whether we're logged in and allow us to log in or log out.

5. In the `BlazorWebApp.Client` project, in the `_Imports` file, add the following:

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

6. Add the `authorize` attribute to the component we wish to secure. To do this, find the following components:

```
Pages/Admin/BlogPostEdit.razor
Pages/Admin/BlogPostList.razor
Pages/Admin/CategoryList.razor
Pages/Admin/TagList.razor
```

In each of the preceding components, add the following attribute:

```
@attribute [Authorize]
```

This is all it takes: some configuration, and then we are all set.

7. Now, start our project and see if you can access the `/admin/blogposts` page. Log in (create a user) and see if you can access the page now.

Our admin interface is secured! In the next section, we will secure the Blazor WebAssembly version of our blog and the API.

Securing Blazor WebAssembly

In the previous editions of this book, we built two versions of the blog, one for the `Blazor server` and one for `Blazor WebAssembly`. In this edition, the whole point is that we don't have to choose one over the other.

As previously mentioned, there are two projects, `BlazorWebApp` and `BlazorWebApp.Client`. In the client project, we add all the components we want to be able to run as WebAssembly. Here is the really cool part. If we are running it as InteractiveAuto or InteractiveWebAssembly, it will first render on the server, using the configuration found in the `BlazorWebApp` project. But the next time the site runs, it will load the WebAssembly version and use the configuration in the `BlazorWebApp.Client` project.

So, the same component can use a different dependency injection. In one case, it will use direct data access, and in the other, it will use the API client we created in the previous chapter.

Configuring the client application

As mentioned, the server handles all the login stuff and serializes a state to the DOM. When WebAssembly takes over, it looks for that state and considers itself to be logged in. Let's add authentication to the server:

1. In the `BlazorWebApp.Client` project, in `Program.cs`, add the following lines just above `builder.Build().RunAsync();`:

```
builder.Services.AddAuthorizationCore();
builder.Services.AddCascadingAuthenticationState();
builder.Services.AddAuthenticationStateDeserialization();
builder.Services.AddHttpClient("Api", client => client.BaseAddress =
    new Uri(builder.HostEnvironment.BaseAddress));
```

This will enable Authentication, add cascading Authentication State to our components, and get the logged-in user from the serialized state. The name of the `HttpClient` is `Api`; this is the name we used in *Chapter 9*, *Creating an API*.

2. We also need to set up dependency injection so that when we ask for an `IBlogRepository`, we will get the `BlogApiWebClient` that we created in *Chapter 9*, *Creating an API*. In `Program.cs`, add the following code below the `AddAuthenticationStateDeserialization` line:

```
builder.Services.AddTransient<IBlogRepository, BlogApiWebClient>();
```

Make sure to add any missing using statements.

Now, when we ask for an `IBlogRepository` interface, we get the API web client that accesses the data through an API.

The really cool thing here is that, depending on whether the component is rendered on the server (Prerender, Static, or InteractiveServer), the client (InteractiveWebAssembly), or a combination (InteractiveAuto), it will choose the right client for that scenario. This gives the application the ability to choose the one best suited for the current render situation.

Running the application

Now, everything is in place for us to secure the WebAssembly part of the app.

This sample is about securing WebAssembly when running with an ASP.NET backend.

This is also the recommended way of handling authentication. It is possible to secure a Blazor WebAssembly Standalone site, but since the recommendations are to use a server backend, that is the only scenario covered in this book.

Let's give it a try. In the BlazorWebApp project, open the `Components/App.razor`.

It's time to change the render mode. Let's go all in on WebAssembly this time and switch the render mode to use it.

Change `<Routes @rendermode="InteractiveServer" />` to:

```
<Routes @rendermode="InteractiveWebAssembly" />
```

That's it! Now, our component will first render on the server (since we are running server prerender), and then WebAssembly will take over and render the component again. It will retrieve the authentication information from the component state and use our web API to fetch

the data. This is because the WebAssembly application is configured to use `BlogApiWebClient` when we ask for an instance of `IBlogRepository`. So, the same component is first prerendered on the server using direct data access, then again using the WebAPI. Pretty cool!

Now, run the project, navigate to `/Admin/Tags` and try to edit some tags.

But what if different users have different permissions?

That is where roles come in.

Adding roles

Sometimes, being logged in is not enough; we want only Admins to be able to do or see something. This is solved by adding roles.

Configuring Auth0 by adding roles

Let's start by adding roles in Auth0:

1. Log in to `Auth0`, navigate to **User Management** | **Roles**, and click **Create Role**.
2. Enter the name `Administrator`, the description `Can do anything`, and press **Create**.
3. Go to the **Users** tab, click **Add Users**, search for your user, and then click **Assign**. You can also manage roles from the **Users** menu on the left.
4. By default, roles won't be sent to the client, so we need to enrich the data to include roles. We do that by adding an action. Go to **Actions**, then **Triggers**.
5. Actions are a way to execute code in a particular flow. We want `Auth0` to add our roles when we log in. Click **Post-Login**, and we should now see the login trigger.
6. Click **Create Action**.
7. Name our new action **Add Roles**.
8. Set the trigger to be **Login/Post Login**.
9. Leave **Runtime** as is and press **Create**.
10. We will see a window where we can write our action. Replace all the code with the following:

```
/**
 * @param {Event} event - Details about the user and the context
   in which they are logging in.
 * @param {PostLoginAPI} api - Interface whose methods can be
   used to change the behavior of the login.
 */
```

```
exports.onExecutePostLogin = async (event, api) => {
  const claimName  =
      'http://schemas.microsoft.com/ws/2008/06/identity/claims/role'
  if (event.authorization) {
      api.idToken.setCustomClaim(claimName,
          event.authorization.roles);
      api.accessToken.setCustomClaim(claimName,
          event.authorization.roles);
  }
}
```

11. Click **Deploy** and then **Back to** Triggers.
12. Click **Custom** again, and we will see our newly created action.
13. Drag the **Add Roles** action to the arrow between **Start** and **Complete**.
14. Click **Apply**.

Now, we have an action that will add the roles to our login token, and our user is now an administrator.

> **Note**
>
> It's worth noting that roles are a paid feature in Auth0 and will only be free during the trial.

Now, let's set up Blazor to use this new role.

Adding roles to Blazor

Since we are using the Auth0 library, the setup is almost complete for Blazor.

Let's modify a component to show whether the user is an administrator.

In the `BlazorWebApp.Client` project, open `Layout/NavMenu.razor`. At the top of the component, add the following:

```
<AuthorizeView Roles="Administrator">
    <Authorized>
        Hi admin!
    </Authorized>
    <NotAuthorized>
        You are not an admin =(
```

```
    </NotAuthorized>
</AuthorizeView>
```

Now, run our project. If we log in (remember to use the correct port number), we should be able to see text to the left saying, **Hi Admin!**. However, the text is black on top of a dark blue background, so it might not be very visible.

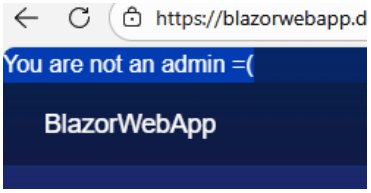


Figure 10.2: The not-so-visible message

We will take care of this in *Chapter 11*, *Sharing Code and Resources*.

Summary

In this chapter, we learned how to add authentication to our existing site. It is easier to add authentication when creating a project, but now we have a better understanding of what is going on under the hood and how to handle adding an external source for authentication.

This has been the most satisfying chapter to update because the .NET 9 and .NET 10 updates have targeted this process, making it more straightforward and requiring less code to use authentication.

Throughout the book, we have created shared components.

In the next chapter, we will look at sharing even more things, like static files and CSS, and try to make everything look nice.

11

Sharing Code and Resources

Throughout the book, we have been building a project that can run in many different hosting models. This is a great way to build our projects if we want to switch technologies further down the road or, as we did at my old job, share components between the customer portal and our internal **customer relationship management** (**CRM**) system.

Always consider whether a part of the component we are building could be made shareable. This allows us to reuse it, and any improvements made later will benefit all components that use it.

But it's not only about sharing components inside our own projects. What if we want to create a library that can be shared with other departments or even an open-source project sharing components with the world?

In this chapter, we will look at some of the things we already use when sharing components, and also at sharing CSS and other static files.

In this chapter, we will cover the following topics:

- Adding static files
- CSS isolation

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter10` folder as a starting point.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter11`.

If you are jumping into this chapter using the code from GitHub, make sure you have added `Auth0` account information in the settings files. You can find the instructions in *Chapter 10*, *Adding Authentication and Authorization*.

Adding static files

Blazor can use static files, such as images, CSS, and JavaScript. If we put our files in the `wwwroot` folder, they will automatically be exposed to the internet and be accessible from the root of our site. The nice thing about Blazor is that we can do the same with a library; it is super easy to distribute static files within a library.

At work, we share components between all our Blazor projects, and the shared library can also depend on other libraries. By sharing components and building our own components (sometimes on top of other libraries), we ensure we have the same look and feel throughout a site. We also share static content like images and CSS, and this makes it simple and fast if we need to change something and want all our sites to be affected.

To link to a resource in another library/assembly, we can use the `_content` folder.

Take a look at this example:

```
<link rel="stylesheet"
      href="_content/SharedComponents/MyBlogStyle.min.css" />
```

The HTML `link` tag, `rel`, and `href` are ordinary HTML tags and attributes, but adding the URL that starts with `_content` tells us that the content we want to access is in another library. The name of the library (assembly name), in our case, `SharedComponents`, is followed by the file we want to access, which is stored in the `wwwroot` folder in our library. Make sure the library is properly referenced in the project; otherwise, the `_content` path won't resolve and the static files won't be served.

Blazor is, in the end, just HTML, and HTML can be styled using CSS. As mentioned, the Blazor templates are using Bootstrap by default, and we will continue to use that as well.

There is an excellent site with easy-to-use Bootstrap themes ready to be downloaded, which can be found at `https://bootswatch.com/`.

I like the **Darkly** theme, so that's the one we'll use, but feel free to experiment with this later on.

Choosing between frameworks

I often get asked about how to style Blazor apps, and the truth is you can use all the tools you are familiar with. In the end, Blazor will output HTML. There are many languages and frameworks we can use to write our CSS.

We can use CSS, **Syntactically Awesome Stylesheets** (**SASS**), and **Leaner CSS** (**LESS**). As long as the output is CSS, we can use it.

In this chapter, we will stick with **Bootstrap** and continue using CSS. SASS and LESS are beyond the scope of this book.

Tailwind is a popular framework in the Blazor community, and it is absolutely possible to use it together with Blazor. Tailwind is very component-focused and needs a bit of configuration to start, but if it is something you have worked with and like, you can use it together with Blazor.

Adding a new style

Many templates use Bootstrap as a base, so if you are looking for a design for your website, using a Bootstrap-based template will be an easy implementation.

The problem with Bootstrap (and why some people don't like it) is that many sites use Bootstrap, and that makes "all sites look the same." This can be good if we are building a **line of business** (**LOB**), but it can be bad if we are trying to be innovative. Bootstrap is also quite large when it comes to downloading, so that is also an argument against it.

This chapter is about making our blog look a bit nicer, so we will stick with Bootstrap, but we should know that if we use something else to handle our CSS, it will work with Blazor.

One of these template sites is *Bootswatch*, which offers different themes built on top of Bootstrap. Let's add it:

1. Right-click on the **MyBlog** solution, select **Add** | **New Project...**.
2. Select **Razor Class Library**, click **Next**.
3. Name the project `SharedComponents`.
4. Select **.NET 10** and uncheck both **Support pages and views** and **Enlist in Aspire Orchestration**.
   Pages and views are related to Razor pages, not Blazor. This project is a shared library and will not run on its own using Aspire.
5. Now go to the new `SharedComponents` library. Let's clean it up a bit. Delete `Component1.razor`, `ExampleJsInterop.cs`, and all the files in the `wwwroot` folder.

6. Grab the `SharedComponents` project and drag it to the `BlazorWebApp.Client` project to reference our shared components.
   Now everything is set up for our shared library, let's add some stuff to it!
7. Navigate to `https://bootswatch.com/darkly/`.
8. In the top menu called **Darkly**, there are some links. Download `bootstrap.min.css`.
9. In the `SharedComponents` project, in the `wwwroot` folder, add the `bootstrap.min.css` file.
10. In the `BlazorWebApp` project, open `Program.cs` and replace the line:

    ```
    .AddAdditionalAssemblies(typeof(
        BlazorWebApp.Client._Imports).Assembly);
    ```

    With:

    ```
    .AddAdditionalAssemblies(
        typeof(BlazorWebApp.Client._Imports).Assembly,
        typeof(SharedComponents._Imports).Assembly);
    ```

    We are telling Blazor that the router should also look for components or routes in the `SharedComponents` assembly. This line sets up the server-side rendering so it can find components and their routes. But to make this complete, we also need to add the assembly to `Routes.razor`.
11. In `BlazorWebApp.Client` project, open `Routes.razor` and on the `Router` component, add the following:

    ```
    AdditionalAssemblies="[typeof(SharedComponents._Imports).Assembly]">
    ```

    (The existing parameters are removed for brevity).

    Now everything is set up for us to use any components in the `SharedComponents` library, and we will do that in the next chapter.

We have all the prerequisites and CSS that we can add to our site.

Adding CSS

Now, it's time to add a new style to our site:

1. In the `BlazorWebApp` project, open `Components/App.razor`.

2. Locate the following row:

```
<link rel="stylesheet"
      href="@Assets["lib/bootstrap/dist/css/bootstrap.min.css"]" />
```

   Replace it with:

```
<link rel="stylesheet"
      href="@Assets["_content/SharedComponents/bootstrap.min.css"]" />
```

   The `_content` path indicates that the asset comes from another assembly. The folder name that follows (`SharedComponents`) is the name of that assembly.

   At publish time, all required assets are automatically placed in the `wwwroot/_content` folder, so no additional configuration is required. This also gives us asset compression out of the box.

3. Run the project by pressing *Ctrl* + *F5*.

Great! Our Blazor project is now updated to use the new style. The main color should now be dark, but there is still some work to do.

The top and left menus are still the same, and we will get to those in just a little bit.

Making the admin interface more usable

Let's now clean up the interface further. We have only started with the admin functionality, so let's make it more accessible. The menu on the left is no longer required, so let's change it so that it is only visible if you are an administrator.

In the `BlazorWebApp.Client` project, open `/Layout/MainLayout.razor` and put `AuthorizeView` around the `sidebar div` like this:

```
<AuthorizeView Roles="Administrator">
    <div class="sidebar">
        <NavMenu />
    </div>
</AuthorizeView>
```

In this case, we are not specifying `Authorized` or `NotAuthorized`. The default behavior is `Authorized`, so if we are only looking for an authorized state, we don't need to specify it by name.

Start the project to see it in action. The menu should not be shown if we are not logged in.

Now, we need to make the menu look better. Even though the counter is really fun to click on, it doesn't make much sense for our blog.

Making the menu more useful

We should replace the links with links to our admin pages instead.

In the `BlazorWebApp.Client` project, open the `Layout/NavMenu.razor` file.

Edit the code so that it looks like this:

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

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

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

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Admin/Blogposts">
                <span class="bi bi-plus-square-fill-nav-menu"
                    aria-hidden="true"></span> Blog posts
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Admin/Tags">
                <span class="bi bi-plus-square-fill-nav-menu"
                    aria-hidden="true"></span> Tags
            </NavLink>
        </div>
```

```
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Admin/Categories">
                <span class="bi bi-plus-square-fill-nav-menu"
                    aria-hidden="true"></span> Categories
            </NavLink>
        </div>
    </nav>
</div>
```

If we start the project and resize the screen, we will notice that the menu shows up on large screens but is hidden on smaller ones. One thing worth noticing is that `NavMenu` doesn't contain code, so the menu hiding and showing up depending on screen size is all done by using CSS.

Great! Our blog is looking more like a blog, but we can do more!

Making the blog look like a blog

The admin interface is done (at least, for now), so we can focus on the front page of our blog. The front page should have the title of the blog post and some descriptions:

1. In the `BlazorWebApp.Client` project, open the `Pages/Home.razor` file.
2. Add a `using` statement for `Markdig` at the top of the file:

```
@using Markdig;
```

3. Add the code below to create the `OnInitializedAsync` method to handle the instantiation of the `Markdig` pipeline (this is the same code we have in the `Post.razor` file):

```
MarkdownPipeline pipeline;
protected override Task OnInitializedAsync()
{
    pipeline = new MarkdownPipelineBuilder()
                .UseEmojiAndSmiley()
                .Build();
    return base.OnInitializedAsync();
}
```

4. Inside the `Virtualize` component, change the content (`RenderFragment`) to the following:

```
<article>
    <h2>@p.Title</h2>
    @((MarkupString)Markdig.Markdown.ToHtml(
        new string(p.Text.Take(100).ToArray()), pipeline))
    <a href="/Post/@p.Id">Read more</a>
</article>
```

5. Also, remove the `<ul>` tags.

Now, run the project using *Ctrl + F5* and look at our new front page. Our blog is starting to take form, but we still have work to do.

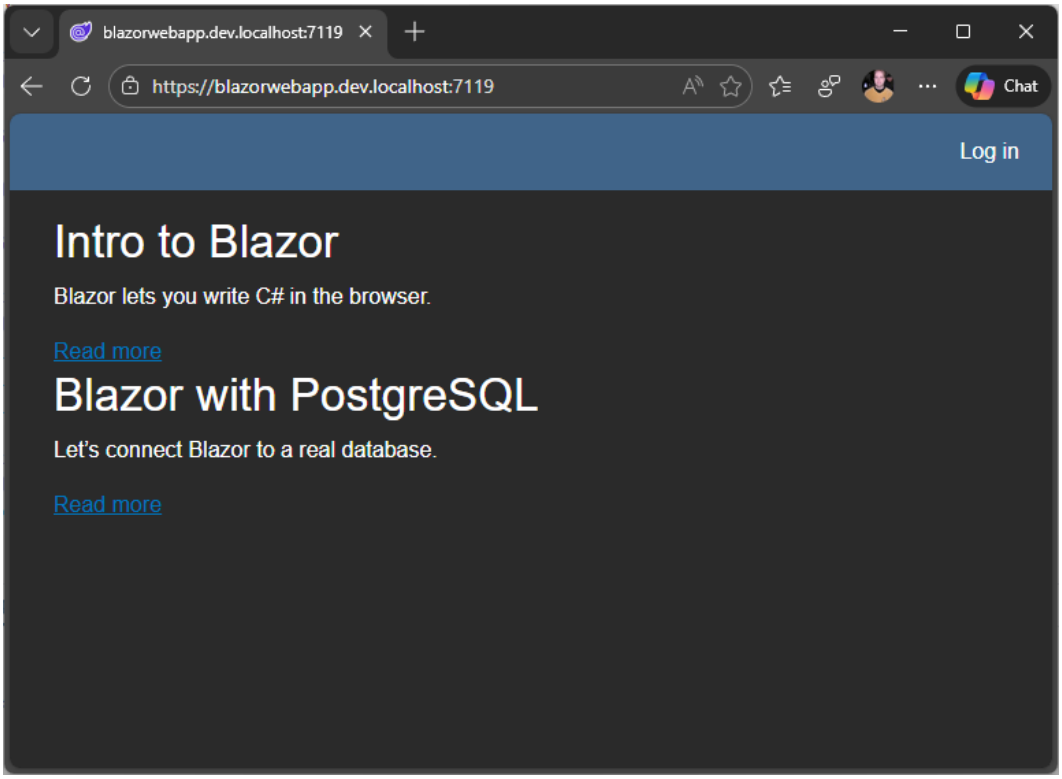


Figure 11.1: New shiny dark theme

CSS isolation

In .NET 5, Microsoft added something called isolated CSS. This is something that many other frameworks have as well. The idea is to write CSS specifically for one component. The upside, of course, is that the CSS that we create won't impact any of the other components.

The Blazor template already uses CSS isolation for `Layout/MainLayout.razor` and `NavMenu.razor`. If we expand `MainLayout.razor`, we will see a file called `MainLayout.razor.css`. The file appears nested because it follows the same naming convention.

We can also use SASS here by adding a file called `MainLayout.razor.scss`. The important thing is that the file we add should generate a file called `MainLayout.razor.css` for the compiler to pick up.

This naming convention will ensure that the CSS and the HTML output are rewritten.

CSS has the following naming convention:

```
main {
    flex: 1;
}
```

It will be rewritten as follows:

```
main[b-bfl5h5967n] {
    flex: 1;
}
```

This means the elements need to have an attribute called `b-bfl5h5967n` (in this case) for the style to be applied. This is a randomly generated string for this component.

The `div` tag that has the `CSS` tag within the `MainLayout` component will be output like this:

```
<main b-bfl5h5967n>
```

For all of this to happen, we also need to have a link to the CSS (which is provided by the template), and it looks like this:

```
<link rel="stylesheet" href="@Assets["ASSEMBLYNAME.styles.css"]" />
```

This becomes especially useful when working with component libraries.

We already have components with isolated CSS in our shared library (`NavMenu` and `MainLayout`). The CSS for a component, such as `NavMenu`, is included in the `{ASSEMBLYNAME}.styles.css` file automatically.

That means we don't have to do anything extra to include our shared CSS.

If we are building a component library for others, isolated CSS is a great option when our components require specific styling to work correctly. The styles are bundled with the component, so users don't have to reference any CSS manually. It also reduces the risk of our styles accidentally affecting the rest of the application.

If we are starting from an empty Blazor template, we do need to make sure to include a reference to the generated `.styles.css` file.

Isolated CSS works best when the styles are specific to a single component. It keeps everything close together, making it easier to find and maintain. We can also take advantage of things like CSS variables for colors and similar values.

That said, we should be a bit careful. If we start styling common elements like buttons in multiple isolated CSS files, we might end up with inconsistent styling across the app.

As mentioned, isolated CSS only affects the HTML elements inside the component. But what happens if we nest components?

If we open `Layout/NavMenu.razor.css`, we can see that for the `.nav-item` styles, some of them are using the `::deep` keyword. This indicates that even child components should be affected by this style.

Take a look at this code:

```
.nav-item ::deep a {…}
```

It is targeting the `<a>` tag, but the Razor code looks like this:

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="Admin/Blogposts">
        <span class="bi bi-plus-square-fill-nav-menu"
            aria-hidden="true"></span> Blog posts
     </NavLink>
</li>
```

It is the `NavLink` component that renders the `<a>` tag. By adding `::deep`, we are saying we want to apply this style to all elements with the `.nav-item` class and all the `<a>` tags inside that element.

There is one more thing we need to know about `::deep`. It makes sure to share the ID of the attribute (`b-bfl5h5967n`, for example), and it needs an HTML tag to do so. Therefore, if we have a component that consists only of other components and does not include any HTML tags, we need to wrap the content in an HTML tag for `::deep` to work.

So when you find yourself wondering why your isolated CSS is not working, it is because of one or more things:

- You are styling a component within a component and need the `::deep` keyword.
- You are styling a component that only contains other components. Since isolated CSS only applies to HTML elements, you need to wrap those components in an element (for example, a div) to apply styles.
- CSS Specificity. Honestly, it is probably the other two in 99% of the cases. But if you don't know CSS specificity, you should have a look at that as well.

Every time I talk to someone about Isolated CSS, I always bring up `::deep` and a surrounding HTML element. I really mean it... every time!

I even did a video on this specific topic, because it will trip you up at least once.

Now, fast forward a bit, I found myself trying to get an isolated CSS to work during one of our streams. 1.5h of me trying to get it to work. Spoilers: it was ::deep... it is always ::deep!

So it happens to me as well, for 1.5h!

Fixing the background

Before we summarize this chapter, let us do one more thing.

Let's fix the background color of the menu:

1. Open `/Layout/MainLayout.razor.css`.
2. Look for the `.sidebar` style and replace it with the following code:

```
.sidebar {
    background-image: linear-gradient(180deg,
        var(--bs-body-bg) 0%,var(--bs-gray-800) 70%);
}
```

3. Replace the `.top-row` style with the following code:

```
.top-row {
    background-color: var(--bs-primary);
    justify-content: flex-end;
    height: 3.5rem;
    display: flex;
    align-items: center;
}
```

   We replaced the background color and removed a border.

4. In the `.top-row ::deep a, .top-row ::deep .btn-link` style, add the following:

```
color:white;
```

Now, we are able to see the log-in/logout link a bit better. We now have a working admin interface and a good-looking site. It looks something like this:

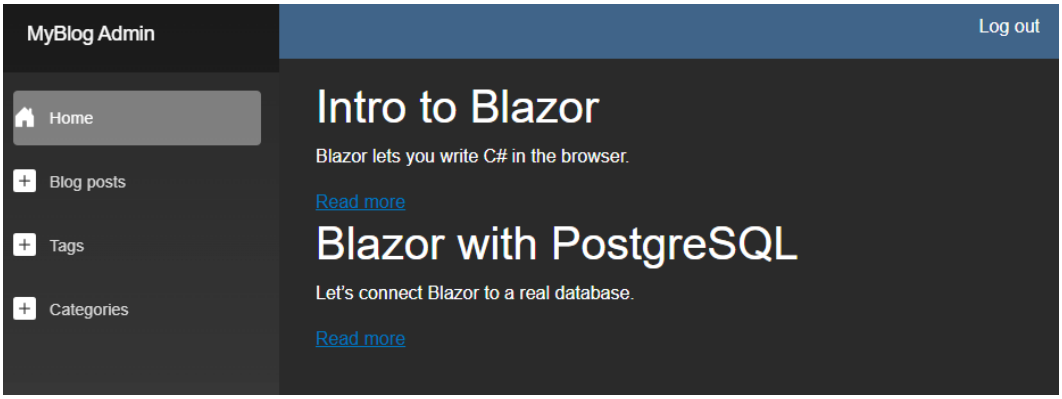


Figure 11.2: New blog style

That brings us to the end of this chapter.

Summary

In this chapter, we have added a shared CSS.

Using shared libraries like this is the way to create shared libraries for others to use, and it is also a great way to structure our in-house teams for sharing styles or components between different projects. If you have a site already, you can build your Blazor components in a shared library.

We talked about how we can use SASS and CSS in our site, both *regular* CSS and *isolated* CSS.

In the next chapter, we will learn about the one thing we are trying to avoid (at least, I am) as Blazor developers: JavaScript.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

12

JavaScript Interop

In this chapter, we will take a look at JavaScript. In specific scenarios, we still need to use JavaScript, or we want to use an existing library that relies on JavaScript. Blazor uses JavaScript to update the **Document Object Model** (**DOM**), download files, and access local storage on the client.

So, there are, and always will be, cases when we need to communicate with JavaScript or have JavaScript communicate with us. Don't worry, the Blazor community is an amazing one, so chances are someone has already built the interop we need.

In this chapter, we will cover the following topics:

- Why do we need JavaScript?
- .NET to JavaScript
- JavaScript to .NET
- Implementing an existing JavaScript library
- JavaScript interop in WebAssembly

Technical requirements

Ensure you have followed the previous chapters or use the `Chapter11` folder as a starting point.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter12`.

If you are jumping into this chapter using the code from GitHub, make sure you have added the `Auth0` account information in the settings files. You can find the instructions in *Chapter 10*, *Adding Authentication and Authorization*.

Why do we need JavaScript?

Many say Blazor is the JavaScript killer, but the truth is that Blazor needs JavaScript to work. Some events only get triggered in JavaScript, and if we want to use those events, we need to make an interop.

I jokingly say that I have never written so much JavaScript as when I started developing with Blazor. Calm down... it's not that bad.

I have written a couple of libraries that require JavaScript to work. They are called `Blazm.Components` and `Blazm.Bluetooth`.

`Blazm.Components` is a grid component that uses JavaScript interop to trigger C# code (JavaScript to .NET) when the window is resized and to remove columns if they can't fit inside the window.

When that is triggered, the C# code calls JavaScript to get the size of the columns based on the client width, which only the web browser knows, and based on that answer, it removes columns if needed.

`Blazm.Bluetooth` makes it possible to interact with Bluetooth devices using Web Bluetooth, which is a web standard accessible through JavaScript.

`Blazm.Bluetooth` uses two-way communication. Bluetooth events can trigger C# code, and C# code can iterate over devices and send data to them. They are both open source, so if you are interested in looking at a real-world project, you can check them out on my GitHub: `https://github.com/EngstromJimmy`.

As mentioned earlier, in most cases, I would argue that we won't need to write JavaScript ourselves. The Blazor community is large, so chances are that someone has already written what we need. But we don't need to be afraid of using JavaScript, either. Next, we will look at different ways to add JavaScript calls to our Blazor project.

.NET to JavaScript

Calling JavaScript from .NET is pretty simple. There are two ways of doing this:

- Global JavaScript
- Collocated JavaScript

We will go through both ways to see what the difference is.

Global JavaScript (the old way)

To access a JavaScript method, we need to make it accessible. One way is to define it globally through the JavaScript window object. This is a bad practice since it is accessible by all scripts and could replace the functionality in other scripts if we accidentally use the same names.

What we can do, for example, is use scopes, create an object in global space, and put our variables and methods on that object so that we lower the risk a bit, at least.

Using a scope could look like this:

```
<script>
    window.myscope = {};
    window.myscope.methodName = () => { alert("this has been called"); }
</script>
```

We create an object with the name `myscope`. Then, we declare a method on that object called `methodName`. In this example, there is no code in the method; this only demonstrates how it could be done.

Then, to call the method from C#, we would call it using `JSRuntime` like this:

```
@using Microsoft.JSInterop
@inject IJSRuntime jsRuntime
await jsRuntime.InvokeVoidAsync("myscope.methodName");
```

There are two different methods we can use to call JavaScript:

- `InvokeVoidAsync`, which calls JavaScript, but doesn't expect a return value
- `InvokeAsync<T>`, which calls JavaScript and expects a return value of type `T`

We can also send in parameters to our JavaScript method if we want. We also need to refer to the JavaScript file somewhere on the page (usually the `head` tag), which must be stored in the `wwwroot` folder.

The other way is **Collocated JavaScript**, which uses the methods described here but with modules.

Collocated JavaScript

In .NET 5, we got a new way to add JavaScript using Collocated JavaScript, which is a much nicer way to call JavaScript. It doesn't use global methods. This is awesome for component vendors and end users because JavaScript will be loaded when needed. It will only be loaded

once (Blazor handles that for us), and we don't need to add a reference to the JavaScript file, which makes it easier to start and use a library.

So, let's implement that instead.

Collocated JavaScript can be stored in the `wwwroot` folder, but since .NET 6, we can add it the same way we add isolated CSS. Add them to your component's folder and name it `js` at the end (`mycomponent.razor.js`).

Let's do just that!

In our project, we can delete categories and components. Let's implement a simple JavaScript call to reveal a prompt to make sure that the user wants to delete the category or tag. But we have talked about doing things in a reusable way, so let's do that:

1. In the `BlazorWebApp.Client` project, in the `ReusableComponents` folder, move all the files that are `.razor` from the client project into the root of the `SharedComponents` project.
   By the end, you should have three `.cs` files in `BlazorWebApp.Client/ReusableComponents` and five (or six, counting `_Imports.razor`) in the `SharedComponents` library.

   We could have moved them as well, but then we also need to change the namespaces, so let's keep them there.

   Great, we are on the way to making these components truly reusable.

2. In the `SharedComponents` project, open `_Imports.razor` and add:

   ```
   @using Microsoft.AspNetCore.Components.Forms
   @using Microsoft.AspNetCore.Components.Web.Virtualization
   ```

3. In `BlazorWebApp.Client`, open `_Imports.razor` and add:

   ```
   @using SharedComponents
   ```

   Now our client project knows about our shared projects namespace, and we can start using the components.

4. In the `SharedComponents` project, select the `BlogButton.razor` file, create a new JavaScript file, and name the file `BlogButton.razor.js`.

5. Open the new file (it is located under `BlogButton.razor` in the solution explorer) and add the following code:

```
export function showConfirm(message) {
    return confirm(message);
}
```

Collocated JavaScript uses the standard **EcmaScript** (**ES**) modules and can be loaded on demand. The methods it exposes are accessible only through that object, not globally, unlike the *old* way. It is possible to use TypeScript here as well, as long as the end result is a JavaScript file.

6. Open `BlogButton.razor` and inject `IJSRuntime` at the top of the file:

```
@using Microsoft.JSInterop
@inject IJSRuntime jsRuntime
```

7. In the code section, let's add a method that will call JavaScript:

```
    IJSObjectReference jsmodule;
    [Parameter]
    public string? ConfirmMessage { get; set; } = null;
    private async Task<bool> ShouldExecute()
    {
        if (ConfirmMessage != null)
        {
            jsmodule = await
                jsRuntime.InvokeAsync<IJSObjectReference>(
                "import",
                "/_content/SharedComponents/BlogButton.razor.js");
            return await jsmodule.InvokeAsync<bool>(
                "showConfirm", ConfirmMessage);
        }
        else
        {
            return true;
        }
    }
```

`IJSObjectReference` is a reference to the specific script we will import later. It has access to the exported methods in our JavaScript and nothing else.

We run the `Import` command and send the filename as a parameter. This will run the JavaScript command `let mymodule = import("/_content/SharedComponents/BlogButton.razor.js")` and return the module. We also add a `ConfirmMessage` parameter so we know that if we have a `ConfirmMessage`, we should show a confirm message.

8. Then, in our `OnButtonClick` method, we first check whether we should execute the method or not. Change it to the following:

```
if (OnClick.HasDelegate && await ShouldExecute())
 {
     await OnClick.InvokeAsync(args);
 }
```

Now, we can use our button to confirm whether we want to delete the `Category` or `Tag`.

9. Open `ItemList.razor`, and let's add our `BlogButton` component to the `ItemList` component. Inside the Virtualize component, change the delete button to the following:

```
<BlogButton
    ConfirmMessage="Are you sure you want to delete this item?"
    Type="BlogButton.ButtonType.Delete"
    OnClick="@(()=> {DeleteEvent.InvokeAsync(item);})">
    Delete</BlogButton>
```

Instead of just calling our `Delete` event callback, we first call our new method. Let JavaScript confirm that you really want to delete it, and if so, then run the `Delete` event callback.

This is a simple JavaScript implementation. Notice how we implement the logic in the button and use it in our shared `ItemList` component, which will ultimately be used in both `CategoryList` and `TagList`. This is the power of using components. And we also put all of this in a shared library, leveraging Collocated JavaScript to get JavaScript working without manually referencing any JavaScript files.

This is calling JavaScript from .NET code. What about the other way around?

JavaScript to .NET

I would argue that calling .NET code from JavaScript isn't a very common scenario, and if we find ourselves in that scenario, we might want to think about what we are doing.

As Blazor developers, we should avoid using JavaScript as much as possible.

I am not bashing JavaScript in any way, but I see this often happens where developers use what they used before and kind of shoehorn it into their Blazor project.

They end up solving things with JavaScript that could have been a simple if statement in Blazor.

This often happens because that was the only option before. In traditional web development, if we wanted to show or hide something, react to events, or update the UI, JavaScript was the tool we had. So it's easy to fall back into that habit.

In Blazor, we already have that logic in C#. We can use conditions, event handlers, and binding directly in our components instead of reaching for JavaScript.

So, it's important to think about when JavaScript is actually needed and when Blazor already gives us a simpler solution.

There are three ways of doing a callback from JavaScript to .NET code:

- A static .NET method call
- An instance method call
- A component instance method call

Let's take a closer look at them.

These samples can be found in the GitHub repo in the `SharedComponents/JavaScriptDemo/JSInteropSamples` folder. Please note that we are currently running our app in InteractiveWebAssembly and in debug mode, so if nothing happens when you click a button, just wait a bit for WebAssembly to fully load.

Static .NET method call

To call a .NET function from JavaScript, we can make the function static, and we also need to add the `JSInvokable` attribute to the method.

We can add a function such as this in the `code` section of a Razor component or inside a class:

```
[JSInvokable("ReturnArrayAsync")]
public static Task<int[]> ReturnArrayAsync()
{
```

```
    return Task.FromResult(new int[] { 1, 2, 3 });
}
```

In the JavaScript file, we can call that function using the following code:

```
DotNet.invokeMethodAsync('SharedComponents', 'ReturnArrayAsync')
      .then(data => {
        data.push(4);
        alert(data);
      });
```

The `DotNet` object comes from the `Blazor.js` or `blazor.server.js` file.

`SharedComponents` is the name of the assembly, and `ReturnArrayAsync` is the name of the static .NET function.

It is also possible to specify the name of the function in the `JSInvokable` attribute if we don't want it to be the same as the method name, like this:

```
[JSInvokable("MadeUpName")]
```

In this sample, JavaScript calls back to .NET code, which returns an `int` array.

It is returned as a promise in the JavaScript file that we are waiting for, and then (using the `then` operator) we continue with the execution, adding a `4` to the array, and then outputting the values in a popup.

Instance method call

This method is a bit tricky; we need to pass an instance of the .NET object to call it (this is the method that `Blazm.Bluetooth` is using).

Please note that this is not a full step-by-step tutorial. Please take a look at the repo for a full example.

Let's start with the component so we can see where the data comes from:

```
<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>
```

Here, we bind the value of the input to a field called `name`. This means the value is stored in the component instance.

We also have a method on the component that returns a message:

```
private DotNetObjectReference<JSToReferenceNET>? dotNetHelper;

public async Task TriggerDotNetInstanceMethod()
{
    dotNetHelper = DotNetObjectReference.Create(this);
    result = await JS.InvokeAsync<string>(
        "callreferencenetfromjs", dotNetHelper);
}

[JSInvokable]
public string GetHelloMessage() => $"Hello, {name}!";
```

Notice that we are not passing the name around as a parameter. Instead, the method reads the value directly from the component's state.

Now, to allow JavaScript to call this method, we need to pass a reference to the component instance. That's where `DotNetObjectReference<T>` comes in.

This gives JavaScript access to the current component instance.

Next, we need some JavaScript that can call back into .NET:

```
export function callreferencenetfromjs(dotnetHelper) {
    return dotnetHelper.invokeMethodAsync(
        'GetHelloMessage').then(r => alert(r));
}
```

This is the full sample:

```
@page "/jstoreferencenet"
@implements IDisposable
@inject IJSRuntime jsRuntime

<h3>This is a demo how to call .NET from JavaScript using a .NET reference</h3>

<p>
    <label>
        Name: <input @bind="name" />
```

```
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
    private string? name;
    private string? result;
    private DotNetObjectReference<JSToReferenceNET>? dotNetHelper;
    IJSObjectReference jsmodule;
    public async Task TriggerDotNetInstanceMethod()
    {
        dotNetHelper = DotNetObjectReference.Create(this);
        jsmodule = await
            jsRuntime.InvokeAsync<IJSObjectReference>(
            "import",
            "/_content/SharedComponents/JavaScriptDemo/
                JSInteropSamples/JSToReferenceNET.razor.js");
        await jsmodule.InvokeAsync<string>(
            "callreferencenetfromjs", dotNetHelper);
    }

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {name}!";

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

Let's go through the class. We inject `IJSRuntime` because we need one to call the JavaScript function. To avoid any memory leaks, we also have to make sure to implement `IDisposable`,

and toward the bottom of the file, we make sure to dispose of the `DotNetObjectReference` instance.

We create a private `DotNetObjectReference<JSToReferenceNET>` to hold a reference to the current component instance, which we pass to JavaScript.

We also use an `IJSObjectReference` to load and call our JavaScript module.

Then, we create an instance of type `DotNetObjectReference` using `DotNetObjectReference.Create(this)`.

Now, we have this `objRef`, which we will send to the JavaScript method. First, we load the JavaScript module, then call `JavaScriptMethod`, and pass in the reference to our `JSToReferenceNET` component instance. Now, the JavaScript `callreferencenetfromjs` method will run `dotnetHelper.invokeMethodAsync("GetHelloMessage")`, which will make a call to `GetHelloMessage` and get back a string with `"Hello, "` followed by the name in the box.

This is a simple example, but it clearly shows the mechanics. The goal here isn't the scenario itself, but to understand how JavaScript can call back into a specific .NET instance.

Basically, what is happening is:

1. We call JavaScript and pass a reference to the current .NET component instance.
2. JavaScript calls back into that instance.
3. The .NET method runs and returns a value, which JavaScript then uses.

In a real-world scenario, this pattern is useful when JavaScript needs to notify .NET about something happening in the browser, for example, events, external APIs, or device input.

Since calling .NET from JavaScript is rare, we won't go into more examples. Instead, we'll dive into considerations when implementing an existing JavaScript library.

Implementing an existing JavaScript library

The best approach, in my opinion, is to avoid porting JavaScript libraries. Blazor needs to keep the DOM and the render tree in sync, and having JavaScript manipulate the DOM can jeopardize that.

Most component vendors, such as Telerik, Syncfusion, Radzen, and, of course, Blazm, have native components. They don't just wrap JavaScript; they're explicitly written for Blazor in C#. Even though the components use JavaScript to some extent, the goal is to keep it to a minimum.

So, if you are a library maintainer, my recommendation would be to write a native Blazor version of the library, keep JavaScript to a minimum, and, most importantly, not force Blazor developers to write JavaScript to use your components.

Some components will be unable to use JavaScript implementations because they need to manipulate the DOM.

Blazor is pretty smart when syncing the DOM and render tree, but try to avoid manipulating the DOM. If we need to use JavaScript for something, make sure to put an HTML tag outside the manipulation area, and Blazor will then keep track of that tag and not think about what is inside the tag.

Since we started with Blazor at my workplace very early, many vendors had not yet completed their Blazor components. We needed a graph component fast. On our previous website (before Blazor), we used a component called **Highcharts**.

For this demo, we will do the same thing we did back in the day, but use a library called `chart.js`. Let's walk through what we (the team at my former workplace) did (please note this is not a full tutorial).

First, the app needs a reference to the `Chart.js` JavaScript:

```
<script
    src="https://cdnjs.cloudflare.com/ajax/libs
        /Chart.js/4.5.0/chart.umd.min.js">
</script>
```

Depending on your needs, you might want to make a copy of the JavaScript file to avoid external dependencies.

The `ChartJs` Razor component looks like this:

```
@inject IJSRuntime JS

<canvas @ref="_canvas"
        width="@Width"
        height="@Height"
        style="max-width:@($"{Width}px"); max-height:@($"{Height}px")">
</canvas>

@code {
    [Parameter, EditorRequired]
    public object Config { get; set; } = default!;
```

```
    [Parameter] public int Width { get; set; } = 300;
    [Parameter] public int Height { get; set; } = 300;

    private ElementReference _canvas;
    private IJSObjectReference? _chart;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender)
            return;

        _chart = await JS.InvokeConstructorAsync("Chart",_canvas,Config);
    }

    public IJSObjectReference? Chart => _chart;

    public async ValueTask DisposeAsync()
    {
        if (_chart is not null)
            await _chart.InvokeVoidAsync("destroy");
    }
}
```

We run our JavaScript interop in the `OnAfterRenderAsync` method because, otherwise, it would throw an exception, as you may recall from *Chapter 6*, *Understanding Basic Blazor Components*.

We are creating the JavaScript object like this:

```
_chart = await JS.InvokeConstructorAsync("Chart",_canvas, Config);
```

We are doing everything from within C#. You don't need JavaScript files for this.

Now, the only thing left to do is to use the component, and that looks like this:

```
@page "/pacman"
<h3>Pac-Man Chart</h3>

<button @onclick="ToggleMouth">
    @(isMouthOpen ? "Close mouth" : "Open mouth")
</button>
```

```
<ChartJs @ref="_chart"
         Config="@config" />

@code {
    private ChartJs? _chart;

    private bool isMouthOpen = true;
    private double value = 65;

    private object config = default!;

    protected override void OnInitialized()
    {
        config = CreatePacmanConfig();
    }

    private object CreatePacmanConfig()
    {
        return new
        {
            type = "pie",
            data = new
            {
                datasets = new[]
                {
                new
                {
                    data = new[] { value },
                    backgroundColor = new[]
                    {
                        "#FFD93D", // Pac-Man
                    },
                    borderWidth = 0
                }
            }
            },
            options = new
            {
                rotation = 120,
                circumference = 270, // mouth open
```

```
                plugins = new
                {
                    legend = new { display = false },
                    tooltip = new { enabled = false }
                },
                animation = new
                {
                    animateRotate = true,
                    duration = 300
                }
            }
        };
    }


    private async Task ToggleMouth()
    {
        if (_chart?.Chart is null)
            return;

        isMouthOpen = !isMouthOpen;

        var circumference = isMouthOpen ? 270 : 360;

        await _chart.Chart.SetValueAsync(
            "options.circumference",
            circumference);

        await _chart.Chart.InvokeVoidAsync("update");
    }
}
```

The really amazing thing here is that we have a working JavaScript interop, and we had to write 3 lines of JavaScript. With .NET 10, we got the ability to call methods and set properties on JavaScript objects:

```
await _chart.Chart.InvokeVoidAsync("update");
```

This line will call the update method on the chart instance. This was previously done by adding a JavaScript method and then calling that from Blazor. This opens up so many doors.

In .NET 10, we got these methods that we can call on a JavaScript object:

- `SetValueAsync`
- `GetValueAsync`
- `InvokeAsync<T>`
- `InvokeVoidAsync`

  And also, this one that can create JavaScript objects:

- `InvokeConstructorAsync`

With .NET 10, Blazor can interact with JavaScript objects by setting and reading properties and calling methods. This enables a more object-oriented style of JavaScript interop, where JavaScript objects are created once and then managed from .NET.

Previously, we had to keep these objects in JavaScript and create methods to access what we needed.

This test code will show a pie chart that looks like *Figure 12.1*:

Figure 12.1: Chart example

We have now gone through how we got a JavaScript library to work with Blazor, so this is an option if there is something we need.

As mentioned, the component vendors are investing in Blazor, so chances are that they have what we need, so we might not need to invest time in creating our own component library.

All the things mentioned so far in this chapter will work great for Blazor Server and Blazor WebAssembly, but WebAssembly can make JavaScript interop even more powerful (in terms of speed) and lead to an easier syntax.

JavaScript interop in WebAssembly

With Blazor WebAssembly, we have direct access to the `JSRuntime` (since all the code is running inside the browser). Direct access will give us a really big performance boost. For most applications, we are doing one or two JavaScript calls. Performance is not really going to be a problem. Some applications are more JavaScript-heavy, though, and would benefit from using the `JSRuntime` directly.

We have had direct access to `JSRuntime` using `IJSInProcessRuntime` and `IJSUnmarshalledRuntime`. But after .NET 7, both became obsolete, and we have gotten a nicer syntax.

In the GitHub repository, I have added a project called `BlazorWebAssemblyStandaloneApp`, with a couple of files if you want to try the code.

We will start by looking at calling JavaScript from .NET. These samples must run in a WebAssembly-only project. They are included in the source on GitHub for reference.

To be able to use these features, we need to enable them in the project file by enabling `AllowUnsafeBlocks`:

```
<PropertyGroup>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

.NET to JavaScript

To show the difference between calling JavaScript from the more generic version that works with Blazor Server and WebAssembly, compared to the one that only works with WebAssembly, the following sample uses the same `ShowAlert` function as earlier in the chapter.

The Razor file looks like this:

```
@page "/nettojswasm"
@using System.Runtime.InteropServices.JavaScript
<h3>This is a demo how to call JavaScript from .NET</h3>
<button @onclick="ShowAlert">Show Alert</button>
@code {
```

```
    protected async void ShowAlert()
    {
        ShowAlert("Hello from .NET");
    }
    protected override async Task OnInitializedAsync()
    {
        await JSHost.ImportAsync("nettojs",
            "../JSInteropSamplesWasm/NetToJS.razor.js");
    }
}
```

We are using `JSHost` to import the JavaScript and give it the name `"nettojs"`. A source generator generates the implementation for calling JavaScript, and to be sure that it can pick up what it should do, we need to add some code in a code-behind. We will go more in depth on source generators in *Chapter 20*, *Examining Source Generators*. The code-behind looks like this:

```
using System.Runtime.InteropServices.JavaScript;
namespace BlazorWebAssemblyStandaloneApp.JSInteropSamplesWasm;
public partial class NetToJS
{
    [JSImport("showAlert", "nettojs")]
    internal static partial string ShowAlert(string message);
}
```

The JavaScript file looks like this:

```
export function showAlert(message) {
    return alert(message);
}
```

We add a `JSImport` attribute to a method, which will automatically be mapped to the JavaScript call.

This is a much nicer implementation, I think, and a lot faster.

Next, we will look at calling .NET from JavaScript.

JavaScript to .NET

When calling a .NET method from JavaScript, a new attribute makes that possible, called `JSExport`.

The Razor file implementation looks like this:

```
@page "/jstostaticnetwasm"
@using System.Runtime.InteropServices.JavaScript
<h3>This is a demo how to call .NET from JavaScript</h3>

<button @onclick="ShowMessage">Show alert with message</button>

@code
{
    protected override async Task OnInitializedAsync()
    {
        await JSHost.ImportAsync("jstonet",
            "../JSInteropSamplesWasm/JSToStaticNET.razor.js");
    }
}
```

Calling `JSHost.ImportAsync` is not necessary for the `JSExport` part of the demo, but we need it to call JavaScript so that we can make the .NET call from JavaScript.

Similarly, here we need to have the methods in a code-behind class that looks like this:

```
using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;
namespace BlazorWebAssemblyStandaloneApp.JSInteropSamplesWasm;

[SupportedOSPlatform("browser")]
public partial class JSToStaticNET
{
    [JSExport]
    internal static string GetAMessageFromNET()
    {
        return "This is a message from .NET";
    }

    [JSImport("showMessage", "jstonet")]
    internal static partial void ShowMessage();
}
```

Here, we are using the `SupportedOSPlatform` attribute to ensure that this code can only run on a browser.

The JavaScript portion of this demo looks like this:

```
export async function setMessage() {
    const { getAssemblyExports } = await globalThis.getDotnetRuntime(0);
    var exports = await getAssemblyExports(
        "BlazorWebAssemblyStandaloneApp.dll");
    alert(exports.BlazorWebAssemblyStandaloneApp.
        JSInteropSamplesWasm.JSToStaticNET.GetAMessageFromNET());
}

export async function showMessage() {
    await setMessage();
}
```

We call the `showMessage` JavaScript function from .NET, and it will then call the `setMessage` function.

The `setMessage` function uses the `globalThis` object to access the .NET runtime and get access to the `getAssemblyExports` method.

It will retrieve all the exports for our assembly and then run the method. The .NET method will return the `"This is a message from .NET"` string and show the string in an alert box.

Even though I prefer not to make any JavaScript calls in my Blazor applications, I love having the power to bridge between .NET code and JavaScript code with ease.

Summary

This chapter taught us about calling JavaScript from .NET and calling .NET from JavaScript. In most cases, we won't need to do JavaScript calls, and chances are that the Blazor community or component vendors have solved the problem for us.

We also looked at how we can port an existing library if needed.

In the next chapter, we will continue to look at state management.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/8VuShAAtK`

13

Managing State – Part 2

In this chapter, we continue to look at managing state.

Most applications manage state in some form. State is simply information that is persisted in some way. It can be data stored in a database, session state, or even something stored in a URL.

The user state is stored in memory, either in the web browser or on the server. It contains the component hierarchy and the most recently rendered UI (render tree). It also contains the values or fields and properties in the component instances, as well as the data stored in service instances during dependency injection.

If we make JavaScript calls, the values we set are also stored in memory. Blazor Server relies on the circuit (SignalR connection) to hold the user state, and Blazor WebAssembly relies on the browser's memory.

But when we have a mix of both states, state management becomes a bit trickier. If we reload the page, the circuit and the memory will be lost. The same goes for switching pages; if there are no more InteractiveServer components on the page, the SignalR connection will be terminated and the state lost.

Managing state is not just about handling connections, but about keeping data even if we reload the browser. Saving state between page navigations or sessions improves the user experience and could be the difference between a sale and no sale. Imagine reloading the page, and all your items in the shopping cart are gone; the chances are you won't shop there again. Now imagine returning to a page a week or a month later, and all those things are still there, you would be more likely to continue with your purchase.

In this chapter, we will cover the following topics:

- Persistent component state
- Storing data on the server side

- Storing data in the URL
- Implementing browser storage
- Using an in-memory state container service
- State management frameworks
- Root-level cascading values

We have already talked about and even implemented some of these things, so let's take this opportunity to recap the things we have already talked about, as well as introduce some new techniques.

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter12` folder as a starting point.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter13`.

If you are jumping into this chapter using the code from GitHub, make sure you have added `Auth0` account information in the settings files. You can find the instructions in *Chapter 10*, *Adding Authentication and Authorization*, and also rerun the database scripts to populate the database.

Persistent component state

You might have noticed that when Blazor loads our list of blog posts, it loads fast, then flickers, and then shows the content again.

This is because of pre-rendering. Blazor first renders the content on the server, and when the site becomes interactive, another hosting model steps in and does not reuse the state. This was a common problem when .NET 8 was released, because prerendering became the default behavior.

Blazor is not alone with prerendering; Angular has SSR as well as NextJS (for React). The reason this is a good thing is that it loads faster and works well with SEO. A site that doesn't prerender needs to fully load before getting the information, which, at least as perceived, loads slowly.

There was a way to handle Persistent Component State manually, but it involved a lot of code, and the benefit wasn't clear enough. It might be an internal site that didn't need SEO, so the effort wasn't worth it and developers ended up turning it off.

But with .NET 10, it became so much easier, thanks to its best new feature: the `PersistentState` attribute.

Let's have a look:

1. Start our blog and watch when it loads the blog post list. You might notice it is empty, and then the content appears. There was no flickering, but I promised you flickering! Now, if we instead navigate to `/weather`, there it is, the flicker.
   So what is up with that?

   Well, I was a bit surprised at first as well, but the issue is that the `Virtualize` component only runs in interactive mode, so we are not getting any data during prerendering. The virtualized component is also overkill if we don't have a very long list. But since we have a blog, we need to fix that.
2. In the `BlazorWebApp.Client` project, open `Home.razor`.
3. Delete the `LoadPosts` method and the `totalBlogposts` property.
4. Next, we need to add the ability to load the blog posts and rewrite the Virtualize parts to use a foreach statement instead. This is the resulting page:

```
@page "/"
@using Markdig;

@inject IBlogRepository _repository

@if (BlogPosts == null)
{
    <p>Loading</p>
}
else
{
    @foreach (BlogPost p in BlogPosts)
    {
        <article>
            <h2>@p.Title</h2>
            @((MarkupString)Markdig.Markdown.ToHtml(new
                string(p.Text.Take(100).ToArray()), pipeline))
            <a href="/Post/@p.Id">Read more</a>
        </article>
    }
}

@code {
    MarkdownPipeline pipeline;
```

```
    public List<BlogPost>? BlogPosts { get; set; }

    protected override async Task OnInitializedAsync()
    {
        pipeline = new MarkdownPipelineBuilder()
                    .UseEmojiAndSmiley()
                    .Build();
        BlogPosts = await _repository.GetBlogPostsAsync(30, 0);
        await base.OnInitializedAsync();
    }
}
```

As mentioned, the main Virtualize change was removing the `virtualize` component and replacing it with a `foreach` statement. We also added a `public List<BlogPost>? BlogPosts { get; set; }` property and loaded the blog posts in the `OnInitializedAsync` method. Also, we have to make sure we add `async` in front of the method.

5. Now run the project again. You should see fast initial content, followed by **Loading** (when WebAssembly kicks in), and then content again.
Great, we fixed the SEO issue, and we now have prerendering working. But we also added a problem: the flicker. This is where the `PersistentState` keyword comes in.

6. On the `Blogpost` property, add the `PersistentState` attribute like this:

```
[PersistentState]
public List<BlogPost>? BlogPosts { get; set; }
```

7. Then, where we load our blog posts, add `??`, so it looks like this:

```
BlogPosts ??= await _repository.GetBlogPostsAsync(30, 0);
```

Here we are instructing C# that if `BlogPosts` is null (which they are on the first load), then run the `GetBlogPostsAsync` method. But if `BlogPosts` has a value, just use that value (there's no need to go and get that value again). So the first time the component loads (during prerender), it will persist the state (thanks to the `PersistentState` attribute). Save the state as a Base64-encoded string in the DOM, and when the interactive part of Blazor takes over, it will automatically pick it up.

8. Now run the project and see blog posts load without a flicker.

This is not the only thing this attribute does for us. If we are running InteractiveServer it will help us persist the data if the connection gets lost. Some people choose not to use InteractiveServer because it requires a constant connection. If that connection, for any reason, breaks, the state disappears as well. For InteractiveWebassembly, the argument against is usually the long load time.

This amazing attribute resolves both problems. Prerendering makes the site load quickly, preserves the state on the server, and enables us to do quick, easy SEO. For an interactive server, it will keep the state, so a constant connection is no longer as necessary as before.

Note

Read more about the `PersistentState` attribute here: `https://learn.microsoft.com/en-us/aspnet/core/blazor/state-management/prerendered-state-persistence?view=aspnetcore-10.0`.

This is by far my favorite addition to Blazor because it makes quality of life so much better for a Blazor developer. When talking about state management and storing data, the first thing you might think of is a database, and that's what we're going to take a look at next.

Storing data on the server side

There are many different ways in which to store data on the server side. The only thing to remember is that Blazor WebAssembly (or InteractiveWebAssembly) will always need an API. Blazor Server (or InteractiveServer) doesn't need an API since we can access the server-side resources directly.

I have had discussions with many developers regarding APIs or direct access, which all boil down to what you intend to do with the application. If you are building a Blazor Server application and have no interest in moving to Blazor WebAssembly, I would probably go for direct access, as we have done in the `MyBlog` project.

I would not do direct database queries in the components, though. I would keep them in a repository, not a Web API. As we have seen, exposing those API functions in a repository, as we did in *Chapter 9*, *Creating an API*, does not require a lot of steps. We can always start with direct server access and move to an API if we want to.

When it comes to storing data, we can use blob storage, key-value storage, relational databases, document databases, table storage, and so on. There is no end to the possibilities. If .NET can communicate with the technology, we can use it.

One way to store state you might not think of is in the URL, and that's what we're going to look at in the next section.

Storing data in the URL

At first glance, this option might sound horrific, but it's not. Data, in this case, can be the blog post ID or the page number if we use paging. Typically, the things you want to save in the URL are things you want to be able to link to later on, such as blog posts in our case.

To read a parameter from the URL, we use the following syntax:

```
@page "/posts/{PageNumber:int}"
```

The URL is `posts` followed by the page number (for paging through blog posts).

To find the route with `PageNumber`, `PageNumber` must be an integer; otherwise, the route won't be found.

We also need a `public` parameter with the same name:

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

If we store data in the URL, we need to make sure to use the `OnParametersSet` or `OnParametersSetAsync` method; otherwise, the data won't get reloaded if we change the parameter. If the parameter changes, Blazor won't run `OnInitializedAsync` again.

This is why our `post.razor` component loads the things that change based on the parameter in the URL in `OnParametersSet` and loads the things that are not affected by the parameter in `OnInitializedAsync`.

We can use optional parameters by specifying them as nullable, like this:

```
@page "/post/{PageNumber:int?}"
```

So this route would match "`/post/`" and "`/post/42`", for example.

Route constraints

When we specify what type the parameter should be, this is called a **route constraint**. We add a constraint so the match will only happen if the parameter value can be converted into the type we specified.

The following constraints are available:

- `bool`
- `datetime`
- `decimal`
- `float`
- `guid`
- `int`
- `long`

The URL elements will be converted to a C# object. Therefore, it's important to use an invariant culture when adding them to a URL.

Note that `string` is not part of the list because that is the default behavior.

Using a query string

So far, we have only talked about routes that are specified in the `page` directive, but we can also read data from the query string.

`NavigationManager` gives us access to the URI, so by using this code, we can access the query string parameters:

```
@inject NavigationManager Navigation
@code {
    var query = new Uri(Navigation.Uri).Query;
}
```

We won't dig deeper into this, but know that it is possible to access query string parameters if we need to.

A much nicer way to access the query string is to use the query parameter using an attribute like this:

```
[Parameter, SupplyParameterFromQuery(Name = "parameterName")]
public string ParameterFromQuery { get; set; }
```

This syntax is a bit nicer to work with; it makes it easier to access parameters and easier to use inside our code as well.

Having data in the URL does not really mean storing the data. If we navigate to another page, we need to make sure to include the new URL; otherwise, it would be lost. We can use the

browser storage instead if we want to store data that we don't need to include every time in the URL.

Sometimes we want to store data on the client, and that's where browser storage comes in.

Implementing browser storage

The browser has a bunch of different ways of storing data in the web browser. They are handled differently depending on the type we use.

Local storage is scoped to the user's browser application. The data will still be saved if the user reloads the page or even closes the web browser. The data is shared across tabs.

Session storage is scoped to the **Browser** tab; if you reload the tab, the data will be saved, but if you close the tab, the data will be lost. Session storage is, in a way, safer to use because we avoid risks with bugs that may occur due to multiple tabs manipulating the same values in storage.

To be able to access the browser storage, we need to use JavaScript. Luckily, we won't need to write the code ourselves.

In .NET 5, Microsoft introduced **Protected Browser Storage**, which uses data protection in ASP.NET Core but is not available in WebAssembly. We can, however, use an open-source library called `Blazor.Storage`, which can be used by both Blazor Server and Blazor WebAssembly.

> **Note**
>
> Please note that this data is stored in clear text on the user's computer, so be careful with what you store!

Implementing session storage

For Blazor WebAssembly, we will use `Blazor.Storage` to implement session storage:

1. In the `BlazorWebApp.Client` project, add a `NuGet` reference to `Blazor.Storage`. This package doesn't have a lot of downloads, but it builds upon a heavily used package that is no longer maintained.
2. Add a new folder called `Services`.
3. In the new folder, create a new class called `BlogBrowserStorage.cs`.

4. Open the new file and replace the content with the following code:

```
using Blazored.SessionStorage;
namespace BlazorWebApp.Client.Services;
public class BlogBrowserStorage
{
    ISessionStorageService Storage { get; set; }
    public BlogBrowserStorage(ISessionStorageService storage)
    {
        Storage = storage;
    }
    public async Task DeleteAsync(string key)
    {
        await Storage.RemoveItemAsync(key);
    }
    public async Task<T?> GetAsync<T>(string key)
    {
        return await Storage.GetItemAsync<T>(key);
    }
    public async Task SetAsync(string key, object value)
    {
        await Storage.SetItemAsync(key, value);
    }
}
```

We are creating some methods to get, set, and delete data from the session storage.

The `Blazor.Storage` library still uses the `Blazored` namespaces to maintain compatibility, which is a nice touch.

5. In the `Program.cs` file, add the following namespaces:

```
using Blazored.SessionStorage;
using BlazorWebApp.Client.Services;
```

6. Then, just above `await builder.Build().RunAsync();` add the following:

```
builder.Services.AddBlazoredSessionStorage();
builder.Services.AddScoped<BlogBrowserStorage>();
```

The `AddBlazoredSessionStorage` extension method hooks up everything so that we can start using the browser session storage. Then we add our `BlogBrowserStorage` – in this case, we are not using an interface but instead using the class directly.

7. In the BlazorWebApp project, in `Program.cs`, add the exact same thing, starting with the namespaces:

```
using Blazored.SessionStorage;
using BlazorWebApp.Client.Services;
```

8. Now add the dependency injection (exactly where you put this doesn't matter, but I usually try to keep dependency injection things together):

```
builder.Services.AddBlazoredSessionStorage();
builder.Services.AddScoped<BlogBrowserStorage>();
```

Now we have a service that we can use to access data in the browser storage.

Implementing the shared code

We also need to implement some code that calls the services we just created:

1. In the `BlazorWebApp.Client` project, open `Pages/Admin/BlogPostEdit.razor`. We are going to make a couple of changes to the file.
2. Inject `BrowserStorage` by adding the following code:

```
@using BlazorWebApp.Client.Services
@inject BlogBrowserStorage _storage
```

3. Since we can only run JavaScript calls when doing an action (like a click) or in the `OnAfterRender` method, create an `OnAfterRenderMethod`:

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender && string.IsNullOrEmpty(Id))
    {
        var saved = await
            _storage.GetAsync<BlogPost>("EditCurrentPost");
        if (saved != null)
        {
            Post = saved;
            StateHasChanged();
```

```
        }
    }
    await base.OnAfterRenderAsync(firstRender);
}
```

When we load the component and the `Id` is `null`, this means we are editing a new file; then we can check whether we have a file saved in browser storage.

This implementation can only have one file in the drafts and only saves new posts. If we were to edit an existing post, it would not save those changes.

4. Now we need our `UpdateHTML` method to become async. Change the method to look like this:

```
protected async Task UpdateHTMLAsync()
{
    if (!string.IsNullOrEmpty(Post.Text))
    {
        markDownAsHTML = Markdig.Markdown.ToHtml(
            Post.Text, pipeline);
        if (string.IsNullOrEmpty(Post.Id) &&
            RendererInfo.IsInteractive)
        {
            await _storage.SetAsync("EditCurrentPost", Post);
        }
    }
}
```

If `Id` on the blog post is `null`, we will store the post in the browser storage. Make sure to change all the references from `UpdateHTML` to `UpdateHTMLAsync`.

The problem that may occur is that we try to run `_storage.SetAsync` while prerendering. Prerendering happens on the server before we have a connection to the browser. That means that we can't run JavaScript while we are prerendering. That's why we are using `RendererInfo.IsInteractive`; this means that the piece of code will only run when we are running in an interactive mode, for example, InteractiveServer or InteractiveWebAssembly.

5. Make sure to await the call as well in the `OnParametersSetAsync` method, like this:

```
await UpdateHTMLAsync();
```

We are done with the code part. Now it's time to test the implementation.

6. Run the project by pressing *Ctrl + F5*.
7. Log in to the site (so we can access the admin tools).
8. Click **Blog posts** and then **New blog post**. You may notice that there is a delay between loading the page and the components showing up. This is the initial WebAssembly load time to get everything started.
9. Type anything in the boxes. As soon as we type something in the text area, it will save the post to storage.
10. Click **Blog posts**, so we navigate away from our blog post (note that we have our popup asking if we are sure).
11. Click **New blog post**, and all the information should still be there.
12. Press *F12* to see the browser developer tools. Click **Application | Session storage | https://localhost:portnumber**.
    You should see one post with the key **EditCurrentPost**, and the value of that post should be a JSON string, as seen in *Figure 13.1*. If we were to change the data in the storage, it would also change in the application, so keep in mind that this is plain text and the end user can manipulate the data:

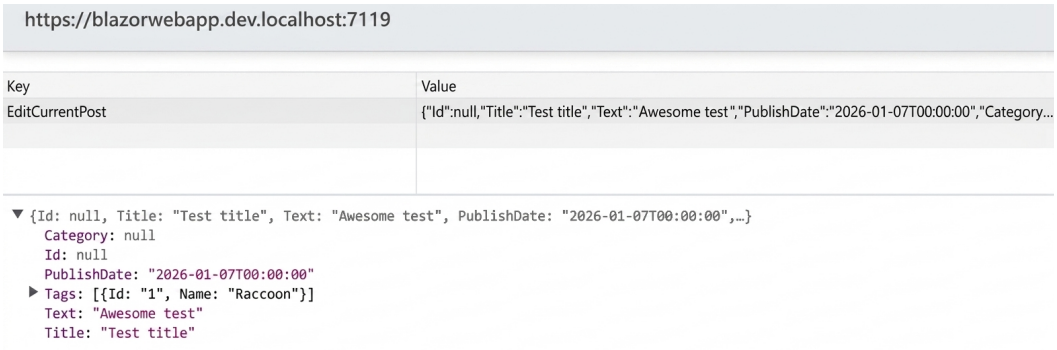


Figure 13.1: Browser storage that is unprotected

Now, we have implemented protected session storage that works with InteractiveServer and InteractiveWebAssembly.

My personal go-to is to use state containers, which just happens to be the next thing we can talk about in the next section.

Using an in-memory state container service

When it comes to in-memory state containers, we simply use dependency injection to keep the instance of the service in memory for the predetermined time (scoped, singleton, or transient).

In *Chapter 6*, *Understanding Basic Blazor Components*, we discussed how the scope of dependency injections differs between Blazor Server (InteractiveServer) and Blazor WebAssembly (InteractiveWebAssembly). The big difference for us in this section is the fact that Blazor WebAssembly runs inside the web browser and doesn't have a connection to the server or other users.

To show how the in-memory state works, we will do something that might seem a bit overkill for a blog, but it will be cool to see. When we edit our blog post, we will update all the web browsers connected to our blog in real time (I did say overkill!).

InteractiveServer has a SignalR connection to all the clients, and InteractiveWebAssembly is running in the browser, so we need to implement this very differently depending on the hosting model. Let's start with Blazor Server.

Implementing real-time updates on InteractiveServer

The implementation for InteractiveServer can also be used for InteractiveWebAssembly. Since WebAssembly is running in our browser, it would only notify the users connected to the site, which would be just you. So it would work, but perhaps not as intended.

But it might be good to know that the same thing works in Blazor Server as well as Blazor WebAssembly:

1. In the `BlazorWebApp.Client` project, in the folder `Interfaces`, create an interface called `IBlogNotificationService.cs`.
2. Add the following code:

```
using BlazorWebApp.Client.Models;
namespace BlazorWebApp.Client.Interfaces;
public interface IBlogNotificationService
{
    event Action<BlogPost>? BlogPostChanged;
    Task SendNotification(BlogPost post);
}
```

Here we have an action that we can subscribe to when the blog post is updated and a method we can call when we update a post.

3. In the `BlazorWebApp` project, create a new folder called `Services` folder, and in that folder, add a new class called `BlazorServerBlogNotificationService.cs` (it might seem unnecessary to give the class a name that includes `BlazorServer`, but it makes sure we can easily tell the classes apart).
4. Replace the content with the following code:

```
using BlazorWebApp.Client.Models;
using BlazorWebApp.Client.Interfaces;
namespace BlazorServer.Services;

public class BlazorServerBlogNotificationService :
    IBlogNotificationService
{
    public event Action<BlogPost>? BlogPostChanged;
    public Task SendNotification(BlogPost post)
    {
        BlogPostChanged?.Invoke(post);
        return Task.CompletedTask;
    }
}
```

   The code is pretty straightforward. If we call `SendNotification`, it will check whether anyone is listening for the `BlogPostChanged` action and whether to trigger the action.

5. Next, in the `BlazorWebApp` project, in Program.cs, add the dependency injection:

```
builder.Services.AddSingleton<IBlogNotificationService,
    BlazorServerBlogNotificationService>();
```

6. Add the necessary namespaces:

```
using BlazorWebApp.Client.Services;
```

   Whenever we ask for an instance of the `IBlogNotificationService` type, we will get back an instance of `BlazorServerBlogNotificationService`.

   We add this dependency injection as a `Singleton`. I can't stress this enough: when using Blazor Server, this will be the same instance for *ALL* users, so we must be careful what data we expose when we use `Singleton`. This means that if we store user-specific data here, it could accidentally be shared between users.

In this case, we want the service to notify all the visitors of our blog that the blog post has changed.

7. In the `BlazorWebApp.Client` project, open `Pages/Post.razor`.
8. Add the following code at the top (or close to the top) of the page:

```
@using BlazorWebApp.Client.Interfaces
@inject IBlogNotificationService _notificationService
@implements IDisposable
```

We added a dependency injection for `IBlogNotificationService` and implemented `IDisposable` to prevent any memory leaks.

9. At the top of the `OnInitializedAsync` method, add the following:

```
_notificationService.BlogPostChanged += PostChanged;
```

We added a listener to the event so we know when we should update the information.

10. We also need the `PostChanged` method, so add this code:

```
private async void PostChanged(BlogPost post)
{
    if (BlogPost?.Id == post.Id)
    {
        BlogPost = post;
        await InvokeAsync(StateHasChanged);
    }
}
```

If the parameter has the same ID as the post we are currently viewing, then replace the content with the post in the event and call `StateHasChanged`.

Since this is happening on another thread, we need to call `StateHasChanged` using `InvokeAsync` so that it runs on the UI thread.

11. We also need to stop listening to the updates by implementing the `Dispose` method. Add the following:

```
void IDisposable.Dispose()
{
```

```
    _notificationService.BlogPostChanged -= PostChanged;
}
```

This removes the event listener to prevent any memory leaks.

12. In the `BlazorWebApp.Client` project, open the `Pages/Admin/BlogPostEdit.razor` file. When we make changes to our blog post, we need to send a notification as well. So, at the top of the file, add the following:

```
@using BlazorWebApp.Client.Interfaces
@inject IBlogNotificationService _notificationService
```

We add a namespace and inject our notification service.

13. In the `UpdateHTMLAsync` method, add the following just inside the `!string.IsNullOrEmpty(Post.Text)if` statement:

```
await _notificationService.SendNotification(Post);
```

Every time we change something, it will now send a notification that the blog post has changed. (I do realize that it would make more sense to do this when we save a post, but it makes for a much cooler demo!)

Now, let's make some changes so we can test the InteractiveServer implementation.

14. In the `BlazorWebApp` project open `Components/App.razor` and change this line:

```
<Routes @rendermode="InteractiveWebAssembly" />
```

To:

```
<Routes @rendermode="InteractiveServer" />
```

With this, we are running our whole project as InteractiveServer, only prerendering and then connecting using SignalR.

15. Run the project by pressing *Ctrl* + *F5*.
16. Copy the URL and open another web browser. We should now have two web browser windows open, showing us the blog.
17. In the first window, open a blog post (doesn't matter which one), and in the second window, log in and edit the same blog post. When we change the text of the blog post in the second window, the change should be reflected in real time in the first window.

I am constantly amazed at how this pretty advanced feature, which would be a bit tricky to implement without using Blazor, only takes around 10 steps using it (not counting the test).

Next, we will implement the same feature for Blazor WebAssembly, but Blazor WebAssembly runs inside the user's web browser. There is no real-time communication protocol like SignalR, built in, as with Blazor Server.

Implementing real-time updates on Blazor WebAssembly

We already have a lot of things in place. We only need to add a real-time messaging system. Since SignalR is both easy to implement and awesome, let's use that.

The first time I used SignalR, my first thought was, wait, it can't be that easy. I must have forgotten something, or something must be missing. Hopefully, we will have the same experience now.

Let's see whether that still holds true today:

1. In the `BlazorWebApp` project, add a new folder called `Hubs`.
2. In the new folder, create a class called `BlogNotificationHub.cs`.
3. Replace the code with the following:

```
using BlazorWebApp.Client.Models;
using Microsoft.AspNetCore.SignalR;
namespace BlazorWebApp.Hubs;
public class BlogNotificationHub : Hub
{
    public async Task SendNotification(BlogPost post)
    {
        await Clients.All.SendAsync("BlogPostChanged", post);
    }
}
```

The class inherits from the `Hub` class, and there is a method called `SendNotification` (keep that name in mind; we will come back to that).

We call `Clients.All.SendAsync`, which means we will send a message called `BlogPostChanged` with the content of a blog post.

The name `BlogPostChanged` is also important, so keep that in mind as well.

4. In the `Program.cs` file add the following:

```
builder.Services.AddSignalR();
```

This adds SignalR. We already have access to SignalR since this project is a mix of hosting models.

5. Add the following namespace:

```
using BlazorWebApp.Hubs;
```

6. Just above `app.MapRazorComponents<App>()`, add:

```
app.MapHub<BlogNotificationHub>("/BlogNotificationHub");
```

Here, we configure what URL `BlogNotificationHub` should use. In this case, we are using the same URL as the name of the hub.

The URL here is also important. We will use that in just a bit.

7. In the `BlazorWebApp.Client`, add a reference to the `Microsoft.AspNetCore.SignalR.Client NuGet` package.

8. In the `Services` folder, create a class called `BlazorWebAssemblyBlogNotificationService.cs`. In this file, we will implement the SignalR communication.
Add the following namespaces:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.SignalR.Client;
using BlazorWebApp.Client.Models;
using BlazorWebApp.Client.Interfaces;
```

9. Then add this class:

```
public class BlazorWebAssemblyBlogNotificationService :
    IBlogNotificationService, IAsyncDisposable
{
    public BlazorWebAssemblyBlogNotificationService(
        NavigationManager navigationManager)
    {
        _hubConnection = new HubConnectionBuilder()
```

```
            .WithUrl(navigationManager.ToAbsoluteUri(
                "/BlogNotificationHub"))
            .Build();
            _hubConnection.On<BlogPost>("BlogPostChanged", (post) =>
            {
                BlogPostChanged?.Invoke(post);
            });
            _hubConnection.StartAsync();
        }
        private readonly HubConnection _hubConnection;
        public event Action<BlogPost>? BlogPostChanged;

        public async Task SendNotification(BlogPost post)
        {
            await _hubConnection.SendAsync("SendNotification", post);
        }
        public async ValueTask DisposeAsync()
        {
            await _hubConnection.DisposeAsync();
        }
    }
```

A lot is happening here. The class is implementing `IBlogNotificationService` and `IAsyncDisposable`.

In the constructor, we use dependency injection to get `NavigationManager` so we can figure out the URL to the server.

Then, we configure the connection to the hub. Then, we specify the URL to the hub; this should be the same as we specified in *step 6*.

Now, we can configure the hub connection to listen for events. In this case, we listen for the `BlogPostChanged` event, the same name we specified in *step 3*. When someone sends the event, the method we specify will run.

The method, in this case, triggers the event we have in `IBlogNotificationService`. Then, we start the connection. Since the constructor can't be async, we don't await the `StartAsync` method here. This can lead to timing issues if we try to use the connection before it is fully established. In a real application, it's better to initialize the connection in an async lifecycle method or ensure it has started before using it.

`IBlogNotificationService` also implements the `SendNotification` method, and we trigger the event with the same name on the hub, which will result in the hub sending the `BlogPostChanged` event to all connected clients.

The last thing we do is make sure that we dispose of the hub connection.

10. Now, in the `Program.cs` file, we need to register the interface and its implementation in dependency injection. Just above `await builder.Build().RunAsync();`, add the following:

```
builder.Services.AddSingleton<IBlogNotificationService,
BlazorWebAssemblyBlogNotificationService>();
```

11. Next, let's switch to only using InteractiveWebassembly. In the `BlazorWebApp` project, in `Components/App.razor`, change this line:

```
<Routes @rendermode="InteractiveServer" />
```

To:

```
<Routes @rendermode="InteractiveWebAssembly" />
```

At this point, it's time to carry out testing.

12. Run the project by pressing *Ctrl* + *F5*.
13. Copy the URL and open another web browser. We should now have two web browser windows open, showing us the blog.
14. In the first window, open a blog post (it doesn't matter which one), and in the second window, log in and edit the same blog post. When we change the text of the blog post in the second window, the change should be reflected in real time in the first window.

In around 10 steps (not counting testing), we have implemented real-time communication between the server and client, with a Blazor WebAssembly client running .NET code inside the web browser.

And no JavaScript!

State management frameworks

Speaking of JavaScript, in the JavaScript framework world of Angular, React, and so on, there are frameworks we can use to manage state (**Redux** and **ngRX**, to name a couple). This is the case for Blazor as well. Very simply, at the moment we have a state that we can change using methods; if the state changes, the components that are listening to that change will be notified.

There are a bunch of frameworks that implement the Flux pattern (just like ngRX) for Blazor. I have personally never used a framework but instead built a Singleton service and connected my components to that (basically what these frameworks do).

Check out Fluxor or Blazor-State if you want to dive deeper into that.

There is also another way to share state between components, which is called root-level cascading values.

Root-level cascading values

Root-level cascading values are a new feature in .NET 8. This is a great way to share state not only between components, but also between different render modes. It will automatically add a cascading value; we have already used this feature when we added `AddCascadingAuthenticationState()`, which uses the root-level cascading value in the background.

This does not share the value between InteractiveServer and InteractiveWebAssembly, though, but does give us a way to share the state between components without using dependency injection.

The really nice thing is that if the value changes, it will automatically change the parameter and trigger a rerender of the component. No special code is needed inside the component. But subscribing to value changes does have a cost, so be careful with how many things you use with root-level cascading values.

Getting a theme could look something like this:

```
@(Preferences?.DarkTheme)

@code {

    [CascadingParameter(Name = "Preferences")]
    public Preferences Preferences { get; set; }
}
```

And in `Program.cs` we would add this:

```
builder.Services.AddCascadingValue<Preferences>(sp =>
{
    var preferences = new Preferences { DarkTheme = true };
    var source = new CascadingValueSource<Preferences>(
        "Preferences", preferences, isFixed: false);
```

```
    return source;
});
```

It is possible to update the values by calling the `NotifyChangedAsync` method on the `CascadingValueSource`. An implementation could look something like this:

```
builder.Services.AddCascadingValue<Preferences>(sp =>
{
    var preferences = new Preferences { DarkTheme = true };
    var source = new CascadingValueSource<Preferences>(
        "Preferences", preferences, isFixed: false);

    if (preferences is INotifyPropertyChanged changed)
        changed.PropertyChanged += (sender, args) =>
            source.NotifyChangedAsync();

    return source;
});
```

Here, we are using the `INotifyPropertyChanged` interface to call the `NotifyChangedAsync` when we change the property. On GitHub, you can find a full example of this if you want to play further with it.

It is also using per component render mode so we can render components as SSR, WebAssembly, Auto and Server on the same page. Super neat, but also opens up to many problems so that is why it is not really covered in this book.

Summary

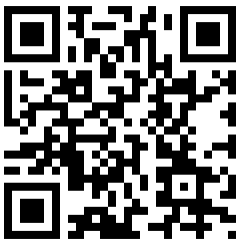
In this chapter, we learned how we can handle state in our application and how we can use local storage to store data. We also made sure to include SignalR to be able to use real-time communication with the server.

Almost all applications need to save data in some form. Perhaps it can be settings or preferences. The things we covered in the chapter are the most common ones, but we should also know that there are many open-source projects we can use to persist state. I personally prefer the components to load state from a database when needed, be self-contained, and not have to rely on state coming from or being somewhere else. This approach has served me well in the past.

In the next chapter, we will take a look at debugging. Hopefully, you haven't needed to know how to debug yet!

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

Part 3

Running Blazor with Confidence

In this part of the book, we'll look at the things that help us build better applications and prepare them for production. We'll debug our code, explore tracing and metrics, test our components, and look at what to think about when deploying a Blazor application. By the end of this part, we'll have a better understanding of how to find problems, verify our code, and get our application ready for users.

This part includes the following chapters:

- *Chapter 14*, *Debugging the Code*
- *Chapter 15*, *Exploring Tracing and Metrics*
- *Chapter 16*, *Testing*
- *Chapter 17*, *Deploy to Production*

14

Debugging the Code

In this chapter, we will take a look at debugging. Hopefully, you haven't gotten stuck earlier in the book and had to jump to this chapter, but luckily, the debugging experience with Blazor is a good one.

Debugging code is an excellent way to solve bugs, understand the workflow, or look at specific values. Blazor has three different ways to debug code, and we will look at each one.

In this chapter, we will cover the following:

- Making things break
- Debugging Blazor Server
- Debugging Blazor WebAssembly
- Debugging Blazor WebAssembly in the browser
- Hot Reload

Technical requirements

Make sure you have followed the previous chapters or use the `Chapter13` folder as a starting point.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter14`.

If you are jumping into this chapter using the code from GitHub, make sure you have added `Auth0` account information in the settings files. You can find the instructions in *Chapter 10*, *Adding Authentication and Authorization,* and you should also rerun the database scripts to populate the database.

To debug something, we should first make something break!

Making things break

Edsger W. Dijkstra once said,

> *"If debugging is the process of removing software bugs, then programming must be the process of putting them in."*

This is definitely true in this section because we will add a page that will throw an exception:

1. In the `BlazorWebApp.Client` project, in the `Pages` folder, create a new Razor component called `ThrowException.razor`.
2. Replace the contents of the file with the following code block:

```
@page "/ThrowException"
<button @onclick="@(()=> {throw new Exception(
    "Something is broken"); })">Throw an exception</button>
```

This page shows a button, and when you press it, it throws an exception.

Great! We have our application's Ivan Drago (he wants to break you, but we might just beat him with some fancy debugging).

The next step is to take a look at debugging Blazor Server.

Debugging Blazor Server

If you have debugged any .NET application in the past, you will feel right at home. Don't worry; we will go through it if you haven't. Debugging Blazor Server is just as we might expect and provides the best debugging experience of the three different types we will cover.

Let's give it a try!

1. In the `BlazorWebApp` project, open `Components/App.razor`.
2. Change the Routes component's render mode so it looks like this:

```
<Routes @rendermode="InteractiveServer" />
```

3. Press *F5* to start the project.
4. Using the web browser, navigate to `/throwexception`.
5. Press *F12* to show the web browser developer tools.
6. In the developer tools, click **Console**.

7. Click the **Throw exception** button on our page.
   We should now be able to see the exception message in the developer tools, as shown in *Figure 14.1*:

```
crit:                                                    blazor.web.ej96puwv3k.js:1
Microsoft.AspNetCore.Components.WebAssembly.Rendering.WebAssemblyRenderer[100]
      Unhandled exception rendering component: Something is broken
System.Exception: Something is broken
   at BlazorWebApp.Client.Pages.ThrowException.<>c.<BuildRenderTree>b__0_0() in
C:\Code\Web-Development-with-Blazor-
4E\Chapter14\MyBlog\BlazorWebApp\BlazorWebApp.Client\Pages\ThrowException.razor:line
2
   at Microsoft.AspNetCore.Components.EventCallbackWorkItem.InvokeAsync[Object]
(MulticastDelegate delegate, Object arg)
   at Microsoft.AspNetCore.Components.EventCallbackWorkItem.InvokeAsync(Object arg)
   at
Microsoft.AspNetCore.Components.ComponentBase.Microsoft.AspNetCore.Components.IHandle
Event.HandleEventAsync(EventCallbackWorkItem callback, Object arg)
   at Microsoft.AspNetCore.Components.EventCallback.InvokeAsync(Object arg)
   at Microsoft.AspNetCore.Components.RenderTree.Renderer.DispatchEventAsync(UInt64
eventHandlerId, EventFieldInfo fieldInfo, EventArgs eventArgs, Boolean
waitForQuiescence)
```

Figure 14.1: Exception in the web browser

   Even though we get the error in the console in the web browser, it is a real C# error.

   This makes it quite easy to find the problem if there is an exception in an app during production (perish the thought). Being able to see the error so easily has saved us many times.

   Now let's try a breakpoint.

8. In Visual Studio, in the `BlazorWebApp.Client` project, open `Pages/Home.razor`.
9. In the OnInitializedAsync method, set a breakpoint on the row `await base.OnInitializedAsync();` by clicking the leftmost border (making a red dot appear). You can also add a breakpoint by pressing *F9*.
10. Go back to the web browser and navigate to `https://localhost:portnumber/` (the port number may vary).

Visual Studio should now hit the breakpoint, and by hovering over variables, you should be able to see the current values.

Both breakpoints and exception debugging work as we might expect. Next, we will take a look at the second method: debugging Blazor WebAssembly.

Debugging Blazor WebAssembly

Blazor WebAssembly can, of course, be debugged as well. There are a couple of things to keep in mind. Debugging InteractiveWebAssembly, like the ones we have in our blog, is going to work just the same as with Blazor Server. Breakpoints and exceptions will also work just the same. However, there is an option to run Blazor WebAssembly as a standalone app, and that works a bit differently.

To be able to play around with that, we need to add another project:

1. Right-click on the **MyBlog solution**, select **Add | New Project**..., and select **Blazor WebAssembly Standalone App**.
2. Change the project name to `BlazorWebAssemblyApp`.
3. Leave the default values as is and click **Create**.
4. Right-click on our **BlazorWebAssemblyApp** project and select **Set as Startup Project**.
5. In the newly created project, in the **Pages** folder, open `Counter.razor` and add a breakpoint on the `currentCount++` row.
6. Run the project by pressing *F5*, go to **Counter**, and click the **Counter** button. Behold, the breakpoint gets hit.

This has not always been the case, and I was actually pleasantly surprised it worked that well. In previous versions of .NET, you had to click on another page and then back to have breakpoints hit.

Debugging Blazor WebAssembly is made possible by the following line of code in the `launchSettings.json` file:

```
"inspectUri": "{wsProtocol}://{url.hostname}:
    {url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}"
```

This only works with a browser that has been opened by the IDE, so that is good to know.

But this code is supplied when we create the project, so we don't need to add it manually.

Now let's see what happens with our exception:

1. Copy our `ThrowException.razor` file and put it in the `BlazorWebAssemblyApp/Pages` folder.
2. In the web browser, navigate to `https://localhost:portnumber/throwexception`.

3. Click the **Throw exception** button.
4. We get the exception in the developer tools in the web browser, as shown in *Figure 14.2*:

```
▶ crit:                                          blazor.webassembly.66stpp682q.js:1
Microsoft.AspNetCore.Components.WebAssembly.Rendering.WebAssemblyRenderer[100]
      Unhandled exception rendering component: Something is broken
System.Exception: Something is broken
   at BlazorWebAssemblyApp.Pages.ThrowException.<>c.<BuildRenderTree>b__0_0() in
C:\Code\Web-Development-with-Blazor-
4E\Chapter14\MyBlog\BlazorWebAssemblyApp\Pages\ThrowException.razor:line 2
   at Microsoft.AspNetCore.Components.EventCallbackWorkItem.InvokeAsync[Object]
(MulticastDelegate delegate, Object arg)
   at Microsoft.AspNetCore.Components.EventCallbackWorkItem.InvokeAsync(Object arg)
   at
Microsoft.AspNetCore.Components.ComponentBase.Microsoft.AspNetCore.Components.IHandleEv
ent.HandleEventAsync(EventCallbackWorkItem callback, Object arg)
   at Microsoft.AspNetCore.Components.EventCallback.InvokeAsync(Object arg)
   at Microsoft.AspNetCore.Components.RenderTree.Renderer.DispatchEventAsync(UInt64
eventHandlerId, EventFieldInfo fieldInfo, EventArgs eventArgs, Boolean
waitForQuiescence)
```

Figure 14.2: WebAssembly error

We have one method left to explore: debugging in the web browser.

Debugging Blazor WebAssembly in the web browser

Since the application is running inside the web browser, we can also debug it directly in the browser itself, without relying on Visual Studio. This was also the first experience that was released for Blazor. The experience lacks a bit, though, but it is nice to know that it exists. Let's get started:

1. In Visual Studio, start the project by pressing *Ctrl* + *F5* (run without debugging).
2. In the web browser, press *Shift* + *Alt* + *D*.
   We will get an error message with instructions on how to start the web browser in debug mode.

   I am running Edge, so the way to start Edge would be something like this:

   ```
   msedge --remote-debugging-port=9222 --user-data-dir="C:
   \Users\Jimmy\AppData\Local\Temp\blazor-edge-debug" --no-first-run https://
   localhost:5001/
   ```

   The `port` and `user-data-dir` values will differ from this example, so copy the command from your web browser.
3. Press *Win* + *R*, then paste the command.

4. A new instance of Chrome or Edge will open. In this new instance, press *Shift* + *Alt* + *D*.
5. We should now see a source tab containing C# code from our project. From here, we can set breakpoints that will be hit and hover over variables. Make sure to have only this site open in the web browser (do not have multiple tabs open).

The debug UI can be seen in *Figure 14.3*:

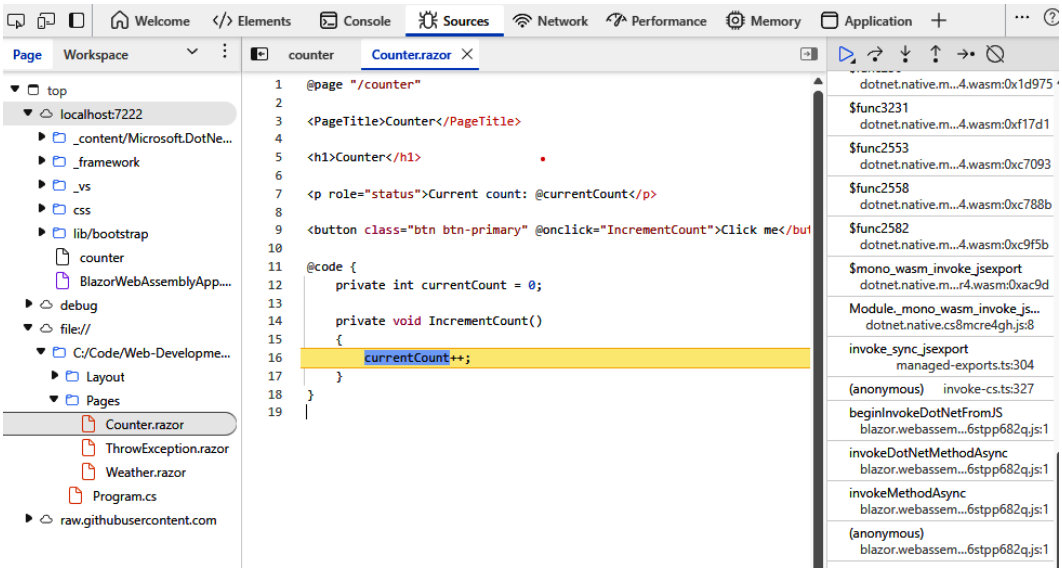


Figure 14.3: The in-browser debug UI

Debugging C# code in the browser is pretty amazing, but since we have been directly debugging in Visual Studio, I don't see much use for this kind of debugging.

Next, we will look at something that might not fall under debugging but is useful when developing Blazor apps.

Hot Reload

In Visual Studio and the **dotnet CLI**, we can enable **Hot Reload**. This means that as soon as we make changes in our application, our Blazor app will automatically get reloaded, and we will (in most cases) not lose the state.

To set this up, do the following:

1. In Visual Studio, right-click the `MyBlog.AppHost` project and select **Set as Startup project**.
2. In Visual Studio, there is a small fire icon. We can use this button to trigger hot reload manually.
   It is only clickable when the application is running, with or without debugging.
3. Select the **Hot Reload on File Save** option.
4. Start the project by pressing *Ctrl* + *F5*.
5. In the web browser, bring up the counter page by adding `/counter` to the URL.
6. Make a change to the `BlazorWebApp.Client/Pages/Counter.razor` file and click **Save**.

Our web browser should now reload, and the change will be shown. It takes a couple of seconds to update, depending on the project's size.

Note

In .NET 10, Microsoft made several important changes to how Hot Reload works. Previously, parts of the pipeline were split, with the Razor engine and Roslyn running in different places and communicating across process boundaries. This often led to slow updates, limited supported changes, or Hot Reload failing altogether.

With .NET 10, Roslyn is now located much closer to the running application. This reduces friction in the edit-and-apply loop and allows code changes to be analyzed and applied more directly. The result is faster feedback, fewer failed Hot Reload attempts, and more consistent behavior, especially when working with Blazor and Razor files.

These improvements are the result of changes in both .NET 10 and Visual Studio 2026, working together to make Hot Reload feel more predictable and less like a best-effort feature.

As we can see, Hot Reload does save time and is pretty amazing. Not having to recompile the project, start up a web browser, and seeing the changes in the browser just seconds after you save the file is simply amazing.

Summary

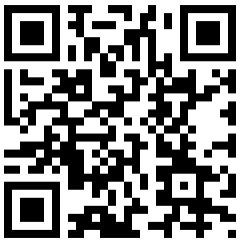
This chapter looked at different ways to debug our Blazor application. There will always be moments when we need to step through the code to find a bug or see what is happening. When these moments are upon us, Visual Studio delivers world-class functionality to help us achieve our goals.

The nice thing is that debugging Blazor applications, whether it's Blazor Server or Blazor WebAssembly, works as expected from a Microsoft product. We get C# errors that are (in most cases) easy to understand and solve.

In the next chapter, we will take a look at telemetry.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

15

Exploring Tracing and Metrics

Once an app is running in production, guessing why something feels slow without insights isn't very useful. This is where metrics and tracing come in. They provide real data on what the app is doing, how users interact with it, and where time is spent.

Metrics help us answer questions like how often users navigate between pages, how long event handlers take to run, or how expensive rendering is. Tracing goes a step further and helps us understand what happened during a specific interaction, especially when something fails.

Blazor integrates with OpenTelemetry, so we can plug into standard tooling rather than build our own monitoring setup.

In this chapter, we will cover the following:

- Blazor Server metrics and tracing
- Blazor WebAssembly diagnostics

Technical requirements

For this chapter, we will use a completely new project. We will use the templates that are provided to us.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter15`.

Blazor Server metrics and tracing

Logs, metrics, and traces all help us understand what our app is doing, but they answer different questions:

- **Logs** tell us *what happened*. They are individual messages like errors, warnings, or information we explicitly write to explain what the app is doing at a specific moment.
- **Metrics** tell us *how often* and *how long*. They are numbers collected over time, things like how many times a page was visited, how long rendering takes, or how many users are connected right now. Metrics are great for spotting trends and changes.
- **Traces** tell us *how something happened*. They follow a single request or user interaction through the system, showing which parts ran and in what order. Traces are especially useful when something goes wrong and we need to understand the full path.

A simple way to remember it is: logs explain events, metrics show patterns, and traces show flow.

Blazor Server now includes built-in telemetry, which makes it easier to understand what's going on inside your application. One of the nice things here is that these metrics are automatically picked up by Aspire without any extra setup. This means you can get observability up and running almost immediately.

This is also why I often say that, even though Aspire is designed for distributed systems, it still brings real value to a single-project application. You get insights and monitoring "for free," without needing to wire everything up yourself.

To take a look at some of the metrics added in .NET 10, let's create a new Blazor Server project:

1. Start Visual Studio 2026.
2. Click on **Create a new project**.
3. Select **Blazor Web App**.
4. Name the project `BlazorServer`.
5. Select:
    - **Framework**: **.NET 10.0**
    - **Authentication type**: **None**
    - **Configure for HTTPS**: **Yes**
    - **Interactive render mode**: **Server**
    - **Interactivity location**: **Global**
    - **Include sample pages**

    - **Do not use top-level statements**
    - **Use the dev.localhost TLD in the application URL**
    - **Enlist in Aspire orchestration**: **Yes**
    - **Aspire Version**: Pick the highest number
6. Click **Create**.

Awesome, we now have a clean Blazor Server/Interactive Server project with Aspire configured.

Before we run the application, let's talk about what this adds.

Enabling metrics and tracing

To enable metrics and tracing in a standard ASP.NET Core app, we typically configure OpenTelemetry ourselves. That means registering the Blazor meters for components, component lifecycle, and, if we're using Blazor Server, circuits. We would also register activity sources to trace navigation, event handling, and circuit lifetime.

However, since this chapter is focused on Aspire, we don't really need to do any of that manually. Aspire already wires this up for us. Under the hood, this means it registers the necessary OpenTelemetry meters and activity sources and connects them to a dashboard, so we don't have to configure that ourselves. When we run the app through Aspire, metrics and traces are collected automatically and exposed in the Aspire dashboard. We can see navigation, rendering behavior, circuits, logs, and traces without adding extra configuration code to our project.

So instead of worrying about registering meters and tracing sources, we can focus on reading and understanding the data Aspire provides. This reduces setup and makes it easier to get useful insights during development.

What the metrics tell us

Blazor exposes several built-in meters that give us insight into how the app behaves.

At the component level, we can see how often navigation happens and how long it takes to handle browser events, including our own business logic. This is useful when an app feels sluggish after clicks or form submissions.

At the lifecycle level, we get measurements for parameter updates and rendering. We can see how long render batches take and how large they are, which helps us understand whether we're re-rendering too much or pushing too much UI at once.

If we're using Blazor Server, we also get circuit-related metrics. These tell us how many circuits are active, how many are connected, and how long they live. This is especially useful when we want to understand memory usage or connection patterns on the server.

Tracing in Blazor

In addition to metrics, Blazor also emits tracing activities. These traces come from the Microsoft.AspNetCore.Components activity source and fall into three main areas: circuit lifecycle, navigation, and event handling:

- When a circuit starts, Blazor emits a trace that represents the circuit's lifetime. This trace includes a unique circuit ID and can be linked to the underlying HTTP and SignalR traces. This makes it much easier to follow a user session across different layers of the stack.
- Navigation tracing tracks route changes. Each navigation records which route was visited and which component handled it. With this data, we can see which pages a user visited during a session and how navigation flows through the app.
- Event handling tracing tracks user interactions like clicks. When an event is handled, Blazor records which HTML attribute triggered it, which component received it, and which C# method ran. If an exception happens, that information is included as well. This makes it much easier to answer questions like which click caused the error, on which page it happened, and in which circuit and request context.

Overall, Blazor gives us a lot of performance data out of the box. We don't have to wire everything up ourselves; we just need to enable it (using Aspire).

Now let's see it in action:

1. Start our new project by pressing *F5*.
2. Go to the **Counter** page and click on the **Click me** button a couple of times.
3. Go to the Aspire portal and click **Traces**.
4. You will see a bunch of traces, such as:
    - `Microsoft.AspNetCore.Components.Server.ComponentHub/StartCircuit`
    - `Route /counter -> BlazorServer.Components.Pages.Counter`
    - `Event onclick -> BlazorServer.Components.Pages.Counter.IncrementCount`

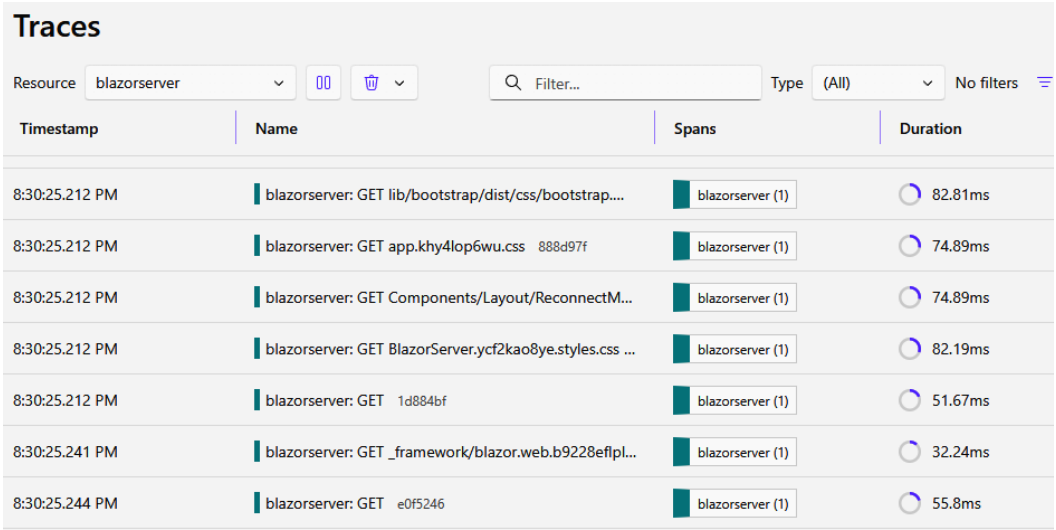


Figure 15.1: Traces view

5. In the top right corner, you will find a filter icon. Press it, and add the following filters:
    - **Parameter**: **Source**
    - **Condition**: **Contains**
    - **Value**: **Microsoft.AspNetCore.Components.Server.Circuits**
6. Click **Apply filter**. This way, you will see all the traces that have to do with circuits.
7. Now, click **Metrics** in the left menu. In this view, we have access to a bunch of different metrics like lifecycle events, active requests, navigation events, and much more. Click on `Microsoft.AspNetCore.Components.Server.Circuits` / `aspnetcore.components.circuit.active`. This shows a graph of the number of running circuits we currently have.

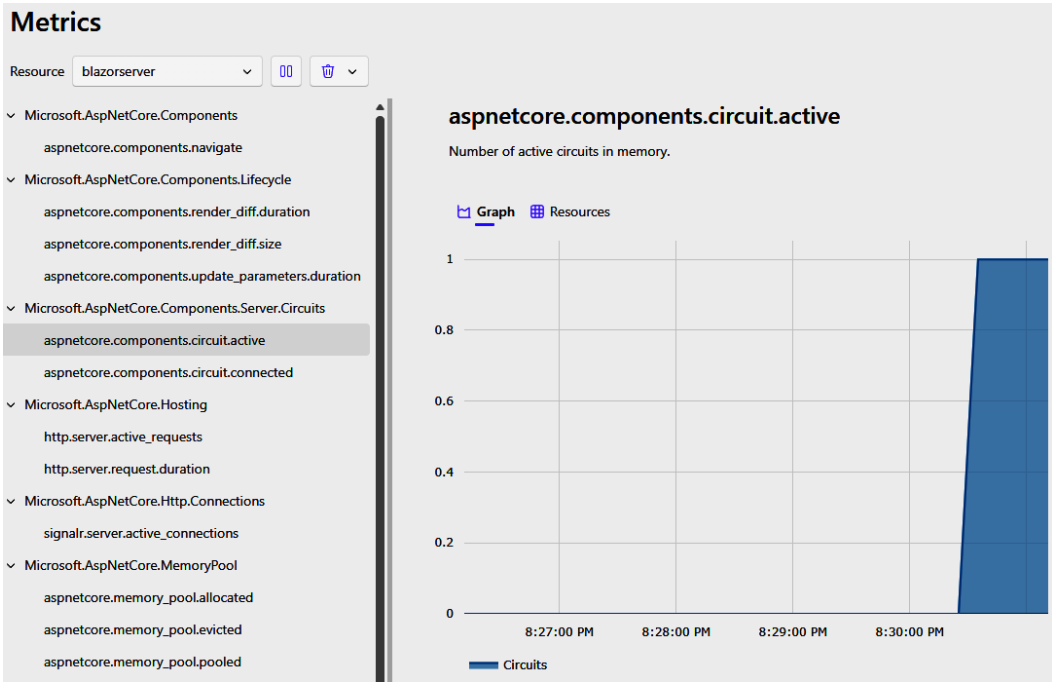


Figure 15.2: Metrics view

8. If you go back to the Blazor website and open up a new tab with the website, you will see the graph change.

If we use Aspire, we get a lot of this for free. If we don't use Aspire, we need to add the sources and meters to our OpenTelemetry config, and we will get everything on our portal of choice.

But this is so far all Blazor Server. What about Blazor WebAssembly?

Blazor WebAssembly diagnostics

Everything we've looked at so far in this chapter – including metrics, traces, circuits, and navigation telemetry – applies to Blazor Server. That data comes from the server runtime and is automatically picked up by Aspire.

But Blazor WebAssembly is different.

In **Blazor WebAssembly**, the app runs entirely inside the browser. There is no server-side runtime managing components, circuits, or navigation. Because of that, the built-in Blazor metrics and traces simply don't exist for WebAssembly apps. There's nothing for Aspire to collect, because the runtime is running in the browser, not on the server.

That doesn't mean we're blind; it just means we use different tools.

Instead of metrics and traces, Blazor WebAssembly relies on diagnostics. These diagnostics come from the underlying .NET runtime that runs inside the browser, giving us insight into what's actually happening under the hood.

Through this runtime, we can inspect things like memory usage, garbage collection, CPU usage, and other runtime-level metrics. The .NET WebAssembly runtime also exposes diagnostic hooks that we can call directly from JavaScript, which let us collect very concrete data from a running app.

For example, we can collect a garbage collection dump to understand memory usage, sample CPU usage over a period of time, or gather runtime metrics for a fixed duration. These diagnostics are lower-level than metrics and traces. They don't tell us how users navigate through the app or which component handled an event. Instead, they help us answer questions like why memory keeps growing, why the app slows down over time, or whether the CPU is doing more work than expected.

On top of that, we also have the browser's built-in tooling. The browser performance and memory tools work just as well for Blazor WebAssembly as they do for any other web app. We can inspect allocations, record performance timelines, and see how WebAssembly execution behaves over time.

In practice, it looks like this:

- For Blazor Server, we rely heavily on metrics and traces, and Aspire gives us most of that for free
- For Blazor WebAssembly, we rely on runtime diagnostics and browser tooling instead

In practice, this works well. Server apps benefit from high-level observability, while WebAssembly apps give us very powerful low-level diagnostics when we need to dig deeply.

For all of these diagnostics, we need to have the .NET WebAssembly Build tools installed.

We can do that by running this console command:

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

Once it is installed, we can start using the diagnostics tools.

The first one we will take a look at is the one built into the browser.

Browser developer tools diagnostics

The browser tools are great when it comes to measuring performance in JavaScript applications, but for WebAssembly we need to add that capability to our project.

For WebAssembly, we will do the same thing as with Blazor Server: we will create a new project:

1. In Visual Studio, right-click on the solution and select **Add | New Project...**.
2. Pick **Blazor WebAssembly Standalone App**, name it `WasmBrowserDiagnostics`, and click **Next**.
3. Configure the project with the following options:
    - **Framework**: **.NET 10.0**
    - **Authentication type**: **None**
    - **Configure for HTTPS**: **Yes**
    - **Progressive Web Application**: **No**
    - **Include sample pages**: **Yes**
    - **Do not use top-level statements**: **No**
    - **Enlist in Aspire orchestration**: **No**
4. Click **Create**.
5. To enable WebAssembly profiling, we need to add a couple of things to the project file. Open the `WasmBrowserDiagnostics` project file and add the following inside the `<PropertyGroup>`:

    ```
    <WasmProfilers>browser</WasmProfilers>
    <WasmNativeStrip>false</WasmNativeStrip>
    <WasmNativeDebugSymbols>true</WasmNativeDebugSymbols>
    <WasmDebugLevel>0</WasmDebugLevel>
    ```

    These settings control how Blazor WebAssembly behaves when it comes to debugging and diagnostics (most of the time, we don't need to touch them, but when we start profiling or digging into performance issues, they suddenly matter a lot):

    - `<WasmProfilers>browser</WasmProfilers>`: This enables browser-based profiling support. In practice, this means the browser's developer tools can properly profile the WebAssembly code that runs our Blazor app. If we want to use the browser's **Performance** tab to see CPU usage, call stacks, or timelines, this needs to be enabled. Without it, profiling data is either missing or much harder to understand.

- `<WasmNativeStrip>false</WasmNativeStrip>`: This controls whether native WebAssembly symbols are stripped out during the build. Stripping makes the app smaller, which is good for production, but bad for diagnostics. By setting this to false, we keep the native symbols. That makes stack traces, dumps, and profiling output much more useful when we're trying to understand what's going on.
- `<WasmNativeDebugSymbols>true</WasmNativeDebugSymbols>`: This tells the build to include native debug symbols for the WebAssembly runtime. These symbols are what allow tools to show readable function names instead of cryptic addresses. If we want meaningful output when collecting GC dumps or CPU samples, this needs to be turned on.
- `<WasmDebugLevel>0</WasmDebugLevel>`: This one controls how much debug information is included. A value of 0 keeps things minimal. That might sound counterintuitive, but it's often what we want when profiling. Higher debug levels can change performance characteristics and make profiling less accurate. With level 0, we get cleaner data that's closer to how the app behaves in real life.

In short, these settings trade app size and build time for better insight. They're not something we'd normally ship to production, but when we're diagnosing performance issues in Blazor WebAssembly, they make the tools actually usable.

6. Now right-click in the `WasmBrowserDiagnostics` project and select **Set as Startup Project**.
7. Run the project by pressing *F5*.
8. In your web browser, press *F12* to open the **Developer tools**.
9. At the top of the window, click **Performance**.
10. Press the **Record** button, click around in our Blazor app for a couple of seconds, and press the **Stop** button.
    Now we have a couple of seconds' worth of diagnostics where you can see click events and what is taking time in the UI:

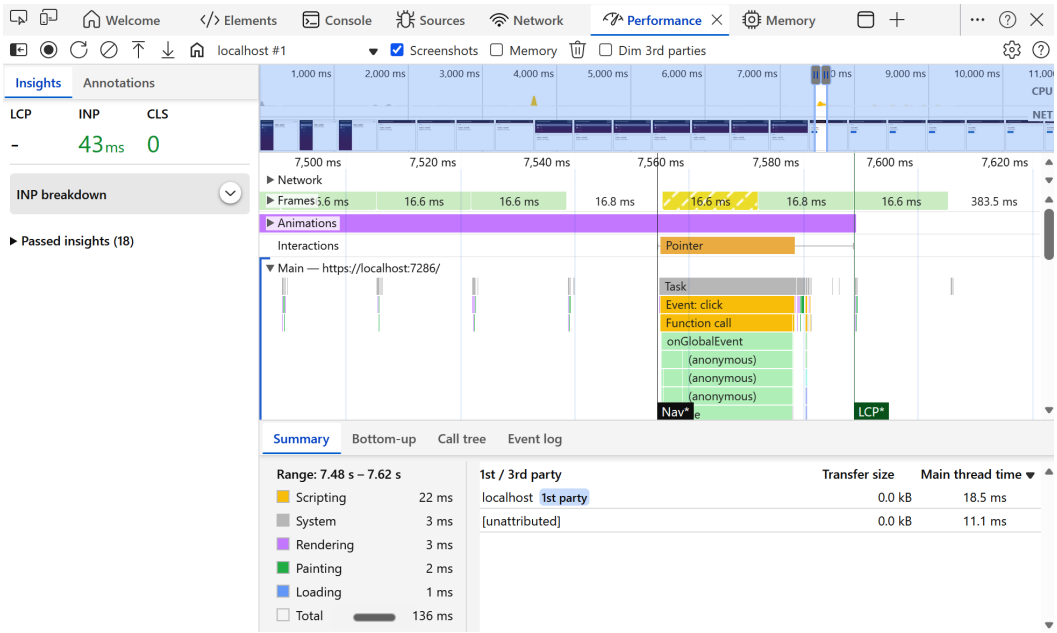


Figure 15.3: Performance tab

With a few lines in the project file, we have now enabled better diagnostics directly in the web browser. Interpreting the data is a book all on its own. The nice thing is that we now have access to the data. However, this is not the only way to access diagnostics. Microsoft has added a few new JavaScript calls that we can run to get additional information. That is what we will look at next.

WebAssembly Event Pipe diagnostics

So far, we've looked at runtime diagnostics we can trigger directly from the browser. In some cases, though, we want even more detailed insight into what the .NET runtime is doing over time.

That's where WebAssembly EventPipe diagnostics come in. EventPipe is a low-level diagnostics pipeline built into .NET that continuously streams runtime events, such as garbage collection activity, CPU sampling, and threading behavior. In a WebAssembly context, this lets us capture a timeline of what the runtime is doing, rather than just taking snapshots at a single point in time.

For this, we'll create a new project. The reason is that we want a clean WebAssembly app configured specifically for diagnostics, without mixing these settings into our existing project.

1. In Visual Studio, right-click on the solution and select **Add | New Project...**.

2. Pick Blazor WebAssembly Standalone App, name it **WasmDiagnostics**, and click **Next**.
3. Configure the project with the following options:
    - **Framework**: **.NET 10.0**
    - **Authentication type**: **None**
    - **Configure for HTTPS**: **Yes**
    - **Progressive Web Application**: **No**
    - **Include sample pages**: **Yes**
    - **Do not use top-level statements**: **No**
    - **Enlist in Aspire orchestration**: **No**
4. Click **Create**.
5. To enable WebAssembly profiling, we need to add a couple of things to the project file. We will set this project up for three different diagnostics: GC Dump, CPU samples, and metrics. So, open the `WasmDiagnostics` project file and add the following lines inside the `<PropertyGroup>`:

```
<EnableDiagnostics>true</EnableDiagnostics>
<WasmDebugLevel>0</WasmDebugLevel>

<WasmPerformanceInstrumentation>all</WasmPerformanceInstrumentation>

<MetricsSupport>true</MetricsSupport>
<EventSourceSupport>true</EventSourceSupport>
```

These settings control whether the WebAssembly runtime exposes the data we need for EventPipe diagnostics; without them, tools like GC dumps, CPU samples, and runtime metrics simply won't have anything useful to work with:

- `<EnableDiagnostics>true</EnableDiagnostics>`: This is the master switch. If this is turned off, none of the runtime diagnostics work. GC dumps, CPU sampling, metrics, EventPipe – all of it depends on this being enabled. If we plan to do any kind of diagnostics in Blazor WebAssembly, this must be true. These settings are typically only enabled during development or when diagnosing issues and are usually turned off in production.
- `<WasmDebugLevel>0</WasmDebugLevel>`: This controls how much debug information is included in the build. A value of 0 keeps things lean and closer to real-world performance. This is usually what we want when profiling because

higher debug levels can distort timing and CPU behavior. This setting affects all diagnostic areas.

- `<WasmPerformanceInstrumentation>all</WasmPerformanceInstrumentation>`: This enables performance instrumentation in the WebAssembly runtime. In practical terms, this is what makes CPU sampling possible. Without it, CPU samples either don't work or give very limited data. If we want to understand where time is spent while the app is running, this setting needs to be enabled.
- `<MetricsSupport>true</MetricsSupport>`: This enables runtime metrics support for WebAssembly. This is required when we want to collect runtime metrics over time, such as allocation behavior or GC activity. These are low-level runtime metrics, not Blazor Server metrics, and they are what tools use when we call collectMetrics.
- `<EventSourceSupport>true</EventSourceSupport>`: This enables EventSource and EventPipe support in the WebAssembly runtime. This is the foundation for GC dumps and other EventPipe-based diagnostics. If this is disabled, we won't be able to collect meaningful GC data or listen to runtime events.

In short, these settings don't make the app faster or slower by themselves. They determine how much insight we get when something goes wrong. We normally enable them only when diagnosing problems, then turn them off again once we're done.

6. Now right-click on the `WasmDiagnostics` project and select **Set as Startup**.
7. Run the project by pressing *F5*.

Keep the app running, as in the next sections we will run commands and need the app running for that. First of all, we will start collecting some data.

Collecting CPU samples

CPU sampling is what we use when the app feels sluggish, but nothing is obviously broken. Maybe scrolling stutters, maybe a page feels heavy, or maybe some background work is eating more CPU than expected. This tool helps us answer the question: what is actually doing work right now?

To collect CPU samples, we can use this command:

```
globalThis.getDotnetRuntime(0).collectCpuSamples({durationSeconds: 5});
```

This command instructs the .NET runtime running in the browser to sample CPU usage for a short period. Instead of taking a snapshot at a single moment, it observes what the app is doing while it's running and records where CPU time is spent.

The `durationSeconds` value controls how long the sampling runs. In this case, we're collecting samples for five seconds. That's usually enough if we actively use the app during that time, clicking buttons, navigating pages, or triggering whatever feels slow.

When the sampling finishes, the runtime generates a profiling file and downloads it through the browser. Depending on the browser, it may automatically save the file or ask where to store it. That file can be opened directly in Visual Studio. If you open Visual Studio and load the file, you'll get a performance view showing call stacks, methods, and where CPU time was spent during the sampling window. This makes it much easier to spot hot paths, expensive methods, or code that runs far more often than we expected.

A good workflow is to start the CPU sampling, immediately reproduce the slow behavior in the app, and then stop touching the app until the sampling completes. That way, the collected data reflects real usage instead of idle time.

CPU sampling doesn't tell us everything, but it's one of the fastest ways to turn "this feels slow" into something concrete we can actually fix. Let's see it in action:

1. In the web browser, press *F12* to open the **Developer tools**.
2. At the top of the window, click **Console**.
3. In the console, run the following command:

   ```
   globalThis.getDotnetRuntime(0).collectCpuSamples({durationSeconds: 5});
   ```

   This will collect CPU samples for 5 seconds. Once it's done, you should have a `.nettrace` file.

4. Drag the `.nettrace` file to Visual Studio, and Visual Studio will open the file.

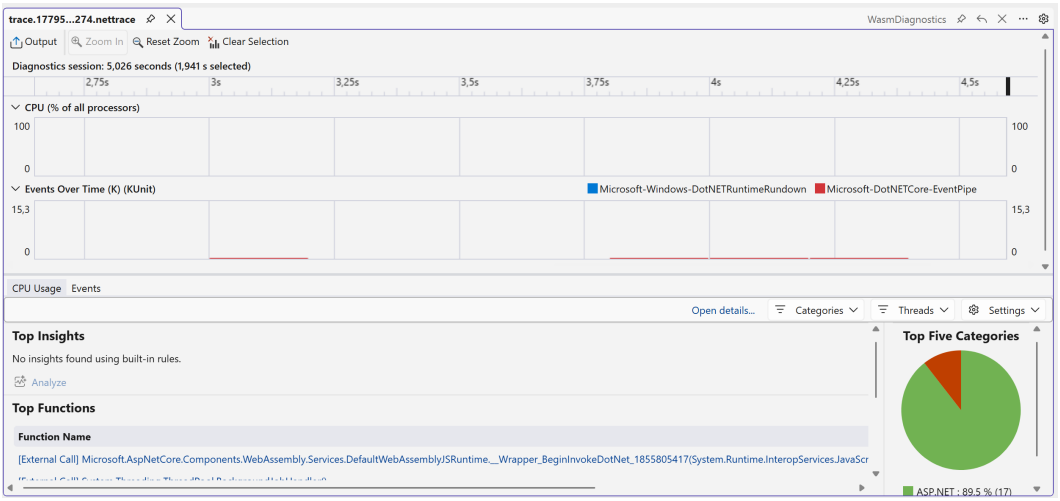


Figure 15.4: CPU samples trace view

This is the same kind of diagnostic output we are used to from .NET applications.

Right now, it is a bit more work to access the information, but who knows, that might change in the future. Hopefully, we won't need to use these features, since we mostly need them when something is wrong.

Next, we will take a look at collecting metrics.

Collecting runtime metrics

Runtime metrics focus on things like memory allocations, garbage collection activity, and general runtime behavior. This makes them useful when the app feels fine at first but slowly degrades, uses more memory than expected, or behaves differently under load.

A common workflow is to collect metrics while reproducing the issue we're investigating. For example, navigating back and forth between pages or repeatedly performing the action that feels heavy. This helps confirm whether the runtime is under pressure or is behaving as expected.

To collect runtime metrics, we can use this command:

```
globalThis.getDotnetRuntime(0).collectMetrics({ durationSeconds: 5 });
```

This command tells the .NET WebAssembly runtime to collect runtime metrics over a fixed period of time. Instead of focusing on individual methods like CPU sampling does, this gives us a broader view of how the runtime behaves while the app is running.

The `durationSeconds` value controls how long metrics are collected. In this example, we collect metrics for five seconds. During that time, we should actively use the app so the data reflects real behavior, including navigation, clicks, rendering, and background work.

When the collection finishes, the runtime generates a diagnostics file and downloads it through the browser, just like with CPU sampling. We can open this file directly in Visual Studio. Visual Studio will show a timeline and metric views that make it easy to see trends, such as frequent garbage collections or high allocation rates during the capture window.

Runtime metrics don't tell us which line of code is slow, but they are very good at answering questions like "are we allocating too much?" or "is the GC working harder than it should?" When used together with CPU samples, they give us a much clearer picture of what's going on inside a Blazor WebAssembly app.

Let's see it in action.

1. In the web browser, press *F12* to open the **Developer tools** and click **Console**.
2. In the console, run the following command:

   ```
   globalThis.getDotnetRuntime(0).collectMetrics({ durationSeconds: 5 });
   ```

   This will collect metrics for 5 seconds. Once it's done, you should have a `.nettrace` file.
3. Drag the `.nettrace` file to Visual Studio, and Visual Studio will open the file.

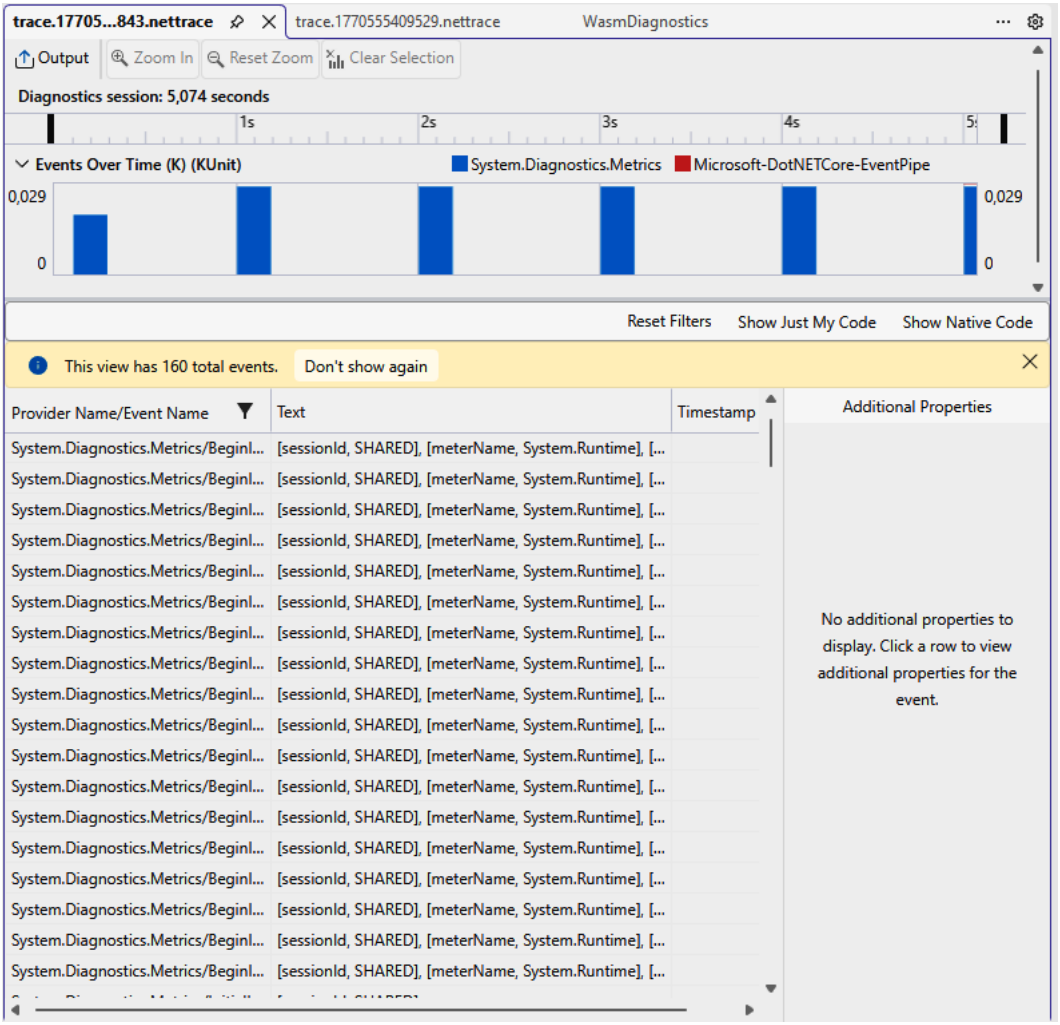


Figure 15.5: Metric view

This is also very similar to the kind of metric data we're used to from regular .NET applications. Instead of focusing on individual methods, this gives us a broader picture of how the runtime behaves over time, such as allocations and garbage collection activity. As with CPU sampling, collecting and inspecting this data in WebAssembly today requires a bit more effort, but the information is familiar and very useful when something feels off.

Next, we'll take a look at collecting GC dumps.

Collecting a GC dump

The following command tells the .NET WebAssembly runtime to collect a GC dump. A GC dump is a snapshot of memory at a specific point in time. It shows which objects are alive, how much memory they use, and why they haven't been collected.

To collect a GC dump, we can use this command:

```
globalThis.getDotnetRuntime(0).collectGcDump();
```

This is the tool we use when memory usage keeps growing or when the app feels fine at first but slowly degrades over time. A GC dump helps us answer questions like what is holding on to memory, whether objects are living longer than expected, or if something is accidentally keeping references alive.

It just takes a few extra steps in Blazor WebAssembly to get the data into a usable format.

Let's see it in action.

1. In the web browser, press *F12* to open the **Developer tools** and click **Console**.
2. In the console, run the following command:

   ```
   globalThis.getDotnetRuntime(0).collectGcDump();
   ```

   This will create a `.nettrace` file.

   When we run this command from the browser's developer tools console, the runtime generates a `.nettrace` file and downloads it to the local machine, usually into the Downloads folder.

   At this point, the file is not yet in a format we can inspect directly. To analyze it as a GC dump, we need to convert it. We do that using the `dotnet-gcdump` tool, which converts the `.nettrace` file into a `.gcdump` file.

3. To install the `dotnet-gcdump` tool, run this command:

   ```
   dotnet tool install --global dotnet-gcdump
   ```

4. Navigate to the `download` folder in Windows Terminal.
5. In the terminal, run the following code:

   ```
   dotnet-gcdump convert .\trace.1770557063444.nettrace
   ```

Replace the filename with your own filename.

Once converted, the `.gcdump` file can be opened in Visual Studio, where we get a full memory analysis view. From there, we can inspect object types, memory usage, and reference paths to understand why memory isn't being released.

Unlike CPU samples and metrics, a GC dump is not time-based. It captures memory exactly at the moment we trigger it, so timing matters. We usually want to reproduce the issue first, then collect the dump while the app is in the problematic state. This is the same kind of memory analysis we're used to from regular .NET applications.

6. Now we should have a `.gcdump` file. Drag that file into Visual Studio to see the collected data.

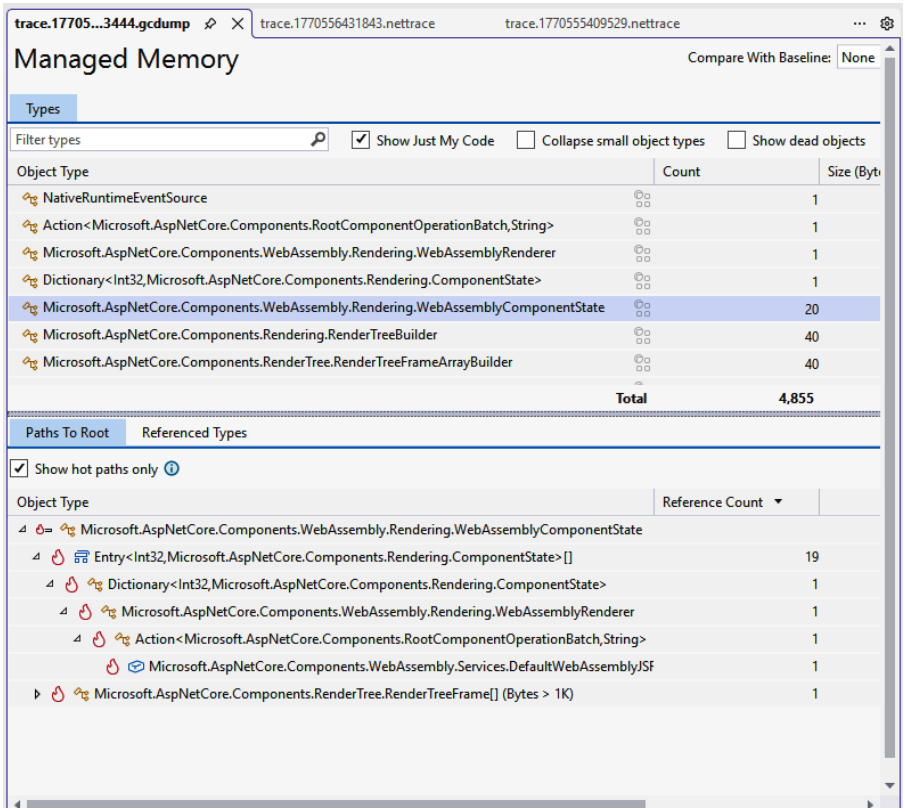


Figure 15.6: GC-dump view

Overall, this is the same kind of diagnostic tooling we're used to from regular .NET applications, just exposed in a different way. CPU samples, runtime metrics, and GC dumps give us deep insight into how the WebAssembly runtime behaves when something doesn't feel

right. It takes a bit more effort to collect and analyze this data compared to server-side apps, but when we need it, the information is clear, familiar, and very effective for tracking down real problems.

Summary

In this chapter, we explored tracing, metrics, and diagnostics in Blazor, and how they help us understand what our applications are doing when things don't feel quite right. We looked at how Blazor Server exposes rich telemetry that Aspire can pick up automatically, giving us insight into navigation, rendering, events, and circuits with very little setup.

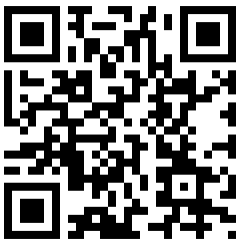
We also looked at Blazor WebAssembly and how the story is different there. Instead of metrics and traces, we rely on runtime diagnostics and browser tooling. CPU samples, runtime metrics, and GC dumps give us deep insight into how the WebAssembly runtime behaves, even if it takes a bit more effort to collect the data.

The common theme is visibility. When we have the right information, performance problems stop being guesswork and start being solvable. We don't need these tools every day, but when we do, they are incredibly powerful.

In the next chapter, we'll switch focus and look at testing and how to make sure our code keeps working as we change and evolve it.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

16

Testing

In this chapter, we will take a look at testing.

Writing tests for our projects will help us develop things rapidly, allowing us to ensure we haven't broken anything with the latest change. It also means we don't have to invest our time in testing the components manually since it is all done by the tests. Testing will improve the quality of the product since we'll know that things that worked earlier still function as they should.

But writing tests for UI elements isn't always easy; the most common way is to spin up the site, use tools that click on buttons, and then read the output to determine whether things work. The upside of this method is that we can test our site on different browsers and devices. The downside is that it usually takes a lot of time to do these tests. We need to spin up the website, start a web browser, verify the test, close the web browser, and repeat for the next test.

We can use this method in Blazor as well (as with any ASP.NET site), and it's a great way to run full end-to-end tests to ensure everything works together as expected.

These kinds of tests usually run in a real browser and interact with the application the same way a user would. This allows us to test navigation, JavaScript, backend communication, and the application as a whole.

The downside is that these tests are often slower and more expensive to run. We need to start the application, launch a browser, execute the test, and then repeat the process for the next test.

Blazor also gives us another option for testing components. Instead of running the full application in a browser, we can test components in isolation using a framework called **bUnit**.

bUnit renders Blazor components internally without spinning up a browser. This makes the tests much faster and gives quick feedback during development. The tradeoff is that we are testing the component itself rather than the full application running in a real browser.

Steve Sanderson created an early prototype of a testing framework for Blazor that Microsoft MVP Egil Hansen later continued developing. Egil's bUnit framework has since become an industry standard in the Blazor community for testing components.

With bUnit, we can render our components in a test, poke at them, trigger events, and verify what actually gets rendered, all without spinning up a browser. It gives us a fast and reliable way to validate that our components behave the way we expect.

In this chapter, we'll use bUnit to test real components. We'll look at how to set up tests, interact with components, and verify both behavior and markup so we can feel confident that our UI does what it should.

This chapter covers the following topics:

- What is bUnit?
- Setting up a test project
- Mocking the API
- Writing tests
- Blazm extension

Technical requirements

Make sure you have read the previous chapters or use the `Chapter15` folder as a starting point.

You can find the source code for this chapter's end result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter16`.

If you are jumping into this chapter using the code from GitHub, make sure you have added `Auth0` account information in the settings files. You can find the instructions in *Chapter 10*, *Adding Authentication and Authorization,* and also rerun the database scripts to populate the database.

What is bUnit?

As mentioned in the introduction, some tests spin up web browsers to test pages/components, but bUnit takes another approach.

bUnit is made specifically for Blazor and is able to define and set up tests using C# or Razor syntax. It can also mock JavaScript interop and Blazor's authentication and authorization.

To make our components more testable, we sometimes need to think about testability from the start or make small adjustments to our code. These changes are usually beneficial for the codebase as well. Code that is hard to test is often a sign that it is too tightly coupled or doing too many things. You'll notice this quickly when working with bUnit. Components with clear parameters, predictable events, and minimal coupling are easy to test, while tightly coupled components are harder to render and verify.

This becomes especially important because of how bUnit works internally. Instead of launching a real browser, bUnit renders Blazor components directly in memory and exposes the rendered output to our tests.

This is also a limitation; we are not testing the real site, but rather the component, so think of this as unit tests, not integration tests. In other words, we're testing the component in isolation, not how it behaves together with the full application, browser, and backend services.

Let's get our hands dirty and create a test project.

Setting up a test project

To be able to run tests, we need a test project:

1. To install the bUnit templates, open **PowerShell** and run the following command:

   ```
   dotnet new install bunit.template
   ```

2. In Visual Studio, right-click **MyBlog** solution and choose **Add | New Project**.
3. Search for `bUnit`, select **bUnit Test Project** in the results, and then click **Next**.
   Sometimes, finding the template by searching can take a bit of time. Instead, you can change the **Project Type** dropdown to **bUnit** to quickly narrow down the list and find it right away. We might need to reboot Visual Studio to find it.
4. Name the project `MyBlog.Tests`, leave the location as is, and click **Next**.
5. Select **xUnit** as the unit test framework, set the target framework to **.net 10.0**, and click **Create**.
   Great! We now have a test project.

Before we mock the API, let's look at the different methods available to us so we can get a feel for how bUnit works.

In `MyBlog.Tests`, we should have the following four files:

- `_Imports.razor` contains the namespaces that we want all of our Razor files to have access to

- `Counter.razor` is a copy of the same `Counter` components we get by default in the Blazor template
- `CounterCSharpTest.cs` contains tests written in C#
- `CounterRazorTests.razor` contains tests written in Razor

Let's start with the `CounterCSharpTest.cs` file, which contains two tests: one that checks that the counter starts at `0` and one that clicks the button and verifies the counter is now `1`. These tests verify the two core behaviors of the Counter component: that the counter starts at `0` and that clicking the button increments the value.

The first test, `CounterStartsAtZero`, looks like this:

```
[Fact]
public void CounterStartsAtZero()
{
    // Arrange
    var cut = RenderComponent<Counter>();
    // Assert that content of the paragraph shows counter
    // at zero
    cut.Find("p").MarkupMatches("<p>Current count: 0</p>");
}
```

Let's break this down. The `Fact` attribute (from xUnit) tells the test runner that this is a *normal* test with no parameters. We can also use the `Theory` attribute to tell the test runner that the `test` method requires parameter values, but we don't need them in this use case.

Then we arrange the test. Simply put, we set up everything we need to do the test. Egil uses `cut` as the component's name, which stands for **component under testing**. In this case, we call the RenderComponent method and pass in the Counter component type.

Next, we assert whether the component outputs the correct thing or not. We use the `Find` method to find the first paragraph tag and then verify that the HTML looks like `<p>Current count: 0</p>`.

The second test, `ClickingButtonIncrementsCounter`, is a bit more advanced, and it looks like this:

```
[Fact]
public void ClickingButtonIncrementsCounter()
{
    // Arrange
```

```
    var cut = RenderComponent<Counter>();
    // Act - click button to increment counter
    cut.Find("button").Click();
    // Assert that the counter was incremented
    cut.Find("p").MarkupMatches("<p>Current count: 1</p>");
}
```

As with the previous test, we start by arranging the test by rendering our `Counter` component. The next step is acting, where we click the button. We look for the button and click it in our `Counter` component. There is only one button, so in this case, it's safe to look for the button this way.

Then it's time to assert again, and we check the markup in the same way as the previous test, but we look for `1` instead of `0`.

There is also an alternative method where we can write our tests with Razor syntax. If we look at the `CounterRazorTests.razor` files, we can see the exact same tests but with different syntax:

```
    [Fact]
    public void CounterStartsAtZero()
    {
        // Arrange
        var cut = Render(@<Counter />);
        // Assert that content of the paragraph shows counter at zero
        cut.Find("p").MarkupMatches(@<p>Current count: 0</p>);
    }
```

It is really only the way we render the component that differs. This does the same thing and is only a matter of preference. I prefer using the Razor version, as it is easier to read and it's also easier to add parameters to our component while testing, since we can write them directly in markup instead of constructing them in code.

Now, let's run the tests and see whether they pass:

1. In Visual Studio, bring up **Test Explorer** by searching for it using *Ctrl* + *Q*. We can also find it in **View | Test Explorer**.
2. Click **Run All Tests** in the view. The Test Explorer should look like *Figure 16.1*:

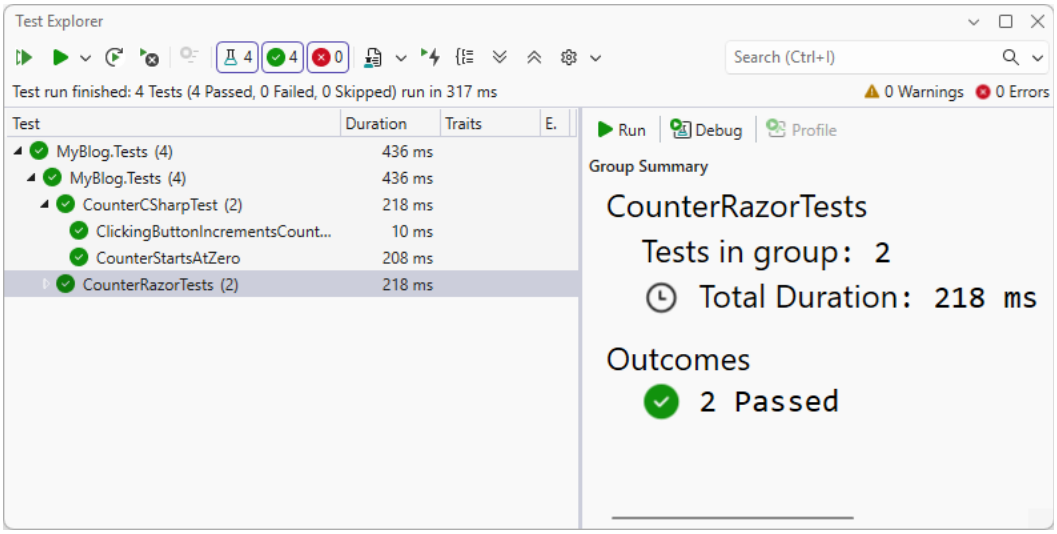


Figure 16.1: Visual Studio Test Explorer

Wonderful! Now, our first test is running and hopefully passing.

Next, we will take a look at mocking the API.

Mocking the API

There are different ways to test our application.

Testing the API is beyond the scope of this book, but we still need to test the components that depend on it. We could spin up the API and test against it, but in this case, we are only interested in testing the Blazor component.

Instead, we can mock the API or create a fake version that doesn't read from the database but instead reads from a predefined dataset. This way, we always know what the output should be.

Luckily, the interface we created for our API is all we need to build a mock API.

Note

We won't implement 100% of the project's tests, so we don't need to mock all the methods. Please feel free to implement tests for all methods as an end-of-chapter exercise.

There are two ways we can implement the mock API: one option is to spin up an in-memory database, but to keep things simple, instead, we will generate posts when we ask for them:

1. In the `MyBlog.Tests` project, add a project reference to the `BlazorWebApp` project (just drag the project to the test project to add a reference).
2. Create a new class called `BlogApiMock.cs`.
3. Add the following namespaces:

```
using BlazorWebApp.Client.Interfaces;
using BlazorWebApp.Client.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

4. Implement the `IBlogRepository` interface. The class should look like this:

```
internal class BlogApiMock : IBlogRepository
{
}
```

   Now, we will implement each method to obtain the data.

5. For `BlogPost`, add the following code in the class:

```
public async Task<BlogPost?> GetBlogPostAsync(string id)
{
    BlogPost post = new()
    {
        Id = id,
        Text = $"This is a blog post no {id}",
        Title = $"Blogpost {id}",
        PublishDate = DateTime.Now,
        Category = await GetCategoryAsync("1"),
    };
    post.Tags.Add((await GetTagAsync("1"))!);
    post.Tags.Add((await GetTagAsync("2"))!);
    return post;
}
public Task<int> GetBlogPostCountAsync()
{
    return Task.FromResult(10);
```

```
}
public async Task<List<BlogPost>> GetBlogPostsAsync(
    int numberofposts, int startindex)
{
    List<BlogPost> list = new();
    for (int a = 0; a < numberofposts; a++)
    {
        list.Add((await GetBlogPostAsync($"{startindex + a}"))!);
    }
    return list;
}
```

When we try to get a blog post through the mock API, we create a blog post on the fly and fill it with predefined information that we can use later in our tests. The same thing goes for getting a list of blog posts.

`GetBlogPostCountAsync` returns that they have a total of ten blog posts in the database.

6. For categories, add the following code:

```
    public async Task<List<Category>> GetCategoriesAsync()
    {

        List<Category> list = new();
        for (int a = 0; a < 10; a++)
        {
            list.Add((await GetCategoryAsync($"{a}"))!);
        }
        return list;

    }

    public Task<Category?> GetCategoryAsync(string id)
    {

        return Task.FromResult<Category?>(new Category() {
            Id = id, Name = $"Category {id}" });

    }
```

Here, we do the same thing: we create categories named `Category` followed by an incrementing number.

7. For comments, add the following code:

```
    public Task<List<Comment>> GetCommentsAsync(string blogPostId)
    {

        var comments= new List<Comment>
        {
            new Comment { BlogPostId = blogPostId, Date =
                DateTime.Now, Id = "Comment1", Name = "Rocket
                    Raccoon", Text = "I really want that arm!" }
        };
        return Task.FromResult(comments);

    }
```

When we call the `GetCommentsAsync` method, it will generate exactly one comment and return that comment.

8. Tags are going to be the same way that categories were handled. It's going to return 10 tags, and each of them will have a unique name with an incrementing number. Add the following code:

```
public Task<Tag?> GetTagAsync(string id)
{

    return Task.FromResult<Tag?>(new Tag() {
        Id = id, Name = $"Tag {id}" });

}

public async Task<List<Tag>> GetTagsAsync()
{

    List<Tag> list = new();
    for (int a = 0; a < 10; a++)
    {
        list.Add((await GetTagAsync($"{a}"))!);
    }
```

```
        return list;

    }
```

9. We will not add tests for other methods in the API. We do need to add them to the mock class to fulfill the interface. Add the following code:

```
    public Task<BlogPost?> SaveBlogPostAsync(BlogPost item)
    {
        return Task.FromResult<BlogPost?>(item);
    }

    public Task<Category?> SaveCategoryAsync(Category item)
    {
        return Task.FromResult<Category?>(item);
    }

    public Task<Tag?> SaveTagAsync(Tag item)
    {
        return Task.FromResult<Tag?>(item);
    }

    public Task<Comment?> SaveCommentAsync(Comment item)
    {
        return Task.FromResult<Comment?>(item);
    }

    public Task DeleteBlogPostAsync(string id)
    {
        return Task.CompletedTask;
    }

    public Task DeleteCategoryAsync(string id)
    {
        return Task.CompletedTask;
    }

    public Task DeleteTagAsync(string id)
    {
        return Task.CompletedTask;
    }
```

```
    public Task DeleteCommentAsync(string id)
    {
        return Task.CompletedTask;
    }
```

We now have a mock API that returns the same data repeatedly, allowing us to run reliable tests. Now it's time to actually write some tests.

Writing tests

As I mentioned earlier in the chapter, we won't create tests for the entire site; we will leave that to you to finish later if you want to. This is just to get a feel for how to write tests.

Let's get started:

1. In the `MyBlog.Tests` project, create a new folder called `Pages`. This is just so we can keep a bit of a structure; we keep the same folder structure as the project we are testing, so the tests are easier to find.
2. In the `_Imports` file, add the following namespaces:

```
@using BlazorWebApp.Client.Interfaces
@using BlazorWebApp.Client.Pages
@using SharedComponents;
```

3. Select the `Pages` folder and create a new Razor component called `HomeTest.razor`.
4. In the `HomeTest.razor` file, inherit from `TestContext` by adding the following code:

```
@inherits BunitContext
```

5. Now, we will add the test. Add the following code:

```
@code{
[Fact(DisplayName ="Checks that the Home component shows 30 posts")]
    public void Show30Blogposts()
    {
        // Act
        var cut = Render(@<Home />);
        // Assert that the content has 30 article tags
        // (each representing a blogpost)
        Assert.Equal(30,cut.FindAll("article").Count());
```

```
    }
}
```

Here we give our test a display name so we understand what it does. The test itself is pretty simple – we know we have `30` blog posts from the mock API, and we also know that each blog post is rendered within an `article` tag. We find all `article` tags and make sure we have `30` in total.

Since we are injecting `BlogApiMock` in our components, we need to configure dependency injection, which we can do in the constructor.

6. We need to add the `HomeTest` method:

```
public HomeTest()
{
    Services.AddScoped<IBlogRepository, BlogApiMock>();
}
```

This method runs when the class is created. Here we declare that if components request an instance of `IBlogRepository`, it will return an instance of our mock API.

This works the same way as with Blazor Server, where we return an API that talks directly to the database, and with Blazor WebAssembly, where we return an instance of the API that talks to a web API.

In this case, it will return our mock API, which returns data that is easy to test.

Now, we need to run the actual test.

7. Delete the example test files:
   - `Counter.razor`
   - `CounterRazorTest.cs`
   - `CounterRazorTests.razor`
8. In Visual Studio, bring up Test Explorer by searching for it using *Ctrl* + *Q*. We can also find it in **View** | **Test Explorer**.
9. Run our tests to see whether we get a green light, as shown in *Figure 16.2*:

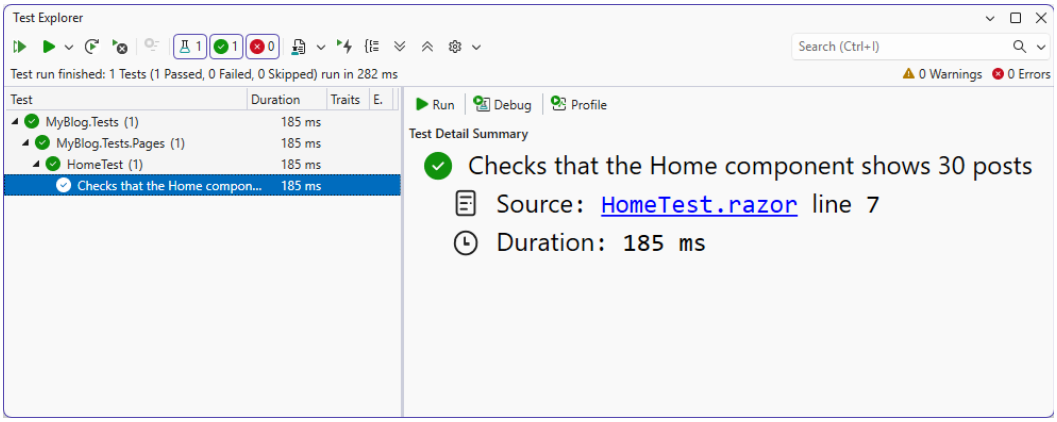


Figure 16.2: Test Explorer with HomeTest

Now, we have a test that checks that 30 posts are rendered.

As you can see, bUnit is an excellent framework for testing. The fact that it is explicitly written for Blazor so that it takes advantage of Blazor's power makes it amazing to work with, and it is super fast compared to more integrated tests, enabling a fast inner developer loop.

Now, we have a simple test for our blog, but bUnit supports more advanced features, such as authentication.

Authentication

Using bUnit, we can test authentication and authorization.

It is, however, not the components themselves that are doing the authentication. We added `AuthorizeRouteView` to `Routes.razor` in *Chapter 10*, *Adding Authentication and Authorization*, so testing that in individual components won't make a difference.

We can use `AuthorizeView` though. We have it in our blog in the `LoginStatus` component, which displays a login link when we are not authorized and a logout link when we are authorized. Please feel free to add the tests below as we did in the previous section, or use them as a reference.

Let's start by testing the unauthorized scenario. Using bUnit's authentication helpers, we can configure a test user that is not logged in and verify that the component displays the correct link.

We can use the `AddTestAuthorization` method to authorize our tests like this:

```
    [Fact(DisplayName ="Checks if log in is showed")]
    public void ShouldShowLogin()
```

```
    {
        // Arrange
        this.AddAuthorization();
        // Act
        var cut = Render(@<LoginStatus />);

        // Assert that there is a link with the text Log in
        Assert.Equal("Log in",cut.Find("a").InnerHtml);
    }
```

This method adds `BunitAuthorizationContext` but is not authorized. The page will then display a link with the text `Log in`.

To test when the user is authorized, we just set the user as authorized:

```
[Fact(DisplayName = "Checks if logout is showed")]
public void ShouldShowLogout()
{
    // Arrange
    var authContext = this.AddAuthorization();
    authContext.SetAuthorized("Testuser", AuthorizationState.Authorized);
    // Act
    var cut = Render(@<LoginStatus />);

    // Assert that there is a link with the text Log out
    Assert.Equal("Log out", cut.Find("a").InnerHtml);
}
```

We can add claims, roles, and other identity data to simulate different authorization scenarios in our tests. This allows us to control exactly how the component should behave for different types of users.

The user we configure in bUnit isn't coming from our database; it's a test user we define in memory.

Authentication and authorization can be tricky to test since they normally depend on external systems and user data. With bUnit, we can simulate these scenarios by configuring a test user in memory. We can add claims, roles, and other identity data to control exactly what the user is allowed to do and verify how our components respond.

In other words, authentication and authorization are fully mocked by bUnit, which makes these kinds of tests much simpler to write.

Testing JavaScript is a bit harder, but bUnit has a solution for that as well.

Testing JavaScript

Testing JavaScript is not supported directly by bUnit. We can, however, test the JavaScript interop used by our components and verify that the expected JavaScript calls are made.

In this book, we have used JavaScript module syntax. One example is the `SharedComponents\BlogButton.razor` component, where we use JavaScript interop to display a confirmation dialog before deleting an item.

The JavaScript interop call looks like this:

```
jsmodule = await jsRuntime.InvokeAsync<IJSObjectReference>(
    "import", "/_content/SharedComponents/BlogButton.razor.js");
return await jsmodule.InvokeAsync<bool>("showConfirm", ConfirmMessage);
```

We load the JavaScript module and then call the `showConfirm` method.

JavaScript testing in bUnit can be done in two modes: `strict` and `loose`. The default value is `strict`, so we need to specify every module and every method. If we choose `loose`, all methods will just return the default value. For a Boolean, it would return `false`, for example. In our case, we want to control what the method returns, so we are going to use the strict mode.

To test the preceding JavaScript call, we can do that by adding something like this:

```
var moduleInterop = this.JSInterop.SetupModule(
    "/_content/SharedComponents/BlogButton.razor.js");
    var showconfirm = moduleInterop.Setup<bool>(
        "showConfirm", "Are you sure?").SetResult(true);
```

Here, we set up a module with the same JavaScript path as before. Then, we specify the method and any parameters. Lastly, we specify the result. In this case, we return `true`, which would return from JavaScript if we want to delete the item. We could also verify whether the JavaScript method is being called, which we will do in the next example.

A complete example for testing this in the `ItemList` component would look like this:

```
@inherits BunitContext

@code {
[Fact(DisplayName = "Test if js method showConfirm is called upon
    using JS interop")]
```

```
public void ShouldShowConfirm()
{

    // Arrange
    var moduleInterop = this.JSInterop.SetupModule("/_content/
        SharedComponents/BlogButton.razor.js");

    moduleInterop.Setup<bool>(
        "showConfirm", "Are you sure?").SetResult(true);

    var cut = Render(@<BlogButton OnClick="()=>{}"
        ConfirmMessage="Are you sure?"/>);

    // Act
    var buttons = cut.FindAll("button");
    buttons.First().Click();

    // Assert
    JSInterop.VerifyInvoke("showConfirm");

  }
}
```

Great job, we now have tests in our project! Even though we aren't covering all scenarios, we now have a solid foundation for testing our components.

It's worth noting that when we test JavaScript interop with bUnit, we're not actually testing the JavaScript itself. Instead, we're testing components that *depend* on JavaScript and verifying that the expected calls are made. This means the JavaScript code could still be broken without these tests failing.

To fully cover those scenarios, JavaScript should be tested separately using appropriate tools. That is outside the scope of this book, but it's important to be aware of.

If you want to learn more about bUnit, check out the following link: `https://bunit.dev/docs/getting-started/index.html`.

Their documentation is fantastic.

It is also good to know that other testing frameworks are available. We use a combination of Playwright tests and bUnit, but they serve different purposes.

bUnit is great for fast unit tests where we verify components in isolation. Playwright, on the other hand, runs full end-to-end tests in a real browser, which means we can verify that everything works together, including JavaScript, navigation, and backend communication.

You can find Playwright here: `https://playwright.dev/dotnet/docs/intro`.

Before we summarize this chapter, we have one more thing to talk about, which is the Blazm Extension.

Blazm extension

There are things that are a bit tedious when developing Blazor applications. We have done many of those things throughout the book. I tend to spell things wrong when I code, and when creating an isolated CSS or JavaScript file, I tend to get the name wrong from time to time and even get the file extension wrong. So, I thought, is there a better way to do this?

Spoiler: Yes, there is!

I built a Visual Studio extension that adds some very nice features.

But why in the world have I waited so long to talk about this!? Well, it's important to learn the "real" way first, then take the shortcuts.

You can check out the extension here: `https://marketplace.visualstudio.com/items?itemName=EngstromJimmy.BlazmExtension`.

The extension can help us add a code-behind file and isolated CSS and JavaScript files, move namespaces into the `_imports` file and much more.

But it can also help us generate tests – not the whole way, but it will help us on the way. Do you remember the Alerts component we used in *Chapter 4*, *Uncovering Aspire*? We can right-click the component and choose **Generate bUnit test**, and then **as Razor syntax**.

It will generate the code to the clipboard so we can paste it where we want it. This is the code it will generate:

```
@inherits BunitContext
@using Bunit
@using SharedComponents;
@code
{
    [Fact]
    public void AlertTest()
    {
        //Arrange
```

```
            SharedComponents.ReusableComponents.Alert/AlertStyle style =
                default!;

            var cut = Render(@<Alert
                Style="@style"
                >
                <ChildContent>
                <b>ChildContent fragment</b></ChildContent>
            </Alert>
            );
            //Act

            //Assert
        }
    }
```

The result is not perfect as we can see, but it gives us something to stand on.

If we were to write a test for the `Alert` component, it would look something like this:

```
[Fact]
public void AlertStyleTest()
{
    //Arrange
    Alert.AlertStyle style = Alert.AlertStyle.Primary;
    var cut = Render(@<Alert Style="@style">
        <ChildContent>
            <b>ChildContent fragment</b>
        </ChildContent>
    </Alert>
    );
    //Act

    //Assert
    cut.MarkupMatches("""<div class="alert alert-primary"
        role="alert"><b>ChildContent fragment</b></div>""");
}
```

We had to clean up some namespaces and add an assertion, but it's pretty neat if you ask me (then again, I am pretty biased on this topic!). I really hope this extension will help you, and I would love for you to give it a five-star review if you enjoy it.

Summary

In this chapter, we looked at testing our application. We looked at how we can mock an API to make reliable tests. We also covered how to test JavaScript interop as well as authentication.

Tests can speed up our development and, most importantly, build quality. With bUnit combined with dependency injection, it is easy to build tests that can help us test our components.

Since we can test every component by itself, we don't have to log in, navigate to a specific place on our site, and then test the entire page, as many other testing frameworks would have us do.

Now, our site contains reusable components, authentication, APIs, Blazor Server, Blazor WebAssembly, authentication, shared code, JavaScript interop, state management, and tests. We only have one more thing to do: ship it!

In the next chapter, it's time to ship.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/cBPYbekR35`

17

Deploy to Production

In this chapter, we will look at the options available when deploying our Blazor application to production. Since there are many options, going through them all would fill a book.

We won't go into detail; rather, we'll cover the different things we need to think about so we can deploy to any provider.

In the end, deployment is what we need to do to use what we build.

In this chapter, we will cover the following:

- Continuous delivery options
- Hosting options

Technical requirements

This chapter is about general deployment, so we won't need any code.

Continuous delivery options

When deploying anything to production, we should consider removing any uncertain factors. For example, if we are deploying from our own machine, how do we know it's the latest version? How do we know that our teammates didn't recently solve a problem and we don't have the fix in our branch? To be honest, how do we even know that the version in source control is the same as in production, or if the version in production even exists in source control? You know the old saying: "Friends don't let friends right-click and publish" (to production, that is)?

This is where **Continuous Integration and Continuous Delivery/Deployment** (**CI/CD**) come into the picture. We make sure that something else builds the deployment to production. Deployment is a large topic with many valid approaches. Rather than focusing on one specific

provider or deployment platform, this chapter focuses on the concepts and considerations that apply regardless of hosting choice. CI/CD platforms help automate the process of building, testing, and deploying our applications. Instead of manually publishing from a developer machine, the deployment process becomes repeatable, traceable, and consistent.

GitHub Actions and Azure DevOps Pipelines are two popular CI/CD platforms commonly used by .NET developers:

- **GitHub Actions** is GitHub's built-in automation platform. Using workflow files stored directly in the repository, we can automatically build, test, and deploy our applications whenever code is pushed or pull requests are created. GitHub Actions integrates well with ASP.NET Core and supports deployment to many different cloud providers and hosting environments.
  Learn more about GitHub Actions for .NET applications here: `https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing/building-and-testing-net`

- **Azure DevOps Pipelines** is Microsoft's enterprise-focused CI/CD platform. Pipelines can automate builds, tests, releases, and deployments across many different environments. Azure DevOps also includes features such as boards, repositories, test plans, and artifact management, making it a complete DevOps platform.
  Learn more about Azure DevOps Pipelines for ASP.NET Core here: `https://learn.microsoft.com/en-us/azure/devops/pipelines/ecosystems/dotnet-core`

There are many other CI/CD systems available as well, such as Jenkins, TeamCity, and GitLab CI/CD. No matter which platform we choose, the overall goal remains the same: automate builds, run tests, validate deployments, and ensure production always runs a known version of the application.

If the CI/CD system we are currently using supports ASP.NET deployment, it can handle Blazor as well. Blazor is built on top of ASP.NET Core, so the same pipelines and tools apply, even though different hosting models (such as Blazor Server and WebAssembly) may have slightly different runtime considerations. If we have tests (which we should), we should also run them as part of our CI/CD pipeline. The nice thing is that component tests can run as part of the CI/CD pipeline without requiring browsers or specialized hardware.

Since Blazor is part of ASP.NET, nothing is stopping us from taking automated testing of our site even further, for example, by adding integration tests or end-to-end tests that run in a real browser and verify the full application flow. There are also tools specifically for WebAssembly, such as wasm-tools, which can help with building and optimizing WebAssembly applications. We will take a closer look at this in *Chapter 19*, *Going Deeper into WebAssembly*.

Additional deployment resources

ASP.NET Core deployment documentation: `https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/`

Azure Static Web Apps: `https://learn.microsoft.com/en-us/azure/static-web-apps/overview`

Hosting ASP.NET Core on IIS: `https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/`

ASP.NET Core publish modes (framework-dependent versus self-contained): `https://learn.microsoft.com/en-us/dotnet/core/deploying/`

CI/CD helps us automate the process of building and deploying our application. The next step is deciding where the application will run, so let's look at some hosting options for Blazor applications.

Hosting options

When it comes to hosting Blazor, there are many options. Any cloud service that can host ASP.NET Core sites should be able to run Blazor without any problems.

We need to think about some things, so let's go through the options one by one.

Hosting Blazor Server/InteractiveServer

If the cloud provider supports enabling or disabling WebSockets, we have to enable them, since that's the protocol used by SignalR.

Sometimes, the cloud provider may support .NET Core 3.x but not .NET 10 out of the box. But don't worry; by publishing our application with the deployment mode set to self-contained (which means the .NET runtime is included with the app), we ensure the deployment includes any files necessary to run the project (this might not be true for all hosting providers).

This is also a good way to ensure we are running the exact framework version we expect.

Hosting InteractiveWebAssembly

Since InteractiveWebAssembly uses the same .NET backend as our Blazor Server setup, we deploy it in the same way. The same hosting requirements apply, including enabling WebSockets if SignalR is used.

Hosting Blazor WebAssembly Standalone

If we are using the Blazor WebAssembly Standalone template, we don't need to think about .NET Core hosting. We can host our application in Azure Static Web Apps or even GitHub Pages. This is one of the upsides of doing a Blazor WebAssembly Standalone site.

Hosting on IIS

We can also host our application on **Internet Information Server (IIS)**. Install the hosting bundle, and it will also ensure the ASP.NET Core IIS module is included if installed on a machine with IIS.

You need to ensure the WebSocket protocol is enabled on the server.

At my old job, we ran our sites on IIS and used Azure DevOps to deploy our sites. Since we are using Blazor Server, the downtime is very evident. As soon as the web loses the SignalR connection, the site will display a reconnect message.

For the sites we are using, there is about 8 to 10 seconds of downtime when deploying a new version, which is pretty quick. You can use sticky sessions to resolve this. In .NET 10, the need for a constant connection is a little better; connections can be persisted for a period of time.

Summary

In this chapter, we discussed why we should use CI/CD, as it significantly improves the quality of the application. We looked at some of the steps we need to take to run our Blazor app on any cloud provider that supports .NET 10.

Deploying is perhaps the most important step for an application. Without deploying our application, it's just code. With the things we mentioned in this chapter, such as CI/CD, hosting, and deployment, we are now ready to deploy the code.

In the next chapter, we will dig deeper into how we can port a current site, use Blazor with other technologies, or use other technologies with Blazor.

Part 4

Going Beyond the Main Application

In this final part of the book, we'll look at how Blazor fits into other scenarios. We'll see how to combine Blazor with existing applications, go deeper into WebAssembly, look at source generators, visit .NET MAUI and Blazor Hybrid, and wrap up with lessons learned from running Blazor in production. By the end of this part, we'll know where Blazor can take us next and how to keep learning after the book ends.

This part includes the following chapters:

- *Chapter 18*, *Moving from, or Combining, an Existing Site*
- *Chapter 19*, *Going Deeper into WebAssembly*
- *Chapter 20*, *Examining Source Generators*
- *Chapter 21*, *Visiting .NET MAUI*
- *Chapter 22*, *Where to Go from Here*

18

Moving From, or Combining, an Existing Site

In this chapter, we will take a look at how we can combine different technologies and frameworks with Blazor.

In many cases, rewriting everything from scratch isn't an option. We might have an existing Angular, React, or MVC application that works well but could benefit from more interactivity or a simpler development model. Blazor gives us the option to introduce those improvements gradually, without replacing everything at once.

There are different options when it comes to moving from an existing site; the first question is, do we want to move from it, or do we want to combine it with the new technology?

Microsoft has a history of making it possible for technology to co-exist, and this is what this chapter is all about.

How can we use Angular and React in our Blazor site, or how can we introduce Blazor into an existing Angular and React site?

As such, we will cover the following topics in the chapter:

- Introducing web components
- Exploring custom elements
- Exploring the Blazor component
- Adding Blazor to an Angular site
- Adding Blazor to a React site
- Adding Blazor to MVC/Razor Pages

- Adding web components to a Blazor site
- Migrating from web forms

Combining technologies can be very useful, either because we can't convert a whole site in one go or because other technologies are a better fit for what we are trying to accomplish.

Having said that, I prefer using one technology on my site, not mixing Blazor with Angular or React. But during a migration period or if our team is mixed, there are benefits to mixing.

However, there is a cost to mixing technologies, which we will look at throughout the chapter. While writing this chapter and revisiting Angular and React, I was reminded how much I enjoy working with Razor syntax. React uses JavaScript with HTML-like syntax (JSX), while Angular uses templates, both of which in different ways resemble Razor. Personally, I still prefer the Razor approach. There are also more moving parts involved, such as Node.js modules, npm, TypeScript, and build tooling.

One of the things I personally appreciate about Blazor is that it can reduce the number of those tools we need to deal with.

Technical requirements

This chapter is a reference chapter and is not connected in any way with the other chapters of the book.

You can find the source code for this chapter's examples at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter18`.

Introducing web components

To work with JavaScript, whether it's bringing JavaScript to Blazor or bringing Blazor into JavaScript, we can use a technology called **web components**.

Web components are a set of web platform APIs that allow us to create new, custom, reusable HTML tags. They are packaged in an encapsulated way, and we can use them very similarly to how we use components in Blazor.

The really nice thing is that we can use them in any JavaScript library or framework that supports HTML.

Web components are built on top of existing web standards like shadow DOM, ES modules, HTML templates, and custom elements.

We will also recognize some of these technologies or variations of them in Blazor. Shadow DOM is the same as Blazor's Render tree, and ES Modules are the type of JavaScript modules we did in *Chapter 12, JavaScript Interop*.

The technology we will take a look at in this chapter is **Custom Elements**.

Exploring custom elements

To bring Blazor into an existing Angular or React site, we use the **Custom Elements** feature. It was introduced as an experimental feature in .NET 6 and has been a part of the framework since .NET 7.

The idea is to create parts of your site in Blazor without having to migrate fully over to Blazor.

When working with Custom Elements, it's important to understand that we are crossing the boundary between Blazor and standard web platform features.

The `RegisterCustomElement` call exposes a Blazor component as a native browser custom element. From the outside, it behaves like any other HTML tag, but under the hood, it still needs the Blazor runtime to function.

Because of that, we also need to make sure the required Blazor framework files and JavaScript runtime are available to the page where the custom element is used. This is why we need to add some JavaScript and configure how resources are loaded in the upcoming steps.

For this feature to work, we need an ASP.NET backend or must manually ensure the `_framework` files are available. This is so that we can serve the Blazor framework files.

There are two ways of running `CustomElements`; we can run it as Blazor WebAssembly or as Blazor Server. Since we are adding Blazor to a client framework like React or Angular, the most relevant method is to run it as Blazor WebAssembly. Therefore, the examples in the first couple of sections of the chapter will be for Blazor WebAssembly.

In the GitHub repo, there is a folder called `CustomElements` in which you will find the code for the projects, from which we will see sample code in this chapter.

It is worth noting that, since the components are served and used on the client, there is nothing that prevents us (or anyone who means us harm) from decompiling the code (if we are using WebAssembly). This is something client-side developers of all frameworks deal with all the time, but it is worth mentioning again.

Exploring the Blazor component

The first thing we need to try out is a Blazor component. I have created a counter component inside a Blazor WebAssembly project named `BlazorCustomElements`.

The default template includes a lot of things, while the repo project is stripped to the bare minimum, making it easy to understand.

The component is nothing new; it's a counter component with a parameter that sets how much the counter should increment. It looks like this:

```
<h1>Blazor counter</h1>
<p role="status">Current count: @currentCount</p>
<p>Increment amount: @IncrementAmount</p>
<button class="btn btn-primary"
        @onclick="IncrementCount">Click me</button>
@code {
    private int currentCount = 0;
    [Parameter] public int IncrementAmount { get; set; } = 1;
    private void IncrementCount()
    {
        currentCount += IncrementAmount;
    }
}
```

The project also needs a reference to these `NuGet` packages:

```
Microsoft.AspNetCore.Components.CustomElements
```

In `Program.cs`, we need to register the Component/Custom Element like this:

```
builder.RootComponents.RegisterCustomElement<Counter>(
    "my-blazor-counter");
```

That's it for the Blazor project.

Now it's time to use our custom element.

We either need to make the WebAssembly file available using a static hosting solution, or we need some project to host it; we will use an ASP.NET server site.

In the project `BlazorWebApplication`, the `Program.cs` looks like this:

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

builder.AddServiceDefaults();
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
        policy
```

```
            .AllowAnyOrigin()
            .AllowAnyHeader()
            .AllowAnyMethod());
});
var app = builder.Build();

app.MapDefaultEndpoints();
app.UseCors();
app.MapGet("/", () => "Hello World!");
app.UseStaticFiles(new StaticFileOptions {
    ServeUnknownFileTypes = true });
app.UseBlazorFrameworkFiles();
app.Run();
```

The important part here is the **CORS** (short for **Cross-Origin Resource Sharing**), which is a browser security feature that decides if your app is allowed to call another domain. By default, the browser blocks those requests unless the server says it's okay. In our case, we are allowing anyone to get our files. In a production scenario, access should be limited. `UseStaticFiles` and `UseBlazorFrameworkFiles` are also important so that we can get all the framework files from the server.

Since we are getting the files from another server completely, we need to rewrite how Blazor gets the files. This could have been avoided by serving the Blazor files from the same server as Angular. But I think this is the more common scenario when it comes to hosting Angular; the Angular part is only Angular, and there is a .NET backend somewhere else.

To handle that, I have added a small JavaScript that hooks everything up. It looks like this, and is served by `BlazorWebApplication`. This is `wwwroot/CustomElementsScript.js`:

```
// Make sure that we can load _content files from the server
// where the custom element is hosted
var scriptUrl = document.currentScript.src;
var url = new URL(scriptUrl);
var customElementBaseUrl = url.origin;

const importMap = {
    imports: {
        [`${window.location.origin}/_content/`]:
            `${customElementBaseUrl}/_content/`,
    }
}
```

```
const importMapJson = JSON.stringify(importMap);
const scriptimportmap = document.createElement('script');
scriptimportmap.type = 'importmap';
scriptimportmap.textContent = importMapJson;
document.head.appendChild(scriptimportmap);

// Make sure that we can load the blazor.webassembly.js file
// from the server where the custom element is hosted
var blazorscript = document.createElement('script');
blazorscript.src =
    `${customElementBaseUrl}/_framework/blazor.webassembly.js`;
blazorscript.type = 'text/javascript';
blazorscript.setAttribute('AutoStart', 'false');
blazorscript.onload = function () {
    Blazor.start({
        loadBootResource: function (type, name, defaultUri, integrity) {
            if (type == 'dotnetjs') {
                return `${customElementBaseUrl}/_framework/${name}`;
            } else {
                return fetch(`${customElementBaseUrl}/_framework/${name}`,
                {
                    cache: 'no-cache',
                    integrity: integrity
                });
            }
        }
    });
};

document.head.appendChild(blazorscript);
```

What it does is rewrite how Blazor gets all the files, making sure they are fetched from the same URL as where the JavaScript file was fetched from.

It adds all the necessary JavaScript files and an Import map to rewrite any calls to `_content` and instead redirect them toward the BlazorWebApplication.

Adding a reference to this script will hook up everything needed for loading WebAssembly. Let's look at how we can add Blazor to an existing Angular site.

Adding Blazor to an Angular site

This demo in this section is created using the ng command.

The source code for this demo can be found inside `AngularFrontend`. A really cool thing is that we are hosting an ASP.NET backend with our Blazor components, along with an Angular frontend and a React frontend, all running on the same backend within Aspire.

There is no need to run NPM or anything like that. Aspire will run that for you.

By default, Angular will be upset when we add our custom element because it does not recognize the tag. To fix this, we need to tell Angular that we are using custom elements. In the `AngularFrontend/src/app/app.ts`, add the following things:

```
import {  ……  , CUSTOM_ELEMENTS_SCHEMA} from '@angular/core';
```

A bit further down in the same file, add:

```
  schemas: [
    CUSTOM_ELEMENTS_SCHEMA // Tells Angular we will have custom tags
                           // in our templates
  ]
```

Now Angular is okay with having custom elements.

Next, it's time to add our component. In `AngularFrontend/src/app/app.html`, we add our custom tag:

```
<my-blazor-counter increment-amount="10"></my-blazor-counter>
```

In this case, we set the `increment-amount` parameter to `10`, which will increase the counter by `10` every time we click it.

To make this all work, we need to load a couple of JavaScript scripts in `AngularFrontend / src/index.html`.

It is important to handle where we are loading our data from, but our script will solve that for us. We just need to add the following:

```
<script src="https://localhost:7154/CustomElementsScript.js"></script>
```

We now have a working Angular/Blazor WebAssembly hybrid. I was honestly amazed at how easy and straightforward this was the first time I tried it. It makes it so easy to include some Blazor components on your Angular site, so you can convert it into Blazor step by step and component by component.

Rewriting this part for this edition got me again; this time, I separated things a bit, using the Angular CLI (ng) to create a project and adding an external site to handle everything. I actually made it a lot easier to hook everything up.

Next, we will do the same using the React site.

Adding Blazor to a React site

Adding Blazor to a React site is very similar to Angular. This demo is based on the React and ASP.NET Core template in Visual Studio. The project is called `ReactFrontend`.

Next, it's time to add our component. In `ReactFrontend/src/App.tsx`, we add our custom tag:

```
<my-blazor-counter increment-amount="10"></my-blazor-counter>
```

In this case, we set the `increment-amount` parameter to `10`, which increments the counter by 10 each time we click it.

To make React be ok with custom elements, we can create a file called `custom-elements.d.ts` with the following content:

```
import 'react';

declare module 'react' {
    namespace JSX {
        interface IntrinsicElements {
            [elemName: string]: any;
        }
    }
}
```

To load and run the Blazor custom element, we need to include the required JavaScript that bootstraps the Blazor runtime and ensures the component can fetch its resources. In `ReactFrontend/index.html`, we need to add:

```
<script src="https://localhost:7154/CustomElementsScript.js"></script>
```

This script ensures our components load, just as they did with Angular.

We now have a working React/Blazor WebAssembly hybrid. This is very similar to Angular, and I was amazed at how easy and straightforward this was as well. It makes it so easy to include some Blazor components on your React site, so you can convert it to Blazor step by step, component by component.

Next, we will do the same using a Razor Pages site.

Adding Blazor to MVC/Razor Pages

When I started working with Blazor in a previous project, this was exactly the scenario we wanted to address. We had an MVC/Razor Pages mix, and it was time for an upgrade.

We solved it by implementing Razor Pages that referred to Razor components. Looking back at it now, it was not a pretty solution, at least not for a while, until we got to the point where most of the code was rewritten in Blazor.

The challenge is that when we navigate to a page with a Blazor component (a Razor component), the page connects to the server and establishes a WebSocket. If we navigate away from a Blazor page to an MVC page, for example, the entire page reloads, and the script reloads as well. A new connection is established, leaving the old one on the server for 3 minutes.

We don't have many users, and for us, that technique was enough to finish the migration and launch a new Blazor version of the site.

But I have some good news!

We can also use the same custom elements to run on a Razor Pages site.

Let's take a look!

The source code for this demo can be found inside the `RazorPagesProject` project on GitHub.

In the previous examples with Angular and React, we used WebAssembly as those technologies are client-side. Razor Pages is server-side, and even though we could use WebAssembly here as well, this is an excellent opportunity to take a look at using Blazor Server. Since Razor Pages are part of ASP.NET, we can add Blazor to an existing site without using Custom Elements. It doesn't really make sense to go the custom elements route when we can have the real thing.

First, we need a reference to our Blazor library. I added the `BlazorCustomElements` project as a reference.

We need to add a NuGet package: `Microsoft.AspNetCore.Components.WebAssembly.Server`. This is so we can host the necessary files for WebAssembly support.

We need to add an `App.Razor` file. For the time being, it is fine that the file is empty. Normally, this file contains routing, but we don't need to use that when we are hosting inside of Razor Pages. In that case, we are referring directly to the component, and we're not using routing. Then we need to enable Blazor Server and Blazor WebAssembly in our Razor Pages by adding the following code to `Program.cs`:

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

And also add:

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

As you can see, it is not the same way we are used to, but we are not using Blazor in the same way here, so the setup looks a bit different. This is basically the old syntax for Blazor server, before .NET 8.

In `Pages/Shared/_Layout.cshtml`, we need to add the JavaScript:

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

Last but not least, we need to add our component.

To demonstrate the different ways we can host the component, we will add it three times, each with a different render mode.

In `Pages/Index.cshtml`, we add:

```
<component type="typeof(Counter)" render-mode="Server" />
<component type="typeof(Counter)" render-mode="WebAssembly" />
<component type="typeof(Counter)" render-mode="WebAssemblyPrerendered" />
```

And we are done; we now have Razor Components (running inside of Blazor) inside our Razor Pages site (which, of course, is an ASP.NET site with Razor Pages turned on).

The cool part is that with only a few changes, we can switch this implementation to run WebAssembly instead of Blazor Server for the Blazor components.

Again, I am super impressed by this; it makes it so simple to migrate existing sites to Blazor.

Next, we will look at how we can use Angular or React controls on our Blazor website.

Adding web components to a Blazor site

We have looked at adding Blazor to an existing Angular, React, and even MVC/Razor Pages site.

But sometimes, that perfect library you love to use might not have a Blazor counterpart. We know we can implement JavaScript interop and build it ourselves, but can we also use Angular and React libraries in Blazor?

We have two options here: either convert our site to Angular/React and use those examples, or convert the JavaScript library into a web component and use it in Blazor.

Until now, we haven't used **npm** (**Node Package Manager**) or anything like that because, in most cases, we don't need it. npm is a tool used to install and manage JavaScript libraries and dependencies.

Now that we are mixing technologies, npm is the easiest way to bring in those libraries. npm is outside the scope of this book, so we won't go into any details about it.

How to convert Angular, React, or any other web-based framework or library into a web component is also outside the scope of this book, so we won't delve deeper into that concept. The source code for this section can be found inside the `BlazorProject` project on GitHub.

We can browse some of the web components on this site: `https://www.webcomponents.org/`.

I found a Markdown editor on GitHub. Even though we are not implementing it on our blog, feel free to go back and do so if you want to.

We can read about the editor here:

`https://www.webcomponents.org/element/@github/markdown-toolbar-element`.

To get the required JavaScript files, we need to set up npm. In the sample project, this has already been done for you, but the steps are shown here for reference.

In the project folder (`BlazorProject.Client`), run the following commands:

```
npm init
npm install --save @github/markdown-toolbar-element
```

This will bring down the JavaScript we need.

Next, copy the `BlazorProject\node_modules\@github\markdown-toolbar-element\` folder to the `wwwroot` folder (in the server project) and include it in the `BlazorProject` project.

This approach makes the files part of our application, which is useful for demos and controlled environments. In a real-world scenario, we could also load these files from a CDN or another external source instead of including them directly in the project.

Now, JavaScript will be accessible from our project.

In `app.razor`, we need to add a reference to the JavaScript, and we put it below the Blazor JavaScript:

```
<script type="module"
    src="markdown-toolbar-element/dist/index.js"></script>
```

This component is an ES6 module, so we set the type to `"module"`.

Now, all that is remaining is to add our component. In the demo project, I added it to the `MarkdownDemo` component in `BlazorProject.Client`.

First, let's take a look at the component:

```
<markdown-toolbar for="textarea" role="toolbar">
    <md-bold class="btn btn-sm" tabindex="0">bold</md-bold>
    <md-header class="btn btn-sm" tabindex="-1">header</md-header>
    <md-italic class="btn btn-sm" tabindex="-1">italic</md-italic>
    <md-quote class="btn btn-sm" tabindex="-1">quote</md-quote>
    <md-code class="btn btn-sm" tabindex="-1">code</md-code>
    <md-link class="btn btn-sm" tabindex="-1">link</md-link>
    <md-image class="btn btn-sm" tabindex="-1">image</md-image>
    <md-unordered-list class="btn btn-sm" tabindex="-1">
        unordered-list</md-unordered-list>
    <md-ordered-list class="btn btn-sm" tabindex="-1">
        ordered-list</md-ordered-list>
    <md-task-list class="btn btn-sm" tabindex="-1">
        task-list</md-task-list>
    <md-mention class="btn btn-sm" tabindex="-1">mention</md-mention>
    <md-ref class="btn btn-sm" tabindex="-1">ref</md-ref>
    <md-strikethrough class="btn btn-sm" tabindex="-1">
        strikethrough</md-strikethrough>
</markdown-toolbar>
```

Here we have `textarea` with binding to a C# variable. We are mixing JavaScript components with C# variables:

```
<textarea @bind="markdown" @bind:event="oninput" rows="6"
    class="mt-3 d-block width-full" id="textarea" contenteditable="false"
    spellcheck="false"></textarea>
@markdown
@code
{
    private string markdown = "Hello, **world**!";
}
```

The important part here is the `for="textarea"` attribute on markdown-toolbar. It points to the `textarea` with the matching `id`. When we click one of the toolbar buttons, the web component updates that `textarea` directly by inserting the corresponding Markdown syntax, such as bold, italic, or a link, at the current cursor position.

Since the `textarea` is still a normal HTML element with Blazor binding, Blazor picks up the change and updates the markdown C# variable as well. This means the JavaScript web component handles the editing behavior, while Blazor still keeps track of the value.

We have integrated a web component into our Blazor project, which binds to a C# variable.

This is super powerful and opens new possibilities for adding existing functionality to our Blazor site.

Now we know how to handle SPA frameworks like React and Angular. But what about server frameworks like Web Forms? This is what we will look at next.

Migrating from Web Forms

Last but not least, we have **Web Forms**.

Jeff Fritz has been working on a project that fills the gap between WebForms and Blazor.

You can find it here: `https://github.com/FritzAndFriends/BlazorWebFormsComponents`.

To quote Jeff: *A collection of Blazor components that are drop-in replacements for the ASP.NET Web Forms control of the same name. The library also includes some other shims and modules that make migrating to Blazor a smooth experience.*

I haven't personally used it. I'm not sure whether it's a good idea, but I do know that Jeff does a lot of amazing things in the community, so I would assume this is really good stuff as well.

The first thing we should know is that Blazor is in many ways very similar to Web Forms, so the learning curve to get to Blazor is almost nonexistent since we have state management in Web Forms as well as Blazor.

There are some migration strategies where you would use **Yet Another Reverse Proxy** (**YARP**). Still, my recommendation would be to migrate a part of the website to Blazor and have two sites running until we reach the point where it is feature-complete. Moving to Blazor is fairly quick to do, and in the end, I believe it will save you time.

When we moved our site from MVC to Blazor, we realized that in some cases it was faster to rewrite the component in Blazor than to try to solve it in MVC.

Web forms should convert even faster since the backend code should be closer to Blazor than to MVC.

So, what should we do? Should we upgrade or keep using web forms? Upgrade – you will not be disappointed!

Summary

We discussed adding Blazor to other technologies, like Angular, React, and Razor Pages, using web components in this chapter. We looked at how to add web components to a Blazor project and leverage JavaScript libraries in our Blazor app.

Upgrading a current site to Blazor can be a lot of work. At my former employer, we made this journey over six years ago. In our case, we wanted to update our MVC site to be more interactive. We went for Blazor, and I would argue it saved our project and made us more productive, resulting in a more interactive user experience.

In the next chapter, we will delve deeper into Blazor WebAssembly.

19

Going Deeper into WebAssembly

In this chapter, we will go deeper into technologies relevant only to Blazor **WebAssembly**.

Most things in Blazor can be applied to Blazor Server and Blazor WebAssembly. Still, since Blazor WebAssembly runs in the web browser, we can optimize the code and use libraries we can't use server-side.

We will also look at some common problems and how to solve them.

In this chapter, we will cover the following:

- Exploring the WebAssembly template
- .NET WebAssembly build tools
- AOT compilation
- WebAssembly **Single Instruction, Multiple Data** (**SIMD**)
- Trimming
- Lazy loading
- Progressive web apps
- Native dependencies
- Common problems

Some parts of this chapter are a great opportunity to follow along, while other parts are for reference so that you can find the right information when you need it. The numbered steps are easy to follow along with.

Technical requirements

This chapter is a reference chapter and is not connected with the book's other chapters. You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter19`.

Exploring the WebAssembly template

The WebAssembly standalone template looks slightly different from the templates we looked at in *Chapter 2*, *Creating Your First Blazor App*. In the Blazor Web App template, our entry point is the `App.razor` file. It contains the HTML tags we need to get started. On the other hand, the WebAssembly template had an `Index.html` file.

Let's create a project so we can take a look:

1. Create a new project and use the **Blazor WebAssembly Standalone App** template.
2. Name the project `BlazorWebAssembly`.
3. Leave the defaults as is and press **Create**.

Let's look at the `wwwroot` folder. Here we have an `Index.html` file that contains all the CSS, JavaScript, and so on. This is the same content as the `App.razor` file in the Blazor Web App template. We also have an `App.razor` file in the WebAssembly project, but that contains the same things as the `Routes.razor` file in the Blazor WebApp project template. This can be a bit confusing if we work with both templates.

Let's take a look at each file but only focus on the things that are specific to WebAssembly. In `Index.html`, we have some interesting code:

```
<div id="app">
    <svg class="loading-progress">
        <circle r="40%" cx="50%" cy="50%" />
        <circle r="40%" cx="50%" cy="50%" />
    </svg>
    <div class="loading-progress-text"></div>
</div>
```

This is a `div`, and the content is a progress bar showing the WebAssembly loading progress using SVG. In the `css/app.css` file, we have this:

```
.loading-progress circle:last-child {
    stroke: #1b6ec2;
    stroke-dasharray:
        calc(3.141 * var(--blazor-load-percentage, 0%) * 0.8), 500%;
    transition: stroke-dasharray 0.05s ease-in-out;
}

.loading-progress-text:after {
    content: var(--blazor-load-percentage-text, "Loading");
}
```

These are just some of the CSS classes for loading progress, but what is interesting is that Blazor provides two CSS values: `--blazor-load-percentage-text` and `--blazor-load-percentage`. They give us an indication of how much time is left when loading our WebAssembly app. This is a great way to customize our progress indicator. The content of the `div` will be replaced by the WebAssembly app once it has loaded.

If we take a look at `Program.cs`, it contains a bit more now when we are running WebAssembly standalone:

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

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

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

Here, we set up our WebAssembly project and tell .NET that we want to render the `App.Razor` component in the `HTML` tag with the id "app". We also tell .NET to render the `HeadOutlet` as the last child of the `head` tag.

Run the project and explore it for a bit. The components inside of the project are the same as the ones we have already looked at in *Chapter 6*, *Understanding Basic Blazor Components*, so there is nothing new going on there.

When we start the project the first time, it takes a couple of seconds to load our app. This is when everything is downloaded and started. The next time our users visit our site, much of the files will be cached and we won't need to download them again.

.NET WebAssembly build tools

For more advanced WebAssembly scenarios, we need additional tooling installed. These tools provide features such as **ahead-of-time** (**AOT**) compilation, support for native dependencies (for example, C or C++ libraries), and low-level optimizations like SIMD.

There are two ways of installing the tools. We can select the .NET WebAssembly Build Tools option when installing Visual Studio (or add them using the Visual Studio installer), or we can install them using the command line:

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

The .NET WebAssembly build tools are based on Emscripten, a compiler toolchain for the web platform. It is responsible for compiling .NET and native code into WebAssembly so it can run efficiently in the browser.

We will use these tools in the upcoming sections when exploring AOT compilation, native dependencies, and other advanced WebAssembly features.

AOT compilation

By default, the only thing that is running as WebAssembly in a Blazor WebAssembly app is the runtime. Everything else is ordinary .NET assemblies running on the browser using a .NET **Intermediate Language** (**IL**) interpreter implemented in WebAssembly.

I was not too fond of that when I started playing around with Blazor; it felt wasteful to run everything using IL instead of something the browser would understand natively. Then, I realized the browser was running the same code as I would on the server. The same code in the browser! This means that I can use most Nuget packages, the same components, and so on. That is pretty amazing!

However, we have the option to compile directly to WebAssembly; this is called **ahead-of-time** (**AOT**) compilation. It has a downside: the app download size will increase, but it will run and load faster.

An AOT-compiled app is generally twice the size of an IL-compiled app. AOT will take the .NET code and compile that directly into WebAssembly.

AOT compilation still uses assembly trimming to reduce the amount of managed code included in the application. However, the native WebAssembly generated from .NET IL is

significantly larger than the original IL code, which increases the download size and also makes the output less compressible over HTTP.

AOT is not for everyone; most apps running without AOT will work fine. However, for CPU-intensive apps, there is a lot to gain by using AOT.

My ZX Spectrum emulator is one of those apps; it runs many iterations per second, and the performance gain from running AOT for these scenarios is remarkable.

To compile our Blazor WebAssembly project using AOT, we add the following property in the `csproj` file:

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

AOT compilation is only performed when the app is published. It can take a long time to compile (seven minutes for the ZX Spectrum emulator), so it is pretty nice that we don't have to wait for that every time we compile our application.

However, running in release mode may be a problem, so if you want to do a quick test in release mode, temporarily disable the preceding setting.

Don't forget to enable it again; I have some experience in that area. If you want to read more about AOT, you can find it at this link: `https://learn.microsoft.com/en-us/aspnet/core/blazor/webassembly-build-tools-and-aot?view=aspnetcore-10.0`.

WebAssembly Single Instruction, Multiple Data (SIMD)

WebAssembly supports **Single Instruction, Multiple Data** (**SIMD**), a CPU feature that allows one instruction to operate on multiple data points at once. This is especially useful for workloads that operate on large blocks of data, such as vector math, image processing, audio, video, and other numeric-heavy operations.

In .NET 10, SIMD's support in Blazor WebAssembly is stable, on by default, and treated as a normal part of the runtime when the browser supports it. If you need to turn it off, you can do so in the project file:

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

SIMD does not require AOT compilation to work. However, using AOT in .NET 10 can further improve performance in scenarios where SIMD-heavy code is used.

When does this matter?

For most Blazor applications, SIMD will not make a noticeable difference. Typical UI-driven apps spend most of their time waiting on user input, network calls, or rendering.

SIMD starts to matter when:

- You are doing heavy numeric calculations
- You process large arrays or buffers in tight loops
- You work with images, audio, video, or custom rendering

If your app does not fall into these categories, you probably don't need to think about SIMD at all.

SIMD is a low-level optimization and outside the scope of this book, but it's useful to know that it exists if you ever run into performance limits in Blazor WebAssembly.

Trimming

By default, when publishing a Blazor WebAssembly app, trimming will be performed. It will remove unnecessary items and, in doing so, reduce the app's size.

If our application uses reflection, the trimmer may have problems identifying what can and cannot be removed, since reflection can access code dynamically. This can lead to required code being removed if we're not careful.

To read more about trimming options, you can look here: `https://learn.microsoft.com/en-us/dotnet/core/deploying/trimming/trimming-options?pivots=dotnet-10-0`.

There are things we can do to make our libraries trimmable, and more importantly, not trim away things we actually need. For most applications, trimming is automatic and works without us doing any optimizations.

Lazy loading

When working with Blazor WebAssembly, one of the challenges is the application's download size. Even though it's not a big problem, in my opinion, we can do some things to improve download and loading times. We will get back to the application download size in the *Common problems* section later in this chapter.

When navigating to a Blazor WebAssembly application, all the DLLs for our application and the DLLs from the .NET runtime/framework assemblies are downloaded. It takes a bit of time to get everything started up. We can load DLLs as needed using **lazy loading** to solve this.

Let's say that our application is massive, where there is a reporting part of the application. Reporting is perhaps not used every day and not used by everyone, and it would make sense to remove that part from the initial download and only load it when we need to.

To make that happen, the part we want to lazy load must be in a separate project/DLL. In the `csproj` file of the Blazor WebAssembly client project, add a reference to the DLL by adding the following code:

```
<ItemGroup>
    <BlazorWebAssemblyLazyLoad Include="{ASSEMBLY NAME}.dll" />
</ItemGroup>
```

The snippet will make sure the file is not downloaded from the start.

To load the DLL when we need it, we will use a built-in service called `LazyAssemblyLoader`.

The `LazyAssemblyLoader` service will make a JS interop call to download the assembly and load it into the runtime.

We make sure to download the necessary assemblies/DLLs in the router (`App.razor`) before we navigate to the component:

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject LazyAssemblyLoader AssemblyLoader
@inject ILogger<App> Logger
<Router AppAssembly="@typeof(App).Assembly"
    OnNavigateAsync="@OnNavigateAsync">
    ...
</Router>
@code {
    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path == "{PATH}")
            {
                var assemblies = await
```

```
                    AssemblyLoader.LoadAssembliesAsync(
                    new[] { {LIST OF ASSEMBLIES} });
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}", ex.Message);
        }
    }
}
```

Here we inject `LazyAssemblyLoader`; it is registered as a singleton by default in a Blazor WebAssembly project. We also set up an `OnNavigateAsync` event, and in that method, check the path and make sure to load the assemblies we need.

`LazyAssemblyLoader` can also be used for routable components by doing something similar to this:

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject LazyAssemblyLoader AssemblyLoader
@inject ILogger<App> Logger
<Router AppAssembly="@typeof(App).Assembly"
    AdditionalAssemblies="@lazyLoadedAssemblies"
    OnNavigateAsync="@OnNavigateAsync">
    ...
</Router>
@code {
    private List<Assembly> lazyLoadedAssemblies = new();
    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path == "{PATH}")
            {
                var assemblies = await
                    AssemblyLoader.LoadAssembliesAsync(
                    new[] { {LIST OF ASSEMBLIES} });
                lazyLoadedAssemblies.AddRange(assemblies);
```

```
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}", ex.Message);
        }
    }
}
```

We need to replace `{PATH}` in the preceding snippet with the path where we want to load the assemblies, e.g., `"/fetchdata"`. The `{LIST OF ASSEMBLIES}`, which contains a list of assemblies we wish to load, can be `"sample.dll"`.

This makes it possible, for example, not to load the admin interface for users who don't have access to it. We can, of course, trigger the download of additional assemblies when it makes sense (and not wait for a user to hit a specific part of the application before downloading them).

Progressive web apps

A **Progressive Web App** (**PWA**) is a web application that can behave like a native app; it can be installed on a device, work offline, and provide a more app-like experience.

Blazor WebAssembly can create PWAs. PWAs make it possible to download the web as an app to your phone or computer. They will make it possible to add nice-looking icons and launch our web in a web browser without a URL input field, so it will feel more like an app.

When creating our project, we select PWA. This adds the necessary configuration, including a web app manifest (for app metadata like icons and name) and a service worker (for caching and offline support), along with supporting JavaScript.

PWAs are beyond the scope of this book, but there are great resources to get us started. You can find more information here: `https://learn.microsoft.com/en-us/aspnet/core/blazor/progressive-web-app?view=aspnetcore-10.0&tabs=visual-studio`.

It is worth mentioning that WebAssembly can run PWAs in offline mode, which is why it is less common to see Blazor Server being used as a PWA, but we can still use the same config to get the app feeling.

Native dependencies

Since we are running WebAssembly, we can use WebAssembly assemblies written in other languages in our project. This means that we can use any native dependencies right inside our project.

One way is to add C files right into our project. In the `Chapter19` folder in the repository, you will find an example. I have added a file called `Test.c` with the following content:

```
int fact(int n)
{
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

This C function calculates a factorial, meaning it multiplies a number by all the numbers below it (for example, 5 becomes 5 × 4 × 3 × 2 × 1). It uses recursion, calling itself with a smaller number until it reaches 0.

In the project file, I have added a reference to that file:

```
<ItemGroup>
    <NativeFileReference Include="Test.c" />
</ItemGroup>
```

In `Home.razor`, I have added the following code:

```
@page "/"
@using System.Runtime.InteropServices
<PageTitle>Native C</PageTitle>
<h1>Native C Test</h1>
<p>
    @@fact(3) result: @fact(3)
</p>
@code {
    [DllImport("Test")]
    static extern int fact(int n);
}
```

In our C# project, we now have a C file that we can call from our Blazor project. It is compiled into WebAssembly, and then we can reference that WebAssembly file (which happens automatically). We can take this even further by using a library that is using a C++ library.

Skia is an open-source graphics engine written in C++, which you can read more about here: `https://github.com/mono/SkiaSharp`. We can add that library to a Blazor WebAssembly app by adding the NuGet package `SkiaSharp.Views.Blazor`.

In the `Chapter19` folder in the repository, you can explore a project called `SkiaSharpDemo`.

There, in the `Home.razor` file, I have added the following code:

```
<SKCanvasView OnPaintSurface="@OnPaintSurface" />
@code {
    private void OnPaintSurface(SKPaintSurfaceEventArgs e)
    {
        var canvas = e.Surface.Canvas;
        canvas.Clear(SKColors.White);
        using var paint = new SKPaint
        {
            Color = SKColors.Black,
            IsAntialias = true,
            TextSize = 24
        };
        canvas.DrawText("Raccoons are awesome!", 0, 24, paint);
    }
}
```

The page will draw `"Raccoons are awesome"` on the canvas.

Because this example relies on native dependencies, you might need to restore the required workload before running the project:

```
dotnet workload restore
```

In this case, we are using a C# library that is using a C++ library.

We can even refer to libraries that have already been built with Emscripten directly by adding **object files** (`.o`), **archive files** (`.a`), **bitcode** (`.bc`), and **standalone WebAssembly modules** (`.wasm`). If we find a library written in another language, we could compile that to WebAssembly and then use it from our Blazor application. This opens up so many doors!

We have now looked at several advanced aspects of Blazor WebAssembly, from performance optimizations to working with native dependencies. While these features open up many possibilities, they can also introduce new challenges.

Next, we will look at some common problems I have encountered.

Common problems

The most common concerns regarding Blazor WebAssembly are download size and load time. While a small project is often around 1 MB and benefits from caching, the initial load time is usually the bigger concern, since all required files must be downloaded before the app can start. That said, download size can still matter, especially for first-time visitors or users on slower connections.

There are a couple of ways to improve this experience.

Progress indicators

When it comes to **User Experience** (**UX**), we can give users a perceived sense of speed.

The default Blazor WebAssembly template has a loading progress indicator, which gives the users something to look at instead of a blank page. It is built so that it is easy to customize using CSS variables. We can use the `--blazor-load-percentage` and `--blazor-load-percentage-text` variables to customize and create our progress bar.

The progress bar doesn't even have to indicate what is happening; Dragons Mania Legends has comments like "Sewing mini Vikings," which is obviously not what is going on. So, depending on the application we are building, showing something is more important than showing nothing.

Prerendering

Prerendering is the process of generating the initial HTML for a page on the server (or at build time) before the WebAssembly app has fully loaded in the browser.

Prerendering is my favorite feature, coming right out of the box when using the Blazor Web App template. This is why I usually recommend using it. It solves the wait time by showing content immediately, while the interactive WebAssembly app loads in the background using the Auto render mode.

But what if you want to use WebAssembly with prerendering? Well, the simple way is to use the Blazor Web App template, but you might have limitations—you can't or don't want to run the site on an ASP.NET server. Then you can prerender on publish. This doesn't work for dynamic content, but it can still make the page feel much faster while WebAssembly is downloading in the background.

Check out `https://github.com/jsakamoto/BlazorWasmPreRendering.Build`.

It will crawl your site and generate HTML prerendered content when publishing, so loading feels super fast. I have a couple of sites hosted on GitHub Pages that use this. It's easy to use and makes a big difference.

Summary

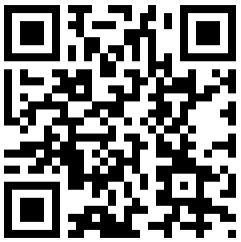
In this chapter, we looked at some of the Blazor WebAssembly-specific things in Blazor. For the most part, we can reuse components in both Blazor Server and Blazor WebAssembly, and we can speed up WebAssembly by using what we learned in this chapter.

We also looked at native dependencies, opening up the possibilities to reuse other libraries and mix languages. If our application doesn't need to support both scenarios, we can use WebAssembly to the fullest.

In the next chapter, we will examine *source generators*.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

20

Examining Source Generators

In this chapter, we will look at writing code that generates code using source generators. This isn't a Blazor-specific feature, but it's used behind the scenes in Blazor and is something we can take advantage of in our own applications. The subject of source generators is a book in itself, but I wanted to introduce them since they help reduce repetitive coding, and honestly, they're one of my favorite features. I am the kind of person who will spend a whole day writing source code to avoid a repetitive task, even if that task only takes 10 minutes each time. Repetitive tasks have never been my favorite.

In this chapter, we will cover the following:

- What a source generator is
- How to get started with source generators
- Community projects

The idea for this chapter is for you to use it as a reference so that you can implement a new project on your own.

Technical requirements

This chapter is a reference chapter and is not connected in any way with the book's other chapters.

You can find the source code for this chapter's result at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter20`.

What a source generator is

In many cases, we find ourselves writing the same kind of code repeatedly. In the past, I have used T4 templates to generate code, and I have even written stored procedures and

applications that help me generate code. But now, **Source generators** are part of the .NET compiler platform (Roslyn) SDK.

A generator gives us access to a compilation object representing all the user code currently being compiled. From there, the object can be inspected, and we can, based on that, write additional code.

Okay, this sounds complicated, and I would be lying if I said it was easy to write a source generator, but it instantly saves us a lot of time. So, let's break it down a bit.

When we compile our code, the compiler does the following steps:

1. The compilation runs
2. Source generators analyze code
3. The source generators generate new code
4. The compilation continues

Steps 2 and *3* are what source generators do.

In Blazor, source generators are used all the time; they take `.razor` files and convert them to C# code.

We can look at what Blazor generates by adding the following to our `.csproj` file:

```
<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
```

Adding this code will emit generated files into the `obj` folder for the `razor` component.

We can find them here:
`\obj\Debug\net10.0\generated\Microsoft.NET.Sdk.Razor.SourceGenerators\Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator`

We can choose where to emit the files by using:

```
<CompilerGeneratedFilesOutputPath>THEPATH
    </CompilerGeneratedFilesOutputPath>
```

You can replace `THEPATH` with the path you would like the files emitted to.

In the folder the files are emitted to, we can find a file called `Pages_Counter_razor.g.cs`, which is the C# representation of the counter component. This file is generated by the `Microsoft.NET.Sdk.Razor.SourceGenerators` generator, a built-in and quite advanced source generator that converts Razor components into C# during the build process.

Let's think of a scenario: at work, we create services and interfaces for those services. The only use of these interfaces is for testing purposes, the same way we have built our repositories throughout the book.

In this case, adding a method to a service means we need to add the method to the class and the interface. We tried simplifying the process by putting the interface and the class in the same file. However, we still forgot about the interface, pushed the code, and didn't notice the mistake until everything was built and a NuGet package was generated.

Luckily, we found a source generator called **AutomaticInterface**. Adding an attribute to our class will generate the interface for us.

Let's take a look at this example:

```
public class SampleService
{
    public double Multiply(double x, double y)
    {
        return x * y;
    }
    public int NiceNumber => 42;
}
```

This is a simple service class. By adding the `GenerateAutomaticInterface` attribute, the code will automatically generate an interface. We can then add a reference to that interface:

```
[GenerateAutomaticInterface]
public class SampleService: ISampleService
…
```

Now the generated interface will always be up to date. This is an excellent example of when source code generators save time and remove pain points.

Source generators are powerful. Using them, we can get access to a syntax tree that we can query. We can also iterate over all classes and find the ones with a specific attribute or that implement an interface, for example, and based on that, generate code.

However, there are some limitations. There is no way to know in what order the source generators will run, so we can't generate code based on generated code. We can only add code, not modify code.

That said, they are still a great way to remove repetitive code and push work to compile time instead of runtime.

If we need to control the order, one workaround is to split things into separate projects. Since dependencies are built before the project that uses them, we can use that to influence the order in which source generators run.

The following section will look at how we can build our source generators.

How to get started with source generators

It's time to look at how we can build our source code generators. The `Chapter20` folder is a finished example of what we discuss here. The instructions here won't be a step-by-step guide, but more of a walkthrough of the key concepts and how everything fits together.

To create a source generator, we first need to set up a class library targeting *.NET Standard 2.0*. In that project, we also need to add references to the NuGet packages `Microsoft.CodeAnalysis.CSharp` and `Microsoft.CodeAnalysis.Analyzers`, and ensure that the project file includes `<LangVersion>latest</LangVersion>`.

Once the project is set up, we can create our source generator. A source generator is simply a class that:

- Has the `[Generator]` attribute
- Implements `IIncrementalGenerator`

The template code should look something like this:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Text;
using System.Text;

namespace SourceGenerator;

[Generator]
public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(
        IncrementalGeneratorInitializationContext context)
    {
        context.RegisterPostInitializationOutput(static ctx =>
        {
            //Source Generation goes here
        });
```

```
    }
}
```

In the `Initialize` method, we add any initialization we may need and hook up the code used for generation. The generator we are building now is, of course, a silly example, but it also shows some of the power of source generators.

Then, in the `RegisterPostInitializationOutput`, we add the following code:

```
// Build up the source code
string source = """
namespace BlazorWebAssembly;
public class GeneratedService
{
    public string GetHello()
    {
        return "Hello from generated code";
    }
}
""";
// Add the source code to the compilation
ctx.AddSource("GeneratedService.g.cs",
    SourceText.From(source, Encoding.UTF8));
```

It will take the code in the source variable and save it as `GeneratedService.g.cs`. We also use raw string literals in this file – this is the feature in .NET 7 I have been the most excited about. By adding three double quotes, we don't need to escape the string; we are free to add more double quotes inside the string. If you want to escape more than three double quotes, you can add more at the start and end.

To add a source generator to our project, we can add the project like this:

```
  <ItemGroup>
    <ProjectReference
        Include="..\SourceGenerator\SourceGenerator.csproj"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false"/>
  </ItemGroup>
```

When we compile our project, the `GeneratedService` will be generated and we can use the code.

Now we can inject the service and use it inside of our components (this is from `Home.razor`):

```
@page "/"
@inject GeneratedService service
<h1>@service.GetHello()</h1>
```

Don't forget to add it to `Program.cs` as well:

```
builder.Services.AddScoped<GeneratedService>();
```

This example isn't really useful in a real-world scenario, but it shows that getting started with source generators isn't that complicated.

In a real-world scenario, source generators are often used to scan existing code (for example, classes with specific attributes or interfaces) and generate supporting code such as mappings, APIs, or boilerplate logic. We won't go deeper into that here, but in the next section, you'll find some great resources and community projects that demonstrate more practical use cases.

Sometimes the Visual Studio editor won't pick up these generated files, and we will see some red squiggles in the code editor. This is often due to how source generators run during compilation; the order is not guaranteed, and the editor doesn't always reflect generated code immediately, even though the project builds correctly.

If this happens, try building the project to verify that everything compiles as expected. You can also inspect the generated files directly in the output folder (for example, under the `obj` directory if you have enabled generated file output). In some cases, restarting Visual Studio can also help the editor catch up with the generated code.

There are limitations: if a razor depends on a generator, we might run into problems. The solution is to make sure you break apart the source generation. If we need to have a source-generated file and use it in a Razor file, move the source generation to a separate project. This way, we will ensure the dependencies are generated first.

In the next section, we will look at some of the source generators we can use in our projects.

Community projects

Source generators have been around since .NET 5/6, and there are a lot of community/open-source projects we can use in our projects. Let's explore them in the following sections.

AutomaticInterface

We have already talked about `AutomaticInterface`. Generating interfaces without having to write the same thing twice will save time and help you avoid problems, especially if you use interfaces only for testing.

We can find it here: `https://github.com/codecentric/net_automatic_interface`.

Blazorators

David Pine, with many contributors, has built Blazorators which can take a TypeScript definition file and generate JavaScript interop, ready to be used in any Blazor project. Blazorators takes away a lot of the pain points when writing JavaScript interop.

Check out his project here: `https://github.com/IEvangelist/blazorators`.

C# source generators

Amadeusz Sadowski, with many contributors, has made an impressive list of where to find more information on source generators and some outstanding ones. You can find this fantastic resource here: `https://github.com/amis92/csharp-source-generators`.

Roslyn SDK samples

Microsoft has added some samples to their Roslyn SDK repository. It's a great start to dig a bit deeper into source generators. You can find the samples here: `https://github.com/dotnet/roslyn-sdk/tree/main/samples/CSharp/SourceGenerators`.

Microsoft Learn

Microsoft Learn is an excellent source for learning anything C# related, and source generators are no exception. If you think, just like me, that source generators sound like the best thing since sliced bread, I recommend that you dive into the documentation found at Microsoft Learn: `https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview`.

Summary

In this chapter, we looked at code that writes code to save time and reduce repetitive tasks.

Blazor uses source generators to convert Razor code into C# code, so, indirectly, we are using them all the time.

In the next chapter, we will look at Blazor Hybrid by visiting .NET MAUI.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/8VuShAAtK`

21

Visiting .NET MAUI

So far, we have talked about Blazor WebAssembly and Blazor Server.

In this chapter, we will visit .NET MAUI, Microsoft's cross-platform development platform. We'll look at how to create applications that run on Android, iOS, macOS, and Windows, and explore how the templates and project structure work.

We'll also look at Blazor Hybrid and how we can reuse our existing Blazor knowledge to build native apps. The goal is not to go deep into MAUI, but to give you enough understanding to get started and see where it fits in your toolbox.

This chapter will not be a deep dive into .NET MAUI, since that can be a book all in itself.

In this chapter, we will cover the following:

- What is .NET MAUI?
- Creating a new project
- Looking at the template
- Developing for Android
- Developing for iOS
- Developing for macOS
- Developing for Windows

The idea for this chapter is for you to use it as a reference so that you will be able to implement a new project on your own.

Technical requirements

This chapter is a reference chapter and is not connected in any way with the book's other chapters.

You can find the source code for this chapter at `https://github.com/PacktPublishing/Web-Development-with-Blazor-4E/tree/main/Chapter21`.

What is .NET MAUI?

We'll start with a bit of history.

Xamarin is a software company founded in May 2011 by the engineers who created Mono, a free and open-source version of the .NET Framework. Microsoft acquired the company in 2016, and it is now a vital part of the .NET development platform, providing tools and services for building native cross-platform mobile apps using C# and .NET.

Xamarin's technology allows developers to write native iOS, Android, and Windows apps using a single shared code base, making it easier to develop and maintain apps for multiple platforms.

.NET Multi-Platform App UI (**MAUI**) is the new framework from Microsoft, which is an evolution of Xamarin.Forms. The framework provides a way to create one UI, deploy it to many different platforms, and get native controls on each platform.

.NET MAUI can also host Blazor, which is called Blazor Hybrid. This way we can render Blazor content inside of a .NET MAUI app, using the same controls and code that we build for the web. The controls that are rendered using Blazor Hybrid are web controls, so we will not get the native controls. We can, however, mix native and Blazor Hybrid content.

This whole "native vs shared" discussion isn't new, and I've had some strong opinions about it for a while.

Many years ago, I sat in a meeting with a bunch of consultants. The company I was working for wanted to invest in an app, and we turned to one of the big consultancy firms in Sweden to get some help on how we should proceed.

After a week, we had another meeting, during which they presented their findings. Their recommendation was to build natively and not use any of the cross-platform frameworks.

They had a bunch of arguments, but two that really stuck with me are as follows:

- Native apps look better and give the user a "real" device experience
- Shared code (between platforms) means that if one platform has a bug, the same bug is now in all platforms

Since .NET MAUI (formerly Xamarin.Forms) uses native controls, there is no way for the users to know the difference between developing a native app and developing using .NET MAUI. In the end, it will look and feel like a native app. This is not true for Blazor Hybrid, which uses web controls. So, there are some valid arguments for the first point. Now, going back to the first

argument, that native apps look better and provide a more *real* device experience, we must ask ourselves how important that really is. Looking at the apps on my iPhone, not many apps look the same, and as long as we provide a good UX, the exact look and feel matters less. Don't get me wrong, a common UX is still important, but if the button is blue or native black, it doesn't really matter.

The second argument made me so angry. Were they trying to convince us that sharing code was terrible? Yes, they were. Sharing code between platforms is fantastic; you only need to write the code once, fix a bug once, and fix it on all platforms.

.NET MAUI gives us both options. We can use native UI with C# code or use Blazor Hybrid to get web controls.

Creating a new project

To develop cross-platform applications, we must install cross-platform tools in Visual Studio.

If you haven't done that, please open the Visual Studio installer and select the **.NET Multi-Platform App UI** development workflow.

.NET MAUI templates

.NET MAUI has a few templates: **.NET MAUI App**, **.NET MAUI Class Library**, **.NET MAUI Blazor App**, and **.NET MAUI Blazor Hybrid and Web App**. Let's take a look at each one.

.NET MAUI App

The .NET MAUI template uses XAML to create applications.

XAML is also used for **Windows Presentation Foundation** (**WPF**) and **Universal Windows Platform** (**UWP**). Every XAML version differs just a bit, but if you have worked with **WPF** or **UWP** before, they should feel familiar.

The XAML is converted into native elements. This way, if our app runs on Windows, it will have the look and feel of a Windows application. If we run it on an iOS device, it will look and feel like a native iOS app.

This is probably our best option if we want to use our C# skills to create a cross-platform application. Using this approach, we will get the native feel without the need to write native code in Kotlin or Swift.

.NET MAUI Class Library

.NET MAUI Class Library is used to share content, classes, and functionality between applications. It is typically used to keep shared logic, services, and models in one place, making it easier to reuse code across multiple MAUI apps or even other .NET applications.

.NET MAUI Blazor Hybrid app

Since this is a book about Blazor, we will focus on the `.NET MAUI Blazor Hybrid App` template. This is a template that embeds a Blazor application inside a native shell.

For the .NET MAUI Blazor App project, we need at least:

- Android 7.0 (API 24) or higher
- iOS 14 or higher
- macOS 11 or higher, using Mac Catalyst

The .NET MAUI Blazor Hybrid App project uses BlazorWebView to render the Blazor content. It is not the same as Blazor Server and does not run WebAssembly. Instead, it is a third way of running Blazor applications, with a different execution model where the code runs locally inside the native app through a WebView.

.NET MAUI Blazor Hybrid app and Web App

This is similar to the .NET MAUI Blazor Hybrid App template, but with a web project already connected. This means we can share components between the web and native app right away, making it a great option if we want to support both platforms using the same code.

Starting a project

As mentioned, we will be using the `.NET MAUI Blazor Hybrid App` template.

Let's start a new project and dig a bit deeper:

1. In Visual Studio, create a new `.NET MAUI Blazor Hybrid App` project.
2. Name the project `BlazorHybridApp` and make sure you select **.NET 10**.
   Skip the Aspire orchestration.
3. At the top of Visual Studio, select **Windows Machine** and run the project.

That's it. We now have our first cross-platform Blazor Hybrid app!

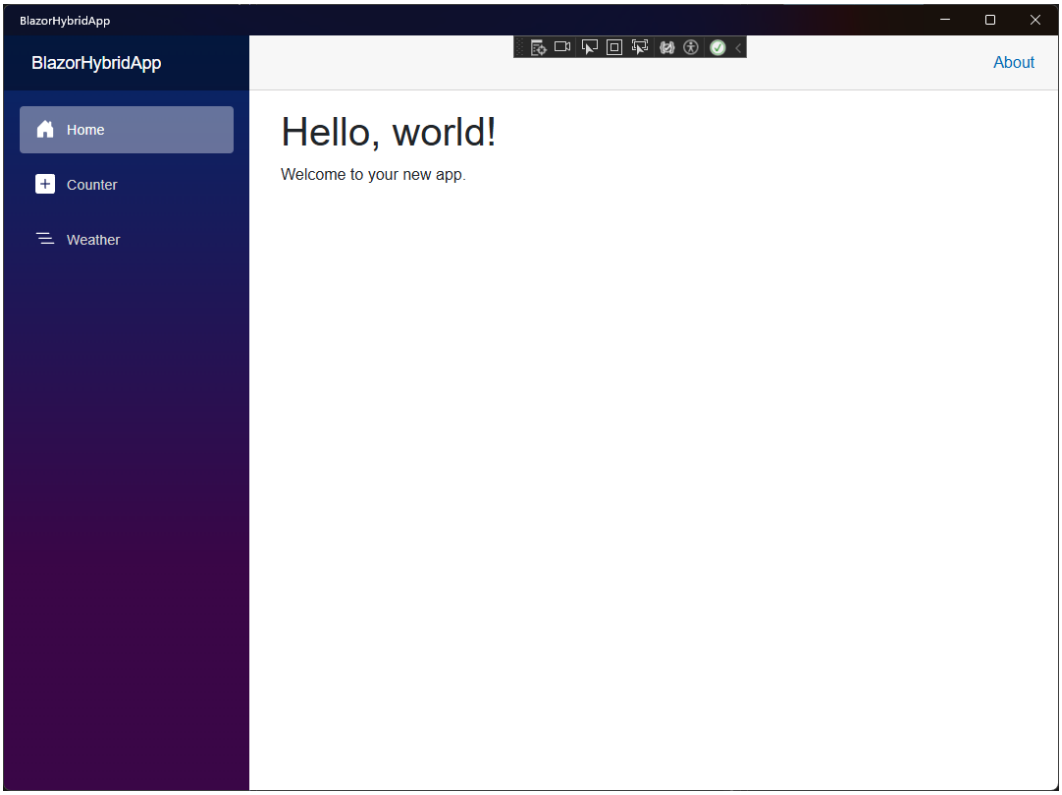


Figure 21.1: .NET MAUI app running on Windows

We might need to enable developer mode on our machine. If there is a message asking us to, just follow the instructions and run the app again.

Great! We now have a project. In the next section, we will take a look at what the template looks like.

Looking at the template

When running the project, we should recognize the UI. It is the same *Hello, world!* page, the same counter, and the same weather forecast as we have seen in the other templates we have looked at in the book.

If we take a look in the `Components/Pages` folder, we'll find the Razor components, and if we open the `Counter.razor` file, we will find a familiar component that looks like this:

```
@page "/counter"
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">
    Click me</button>
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

To create a Blazor Hybrid app, adding components like this is all that you need to know to get started, but let's dig a bit deeper. The template is a .NET MAUI App with some added Blazor startup code.

To understand what is happening, we will start in the `Platforms` folder. In the `platforms` folder, we will find different folders for each platform we can develop for: Android, iOS, Mac Catalyst, and Windows.

This is the starting point for each platform, and they have slightly different implementations, but in the end, they all point to the `MauiProgram` file located at the project's root.

The `MauiProgram` class sets everything up, like fonts and dependency injection:

```
namespace BlazorHybridApp;
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            });
        builder.Services.AddMauiBlazorWebView();
```

```
#if DEBUG
        builder.Services.AddBlazorWebViewDeveloperTools();
        builder.Logging.AddDebug();
#endif
        return builder.Build();
    }
}
```

The essential thing in the file is `UseMauiApp<App>`, which gives us a clue about the next step: loading the `App.xaml` file.

Then, in the root we will find the `App.xaml` file, which has a bunch of resources for styling, but the Blazor magic starts to happen in `App.xaml.cs`:

```
namespace BlazorHybridApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
        }

        protected override Window CreateWindow(
            IActivationState? activationState)
        {
            return new Window(new MainPage()) {
                Title = "BlazorHybridApp" };
        }
    }
}
```

This creates a new Window containing the page `MainPage`.

In `MainPage.xaml`, we have reached the first Blazor reference in the app, the `BlazorWebView`:

```
<BlazorWebView x:Name="blazorWebView" HostPage="wwwroot/index.html">
    <BlazorWebView.RootComponents>
        <RootComponent Selector="#app"
            ComponentType="{x:Type local:Components.Routes}" />
```

```
    </BlazorWebView.RootComponents>
</BlazorWebView>
```

In this case, we are referring to `index.html` in the `wwwroot` folder, which serves as the host page. We then configure the root component, which tells Blazor which component to render and which element in the page it should be rendered into, similar to how this is set up in `Program.cs` for Blazor Server and Blazor WebAssembly.

Here, we can also add XAML components, which makes it possible to mix XAML and Blazor components. Even though the implementation looks different, we should be familiar with the concepts.

The `index.html` is almost the same as in Blazor WebAssembly:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
        maximum-scale=1.0, user-scalable=no, viewport-fit=cover" />
    <title>BlazorHybridApp</title>
    <base href="/" />
    <link rel="stylesheet" href="lib/bootstrap/dist
        /css/bootstrap.min.css" />
    <link rel="stylesheet" href="app.css" />
    <link rel="stylesheet" href="BlazorHybridApp.styles.css" />
    <link rel="icon" href="data:,">
</head>

<body>

    <div class="status-bar-safe-area"></div>

    <div id="app">Loading...</div>

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

```
    <script src="_framework/blazor.webview.js"
        autostart="false"></script>

</body>

</html>
```

The only difference worth mentioning is the JavaScript that differs from the others (Blazor Server and Blazor WebAssembly implementations). From this point, the application is now running pure Blazor.

To see how this is wired up, we can look at `MainPage.xaml`. Here, we are loading a Razor file called `Routes`. This is a familiar name from the Blazor Web App template. It looks like this:

```
<Router AppAssembly="typeof(MauiProgram).Assembly"
    NotFoundPage="typeof(Pages.NotFound)">
    <Found Context="routeData">
        <RouteView RouteData="routeData"
            DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

This is where we find the router, configure where to find the Razor components, and handle the requests that are not found.

We will not go deeper into the Blazor parts because everything past our router is the same as any other Blazor hosting model (Blazor Server and Blazor WebAssembly). There is a `MainLayout`, `NavMenu`, and component for each function (`Hello, world!`, `Counter`, and `Weather`).

With Blazor Server and Blazor WebAssembly, we can access native things, but we need to make JavaScript calls to access local resources like Bluetooth, a battery, and a flashlight, to name a few. Blazor Hybrid adds the ability to write code that directly accesses local resources. We can access the flashlight (because we all love things that light up) by using code similar to this:

```
try
{
    if (FlashlightSwitch.IsToggled)
        await Flashlight.Default.TurnOnAsync();
    else
        await Flashlight.Default.TurnOffAsync();
```

```
}
catch (FeatureNotSupportedException ex)
{
    // Handle not supported on device exception
}
catch (PermissionException ex)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to turn on/off flashlight
}
```

This code will not work if we run a Blazor Server or Blazor WebAssembly app. If we still want to share components between .NET MAUI and Blazor web apps, we can do that using dependency injection, just like we have done a couple of times in the book already, including once for the web and once for mobile.

Next, we will get our amazing app to run on Android.

Developing for Android

There are two options when it comes to developing for Android. We can run our application in an *emulator* or on a *physical device*. We will review both options.

To publish our application, we need to have a Google Developer license, but we don't need one for development and testing.

Running in an emulator

To begin, we need to install an emulator to run our app on an Android emulator:

1. In Visual Studio, open **Tools | Android | Android Device Manager**.
2. Click **New**, and configure a new device (the default settings should be fine):

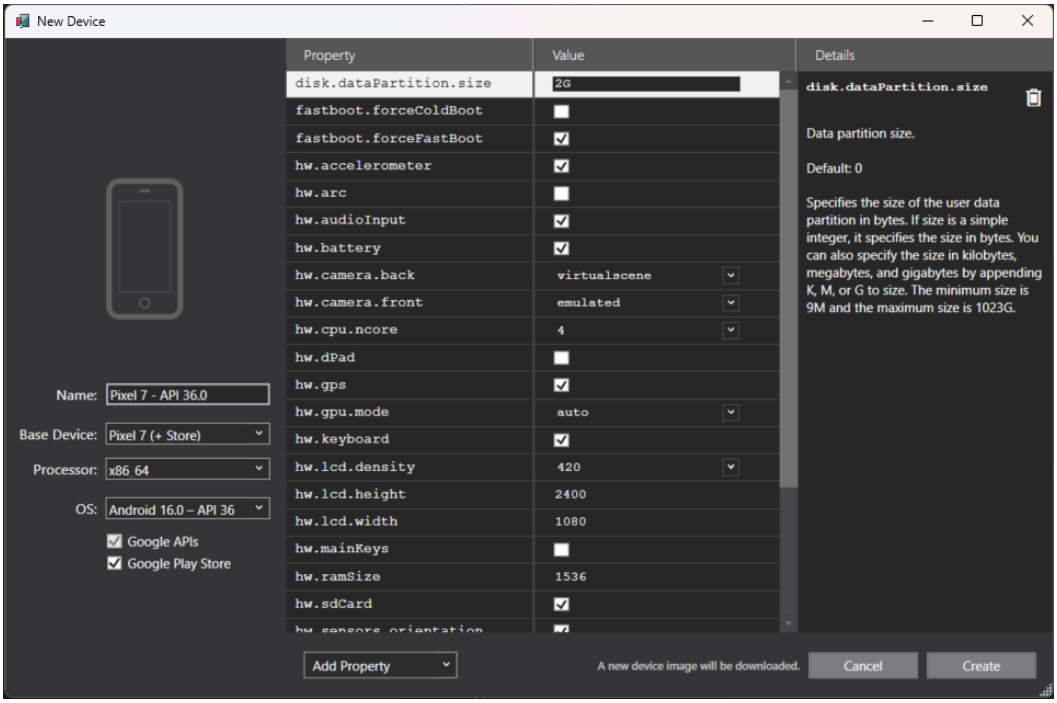


Figure 21.2: Android device configuration

3. Click **Create** to download a device image and configure it.
   Accept any agreements.

4. Select the newly created emulator at the top of Visual Studio and run the project.
   Starting the emulator can take a couple of minutes. For faster development, avoid closing it between runs, since restarting the emulator adds significant overhead.

To get the emulator to run fast, we can enable hardware acceleration, depending on the processor we use. To enable hardware acceleration, please refer to the official documentation: `https://learn.microsoft.com/en-us/dotnet/maui/android/emulator/hardware-acceleration?view=net-maui-10.0`.

Great! We now have our app running inside an Android emulator:

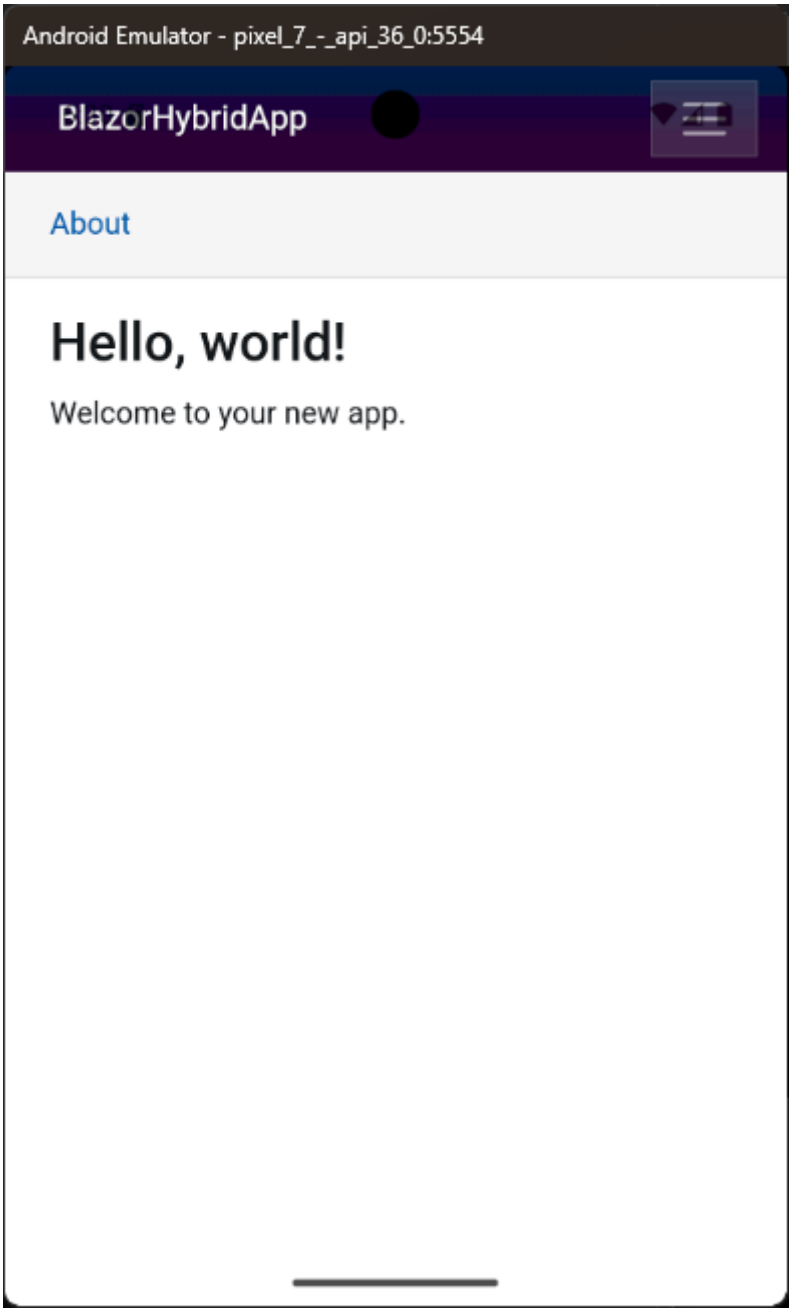


Figure 21.3: App running inside an Android emulator

Next, we will run the application on a physical device.

Running on a physical device

If we want to try our application on a physical device, we need to do a few things on our Android device. This may differ from device to device.

First, we need to make sure the phone is developer-unlocked:

1. Go to the **Settings** screen
2. Select **About phone**
3. Tap **Build Number** seven times until the message **You are now a developer!** is visible

Second, we need to enable USB debugging:

1. Go to the **Settings** screen
2. Select **Developer options**
3. Turn on the **USB debugging** option
4. Some devices also need to enable **Install via USB**

We are now all set to try our app on a physical device:

1. Connect your device to the computer using a USB cable
2. Unlock your phone and allow communication with the computer (if such a question shows up)
3. In the menu at the top of Visual Studio, click the arrow under **Android local devices** and select your device
4. Press **Run**, and Visual Studio will deploy the application to the device

We should now have our application running on our device.

It is an extraordinary feeling to run code on another device. Over the years, I have developed over 100 applications for Windows 8 and Windows Phone. However, to this day, it still gives me the same feeling to see my application deploy to another physical device.

Next, we will look at what options we have for developing for iOS.

Developing for iOS

Apple does not allow iOS code to be compiled on something that is not an Apple computer. To get around this, there are cloud options like MacinCloud and MacStadium, but we won't go into those options in this book. I haven't tested them myself, so I don't know if they work well or not.

A Mac with Xcode is sufficient for the simulator; Apple Developer membership is required for physical device provisioning and distribution.

Before that, though, to enable our iOS device to work, we need to set it to Developer Mode:

1. Open up your iPhone's **Settings** app.
2. Scroll down and tap **Privacy & Security**.
3. Look for the **Developer Mode** toggle and turn it on. If you can't find it, you might need to connect your phone to Xcode. Where to find the developer mode differs between versions of the operating system, but ask Google or Bing for help. There are many resources to be found on how to solve it with your version.
4. Your iOS device might give you a heads-up about enabling **Developer Mode**, potentially making your device a bit less secure. No worries, just tap **Restart** to move forward.
5. Once your device reboots, unlock it. You'll see another alert asking if you're sure about enabling **Developer Mode**. Tap **Turn On**, and if it asks for your passcode, enter it.

> **Note**
>
> **Hot restart**
>
> Hot restart used to be an option for testing on an iOS device, but it is no longer supported in Visual Studio 2026. Since this used to be possible, I wanted to mention it in case you run into older blog posts or docs that still talk about Hot Restart.

Next, we will look at how to set up a simulator.

Simulator

A simulator runs the app on a Mac but shows the result on a PC.

A simulator differs from an emulator in that an emulator runs the code on the machine (in our case, a PC), while a simulator runs on top of the native OS (macOS), mimicking an iPad or an iPhone.

To use iOS simulators, we need an Apple computer on the same network, as the simulator runs on macOS. Visual Studio will help us with the connection, but we first need to prepare the Mac.

On the Mac, install Xcode from the App Store. After installation, open Xcode to accept the license agreement and complete the setup, including selecting the devices you want to develop for.

We also need to open remote access to the Mac. We can do that by doing the following:

1. On the Mac, invoke Spotlight by pressing *cmd* + *space*, searching for **remote login**, and opening the **Sharing System Preferences**
2. Enable the **Remote Login** option to allow Visual Studio to connect to the Mac
3. Set access for **Only these users** and ensure your user is included in the list or group

We now have everything prepared on the Mac. In Visual Studio on the PC, we can now pair our Mac:

1. Select **Tools | iOS | Pair to Mac**.
2. Follow the instructions in the wizard.
3. Select your Mac from the list, and click **Connect**.
   Visual Studio can now help you install the things you need to get started. It might take a while for the Mac to install everything, so if it doesn't work, the simulators are probably not installed yet.
4. In the dropdown at the top of Visual Studio, we can select **iOS Simulators** and then choose a device to run our app.

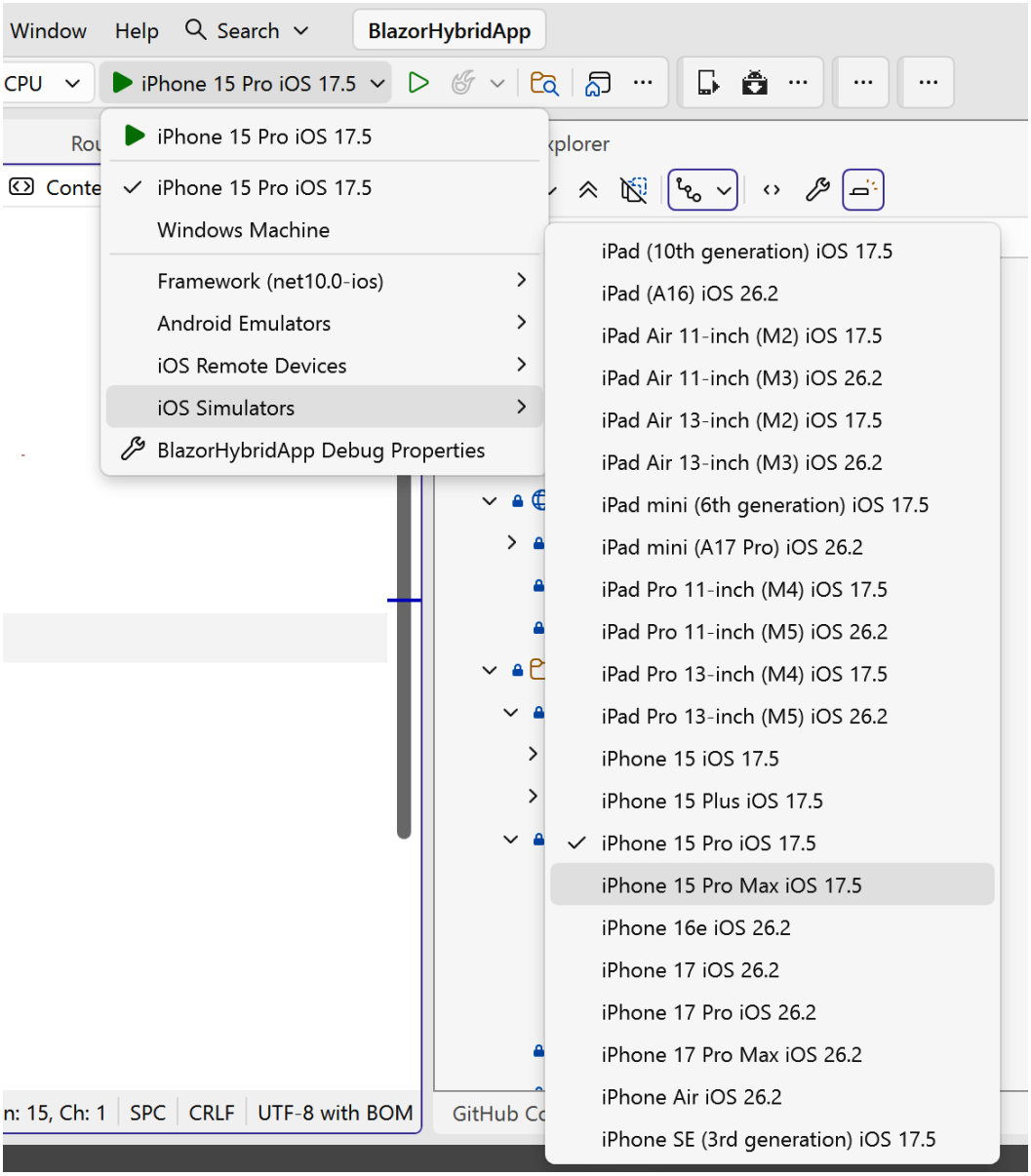


Figure 21.4: Device selection in Visual Studio

5. Run the app, and the simulator will start. This is what the app would look like if we ran it on an iPhone:

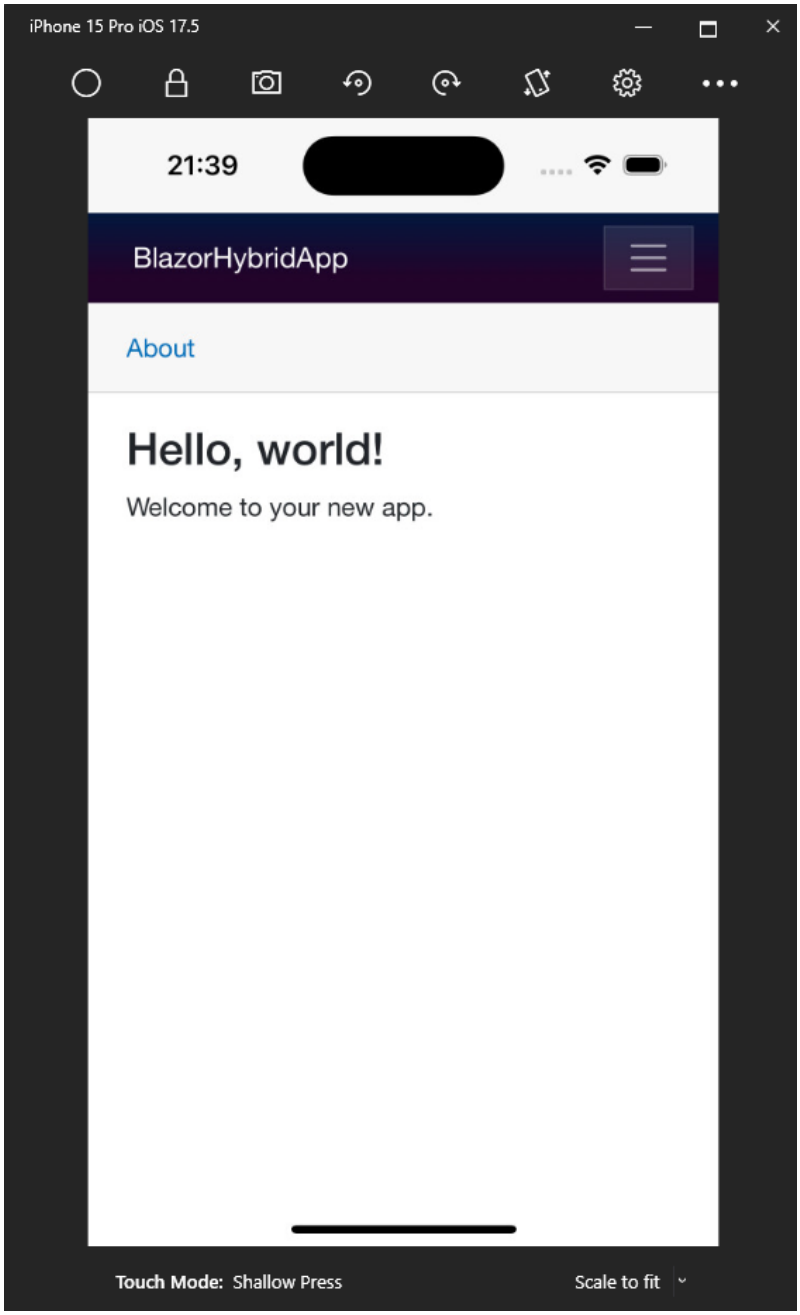


Figure 21.5: App running in iPhone simulator

We now have a way of running and testing on iOS devices. We can also connect an iPhone directly to the Mac and run the application over Wi-Fi. There is more information on debugging over Wi-Fi in the official docs: `https://learn.microsoft.com/en-us/xamarin/ios/deploy-test/wireless-deployment`.

Next, we will build an app for macOS.

Developing for macOS

We don't have an option for macOS to run or deploy from a Windows machine. To run our application on the Mac, follow these steps:

1. On the Mac, open our project in VS Code.
2. This is the best source to keep up to date with installing the tools on a Mac: `https://learn.microsoft.com/en-us/dotnet/maui/get-started/installation?view=net-maui-10.0&tabs=visual-studio-code`.
   Follow the instructions to install the .NET SDK, .NET MAUI workload, and Xcode (including the required command-line tools). These components change over time, so it's best to rely on the official documentation for the latest setup steps.
3. Run the project, and our app will show up:

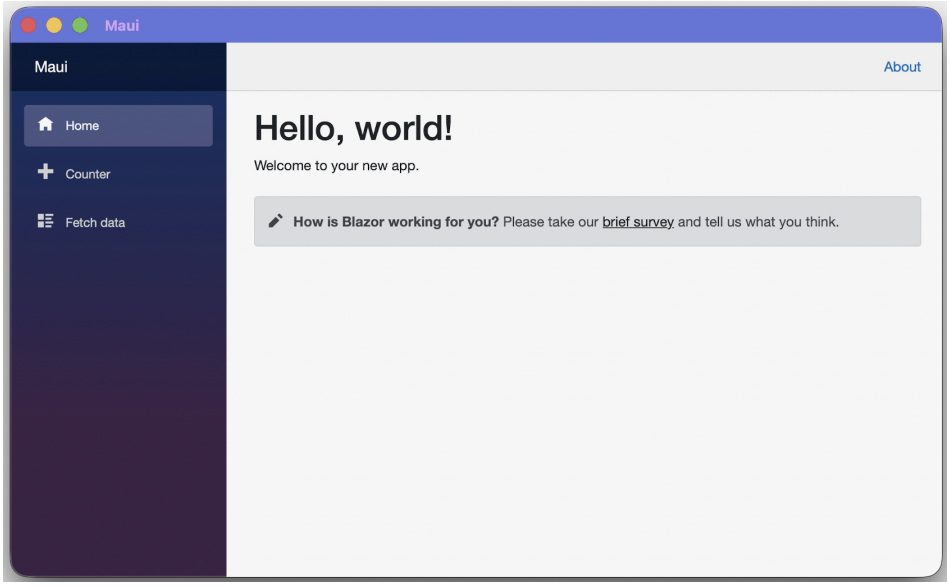


Figure 21.6: App running on macOS

In this case, we are running the application on the same platform, with no emulators or simulators, which is much less complicated than running it on a separate device.

Next, we will run our application on Windows.

Developing for Windows

Running the application on Windows is what we did in *step 3* of the *Starting a project* section. To reiterate, perform the following step:

1. Change the dropdown to **Windows Machine** and run the project. We can see the result in *Figure 21.1* at the beginning of the chapter.

As with macOS, we run the application on the same platform, with no emulators or simulators, which is much less complicated than running it on a separate device.

Summary

In this chapter, we looked at cross-platform development with Blazor Hybrid. I mentioned this before in this chapter, but it is worth mentioning again that running code on a phone or a device that is not a computer is such a fun thing to do. You can't beat that feeling. Even if you don't intend to develop for mobile devices, give it a try.

With .NET MAUI, we can leverage our existing C# knowledge and, perhaps more importantly, our Blazor knowledge to create mobile applications.

In the next chapter, we'll look at what happens when Blazor meets the real world. We'll go through lessons learned from running Blazor in production, covering things like memory, concurrency, and errors, and talk about where to go next.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

22

Where to Go from Here

The book is now coming to an end. Here, I want to leave you with some of the things we have encountered while running Blazor in production ever since it was in preview. We will also talk about where to go from here.

In this chapter, we will cover the following topics:

- Learnings from running Blazor in production
- The next steps

Learnings from running Blazor in production

Since Blazor was in preview, we have been running Blazor Server in production. In most cases, everything has run without issues. However, occasionally, we have encountered a few problems, and I will share those learnings with you in this section.

We will look at the following:

- Solving memory problems
- Solving concurrency problems
- Solving errors
- Old browsers

These are some of the issues we ran into, and we have solved them using approaches that have worked well for us in production.

Solving memory problems

Our latest upgrade added many users and, with that, a bigger load on the server. The server manages memory quite well, but with this release, the backend system was a bit slow, so users would need to press *F5* to reload a page. Then, the circuit would disconnect and a new circuit

would be created. The old circuit would wait for the user to connect to the server again for 3 minutes (by default).

The user would then have a new circuit and would never connect to the old circuit again, but for three minutes, the user's state would still take up memory. This is probably not a problem for most applications, but we are loading a lot of data into memory—the data, the render tree, and everything surrounding that will be kept in memory.

So, what can we learn from that? Blazor is a single-page application. Reloading the page is like restarting an app, which means we should always make sure to add a possibility to update the data from within the page (if that makes sense for the application). We could also update the data as it changes, as we did in *Chapter 13*, *Managing State – Part 2*.

In our case, we added more memory to the server and then made sure there were reload buttons in the UI that refreshed the data without reloading the whole page. The ultimate goal is to add real-time updates that continuously update the UI when the data changes.

If adding more memory to the server isn't an option, we can try to change the garbage collection from the server to the desktop. The .NET garbage collection has two modes:

- **Workstation** mode is optimized for running on a workstation that typically doesn't have a lot of memory. It runs the garbage collection multiple times per second.
- **Server** mode is optimized for servers where there is usually lots of memory and prioritizes speed, meaning it will only run the garbage collector every 2 seconds.

The mode of the garbage collector can be set in the project file or the `runtimeconfig.json` file by changing the `ServerGarbageCollection` node:

```
<PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Adding more memory is probably a better idea, though.

We have also noticed the importance of disposing of our database contexts.

Make sure to use `IDbContextFactory` to create an instance of the data context and, when we are done, dispose of it by using the `Using` keyword. Then the data context will only be available for a short time and then disposed of, freeing up memory fast.

Solving concurrency problems

We have often run into problems where the data context was already in use and couldn't access the database from two different threads.

This is solved by using `IDbContextFactory` and disposing of the data context when we are finished using it.

In a non-Blazor site, having multiple components to load at the same time is never a problem (because the web does one thing at a time), so the fact that Blazor can do multiple things at the same time is something we need to think about when we design our architecture.

Solving errors

Blazor usually gives us an error that is easy to understand, but in some rare cases, we do run into problems that are hard to figure out. We can add detailed errors to our circuit (for Blazor Server) by adding the following option in `Program.cs`:

```
builder.Services
.AddRazorComponents()
    .AddInteractiveServerComponents(options =>
    {
        options.DetailedErrors = true;
    });
```

By doing so, we will get more detailed errors. I don't recommend using detailed errors in a production scenario, however. With that said, we have the setting turned on for an internal app in production because the internal users are briefed on it and understand how to handle it. It makes it easier for us to help our users, and the error message is only visible in the developer tools of the web browser and not in the interface of the user.

Old browsers

Some of our customers were running old browsers on old systems, and even though Blazor supports all major browsers, that support doesn't include really old browsers. We ended up helping those customers upgrade to Edge or Chrome simply because we didn't think they should be browsing the web using browsers that no longer receive security patches.

Even our TV at home can run Blazor WebAssembly, so old browsers are probably not a big problem, but it can be worth thinking about when it comes to browser support. What browsers do we need/want to support?

I didn't want to remove this advice, but as time goes on, there are fewer and fewer old web browsers to worry about.

The next steps

At this point, we know the difference between Blazor Server and Blazor WebAssembly, and we know when to choose what, and picking one of them is not really that important. We know

how to create reusable components, make APIs, manage state, and much more. But where do we go from here? What are the next steps?

The community

The Blazor community is not as big as other frameworks but is growing fast. Many people share content with the community through blogs or videos. Dometrain and YouTube have a lot of tutorials and courses. Twitch has a growing amount of Blazor content, but it is not always easy to find in the vast content catalog.

There are a number of resources worth mentioning:

- *Jimmy Engström* – I wouldn't be much of a Blazor enthusiast if I didn't make it into my own list. I talk about Blazor and throw in a pun here and there. When we stream, we do that using *CodingAfterWork* (see below). My blog has a lot of Blazor content, with more to come: `https://engstromjimmy.com/`. You can also find me on X: `@EngstromJimmy`.
  I have multiple Dometrain courses on Blazor: `https://dometrain.com/author/jimmy-engstrom/`
- The **Blazm** component library that we have written can be found here: `http://blazm.net/`. There are many better grid components out there, but this shows how easy and yet complex a grid component can be.
- **Coding after Work** has many episodes of our podcast and our stream covering Blazor; please follow us on social media: `https://codingafterwork.com/findus`.
- *Daniel Roth* is the PM for Blazor. Amazing to listen to, he has been a guest on our podcast. Search for him on YouTube. X: `@danroth27`.
- *Steve Sanderson* is the guy who invented Blazor; he is definitely worth a follow. He continues to do groundbreaking things in his talks; search for him on YouTube. Make sure to see his NDC Oslo talk where he shows Blazor for the first time. X: `@stevensanderson`.
  He is no longer on the ASP.NET team but he continues to do really amazing stuff.
- **Awesome-Blazor** has a huge list of Blazor-related links and resources that can be found here: `https://github.com/AdrienTorris/awesome-blazor`.
- *Jeff Fritz* shares Blazor knowledge (among other things) on Twitch: `https://www.twitch.tv/csharpfritz`. X: `@csharpfritz`.
- *Carl Franklin* has done a lot of Blazor videos on: `https://BlazorTrain.com/`. X: `@carlfranklin`.

- *John Hilton* has a lot of Blazor content. You can find him here: `https://jonhilton.net/`. X: `@jonhilt`.
- *Patrick God* has a lot of great content on his YouTube channel: `https://www.youtube.com/@PatrickGod`. X: `@_PatrickGod`.
- *David Pine* is a fellow author and the creator of Blazorators and can be found here: `https://github.com/IEvangelist/blazorators`. X: `@davidpine7`.
- *Peter Morris* is the creator of Fluxor and is a great person to follow. X: `@MrPeterLMorris`.
- *Michael Washington* is a fellow author, and we can find him here: `https://adefwebserver.com/`. X: `@ADefWebserver`.
- *Stacy Cashmore* is a frequent speaker and a fellow author and MVP. She was one of the technical reviewers for this book and did an amazing job.
  You can find her here: `https://www.stacy-clouds.net/`.
- *Eric Johansson* is a regular on Twitch, showing his projects and modernizing his .NET Framework apps to a more modern platform. He is a speaker and fellow MVP: `https://www.twitch.tv/thindal`. X: `@EricJohansson`.
- *Egil Hansen* is the creator of bUnit. We can find him here: `https://egilhansen.com/about/`. X: `@egilhansen`.
- *Junichi Sakamoto* has made loads of fantastic Blazor libraries, everything from connecting to gamepads to translation and pre-rendering. You can find his projects here: `https://github.com/jsakamoto`. X: `@jsakamoto`.
- **Blazor University** has a lot of training material and is a great resource to learn more: `https://blazor-university.com/`.
- *Gerald Versluis* has plenty of content on his YouTube channel related to all kinds of .NET things: `https://youtube.com/GeraldVersluis`. X: `@jfversluis`.
- *Maddy Montaquilla* is amazing to watch. She is now focusing on Aspire, and as we have learned in this book, Aspire is amazing. Search for her on YouTube to watch her videos. X: `@maddymontaquila`.
- *James Montemagno* has a great YouTube channel with loads of .NET MAUI content: `https://www.youtube.com/JamesMontemagno`. X: `@JamesMontemagno`.
- *Daniel Hindrikes* has some great .NET MAUI content on this YouTube channel: `https://www.youtube.com/@DanielHindrikes`. X: `@hindrikes`.

The components

Most third-party component vendors, such as Progress Telerik, DevExpress, Syncfusion, Radzen, ComponentOne, and many more, have invested in Blazor. Some cost money and some are free. There are also a lot of open-source component libraries that we can use.

This question comes up a lot: *I am new to Blazor. What third-party vendor should I use?* My recommendation is to try to figure out what you need before investing in a library (either in terms of money or time).

Many vendors can do all the things we need, but in some cases, it will take a bit more effort to make an app work. We started to work on a grid component ourselves, and after a while, we decided to make it open source.

This is how Blazm was born. We had a few special requirements (nothing fancy), but they required us to write a lot of code over and over again to make it work in a third-party vendor component.

We learned so much from writing our component, which is really easy to do. My recommendation is not always to write your own components. It is much better to focus on the actual business problem you are trying to solve.

For us, building a pretty advanced grid component taught us so much about the inner workings of Blazor.

Think about what you need and try out the different vendors to see what works best for you. Perhaps it would be better to build the component yourself, at least in the beginning, to learn more about Blazor.

But always look at your code. If you repeat the same code, wrap it in a component. Always think: *Could this be a reusable component?*

We currently use a component vendor, but we wrap all the components in one of our components. This way, it is easy to set defaults and add logic that is right for us, just as we have learned throughout the book.

Summary

In this chapter, we examined some of the challenges we have encountered while running Blazor in production and discussed where to go from here.

Throughout the book, we have learned how Blazor works and how to create basic and advanced components. We implemented security with both authentication and authorization. We created and consumed an API connected to a database.

We made JavaScript calls and real-time updates. We debugged our application and tested our code, and last but not least, we looked at deploying to production.

We are now ready to apply all this knowledge to the next adventure, another app. I hope you have had as much fun reading this book as I have had writing it. Being part of the Blazor community is so much fun, and we learn new things every day.

Thank you for reading this book. Please stay in touch. I would love to learn about the things you build!

Welcome to the Blazor community!

Learn more on Discord

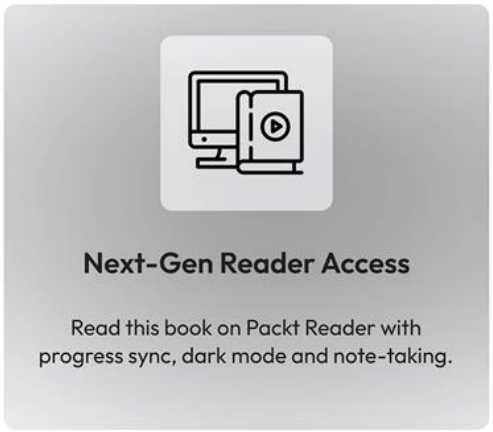
To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

`https://discord.gg/8VuShAAtK`

23

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:



Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> **Note**: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 23.1* on desktop), search for this book by name and select the correct edition.

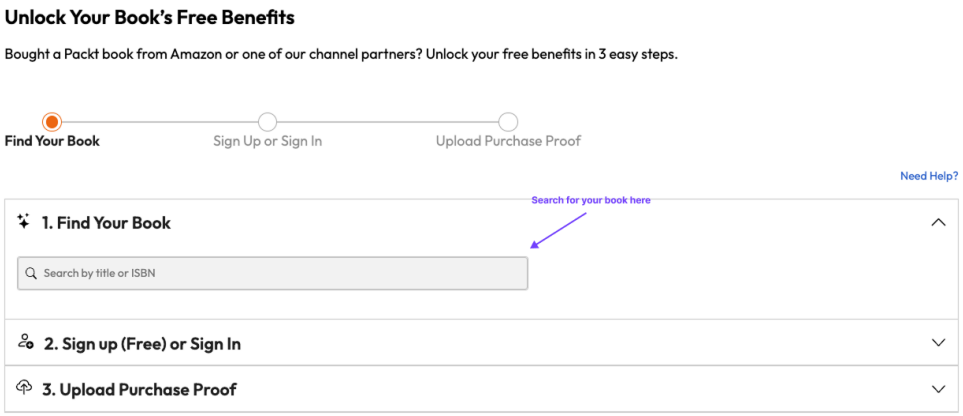


Figure 23.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.

www.packtpub.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.

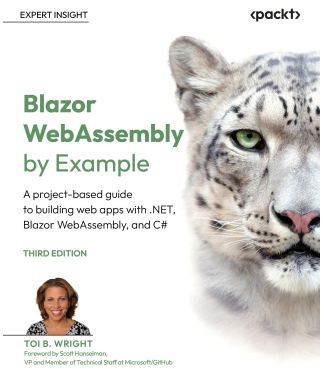
Why subscribe?

- 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

At www.packtpub.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:



Blazor WebAssembly by Example – Third Edition

Toi B. Wright

ISBN: 978-1-80742-867-9

- Integrate AI features into modern Blazor web applications
- Build client-side web apps using C# and Blazor WebAssembly
- Master Razor components, QuickGrid, templated components, events, and Razor class libraries
- Build and consume ASP.NET Web APIs with Entity Framework
- Learn how to call JavaScript with Blazor WebAssembly
- Build high-performance progressive web apps (PWAs) with native app capabilities

Apps and Services with .NET 10 – Third Edition

Mark J. Price

ISBN: 978-1-83546-220-1

- Familiarize yourself with a variety of technologies to implement services, such as gRPC and GraphQL
- Store and manage data locally and cloud-natively with SQL Server
- Implement web services with native AOT publish support
- Leverage Dapper for improved performance over EF Core
- Implement popular third-party libraries such as Serilog, Humanizer, and Noda Time
- Build web apps using Blazor, mobile apps using MAUI, and desktop apps using Avalonia

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.

Share your thoughts

Now you've finished *Web Development with Blazor, Fourth Edition*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

`https://packt.link/r/1806112892`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Y