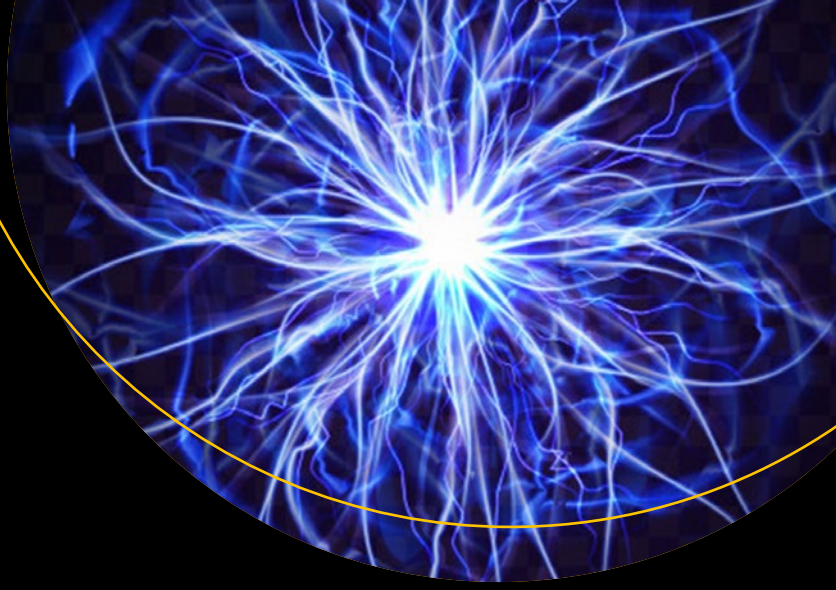

Microsoft Blazor

Building Web Applications in .NET

Second Edition

Peter Himschoot

Apress®

Microsoft Blazor

Building Web Applications in .NET

Second Edition

Peter Himschoot

Apress®

Microsoft Blazor: Building Web Applications in .NET

Peter Himschoot
Melle, Belgium

ISBN-13 (pbk): 978-1-4842-5927-6 ISBN-13 (electronic): 978-1-4842-5928-3
https://doi.org/10.1007/978-1-4842-5928-3

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484259276. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Peter Himschoot works as a lead trainer, architect, and strategist at U2U Training. Peter has a wide interest in software development, which includes applications for the Web, Windows, and mobile devices. Peter has trained thousands of developers, is a regular speaker at international conferences, and has been involved in many web and mobile development projects as a software architect. Peter has been a Microsoft Regional Director from 2003 to 2019, a group of trusted advisors to the developer and IT professional audiences, and to Microsoft. He can be reached on Twitter @peterhimschoot.

About the Technical Reviewer

Gerald Versluis (@jfversluis) is a software engineer at Microsoft from the Netherlands. With years of experience working with Xamarin, Azure, ASP.NET, and other .NET technologies, he has been involved in a number of different projects and has been building several real-world apps and solutions.

Not only does he like to code, but he is also passionate about spreading his knowledge, as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles, live coding, and contributing to open source projects in his spare time. He can be reached on Twitter @jfversluis and his website https://gerald.verslu.is.

Acknowledgments

When Jonathan Gennick from Apress asked me if I would be interested in writing a book on Blazor, I felt honored and of course I agreed that Blazor deserves a book. Writing a book is a group effort, so I thank Jonathan Gennick and Jill Balzano for giving me tips on styling and writing this book, and I thank Gerald Versluis for doing the technical review and pointing out sections that needed a bit more explaining. I also thank Magda Thielman and Lieven Iliano from U2U, my employer, for encouraging me to write this book.

I thoroughly enjoyed writing this book, and I hope you will enjoy reading and learning from it.

Second Edition

As the first edition of *Blazor Revealed* was published (using pre-release software), the Blazor team had made a bunch of changes to the razor syntax, stopping my examples in *Blazor Revealed* from working. Now that Blazor has been released and is completely official (YEAH!!!!), the time has come to publish an updated version of *Blazor Revealed*.

Should you get stuck with an example, I invite you to consult the accompanying code samples for comparison purposes.

Introduction

Back in 2018, I was attending the *Microsoft Most Valued Professional and Regional Directors Summit* where we were introduced to Blazor for the first time by *Steve Sanderson* and *Daniel Roth*. And I must admit, I was super excited about Blazor! We learned that Blazor is a framework that allows you to build single-page applications using C# and allows you to run any standard .NET library in the browser. Before Blazor, your options for building a SPA were Angular, React, Vue.js, and others using JavaScript or one of the other higher-level languages like TypeScript (which get compiled into JavaScript anyway). I was so excited then that I ended up writing the first edition of this book, and now I have updated it for you.

In this introduction, I will show you how browsers are now capable of running .NET assemblies in the browser using WebAssembly, Mono, and Blazor.

A Tale of Two Wars

Think about it. The browser is one of the primary applications on your computer. You use it every day. Companies who build browsers know that very well and are bidding for you to use their browser. At the beginning of mainstream Internet, everyone was using *Netscape*, and Microsoft wanted a share of the market, so in 1995 they built *Internet Explorer 1.0*, released as part of Windows 95 Plus! pack. Newer versions were released rapidly, and browsers started to add new features such as `<blink>` and `<marquee>` elements. This was the beginning of the first browser war, giving people (especially designers) headaches because some developers were building pages with blinking marquee controls 😊. But developers were also getting sore heads because of incompatibilities between browsers. *The first browser war was about having more HTML capabilities than the competition.*

But all of this is now behind us with the introduction of HTML5 and modern browsers like Google Chrome, Microsoft Edge, Firefox, Safari, and Opera. HTML5 not only defines a series of standard HTML elements but also rules on how these should render, making it a lot easier to build a website that looks the same in all modern browsers. Then, in 1995 *Brendan Eich* wrote a little programming language known as *JavaScript* (initially called *LiveScript*) in 10 days (What!?). It was called JavaScript because its syntax was very similar to Java.

JavaScript and Java are not related. Java and JavaScript have as much in common as ham and hamster (I don't know who formulated this first, but I love this phrasing).

Little did Mr. Eich know how this language would impact the modern web and even desktop application development. In 1995 *Jesse James Garett* wrote a white paper called *Ajax (Asynchronous JavaScript and XML)*, describing a set of technologies where JavaScript is used to load data from the server and that data is used to update the browser's HTML. This avoids full-page reloads and allows for client-side web applications, which are written in JavaScript that runs completely in the browser. One of the first companies to apply Ajax was Microsoft when they built *Outlook Web Access (OWA)*. OWA is a web application almost identical to the Outlook desktop application proving the power of Ajax. Soon other Ajax applications started to appear, with Google Maps stuck in my memory as one of the other keystone applications. Google Maps would download maps asynchronously and with some simple mouse interactions allowed you to zoom and pan the map. Before Google Maps, the server would do the map rendering and a browser displayed the map like any other image by downloading a bitmap from a server.

Building an Ajax website was a major undertaking that only big companies like Microsoft and Google could afford. This soon changed with the introduction of JavaScript libraries like jQuery and knockout.js (Knockout was also written by Steve Sanderson, the author of Blazor!). Today we build rich web apps with Angular, React, and Vue.js. All of them are using JavaScript or higher-level languages like TypeScript which get transpiled into JavaScript.

Transpiling will take one language and convert it into another language. This is very popular with TypeScript which gives you a modern high-level language. You need JavaScript to run it in a browser, so TypeScript gets "transpiled" into JavaScript.

This brings us back to JavaScript and the second browser war. JavaScript performance is paramount in modern browsers. Chrome, Edge, Firefox, Safari, and Opera are all competing with one another, trying to convince users that their browser is the fastest with cool-sounding names for their JavaScript engine like *V8* and *Chakra*. These engines use the latest optimization tricks like JIT compilation where JavaScript gets converted into native code as illustrated in Figure 1.

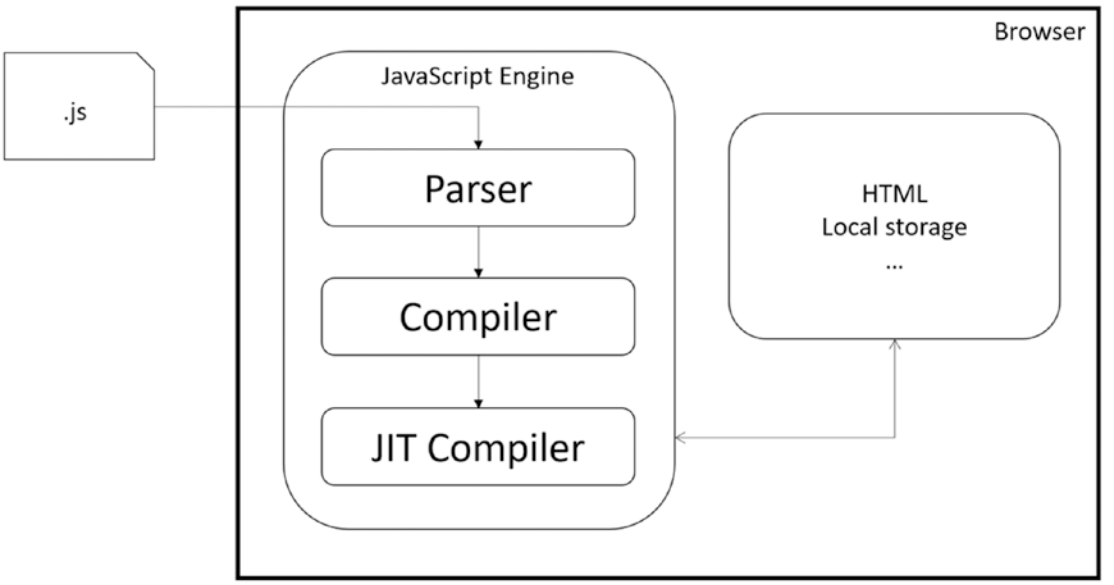

Figure 1. *The JavaScript execution process*

This process takes a lot of effort because JavaScript needs to be downloaded into the browser, where it gets parsed, then compiled into bytecode and then Just-In-Time converted into native code. So how can we make this process even faster?

The second browser war is all about JavaScript performance.

Introducing WebAssembly

WebAssembly allows you to take the parsing and compiling to the server. With WebAssembly, you compile your code in a format called WASM (an abbreviation of WebASseMbly), which gets downloaded by the browser where it gets Just-In-Time compiled into native code as in Figure 2. Open your browser and google "*webassembly demo zen garden.*" One of the links should take you to `https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html` where you can see an impressive ray-trace demo of a Japanese Zen garden with a screenshot in Figure 3.

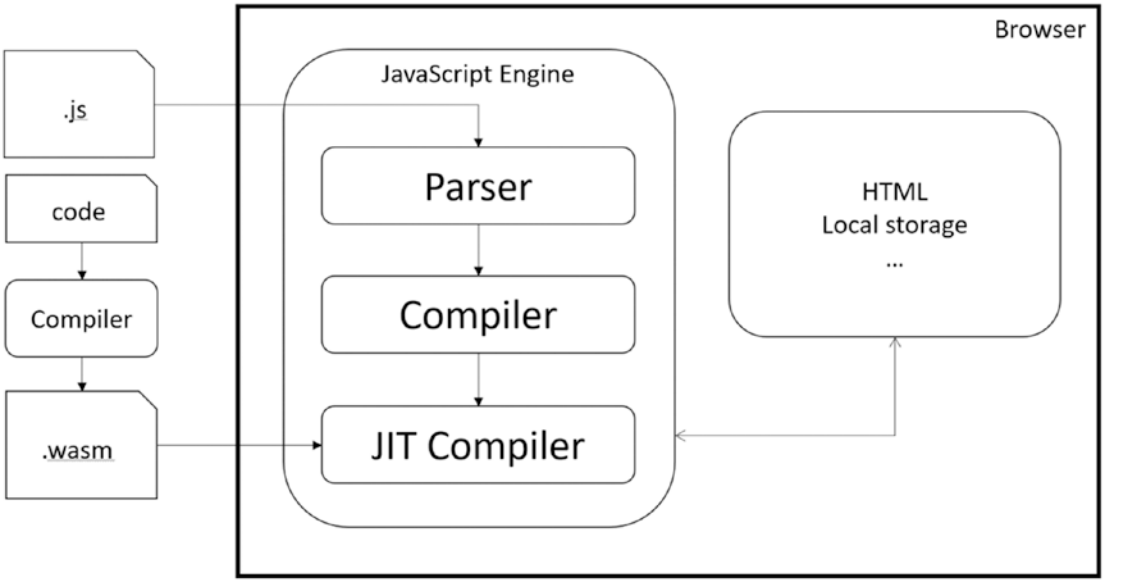

Figure 2. *The WebAssembly execution process*

Figure 3. *Japanese Zen garden*

From the official site webassembly.org

> *WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.*

So WebAssembly as a new binary format optimized for browser execution, it is NOT JavaScript. There are compilers for languages like C++ and Rust which compile to WASM. Some people have compiled C++ applications to wasm, allowing to run them in the browser. There is even a Windows 2000 operating system compiled to wasm!

Which Browsers Support WebAssembly?

WebAssembly is supported by all major browsers: Chrome, Edge, Safari, Opera, and Firefox, including their mobile versions. As WebAssembly will become more and more important, we will see other modern browsers follow suit, but don't expect Internet Explorer to support WASM. You can check for this on `https://caniuse.com/#search=wasm`.

WebAssembly and Mono

Mono is an open source implementation of the .NET CLI specification, meaning that Mono is a platform for running .NET assemblies. Mono is used in *Xamarin* for building mobile applications that run on Windows, Android, and iOS mobile operating systems. You can also use it to build applications for macOS, Linux, Tizen, and others. Mono also allows you to run .NET on Linux (its original purpose) and is written in C++. This last part is important because we saw that you can compile C++ to WebAssembly. So, what happened is that the Mono team decided to try to compile Mono to WebAssembly, which they did successfully. There are two approaches. One is where you take your .NET code and you compile it together with the Mono runtime into one big WASM application. However, this approach takes a lot of time because you need to take several steps to compile everything into WASM, not so practical for day-to-day development. The other approach takes the Mono runtime and compiles it into WASM, and this runs in the browser where it will execute .NET Intermediate Language just like normal .NET does. The big advantage is that you can simply run .NET assemblies without having to compile them first into WASM. This is the approach currently taken by Blazor. But Blazor

is not the only one taking this approach. For example, there is the *Ooui* project which allows you to run *Xamarin.Forms* applications in the browser. The disadvantage of this is that it needs to download a lot of .NET assemblies. This can be solved by using *Tree Shaking* algorithms which remove all unused code from assemblies.

Interacting with the Browser with Blazor

WebAssembly with Mono allows you to run .NET code in the browser. *Steve Sanderson* used this to build Blazor. Blazor uses the popular ASP.NET MVC approach for building applications that run in the browser. With Blazor, you build Razor files (Blazor = Browser + Razor) which execute inside to browser to dynamically build a web page. With Blazor, you don't need JavaScript to build a web app which is good news for thousands of .NET developers who want to continue using C# (or F#).

How Does It Work?

Let's start with a simple razor file in Listing 1 which you can find when you create a new Blazor project (which we will do in the first chapter).

Listing 1. The `Counter` razor file

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

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

@code {
  private int currentCount = 0;

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

This file gets compiled into .NET code (you'll find out how later in this book) which is then executed by the Blazor engine. The result of this execution is a tree-like structure called the *render tree*. The render tree is then sent to JavaScript which updates the DOM to reflect the render tree (creating, updating, and removing HTML elements and attributes). Listing 1 will result in `h1`, `p` (with the value of `currentCount`), and `button` HTML elements. When you interact with the page, for example, when you click the button, this will trigger the button's click event which will invoke the `IncrementCount` method from Listing 1. The render tree is then regenerated, and any changes are sent again to JavaScript which will update the DOM. This process is illustrated in Figure 4.

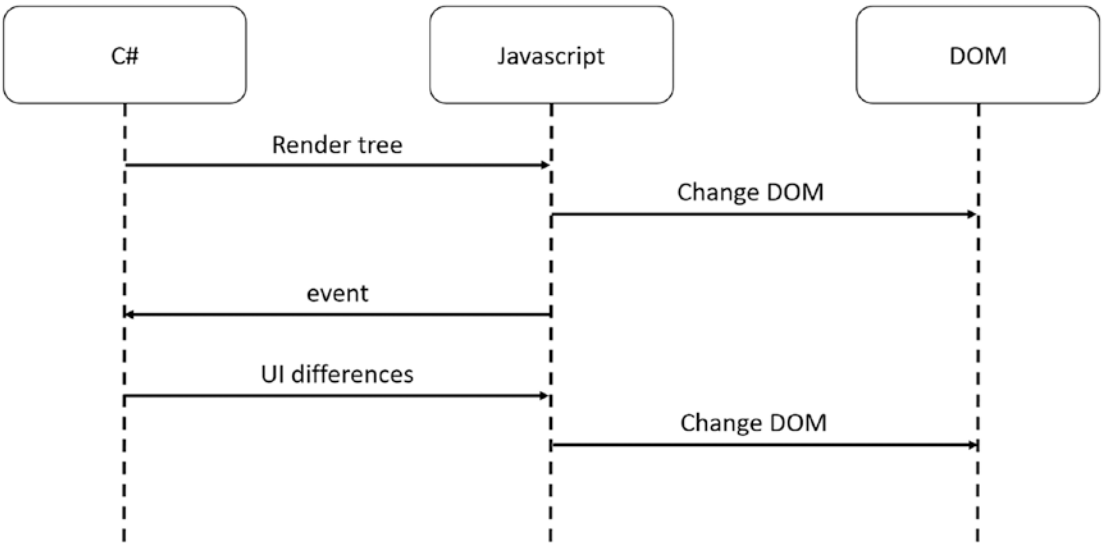

Figure 4. *The Blazor WebAssembly DOM generation process*

This model is very flexible. It allows you to build *Progressive Web Apps* and also can be embedded in *Electron* desktop applications, of which Visual Studio Code is a prime example.

In Chapter 1, section "The Client-Side Blazor Bootstrap Process," we will look at which files get downloaded. One of the drawbacks of Blazor WebAssembly is that this is a substantial download on first use (after this, most files can be cached by the browser or the Blazor runtime itself), especially the Mono runtime itself. If you want to avoid this big download, you can use Server-Side Blazor.

Server-Side Blazor

At the August 7, 2018, *ASP.NET Community Standup,* Daniel Roth introduced a new execution model for Blazor called Server-Side Blazor. In this model, your Blazor site is running on the server resulting in a way smaller download for the browser.

The Server-Side Model

We just saw that Client-Side Blazor (also known as Blazor WebAssembly) builds a *render tree* using the Mono runtime which then gets sent to JavaScript to update the DOM. With Server-Side Blazor, the render tree is built on the server and then gets serialized to the browser using *SignalR*. JavaScript in the browser then deserializes the render tree to update the DOM. Pretty similar to the client-side Blazor model. When you interact with the site, events get serialized back to the server which then executes the .NET code, updating the render tree which then gets serialized back to the browser. I've illustrated this process in Figure 5. The big difference is that there is no need to send the Mono runtime and your Blazor assemblies to the browser. And the programming model stays the same! You can switch between server-side and client-side Blazor (also known as Blazor WebAssembly) with just a couple of small changes to your code.

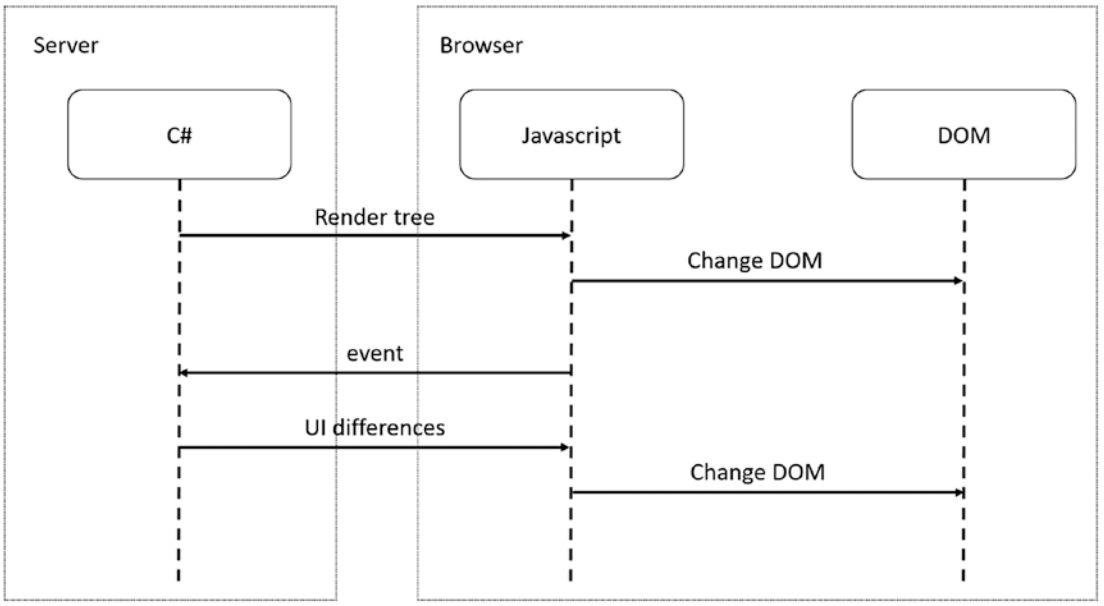

Figure 5. *Blazor Server-Side*

Pros and Cons of the Server-Side Model

The Server-Side model has a couple of benefits, but also some drawbacks. Let's discuss these here so you can decide which model fits your application's needs.

Smaller Downloads

With server-side, your application does not need to download `blazor.wasm` (the Mono runtime) nor all your .NET assemblies. The browser downloads a small JavaScript library which sets up the SignalR connection to the server. This means that the application will start a lot faster, especially on slower connections.

Development Process

Blazor client-side has limited debugging capabilities, resulting in added logging. Because your .NET code is running on the server, you can use the regular .NET debugger. You could start building your Blazor application using the server-side model, and when it's finished, switch to the client-side model by making a small change to your code.

.NET APIs

Because you are running your .NET code on the server, you can use all the .NET APIs you would use with regular Asp.Net Core MVC applications, for example, accessing the database directly. Do note that doing this will stop you from quickly converting it into a client-side application.

Online Only

Running the Blazor application on the server does mean that your users will always need access to the server. This will prevent the application from running in Electron, nor will you be able to run it as a Progressive Web Application (PWA). And if the connection drops between the browser and server, your user could lose some work because the application will stop functioning. Blazor will try to reconnect to the server without losing any data, so most of the time users will not lose any work done.

Server Scalability

All your .NET code runs on the server, so if you have thousands of clients, your server(s) will have to handle all the work. Not only that, Blazor uses a state-full model which will require you to keep track of every user's state on the server. So your server will need more resources than with client-side Blazor.

Summary

In this introduction, we looked at the history of the browser wars and how this resulted in the creation of WebAssembly. Mono allows you to run .NET assemblies, and because Mono can run on WebAssembly, we can now run .NET assemblies in the browser! All of this resulted in the creation of Blazor, where you build razor files containing .NET code which updates the browser's DOM, giving us the ability to build single-page applications in .NET, instead of JavaScript.

CHAPTER 1

Your First Blazor Project

Getting hands-on is the best way to learn. In this chapter, you'll install the prerequisites to developing with Blazor. These include Visual Studio along with some needed extensions. Then you'll create your first Blazor project in Visual Studio, run the project to see it work, and finally inspect the different aspects of the project to get a "lay of the land" view for how Blazor applications are developed.

I learned an important lesson from the first edition of this book: Never underestimate the speed at which Microsoft innovates! All code samples in the first edition of *Blazor Revealed* became invalid quite rapidly. I do not expect this to happen again with this edition since it is based on the Release To Manufacture (RTM) version of Blazor. If something does not work, simply consult the sources that come with this book. I will keep these up to date.

The source code for this book is available on GitHub via the book's product page, located at www.apress.com/9781484259276.

Installing Blazor Prerequisites

Working with Blazor requires you to install some prerequisites so in this section, you will install what is needed to get going.

.NET Core

Blazor runs on top of .NET Core, providing the web server for your project which will serve the client files that run in the browser and run any server-side APIs that your Blazor project needs. .NET Core is Microsoft's cross-platform solution for working with .NET on Windows, Linux, and OS X.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_1

You can find the installation files at www.microsoft.com/net/download. Look for the latest version of the .NET Core SDK (you'll need at least version 3.1). Download the installer, then simply run it and accept the defaults.

Verify the installation when the installer is done by opening a new command prompt and typing the following command:

```
dotnet --version
```

Look for the following output to indicate that you installed the correct version. The version number should be at least 3.1.201.

Should the command's output show an older version (e.g., 2.1.200), you will need to download and install a more recent version of .NET Core SDK.

Visual Studio 2019

Visual Studio 2019 (from now on I will refer to Visual Studio as VS) is one of the integrated development environments (IDE) we will use throughout this book. The other IDEs will be Visual Studio Code and Visual Studio for Mac. With either one, you can edit your code, compile it, and run it all from the same application. The code samples are also the same. However, VS only runs on Windows, so if you're using another OS, please continue to the section "Visual Studio Code." All samples should also work fine with Visual Studio for Mac.

Download the latest version of Visual Studio 2019 from www.visualstudio.com/downloads/.

Run the installer and make sure that you install the ASP.NET and web development role as shown in Figure 1-1.

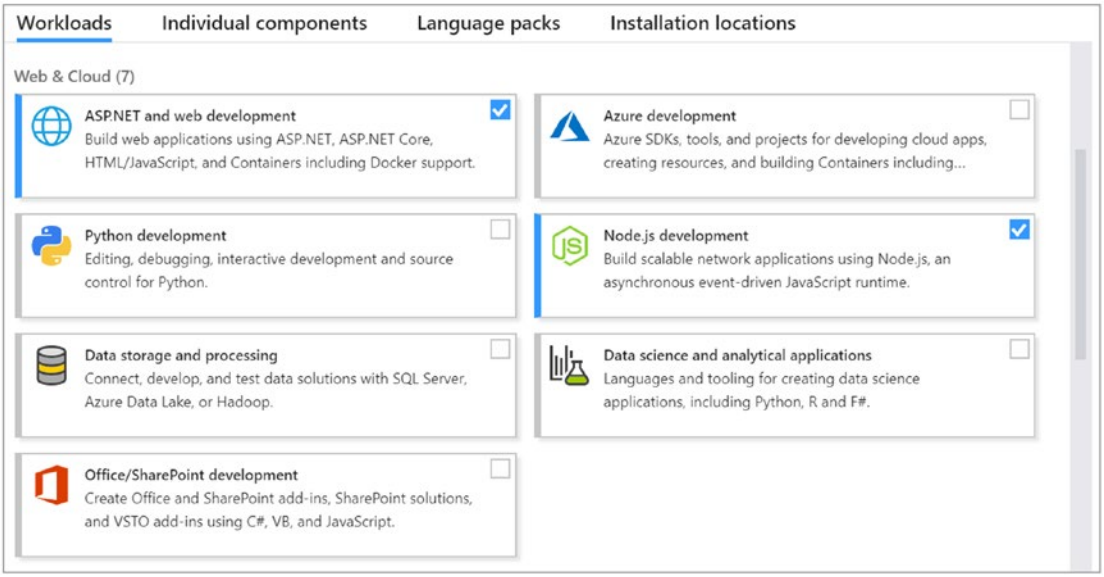

Figure 1-1. *The Visual Studio Installer Workloads selection*

After installation, run Visual Studio from the Start menu. Then open the Help menu and select About Microsoft Visual Studio. The About Microsoft Visual Studio dialog window should specify at least version 16.6.0 as illustrated in Figure 1-2. I am currently using the latest preview version of Visual Studio 2019, but by the time you are reading this, Visual Studio 16.6.0 should be out for general release.

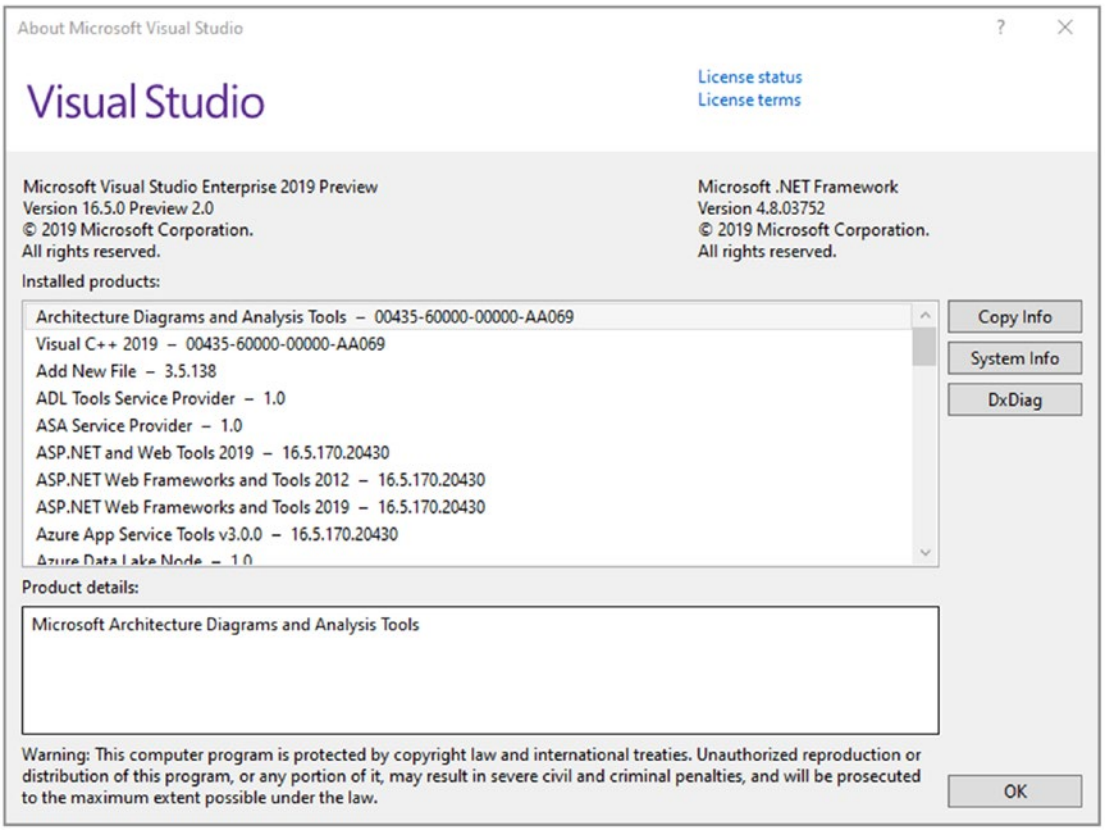

Figure 1-2. *About Microsoft Visual Studio*

Visual Studio Code

Visual Studio Code is a free, modern, cross-platform development environment with an integrated editor, git source control, and debugger. The environment has a huge range of extensions available allowing you to use all kinds of languages and tools directly from Code. So, if you don't have access to (because you're running a non-Windows operating system or you don't want to use) Visual Studio, use Code.

Download the installer from `www.visualstudio.com`. Run it and simply choose the defaults.

After installation, I do advise you to install a couple of extensions for Code, especially the C# extension. Start Code, and at the left side, select the Extensions tab as shown in Figure 1-3.

Figure 1-3. *Visual Studio Code Extensions tab*

You can search for extensions, so start with C# which is the first extension from Figure 1-4. This extension will give you IntelliSense for the C# programming language and .NET assemblies. You will probably get a newer version listed so take the latest.

Click install.

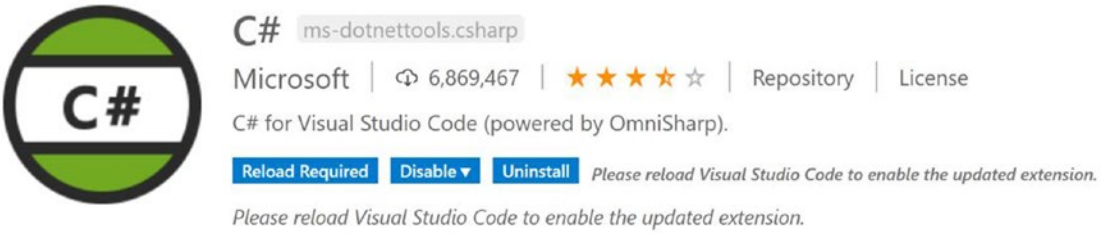

Figure 1-4. *C# for Visual Studio Code*

Installing the Blazor Templates for VS/Code

Throughout this book, we will create several different Blazor projects. Not all of them can be created from Visual Studio or Code, meaning you'll need to install the templates for Blazor projects. This section's example shows how to install those templates from the .NET Core command-line interface, also known as the .NET Core CLI. You should have this command-line interface as part of your .NET Core installation.

Open a command line on your OS, and type the following to install the templates from NuGet:

```
dotnet new -i Microsoft.AspNetCore.Components.WebAssembly.Templates
```

These templates will allow you to quickly generate projects and items. Verify the installation by typing the following command:

```
dotnet new --help
```

This command will list all the templates that have been installed by the command-line interface. You will see four columns. The first shows the template's description, the second column displays the name, the third lists the languages for which the template is available, and the last shows the tags, a kind of group name for the template. Among those listed are the following:

`Razor Component`	`razorcomponent`
`Blazor Server App`	`blazorserver`
`Blazor WebAssembly App`	`blazorwasm`
`Razor Class Library`	`razorclasslib`

Generating Your Project with Visual Studio

With Blazor projects, you have a couple of choices. You can create a stand-alone Blazor project (using the `blazorwasm` template) that does not need server-side code. This kind of project known as *Blazor WebAssembly* has the advantage that you can simply deploy it to any web server which will function as a file server, allowing browsers to download your site just like any other site.

Or you can create a hosted project (adding the `--hosted` option) with client, server, and shared code. This kind of Blazor WebAssembly project will require you to host it where there is .NET Core 3.1 support because you will execute code on the server as well.

The third option is to run all Blazor code on the server (using the `blazorserver` template). This is known as *Blazor Server-Side* (previously known as Blazor components). In this case, the browser will use a SignalR connection to receive UI updates from the server and to send user interaction back to the server for processing.

In this book, we will use the second option (Blazor WebAssembly hosted on ASP.NET MVC Core), but the concepts you will learn in this book are the same for all three options. You can even develop for Blazor WebAssembly and Blazor Server-Side at the same time. Why? Because debugging support for Blazor WebAssembly is limited, so you develop with Blazor Server-Side using all debugger features you know and love. But you can test everything with Blazor WebAssembly ensuring you can run everything in the browser later. This is the way I like to work. However, to pull this off, you need some experience with Blazor first, so keep reading…

Creating a Project with Visual Studio

For your first project, start Visual Studio and select `Create a new project`.

Type `Blazor` in the search box, and select the `Blazor App` project template as illustrated in Figure 1-5.

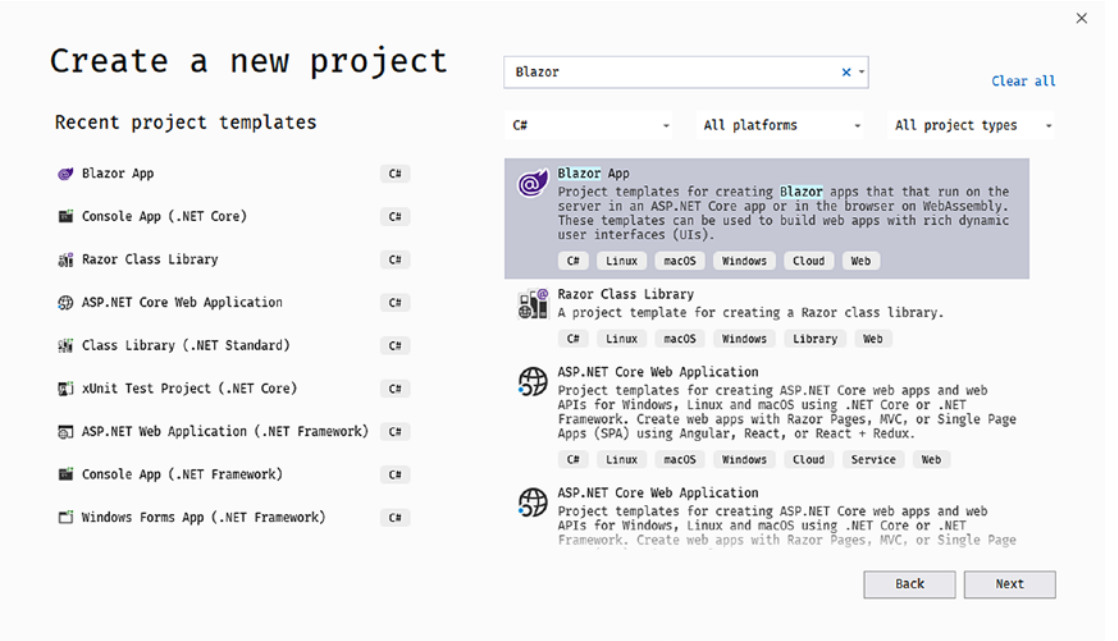

Figure 1-5. *Visual Studio New Project dialog*

Click *Next.*

Name your project *MyFirstBlazor*, leave the rest to the preset defaults, and click *Create.*

On the next screen, you can select what kind of Blazor project you want to generate. Select `Blazor WebAssembly App` as in Figure 1-6, check the `ASP.NET Core hosted` checkbox, and click `Create`.

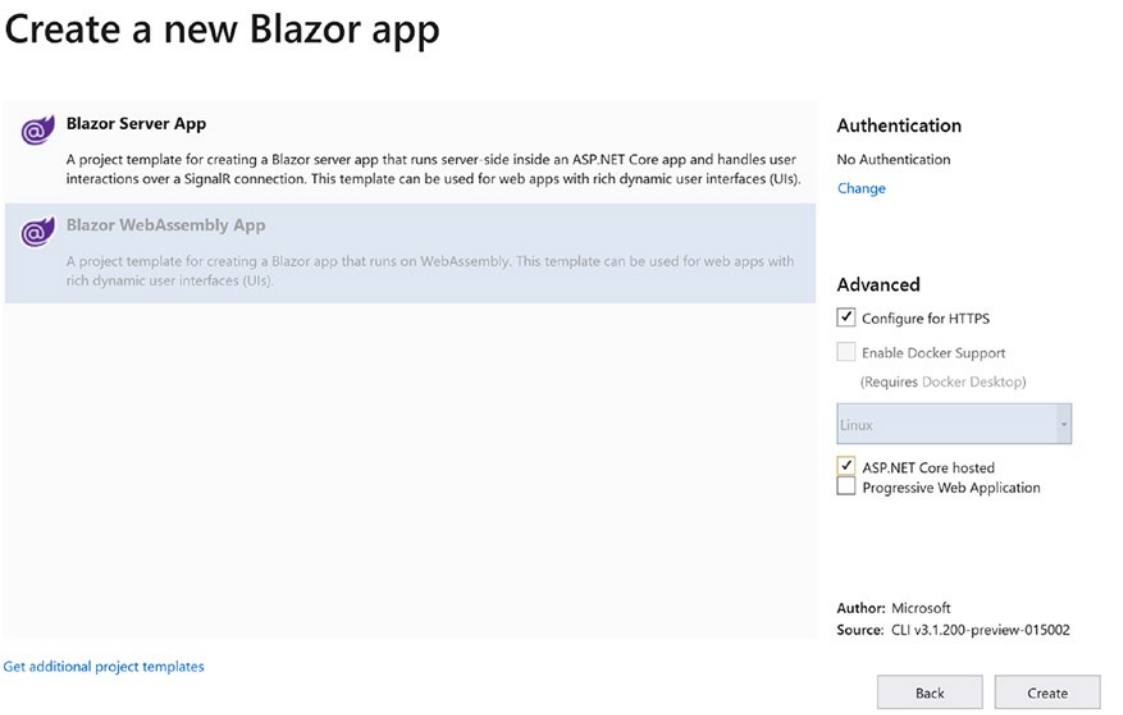

Figure 1-6. *New ASP.NET Core web application*

Wait for Visual Studio to complete. Then build your solution.

Generating the Project with dotnet cli

To generate the project with dotnet cli, start by opening a command line, and change the current directory to wherever you want to create the project. Now execute this command to create a new project from the `blazorwasm` template in the `MyFirstBlazor` directory:

```
dotnet new blazorwasm --hosted -o MyFirstBlazor
```

This command will take a little while because it will download a bunch of NuGet packages from the Internet. When the command is ready, you can build your project using

```
cd MyFirstBlazor
dotnet build
```

Now open your project's folder with Code. When Code has loaded everything (be patient), it will pop a little question as in Figure 1-7.

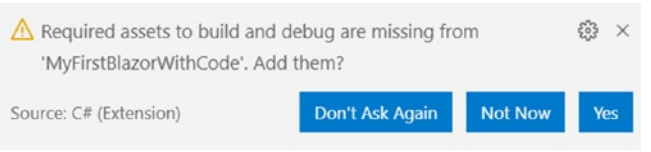

Figure 1-7. *Code asking to add build and debug assets*

Answer Yes. This will add a folder called `.vscode` with configuration files adding support for building and running the project from Code. If you already have a .vscode folder (because you copied an existing project, for example), you will not get this question.

Thanks to this integration with Visual Studio Code, you can simply press F5 to build and run your project…

Running the Project

Press Ctrl-F5 to run (this should work for both Visual Studio and Code). Your (default) browser should open and display the home page as shown in Figure 1-8.

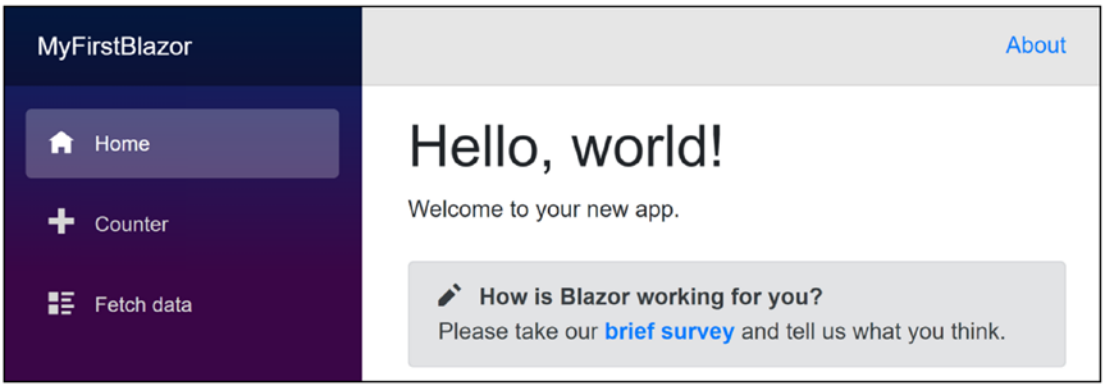

Figure 1-8. *Your first application – Home screen*

This generated single-page application (SPA) has on the left side a navigation menu allowing you to jump between different pages. On the right side, you will see the selected screen, and in Figure 1-8, it is showing the Home screen. And in the top right corner, there is an `About` link to `https://blazor.net/` which is the "official" Blazor documentation website.

The Home Page

The home page shows the mandatory "Hello, world!" demo, and it also contains a survey component you can click to fill out a survey (this is a real survey, so please let Microsoft know you like Blazor!). The *SurveyPrompt* is the first example of a custom Blazor component. We will discuss building components like SurveyPrompt in Chapter 3.

The Counter Page

In the navigation menu, click the Counter tab. Doing so opens a simple screen with a number and a button as illustrated in Figure 1-9. Clicking the button will increment the counter. Try it!

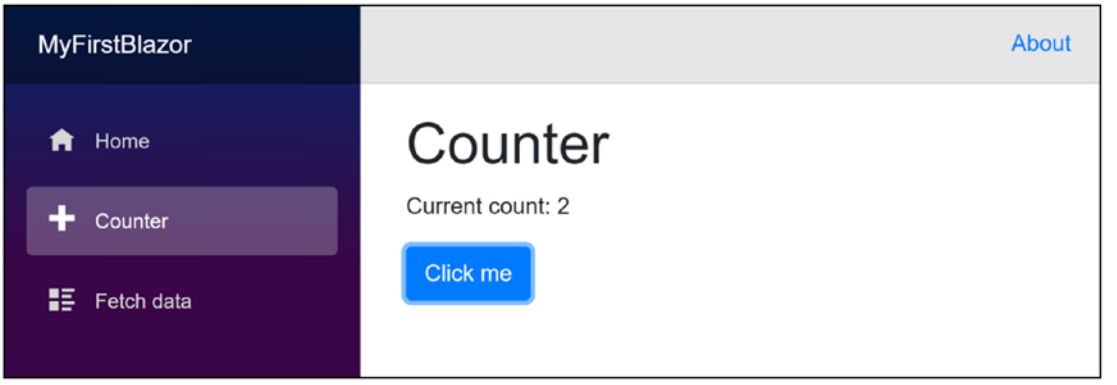

Figure 1-9. *Your first application – Counter screen*

The Fetch Data Page

In the navigation menu, click the `Fetch data` tab. Here you can watch a (random and fake) weather forecast as shown in Figure 1-10. This forecast is generated on the server when asked by the client. This is very important because the client (which is running in the browser) cannot access data from a database directly, so you need a server that can access databases and other data storage.

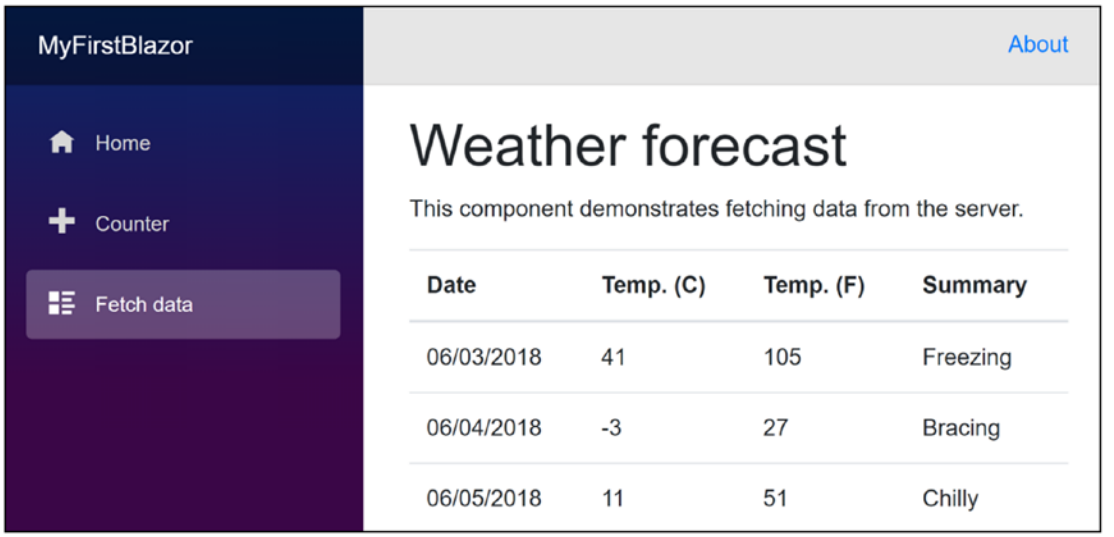

Figure 1-10. *Your first application – Fetch data screen*

Examining the Project's Parts

Now being able to play with these pages is all nice, but let us have a look at how all this works. We will start with the server project which hosts our Blazor website. Then we will look at the shared project which contains classes used by both server and client. Finally, we will examine the client project which is the actual Blazor implementation.

The Solution

Visual Studio, Visual Studio Code, and Visual Studio for Mac use solution files to group projects that will form an application. So, a typical Blazor project consists of a server, a client, and a shared project grouped into a single solution. This simplifies building everything since the solution allows tools to figure out in which order to compile everything. Hey, you could even switch between Visual Studio, VS for Mac, and Code because they both use the same project and solution files!

The Server

Web applications are a bunch of files that get downloaded by the browser from a server. It is the server's job to provide the files to the browser upon request. There is a whole range of existing servers to choose from, for example, IIS on Windows or Apache on

Linux. ASP.NET Core has a built-in server that you generated with the `--hosted` option, which you can then run on Windows, Linux, or OS X.

The topic of this book is Blazor, so we're not going to discuss all the details of the server project that got generated (Microsoft has very good documentation on .NET Core), but I do want to show you an important thing. In the server project (`MyFirstBlazor.Server`), look for `Startup.cs`. Open this file and scroll down to the `Configure` method shown in Listing 1-1.

Listing 1-1. The server project's `Startup.Configure` method

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

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

  app.UseRouting();

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

The `Configure` method is responsible for installing middleware. Middleware objects are little .NET components that each have a clear responsibility. When you type in a URI, the browser sends a request to the server, which then passes it on to the middleware components. Some of these will take the request and return a response; some of them take the response and do something with it. Look at the first lines in the `Configure` method in Listing 1-2.

Listing 1-2. The UseDeveloperExceptionPage middleware

```
if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseWebAssemblyDebugging();
  }
```

Would you like to see a detailed error page when the server has an uncaught exception? The `UseDeveloperExceptionPage` takes care of that. Of course, you don't need that in production (you should handle all exceptions correctly <grin>) so this middleware is only used when running in a development environment. How does the server know if you are running in development or release? The if statement you see here checks an environment variable called `ASPNETCORE_ENVIRONMENT`, and if the environment variable is set to `Development`, it knows you are running in development mode.

The `Fetch` data screen downloads weather information from the server. These kinds of requests will be handled by the MVC middleware. We will discuss this in more detail in Chapter 5.

The Blazor bootstrap process requires a bunch of special files, especially `dotnet.wasm` (`dotnet.wasm` is the .NET runtime compiled as WebAssembly). These are served by the Blazor middleware, which is handled by the `UseBlazorFrameworkFiles` middleware. Later in this chapter, you will see why.

The Shared Project

When you click the `Fetch` data tab, your Blazor project fetches some data from the server. The shape of this data needs to be described in detail (computers are picky things), and in classic projects, you would describe this model's shape twice, once for the client and again for the server because these would use different languages. Not with Blazor! In Blazor, both client and server use C#, so we can describe the model once and share it between client and server as shown in Listing 1-3.

Listing 1-3. The shared WeatherForecast class

```
using System;

namespace MyFirstBlazor.Shared
{
 public class WeatherForecast
 {
  public DateTime Date { get; set; }

  public int TemperatureC { get; set; }

  public string Summary { get; set; }

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

The Client Blazor Project

Open the client project's `wwwroot` folder and look for `index.html`. The contents of that file should appear as shown in Listing 1-4. To be honest, this looks mostly like a normal HTML page. But on closer inspection, you'll see that there is a weird `<app>` HTML tag there.

```
<app>Loading...</app>
```

The `<app>` HTML element does not exist! It is an example of a Blazor component. You will also find an `<script>` element near the end.

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

This script will install Blazor by downloading `dotnet.wasm` and your assemblies.

Listing 1-4. `index.html`

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

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>PizzaPlace</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css"
          rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
</head>

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

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

</html>
```

Routing

What is that <app> element? Open `Program.cs` from the `MyFirstBlazor.Client` project and look for the `Main` method as in Listing 1-5. Here you can see the `App` component is associated with the app tag from `index.html`. A Blazor component uses a custom tag like <app>, and the Blazor runtime replaces the tag with the component's markup which is normal HTML recognized by the browser. We will discuss Blazor components in Chapter 3.

Listing 1-5. The `Configure` method associating the `app` element to the `App` component

```
public static async Task Main(string[] args)
{
  var builder = WebAssemblyHostBuilder.CreateDefault(args);
```

```
  builder.RootComponents.Add<App>("app");

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

The main thing the App component does is to install the router as in Listing 1-6. You can find this code in the app.razor file in the client project. The router is responsible for loading a Blazor component depending on the URI in the browser. When the route is not found, it will display the <NotFound> content, which currently shows a simple not found message. For example, if you browse to the / URI, the router will look for a component with a matching @page directive.

Listing 1-6. The App component

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

In our current MyFirstBlazor project, this will match the Index component which you can find in the Index.razor file from the Pages folder. The Index component displays a Hello World message and the survey link as in Listing 1-7.

Listing 1-7. The Index component

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

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

Layout Components

Look at Figures 1-8 and 1-9. Both have the same menu. This menu is shared among all our Blazor components and is known as a layout component. We will discuss layout components in Chapter 7. But how does Blazor know which component is the layout component? Look again at Listing 1-6. When the route is found, it uses a default layout called `MainLayout`. In our project, the layout component can be found in `MainLayout.razor` from the `Shared` folder, which I've listed in Listing 1-8.

Listing 1-8. The `MainLayout` component

```
@inherits LayoutComponentBase

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

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

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

The first `div` with class `sidebar` contains a single Blazor component: `NavMenu`. This is where your navigation menu gets defined. The sidebar will display a menu, allowing you to navigate between Home, Counter, and Fetch data. We will look in more detail at navigation and routing in Chapter 7.

The next div with class main has two parts. The first is the `About` link you see on every page. The second part contains the `Body`; this is where the selected page will be shown. For example, when you click the `Counter` link in the navigation menu, this is where the `Counter.razor` Blazor component will go.

Debugging Client-Side Blazor

Of course, while building your Blazor app, you will encounter unexpected behavior from time to time. Debugging the server-side can be done just like any .NET project using Visual Studio or Code. But with Client-Side Blazor, your code will be running in the browser. You will be happy to learn that the Visual Studio/Code debugger works with Blazor, although limited. You can put breakpoints in your code, step through your code, and observe variables holding simple types like `bool`, `int`, and `string`.

Debugging with Visual Studio

To enable debugging with Visual Studio, open the `launchSettings.json` file from the Server project. You will need to set the `inspectUri` property in here, like in Listing 1-9 (the template normally will configure this for you). This property enables the IDE to detect that this is a Blazor WebAssembly app and instructs the script debugging infrastructure to connect to the browser through Blazor's debugging proxy.

Listing 1-9. Enable debugging in Visual Studio

```
"MyFirstBlazor.Server": {
      "commandName": "Project",
      "launchBrowser": true,
      "inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/
       debug/ws-proxy?browser={browserInspectUri}",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
```

Now run your application in Visual Studio with the debugger by pressing `F5`. Be patient while your Blazor site starts to run. Now you can put a breakpoint in your code, for example, on the `IncrementCount` method of the `Counter.razor` component as in Figure 1-11, line 14. Simply click in the gray area left to your code (also known as the gutter) and a red dot will appear, indicating that the debugger will stop at this code.

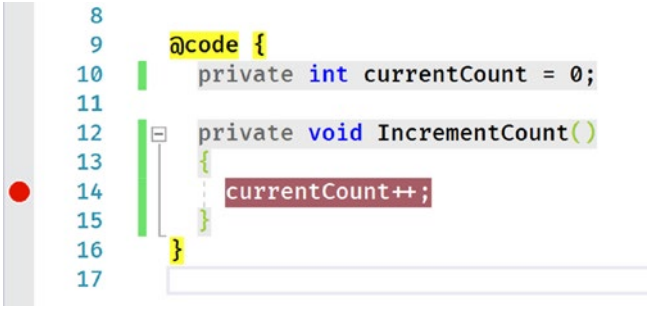

Figure 1-11. *Setting a breakpoint in the IncrementCount method*

Go back to your Blazor application and click the Counter's Click me button. The debugger should stop on the `IncrementCount` method. You can now examine the content of simple variables in the `Locals` window, like in Figure 1-12.

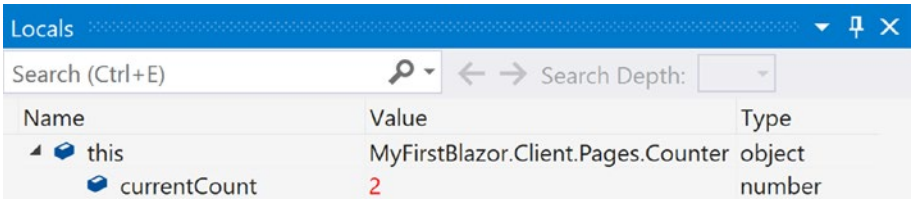

Figure 1-12. *Using the Locals debugger window to inspect simple variables*

Debugging in Chrome

To debug a Client-Side Blazor application, first make sure you build the debug version, which is the default. Then run your application, and in Chrome, press `Shift-Alt-D` in Windows and Linux or `Shift-Cmd-D` on macOS.

The first time you do this with your application, you will need to start the Chrome remote debugger. Instructions for this are automatically generated by chrome as shown in Figure 1-13. Simply follow the instructions.

Figure 1-13. *How to start Chrome with the debug server enabled*

Once the remote debugger has been started, you should see your application running again. Press the magic key combination again. Open the `Sources` tab in the debugger and expand `file://` to see your files. Figure 1-14 shows the files from the initial `MyFirstBlazor` app.

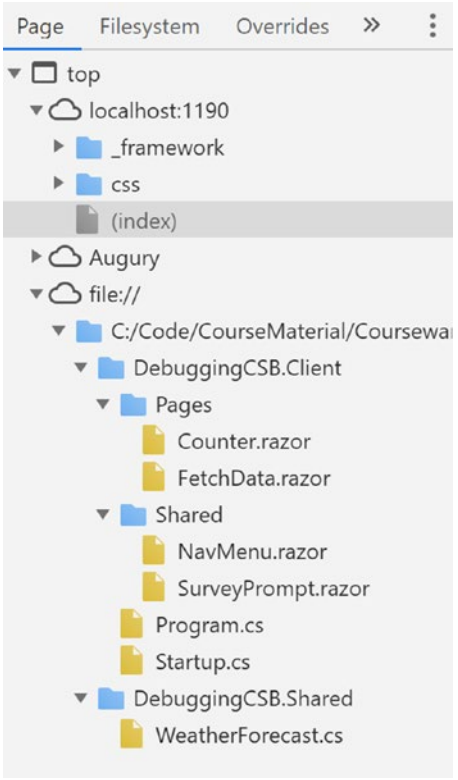

Figure 1-14. *The Chrome debugger with all your components*

Select `Counter.razor`, and add a breakpoint in the `IncrementCount` method, as shown in Figure 1-15, line 14.

```
1  @page "/counter"
2
3  <h1>Counter</h1>
4
5  <p>Current count: @currentCount</p>
6
7  <button class="btn btn-primary" @onclick="@IncrementCount">Click me</button>
8
9  @code {
10     int currentCount = 0;
11
12     void IncrementCount()
13     {
14         currentCount++;
15     }
16 }
17
```

Figure 1-15. *Add a breakpoint in the Chrome debugger*

Switch back to the tab where your application is running and click the `Click me` button. The debugger should stop on the breakpoint. Switch back to the Chrome debugger. Press `F10` to step to the next statement. Press `F8` to continue running.

Open `FetchData.razor` and put a breakpoint on the `@foreach` statement. Switch back to your Blazor application, and select the `Fetch data` tab. Your breakpoint should be hit, meaning that you can also put breakpoints on razor directives.

The Client-Side Blazor Bootstrap Process

Examine Listing 1-4 again. At the bottom, you will find the `<script>` element responsible for bootstrapping Blazor in the browser. Let's look at this process.

Go back to your browser and open its developer tools (most browsers will open the developer tools when you press F12). We will have a look at what happens at the network layer.

> In all screenshots, I will be using the Chrome browser, mainly because it is available on all platforms (Windows, Linux, and OS X) and because it is very popular with a lot of web developers. If you prefer to use another browser, go right ahead!

Refresh your browser to see what gets downloaded from the server as in Figure 1-16. If Figure 1-16 does not match what you see (please ignore some minor details), you want to clear the browser's cache. Browsers use a cache to avoid reloading files from the server, but when you are developing, you will need to clear the cache to ensure you are getting the latest changes from the server. First, you will see `index.html` (shown as localhost) being downloaded, which in turn downloads `bootstrap.min.css` and `site.css`, and then `blazor.webassembly.js`. A little lower you will see that `dotnet.js` gets downloaded, which in turn will download `dotnet.wasm`. This is the Mono runtime compiled to run on WebAssembly!

Name	Status	Type
localhost	200	document
bootstrap.min.css	200	stylesheet
site.css	200	stylesheet
blazor.webassembly.js	200	script
open-iconic-bootstrap.min.css	200	stylesheet
blazor.boot.json	200	fetch
favicon.ico	404	text/plain
dotnet.3.2.0-preview2.20159.2.js	200	script
dotnet.wasm	200	fetch
MyFirstBlazor.Client.dll	200	fetch
Microsoft.AspNetCore.Blazor.HttpClient.dll	200	fetch
System.Text.Json.dll	200	fetch
System.Runtime.CompilerServices.Unsafe.dll	200	fetch
mscorlib.dll	200	fetch

Figure 1-16. *Examining the bootstrap process using the network log*

Now that the .NET runtime is running, you will see (scroll down?) that `MyFirstBlazor.Client.dll` gets downloaded, followed by all its dependencies, including `mscorlib.dll` and `system.dll`. These files contain the .NET libraries containing classes such as `string` used to execute all kinds of things, and they are the same libraries you use on the server. This is very powerful because you can reuse existing .NET libraries in Blazor you or others built before!

If you don't see your assembly and other .NET assemblies, clear your browser's cache storage. Blazor stores its downloaded assemblies in the browser's cache storage after the first initial download.

The Server-Side Blazor Bootstrap Process

Let's look at a server-side Blazor project. Open Visual Studio and create a new project, but this time select Blazor Server App as in Figure 1-17.

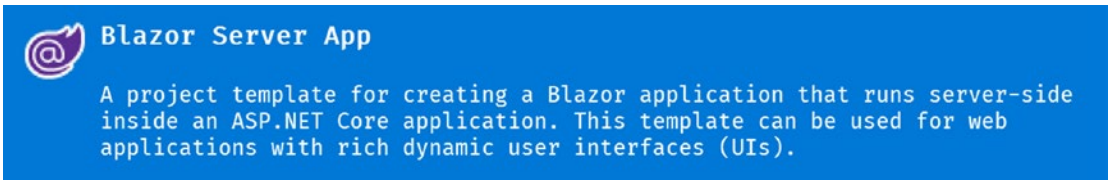

Figure 1-17. *Create a Server-Side Blazor project*

Hit F5 to run your project (since Server-Side Blazor is running on the server, you can easily debug things just like in a regular MVC project). Open the browser's debugger on the network tab and make your page refresh. Compare what gets downloaded now, as in Figure 1-18. As you can see, the total download size is a lot smaller, resulting in your page getting loaded faster. You can also see that a WebSocket is opened between server and browser, allowing the Blazor runtime to exchange UI changes and events.

localhost	200	document	Other	2.0 KB	243 ms
bootstrap.min.css	200	stylesheet	(index)	152 KB	25 ms
site.css	200	stylesheet	(index)	2.5 KB	20 ms
blazor.server.js	200	script	(index)	185 KB	26 ms
open-iconic-bootstrap.min.css	200	stylesheet	(index)	9.2 KB	10 ms
blazor.boot.json	200	fetch	VM229:1	80 B	10 ms
negotiate	200	xhr	VM229:1	285 B	10 ms
open-iconic.woff	200	font	(index)	14.7 KB	9 ms
ng-validate.js	200	script	content-script.js:24	55.6 KB	19 ms
blazor?id=yp4Azf7jjpapv-vMIxWzg	101	websocket	blazor.server.js:1	0 B	Pending
favicon.ico	200	x-icon	Other	31.3 KB	8 ms

Figure 1-18. *Looking at server-side Blazor network activity*

Summary

In this chapter, we installed the prerequisites needed for developing and running Blazor applications. We then created our first Blazor project. This project will be used throughout this book to explain all the Blazor concepts you need to know about. Finally, we examined this solution, looking at the server-side project, the shared project, and the client-side Blazor project, and compared that to a server-side Blazor project.

CHAPTER 2

Data Binding

Imagine an application that needs to display data to the user and capture changes made by that user to save the modified data. One way you could build an application like this is to, once you got the data, iterate over each item of data. For example, for every member of a list, you would generate the same repeating element, and then inside that element, you would generate text boxes, drop-downs, and other UI elements that present data. Later, after the user has made some changes, you would iterate over your generated elements, and for every element, you would inspect the child elements to see if their data was changed. If so, you copy the data back into your objects that will be used for saving that data.

This is an error-prone process and a lot of work if you want to do this with something like jQuery (jQuery is a very popular JavaScript framework which allows you to manipulate the browser's Document Object Model (DOM)).

Modern frameworks like Angular and React have become popular because they simplify this process greatly through *data binding*. With data binding, most of this work for generating UI and copying data back into objects is done by the framework.

A Quick Look at Razor

Blazor is the combination of *Browser* + *Razor* (with a lot of artistic freedom). So, to understand Blazor, we need to understand browsers and Razor. I will assume you understand what a browser is since the Internet has been very popular for over more than a decade. But Razor (as a computer language) might not be that clear (yet). Razor is a markup syntax that allows you to embed code in a web page. Razor can be used to dynamically generate HTML (among others).

Razor made its appearance in ASP.NET MVC. In ASP.NET Core MVC razor is executed at the server-side to generate HTML which is sent to the browser. But in Blazor, this code is executed inside your browser (with Blazor WebAssembly) and will dynamically update the web page without having to go back to the server.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_2

Remember the `MyFirstBlazor` solution we generated from the template in the previous chapter? Open it again with Visual Studio or Code and have a look at `SurveyPrompt.razor` as shown in Listing 2-1.

Listing 2-1. Examining `SurveyPrompt.razor`

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

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

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

As you can see, razor mainly consists of HTML markup. But if you want to have some C# properties or methods, you can embed them in the `@code` section of a razor file. This works because the razor file is used to generate a .NET class and everything in `@code` is embedded in that class. For example, the `SurveyPrompt` component allows you to set the `Title` property, which is set in `Index.razor` as in Listing 2-2.

Listing 2-2. Setting the SurveyPrompt's title (excerpt from `index.razor`)

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

Because the `Title` property can be set in another component, the property becomes a parameter, and because of that, you need to apply the `[Parameter]` attribute, as in

Listing 2-1. `SurveyPrompt` can then embed the contents of the `Title` property in its HTML markup using the `@` syntax (third line in Listing 2-1). This syntax tells razor to switch to C#, and this will get the property and embed its value in the markup.

One-Way Data Binding

One-way data binding is where data flows from the component to the DOM or vice versa, but only in one direction. Data binding from the component to the DOM is where some data, like the customer's name, needs to be displayed. Data binding from the DOM to the component is where a DOM event took place, like the user clicking a button, and we want some code to run.

One-Way Data Binding Syntax

Let's look at an example of one-way data binding in razor. Open the solution we built in Chapter 1 (`MyFirstBlazor.sln`), and open `Counter.razor`, repeated here in Listing 2-3.

Listing 2-3. Examining one-way data binding with `Counter.razor`

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

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

@code {
  private int currentCount = 0;

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

On this page, you get a simple counter, which you can increment by clicking the button as illustrated in Figure 2-1.

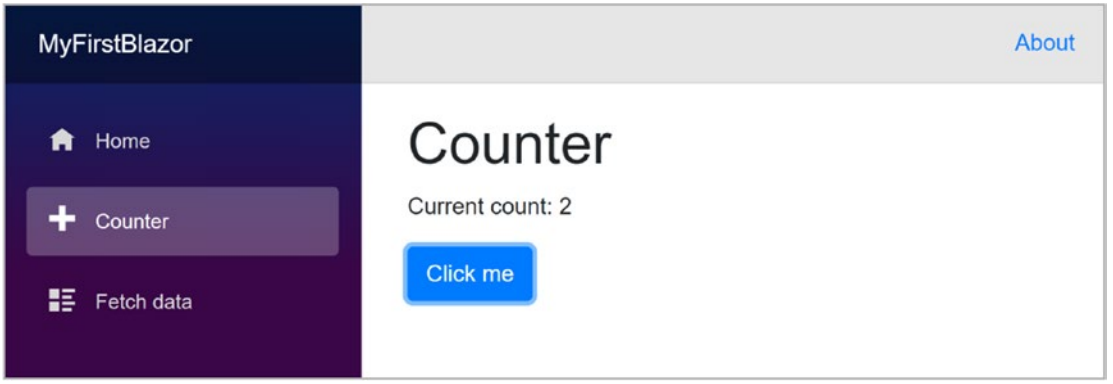

Figure 2-1. *The Counter page*

Let's look at the workings of this page. The `currentCount` field is defined in the `@code` section in `Counter.razor`. This is not a field that can be set from outside so there is no need for the `[Parameter]` attribute.

To display the value of the `counter` in razor, we use the `@currentCount` razor syntax as shown in Listing 2-4.

Listing 2-4. Data binding from the component to the DOM

```
<p>Current count: @currentCount</p>
```

Any time Blazor sees that `currentCount` may have been updated, it will automatically update the DOM with the latest value of `currentCount`.

Attribute Binding

You can also use this same syntax to bind the value of an HTML attribute.

Open `site.css` which you can find in the `wwwroot/css` folder and add these two `css` classes from Listing 2-5.

Listing 2-5. Some simple styles

```
.red-background {
    background: red;
    color: white;
}
```

```
.yellow-background {
    background: yellow;
    color: black;
}
```

Wrap the currentCount in an as in Listing 2-6. Every time you change the value of currentCount, we change its background color.

Listing 2-6. Binding an HTML attribute

```
@page "/counter"

<h1>Counter</h1>

<p>Current count:
  <span class="@backgroundColor">@currentCount</span></p>

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

@code {
  int currentCount = 0;

  string backgroundColor = "red-background";

  void IncrementCount()
  {
    currentCount++;
    backgroundColor = (currentCount % 2 == 0) ?
                      "red-background" : "yellow-background";
  }
}
```

Conditional Attributes

Sometimes you can control the browser by adding some attributes to DOM elements. For example, in Listing 2-7, to disable a button, you can simply use the disabled attribute.

Listing 2-7. Disabling a button using the disabled attribute

```
<button disabled>On Strike</button>
```

With Blazor, you can data-bind an attribute to a Boolean expression (e.g., a property or method of type bool), and Blazor will hide the attribute if the expression evaluates to false (or null) and will show the attribute if it evaluates to true. Go back to the `Counter.razor` and add the code from Listing 2-8.

Listing 2-8. Disabling the `Click me` button

```
<button class="btn btn-primary"
        disabled="@(currentCount >= 10)"
        @onclick="@IncrementCount">
  Click me
</button>
```

Try it. Clicking the button until the `currentCount` becomes `10` will disable the button. As soon as `currentCount` falls below `10`, the button will become enabled again (except there is no way you can do this for the moment).

Event Handling and Data Binding

We update `currentCount` using the `IncrementCount()` method from Listing 2-3. This method gets called by clicking the "Click me" button. This again is a one-way data binding, but in the other direction, from the button to your component. Blazor allows you to react to DOM events (like the click event) this way, instead of using JavaScript. You can also build your own components that have events, where you can use the same syntax to react to them.

Event Binding Syntax

Look at Listing 2-9. Now we are using the `@on<event>` syntax; in this case, we want to bind to the button's `click` DOM event, so we use the `@onclick` attribute on the `button` element, and we pass it the method we want to call.

Listing 2-9. Data binding from the DOM to the component

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

Clicking the button will cause the UI to be updated with the new value of the counter. Whenever the user interacts with the site, for example, by clicking a button, Blazor assumes that the event will have some side effect because a method gets called, so it will update the UI with the latest values. Simply calling a method will not cause Blazor to update the UI. We will discuss this later in this chapter.

Event Arguments

In regular .NET, event handlers of type `EventHandler` can find out more information about the event using the `sender` and `EventArgs` arguments. In Blazor, event handlers don't follow the strict event pattern from .NET, but you can declare the event handler method to take an argument of some type derived from `EventArgs`, for example, `MouseEventArgs`, as shown in Listing 2-10.

Each event uses a specific kind of `EventArgs`, so please refer to online documentation for more information about a specific event.

Listing 2-10. A Blazor event handler taking arguments

```
private void IncrementCount(MouseEventArgs e)
```

Using C# Lambda Functions

Data binding to an event does not always require you to write a method; you can also use C# lambda function syntax with an example shown in Listing 2-11.

Listing 2-11. Event data binding with lambda syntax

```
<button class="btn btn-primary"
        disabled="@(currentCount >= 10)"
        @onclick="@(() => currentCount+=1)">
  Click me
</button>
```

If you want to use a lambda function, you need to wrap it into round braces.

Two-Way Data Binding

Sometimes you want to display some data to the user, and you want to allow the user to make changes to this data. This is common in data entry forms. Here we will explore the two-way data binding syntax.

Two-Way Data Binding Syntax

With two-way data binding, we will have the DOM update whenever the component changes, but the component will also update because of modifications in the DOM. The simplest example is with an `<input>` HTML element.

Let's try something. Modify `Counter.razor` by adding an `increment` field and an `input` using the `@bind` attribute as shown in Listing 2-12.

Listing 2-12. Adding an `increment` and an `input`

```
@page "/counter"

<h1>Counter</h1>

<p>
  Current count:
  <span class="@backgroundColor">@currentCount</span>
</p>

<p>
  <input type="number" @bind="@increment" />
</p>

<button class="btn btn-primary"
        disabled="@(currentCount >= 10)"
        @onclick="IncrementCount">
  Click me
</button>

@code {
  private int currentCount = 0;
  private int increment = 1;
```

```
  private string backgroundColor = "red-background";

  private void IncrementCount(MouseEventArgs e)
  {
    currentCount += increment;
    backgroundColor = (currentCount % 2 == 0) ?
                      "red-background" : "yellow-background";
  }
}
```

Build and run.

You should now be able to increment the counter with other values as in Figure 2-2.

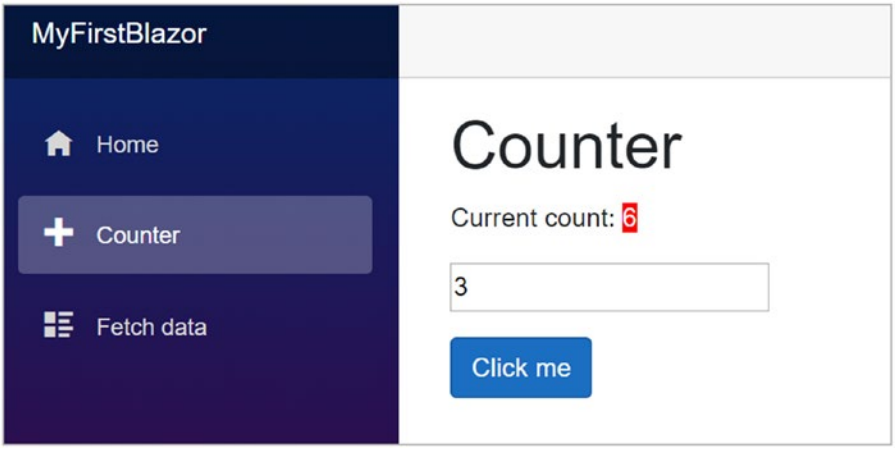

Figure 2-2. *Adding an increment with two-way data binding*

Look at the input element you just added, repeated here in Listing 2-13.

Listing 2-13. Two-way data binding with the @bind syntax

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

Here we are using the @bind syntax which is the equivalent of two different one-way bindings as shown in Listing 2-14.

Listing 2-14. Data binding in both directions

```
<input type="number"
       value="@increment"
       onchange="@((ChangeEventArgs e)
                 => increment = int.Parse($"{e.Value}"))" />
```

This alternative syntax is very verbose and not that handy to use. Using `@bind` is way more practical. However, don't forget about this technique; using the more verbose syntax can sometimes be a more elegant solution!

Binding to Other Events: @bind:{event}

Normally Blazor will update the value in two-way data binding when the onchange event occurs. But maybe this is too late for you. Let's look at how you can change the event that triggers data binding.

Add a second `input` by copying the line from Listing 2-14. Run this example and change the value of one input by typing a number into it. The other input's value will not update immediately. Clicking the other input will update it. This is because we're using the `onchange` event, which triggers when the input loses focus! If you want data binding to occur immediately, you can also bind to other events such as `oninput` by using the explicit `@bind:event` syntax. Update the second input to match Listing 2-15. Typing in the second input will update the first input after each keystroke.

Listing 2-15. Explicit binding to events

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

Preventing Default Actions

In Blazor, you can react to events, and the browser will also react to these. But what if you don't want the browser to behave as normal? Let's say you want to allow the user to increment and decrement an input's value simply by pressing "+" or "–". Change the input to react to the keypress event as in Listings 2-16 and 2-17.

Listing 2-16. Handling keypress events

```
<input type="number"
       @bind="@increment"
       @onkeypress="KeyHandler" />
```

Listing 2-17. The KeyHandler method

```
private void KeyHandler(KeyboardEventArgs e)
{
  Console.WriteLine(e.Code);
  if( e.Key == "+")
  {
    increment += 1;
  }
  if( e.Key == "-")
  {
    increment -= 1;
  }
}
```

Build and run. Pressing "+" and "–" will increment and decrement the value in the input, but you will also see any key you just pressed added to the input html element because this is the default behavior for an input. To stop this default behavior, we can add `@{event}:preventDefault` like in Listing 2-18. Here we use a true boolean constant value to stop the default behavior of the input, but you can use any Boolean expression.

Listing 2-18. Stopping the default behavior of the input

```
private bool shouldPreventDefault = true;
<input type="number"
       @bind="@increment"
       @onkeypress="KeyHandler"
       @onkeypress:preventDefault="true" />
```

Build and run again. Now pressing "+" will increment the input's value as expected.

You can also leave out the value for `preventDefault`, then it will always prevent the default action as in Listing 2-19.

Listing 2-19. Shorter notation

```
<input type="number"
       @bind="@increment"
       @onkeypress="KeyHandler"
       @onkeypress:preventDefault />
```

Stopping Event Propagation

Events also propagate to the parent. Let's look at an example. Start by adding two nested div elements which each handle the `@onmousemove` event as in Listing 2-20.

Listing 2-20. Event propagation example

```
<div style="width: 400px; height: 400px; background: yellow"
     @onmousemove="OuterMouseMove">

  @outerPos

  <div style="width: 300px; height: 300px;
              background: green; margin:50px"
       @onmousemove="InnerMouseMove" >

    @innerPos

  </div>
</div>
```

Also add code from Listing 2-21. These event handlers simply show the mouse position in the element.

Listing 2-21. The event handlers

```
private string outerPos = "Nothing yet";

private void OuterMouseMove(MouseEventArgs e)
  => outerPos = $"Mouse last at {e.ClientX}x{e.ClientY}";

private string innerPos = "Nothing yet";

private void InnerMouseMove(MouseEventArgs e)
  => innerPos = $"Mouse last at {e.ClientX}x{e.ClientY}";
```

Build and run. Move the mouse pointer around in the yellow square. Now do the same for the green rectangle. However, moving the mouse in the green square also updates the yellow one! This is because the mouse move event and others get sent to the element where the event occurs, but also to its parent element, all the way up to the root element! If you want to avoid this, you can stop this propagation by adding the `{event}:stopPropagation` attribute. Add it to the inner square as in Listing 2-22. From now on, moving the mouse in the inner square does not update the outer square.

Listing 2-22. Stopping the event from propagating to the parent

```
<div style="width: 400px; height: 400px; background: yellow"
     @onmousemove="OuterMouseMove">

  @outerPos

  <div style="width: 300px; height: 300px; background: green; margin:50px"
       @onmousemove="InnerMouseMove"
       @onmousemove:stopPropagation>

    @innerPos

  </div>
</div>
```

You can assign a bool expression to this attribute, just like `preventdefault`.

Formatting Dates

Data binding to a `DateTime` value can be formatted with the `@bind:format` attribute as shown in Listing 2-23.

Listing 2-23. Formatting a date

```
<p>
  <h1>Date Format</h1>

  <input @bind="Today" @bind:format="yyyy-MM-dd" />

  <button class="btn btn-primary"
          @onclick="IncrementDate">
    Next Day
  </button>
```

```
</p>
@code {
  private DateTime Today { get; set; } = DateTime.Now;

  private void IncrementDate()
    => Today = Today.AddDays(1);
}
```

In this case, the date will use the Japanese date format (year-month-day – the best one for developers because you can easily compare these, even as a string). Currently, `DateTime` values are the only ones supporting the `@bind:format` attribute.

Reporting Changes

Blazor will update the DOM whenever it thinks changes have been made to your data. One example is when an event executes some of your code, it assumes you've modified some values as a side effect and renders the UI. However, Blazor is not always capable of detecting all changes, and in this case, you will have to tell Blazor to apply the changes to the DOM. A typical example is with background threads. Let us look at an example of this.

Open `Counter.razor` and add another button that will automatically increment the counter when pressed as in Listing 2-24. The `AutoIncrement` method uses a .NET `Timer` instance to increment the `currentCount` every second. A timer instance will run on a background thread, executing the callback delegate at intervals (just like setInterval with JavaScript).

Listing 2-24. Adding another button

```
@page "/counter"

@using System.Threading

<h1>Counter</h1>

<p>
  Current count:
  <span class="@backgroundColor">@currentCount</span>
</p>
```

```
<p>
  <input type="number" @bind="@increment" />
</p>
<button class="btn btn-primary"
        disabled="@(currentCount >= 10)"
        @onclick="IncrementCount">
  Click me
</button>

<button class="btn btn-secondary"
        @onclick="AutoIncrement">Auto Increment</button>

@code {
    private int currentCount = 0;
    private int increment = 1;
    private string backgroundColor = "red-background";

    private void IncrementCount()
    {
      Console.WriteLine("++");
      currentCount += increment;
      backgroundColor = (currentCount % 2 == 0) ?
                        "red-background" :
                        "yellow-background";
    }

    private void AutoIncrement()
    {
      var timer = new Timer(
        callback: (_) => IncrementCount(),
        state: null,
        dueTime: TimeSpan.FromSeconds(1),
        period: TimeSpan.FromSeconds(1));
    }
}
```

> You might find the lambda function argument in the Timer's constructor a little strange. I use an underscore when I need to name an argument that is not used in the body of the lambda function. Call it anything you want, for example, `ignore`; it does not matter. I simply like to use underscore because then I don't have to think of a good name for the argument. C# 7 made this official, it is called discards, and you can find more at https://docs.microsoft.com/en-us/dotnet/csharp/discards.

Run this page. Clicking the "Auto Increment" button will start the timer, but the counter will not update on the screen. Why? Try clicking the "Increment" button. The counter has been updated, so it is a UI problem. If you open the browser's debugger, you will see in the Console tab a ++ appears every second, so the timer works! That's because I've added a `Console.Writeline`, which sends the output to the debugger's console. Sometimes an easy way to see if things are working…

Blazor will re-render the page whenever an event occurs. It will also re-render the page in case of asynchronous operations. However, some changes cannot be detected automatically. In this case, you need to tell Blazor to update the page by calling the `StateHasChanged` method which every Blazor component inherits from its base class.

Go back to the `AutoIncrement` method and add a call to `StateHasChanged` as in Listing 2-25. `StateHasChanged` tells Blazor that some state has changed (Who would have thought!) and that it needs to re-render the page.

Listing 2-25. Adding `StateHasChanged`

```
private void AutoIncrement()
{
  var timer = new Timer(
    callback: (_) =>
    {
```

```
        IncrementCount();
        this.StateHasChanged();
      },
    state: null,
      dueTime: TimeSpan.FromSeconds(1),
      period: TimeSpan.FromSeconds(1));
}
```

Run again. Now pressing "Auto Increment" will work.

As you can see, sometimes we will need to tell Blazor manually to update the DOM.

The Pizza Place Single-Page Application

Let us apply this newfound knowledge and build a nice Pizza ordering website. Throughout the rest of this book, we will enhance this site with all kinds of features.

Create the PizzaPlace Project

Create a new Blazor-hosted project, either using `Visual Studio` or `dotnet cli`. Refer to the explanation on creating a project in the first chapter if you don't recall how. Call the project `PizzaPlace`. You get a similar project to the `MyFirstBlazor` project. Now let's apply some changes!

Out of the box, Blazor uses the popular Bootstrap 4 layout framework, including open-iconic fonts. Expect to see bootstrap and open-iconic (oi) css classes in the code samples. However, you can use any other layout framework, because Blazor uses standard HTML and CSS. This book is about Blazor, not fancy layouts, so we're not going to spend a lot of time choosing nice colors and making the site look great. Focus!

In the server project, throw away `WeatherForecastController.cs`. We don't need weather forecasts to order pizzas. In the shared project, delete `WeatherForecast.cs`. Same thing. In the client project, throw away the `Counter.razor` and `FetchData.razor` files from the `Pages` folder and `SurveyPrompt.razor` from the `Shared` folder. Your solution should look like Figure 2-3.

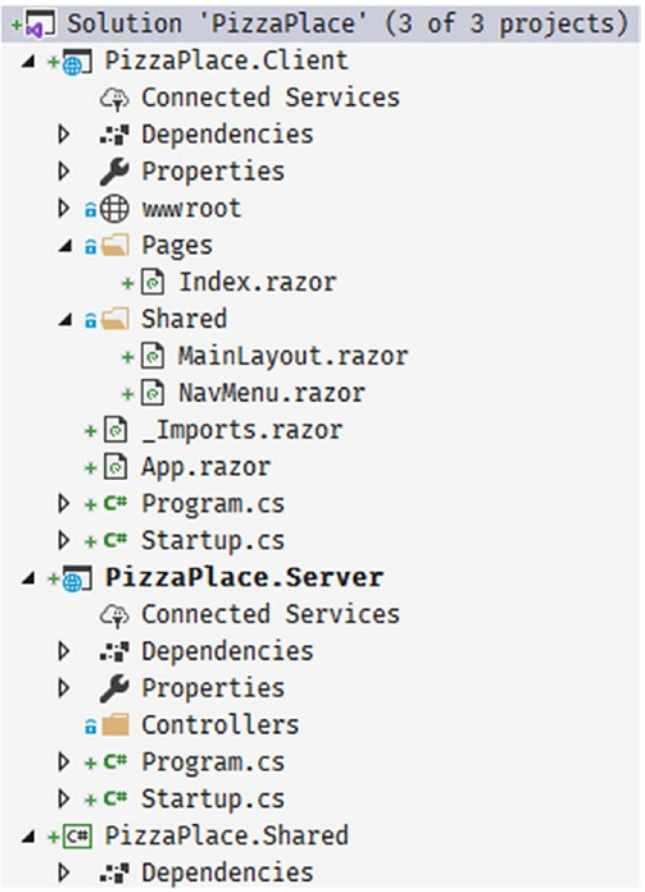

Figure 2-3. *The solution after removing unneeded files*

Add Shared Classes to Represent the Data

In Blazor, it is best to add classes holding data to the `Shared` project. These classes are used to send the data from the server to the client and later to send the data back. What do we need? Start with classes representing a Pizza and how spicy it is as in Listing 2-26.

Listing 2-26. The `Spiciness` and `Pizza` classes

```
using System;

namespace PizzaPlace.Shared
{
  public enum Spiciness
```

```
  {
    None,
    Spicy,
    Hot
  }

  public class Pizza
  {
    public Pizza(int id, string name, decimal price,
                 Spiciness spiciness)
    {
      this.Id = id;
      this.Name = name
        ?? throw new ArgumentNullException(nameof(name),
           "A pizza needs a name!");
      this.Price = price;
      this.Spiciness = spiciness;
    }

    public int Id { get; }
    public string Name { get; }
    public decimal Price { get; }
    public Spiciness Spiciness { get; }
  }
}
```

Our application is NOT about editing pizzas, so I've made this class *immutable*, that is, nothing can be changed once a pizza object has been created. In C#, this is easily done by creating properties with only a getter. You can still set these properties, but only in the constructor.

Next, we will need a class representing the menu we offer. Add a new class to the shared project called `Menu` with the implementation from Listing 2-27.

Listing 2-27. The `Menu` class

```
using System.Collections.Generic;
using System.Linq;
```

```
namespace PizzaPlace.Shared
{
  public class Menu
  {
    public List<Pizza> Pizzas { get; set; }
    = new List<Pizza>();

    public Pizza GetPizza(int id)
    => Pizzas.SingleOrDefault(pizza => pizza.Id == id);

  }

}
```

As in real life, a restaurant's menu is a list of meals, in this case, a pizza meal.

We will also need a `Customer` class in the shared project with implementation from Listing 2-28.

Listing 2-28. The `Customer` class

```
using System.ComponentModel.DataAnnotations;

namespace PizzaPlace.Shared
{
  public class Customer
  {
    public int Id { get; set; }

    public string Name { get; set; }

    public string Street { get; set; }

    public string City { get; set; }

  }
}
```

Each customer has a shopping basket, so add the `Basket` class to the shared project as in Listing 2-29.

Listing 2-29. The `Basket` class, representing the customer's order

```
using System.Collections.Generic;

namespace PizzaPlace.Shared
{
  public class Basket
  {
    public Customer Customer { get; set; } = new Customer();

    public List<int> Orders { get; set; } = new List<int>();

    public bool HasPaid { get; set; } = false;

  }
}
```

Please note that we just keep the pizza id in the `Orders` collection. You will learn why later.

One more class before we group them all together. We'll use a `UI` class to keep track of some UI options, so add this class to the shared project as in Listing 2-30.

Listing 2-30. The `UI` options class

```
namespace PizzaPlace.Shared
{
  public class UI
  {
    public bool ShowBasket { get; set; } = true;
  }

}
```

Finally, we group all these classes into a single `State` class, again in the shared project with implementation from Listing 2-31.

Listing 2-31. The `State` class

```
using System.Linq;

namespace PizzaPlace.Shared
```

```
{
  public class State
  {
    public Menu Menu { get; set; } = new Menu();

    public Basket Basket { get; set; } = new Basket();

    public UI UI { get; set; } = new UI();

  }
}
```

There is another good reason to put all these classes into the Shared project. There is limited debugging for Blazor. By putting these classes into the shared project, we can apply unit testing best practices on the shared classes because it is a regular .NET Core project and even use the Visual Studio debugger to examine weird behavior. The Shared project can also be used by other projects since it is a standard project, for example, a Windows or Xamarin client!

Build the UI to Show the Menu

With the classes in place to represent the data, the next step is to build the user interface that shows the menu. We will start by displaying the menu to the user, and then we will enhance the UI to allow the user to order one or more pizzas.

Display the Menu

The problem of displaying the menu is twofold: Firstly, you need to display a list of data. The menu can be thought of as a list, like any other list. Secondly, in our application, we'll need to convert the spiciness choices from their numeric values into URLs leading to the icons used to indicate different levels of hotness.

Displaying a List of Data

Open `Index.razor`. Add the `@code` section to hold our restaurant's (limited) menu with code from Listing 2-32 by initializing the `State` instance with a `Menu`.

Listing 2-32. Building our application's menu

```
@code {

  private State State { get; } = new State()
  {
    Menu = new Menu
    {
      Pizzas = new List<Pizza>
{
        new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
        new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
        new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
      }
    }
  };

}
```

You will get a bunch of compiler errors. Open the `_Imports.razor` and add a `@using` like in Listing 2-33. All razor files in the `Pages` folder will now automatically recognize the `PizzaPlace.Shared` namespace.

Listing 2-33. Add using statements to `_Imports.razor`

```
@layout MainLayout
@using PizzaPlace.Shared
```

The Pizza Place menu is a list like any other list. You can display it by adding some razor markup in `Index.razor` to generate the menu as HTML as shown in Listing 2-34.

Listing 2-34. Generating the HTML with razor

```
@page "/"
<!-- Menu -->

<h1>Our selection of pizzas</h1>

@foreach (var pizza in State.Menu.Pizzas)
{
```

```
  <div class="row">
    <div class="col">
      @pizza.Name
    </div>
    <div class="col">
      @pizza.Price
    </div>
    <div class="col">
      <img src="@SpicinessImage(pizza.Spiciness)"
           alt="@pizza.Spiciness" />
    </div>
    <div class="col">
      <button class="btn btn-success"
              @onclick="@(() => AddToBasket(pizza))">
        Add
      </button>
    </div>
  </div>
}

<!-- End menu -->
```

I like to use comments to show the start and end of each section on my page. This makes it easier to find a certain part of my page when I come back to it later. In the next chapter, we will convert each section in its own Blazor component, making future maintenance a lot easier to do.

What we are doing here is iterating over each pizza in the menu and generating a row with four columns, one each for the `name`, `price`, `spiciness`, and finally the `order` button. There are still some compiler errors which we will fix next.

Converting Values

We still have a little problem. We need to convert the `spiciness` value to a `URL`, which is done by the `SpicinessImage` method as shown in Listing 2-35. Add this method to the `@code` area of the `Index.razor` file.

Listing 2-35. Converting a value with a converter function

```
private string SpicinessImage(Spiciness spiciness)
  => $"images/{spiciness.ToString().ToLower()}.png";
```

This converter function simply converts the name of the enumeration's value from Listing 2-26 into the URL of an image file which can be found in the Blazor project's `images` folder as shown in Figure 2-4. Add this folder (which can be found in this book's download) to the `wwwroot` folder.

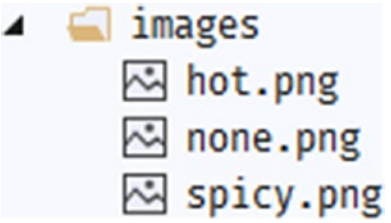

Figure 2-4. *The images folder*

Add Pizzas to the Shopping Basket

Having the menu functioning leads naturally to the adding of pizzas to the shopping basket. When you click the `Add` button, the `AddToBasket` method will be executed with the chosen pizza. You can find the implementation of the `AddToBasket` method in Listing 2-36. To make debugging easier, we're adding a `Console.WriteLine` which will appear in the browser debugger's console.

Listing 2-36. Ordering a pizza

```
private void AddToBasket(Pizza pizza)
{
  Console.WriteLine($"Added pizza {pizza.Name}");
  State.Basket.Add(pizza.Id);
}
```

Our `Basket` class now needs an `Add` method as in Listing 2-37.

Listing 2-37. The Basket's Add method

```
public void Add(int pizzaId)
{
  Orders.Add(pizzaId);
}
```

Look at the `@onclick` event handler for the button from Listing 2-34. Why is this event handler using a lambda? When you order a pizza, you want of course to have your chosen pizza added to the basket. So how can we pass the pizza to `AddToBasket` from Listing 2-36? By using a lambda function, we can simply pass the `pizza` variable used in the `@foreach` loop to it. Using a normal method wouldn't work because there is no easy way to send the selected pizza. This is also known as a *closure* (very similar to JavaScript closures) and can be very practical!

Run the application. You should see Figure 2-5.

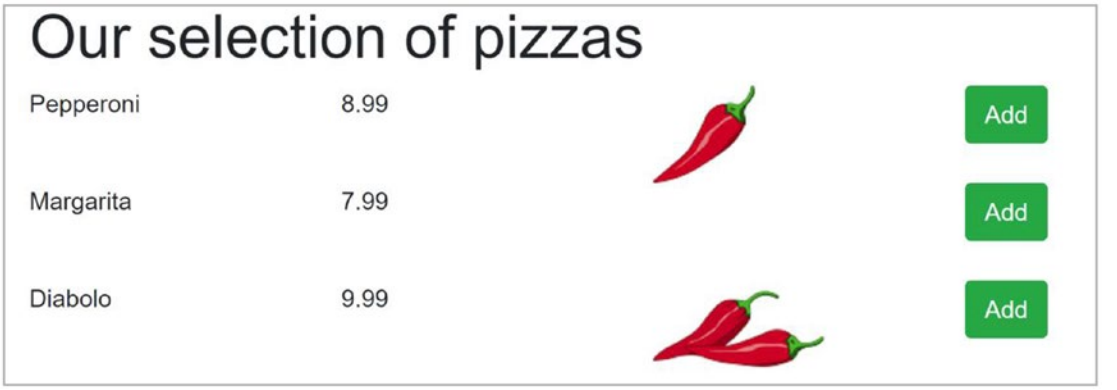

Figure 2-5. *Our Pizza Place's menu*

When you click the `Add` button, you're adding a pizza to the shopping basket. But how can we be sure (since we're not displaying the shopping basket yet)?

Open the browser's debugging tools and look at the `Console`. Each time you click `Add`, you should see some output from the `Console.WriteLine` in the `AddToBasket` method as shown in Figure 2-6.

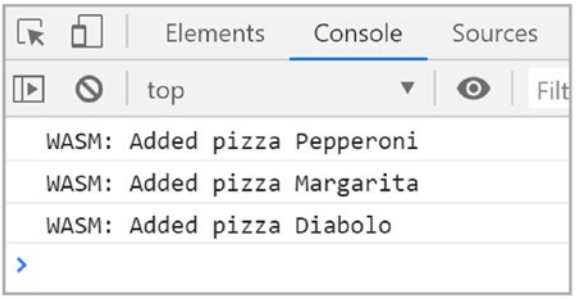

Figure 2-6. *Looking at the Console.WriteLine's output*

Show the Shopping Basket

The next thing on the menu (some pun intended) is displaying the shopping basket. We are going to use a new feature from C# 7 called tuples. I will explain tuples in a moment. This requires adding the `System.ValueTuple` NuGet package.

Add a Package with Visual Studio

To add this NuGet package with Visual Studio, right-click the client project and select `Manage NuGet Packages...` as illustrated in Figure 2-7. Search for the package and install it.

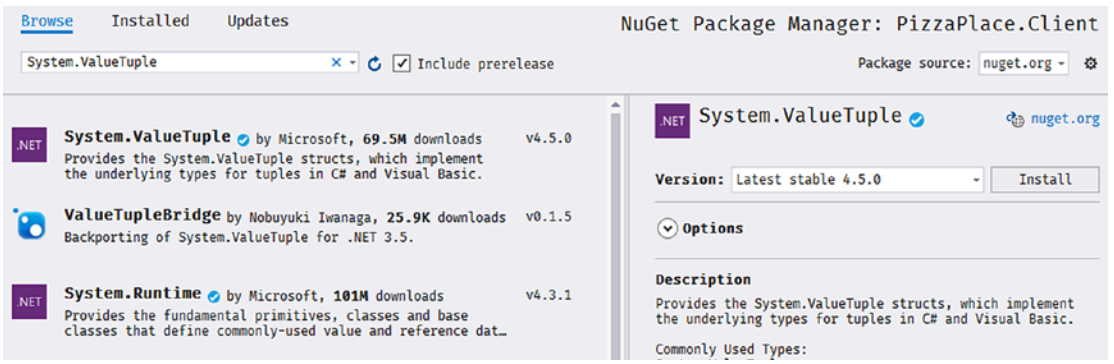

Figure 2-7. *Installing the `System.ValueTuple` package with NuGet*

Add a Package with Visual Studio Code

To add the package with `Visual Studio Code`, select the `PizzaPlace.Client.csproj` file and add a new package reference.

```
<PackageReference Include="System.ValueTuple"
                  Version="4.5.0" />
```

Display the Shopping Basket

Now we are ready to display the shopping basket. Add Listing 2-38 after the menu from Listing 2-34.

Listing 2-38. Displaying the shopping basket

```
<!-- End menu -->
<!-- Shopping Basket -->

@if (State.Basket.Orders.Any())
{
  <h1>Your current order</h1>

  @foreach (var (pizza, pos) in State.Basket.Orders.Select(
   (id, pos) => (State.Menu.GetPizza(id), pos)))
  {
    <div class="row">
      <div class="col">
        @pizza.Name
      </div>
      <div class="col">
        @pizza.Price
      </div>
      <div class="col">
        <button class="btn btn-danger"
                @onclick="@(() => RemoveFromBasket(pos))">
          Remove
        </button>
      </div>
    </div>
```

```
  }
  <div class="row">
    <div class="col"> Total:</div>
    <div class="col"> @State.TotalPrice </div>
    <div class="col"> </div>
  </div>
}

<!-- End shopping basket -->
```

Most of this stuff is very similar, but now we are iterating over a list of tuples (keep reading, a very handy new feature in C# 7 https://docs.microsoft.com/en-us/dotnet/csharp/tuples).

Tuples are very similar to anonymous types from C# in that they let you store and return intermediate multi-part results without you having to build a helper class.

Let's look at this code in a little more detail with Listing 2-39 (repeat from Listing 2-38).

Listing 2-39. Converting the shopping basket for easy display

```
@foreach (var (pizza, pos) in
  State.Basket.Orders.Select(
   (id, pos) => (State.Menu.GetPizza(id), pos)))
```

We are using LINQ's `Select` to iterate over the list of orders (which contain pizza ids). To display the pizza in the shopping basket, we need a pizza, so we convert the id to a pizza with the `GetPizza` method from the `Menu`.

Let's look at the lambda function used in the `Select` shown in Listing 2-40.

Listing 2-40. Creating tuples (repeat from Listing 2-38)

```
(id, pos) => (State.Menu.GetPizza(id), pos)))
```

The LINQ `Select` method has two overloads, and we're using the overload taking an element from the collection (`id`) and the position in the collection (`pos`). We use these to create tuples. Each tuple represents a pizza from the basket and its position in the basket! We could have done the same, creating a little helper class with the pizza and position, but this is now done for us!

The pizza is used to display its name and price, while the position is used in the `Delete` button. This button invokes the `RemoveFromBasket` method from Listing 2-41.

Listing 2-41. Removing items from the shopping basket

```
private void RemoveFromBasket(int pos)
{
  Console.WriteLine($"Removing pizza at pos {pos}");
  State.Basket.RemoveAt(pos);
}
```

And of course, we need to add the `RemoveAt` method to the `Basket` class as in Listing 2-42.

Listing 2-42. The `Basket` class's `RemoveAt` method

```
public void RemoveAt(int index)
{
  Orders.RemoveAt(index);
}
```

At the bottom of the shopping basket, the total order amount is shown. This is calculated by the `State` class. Add the `TotalPrice` method from Listing 2-43 to the `State` class. Don't forget to add a `using System.Linq` statement to the top.

Listing 2-43. Calculating the total price in the `State` class

```
public decimal TotalPrice
  => Basket.Orders.Sum(id => Menu.GetPizza(id).Price);
```

Run the application and order some pizzas. You should see your current order similar to Figure 2-8.

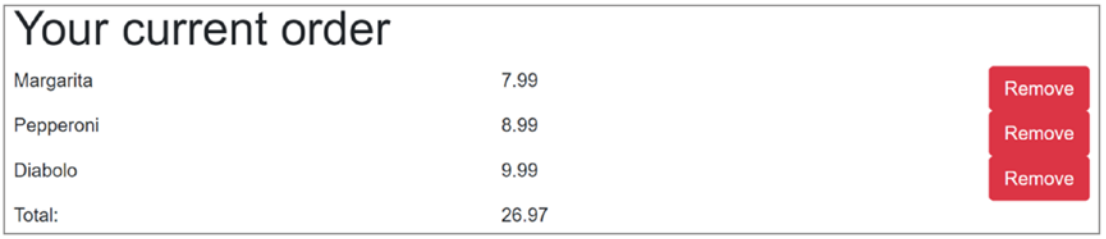

Figure 2-8. *Your shopping basket with a couple of pizzas*

Enter the Customer

Of course, to complete the order, we need to know a couple of things about the customer, especially we need to know the customer's name and address because we need to deliver the order.

Start by adding the following razor to your `Index.razor` page as in Listing 2-44.

Listing 2-44. Adding form elements for data entry

```
<!-- End shopping basket -->
<!-- Customer entry -->

<h1>Please enter your details below</h1>

<fieldset>
  <p>
    <label for="name">Name:</label>
    <input id="name" @bind="State.Basket.Customer.Name" />
  </p>
  <p>
    <label for="street">Street:</label>
    <input id="street" @bind="State.Basket.Customer.Street" />
  </p>
  <p>
    <label for="city">City:</label>
    <input id="city" @bind="State.Basket.Customer.City" />
  </p>

  <button @onclick="PlaceOrder">Checkout</button>

</fieldset>

<!-- End customer entry -->
```

This adds three `labels` and their respective `inputs` for `name`, `street`, and `city`.

You will also need to add the `PlaceOrder` method to your `functions` as shown in Listing 2-45.

Listing 2-45. The `PlaceOrder` method

```
@code {
...
  private void PlaceOrder()
  {
    Console.WriteLine("Placing order");
  }
}
```

The `PlaceOrder` method doesn't do anything yet; we'll send the order to the server later.

Run the application and enter your details, for example, as in Figure 2-9.

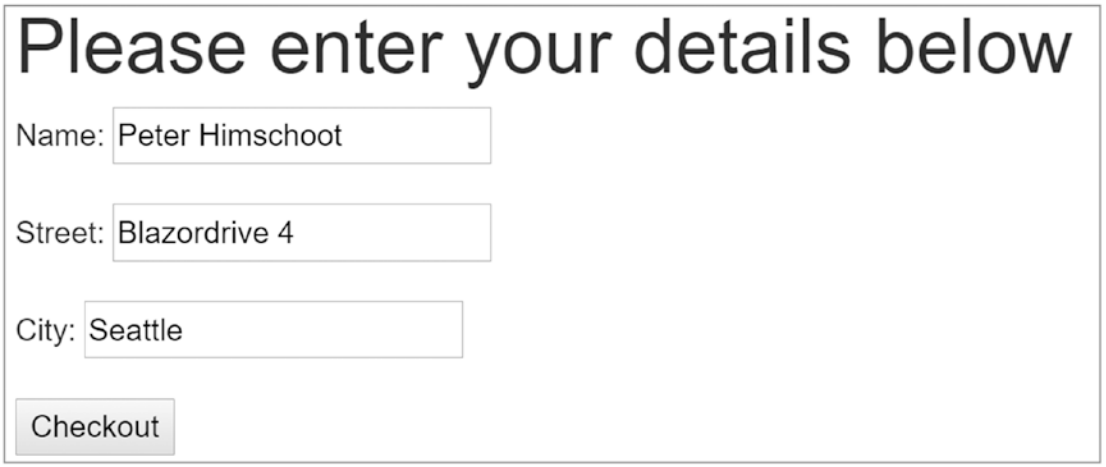

Figure 2-9. *Filling out the customer detail*

Debugging Tip

Blazor WebAssembly has limited debugging, and you want to see the `State` object because it contains the customer's details and order. Will we send the correct information to the server when we press the `Checkout` button? For this, we'll use a simple trick by displaying the `State` on our page, so you can review it at any time. Start by adding a new static class `DebuggingExtensions` to your Blazor client project as in Listing 2-46.

Listing 2-46. The `DebuggingExtensions` class

```
using System.Text.Json;

namespace PizzaPlace.Client
{
  public static class DebuggingExtensions
  {
    public static string ToJson(this object obj)
      => JsonSerializer.Serialize(obj, obj.GetType());
  }
}
```

And at the bottom of `Index.razor`, add a simple paragraph as in Listing 2-47.

Listing 2-47. Showing `State`

```
<!-- End customer entry -->

<p>@State.ToJson()</p>
```

Run your project. As you interact with the page, you'll see `State` change with an example shown in Figure 2-10.

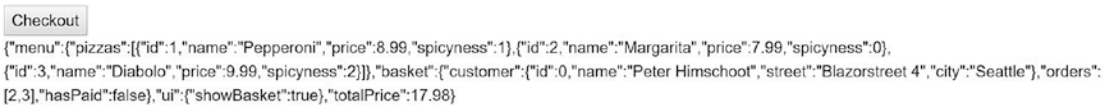

Figure 2-10. *Watching `State` changes*

It should be obvious that we remove this debugging feature when the page is ready 😊.

Validate the Customer Information

But wait! Clicking the `Checkout` button works, even while there is no customer name, address, or city! We need to do some validation! So, let's start with an introduction to Blazor validation.

Let Entities Validate Themselves

Classes like `Customer` should validate themselves because they have the best knowledge about the validity of their properties. .NET has a couple of built-in validation mechanisms, and here we are going to use the standard `System.ComponentModel.DataAnnotations`. With this system, you add attributes to your entity's properties, indicating what kind of validation is required.

Start by adding the `System.ComponentModel.Annotations` package to the `PizzaPlace.Shared` project.

Now add `Required` attributes to the `Customer` class as in Listing 2-48. These annotations make the `Name`, `Street`, and `City` properties mandatory. Use the `ErrorMessage` property to set the validation error message. You can add other attributes like `CreditCard`, `EmailAddress`, `MaxLength`, `MinLength`, `Phone`, `Range`, `RegularExpression`, `StringLength`, and `Url` for further validation.

Listing 2-48. Adding annotations for validation

```
using System.ComponentModel.DataAnnotations;

namespace PizzaPlace.Shared
{
  public class Customer
  {
    public int Id { get; set; }

    [Required(ErrorMessage = "Name is required")]
    [StringLength(50)]
    public string Name { get; set; }

    [Required(ErrorMessage = "Street is required")]
    [StringLength(50)]
    public string Street { get; set; }

    [Required(ErrorMessage = "City is required")]
    [StringLength(50)]
    public string City { get; set; }

  }
}
```

Use FormField and InputText to Enable Validation

Blazor comes with some built-in components that will perform validation for you. Replace the customer entry UI with Listing 2-49. The `EditForm` component will render as the HTML `<form>` element, and it is used to wrap all the inputs that require validation. Set the `Model` property to the instance you need to validate. When the user clicks the `submit` button, the `EditForm` component performs validation, and when there are no validation errors, it will call the `OnValidSubmit` event.

Use the `InputText` component for each field, binding it to the model's property using the `@bind-Value` attribute. Listing 2-49 has three such `InputText` components, one for `Name`, `Address`, and `City`. Other input components also exist for other types, such as `InputDate`, `InputCheckbox`, `InputSelect`, and `InputNumber`. You can even build your own, which we will discuss in the chapter on components.

Listing 2-49. Using `EditForm`, `InputText`, and `DataAnnotationsValidator`

```
<!-- Customer entry -->
<h1>Please enter your details below</h1>

<EditForm OnValidSubmit="PlaceOrder"
          Model="@State.Basket.Customer">
  <DataAnnotationsValidator />

  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="FirstName">Name:</label>
    <div class="col-sm-9">
      <InputText class="form-control"
        @bind-Value="@State.Basket.Customer.Name" />
      <ValidationMessage
        For="@(() => State.Basket.Customer.Name)" />
    </div>
  </div>
  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="LastName">Street:</label>
    <div class="col-sm-9">
```

```
      <InputText class="form-control"
        @bind-Value="@State.Basket.Customer.Street" />
      <ValidationMessage
        For="@(() => State.Basket.Customer.Street)" />
    </div>
  </div>
  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="Birthday">City:</label>
    <div class="col-sm-9">
      <InputText class="form-control"
        @bind-Value="@State.Basket.Customer.City" />
      <ValidationMessage
        For="@(() => State.Basket.Customer.City)" />
    </div>
  </div>
  <div class="form-group mb-0">
    <button type="submit"
            id="BtnRegister" class="btn btn-primary">
      Checkout
    </button>
  </div>
</EditForm>

<!-- End customer entry -->
```

Show Validation Errors

.NET has several validation choices, so we need to tell the `EditForm` which validation technique to use. Do this by adding the `DataAnnotationsValidator` component inside the `EditForm` element as in Listing 2-49, because we are using data annotations for validation.

To show the validation message for each input, you add a `ValidationMessage` component and you set the `For` property to a delegate that returns the field to show validation messages for.

Build and run the `PizzaPlace` project. Click the `Checkout` button. You should get validation errors as shown in Figure 2-11. Blazor validation also adds some styles, and by default, this will put a red border around inputs with validation errors.

Figure 2-11. *Showing validation errors*

Note that the `Checkout` button does not invoke the `PlaceOrder` method if there are validation errors.

Now enter a name, street, and city. You should see the validation errors go away as in Figure 2-12. You will also see green borders appear since the inputs are now valid.

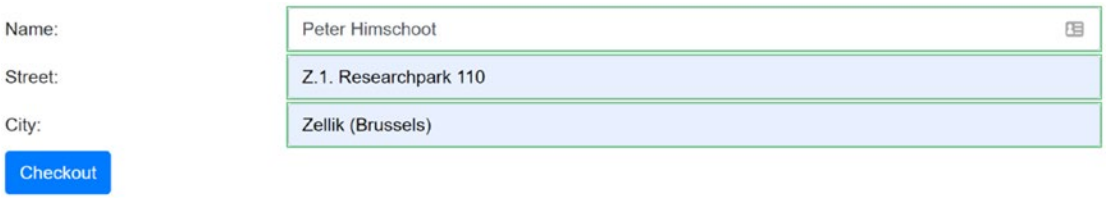

Figure 2-12. *Showing valid inputs after validation*

You can also use a `ValidationSummary` component which shows all the validation errors together as an unordered list. For example, you can add the `ValidationSummary` component below the `DataAnnotationsValidator` as in Listing 2-50. This will show all validation errors as in Figure 2-13.

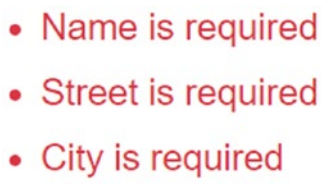

Figure 2-13. *The `ValidationSummary`'s output*

Listing 2-50. Using the `ValidationSummary` component

```
<EditForm OnValidSubmit="PlaceOrder"
          Model="@State.Basket.Customer">
  <DataAnnotationsValidator />
  <ValidationSummary />
```

Customize the Validation Feedback

When you enter a value in an <input> element, Blazor validation gives you feedback about the validity of the value by adding certain CSS classes. Let us have a look at how this is implemented. Run the `PizzaPlace` project and right-click one of the inputs and then select `Inspect` from the menu.

Initially, an untouched input will have the `valid` class, as in Listing 2-51. The `form-control` class comes from the `class` attribute in Listing 2-51.

Listing 2-51. Validation uses the `valid` CSS class

```
<input class="form-control valid">
```

When you make a valid change to an input, the `modified` class is added as in Listing 2-52.

Listing 2-52. Validation adds the `modified` class after a change

```
<input class="form-control modified valid">
```

With an invalid input, you get the `invalid` class, as in Listing 2-53.

Listing 2-53. Bad input uses the invalid css class

```
<input class="form-control modified invalid">
```

Finally, validation messages get the `validation-message` css class.

So, if you want to customize how this feedback looks, you customize your CSS rules. For example, you can use add following CSS to the end of `site.css` from Listing 2-54 to make validation look like Figure 2-14.

Listing 2-54. Some custom CSS rules to change validation feedback

```
form-control.invalid {
  border-left: 5px solid #a94442; /* red */
}

form-control.valid.modified {
  border-left: 5px solid #42A948; /* green */
}
```

Figure 2-14. *Customized validation feedback*

Summary

In this chapter, we looked at data binding in Blazor. We started with one-way data binding where we can embed the value of a property or a field in the UI using the `@SomeProperty` syntax. We then looked at event binding where you bind an element's event to a method using the `on<event>="@SomeMethod"` syntax. Blazor also has support for two-way data binding where we can update the UI with the value of a property and vice versa using the `@bind="SomeProperty"` syntax. Finally, we examined validation where we can use standard .NET validation techniques.

CHAPTER 3

Components and Structure for Blazor Applications

In the previous chapter on data binding, you have built a single monolithic application with Blazor. After a while, this will become harder and harder to maintain.

In modern web development, we build applications by constructing them from components, which typically are again built from smaller components. A Blazor component is a self-contained chunk of user interface. Blazor components are classes built from razor and C# with one specific purpose (also known as *single responsibility principle*) and are easier to understand, debug, and maintain. And of course, you can use the same component in different pages.

What Is a Blazor Component?

To put it in a simple manner, each `razor` file in Blazor is a component. It's that simple! A razor file in Blazor contains markup and has code in the `@code` section. Each page we have been using from the `MyFirstBlazor` project is a component! And components can be built by adding other components as children.

Remember the `MyFirstBlazor` project from the previous chapter? Open it in Visual Studio (or Code) and let's have a look at some of the components in there.

Open `Index.razor` as in Listing 3-1.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_3

Listing 3-1. The Index page

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

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

See SurveyPrompt? That is one of the components that are part of the Blazor template. It takes one parameter Title which we can set where we want to use the component. Let us have a good look at the SurveyPrompt component.

Examining the SurveyPrompt Component

Open SurveyPrompt.razor as in Listing 3-2, which can be found in the Shared folder of the client project.

The component is called SurveyPrompt because a component gets named after the razor file it is in.

Listing 3-2. The SurveyPrompt component

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

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

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

Look at the Razor markup. This is a simple Blazor component that displays an icon in front of the `Title` as shown in Figure 3-1 and then displays a link to the survey (a real survey which you should take 😊 – this will show Microsoft that you're interested in Blazor).

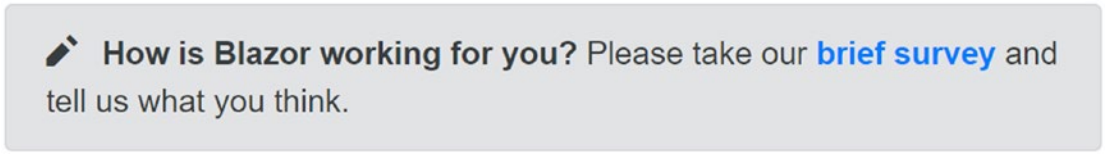

Figure 3-1. *The `SurveyPrompt` component*

The `@code` section simply contains a property `Title` that uses one-way data binding for rendering in the component. Do note the `[Parameter]` attribute. This is required for components that want to expose their public properties to the parent component. This way we can pass data to nested components, for example, how the `Index` component passes the `Title` to the `SurveyPrompt` component.

Building a Simple Alert Component with Razor

Let us build our own Blazor component that will show a simple alert. Alerts are used to draw the attention of the user to some message, for example, a warning.

Create a New Component with Visual Studio

Open the `MyFirstBlazor` solution.

Right-click the `Pages` folder and select `Add` ➤ `New Item...`.

The `Add New Item` window should open as in Figure 3-2.

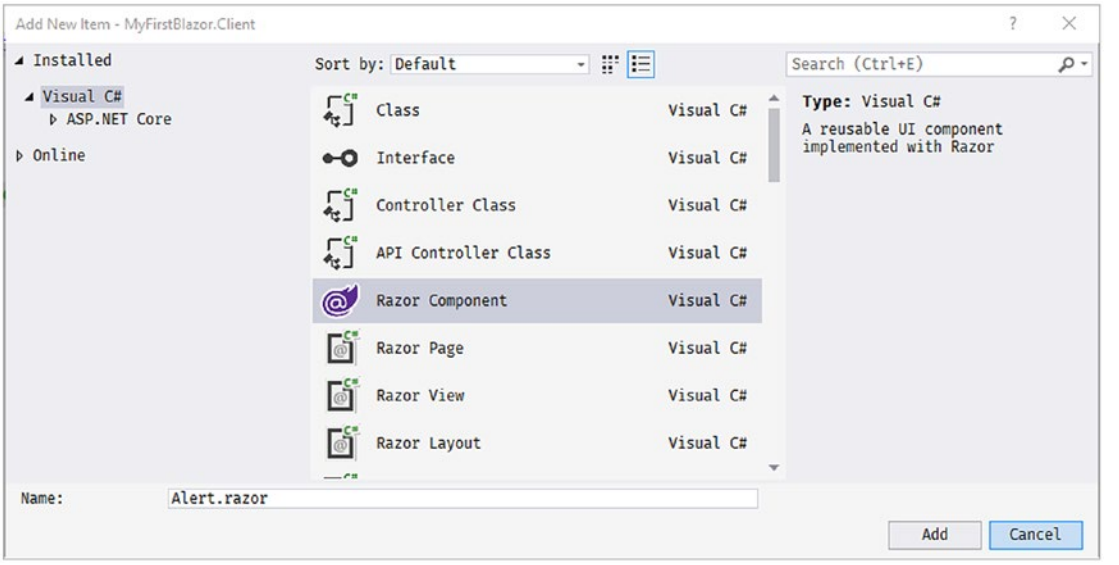

Figure 3-2. *The Add New Item window*

Select `Razor Component` and name it `Alert.razor`. Click `Add`.

Create a New Component with Code

Right-click the `Pages` folder of the client project and select `New File`. Name it `Alert.razor`.

Implement the Alert Component

Remove all existing content from `Alert.razor` and replace it with Listing 3-3. The first line in the `Alert` component uses an `@if` to hide or show its inner content. This is a common technique if you want to conditionally display content. So, if `Show` is `false`, the whole component is not shown. Inside the `@if` there is an `<div>` element with `@ChildContent` as its child. You use `@ChildContent` if you want to access the nested element in the `Alert` component, as you'll see when we use the `Alert` component. The `@ChildContent` will hold this content and needs to be of type `RenderFragment` because this is the way the Blazor engine passes it (we will look at this later in this chapter).

Listing 3-3. The `Alert` component

```
@if (Show)
{
  <div class="alert alert-secondary mt-4" role="alert">
    @ChildContent
  </div>
}

@code {

  [Parameter]
  public bool Show { get; set; }

  [Parameter]
  public RenderFragment ChildContent { get; set; }
}
```

The `Alert` component will display whatever content you nest in it (using bootstrap styling).

The default Blazor templates use Bootstrap 4 for styling. Bootstrap (`http://getbootstrap.com`) is a very popular CSS framework, originally built for Twitter, giving easy layout for web pages. However, Blazor does not require you to use bootstrap, so you can use whatever styling you prefer. In that case, you would have to update all the Razor files in the solution use the other styles, just like in regular web development. In this book, we will use bootstrap, simply because it is there.

Go back to `Index.razor` and add the `Alert` element.

Visual Studio and Code are smart enough to provide you with IntelliSense, as illustrated in Figure 3-3, for the `Alert` component and its parameters!

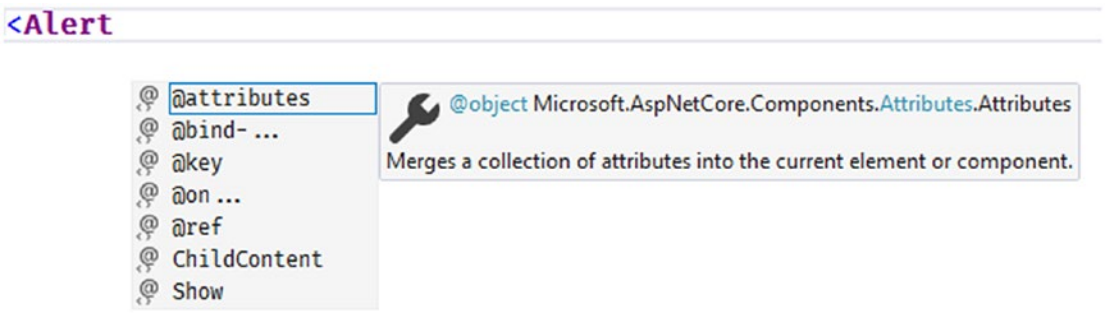

Figure 3-3. *Visual Studio IntelliSense support for custom Blazor components*

Complete the `Alert` and add a button as in Listing 3-4.

Listing 3-4. Using our `Alert` component

```
@page "/"

<Alert Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</Alert>

<button @onclick="@ToggleAlert">Toggle</button>

@code {
  public bool ShowAlert { get; set; } = true;

  public void ToggleAlert()
  => ShowAlert = !ShowAlert;
}
```

Inside the `<Alert>` tag, there is a `<span>` displaying a checkmark icon and a `<strong>` element displaying a simple message. These will be set as the `@ChildContent` property of the `Alert` component. Build and run your project. When you click the `<button>`, it calls the `ToggleAlert` method which will hide and show the `Alert` as in Figure 3-4.

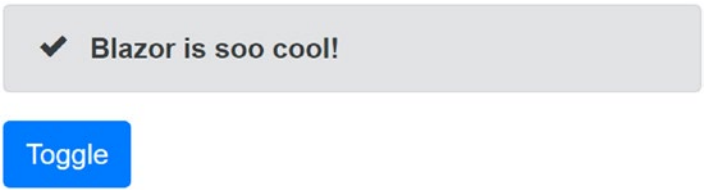

Figure 3-4. *Our simple `Alert` component before clicking the Toggle button*

Separating View and View Model

You might not like this mixing of markup (view) and code (view model). If you like, you can use two separate files, one for the view using razor and another for the view model using C#. The view will display the data from the view model, and event handlers in the view will invoke methods from the view model. Some people prefer this way of working because it's more like the MVVM pattern. Let's try this! Each Blazor component gets generated as a C# partial class. If you want to separate the code from the razor file, put the code in a partial class with the same name as the component. The C# compiler will merge code from both files into a single class.

Create a DismissableAlert Component

If you haven't done this yet, open the `MyFirstBlazor` solution. With Visual Studio, right-click the `Pages` folder and select `Add` ➤ `New Item...`. The `Add New Item` dialog should open as shown in Figure 3-2. This time select `Razor Component` and name it `DismissableAlert.razor`. Also, add a new C# class, and call the file `DismissableAlert.cs`.

With Visual Studio Code, right-click the `Pages` folder, select `New File`, and name it `DismissableAlert.razor`. Do this again to create a new file called `DismissableAlert.cs`.

A `DismissableAlert` is an alert with a little x button, which the user can click to dismiss the alert. It is quite similar to the previous `Alert` component. Replace the markup in the `razor` file with Listing 3-5.

Listing 3-5. The markup for `DismissableAlert.razor`

```
@if (Show)
{
  <div class="alert alert-secondary alert-dismissable fade show mt-4"
  role="alert">
    @ChildContent

    <button type="button" class="close" data-dismiss="alert"
            aria-label="Close" @onclick="@Dismiss">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
}
```

There is no `@code` section, because you will write this in the .cs file. Replace the C# code in `DismissableAlert.cs` with Listing 3-6.

Listing 3-6. The code for `DismissableAlert.cs`

```
using Microsoft.AspNetCore.Components;

namespace MyFirstBlazor.Client.Pages
{
  public partial class DismissableAlert
  {
    [Parameter]
    public bool Show { get; set; }

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

    public void Dismiss()
      => Show = false;
  }
}
```

Do note that this is a partial class with the same name as the Blazor component! So instead of putting your code in the `@code` section of a razor file, you can put the code in a partial class.

Which model is best? I don't think either one is better than the other; it is more a matter of taste. Choose the one you like.

Understanding Parent-Child Communication

Parent and child components typically communicate through data binding. For example, in Listing 3-7 we are using our `DismissableAlert`, which communicates with the parent component through the Parent's `ShowAlert` property. Clicking the `Toggle` button will hide and show the alert. You can try this by replacing the contents of `Index.razor` with Listing 3-7.

Listing 3-7. Using `DismissableAlert`

```
@page "/"

<DismissableAlert Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>

<button @onclick="@ToggleAlert">Toggle</button>

@code {
  public bool ShowAlert { get; set; } = true;

  public void ToggleAlert()
  => ShowAlert = !ShowAlert;
}
```

Add a Timer Component

Start by adding a new class called `Timer` to the `Pages` folder as shown in Listing 3-8 (the timer will not have any visual part, so we don't even need a `.razor` file to build the view).

A Blazor component is a class that inherits the `ComponentBase` class. Since we want to use the `Timer` class as a Blazor component, we need to inherit from `ComponentBase`.

This `Timer` class will invoke a delegate (`Tick`) after a certain number of seconds (`TimeInSeconds`) have expired. The `Tick` parameter is of type `Action`, which is one of the built-in delegate types of .NET. An `Action` is simply a method returning a `void` with no parameters. There are other generic `Action` types, such as `Action<T>` which is a method returning `void` with one parameter of type `T`. This allows the parent component to set the `Action`, so the child will execute the `Action` (in this case, after `TimeInSeconds` has expired).

Listing 3-8. The Timer class

```
using Microsoft.AspNetCore.Components;
using System;
using System.Threading;

namespace MyFirstBlazor.Client.Pages
{
  public class Timer : ComponentBase
```

```
  {
    [Parameter]
    public double TimeInSeconds { get; set; }

    [Parameter]
    public Action Tick { get; set; }

    protected override void OnInitialized()
    {
      var timer = new System.Threading.Timer(
        callback: (_) => InvokeAsync(() => Tick?.Invoke()),
        state: null,
        dueTime: TimeSpan.FromSeconds(TimeInSeconds),
        period: Timeout.InfiniteTimeSpan);
    }
  }
}
```

Now add the `Timer` component to the `Index` page as in Listing 3-9. With this change, the `Timer` component will invoke the `ToggleAlert` method after 5 seconds.

Listing 3-9. Adding the Timer component to dismiss the alert

```
@page "/"

<DismissableAlert Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>

<Timer TimeInSeconds="5" Tick="@ToggleAlert" />

<button @onclick="@ToggleAlert">Toggle</button>

@code {
  public bool ShowAlert { get; set; } = true;

  public void ToggleAlert()
  {
    Console.WriteLine("+++");
```

```
    ShowAlert = !ShowAlert;
  }
}
```

Run the application and wait at least 5 seconds. The alert does not hide! Why?!

Look at the markup, which is in Listing 3-9, for `DismissableAlert`. It shows the component based on the `Show` parameter, and this one gets set through data binding. Does the `ToggleAlert` method get called? Run the Blazor website again, and immediately open the browser's debugger on the Console tab. After a little while, you should see the `Console.WriteLine` output appear. So the `ToggleAlert` method does get called.

Think about this. We invoke a method asynchronously using a `Timer`. When the timer fires, we set the `Index` component `ShowAlert` property to false. But we still need to update the UI. You can manually trigger the UI to update by calling the `StateHasChanged` method.

This is very important! The Blazor runtime updates the UI automatically when an event triggers, like the button click. The Blazor runtime also updates the UI for its own asynchronous methods, but not for our own asynchronous methods like `Timer` does.

Now add a call to `StateHasChanged` in the `ToggleAlert` method. Run again, wait, and after 5 seconds, the alert disappears! To be honest, I don't like the previous solution to our problem. Because a child component calls the `ToggleAlert` method we manually need to call `StateHasChanged`. Is there no better way? And we haven't even solved another problem. When the user dismissed the alert before the timer triggered the `Tick` method, it should reappear after 5 seconds because it will set `ShowAlert` back to true!

We will fix both problems, but first, we need to understand two-way data binding between components.

Use Two-Way Component to Component Data Binding

When the user clicks the `DismissableAlert` component's close button, it sets its own `Show` property to `false`, as intended. The problem is that the parent `Index` component's `ShowAlert` stays `true`. Changing the value of the `DismissableAlert` local `Show` property will not update the `Index` component's `ShowAlert` property. What we need is two-day data binding between components, and Blazor has that.

With two-way data binding, changing the value of the Show parameter will update the value of the ShowAlert property of the parent and vice versa.

You can use the @bind-<<NameOfProperty>> syntax to data-bind any property of a child component. This will use two-way data binding. So update the Index page to use two-way data binding as in Listing 3-10.

Listing 3-10. Using two-way data binding

```
<DismissableAlert @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>
```

Run the website. However, you will not see any valid page. The Blazor runtime encountered a problem. You can discover the problem by opening the browser's debugger. Check the Console. You will see a bunch of red messages, one of them stating

```
Object of type 'Blazor.Components.Client.Pages.DismissableAlert' does not
have a property matching the name 'ShowChanged'
```

Properties that support two-way data binding need a way to tell the parent that the property has changed. Blazor uses a delegate, so the parent component can install some logic when the property has changed. The component is responsible for invoking the delegate when the property changes.

Open the DismissableAlert class and change the Show property implementation as in Listing 3-11.

Here we add an extra parameter which should be called <<yourproperty>>Changed of type Action<<typeofyourproperty>>. For example, the property is named Show of type bool, so we add ShowChanged of type Action<bool>.

Listing 3-11. The DismissableAlert class with two-way binding support

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

namespace MyFirstBlazor.Client.Pages
{
  public partial class DismissableAlert
  {
```

```
    private bool show;
    [Parameter]
    public bool Show
    {
      get => this.show;
      set
      {
        if (value != this.show)
        {
          this.show = value;
          ShowChanged?.Invoke(this.show);
        }
      }
    }

    [Parameter]
    public Action<bool> ShowChanged { get; set; }

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

    public void Dismiss()
      => Show = false;
  }
}
```

Whenever someone or something changes the `Show` property's value, the property's setter triggers the `ShowChanged` delegate. This means the parent component can inject some code into the `ShowChanged` delegate property which will invoke when the property is changed (internally or externally). If your component has a property, `Show`, for example, you want to use in two-way data binding, you will need to have a `ShowChanged` handler like in the preceding example. Also ensure that `ShowChanged` is of type `Action<T>`, where `T` is the type of the property!

When the `DismissableAlert Show` property changes, Blazor will update the parent's `ShowAlert` property because we are using two-way data binding. We could update the UI by calling `StateHasChanged` in the `ToggleAlert` method from Listing 3-9. This works, but this is still an ugly solution because you need to consider two-way data binding. This should just work!

There is a better (but still not the best) way as in Listing 3-12. Here we call StateHasChanged whenever the ShowAlert property gets a new value. This is better because anywhere we update the ShowAlert property, we update the UI.

Listing 3-12. Update the UI when ShowAlert changes the value

```
@page "/"

<DismissableAlert @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>

<Timer TimeInSeconds="5" Tick="@ToggleAlert" />

<button @onclick="@ToggleAlert">Toggle</button>

@code {

  private bool showAlert = true;

  public bool ShowAlert
  {
    get => showAlert;
    set
    {
      if (value != showAlert)
      {
        showAlert = value;
        this.StateHasChanged();
      }
    }
  }

  public void ToggleAlert()
  {
    Console.WriteLine("+++");
    ShowAlert = !ShowAlert;
  }
}
```

Run. Wait 5 seconds.

The `Alert` should automatically hide as illustrated in Figures 3-5 and 3-6.

Figure 3-5. *The Alert being shown*

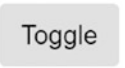

Figure 3-6. *The Alert automatically hides after 5 seconds*

Should your project still not update, you can debug a client-side Blazor project by adding some `Console.WriteLine` statements. These will appear in the browser's `Console` window. You can see examples of this in the book's code which you can download from the Apress site.

Better Use an EventCallback

Now, with the `DismissableAlert` component from the previous section, we have been using two-way data binding between components with the `@bind-Show` syntax, and we used the `ShowChanged` callback to notify the parent component that the `Show` property has changed. To make the parent update its UI, we also added a call to `StateHasChanged` when the parent's `ShowAlert` property gets modified. But there is an even better way!

Blazor has the `EventCallback` type for this. The `EventCallback` type was added to Blazor in .NET Core 3.0 Preview 3 (`https://github.com/aspnet/AspNetCore/issues/6351`). The big difference between `Action<T>` and `EventCallback<T>` is that the latter will invoke `StateHasChanged` for you!

Update the `DismissableAlert` component's `ShowChanged` as in Listing 3-13.

Listing 3-13. Using EventCallback<T>

```
[Parameter]
public EventCallback<bool> ShowChanged { get; set; }
```

Also update the `DismissableAlert` component's `Show` property to invoke the `ShowChanged` callback as in Listing 3-14.

Listing 3-14. Invoking the EventCallback

```
private bool show;
[Parameter]
public bool Show
{
  get => this.show;
  set
  {
    if (value != this.show)
    {
      this.show = value;
      ShowChanged.InvokeAsync(this.show);
    }
  }
}
```

You should also update the `Timer` component to use an `EventCallback` as in Listing 3-15.

Listing 3-15. The improved `Timer` component

```
using Microsoft.AspNetCore.Components;
using System;
using System.Threading;

namespace MyFirstBlazor.Client.Pages
{
  public class Timer : ComponentBase
  {
    [Parameter]
    public double TimeInSeconds { get; set; }

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

```
    protected override void OnInitialized()
    {
      var timer = new System.Threading.Timer(
        callback: (_) => InvokeAsync(() => Tick.InvokeAsync(null)),
        state: null,
        dueTime: TimeSpan.FromSeconds(TimeInSeconds),
        period: Timeout.InfiniteTimeSpan);
    }
  }
}
```

Finally, update the `Index` component's `ShowAlert` property by removing the call to `StateHasChanged` as in Listing 3-16.

Listing 3-16. Index's `ShowAlert` property

```
private bool showAlert = true;

public bool ShowAlert
{
  get => showAlert;
  set
  {
    if (value != showAlert)
    {
      showAlert = value;
      // this.StateHasChanged();
    }
  }
}
```

Build and run. Wait 5 seconds. The alert should hide!

In general, you should prefer `EventCallback<T>` over normal delegates for parent-child communication, such as events and two-way data binding.

Referring to a Child Component

Instead of using data binding in the interaction between the parent and child component, you can also directly interact with a child component. Let's look at an example: We want the alert to disappear by calling its `Dismiss` method. Update your code to match Listing 3-17, where we use the `@ref` syntax to place a reference to a component in a field. Please make sure the field is of the component's type.

Listing 3-17. Referring to a child component

```
@page "/"

<DismissableAlert @ref="alert" @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>

<Timer TimeInSeconds="5" Tick="@(() => alert.Dismiss())" />

<button @onclick="@ToggleAlert">Toggle</button>

@code {

  private DismissableAlert alert;

  public bool ShowAlert { get; set; } = true;

  public void ToggleAlert()
  {
    ShowAlert = !ShowAlert;
  }
}
```

In this example, the Blazor runtime will put a reference to the `DismissableAlert` component in the `alert` field. You can instruct Blazor to do this using the `@ref` syntax. When the timer calls the `Tick` method after five seconds, we can use this reference to call the `DismissableAlert`'s `Dismiss` method.

Communicating with Cascading Parameters

When a higher-level component wants to pass data to an immediate child, life is easy. Simply use data binding. But when a higher-level component needs to share some data with a deeper nested component, passing data using data binding requires each intermediate component to expose that data through a parameter and pass it down to the next level. Blazor solves this problem with *cascading values* and *parameters*. Let us look at an example.

Open `MyFirstBlazor` and add the `CounterData` class from Listing 3-18.

Listing 3-18. The CounterData class

```
using System;

namespace MyFirstBlazor.Client.Pages
{
  public class CounterData
  {
    private int count;
    public int Count
    {
      get => this.count;
      set
      {
        if (value != this.count)
        {
          this.count = value;
          CountChanged?.Invoke(this.count);
        }
      }
    }

    public Action<int> CountChanged { get; set; }
  }
}
```

Use the CascadingValue Component

Our top-level component (called Grandmother) wants to pass this data as a cascading value to any descendant component. You can use the CascadingValue component for this. Look at Listing 3-19 for an example of using the CascadingValue component. Here we pass GrandMother's data field (of type CounterData) as a cascading value. Any component which is part of the ChildContent will now be able to access the CounterData instance from GrandMother.

Listing 3-19. Use the CascadingValue component to pass data to descendants

```
<h3>GrandMother</h3>

@data.Count

<CascadingValue Value="@this.data">
  @ChildContent
</CascadingValue>

@code {

    public CounterData data = new CounterData { Count = 10 };

    protected override void OnInitialized()
    {
      this.data.CountChanged += (newCount) =>
        this.StateHasChanged();
    }

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

}
```

Add the GrandMother Component

Open Index.razor and add the GrandMother Component as in Listing 3-20. This component has two child components, one is a direct GrandChild component (which we will build after this) and another is a GrandChild component wrapped in a DismissableAlert component. This last component knows nothing about CounterData or GrandMother. Still, the GrandMother component will be able to pass its cascading value to the GrandChild component.

Listing 3-20. Using the GrandMother component

```
@page "/"

@using MyFirstBlazor.Components

<DismissableAlert @ref="alert" @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>

<Timer TimeInSeconds="5" Tick="@(() => alert.Dismiss())" />

<button @onclick="@ToggleAlert">Toggle</button>

<GrandMother>
  <GrandChild/>
  <DismissableAlert Show="true">
    <GrandChild/>
  </DismissableAlert>
</GrandMother>

@code {

    MyFirstBlazor.Components.DismissableAlert x;

    private DismissableAlert alert;

    public bool ShowAlert { get; set; } = true;

    public void ToggleAlert()
    {
      ShowAlert = !ShowAlert;
    }
}
```

Receive the Cascading Value

The `GrandChild` component can be found in Listing 3-21. This component has a property of type `CounterData`, and it will receive it from `GrandMother` by adding the `CascadingParameter` attribute. Both `GrandMother` and `GrandChild`(ren) now are sharing the same instance of `CounterData`.

Listing 3-21. Receiving the cascading value

```
<h3>GrandChild</h3>

<button @onclick="Increment">Inc</button>

@code {

    [CascadingParameter()]
    public CounterData gmData { get; set; }

    private void Increment()
    {
      gmData.Count += 1;
    }

}
```

Handle Changes

When you click the Inc button of `GrandChild`, the `CounterData`'s `Count` property increments. The `GrandMother` components want to display this value every time it gets incremented, so `CounterData` notifies the `GrandMother` of changes. The `GrandMother` component subscribes to these changes and calls `StateHasChanged` to update itself. How the shared object handles this notification is up to you, for example, `CounterData` uses a delegate. You could also use `INotifyPropertyChanged`.

Resolving Ambiguities

What if there are several components exposing the same type of cascading value? In this case, you can name the cascading value. For example, you can name the `GrandMother`'s cascading value like in Listing 3-22.

Listing 3-22. Use a named cascading value in GrandMother

```
<CascadingValue Value="@this.data" Name="gm">
  @ChildContent
</CascadingValue>
```

The `GrandChild` component should then receive the cascading value like in Listing 3-23.

Listing 3-23. Receive the named cascading value

```
[CascadingParameter(Name = "gm")]
```

Using Templated Components

Components are Blazor's building block for reuse. Blazor also supports *templated components* where you can specify one or more UI templates as parameters, making templated components even more reusable! For example, your application could be using grids all over the place. You can now build a templated component for a `Grid` taking the type used in the grid as a parameter (very much like you can build a generic type in .NET) and specify the UI used for each item separately! Let's look at an example.

Create the Grid Templated Component

Open the `MyFirstBlazor` project you have been using before. Now add a new `Razor Component` to the `MyFirstBlazor.Client` project's `Pages` folder and name it `Grid` as in Listing 3-24. This is a templated component because it states the `TItem` as a type parameter using the `@typeparam TItem` syntax. This is like a generic type stated in C# with `public class List<T>` where `T` is a type parameter.

You can have more than one type parameter; simply list each type parameter using the `@typeparam` syntax.

Listing 3-24. The `Grid` templated component

```
@typeparam TItem

<table border="1">
  <thead>
    <tr>@Header</tr>
  </thead>
  <tbody>
    @foreach (var item in Items)
    {
      <tr>@Row(item)</tr>
```

```
    }
  </tbody>
  <tfoot>
    <tr>@Footer</tr>
  </tfoot>
</table>

@code {
  [Parameter]
  public RenderFragment Header { get; set; }

  [Parameter]
  public RenderFragment<TItem> Row { get; set; }

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

  [Parameter]
  public IReadOnlyList<TItem> Items { get; set; }
}
```

The `Grid` component has four parameters. The `Header` and `Footer` parameters are of type `RenderFragment` which represents some markup (HTML, Blazor components) which we can specify when we use the `Grid` component (we will look at an example right after explaining the `Grid` component further). Look for the `<thead>` element in Listing 3-24 in the `Grid` component. Here we use the `@Header` razor syntax telling the `Grid` component to put the HTML for the `Header` parameter here. Same thing for the `Footer`.

The `Row` parameter is of type `RenderFragment<TItem>` which is a generic version of `RenderFragment`. In this case, you can specify HTML with access to the `TItem` allowing you access to properties and methods of the `TItem`. The `Items` parameter here is an `IReadOnlyList<TItem>` which can be data-bound to any class with the `IReadOnlyList<TItem>` interface. Look for the `<tbody>` element in Listing 3-24. We iterate over all the items (of type `TItem`) of the `IReadOnlyList<TItem>` and we use the `@Row(element)` razor syntax to apply the `Row` parameter, passing the current item as an argument.

Use the Grid Templated Component

Now let's look at an example of using the `Grid` templated component. Open the `FetchData.razor` component in the `MyFirstBlazor.Client` project. Replace the `<table>` (comment the `<table>` because we will come back to it in later chapters) with the `Grid` component as in Listing 3-25.

The `FetchData` component uses a couple of things such as `@page` and `@inject` we will discuss in later chapters, so bear with the example.

The `FetchData` component uses the `Grid` component specifying the `Items` parameter as the `forecasts` array of `WeatherForecast` instances. Look again at the type of `Items` in the `Grid` component: `IReadOnlyList<TItem>`. The compiler is smart enough to infer from this that the `Grid`'s type parameter (`TItem`) is the `WeatherForecast` type.

Listing 3-25. The `FetchData` component

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

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from the server.</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  @*<table class="table">
      ...
    </table>*@

  <Grid Items="forecasts">
    <Header>
      <th>Date</th>
```

```
      <th>Temp. (C)</th>
      <th>Temp. (F)</th>
      <th>Summary</th>
    </Header>
    <Row Context="forecast">
      <!-- by default called context-->
      <td>@forecast.Date</td>
      <td>@forecast.TemperatureC</td>
      <td>@forecast.TemperatureF</td>
      <td>@forecast.Summary</td>
    </Row>
    <Footer>
      <td colspan="4">Spring is in the air!</td>
    </Footer>
  </Grid>
}

@code {
  private WeatherForecast[] forecasts;

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

Now look at the `<Header>` parameter of the `Grid` component in Listing 3-25. This syntax will bind whatever is inside the `<Header>` to the `Grid`'s `Header` parameter. In this example, we specify some table headers. The grid will put these inside the `<tr>` element from Listing 3-24. Again the `<Footer>` is similar.

Examine the `<Row>` parameter in Listing 3-25. Inside the `<Row>` we want to use the current item from the iteration in Listing 3-24. But how should we access the current item? By default, Blazor will pass the item as the `context` argument (of type `TItem`), so you would access the date of the forecast instance as `@context.Date`. But you can override the name of the argument, and this is what we do with the `Context` parameter (provided by Blazor) using `<Row Context="forecast">`. Now the item from the iteration can be accessed using the `forecast` argument.

Run your solution and select the `Fetch` data link from the navigation menu. Admire your new templated component as in Figure 3-7!

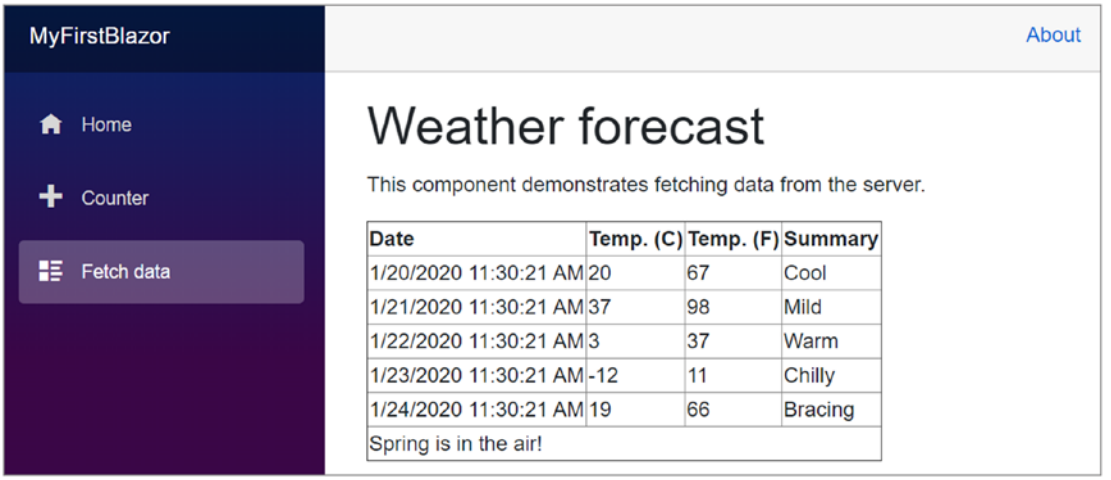

Figure 3-7. *Showing forecasts with the `Grid` templated component*

Now we have a reusable `Grid` component that we can use to show any list of items passing the list to the `Items` parameters and specifying what should be shown in the `Header`, `Row`, and `Footer` parameters! But there is more!

Specify the Type Parameter's Type Explicitly

Normally the compiler can infer the type of the type parameter, but if this does not work as you expect, you can specify the type explicitly. Simply specify the type of your type parameter by specifying it as `TItem` when you use the component as in Listing 3-26.

Listing 3-26. Explicitly specifying the type parameter

```
<Grid Items="forecasts" TItem="WeatherForecast">
```

Razor Templates

You can also specify a `RenderFragment` or `RenderFragment<TItem>` using Razor syntax. A *Razor template* is a way to define a UI snippet, for example, `@<Row>...</Row>`. In this case, we specify a `RenderFragment` without any arguments. But if you need to pass the

argument to the Razor template, you use a lambda function. Let's look at an example. Start by adding a new component called `ListView` as in Listing 3-27. This will show an unordered list of items (of type `TItem`) using `<ul>` and `<li>` HTML elements.

Listing 3-27. The `ListView` templated component

```
@typeparam TItem

<ul>
  @foreach (var item in Items)
  {
    @ItemTemplate(item)
  }
</ul>

@code {
  [Parameter]
  public RenderFragment<TItem> ItemTemplate { get; set; }

  [Parameter]
  public IReadOnlyList<TItem> Items { get; set; }
}
```

Now update the `FetchData` component as in Listing 3-28. Here we specify the `<ListView>`'s `<ItemTemplate>` which is of type `RenderFragment<TItem>` using a Razor template. Look at the `forecastTemplate` in Listing 3-28. This uses a lambda function taking the `forecast` as an argument which returns a `RenderFragment<TItem>` using the `@<li>...</li>` Razor syntax.

In the `<ListView>` component's `<ItemTemplate>`, we simply invoke the template as if it was a lambda function.

Listing 3-28. Using Razor templates to specify the `RenderFragment`

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

<h1>Weather forecast</h1>
```

```
<p>This component demonstrates fetching data from the server.</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  @*<table class="table">
      ...
    </table>*@

  <Grid Items="forecasts" TItem="WeatherForecast">
    ...
  </Grid>

  <ListView Items="forecasts">
    <ItemTemplate>
      @forecastTemplate(context)
    </ItemTemplate>
  </ListView>
}

@code {
  RenderFragment<WeatherForecast> forecastTemplate = (forecast)
    =>@<li>@forecast.Date.ToLongDateString() - @forecast.Summary</li>;

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

Razor templates are a great way to reuse a UI snippet because you can invoke it in different components.

You can also call a razor template directly in your component as in Listing 3-29.

Listing 3-29. Invoking a razor template in your component

```
@forecastTemplate(
  new WeatherForecast {
    Date = DateTime.Now,
    TemperatureC = 26,
    Summary = "Nice!"
  })
```

Building a Component Library

Components should be reusable. But you don't want to reuse a component between projects by copy-pasting the component between them. In this case, it is much better to build a *component library*, and as you will see, this is not hard at all! By putting your Blazor components into a component library, you can include it into different Blazor projects, use it both for Client-Side Blazor and Server-Side Blazor, and even publish it as a NuGet package! What we will do here is to move the `DismissableAlert` and `Timer` component to a library and then we will use this library in our Blazor project.

Create the Component Library Project

With Visual Studio

Right-click your solution, and select `Add New Project`. Look for the `Razor Class Library` project template as in Figure 3-8.

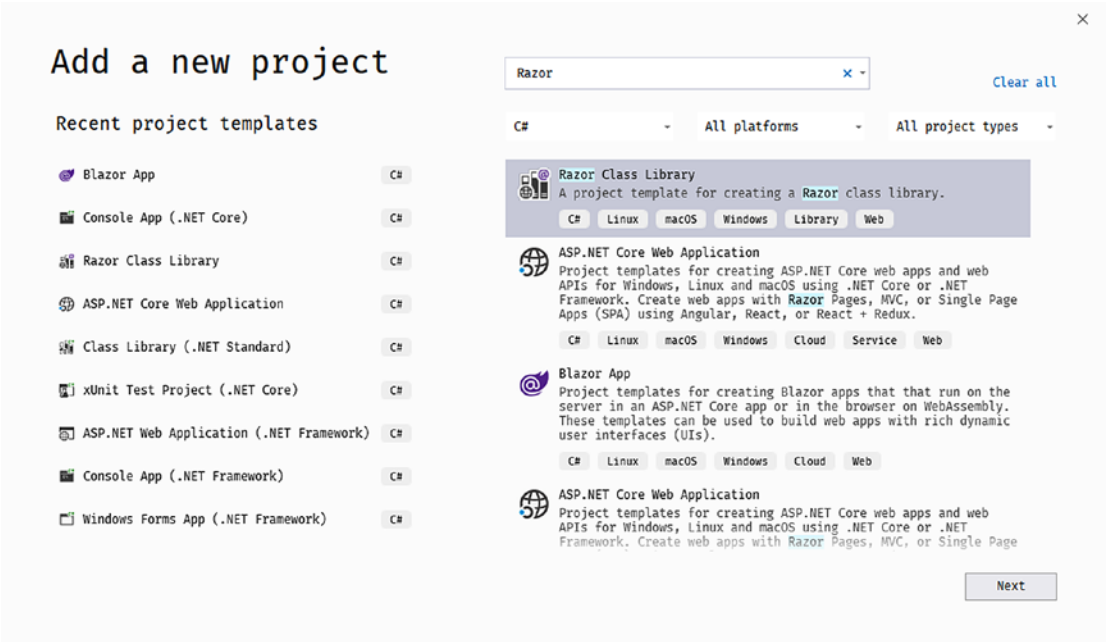

Figure 3-8. *Add a new component library project*

Click `Next`. Name this project `MyFirstBlazor.Components` and click `Create`. In the next screen, click `Create` again.

With Code

Open a command prompt or use the integrated terminal from Visual Studio Code (you can use `Ctrl-`` as a shortcut to toggle the terminal in Code). Change the current directory to the solution folder for `MyFirstBlazor`. Type in the following command:

```
dotnet new razorclasslib -o MyFirstBlazor.Components
```

The `dotnet new` command will create a new project based on the `razorclasslib` template. If you want the project to be created in a subdirectory, you can specify it using the `-o <<subdirectory>>` parameter.

Executing this command should show you output like

```
The template "Razor Class Library" was created successfully.
```

Add it to your solution by typing in the next command:

```
dotnet sln add MyFirstBlazor.Components
```

Add Components to the Library

Previously, we built a couple of components. Some of these are very reusable, so we will move them to our library project. Start with `Timer`.

Move (you can use Shift+Drag-and-Drop) the `Timer.cs` file from your `Client` project to the `Components` project. You should see a new `Timer.cs` file as illustrated in Figure 3-9.

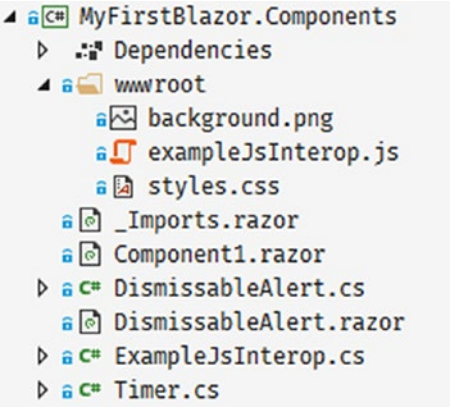

Figure 3-9. *Copying the `Timer.cs` file to the Components project*

Do the same for `DismissableAlert.razor`. Both components are still using the client's namespace, so update their namespace to `MyFirstBlazor.Components`.

Building the solution will still get compiler errors from the client project because we need to add a reference from the client project to the component library, which we will fix in the next part.

Refer to the Library from Your Project

Now that our library is ready, we are going to use it in our project. The way the library works, we can also use it in other projects. Hey, you could even make it into a NuGet package and let the rest of the world enjoy your work!

Referring to Another Project with Visual Studio

Start by right-clicking your client project and select `Add` ➤ `Reference...`.

Visual Studio will show Figure 3-10.

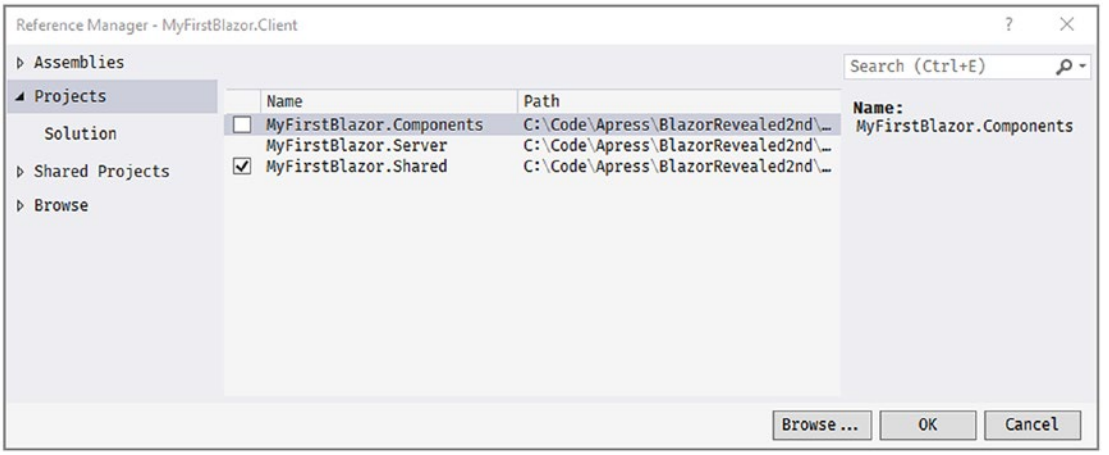

Figure 3-10. *Add a reference to another project*

Make sure you check `MyFirstBlazor.Components` and click OK. Blazor component libraries are just another kind of library/assembly.

Referring to Another Project with Code

Open the `MyFirstBlazor.Client.csproj` file and add another `<ProjectReference>` element to it as in Listing 3-30. It's the last `<ProjectReference>` you need to add.

Listing 3-30. Add a reference to another project

```
<ItemGroup>
  <ProjectReference
    Include="..\..\MyFirstBlazor.Components\MyFirstBlazor.Components.csproj"
  />
  <ProjectReference Include="..\Shared\MyFirstBlazor.Shared.csproj" />
</ItemGroup>
```

Now you have added the component library to your project, but if you want to use the components, you will have to refer to your component library in your `razor` files.

Using the Library Components

Now that you have added the reference to the component library, you can use these components like any other component, except that these components live in another namespace. Just like in C#, you can use the fully qualified name to refer to a component like in Listing 3-31.

Listing 3-31. Using the fully qualified component name

```
<MyFirstBlazor.Components.DismissableAlert
  @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</MyFirstBlazor.Components.DismissableAlert>
```

And like in C#, you can add a `using` statement so you can use the component's name as in Listing 3-32. Add `@using` statements to the top of the razor file.

Listing 3-32. Add a `@using` statement in razor

```
@using MyFirstBlazor.Components

<DismissableAlert @ref="alert" @bind-Show="@ShowAlert">
  <span class="oi oi-check mr-2" aria-hidden="true"></span>
  <strong>Blazor is soo cool!</strong>
</DismissableAlert>
```

With razor, you can add the `@using` statement to the `_Imports.razor` file as in Listing 3-33 which will enable you to use the namespace in all the `.razor` files which are in the same directory or subdirectory. The easiest way to think about this is that Blazor will copy the contents of the `_Imports.razor` file to the top of every `.razor` file in that directory and subdirectory.

Listing 3-33. Add a @using to _Imports.razor

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

Your solution should compile now and run just like before.

Styling Components in a Component Library

You can easily style the components from a component library. Open the `wwwroot` folder from the `MyFirstBlazor.Components` project. Now open the `styles.css` file you will find here and replace its contents with Listing 3-34. This will change the border around the alert.

Listing 3-34. Add a CSS rule

```
.alert-secondary {
  border-color: #bb9c15;
}
```

You can also include other static content in a `component library`. Add all static content you need inside the `wwwroot` folder, such as images, JavaScript files, and other resources. You can then refer to this static resource using a URL that uses the content path to the resource. This URL should start with `_content/{LibraryProjectName}`. For example, to refer to the `styles.css` file in the `MyFirstBlazor.Components` project, add the link from Listing 3-35 to the `index.html` file.

Listing 3-35. Referring to static content in a component library

```
<link href="_content/MyFirstBlazor.Components/styles.css"
      rel="stylesheet" />
```

Running your project should now show a brownish border around the alert.

If you like, you could override the styles from the component library with your own css simply by linking another css file after this one, overriding the styles.

Component Life Cycle Hooks

A Blazor component has a life cycle just like any other .NET object. A component is born, goes through a couple of changes, and then dies. A Blazor component has a couple of methods you can override to capture the life cycle of the component. In this section, we will look at these life cycle hooks because it's very important to understand them very well. Putting code in the wrong life cycle hook will likely break your component. You should also remember that each life cycle method gets called at least once for every component. Even a component with no parameters will see methods like `SetParametersAsync` and `OnParametersSetAsync` called at least once.

OnInitialized and OnInitializedAsync

When your component has been completely initialized, the `OnInitialized` and `OnInitializedAsync` methods are called. Implement one of these methods if you want to do some extra initialization after the component has been created, for example, fetching some data from a server like the `FetchData` component from the `MyFirstBlazor` project.

Use `OnInitialized` for synchronous code as in Listing 3-36.

Listing 3-36. The `OnInitialized` life cycle hook

```
protected override void OnInitialized()
{
}
```

Use `OnInitializedAsync` (Listing 3-37) to call asynchronous methods, for example, making REST calls (we will look at making REST calls in the next two chapters).

Listing 3-37. The `OnInitializedAsync` life cycle hook

```
protected override async Task OnInitializedAsync ()
{
}
```

OnParametersSet and OnParametersSetAsync

When you need one or more parameters for initialization, you use `OnParametersSet` or `OnParametersSetAsync` instead of the `OnInitialized`/`OnInitializedAsync` methods. These methods get called after the parameters have been data-bound. Every time data binding updates one or more of your parameters, these methods get called again, so they are ideal for calculated properties, filtering, and so on. For example, you could have a `DepartmentSelector` component that allows the user to select a department from a company and another `EmployeeList` component that takes the selected department as a parameter. The `EmployeeList` component can then fetch the employees for that department in its `OnParametersSetAsync` method.

Use `OnParametersSet` (Listing 3-38) if you are only calling synchronous methods.

Listing 3-38. The OnParametersSet method

```
protected override void OnParametersSet()
{
}
```

Use `OnParametersSetAsync` (Listing 3-39) if you need to call asynchronous methods. For example, retrieving values from a database that depend on a parameter value should be done in an asynchronous way. In general, any use of methods that take longer than 60 milliseconds should be done asynchronously.

Listing 3-39. The `OnParametersSetAsync` method

```
protected override async Task OnParametersSetAsync()
{
}
```

SetParametersAsync

If you need to execute some code before the parameters are set, you can override the `SetParametersAsync` method. You can find the parameters in the `ParameterView` argument which behaves like a dictionary. Let's look at an example in Listing 3-40. This example uses the `SetParametersAsync` method to inspect the parameters, looking for a "`Param`" parameter. If it is found, we set the parameter; otherwise, we call the base class's `SetParametersAsync`.

There is one snag; when you don't call the base method, the UI doesn't update, so you should call `StateHasChanged`.

Listing 3-40. Overriding `SetParametersAsync`

```
public override async Task
  SetParametersAsync(ParameterView parameters)
{
  if (parameters.TryGetValue(nameof(Param), out int par))
  {
    if (par % 2 == 1)
    {
```

```
            par += 1;
            this.Param = par;
            this.StateHasChanged();
            return;
        }
    }
    await base.SetParametersAsync(parameters);
}
```

OnAfterRender and OnAfterRenderAsync

The `OnAfterRender` and `OnAfterRenderAsync` methods are called after Blazor has completely rendered the component. This means that the browser's DOM has been updated with changes made to your Blazor component. You can use these methods to invoke JavaScript code that needs access to elements from the DOM (which we will cover in the JavaScript Chapter 8). Also avoid calling `StateHasChanged` in this method, as it can cause an infinite loop.

Use `OnAfterRender` shown in Listing 3-41 to call synchronous methods, for example, in JavaScript.

Listing 3-41. The `OnAfterRender` life cycle hook

```
protected override void OnAfterRender(bool firstRender)
{
}
```

Use `OnAfterRenderAsync` as shown in Listing 3-42 to call asynchronous methods, for example, JavaScript methods that return promises or observables.

Listing 3-42. The `OnAfterRenderAsync` life cycle hook

```
protected override Task OnAfterRenderAsync(bool firstRender)
{
}
```

ShouldRender

The `ShouldRender` method returns a Boolean value, indicating if the component should be re-rendered. The component will still render at least once. You want to use this method to stop the component from re-rendering, for example, when making lots of changes resulting in the UI updating too much.

IDisposable

If you need to run some cleanup code when your component is removed from the UI, implement `IDisposable`. You can implement this interface in razor using `@implements`, for example, in Listing 3-43. Normally you put the `@implements` at the top of the `.cshtml` file.

Most of the time dependency injection will take care of calling `Dispose`, so generally, you won't need to implement `IDisposable` if you only need to dispose of your dependencies.

Listing 3-43. Implementing the IDisposable interface in a component

```
@implements IDisposable
```

The `IDisposable` interface requires you to implement a `Dispose` method, which you would put in `@code`, as in Listing 3-44.

Listing 3-44. Implementing the `Dispose` method

```
@code {

  public void Dispose()
  {
    // Cleanup resources here
  }
}
```

If you've separated the view and view model, you implement this interface on the partial class.

Refactoring PizzaPlace into Components

In the previous chapter on data binding, we built a website for ordering pizzas. This used only one component with three different sections. Let us split up this component into smaller, easier to understand components, and try to maximize reuse.

Create a Component to Display a List of Pizzas

Open the `PizzaPlace` Blazor project from the previous chapter. Start by reviewing `index.razor`. This is our main component, and you can say that it has three main sections: a menu, a shopping basket, and customer information.

The menu iterates over the list of pizzas and displays each one with a button to order. The shopping basket also displays a list of pizzas (but now from the shopping basket) with a button to remove it from the order. Looks like both have something in common, they need to display pizzas with an action you choose by clicking the button. So let's create a component to display a list of pizzas, using a nested component to display a pizza's details.

Add a new component to the `Pages` folder called `PizzaItem.razor` with contents from Listing 3-45. You can copy most of the markup from the `Index` component with some changes.

Listing 3-45. The `PizzaItem` component

```
<div class="row">
  <div class="col">
    @Pizza.Name
  </div>
  <div class="col">
    @Pizza.Price
  </div>
  <div class="col">
    <img src="@SpicinessImage(Pizza.Spiciness)"
         alt="@Pizza.Spiciness" />
  </div>
  <div class="col">
    <button class="@ButtonClass"
```

```
            @onclick="@(async () => await Selected.InvokeAsync(Pizza))">
      @ButtonTitle
    </button>
  </div>
</div>

@code {
  [Parameter]
  public Pizza Pizza { get; set; }

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

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

  [Parameter]
  public EventCallback<Pizza> Selected { get; set; }

  private string SpicinessImage(Spiciness spiciness)
      => $"images/{spiciness.ToString().ToLower()}.png";
}
```

The `PizzaItem` component will display a pizza, so it should not come as a surprise that it has a `Pizza` parameter. This component also displays a button, but how this button looks and behaves will differ where we use it. And that is why it has a `ButtonTitle` and `ButtonClass` parameter to change the button's look, and it also has a `Selected` event callback of type `EventCallback<Pizza>` which gets invoked when you click the button. Do you remember why we are using `EventCallback<T>` instead of `Action<T>`?

We can now use this component to display the menu (a list of pizzas). Add a new component to the `Pages` folder called `PizzaList.razor` as in Listing 3-46.

Listing 3-46. The `PizzaList` component

```
<h1>@Title</h1>

@foreach (var pizza in Menu.Pizzas)
{
  <PizzaItem Pizza="@pizza"
             ButtonClass="btn btn-success"
```

```
            ButtonTitle="Order"
            Selected="@Selected" />
}

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

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

  [Parameter]
  public EventCallback<Pizza> Selected { get; set; }
}
```

The `PizzaList` component displays a `Title` and all the pizzas from the `Menu`, so it takes these as parameters. It also takes a `Selected` event callback which you invoke by clicking the button next to a pizza. Note that the `PizzaList` component uses the `PizzaItem` component to display each pizza and that the `PizzaList Selected` event callback is passed directly to the `PizzaItem Selected` event callback. The `Index` component will set this callback, and it will be executed by the `PizzaItem` component.

With `PizzaItem` and `PizzaList` ready, we can use them in `Index`, which you can find in Listing 3-47.

Listing 3-47. Using the `PizzaList` component in `Index.razor`

```
<!-- Menu -->

<PizzaList Title="Our selection of pizzas"
           Menu="@State.Menu"
           Selected="@( async (pizza)
             => AddToBasket(pizza))" />

<!-- End menu -->
```

Run the application and try to order a pizza. Your selected pizza should be added to the shopping basket. Thanks to the `EventCallback<T>` type, there is no need to call `StateHasChanged`. Had we used an `Action<T>` or `Func<T>`, the UI would not update, and you would need to call `StateHasChanged` whenever you receive events from a child component!

Show the ShoppingBasket Component

Add a new razor component called ShoppingBasket.razor to the Pages folder and change its contents to Listing 3-48.

Listing 3-48. The ShoppingBasket component

```
@if (Basket.Orders.Any())
{
  <h1>@Title</h1>

  @foreach (var (pizza, pos) in Pizzas)
  {
    <PizzaItem Pizza="@pizza"
               ButtonTitle="Remove"
               ButtonClass="btn btn-danger"
               Selected="@(async (_)
                 => await Selected.InvokeAsync(pos))" />
  }
  <div class="row">
    <div class="col"> Total:</div>
    <div class="col"> @TotalPrice </div>
    <div class="col"> </div>
  </div>
}

@code {

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

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

  [Parameter]
  public EventCallback<int> Selected { get; set; }

  [Parameter]
  public decimal TotalPrice { get; set; }
```

```
  [Parameter]
  public Func<int, Pizza> GetPizzaFromId { get; set; }

  public IEnumerable<(Pizza pizza, int pos)> Pizzas { get; set; }

  protected override void OnParametersSet()
  {
    base.OnParametersSet();
    Pizzas = Basket.Orders
                   .Select((id, pos)
                    => (pizza: GetPizzaFromId(id), pos: pos));
    TotalPrice = Pizzas.Select(tuple
                    => tuple.pizza.Price).Sum();
  }
}
```

The `ShoppingBasket` component is similar to the `PizzaList` component, but there are some big differences. The basket class keeps track of the order using only ids of pizzas, so we need something to get the pizza object. This is done through the `GetPizzaFromId` delegate. Another change is the `OnParametersSet` method. The `OnParametersSet` method gets called when the component's parameters have been set. Here we override it to build a list of (pizza, position) tuples which we need during data binding and to calculate the total price of the order.

Tuples are just another type in C#. But with modern C#, we get this very convenient syntax, for example, IEnumerable<(Pizza pizza, int pos)> means we have a type that is a list of pizza and position pairs. Think of tuples as a nice replacement for anonymous types, which allow you to quickly have compiler-generated types.

Using the `ShoppingBasket` component in `Index` is easy, as you can see in Listing 3-49.

Listing 3-49. Using the ShoppingBasket component

```
<!-- Shopping Basket -->

<ShoppingBasket Title="Your current order"
                Basket="@State.Basket"
```

```
                GetPizzaFromId="@State.Menu.GetPizza"
                Selected="@( (pos)=> RemoveFromBasket(pos))" />

<!-- End shopping basket -->
```

Add the CustomerEntry Component

Add a new component called `CustomerEntry.razor` to the `Pages` folder as in Listing 3-50.

Listing 3-50. The `CustomerEntry` component

```
<h1>@Title</h1>

<EditForm OnValidSubmit="@Submit"
          Model="@Customer">
  <DataAnnotationsValidator />

  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="FirstName">Name:</label>
    <div class="col-sm-9">
      <InputText class="form-control"
                 @bind-Value="@Customer.Name" />
      <ValidationMessage For="@(() => Customer.Name)" />
    </div>
  </div>
  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="LastName">Street:</label>
    <div class="col-sm-9">
      <InputText class="form-control"
                 @bind-Value="@Customer.Street" />
      <ValidationMessage For="@(() => Customer.Street)" />
    </div>
  </div>
  <div class="form-group row mb-1">
    <label class="col-sm-3 col-form-label"
           for="Birthday">City:</label>
    <div class="col-sm-9">
```

```
      <InputText class="form-control"
                 @bind-Value="@Customer.City" />
      <ValidationMessage For="@(() => Customer.City)" />
    </div>
  </div>
  <div class="form-group mb-0">
    <button type="submit" id="BtnRegister"
            class="@ButtonClass">
      @ButtonTitle
    </button>
  </div>
</EditForm>

@code {

  [Parameter]
  protected string Title { get; set; }

  [Parameter]
  protected string ButtonTitle { get; set; }

  [Parameter]
  protected string ButtonClass { get; set; }

  [Parameter]
  protected Customer Customer { get; set; }

  [Parameter]
  protected EventCallback Submit { get; set; }

}
```

The `CustomerEntry` component uses a `label`, `InputText`, and `ValidationMessage` for each customer property. Now we are ready to complete the `Index` component. Listing 3-51 shows you the whole `Index.razor`.

Listing 3-51. The `Index` component

```
@page "/"
<!-- Menu -->
```

```
<PizzaList Title="Our selection of pizzas"
           Menu="@State.Menu"
           Selected="@( async (pizza)
             => AddToBasket(pizza))" />
<!-- End menu -->
<!-- Shopping Basket -->
<ShoppingBasket Title="Your current order"
                Basket="@State.Basket"
                GetPizzaFromId="@State.Menu.GetPizza"
                Selected="@( (pos) => RemoveFromBasket(pos))"
                />
<!-- End shopping basket -->
<!-- Customer entry -->
<CustomerEntry Title="Please enter your details below"
               ButtonTitle="Checkout"
               ButtonClass="btn btn-primary"
               Customer="@State.Basket.Customer"
               Submit="@PlaceOrder"
               />
<!-- End customer entry -->
<p>@State.ToJson()</p>

@code {

  private State State { get; } = new State()
  {
    Menu = new Menu
    {
      Pizzas = new List<Pizza>
      {
        new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
        new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
        new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
      }
    }
  };
```

```
  private void AddToBasket(Pizza pizza)
  {
    Console.WriteLine($"Added pizza {pizza.Name}");
    State.Basket.Add(pizza.Id);
  }

  private void RemoveFromBasket(int pos)
  {
    Console.WriteLine($"Removing pizza at pos {pos}");
    State.Basket.RemoveAt(pos);
  }

  private void PlaceOrder()
  {
    Console.WriteLine("Placing order");
  }
}
```

Build and run the `PizzaPlace` application. Things should work like before, except for one thing. Remember the debugging tip from the previous chapter? When you change the name of the customer, this tip does not update correctly. Only after pressing the button will this update. Let's fix this.

Use Cascading Properties

The problem is as follows. Whenever the user edits properties from the customer, we want the `CustomerEntry` component to trigger a `CustomerChanged EventCallback`. This way, other components in the UI will update because of changes to the customer. But how can we detect these changes? If we were using `<input>` elements, we could use the `onchanged` event, but unfortunately, the `InputText` component does not have this event.

Examine the InputText Component Internals

Look at the `CustomerEntry` component again. You see an `EditForm` with nested `InputText` components. The `EditForm` provides a cascading value of type `EditContext`, and the `InputText` components use this `EditContext` for things like validation.

If you like, all of this is available on GitHub (https://github.com/dotnet/aspnetcore/tree/master/src/Components) since Blazor is open source. That is what I did to figure out the solution to the problem.

Whenever one of the Input components change, it calls the EditContext.NotifyFieldChanged method. And here is where things get interesting because EditContext has an OnFieldChanged event, which triggers every time a model's property changes.

Let us build a component that uses the EditContext's OnFieldChanged event to notify us of changes.

Catch EditContext from the EditForm

Add a new class to the client project's Pages folder, and name it InputWatcher with the implementation shown in Listing 3-52. The InputWatcher class will have one parameter FieldChanged, of type EventCallback<string>. Internally, the InputWatcher will use the same EditContext instance as the one used by the InputText component. By subscribing to the EditContext's FieldChanged event, all the work will be done by the EditContext instance. You access the EditContext using a cascading parameter.

Listing 3-52. The InputWatcher component

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

namespace PizzaPlace.Client.Pages
{
  public class InputWatcher : ComponentBase
  {
    private EditContext editContext;

    [CascadingParameter]
    public EditContext EditContext
    {
      get => this.editContext;
      set
      {
```

```
        this.editContext = value;
        EditContext.OnFieldChanged += async (sender, e) =>
        {
          await FieldChanged.InvokeAsync(e.FieldIdentifier.FieldName);
        };
      }
    }

    [Parameter]
    public EventCallback<string> FieldChanged { get; set; }

    public bool Validate()
    => EditContext?.Validate() ?? false;
  }
}
```

When the `EditContext` property gets set, the `InputWatcher` simply registers for the `FieldChanged` event and calls its own `FieldChanged` event callback…

Let's use the `InputWatcher` in our `CustomerEntry` component. Add the `InputWatcher` component inside the `EditForm` component, and add a `CustomerChanged` event callback as in Listing 3-53. The `InputWatcher` component invokes the `FieldChanged` method, which triggers the `CustomerChanged` callback.

Listing 3-53. The `CustomerEntry` component with `CustomerChanged` callback

```
<h1>@Title</h1>

<EditForm OnValidSubmit="@Submit"
          Model="@Customer">
  <DataAnnotationsValidator />

  <InputWatcher FieldChanged="@FieldChanged" />

  ...
</EditForm>

@code {
    ...

    [Parameter]
```

```
    public EventCallback<Customer> CustomerChanged { get; set; }

    private void FieldChanged(string fieldName)
    {
      CustomerChanged.InvokeAsync(Customer);
    }
}
```

Build and run. When you make a change to the customer, you should see the customer update in the debugging tip when you tab out of a control. Hey, this was not hard at all!

Disable the Submit Button

You might want to disable the `submit` button as long as there are validation errors. Our freshly introduced `InputWatcher` allows us to do that. Look for the `Validate` method in Listing 3-52. This method calls the `EditContext.Validate` method. We are going to use this to enable/disable the submit button. Start by making the changes from Listing 3-54. First, we add a reference to the `InputWatcher` because we need to call the `Validate` method every time a field changes. Also, add a Boolean field `isInvalid`, and use it to disable the button by binding it to the button's `disabled` attribute. Finally, every time a field gets changed, we update the `isInvalid` by calling the `Validate` method.

Listing 3-54. Disabling the submit button

```
<h1>@Title</h1>

<EditForm OnValidSubmit="@Submit"
          Model="@Customer">
  <DataAnnotationsValidator />

  <InputWatcher @ref="inputWatcher" FieldChanged="@FieldChanged" />

  ...
</EditForm>

@code {
  private InputWatcher inputWatcher;
  private bool isInvalid = true;

  ...
```

```
  private void FieldChanged(string fieldName)
  {
    CustomerChanged.InvokeAsync(Customer);
    isInvalid = !inputWatcher.Validate();
  }
}
```

Run your application, and leave some of the `Customer` properties invalid (that is to say blank). When you press the submit button (a.k.a. `Checkout`), you will get validation errors and the button will disable itself. When you fix the validation errors, the submit button will again be enabled. If you want the button to be disabled right away, change the initial value of `isInvalid` to true.

The Blazor Compilation Model

Every razor (`.razor`) file gets compiled into C# code, and it is very interesting to have a look at this. These files get generated in the `obj` subfolder of your project, and we can look at these generated files. First, with Visual Studio only, select the `PizzaPlace.Client` project in Solution Explorer and click the `Show All Files` button (top right corner of Solution Explorer) as shown in Figure 3-11.

Figure 3-11. *Show All Files*

Now expand the `obj/Debug/netstandard2.1/Razor/Pages` folder. Open `PizzaItem.razor.g.cs`, which you can find in Listing 3-55. I have left out some less important details and edited it for readability. Compare this to Listing 3-45.

Listing 3-55. The PizzaItem.razor.g.cs generated file

```
namespace PizzaPlace.Client.Pages
{
  using System;
  using System.Collections.Generic;
  using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components;
...
public partial class PizzaItem : ComponentBase
{
  protected override void BuildRenderTree(RenderTreeBuilder __builder)
  {
    __builder.OpenElement(0, "div");
    __builder.AddAttribute(1, "class", "row");
    __builder.AddMarkupContent(2, "\r\n  ");
    __builder.OpenElement(3, "div");
    __builder.AddAttribute(4, "class", "col");
    __builder.AddMarkupContent(5, "\r\n    ");
    __builder.AddContent(6, Pizza.Name);
    __builder.AddMarkupContent(7, "\r\n  ");
    __builder.CloseElement();
    __builder.AddMarkupContent(8, "\r\n  ");
    __builder.OpenElement(9, "div");
    __builder.AddAttribute(10, "class", "col");
    __builder.AddMarkupContent(11, "\r\n    ");
    __builder.AddContent(12, Pizza.Price);
    __builder.AddMarkupContent(13, "\r\n  ");
    __builder.CloseElement();
    __builder.AddMarkupContent(14, "\r\n  ");
    __builder.OpenElement(15, "div");
    __builder.AddAttribute(16, "class", "col");
    __builder.AddMarkupContent(17, "\r\n    ");
    __builder.OpenElement(18, "img");
    __builder.AddAttribute(19, "src", SpicinessImage(Pizza.Spiciness));
    __builder.AddAttribute(20, "alt", Pizza.Spiciness);
    __builder.CloseElement();
    __builder.AddMarkupContent(21, "\r\n  ");
    __builder.CloseElement();
    __builder.AddMarkupContent(22, "\r\n  ");
    __builder.OpenElement(23, "div");
```

```
        __builder.AddAttribute(24, "class", "col");
        __builder.AddMarkupContent(25, "\r\n    ");
        __builder.OpenElement(26, "button");
        __builder.AddAttribute(27, "class", ButtonClass);
        __builder.AddAttribute(28, "onclick",
          EventCallback.Factory.Create<MouseEventArgs>(this,
            async () => await Selected.InvokeAsync(Pizza)));
        __builder.AddMarkupContent(29, "\r\n      ");
        __builder.AddContent(30, ButtonTitle);
        __builder.AddMarkupContent(31, "\r\n    ");
        __builder.CloseElement();
        __builder.AddMarkupContent(32, "\r\n  ");
        __builder.CloseElement();
        __builder.AddMarkupContent(33, "\r\n");
        __builder.CloseElement();
    }

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

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

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

    [Parameter]
    public EventCallback<Pizza> Selected { get; set; }

    private string SpicinessImage(Spiciness spiciness)
        => $"images/{spiciness.ToString().ToLower()}.png";

    }
}
```

As you can see here, the bulk of the generated code is the `BuildRenderTree` method. This method created elements, attributes, content, and event handlers. For example, the original `razor` file contains Listing 3-56 which gets generated as Listing 3-57.

Listing 3-56. The original razor

```
<div class="col">
  @Pizza.Name
</div>
```

Listing 3-57. The generated code from razor

```
__builder.OpenElement(3, "div");
__builder.AddAttribute(4, "class", "col");
__builder.AddMarkupContent(5, "\r\n    ");
__builder.AddContent(6, Pizza.Name);
__builder.AddMarkupContent(7, "\r\n  ");
__builder.CloseElement();
__builder.AddMarkupContent(8, "\r\n  ");
```

If you really want, you can directly inherit from `BlazorComponent` and override the `BuildRenderTree` method and generate your custom html directly here. This is only interesting in some very advanced scenarios which we don't cover in this book.

Summary

In this chapter, we covered building Blazor components and component libraries. We also discussed how components can communicate with each other through parameters and data binding. Cascading values are a very nice way of sharing data between components in a hierarchy. We applied this learning by dividing the monolithic `Index` component of the `PizzaPlace` application into smaller components. We also saw that in Blazor you can build templated components, which resemble generic classes. These templated components can be parameterized to render different UIs, which makes them quite reusable! Finally, we had a look at component life cycle hooks (which we will need in further chapters) and how Razor components get compiled into good old C# code.

CHAPTER 4

Services and Dependency Injection

Dependency inversion is one of the basic principles of *good object-oriented design*. The big enabler is *dependency injection*. In this chapter, we will discuss dependency inversion and injection and why it is a fundamental part of Blazor. We will illustrate this by building a Service that encapsulates where the data gets retrieved and stored.

What Is Dependency Inversion?

Currently, our Blazor PizzaPlace application retrieves its data from hard-coded sample data. But in a real-life situation, this data will be stored in a database on the server. Retrieving and storing this data could be done in the component itself, but this is a bad idea. Why? Because technology changes quite often, and different customers for your application might want to use their specific technology, requiring you to update your app for every customer.

Instead, we will put this logic into a *Service object*. A Service object's role is to encapsulate specific business rules and especially how data is communicated between the client and the server. A Service object is also a lot easier to test since we can write unit tests that run on their own, without requiring a user to interact with the application for testing.

But first, let's talk about the dependency inversion principle and how dependency injection allows us to apply this principle.

Understanding Dependency Inversion

Imagine a component that uses a service, and the component creates the service using the `new` operator, as in Listing 4-1.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_4

Listing 4-1. A component using a ProductsService

```
@using  MyFirstBlazor.Client.Services

<div>
  @foreach (var product in productsService.GetAllProducts())
  {
    <div>@product.Name</div>
    <div>@product.Description</div>
    <div>@product.UnitPrice</div>
  }
</div>

@code {
  private ProductsService productsService = new ProductsService();
}
```

This component is now completely dependent on the `ProductsService`! You cannot replace the `ProductsService` without walking over every line of code in your application where the `ProductsService` is used and replace it with another class. This is also known as *tight coupling;* see Figure 4-1.

Figure 4-1. *Traditional layered approach with tight coupling*

Now you want to test the `ProductList` component, and `ProductsService` requires a server on the network to talk to. In this case, you will need to set up a server just to run the test. And if the server is not ready yet (the developer in charge of the server hasn't come around to it), you cannot test your component! Or you are using the `ProductsService` in several places in your location, and you need to replace it with another class. Now you will need to find every use of the `ProductsService` and replace the class. What a maintenance nightmare!

Using the Dependency Inversion Principle

The *dependency inversion principle* states

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

What this means is that the `ProductsList` component (the higher-level module) should not directly depend on the `ProductsService` (the lower-level module). Instead, it should rely on an abstraction. It should rely on an *interface* describing what a `ProductsService` should be able to do, not a class describing how it should work.

The `IProductsService` interface would look like Listing 4-2.

Listing 4-2. The abstraction as described in an interface

```
public interface IProductsService
{
  IEnumerable<Product> GetAllProducts();
}
```

And we change the `ProductsList` component to rely on this abstraction, as in Listing 4-3.

Listing 4-3. The `ProductList` component using the `IProductsService` interface

```
@using MyFirstBlazor.Client.Services

<div>
  @foreach (var product in productsService.GetAllProducts())
  {
    <div>@product.Name</div>
    <div>@product.Description</div>
    <div>@product.UnitPrice</div>
  }
</div>

@code {
  private IProductsService productsService;
}
```

Now the `ProductList` component (the high-level module in the preceding code) only relies on the `IProductsService` interface, an abstraction.

Of course, we now make the `ProductsService` (which is the low-level module) implement the interface as in Listing 4-4.

Listing 4-4. The `ProductsService` implementing the `IProductsService` interface

```
public class ProductsService : IProductsService
{
  public static List<Product> products = new List<Product>
  {
  };

  public IEnumerable<Product> GetAllProducts()
  {
    // Some implementation
  }
}
```

If you want to test the `ProductList` component with dependency inversion in place, you can simply build a hard-coded version of the `ProductsService` and run the test without needing a server, for example, in Listing 4-5. And if you are using the `ProductsService` in different places in your application, all you need to do to replace its implementation is to build another class that implements the `IProductsService` interface and tells dependency injection to use the other class! This is also known as the open/closed principle from SOLID.

Listing 4-5. A hard-coded `ProductsService` used for testing

```
public class HardCodedProductsService : IProductsService
{
  public static List<Product> products = new List<Product>
  {
    new Product {
      Name = "Isabelle's Homemade Marmelade",
      Description = "...",
      UnitPrice = 1.99M
```

```
    }, new Product
    {
      Name = "Liesbeth's Applecake",
      Description = "...",
      UnitPrice = 3.99M
    }
  };

  public IEnumerable<Product> GetAllProducts()
  => products;
}
```

By applying the *dependency inversion principle* (see Figure 4-2), we gained a lot more flexibility.

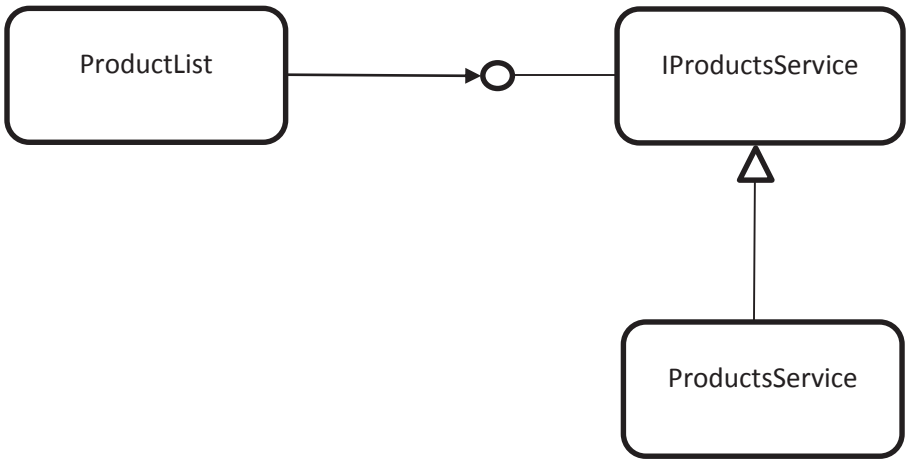

Figure 4-2. *Loosely coupled objects through dependency inversion*

Adding Dependency Injection

If you were to run your application now, you would get a `NullReferenceException`. Why? Because the `ProductsList` component still needs an instance of a class implementing `IProductsService`! We could pass the `ProductsService` in the constructor of the `ProductList` component, for example, in Listing 4-6.

Listing 4-6. Passing the `ProductsService` in the constructor

```
new ProductList(new ProductsService())
```

But if the `ProductsService` also depends on another class, it quickly becomes like Listing 4-7.

Listing 4-7. Creating a deep chain of dependencies manually

```
new ProductList( new ProductsService(new Dependency()))
```

This is of course not a practical way of working! Because of that, we will use *an Inversion-of-Control Container* (I didn't invent this name!).

Applying an Inversion-of-Control Container

An Inversion-of-Control Container (IoCC) is just another object, which specializes in creating objects for you. You simply ask it to create for you an instance of a type and it will take care of creating any dependencies it requires.

It is a little bit like in a movie where a surgeon, in the middle of an operation, needs a scalpel. The surgeon in the movie holds out his (or her) hand and asks for "Scalpel number 5!". The nurse (the Inversion-of-Control Container) who is assisting simply hands the surgeon the scalpel. The surgeon doesn't care where the scalpel comes from or how it was built.

So, how can the IoCC know which dependencies your component needs? There are two ways.

Constructor Dependency Injection

Classes that need a dependency can simply state their dependencies in their constructor. The IoCC will examine the constructor and instantiate the dependencies before calling the constructor. And if these dependencies have their own dependencies, then the IoCC will also build them! For example, if the `ProductsService` has a constructor that takes an argument of type `Dependency`, as in Listing 4-8, then the IoCC will create an instance of type `Dependency` and will then call the `ProductsService`'s constructor with that instance. The `ProductsService` constructor then stores a reference to the dependency in some field, as in Listing 4-8. Should the `ProductsService`'s constructor take multiple arguments, then the IoCC will create an instance for each argument. Constructor injection is normally used for *required* dependencies.

Listing 4-8. The ProductsService's constructor with arguments

```
public class ProductsService {
  private Dependency dep;

  public ProductsService (Dependency dep) {
    this.dep = dep;
  }
}
```

Property Dependency Injection

If the class that the IoCC needs to build has properties that indicate a dependency, then these properties are filled in by the IoCC. The way a property does that depends on the IoCC (in .NET there are a couple of different IoCC frameworks), but in Blazor, you can have the IoCC inject an instance with the `@inject` directive in your razor file, for example, the third line of code in Listing 4-9.

Listing 4-9. Injecting a dependency with the `@inject` directive

```
@using MyFirstBlazor.Client.Services
@using MyFirstBlazor.Shared
@inject IProductsService productsService

<div>
  @foreach (var product in productsService.GetAllProducts())
  {
    <div>@product.Name</div>
    <div>@product.Description</div>
    <div>@product.UnitPrice</div>
  }
</div>

@code {
}
```

If you're using code separation, you can add a property to your class and apply the `[Inject]` attribute as in Listing 4-10.

Listing 4-10. Using the `Inject` attribute for property injection

```
using Microsoft.AspNetCore.Components;
using MyFirstBlazor.Shared;

namespace MyFirstBlazor.Client.Pages
{
  public partial class ProductList
  {
    [Inject]
    public IProductsService productsService { get; set; }
  }
}
```

You can then use this property directly in your razor file, as in Listing 4-11.

Listing 4-11. Using the `ProductsService` property that was dependency injected

```
<div>
  @foreach (var product in productsService.GetAllProducts())
  {
    <div>@product.Name</div>
    <div>@product.Description</div>
    <div>@product.UnitPrice</div>
  }
</div>
```

Configuring Dependency Injection

There is one more thing we need to discuss. When your dependency is a class, then the IoCC can easily know that it needs to create an instance of the class with the class's constructor. But if your dependency is an interface, which it generally needs to be if you are applying the principle of dependency inversion, then which class does it use to create the instance? Without your help, it cannot know.

An IoCC has a mapping between interfaces and classes, and it is your job to configure this mapping. You configure the mapping in your Blazor project's `Program` class. So open `Program.cs`, as in Listing 4-12.

Listing 4-12. The `Program` class

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Microsoft.Extensions.DependencyInjection;
using MyFirstBlazor.Shared;
using System.Threading.Tasks;

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

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

The idea is that you configure the mapping from the interface to the class here, and you use extension methods on `builder.Services`. Which extension method you call from Figure 4-3 depends on the lifetime you want to give the dependency. There are three options for the lifetime of an instance which we will discuss next (ignore `AddOptions` which has to do with configuration).

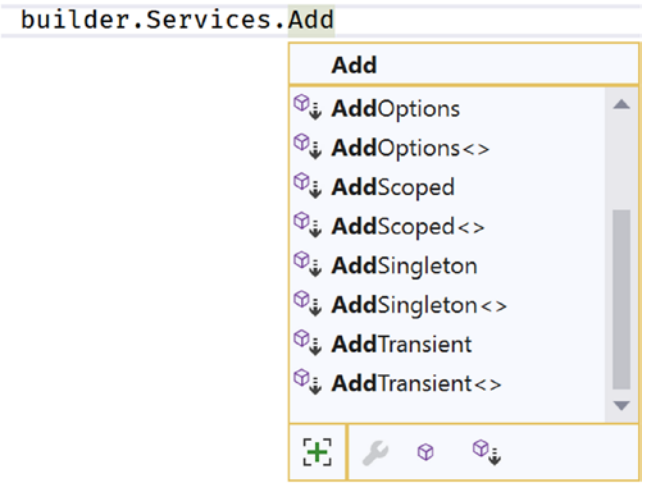

Figure 4-3. *Configuring dependency injection*

Singleton Dependencies

Singleton classes are classes that only have one instance (in the application's scope). These are typically used to manage some global state, for example, you could have a class that keeps track of how many times people have clicked a certain product. Having multiple instances of this class would complicate things because they will have to start communicating with each other to keep track of the clicks. Singleton classes can also be classes that don't have any state, that only have behavior (utility classes such as one that does conversions between imperial and metric units). You configure dependency injection to reuse the same instance all the time with the `AddSingleton` extension method, for example, Listing 4-13.

Listing 4-13. Adding a singleton to dependency injection

```
builder.Services
       .AddSingleton<IProductsService,
                     HardCodedProductsService>();
```

Why not use static methods instead of singletons you say? Static methods and properties are very hard to replace with fake implementations during testing (have you ever tried to test a method that uses a date with `DateTime.Now`, and you want to test it with February 29 of some quantum leap year?). During testing, you can easily replace the real class with a fake class because it implements an interface!

Transient Dependencies

When you configure dependency injection to use a transient class, each time an instance needs to be created by the IoCC, it will create a fresh instance. The IoCC will also dispose of the instance (when your class implements the `IDisposable` interface) when it is no longer needed. Most server-side classes should be transient because each request on a server should not depend on the previous request.

However, in Blazor we are working client-side, and in that case, the UI stays put for the entire interaction. This means that you will have components that only have one created instance and only one instance of the dependency. You might think in that case transient and singleton will do the same thing. But there can be another component that needs the same type of dependency. If you are using a singleton, then both components will share the same instance of the dependency, while transient each gets a unique instance! You should be aware of this.

You configure dependency injection to use transient instances with the `AddTransient` extension method, as in Listing 4-14.

Listing 4-14. Adding a transient class to dependency injection

```
builder.Services
       .AddTransient<IProductsService,
                     HardCodedProductsService>();
```

Scoped Dependencies

When you configure dependency injection to use a scoped dependency, the IoCC will reuse the same instance per scope but uses new instances between different scopes. With ASP.NET Core running on the server, the scope normally corresponds to the current

request. We will discuss the meaning of scope in Blazor a little later in this chapter. Using the server-side scope on the server is especially useful if you use repository objects. Repository objects keep track of all changes made to their objects and then allow you to save (or discard) all changes at the end of the scope/request. If you would use transient instancing for repositories, a single request might lose some changes, which would result in subtle bugs. Let's look at an example. Imagine we have a `DebitService` and another `CreditService`. Both make changes to a bank account and both use a `BankRepository` object as a dependency. A `TransferService` uses a `DebitService` to debit one account, and the `CreditService` credits an account, all using the `BankRepository`.

Look at Listing 4-15.

Listing 4-15. Implementing a `TransferService`

```
public class TransferService {

  private DebitService ds;
  private CreditService cs;
  private BankRepository br;

  public TransferService(
    DebitService ds, CreditService cs, BankRepository br)
  {
    this.ds = ds;
    this.cs = cs;
    this.br = br;
  }

  public Transfer(decimal amount, Account from, Account to) {
    ds.Debit(from, amount);
    cs.Credit(to, amount);
    br.Commit();
  }
}
```

If all three services use the same instance of `BankRepository`, then this should work fine as in Figure 4-4.

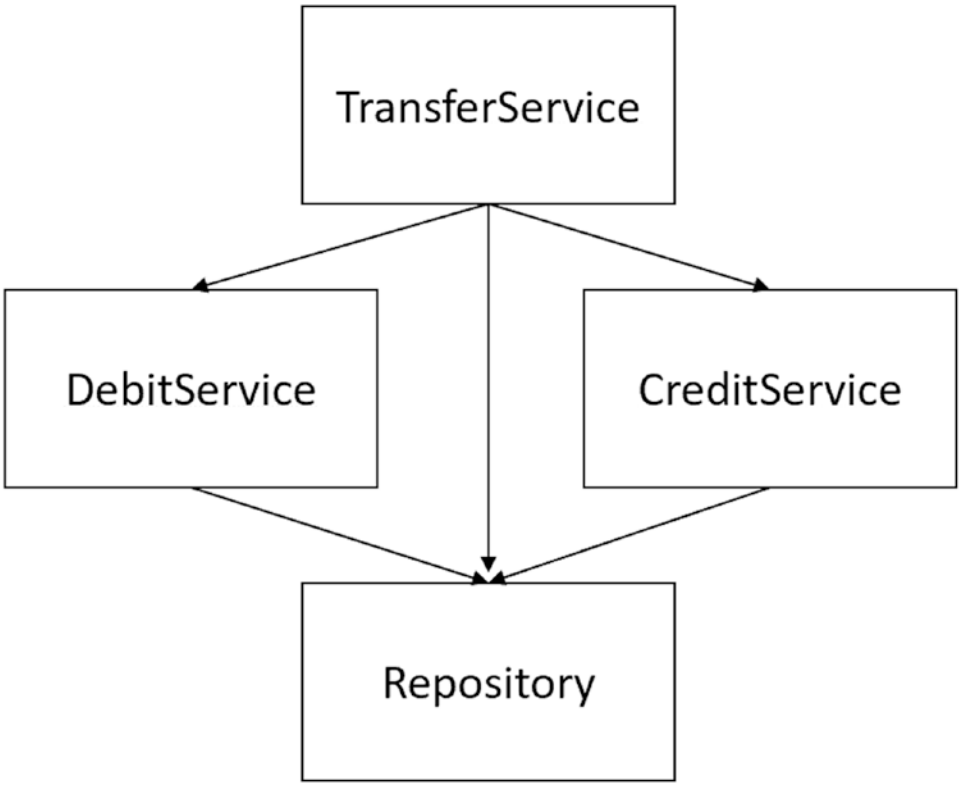

Figure 4-4. *Using a Scoped repository*

But if each receives another instance of `BankRepository`, the `Commit` method will do nothing because no changes were made to the `BankRepository` instance of the `TransferService` as in Figure 4-5.

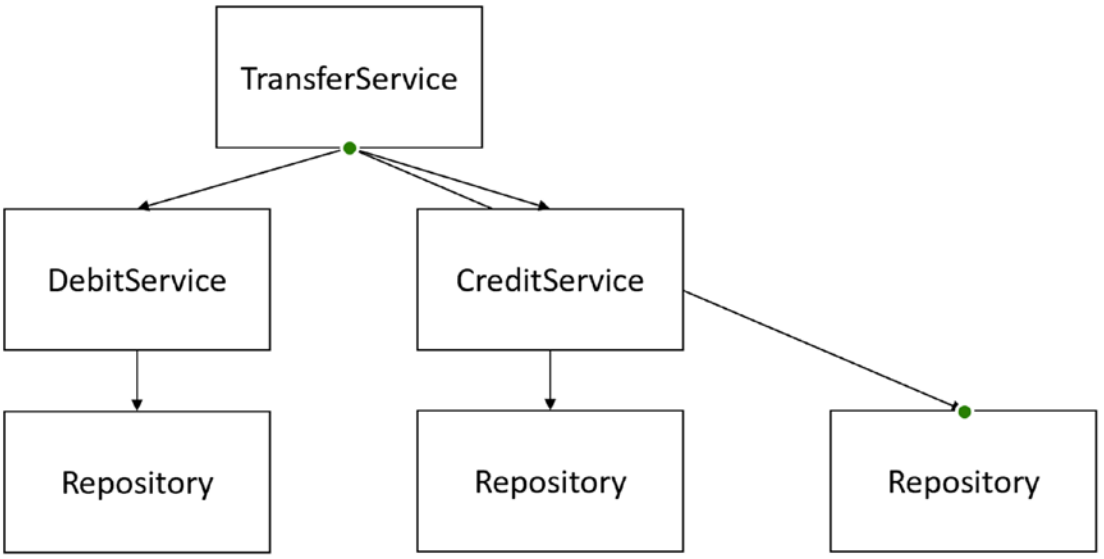

Figure 4-5. *Using a transient repository*

Never use scoped dependencies inside singletons. Your singleton will have a reference to a scoped instance, which will be disposed of after the first request. The singleton does not get disposed, and when it uses the scoped instance, it will get an `ObjectDisposedException` because that is what an instance should do when you call its methods after it is disposed of. Should you forget, the runtime will throw a runtime error, to ensure this problem is found immediately!

Disposing Dependencies

One of the nice extras you get with dependency injection is that it takes care of calling the `Dispose` method of instances that implement `IDisposable`. If the `BankRepository` class of the previous example implements `IDisposable`, cleanup will occur at the end of the lifetime of the instance. In the case of a singleton that would be at the end of the program, for scoped instances at the end of the request and for transient instances, this would normally be when your component is removed from the UI. In general, if your classes implement `IDisposable` correctly, you don't have to take care of anything else.

Understanding Blazor Dependency Lifetime

Let's look at the lifetime of the dependency injected dependencies in Blazor. For this, I have written an experiment which you can find in the included sources for this book.

The source code for this book is available on GitHub via the book's product page, located at www.apress.com/9781484259276.

I started by building three services, each one with a different lifetime. For example, see Listing 4-16. Every time an instance gets created, it gets assigned a GUID. By displaying the instance's GUID, it becomes easy to see which instance gets replaced with a new instance.

Listing 4-16. One of the dependencies used for the experiment

```
public class SingletonService : IDisposable
{
  public Guid Guid { get; set; }

  public SingletonService()
    => Guid = Guid.NewGuid();

  public virtual void Dispose()
    => Console.WriteLine($"{nameof(SingletonService)} disposed.");
}
```

Then I added these three services to the service collection, as in Listing 4-17.

Listing 4-17. Adding the dependencies in dependency injection

```
using Microsoft.Extensions.DependencyInjection;

namespace BlazorLifetime.Shared
{
  public static class DependencyInjection
  {
    public static IServiceCollection AddLifetime(
      this IServiceCollection services)
```

```
    {
      services.AddSingleton<SingletonService>();
      services.AddTransient<TransientService>();
      services.AddScoped<ScopedService>();
      return services;
    }
  }
}
```

And finally, I consume these services in a simple component in Listing 4-18, which I am using in two separate pages because this will allow me to switch from one to the other. This component will display GUIDs for each dependency.

Listing 4-18. The component consuming the dependencies

```
<div>
  <h1>Singleton</h1>
  Guid: @singletonService.Guid
  <h1>Transient</h1>
  Guid: @transientService.Guid
  <h1>Scoped</h1>
  Guid: @scopedService.Guid
</div>

@inject SingletonService  singletonService
@inject TransientService transientService
@inject ScopedService scopedService
```

Client-Side Blazor Experiment

Run the `BlazorLifetime.Server` project, which will start Blazor WebAssembly. We get Figure 4-6 on the first page (your GUIDs will be different). Switching to the other page shows Figure 4-7.

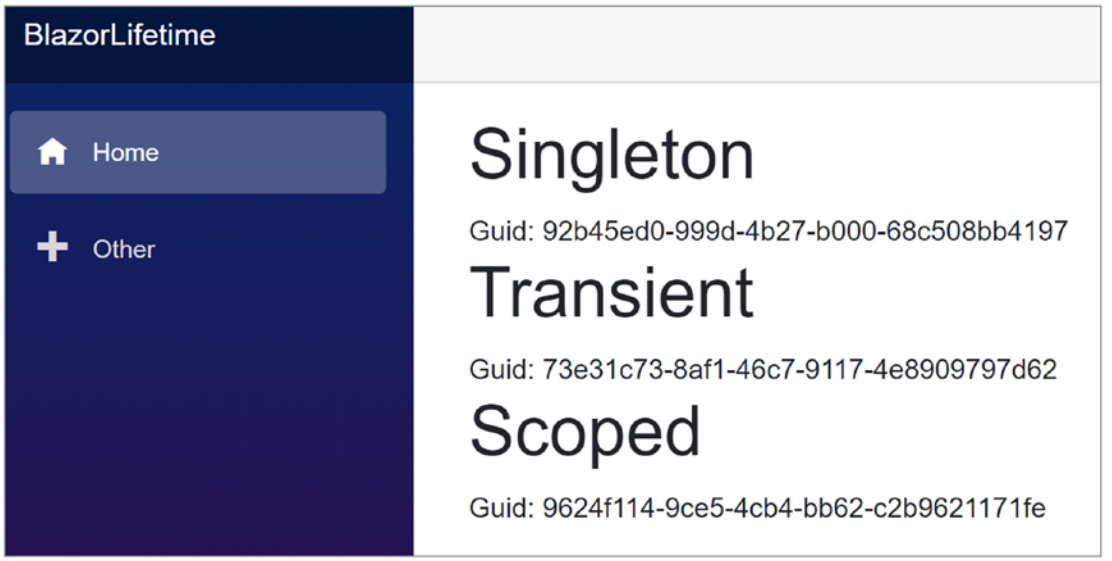

Figure 4-6. *Displaying Client-Side Blazor dependencies*

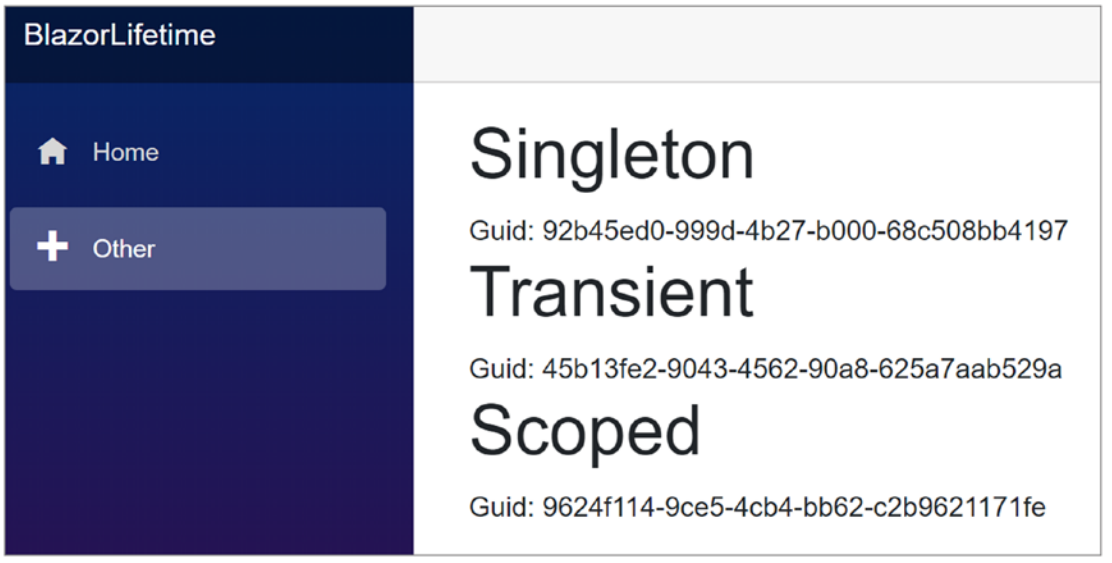

Figure 4-7. *The dependencies from the other page*

As you can see from the GUIDs, the Singleton instance gets reused all the time, which the transient instance gets replaced each time. In Blazor WebAssembly (a.k.a. Client-Side Blazor), the scope is set to the connection. And that is why a scoped instance always returns the same instance.

And what if we open another connection? Opening another tab in the browser with the same URL displays Figure 4-8 for the Home page and Figure 4-9 for the Other page.

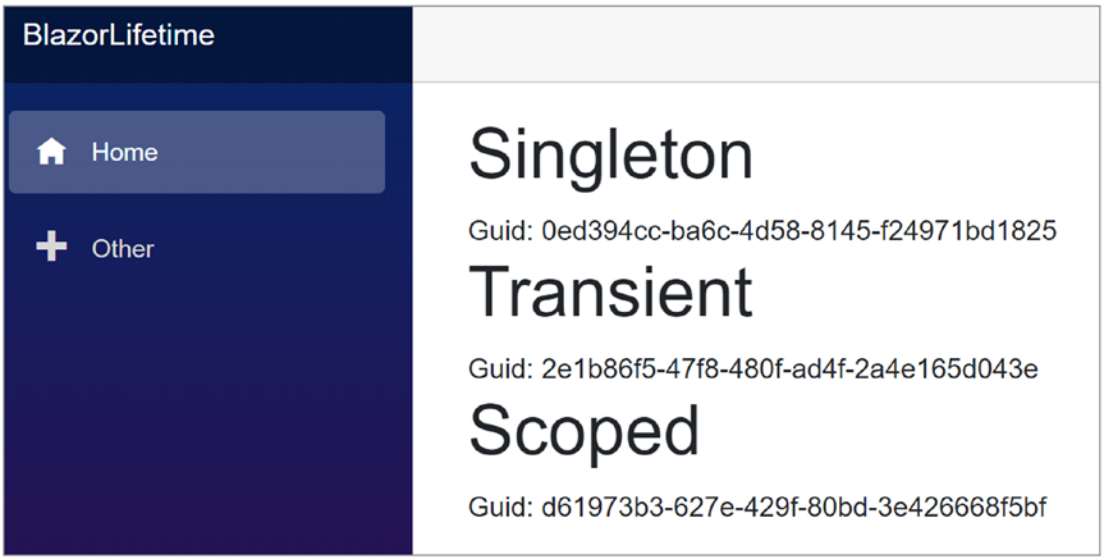

Figure 4-8. *Opening another tab on the Home page*

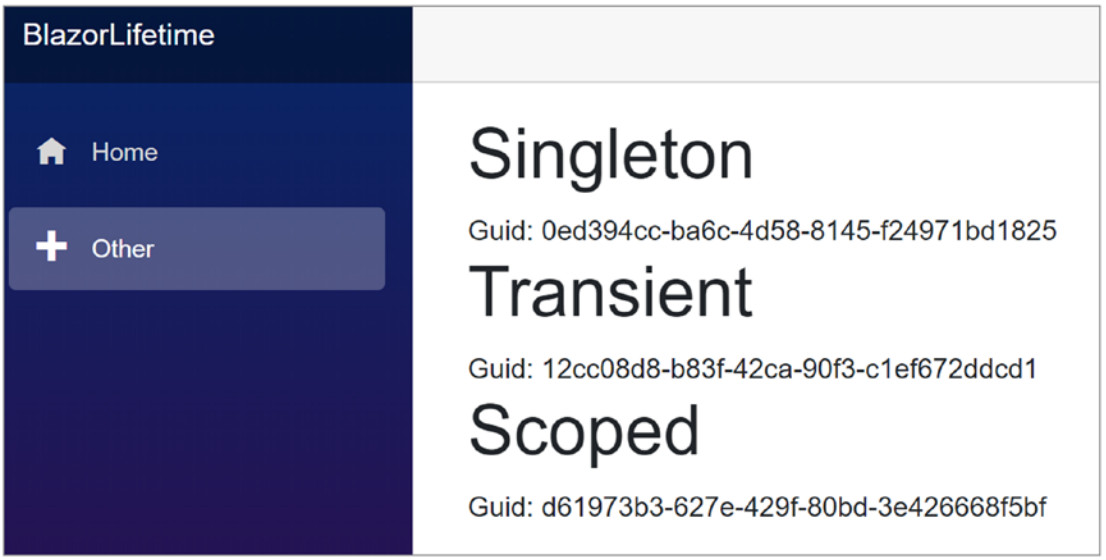

Figure 4-9. *And clicking the Other page*

Since we have a new instance of the Blazor application running in the browser, we get a new instance for the singleton, and because the scope is the connection, we get another instance of the scoped instance. If you expected to see the same instance for the singleton in both tabs, please remember that here each tab holds another runtime of the Blazor application.

Server-Side Blazor Experiment

Now run the BlazorLifetime.ServerSideClient project. Your browser should open on the Home page as in Figure 4-10. Click the Other link to see Figure 4-11 (again with different GUIDs).

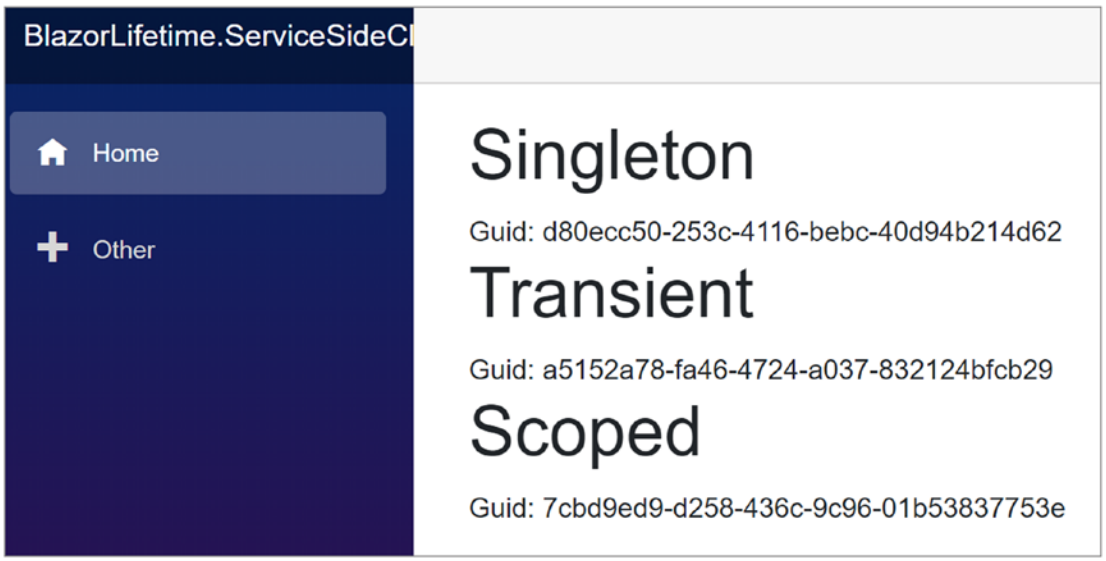

Figure 4-10. *Displaying Server-Side dependencies*

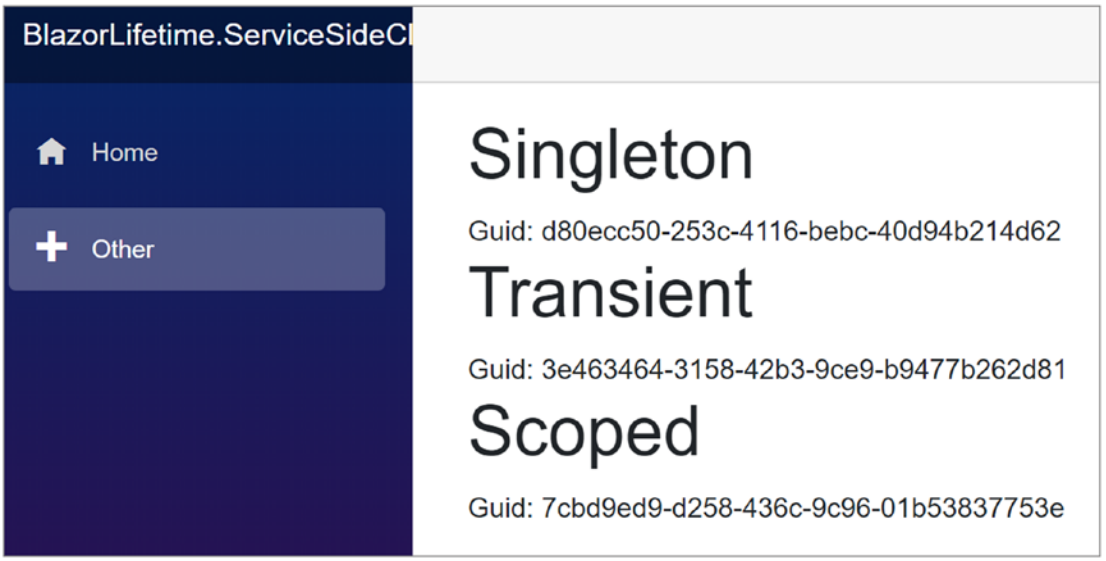

Figure 4-11. *After clicking the Other link*

Open another tab to see Figure 4-12 and click the Other link to see Figure 4-13.

Figure 4-12. *Opening another tab with Server-Side on the Home page*

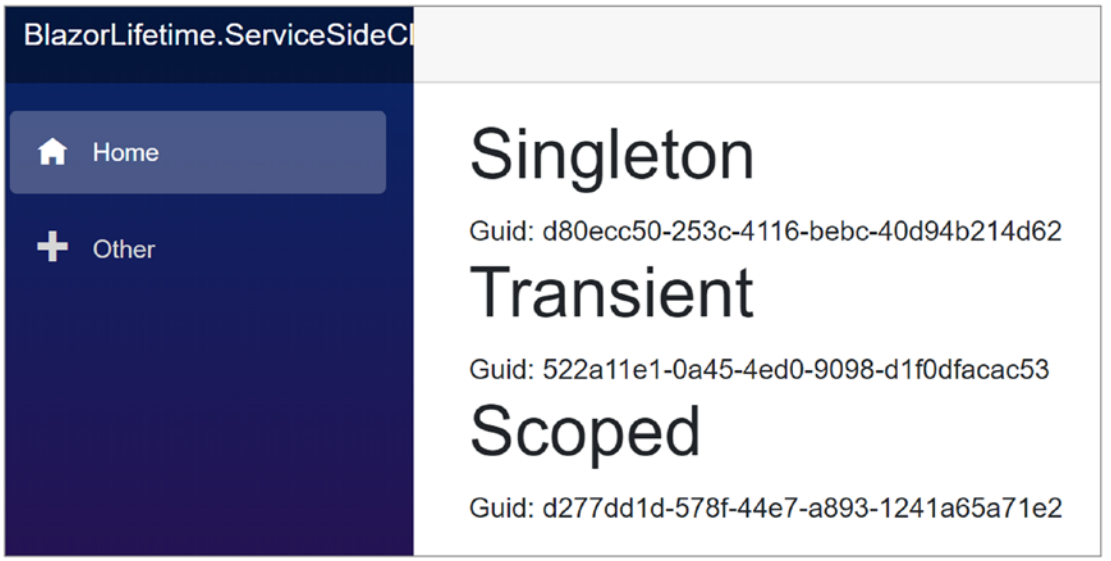

Figure 4-13. *Switching to the Other page*

The Result of the Experiment

Now the experiment is complete; let us draw some conclusions about the lifetime of the injected dependencies. Every time an instance gets created, it gets a new GUID. This makes it easy to see if a new instance gets created or the same instance gets reused.

Transient Lifetime

This one is easy. Transient lifetime means you get a new instance every time.

Singleton Lifetime

With Blazor, we have server-side and client-side. With server-side Blazor, a Singleton gets shared by everyone on that server because it is scoped to the server instance. Everybody gets the same instance.

But with client-side Blazor, the same instance (of Singleton) is scoped to your browser's tab because the Singleton is scoped to the WASM application! Open two tabs in the same browser, and you get two instances of that Singleton. If you need a Singleton with CSB, you need to use a server-side singleton and use REST calls.

Scoped Lifetime

Blazor documentation clearly states that with client-side Blazor a Scoped instance is the same as a Singleton instance because the scope is the application. This is what we see in this experiment. Please use Scoped instances if you need the instance to correspond to the user (actually that user's connection, you get a different connection with each tab).

For service-side Blazor, the Scoped instance is linked to the connection of the user. If the same user opens two tabs in the same browser, each tab receives its own instance of the Scoped instance.

For both CSB and SSB, if you need to have the same instance, no matter which tab the user is using, you cannot rely on dependency injection to do this for you. You will need to do some state handling yourself!

Building Pizza Services

Let's go back to our `PizzaPlace` project and introduce it to some services. I can think of at least two services, one to retrieve the menu and one to place the order when the user clicks the Order button.

Start by reviewing the `Index` component, which is Listing 4-19 with the methods left out for conciseness.

Listing 4-19. The `Index` component

```
@page "/"
<!-- Menu -->
<PizzaList Title="Our selection of pizzas"
           Menu="@State.Menu"
           Selected="@( async (pizza)
             => AddToBasket(pizza))" />
<!-- End menu -->
<!-- Shopping Basket -->
<ShoppingBasket Title="Your current order"
                Basket="@State.Basket"
                GetPizzaFromId="@State.Menu.GetPizza"
                Selected="@( (pos) => RemoveFromBasket(pos))"
                />
```

```
<!-- End shopping basket -->
<!-- Customer entry -->
<CustomerEntry Title="Please enter your details below"
               ButtonTitle="Checkout"
               ButtonClass="btn btn-primary"
               @bind-Customer="@State.Basket.Customer"
               Submit="@PlaceOrder"
               />
<!-- End customer entry -->

<p>@State.ToJson()</p>

@code {

  private State State { get; } = new State()
  {
    Menu = new Menu
    {
      Pizzas = new List<Pizza>
{
        new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
        new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
        new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
      }
    }
  };

  ...

}
```

Pay special attention to the `State` property. We will initialize the `State.Menu` property from the MenuService service, and we will use dependency injection to pass the service.

Adding the MenuService and IMenuService Abstraction

If you are using Visual Studio, right-click the `PizzaPlace.Shared` project and select `Add ➤ New Item`. If you are using Code, right-click the `PizzaPlace.Shared` project and select `Add File`. Add a new interface class `IMenuService` and complete it as in Listing 4-20.

Listing 4-20. The `IMenuService` interface

```
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
  public interface IMenuService
  {
    Task<Menu> GetMenu();
  }
}
```

This interface allows us to retrieve a menu. Do note that the `GetMenu` method returns a `Task<Menu>`; that is because we expect the service to retrieve our menu from a server (we will build this in the following chapters) and we want the method to support an asynchronous call. Let's elaborate on this. Have a look at the `OnInitializedAsync` method from Listing 4-23. This is an asynchronous method using the `async` keyword in its declaration. Inside the `OnInitializedAsync` method, we call the `GetMenu` method using the `await` keyword which requires `GetMenu` to return a `Task<Menu>`. Thanks to the `async/await` syntax, this is easy to do, but it does require that you return a `Task`.

Now add the `HardCodedMenuService` class to the `PizzaPlace.Shared` project, as in Listing 4-21.

Listing 4-21. The `HardCodedMenuService` class

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
  public class HardCodedMenuService : IMenuService
  {
    public Task<Menu> GetMenu()
```

```
      => Task.FromResult(new Menu
      {
        Pizzas = new List<Pizza>
        {
          new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
          new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
          new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
        }
      });
  }
}
```

Now we are ready to use the `IMenuService` in our `Index` component. Start by adding the dependency on `IMenuService` using the `@inject` syntax, as in Listing 4-22.

Listing 4-22. Stating that the `Index` component depends on an `IMenuService`

```
...

@inject IMenuService menuService

@code {
```

We initialize the `State.Menu` property in the `OnInitializedAsync` life cycle method, as in Listing 4-23.

Listing 4-23. Initializing the `Index` component's `Menu`

```
@inject IMenuService menuService

@code {

    private State State { get; } = new State();

    protected override async Task OnInitializedAsync()
    {
      State.Menu = await menuService.GetMenu();
    }

    ...
}
```

> Never call asynchronous services in your Blazor component's constructor; always use `OnInitializedAsync` or `OnParametersSetAsync`.

Now we are ready to configure dependency injection, so open `Program.cs` from the client project. We'll use a transient object as stated in Listing 4-24.

Listing 4-24. Configuring dependency injection for the `MenuService`

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Shared;
using System.Threading.Tasks;

namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder
        .CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService,
                                    HardCodedMenuService>();
      await builder.Build().RunAsync();
    }
  }
}
```

Run your Blazor project. Everything should still work! In the next chapters, we will replace this with a service to retrieve everything from a database.

Ordering Pizzas with a Service

When the user makes a selection of pizzas and fulfills the customer information, we want to send the order to the server, so they can warm up the oven and send some nice

pizzas to the customer's address. Start by adding an `IOrderService` interface to the `PizzaPlace.Shared` project as in Listing 4-25.

Listing 4-25. The `IOrderService` abstraction as a C# interface

```
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
  public interface IOrderService
  {
    Task PlaceOrder(Basket basket);
  }
}
```

To place an order, we just send the basket to the server. In the next chapter, we will build the actual server-side code to place an order; for now, we will use a fake implementation that simply writes the order to the browser's console. Add a class called `ConsoleOrderService` to the `PizzaPlace.Shared` project as in Listing 4-26.

Listing 4-26. The `ConsoleOrderService`

```
using System;
using System.Threading.Tasks;

namespace PizzaPlace.Shared
{
  public class ConsoleOrderService : IOrderService
  {
    public async Task PlaceOrder(Basket basket)
    {
      Console.WriteLine($"Placing order for {basket.Customer.Name}");
      await Task.CompletedTask;
    }
  }
}
```

The `PlaceOrder` method simply writes the basket to the console. However, this method implements the asynchronous pattern from .NET, so we need to return a `Task` instance. This is easily done using the `Task.CompletedTask` property. `Task.CompletedTask` is simply a "no nothing" task and is very handy if you need to implement a method that needs to return a Task instance.

Inject the `IOrderService` into the `Index` component as in Listing 4-27.

Listing 4-27. Injecting the `IOrderService`

```
@inject IMenuService menuService
@inject IOrderService orderService

@code {
```

And use the order service when the user clicks the Order button by replacing the implementation of the `PlaceOrder` method in the `Index` component. Since the `orderService` is asynchronous, we need to invoke it asynchronously, as in Listing 4-28.

Listing 4-28. The asynchronous `PlaceOrder` method

```
private async Task PlaceOrder()
{
  await orderService.PlaceOrder(State.Basket);
}
```

As the final step, configure dependency injection. Again, we will make the `orderService` transient as in Listing 4-29.

Listing 4-29. Configuring dependency injection for the `orderService`

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Shared;
using System.Threading.Tasks;

namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
```

```
    {
      var builder = WebAssemblyHostBuilder
        .CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService,
                                    HardCodedMenuService>();
      builder.Services.AddTransient<IOrderService,
                                    ConsoleOrderService>();
      await builder.Build().RunAsync();
    }
  }
}
```

Think about this. How hard will it be to replace the implementation of one of the services? There is only one place that says which class we will be using, and that is in Listing 4-29. In the next chapter (Chapter 5), we will build the server-side code needed to store the menu and the orders, and in Chapter 6, we will replace these services with the real deal!

Build and run your project again, open your browser's debugger, and open the Console tab. Order some pizzas and click the Order button. You should see some feedback as in Figure 4-14.

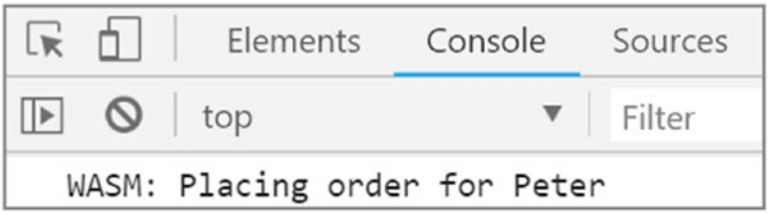

Figure 4-14. *The brower's console showing an order was placed*

Summary

In this chapter, we discussed dependency inversion, which is the best practice for building easily maintainable and testable object-oriented applications. We also saw that dependency injection makes it very easy to create objects with dependencies, especially

objects that use dependency inversion. When you configure dependency injection, you need to be careful with the lifetime of your instances, so let's repeat that:

- *Transient* objects are always different; a new instance is provided to every component and every service.
- *Scoped* objects are the same for a user's connection, but different across different users and connections.
- *Singleton* objects are the same for every object and every request.

CHAPTER 5

Data Storage and Microservices

In general, client-side browser applications need to store some of their data. In some cases, such as games, the application can store its data in the browser itself, using browser local storage. But in most cases, storage will happen on the server, which has access to database engines such as SQL Server. In this chapter, you will learn the basics of storing data using Entity Framework Core and exposing that data using REST and microservices built on top of ASP.NET Core.

What Is REST?

Storing data on the Web is ubiquitous. But how can applications communicate with one another? *Representational State Transfer* (REST) is a protocol built on top of the HTTP protocol for invoking functionality on servers, such as retrieving and storing data from/in a database.

Understanding HTTP

Before talking about REST, you should have a good understanding of the *Hypertext Transfer Protocol*, better known as HTTP. HTTP was created by *Tim Berners-Lee* at CERN in 1989. CERN is a center for elementary physics research, and what do researchers do when they have completed their research? They publish papers with their research findings. Before the Internet, publishing a paper was done literally on paper (hence the name), and it took a lot of time between writing the paper and getting it published in a research magazine. Instead, Tim Berners-Lee devised a way to put papers on a server and allow users to read these papers using a browser.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_5

Also, scientific papers contain a lot of references, and when you want to read a paper like this, it helps to be able to access the referenced papers. The Internet facilitates reading papers through the use of *Hypertext Markup Language* (HTML). Hypertext is an electronic document format that can contain links to other documents. You simply click the link to read the other paper and you can go back to the first paper simply by clicking the back button in your browser.

Universal Resource Identifiers and Methods

Browsers are applications that know how to talk HTTP, and the first thing you do after opening a browser is you type in a *Universal Resource Identifier* (URI). A URI allows a browser to talk to a server, but more is needed. As the name suggests, a URI identifies a resource universally, but you also need to use a *method* to instruct the server to do something with the URI. The most common method is GET. As Figure 5-1 shows, when you type in a URI in the browser, it will do a GET on the server.

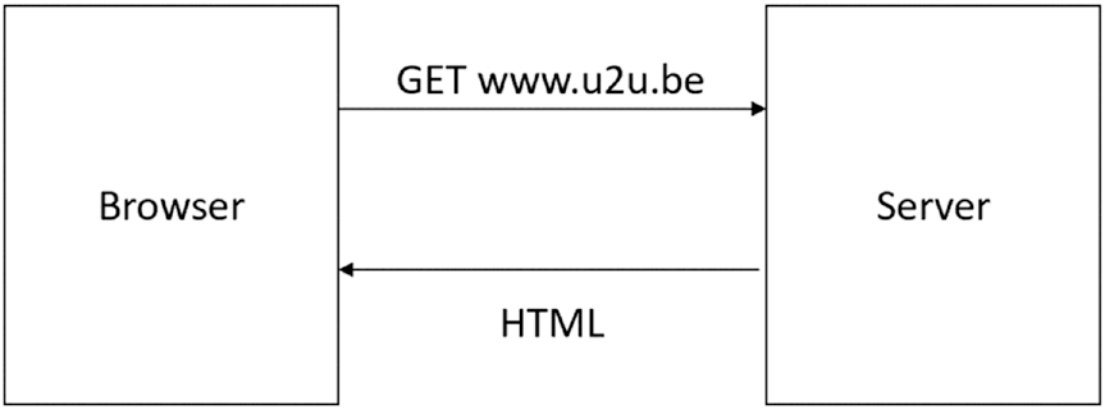

Figure 5-1. *The browser uses the GET method to retrieve a document*

Each time you click a hyperlink in the HTML document, the browser repeats this process with another URI.

But there are other methods. If you want to publish a new paper, you can use the POST method to send the paper to the server, supplying it with a URI. In this case, the server will store the paper at the requested URI. If you want to make a change to your paper, for example, to correct a spelling mistake, you can use the PUT method. Now the server will overwrite the URI contents. And finally, you can delete the paper using the DELETE method and its URI.

HTTP Status Codes

What happens when you ask a server about something it doesn't have? What should the server return? Servers not only return HTML, but they also return a status code about the result. When the server can process the request successfully, it will in general return status code 200 (other successful status codes exist). When the server can't find the resource, it will return a status code 404. Status code 404 simply means not found. The client will receive this status code and can react appropriately. When the browser receives a status code 200, it displays the HTML; when it receives a 404, it displays a not found screen, and so on.

A full list of all HTTP status codes can be found at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

Invoking Server Functionality Using REST

Think about these methods we just talked about. With POST, you can CREATE something on a server; with GET, you can READ it back; with PUT, you can UPDATE something on the server, and with DELETE, you can DELETE things on the server. They are also known as CRUD operations (CREATE-READ-UPDATE-DELETE). *Roy Fielding*, the inventor of REST, realized that using the HTTP protocol you can also use HTTP to work with data stored in a database. For example, if you use the GET method with a URI `http://someserver/categories`, the server can execute some code to retrieve data from the categories relational table and return it. Of course, the server would use a format more appropriate for transferring data, such as XML or JSON. Because there are many different formats for data, the server also needs a way to convey which format it is sending. (At the beginning of the Web, only HTML was used as the format.) This is done through HTTP headers.

HTTP Headers

HTTP headers are instructions exchanged between the client and the server. Headers are key-value pairs, where the client and server agree on the key. Many standard HTTP headers exist. For example, a server can use the *Content-Type header* to tell the client to expect a specific format. Another header is the *Accept header*, which is sent by the client

to the server to politely ask the server to send the content in that format; this is also known as *content negotiation*. Currently, the most popular format is *JavaScript Object Notation* (JSON). And this is the exchange format you will use with Blazor.

JavaScript Object Notation

JSON is a compact format for transferring data. Look at the example in Listing 5-1.

Listing 5-1. An example of JSON

```
{ "book" : {
  "title" : "Blazor Revealed",
  "chapters" : [ "Your first Blazor project", "Data Binding"]
  }
}
```

This JSON format describes a book, an object in memory. Objects are denoted using curly braces. Inside the book are two properties; each property uses a `key : value` notation. The book's `title` is `"Blazor Revealed"`. Note that the property name is also transferred as a string. And finally, the "`chapters`" property is an array of strings, where you use square brackets to indicate an array.

The JSON format is used for transferring data between two machines, but today is also heavily used for configuring tools such as ASP.NET Core. JSON today is way more popular on the Web than XML, probably because of its simplicity.

Some Examples of REST Calls

You need a list of pizzas from a server, and the server exposes the pizzas at URI `http://someserver/pizza`. To get a list of pizzas, you use the GET method, and you use the `Accept` header with value `application/json` to request the JSON format. Look at Figure 5-2 for this example.

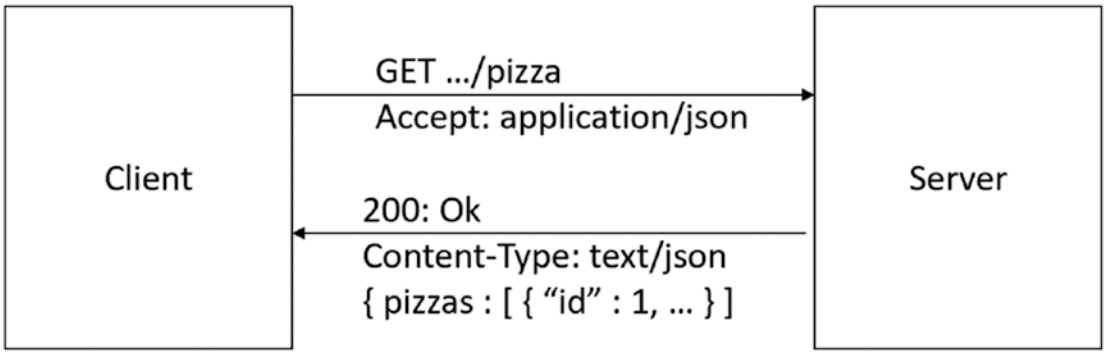

Figure 5-2. *Using REST to retrieve a list of pizzas*

Maybe your client wants to display the details of a pizza with id number 5. In this case, it can append the id to the URI and perform a GET. Should the server not have any pizza with that id, it can return a status code 404, as illustrated in Figure 5-3.

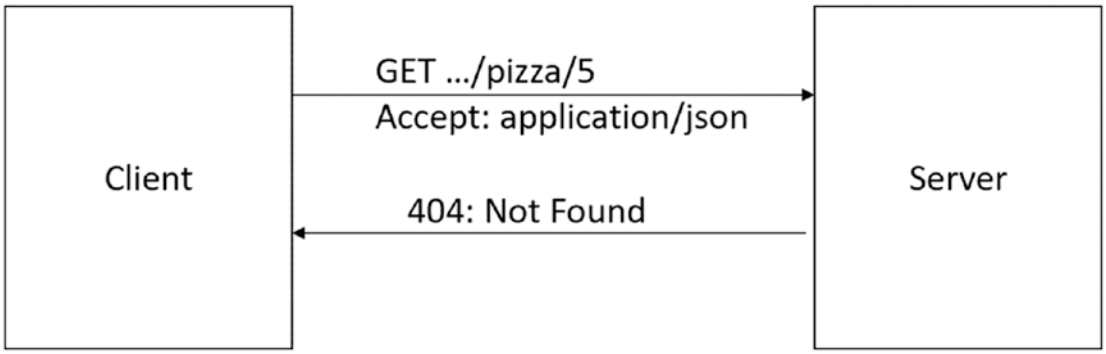

Figure 5-3. *Using REST to retrieve a specific pizza through its unique id*

As the last example, let's send some data from the client to the server. Imagine that the customer has filled in all the details for the order and clicks the Order button. You then send the order as JSON to the server using the POST method (remember POST means insert). The server can then process the order in any way it likes; for instance, it can insert the order into its database and return a 201: Created status code, as in Figure 5-4. REST recommends returning a status code 201 with the `Location` header set to the URI for the newly created resource.

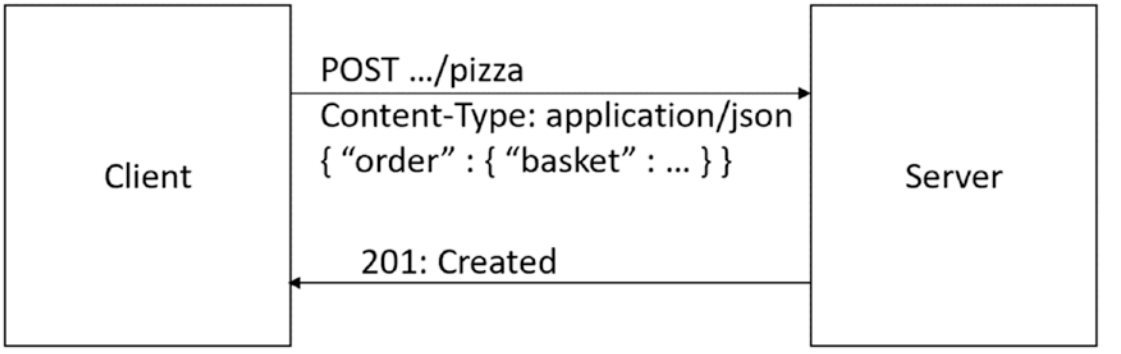

Figure 5-4. *POSTing an order to the server*

Building a Simple Microservice Using ASP.NET Core

So, how do you build a REST service? Your Blazor project uses ASP.NET Core for hosting the Blazor client, and adding a service to your project is easy. But first, let's do a little intro to microservices.

Services and Single Responsibility

A service is a piece of software that listens for requests; when it receives a request, the service handles the request and returns with a response. In real life, you also encounter services and they are very similar. Consider a bank. You step into a bank and you give the teller your account number and some ID and request $100. The teller will check your account; if you have enough money in your account, the teller will deduct the money and give you the cash. Should your account be too low, the teller will refuse. In both cases, you got a response.

Services should also adhere to the principle of single responsibility. They should do one thing very well, and that's it. For example, the pizza service will allow clients to retrieve pizzas, add, update, and delete pizzas. That's it. A single responsibility, in this case, PIZZAS.

You can have other services too, each with their own responsibility. Services that take care of one thing are known as *microservices*.

The Pizza Service

Open the PizzaPlace solution you worked on in previous chapters. In this chapter, you will focus on the `PizzaPlace.Server` project, shown in Figure 5-5.

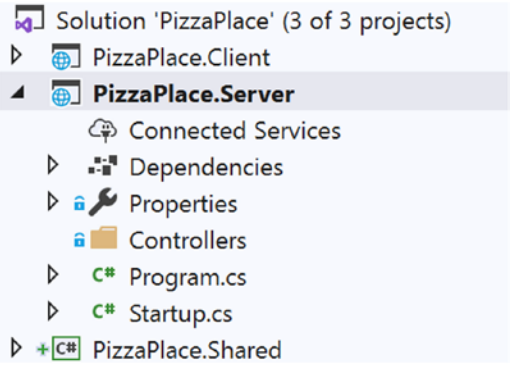

Figure 5-5. *The PizzaPlace.Server project*

The only role this project currently has is to host your Blazor client application, but now you will enhance this role by adding some microservices.

Open `Startup.cs` and look at the `Configure` method, as in Listing 5-2.

Listing 5-2. The `Startup` class's `Configure` method

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseWebAssemblyDebugging();
  }

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

  app.UseRouting();
```

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

The last line with the `endpoints.MapFallbackToFile("index.html")` method takes care of your Blazor client project. But right before it, you see the `endpoints.MapControllers()` method that is used for hosting your services.

How the `MapControllers` method works is not the topic of this book, but I will cover what you need to know. If you want to learn more about ASP.NET Core, there are many good books about this topic, such as *Pro ASP.NET Core MVC* by Adam Freeman.

Next in line is the `Controllers` folder. In Figure 5-5, this folder is empty, and the idea is that you put your service classes here. In ASP.NET, service classes are known as controllers. If you are using Visual Studio, right-click this folder and select Add ➤ Controller. Select API Controller - Empty from Figure 5-6 and click Add.

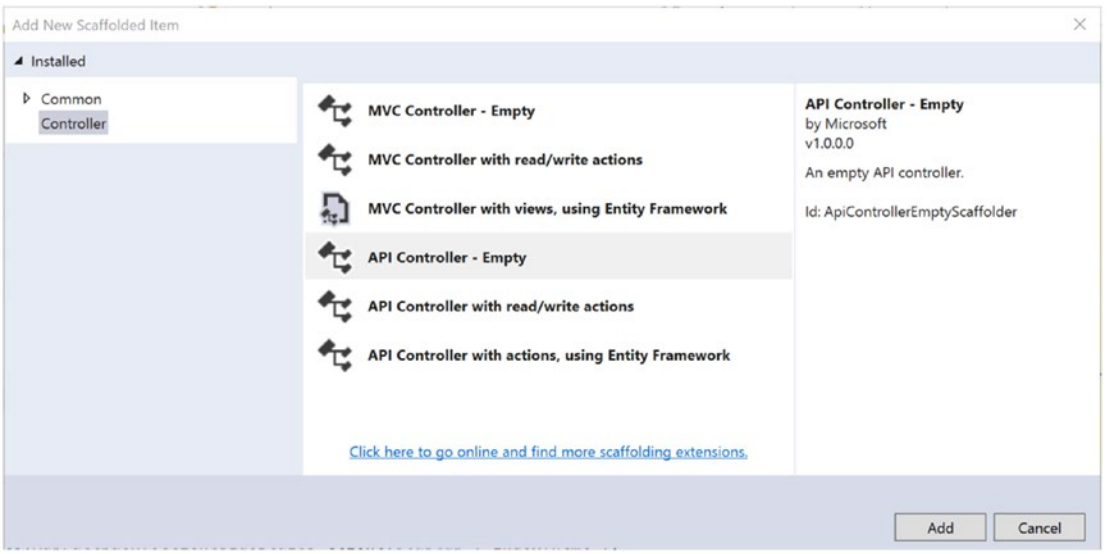

Figure 5-6. *Adding a new controller*

Type PizzasController as in Figure 5-7 and click Add again.

Figure 5-7. *Naming the controller*

If you are using Code, simply right-click the `Controllers` folder and select Add File. Name it `PizzasController.cs`.

This will add a new class called `PizzasController`, inheriting from `ControllerBase`, which you can see in Listing 5-3. Please note that the `Route` attribute indicates that the URI you should use is `api/pizzas`. The `[controller]` part of the route is a placeholder for the name of the controller, but without the Controller part.

Listing 5-3. The empty `PizzasController`

```
using Microsoft.AspNetCore.Mvc;

namespace PizzaPlace.Server.Controllers
{
  [Route("api/[controller]")]
  [ApiController]
  public class PizzasController : ControllerBase
  {
  }
}
```

Let's add a method to retrieve a list of pizzas. For the moment, you will hard-code the list, but in the next section, you will retrieve it from a database. Modify the `PizzasController` as shown in Listing 5-4.

Listing 5-4. Adding a method to the `PizzaController` to retrieve a list of pizzas

```
using Microsoft.AspNetCore.Mvc;
using PizzaPlace.Shared;
using System.Collections.Generic;
using System.Linq;
```

```
namespace PizzaPlace.Server.Controllers
{
  [ApiController]
  public class PizzasController : ControllerBase
  {
    private static readonly List<Pizza> pizzas = new List<Pizza>
        {
          new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
          new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
          new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
        };

    [HttpGet("pizzas")]
    public IQueryable<Pizza> GetPizzas()
      => pizzas.AsQueryable();
  }
}
```

Let's walk through this implementation. First, you declare a hard-coded static list of pizzas. Next is the `GetPizzas` method, which has an attribute of `HttpGet("pizzas")`. This attribute says that when you perform a `GET` on the server with the `pizzas` URI, the server should call the `GetPizzas` method.

The `GetPizzas` method returns an `IQueryable<Pizza>`, and ASP.NET Core will send this result back to the client with the requested format. The `IQueryable<Pizza>` interface is used in .NET to represent data that can be queried, such as database data, and is returned by LINQ queries.

Note that the `GetPizzas` method contains nothing about HOW the data will be transferred to the client. This is all taken care of for you by ASP.NET Core! By default, your implementation in ASP.NET Core will use JSON, which is what you want. ASP.NET Core allows you to pick other formats, including your custom format. This is not the scope of this book.

Time to see if it works. First, ensure that the `PizzaPlace.Server` project is the startup project. Right-click the `PizzaPlace.Server` project and select `Set as Startup Project` from the drop-down menu. The `PizzaPlace.Server` project should be shown as bold, as in Figure 5-5.

Now run your project and wait for the browser to open because you will perform a GET; you can use the browser, but for other methods, you will later use a nice tool called Postman.

Change the URI in the browser to `http://localhost:xxxx/pizzas` where `xxxx` is the original port number in your browser (the port number gets selected by Visual Studio and will be different than mine). You should see the result shown in Figure 5-8.

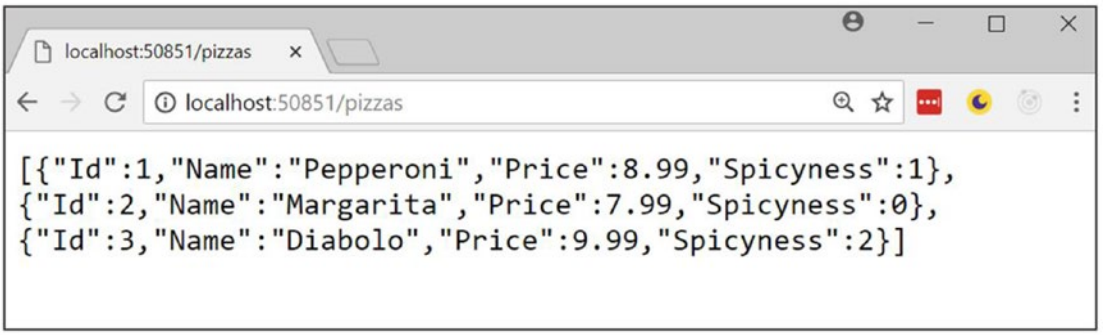

Figure 5-8. *The results of getting a list of pizzas from the pizza service*

A JSON-encoded list of pizzas! It works!

Now you are ready to retrieve the data from a real database using Entity Framework Core.

What Is Entity Framework Core?

Entity Framework Core is the framework Microsoft recommends for working with databases. It allows you to write classes as normal C# classes and then store and retrieve .NET objects from a database without having to be an SQL expert. It will take care of this for you. This is also known as *persistence ignorance*, where your code does not need to know how and where data gets stored!

Using the Code First Approach

But of course, you need to explain to Entity Framework Core what kind of data you want to store. Entity Framework Core uses a technique called *code first*, where you write code to describe the data and how it should be stored in the database. Then, you can use this to generate the database, the tables, and the constraints. If you want to make changes to the database, you can update the database schema with *code first migrations*.

If you already have a database, you can also generate the code from the database, also known as EF Database First, but this is not the target of this book.

In the code first approach, you describe the classes (also known as entities) that will map to database tables. You already have the `Pizza` class (which you can find in the `PizzaPlace.Shared` project) to describe the Pizza table in the database. But you need to do more.

In this part, you will be using SQL Server. If you installed Visual Studio on your Windows machine, SQL Server was installed too. If you don't have SQL Server on your machine, you can install a free version of SQL Server, or use a SQL Server instance in the cloud, for example, SQL Server on Azure (`https://azure.microsoft.com/en-us/get-started/`). You can even install SQL Server on OSX! There are some nice articles on the Web that explain how.

You need to add Entity Framework Core to the `PizzaPlace.Server` project. If you are using Visual Studio, right-click the server project and select Manage NuGet Packages..., as shown in Figure 5-9.

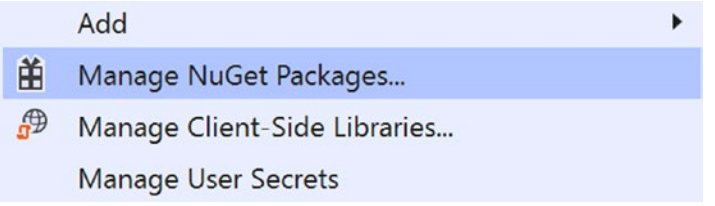

Figure 5-9. *Adding NuGet packages to your project*

The NuGet window will open in Visual Studio. NuGet is a very practical way for installing dependencies such as Entity Framework Core to your project. It will not only install the Microsoft.EntityFrameworkCore.SqlServer library but also all its dependencies.

Select the Browse tab and type Microsoft.EntityFrameworkCore.SqlServer in the search box. You should see this library as the top search result. Select it, then select the Latest stable 3.1.1 version from the Version drop-down, and click the Install button, as shown in Figure 5-10.

By the time you read this book, Microsoft might have deployed a more recent version, so although Figure 5-10 shows version 3.1.1, you should select the latest stable version.

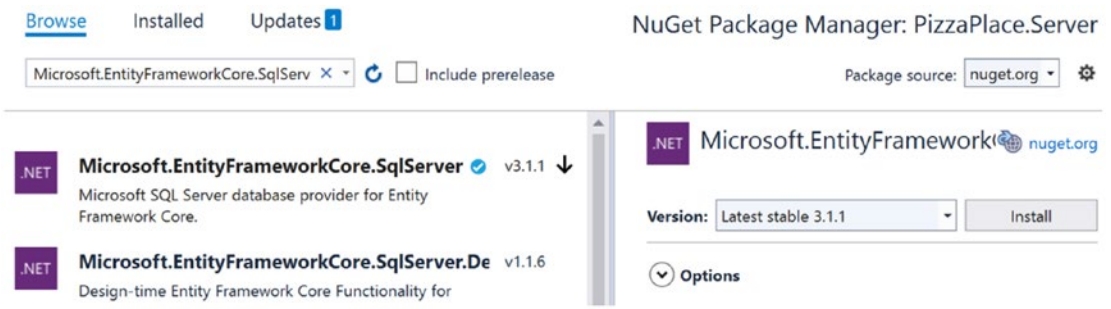

Figure 5-10. *Adding Entity Framework Core using NuGet*

With Code, you open the command prompt and type in the following command after changing the current directory of your `PizzaPlace.Server` project:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

With this dependency installed, you are ready to make some code changes. Entity Framework Core requires that entity classes have a default constructor and that properties are read-write. Update the `Pizza` class by adding a default constructor and adding setters for the properties, as shown in Listing 5-5.

Listing 5-5. Modifying the `Pizza` class for Entity Framework Core

```
using System;

namespace PizzaPlace.Shared
{
  public enum Spiciness
  {
    None,
    Spicy,
    Hot
  }
```

```
  public class Pizza
  {
    public Pizza() { }

    public Pizza(int id, string name, decimal price,
                 Spiciness spiciness)
    {
      Id = id;
      Name = name
        ?? throw new ArgumentNullException(nameof(name),
           "A pizza needs a name!");
      Price = price;
      Spiciness = spiciness;
    }

    public int Id { get; set; }
    public string Name { get; set;  }
    public decimal Price { get; set; }
    public Spiciness Spiciness { get; set; }
  }
}
```

Add a new class called `PizzaPlaceDbContext` to the `PizzaPlace.Server` project, as shown in Listing 5-6. This class represents the database, and you do need to give a couple of hints about how you want your data to be stored in SQL Server (or some other database engine; this uses the same code).

Listing 5-6. The `PizzaPlaceDbContext` class

```
using Microsoft.EntityFrameworkCore;
using PizzaPlace.Shared;

namespace PizzaPlace.Server
{
  public class PizzaPlaceDbContext : DbContext
  {
    public PizzaPlaceDbContext(DbContextOptions<PizzaPlaceDbContext> options)
      : base(options) { }
```

```
    public DbSet<Pizza> Pizzas { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);

      var pizzaEntity = modelBuilder.Entity<Pizza>();

      pizzaEntity.HasKey(pizza => pizza.Id);
      pizzaEntity.Property(pizza => pizza.Price)
        .HasColumnType("money");
    }
  }
}
```

First, you need to create a constructor for the `PizzaPlaceDbContext` class taking an `DbContextOptions<PizzaPlaceDbContext>` argument. This is used to pass the connection to the server, which you will do later in this section.

Next, you add a table to the database to represent your pizzas using a public property of type `DbSet<Pizza>`. `DbSet<T>` is the collection class used by Entity Framework Core, but you can think of it as a `List<T>`. Entity Framework Core will use the `DbSet<T>` to map this collection to a table, in this case, the `Pizzas` table.

Finally, you override the `OnModelCreating` method, which takes a `modelBuilder` argument. In the `OnModelCreating` method, you can describe how each `DbSet<T>` should be mapped to the database; for example, you can tell it which table to use, how each column should be called, which type to use, and so on. This `modelBuilder` has a bunch of methods that allow you to describe how the classes should be mapped to your database. In this case, you tell the `modelBuilder` that the `Pizza` table should have a primary key, the `Id` property of the `Pizza` class. You also need to tell how the `Pizza.Price` property should be mapped to SQL Server. You will use the SQL Server `MONEY` type for that. For the moment, this is enough for your current implementation.

Preparing Your Project for Code First Migrations

Now you are ready to tell the `PizzaPlaze.Server` project to use SQL Server as the database. You do this with dependency injection. In ASP.NET Core, you configure dependency injection in the `Startup` class's `ConfigureServices` method. Let's have a look at this method which is shown in Listing 5-7.

Listing 5-7. The `Startup.ConfigureServices` method

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddRazorPages();
}
```

Remember `IServiceCollection` from the chapter on dependency injection? Here dependencies for ASP.NET Core are added, such as dependencies for `Mvc` and `Razor pages`, which are required for your service.

Start by adding a constructor to the `Startup` class as in Listing 5-8.

Listing 5-8. The `Startup` class's constructor

```
...
using Microsoft.Extensions.Configuration;

namespace PizzaPlace.Server
{
  public class Startup
  {

    public Startup(IConfiguration configuration)
      => this.Configuration = configuration;

    public IConfiguration Configuration { get; }
```

You need this constructor to have access to the project's configuration file. The configuration will contain the connection string for the database to talk to.

In the `ConfigureServices`, you need to add any additional dependencies your implementation requires. Add the following code from Listing 5-9 at the end of the method.

Listing 5-9. Adding Entity Framework dependencies

```
services.AddDbContext<PizzaPlaceDbContext>(options
  => options.UseSqlServer(Configuration.GetConnectionString("PizzaDb")));
```

This single statement tells ASP.NET Core that you will be using the `PizzaPlaceDbContext` and that you will be storing it in SQL Server. This code also looks up the connection string for the database in configuration, which you still need to add.

Right-click the `PizzaPlace.Server` project and select Add ➤ New Item. Type JSON in the search box and select App Settings File, as shown in Figure 5-11. Keep the default name of appsettings.json and click Add.

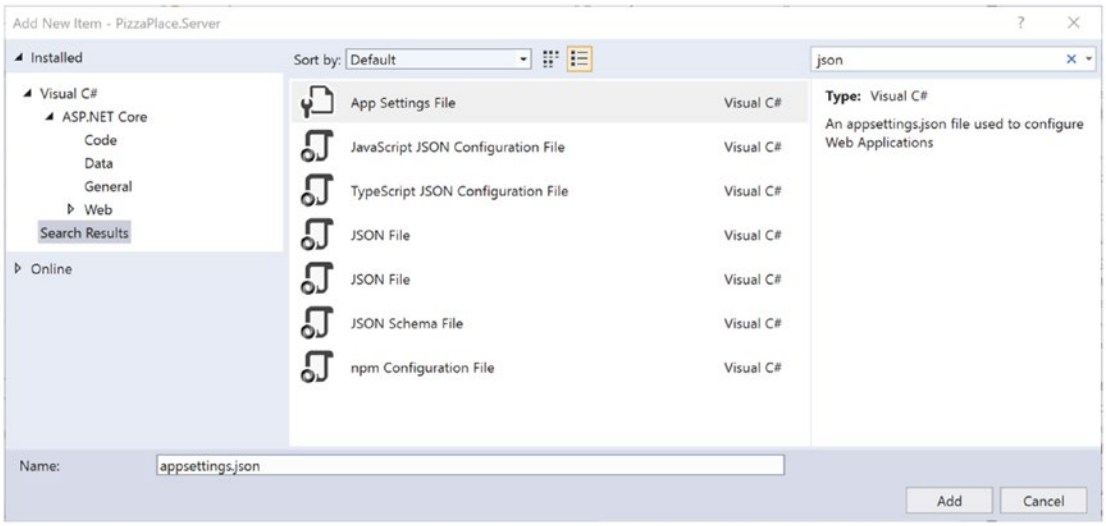

Figure 5-11. *Adding the application configuration file*

With Code, simply add a new file called `appsettings.json`. Double-click the new `appsettings.json` file to open it. ASP.NET Core uses a JSON file for configuration, and you need to add a connection string to the database. A database connection string tells your code where to find the database server, which database to use, and which credentials should be used to log in. Visual Studio added a configuration file such as in Listing 5-10. This connection string uses the `(localdb)\\MSSQLLocalDB` server, which is the server installed with Visual Studio. The only things you need to do are to set the database name by replacing _CHANGE_ME into a more suitable name for your database and to change the name of the connection. Of course, if you are using another database server, you will also have to change the server name too. Or read on to find out how to get the connection string with Visual Studio.

Listing 5-10. The `appsettings.json` configuration file

```
{
  "ConnectionStrings": {
    "PizzaDb": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_
    Connection=True;MultipleActiveResultSets=true"
  }
}
```

Finding Your Database Server's Connection String

If you are not sure which connection string to use, you can find the connection string in Visual Studio by selecting View ➤ SQL Server Object Explorer.

You can connect to a database by clicking the server icon with the little green + sign, shown in Figure 5-12.

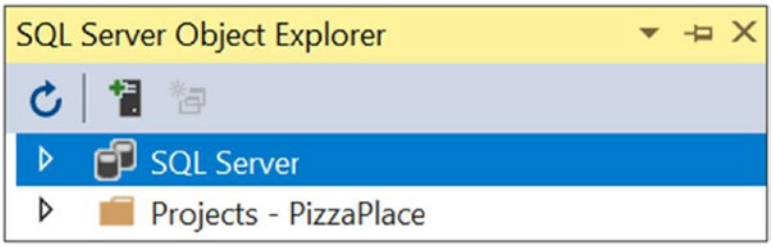

Figure 5-12. *SQL Server Object Explorer*

You can look for available database servers by expanding the Local, Network, or Azure as in Figure 5-13. I recommend that you try to find the `MSSQLLocalDB` database server. If you use another database server, you might need to change how to log in to your database. When you're ready, click Connect.

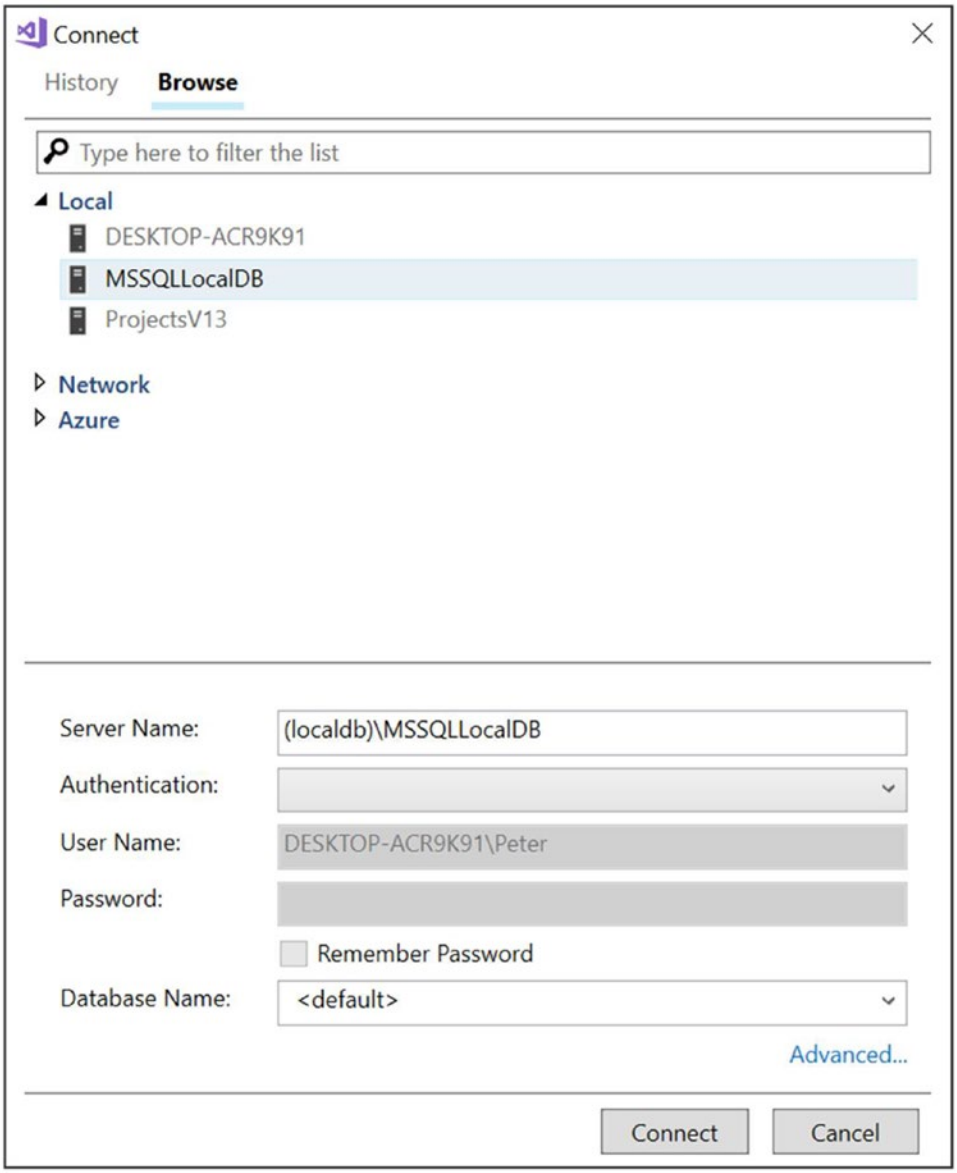

Figure 5-13. *Finding the connection string for a database*

Next, expand SQL Server from Figure 5-13 and select your server. Right-click it and select Properties. Now copy the connection string from the properties window and change the database name to `PizzaDb`.

Creating Your Code First Migration

You are almost ready to generate the database from the code. Start by adding the `Microsoft.EntityFrameworkCore.Design` NuGet package to the `PizzaPlace.Server` project.

Now you need to create a migration. A migration is a C# class that contains the changes that need to be made to the database to bring it up (or down) to the schema your application needs. This is done through a tool.

Start by selecting from the Visual Studio menu View ➤ Other Windows ➤ Package Manager Console, which you can see in Figure 5-14. Or use the command line (cmd.exe) if you prefer.

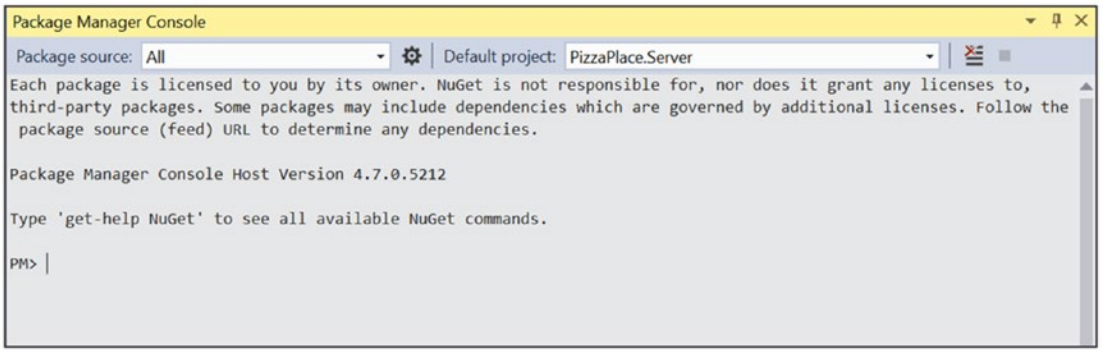

Figure 5-14. *The Package Manager Console*

Make sure that the default project is set to `PizzaPlace.Server`. This will make your commands target the selected project.

If you are using Code, use the integrated terminal or open a command prompt.

You must run the next command in the `PizzaPlace.Server` directory, so make sure you are in the correct directory. Optionally, type the following command to change the current directory to the `PizzaPlace.Server` project's directory:

```
cd PizzaPlace
cd Server
```

You might need to install a command-line tool as well. Run the following command to install the migration tool. You only need to install this tool once.

```
dotnet tool install --global dotnet-ef
```

Now execute the following command to create the migration:

```
dotnet-ef migrations add CreatingPizzaDb
```

Here you use the `dotnet-ef` tool to add a `new migration` called `CreatingPizzaDb`. You should see the following output (please ignore any differences in versions being shown):

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 3.1.1 initialized 'PizzaPlaceDbContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

Should you get an error or warnings, please review the code for the `Pizza` and the `PizzaPlaceDbContext` classes and try again.

This tool created a new `Migrations` folder in the `PizzaPlace.Server` project with two files similar to Figure 5-15 but with a different timestamp.

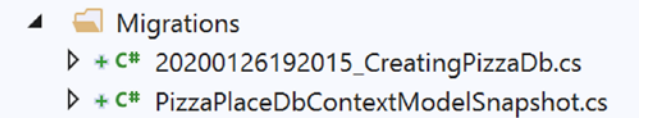

Figure 5-15. *The result of adding the first migration*

Open the `CreatingPizzaDb.cs` file from Listing 5-11 and look at what the tool did.

Listing 5-11. The `CreatingPizzaDb.cs` file

```
using Microsoft.EntityFrameworkCore.Migrations;

namespace PizzaPlace.Server.Migrations
{
  public partial class CreatingPizzaDb : Migration
  {
    protected override void Up(MigrationBuilder migrationBuilder)
    {
      migrationBuilder.CreateTable(
          name: "Pizzas",
          columns: table => new
```

```
            {
              Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
              Name = table.Column<string>(nullable: true),
              Price = table.Column<decimal>(type: "money", nullable: false),
              Spiciness = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
              table.PrimaryKey("PK_Pizzas", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
      migrationBuilder.DropTable(
          name: "Pizzas");
    }
  }
}
```

A migration class has two methods: `Up` and `Down`. The `Up` method will upgrade the database schema. In this case, it will create a new table called `Pizzas` with `Id`, `Name`, `Price`, and `Spiciness` columns.

The `Down` method downgrades the database schema, in this case, by dropping the column.

Generating the Database

Now you are ready to generate the database from your migration. With Visual Studio, go back to the Command Line or Package Manager Console window (View ➤ Other Windows ➤ Package Manager Console), or with Code, open the integrated terminal (View ➤ Terminal) and type the following command:

```
dotnet-ef database update --verbose
```

Because you asked the tool to be verbose, this will generate a lot of output, among which you will find the DDL statements executed, such as in Listing 5-12.

Listing 5-12. An extract from the database generation tool's output

```
CREATE TABLE [Pizzas] (
          [Id] int NOT NULL IDENTITY,
          [Name] nvarchar(max) NULL,
          [Price] money NOT NULL,
          [Spicyness] int NOT NULL,
          CONSTRAINT [PK_Pizzas] PRIMARY KEY ([Id])

);
```

This just created the database for you!

Let's have a look at the database. From Visual Studio, open View ➤ SQL Server Object Explorer and expand the tree for the `PizzaDb` database as in Figure 5-16 (you might need to refresh the database: right-click `Databases` and select `Refresh`).

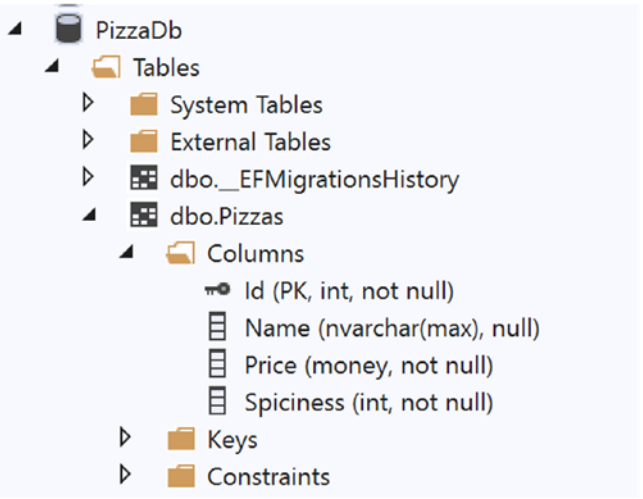

Figure 5-16. *SQL Server Object Explorer showing the PizzaDb database*

If you don't have Visual Studio, you can download *Azure Data Studio* from `www.microsoft.com/en-us/sql-server/developer-tools`. After installation ends, Azure Data Studio will start. Enter your server name and select `PizzaDb` from the drop-down list, as shown in Figure 5-17.

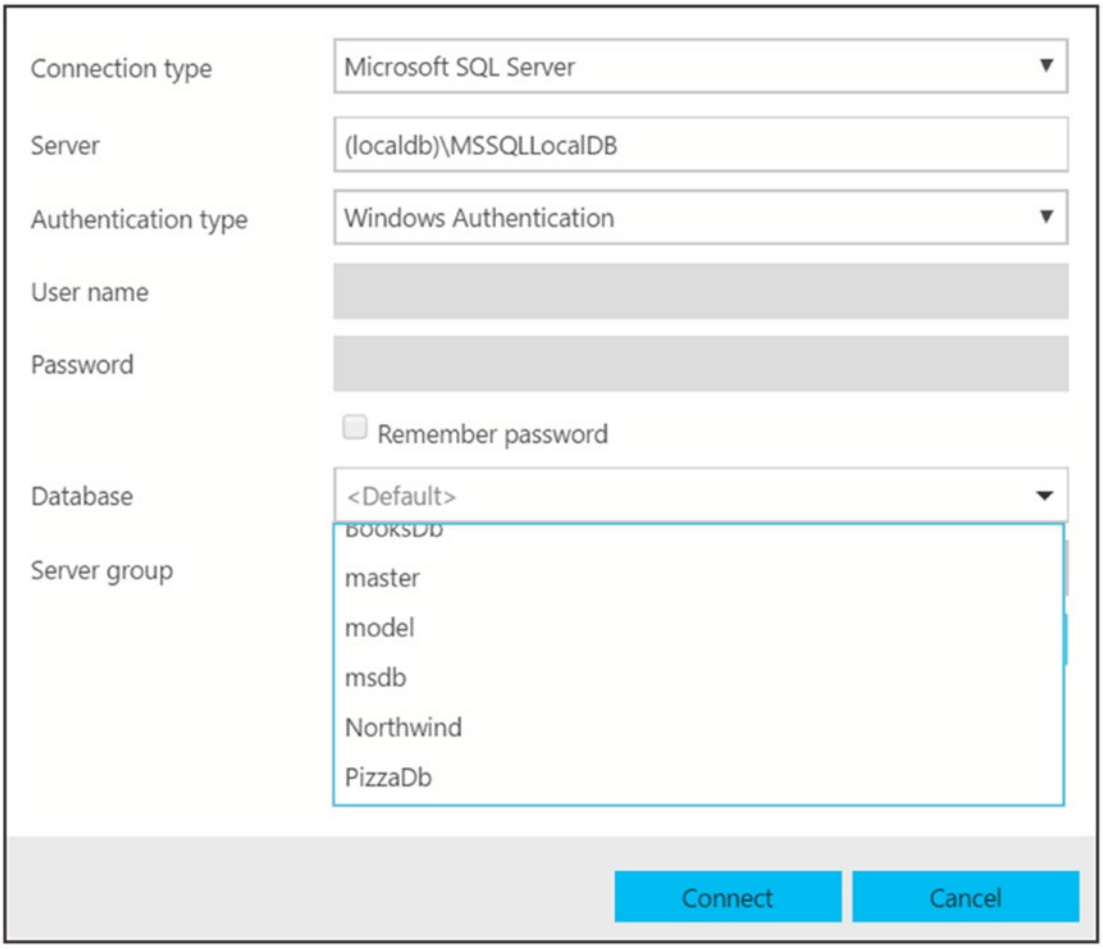

Figure 5-17. *Connection with Azure Data Studio*

Enhancing the Pizza Microservice

Let's add some functionality to the Pizza microservice so it uses the database instead of hard-coded data and add a method to insert a pizza in your database.

Open the `PizzaController` class, which sits in the `Controllers` folder of the `PizzaPlace.Server` project. Start by adding a constructor that takes the `PizzaPlaceDbContext` as an argument, as in Listing 5-13.

Listing 5-13. Injecting a `PizzaPlaceDbContext` instance into the controller

```
using Microsoft.AspNetCore.Mvc;
using PizzaPlace.Shared;
using System.Linq;

namespace PizzaPlace.Server.Controllers
{
  [ApiController]
  public class PizzasController : ControllerBase
  {
    //private static readonly List<Pizza> pizzas = new List<Pizza>
    //    {
    //      new Pizza(1, "Pepperoni", 8.99M, Spiciness.Spicy ),
    //      new Pizza(2, "Margarita", 7.99M, Spiciness.None ),
    //      new Pizza(3, "Diabolo", 9.99M, Spiciness.Hot )
    //    };
    private readonly PizzaPlaceDbContext db;

    public PizzasController(PizzaPlaceDbContext db)
      => this.db = db;
  }
}
```

To talk to the database, the `PizzasController` needs a `PizzaPlaceDbContext` instance, and as you learned in the chapter on *dependency injection*, you can use a constructor to do this. The constructor only needs to save the reference in a local field (for now).

You don't need the hard-coded list of pizzas, so remove the static field, and update the `GetPizza` method to use the `PizzaPlaceDbContext` instead, as in Listing 5-14. To get all the pizzas, you can simply use the `Pizzas` property of the `PizzaPlaceDbContext`. The Entity Framework will access the database when it accesses the `Pizzas` property.

Listing 5-14. Retrieving the pizzas from the database

```
[HttpGet("pizzas")]
public IQueryable<Pizza> GetPizzas()
  => this.db.Pizzas;
```

Now let's add a method to insert a new pizza in the database. Add the `InsertPizza` method from Listing 5-15 to the `PizzasController` class. This method will receive a pizza instance from the client as part of the POST request body, so you add the `HttpPost` attribute with the URI that you should post to. The `pizza` object will be posted in the request body, and this is why the `InsertPizza` method's `pizza` argument has the `FromBody` attribute to tell ASP.NET MVC Core to convert the body to a pizza instance. The method adds the pizza to the `PizzaPlaceDbContext Pizzas` table and then saves it to the database. The `InsertPizza` method then returns a 201: Created status code with the URI of the pizza as the response. There are many possible HTTP status codes that you could return from this method. But the most common of them have special helper methods that make it easy to return a certain status code, for instance, Ok(), NotFound(). In this case, you return a 201: Created status code. You will examine this response with Postman in the next part of this chapter.

Listing 5-15. The `InsertPizza` method

```
[HttpPost("pizzas")]
public IActionResult InsertPizza([FromBody] Pizza pizza)
{
  db.Pizzas.Add(pizza);
  db.SaveChanges();
  return Created($"pizzas/{pizza.Id}", pizza);
}
```

This is an introduction to REST services. Building real services with all the different approaches and best practices can take up a whole book. The idea of this chapter is to get you up and running.

Testing Your Microservice Using Postman

So now, you have your first microservice. But how do you test it? Previously, you used the browser to test the `GetPizzas` method, which uses the GET method. For other methods such as POST, PUT, and DELETE, you need a better tool. Here you will use Postman, which is a tool specifically for testing REST services.

Installing Postman

Open your favorite browser and go to `www.getpostman.com`. Download the application (click the Download the App button from Figure 5-18 and choose your platform) and install it.

By the time you read this book, the installation procedure may have changed a bit, so please follow the instructions from the installer.

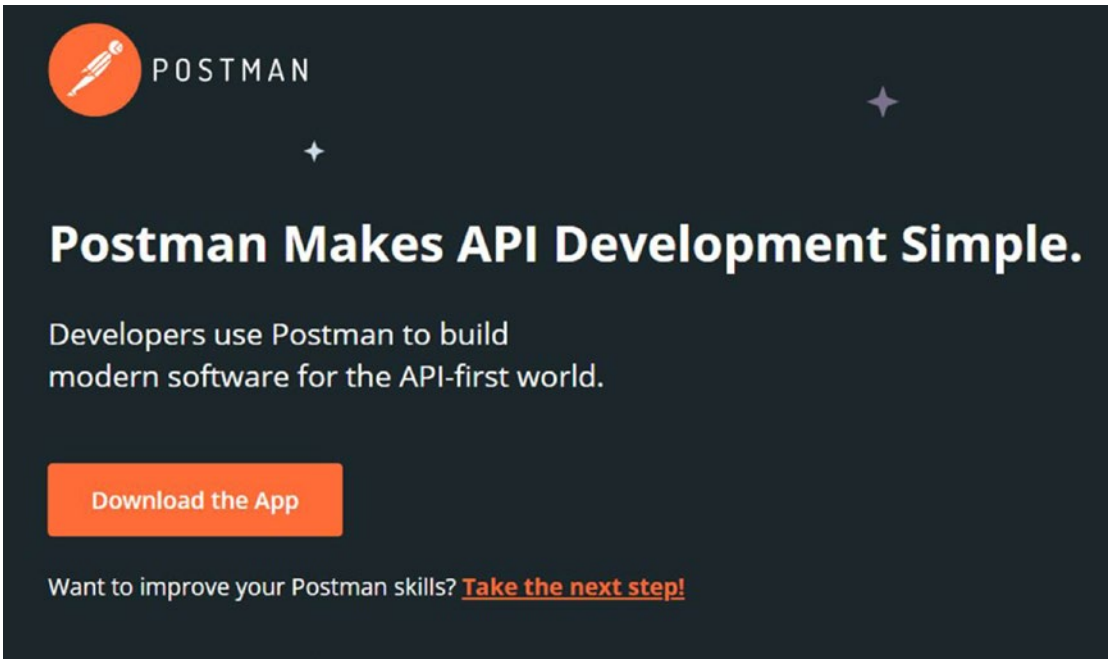

Figure 5-18. *The Postman web page*

After it has installed, run Postman.

Making REST Calls with Postman

Postman will open, and it will ask you what you want to do. Select Request, as shown in Figure 5-19.

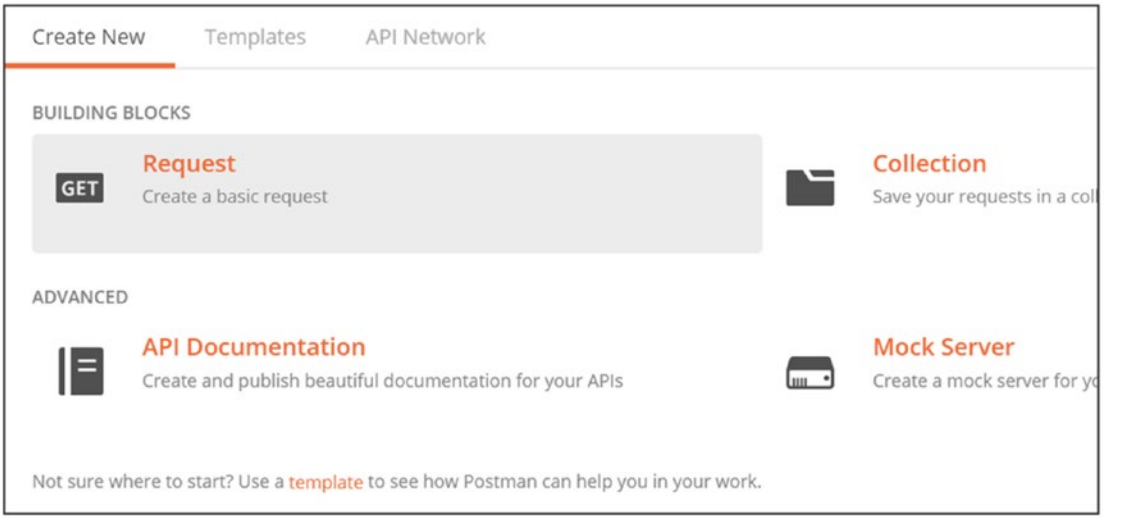

Figure 5-19. *Select Request to get started with Postman*

Then it will ask you where to save the request, so pick a name and a folder, as shown in Figure 5-20.

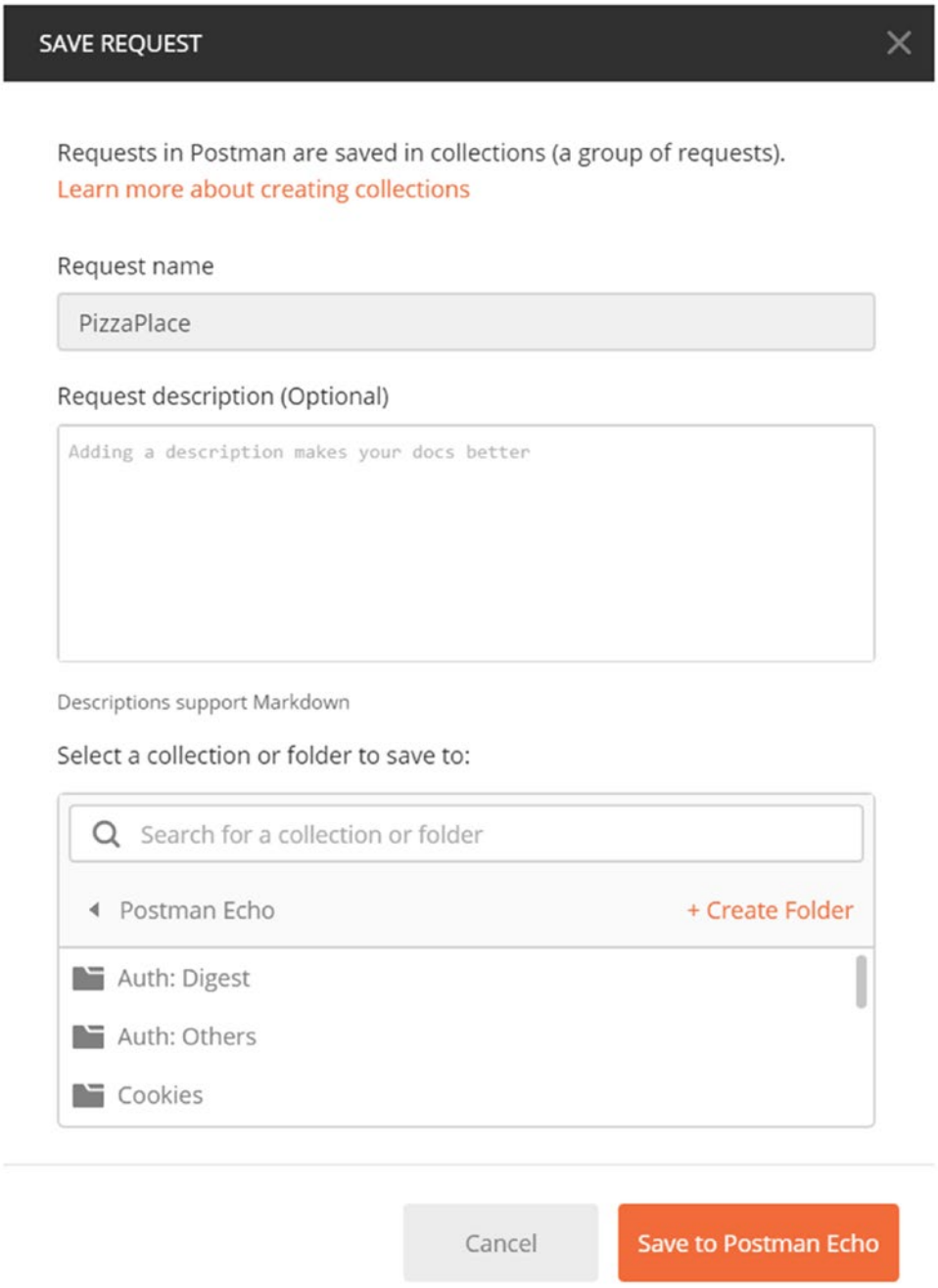

Figure 5-20. *Saving the request*

Making a GET Request

Now run the PizzaPlace solution and copy the URL from the browser. Paste it in Postman's "Enter the Request URL" field and append `/pizzas` as in Figure 5-21. Don't forget that you most likely will have a different port number!

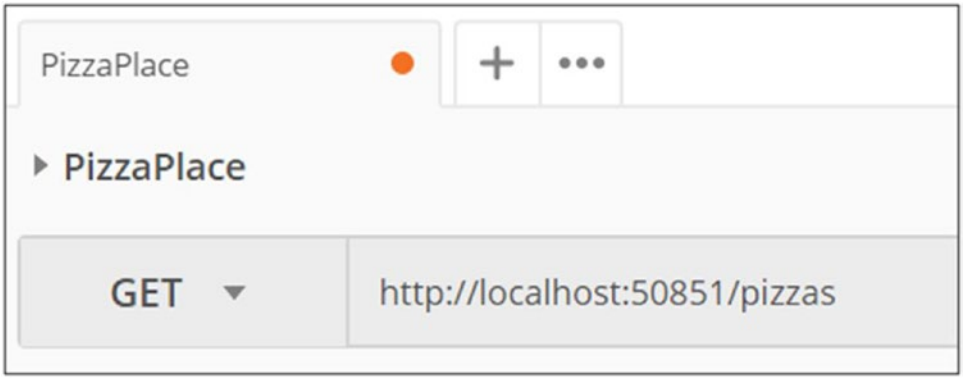

Figure 5-21. *Making a GET request with Postman*

Before you click Send, let's add the `Accept` header. Click the Headers tab and enter Accept as the key and application/json as the value. Please refer to Figure 5-22 for reference.

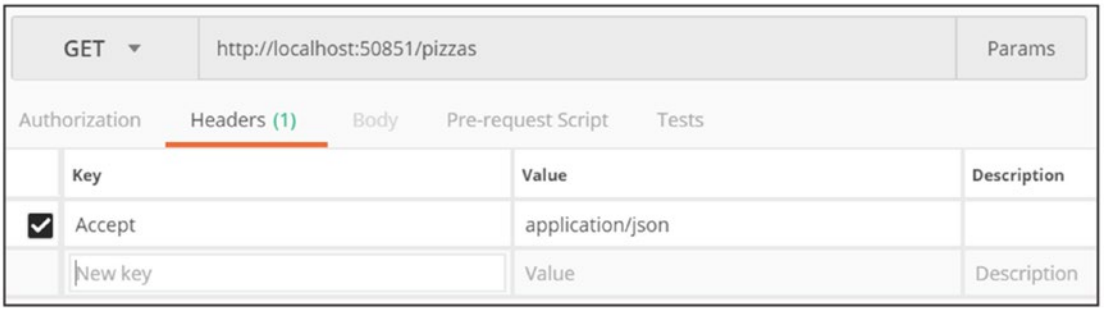

Figure 5-22. *Adding headers to the request in Postman*

Now you can click Send. You should receive an empty list as in Figure 5-23 (which is normal because you haven't added any rows to the pizza table yet).

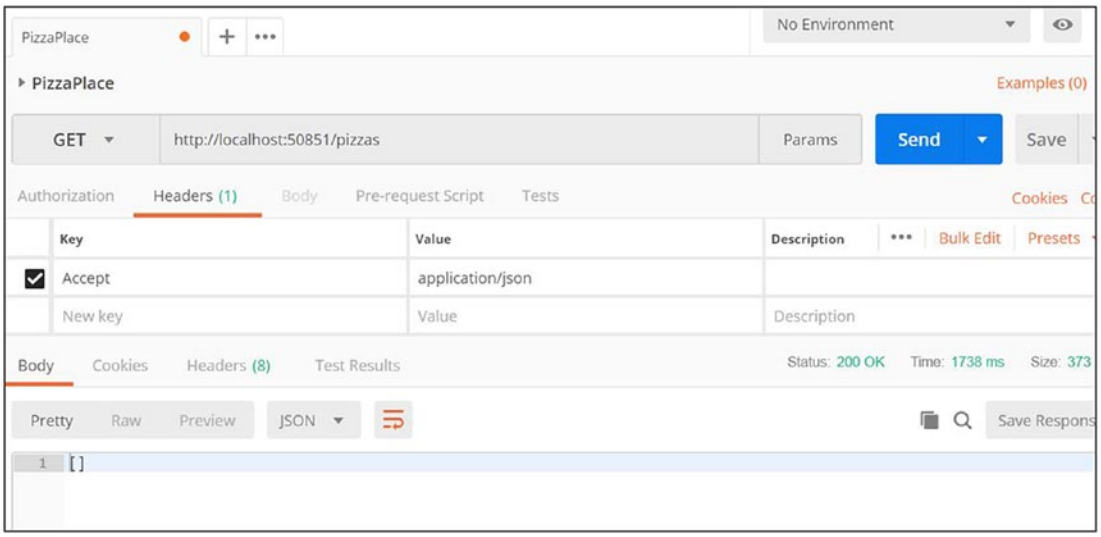

Figure 5-23. *Receiving an empty list of pizzas from the server*

Inserting Pizzas with POST

Let's add a couple of pizzas to the database. At the top of Postman, you will find a tab with a plus sign. Click it to add another tab. Select POST as the method and copy the URI from the previous tab, as shown in Figure 5-24.

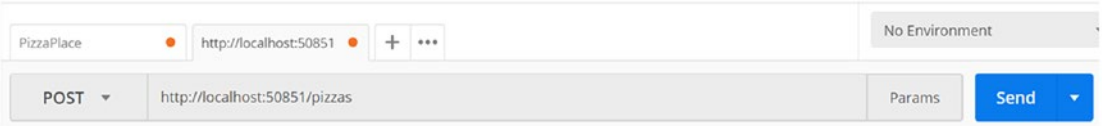

Figure 5-24. *Starting with the POST request*

Now select the Headers section and add a new header with key Content-Type and value application/json like in Figure 5-25.

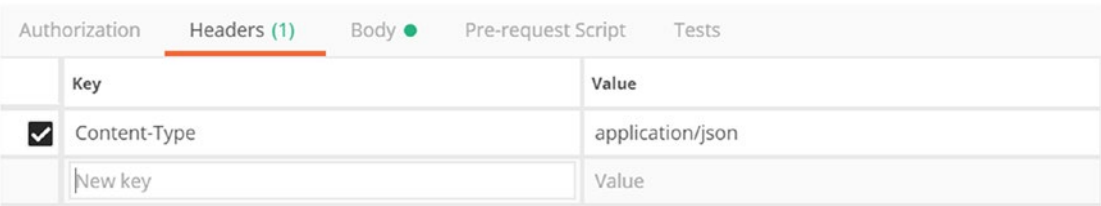

Figure 5-25. *Adding the Content-Type header for the POST request*

Now select the Body section, click the raw format radio button, and enter a pizza object using JSON. Please refer to Figure 5-26. Note that this raw string contains the pizza's properties serialized as JSON and that you don't need to send the `Id` property because the server will generate the id when it gets inserted into the database.

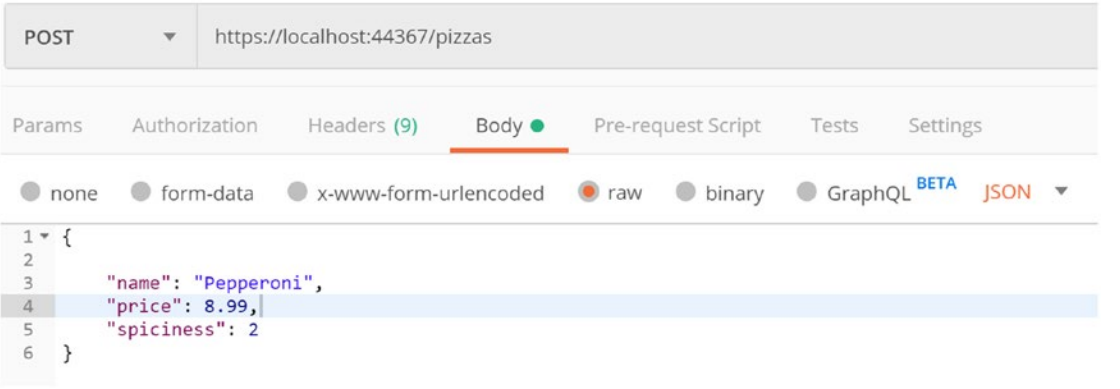

Figure 5-26. *Entering a pizza using JSON*

Ensure your PizzaPlace application is still running.

Click the Send button. If all is well, you should receive a positive 201: Created response. In the response area of Postman, select the Headers tab as in Figure 5-27. Look for the Location header. It will show the new URI given to this pizza. This Location header is returned by the `Created` method you called as the last line of Listing 5-15.

Figure 5-27. *The POST response in Postman*

Click the first tab where you created the GET request and click Send again. Now you should have a list of pizzas (a list of one). Try creating a couple of other pizzas. Figure 5-28 is my result after adding three pizzas.

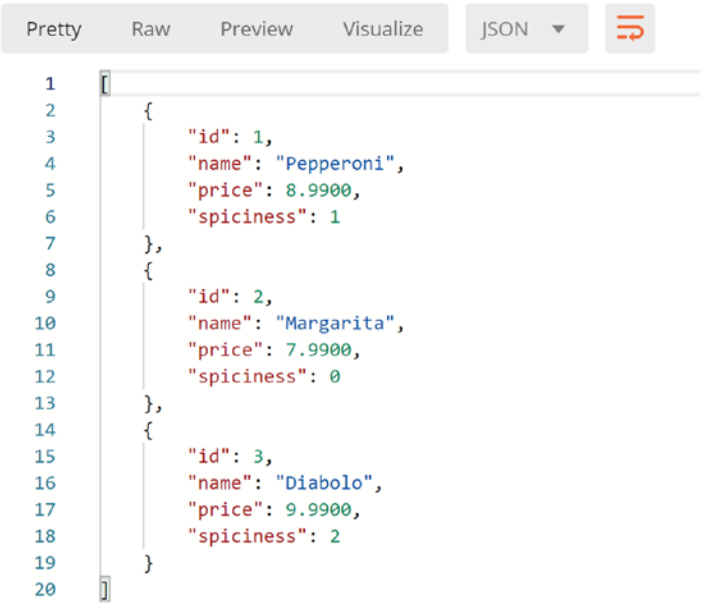

Figure 5-28. *A list of pizzas stored in the database*

Summary

In this chapter, you had a look at how to store data on the server using Entity Framework Core and how to expose that data using REST and microservices. You added a pizza service to the PizzaPlace application and then went on testing it with Postman.

In the next chapter, you will learn how to talk to your service(s) from Blazor.

CHAPTER 6

Communication with Microservices

In the previous chapter, you built a microservice using ASP.NET Core and Entity Framework Core to retrieve the menu of pizzas from the server. In this chapter, you will add support to the Blazor client to talk to that microservice. You will also complete the project by adding support for making an order.

Using the HttpClient Class

Start by opening the `MyFirstBlazor` solution you created in the first chapter. You will use this project to examine the template that was created for you. We will start by looking at the server-side of the solution, then the shared project's code, and then the client-side.

Examining the Server Project

Look at the `MyFirstBlazor.Server` project and look for the `WeatherForecastController` class, which is in Listing 6-1.

Listing 6-1. The `SampleDataController` class

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using MyFirstBlazor.Shared;
using System;
using System.Collections.Generic;
using System.Linq;
```


P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_6

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

    private readonly ILogger<WeatherForecastController> logger;

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

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

The `WeatherForecastController` class exposes one REST endpoint at `/WeatherForecast` to retrieve a list of `WeatherForecast` objects. This time the `WeatherForecastController` uses the `[Route("[controller]")]` attribute to set up the

endpoint to generically listen to a URI that contains the name of the controller (without the suffix "Controller") and then uses the `[HttpGet]` attribute to expect the method name as the third part of the URI.

To invoke this method, you should use a GET on the `WeatherForecast` URI, which you can try with your browser (or if you prefer, Postman). Run the solution and type the URI in your browser (don't forget you will have a different port number) which will result in Figure 6-1 (expect different weather).


```
[{"date":"2020-01-29T15:59:42.1870928+01:00","temperatureC":-20,"summary":"Mild","temperatureF":-3},
{"date":"2020-01-30T15:59:42.1876265+01:00","temperatureC":35,"summary":"Scorching","temperatureF":94},
{"date":"2020-01-31T15:59:42.1876376+01:00","temperatureC":19,"summary":"Scorching","temperatureF":66},
{"date":"2020-02-01T15:59:42.1876381+01:00","temperatureC":-20,"summary":"Chilly","temperatureF":-3},
{"date":"2020-02-02T15:59:42.1876384+01:00","temperatureC":53,"summary":"Bracing","temperatureF":127}]
```


Figure 6-1. *Invoking the service using the browser*

The `Get` method from Listing 6-1 uses a random choice of temperatures and summaries to generate these forecasts, which is great for a demo.

Why Use a Shared Project?

Now open the `WeatherForecast` class from the `MyFirstBlazor.Shared` project, which is in Listing 6-2.

Listing 6-2. The Shared `WeatherForecast` class

```
using System;

namespace MyFirstBlazor.Shared
{
  public class WeatherForecast
  {
    public DateTime Date { get; set; }

    public int TemperatureC { get; set; }

    public string Summary { get; set; }

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

This `WeatherForecast` class is straightforward, containing the `Date` of the forecast, the temperature in Celsius and Fahrenheit, and a `Summary`, but I want to draw your attention to the fact that this class lives in the Shared project. This shared project is used both by the server and the client project.

If you ever created a web app with JavaScript, you should be familiar with the experience of building a class for the server project, for example, in C#, and building another class in JavaScript (or TypeScript) for the client. You must make sure that both classes serialize to the same JSON format; otherwise, you will get runtime errors or, even worse, lose data! If the model grows, you must update both classes again. This is a HUGE maintenance problem in these kinds of projects because you run the risk of updating only one side on a busy workday.

With Blazor, you don't suffer from this because both server and client use C#. And that is why there is a Shared project. You put your classes here and they are shared between the server and client, and then you use them by simply adding a reference to the Shared project. Adding another piece of data means updating a shared class, which works easily! No longer must you update two pieces of code.

Looking at the Client Project

Now, look at the `MyFirstBlazor.Client` project. Inside the `Pages` folder, you will find the `FetchData` component from Listing 6-3.

Listing 6-3. The `FetchData` component

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

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from the server.</p>

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

```
else
{
  <Grid Items="forecasts" TItem="WeatherForecast">
    <Header>
      <th>Date</th>
      <th>Temp. (C)</th>
      <th>Temp. (F)</th>
      <th>Summary</th>
    </Header>
    <Row Context="forecast">
      <td>@forecast.Date</td>
      <td>@forecast.TemperatureC</td>
      <td>@forecast.TemperatureF</td>
      <td>@forecast.Summary</td>
    </Row>
    <Footer>
      <td colspan="4">Spring is in the air!</td>
    </Footer>
  </Grid>
}

@code {
  private WeatherForecast[] forecasts;

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

Let's look at this line by line. The first line in Listing 6-3 adds a Razor `@using` statement for the Shared project's namespace to the component. You need this because you use the `WeatherForecast` class from the Shared project. Just like in C#, you use `@using` statements in Razor to refer to classes from another namespace.

The second line adds the path for routing. You will look at routing in the next chapter. For the moment, you should know that when the URI is `/fetchdata`, the `FetchData` component will be shown in the browser.

On the third line, you inject the `HttpClient` instance using the `@inject` syntax from Razor. The `HttpClient` class is the one you will use to talk to the server. You will learn about the `HttpClient` class in detail later in this chapter.

I do want to point out that you should never instantiate an instance of the `HttpClient` class yourself. Blazor sets up the `HttpClient` class in a special way, and if you create an instance yourself, it simply will not work as expected! Another reason not to create an instance yourself is that this is a dependency of the `FetchData` component and components should never create dependencies themselves!

A little lower down in Listing 6-3 you will find an `@if` statement. Because you fetch the data from the server using an asynchronous way, the `forecasts` field will initially hold a `null` reference. So, if the forecasts field has not been set, you tell the user to wait. If you have a slow network, you can see this happening. When you test your Blazor application on your machine, the network is fast, but you can emulate a slow network using the browser (in this case, using Google Chrome).

How to Emulate a Slow Network in Chrome

Start your Blazor project so the browser opens the home page. Now open the debugger tools from the Chrome browser (on Windows, you do this by pressing F12) and select the Network tab as in Figure 6-2. On the right side, you should see a drop-down list that allows you to select which kind of network to emulate. Select Slow 3G.

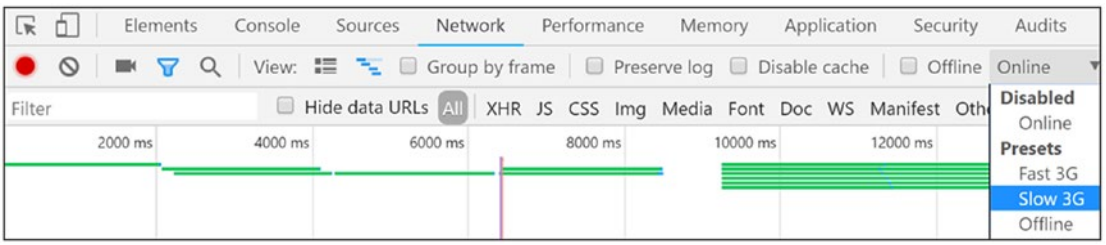

Figure 6-2. *Using the Chrome browser debugger to emulate a slow network*

Next, select the Fetch data tab on your Blazor site (should you already be on this tab, select another tab and then the Fetch data tab). Because you now are using a slow network, the Loading... feedback will appear, as shown in Figure 6-3.

After testing your Blazor website with a slow network, don't forget to select `Online` from the drop-down from Figure 6-2 to restore your network to its normal speed.

Figure 6-3. *The Loading... feedback with a slow network*

If the `forecasts` field holds data, your Razor file will show a table with the forecasts by iterating over them, as you can see in the `else` part of Listing 6-3.

Onto the `@code` section of the `FetchData` Razor file. First, you declare a field called `forecasts` to hold an array of `WeatherForecast` instances. Initially, the `forecasts` field will hold a `null` value. You then override the `OnInitializedAsync` method. Blazor components have two methods that get called when the component has been initialized: `OnInitialized` and `OnInitializedAsync`. Because you fetch the data from the server using an asynchronous API, you need to put your code in `OnInitializedAsync`. The `OnInitializedAsync` method is prefixed with C#'s `async` keyword, which makes it a breeze to call async APIs with the `await` keyword.

Asynchronous communication means that the client needs to wait a fair amount for the result to be returned. Instead of using a call that will stop Blazor from completing other requests (freezing the user interface), you use the `OnInitializedAsync` method, which will wait in the background for the result.

You use the `Http.GetFromJsonAsync<WeatherForecast[]>("SOME URI")` to invoke the server's `GET` endpoint at the `URI`, and you tell the `GetFromJsonAsync` method (using generics) to expect an array of `WeatherForecast` objects. When the result comes back from the server, you put the result into the `forecasts` field and Blazor will take care of re-rendering the UI with your new data, as shown in Figure 6-4.

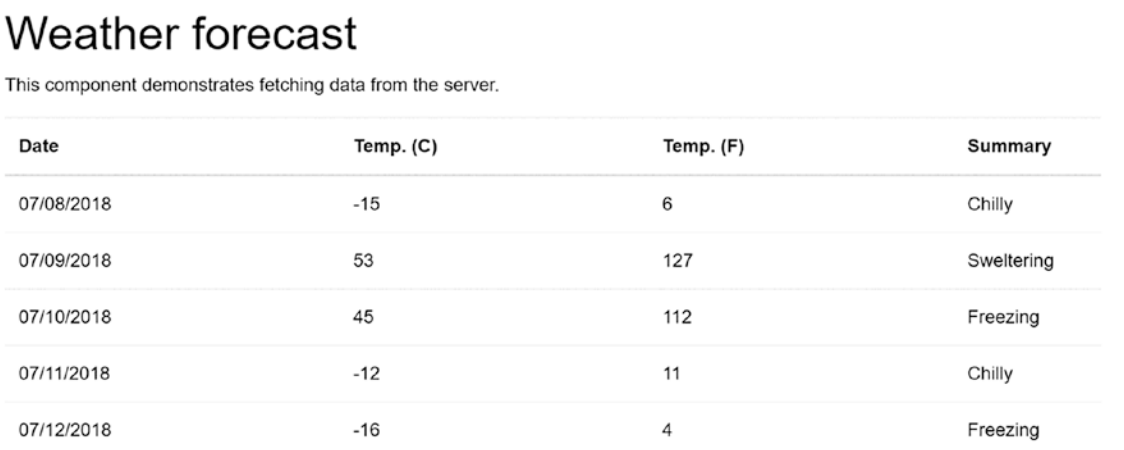

Figure 6-4. *Displaying the WeatherForecast objects*

Understanding the HttpClient Class

All communication between the client and the server passes through the `HttpClient` class. This is the same class other .NET frameworks use, and its role is to make the HTTP request to the server and to expose the result from the server. It also allows you to exchange binary or other formatted data, but in Blazor, we normally use JSON.

Google has defined a more efficient protocol called protocol buffers, which is also supported by Blazor. If you need to send a lot of data, you might want to look at protocol buffers.

The HttpClientJsonExtensions Methods

To make it a lot easier to talk to JSON (micro)services, Blazor provides you with a bunch of handy extension methods that take care of converting between .NET objects and JSON, which you can find in the `HttpClientJsonExtensions` class. I advise you to use these methods, so you don't have to worry about serializing and deserializing JSON.

GetFromJsonAsync

The `GetFromJsonAsync` extension method makes an asynchronous GET request to the specified URI. Its signature is in Listing 6-4.

Listing 6-4. The `GetFromJsonAsync` extension method signature

```
public static Task<T> GetFromJsonAsync<T>(
                        this HttpClient httpClient,
                        string requestUri)
```

Because it is an extension method, you call it as a normal instance method on the `HttpClient` class, as shown in Listing 6-5.

This is also true for the other extension methods.

Listing 6-5. Using the `GetFromJsonAsync` extension method

```
forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>
                  ("api/SampleData/WeatherForecasts");
```

`GetFromJsonAsync<T>` will expect the response to contain JSON as specified by the generic argument. For example, in Listing 6-5, it expects an array of `WeatherForecast` instances. You invoke the `GetFromJsonAsync` method by prefixing it with the `await` keyword, which makes the call asynchronous. Don't forget that you can only use the `await` keyword in methods and lambda functions that are async.

If you want to learn more about asynchronous programming, you can refer to https://docs.microsoft.com/en-us/dotnet/csharp/async.

You can always inspect the request and response using your browser's debugger. Run your Blazor project and open the browser's debugger on the Network tab. Now select the Fetch data tab in your Blazor website to make it load the data and look at the browser's Network tab, as shown in Figure 6-5.

You can always clear the Network tab from previous requests before making the request using the clear button, which in Chrome looks like a circle with a slash through it (the forbidden sign).

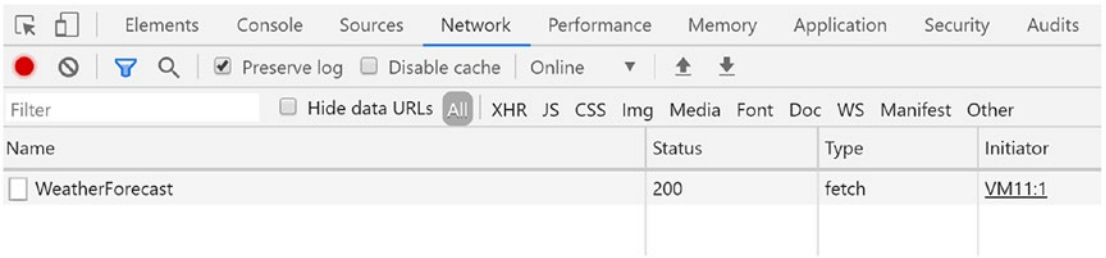

Figure 6-5. *Inspecting the network using the browser's debugger*

See the `WeatherForecast` entry in Figure 6-5? Now you can click that entry to look at the request and response. Let's start with the request preview shown in Figure 6-6 (you will get different results).

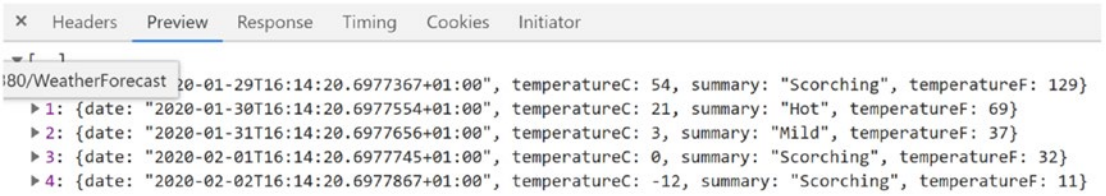

Figure 6-6. *Using the Preview tab to look at the response*

Using the Preview tab, you can see the server's response.

If you want to look at the request and response headers, you can click the Headers tab, as shown in Figure 6-7.

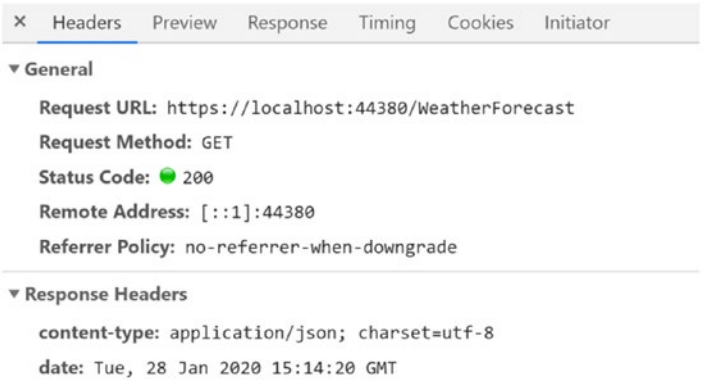

Figure 6-7. *Using the Headers tab to look at the request and the request/response headers*

Here you can see the request's URL and GET method (the request method). It also shows the HTTP status code 200 OK. Scroll down to look at the headers. One of the response headers is Content-Type with a value of `application/json`, which was set by the server telling the client to expect JSON.

PostAsJsonAsync

The `PostAsJsonAsync` extension method makes a POST request with the content argument serialized in the request body as JSON to the specified URI. Its signature is in Listing 6-6.

Listing 6-6. The `PostAsJsonAsync` method's signature

```
public static Task<HttpResponseMessage> PostAsJsonAsync<TValue>(
  this HttpClient client, string? requestUri,
  TValue value, JsonSerializerOptions? options = null,
  CancellationToken cancellationToken = default(CancellationToken))
```

This method returns a HttpResponseMessage instance, which you can check for any errors using the EnsureSuccessStatusCode() method. That is all you need to do if you don't expect any data back from the server. If you do need to retrieve some instance from the PostAsJsonAsync method, you can parse the result back out using code from Listing 6-7.

Listing 6-7. The `PostAsJsonAsync<T>` method's signature

```
HttpResponseMessage response = await this.httpClient.PostAsJsonAsync("/
orders", basket);
response.EnsureSuccessStatusCode();
var jsonResponse = await response.Content.ReadAsStreamAsync();
var review = await JsonSerializer.DeserializeAsync<Basket>(jsonResponse);
```

PutAsJsonAsync

The `PutAsJsonAsync` extension method makes a PUT request with the content argument serialized as JSON in the request body to the specified URI. Its signature is in Listing 6-8. Its usage is very similar to `PostAsJsonAsync`; the only difference is that it uses the PUT method.

Listing 6-8. The `PutAsJsonAsync` method's signature

```
public static Task<HttpResponseMessage> PutAsJsonAsync<TValue>(this
HttpClient client, string? requestUri, TValue value, JsonSerializerOptions?
options = null, CancellationToken cancellationToken =
default(CancellationToken))
```

Retrieving Data from the Server

So now you are ready to implement the services you introduced earlier. Open the `PizzaPlace` solution and look in the `PizzaPlace.Client` project for `Program.cs`, which is shown in Listing 6-9.

Listing 6-9. Your Blazor project's `Program` class

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;
using System.Threading.Tasks;
```

```
namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService, HardCodedMenuService>();
      builder.Services.AddTransient<IOrderService, ConsoleOrderService>();
      builder.Services.AddSingleton<State>();
      await builder.Build().RunAsync();
    }
  }
}
```

In the `Main` method, you added two services, `HardCodedMenuService` and `ConsoleOrderService`. Let's replace these fake implementations with real services that talk to the server.

With Visual Studio, right-click the `PizzaPlace.Client` project and select Add ➤ New Folder from the drop-down menu. With Code, right-click the `PizzaPlace.Client` project and select New Folder. Name this folder `Services`. Now add a new class to this folder called `MenuService`, which can be found in Listing 6-10.

Again, you are applying the principle of single responsibility where you encapsulate how you talk to the server in a service. This way, you can easily replace this implementation with another one should the need occur.

Listing 6-10. The `MenuService` class

```
using Microsoft.AspNetCore.Components;
using PizzaPlace.Shared;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

```
namespace PizzaPlace.Client.Services
{
  public class MenuService : IMenuService
  {
    private readonly HttpClient httpClient;

    public MenuService(HttpClient httpClient)
      => this.httpClient = httpClient;

    public async Task<Menu> GetMenu()
    {
      Pizza[] pizzas = await this.httpClient.GetFromJsonAsync<Pizza[]>
      ("/pizzas");
      return new Menu { Pizzas = pizzas.ToList() };
    }
  }
}
```

You start by adding a constructor to this class taking the `MenuService`'s dependency on `HttpClient`, and you store it in a field named `httpClient`. Then you implement the `IMenuService` interface's `GetMenu` method where you talk to the server calling the `GetFromJsonAsync` on the server's `/pizza` endpoint. Note that the `/pizza` endpoint is relative to the site's base (`<base href="/" />`), which can be found in the `index.html` file. Because the `MenuService` service returns a menu and not a list of pizzas, you wrap the list of pizzas you got from the server into a `Menu` object. That's it!

Using the principle of single responsibility results in many small classes, which are easier to understand, maintain, and test.

You have the service; now you need to tell dependency injection to use the `MenuService`. In the `Program` class's `Main` method, replace it as shown in Listing 6-11.

Listing 6-11. Replacing the `HardCodedMenuService` with the `MenuService`

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
```

```
using PizzaPlace.Shared;
using System.Threading.Tasks;

namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService, MenuService>();
      builder.Services.AddTransient<IOrderService, ConsoleOrderService>();
      builder.Services.AddTransient(sp => new HttpClient { BaseAddress =
      new Uri(builder.HostEnvironment.BaseAddress) });
      builder.Services.AddSingleton<State>();
      await builder.Build().RunAsync();
    }
  }
}
```

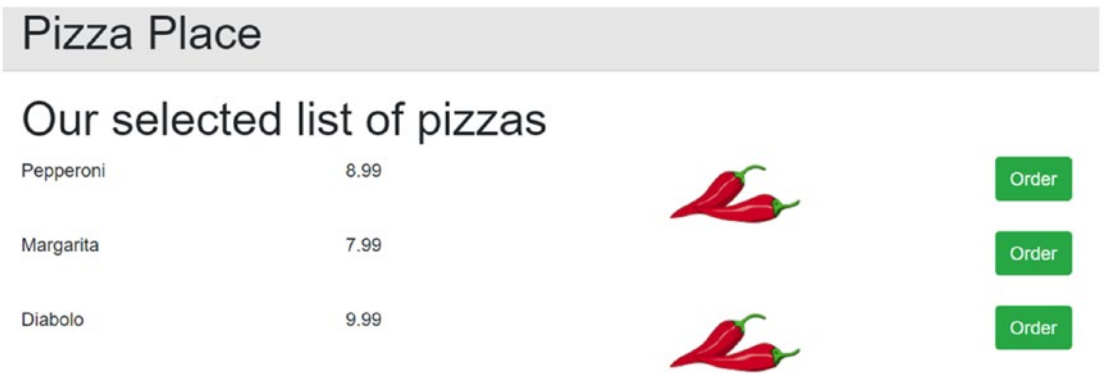

Figure 6-8. *The PizzaPlace application showing the pizzas from the database*

Run your project. You should see the list of pizzas (retrieved from your database) as in Figure 6-8!

You will probably first see an empty menu, especially on a slow network. This might confuse some customers, so let's add some UI to tell the customer to wait a bit. Update `PizzaList.razor` to look like Listing 6-12.

Listing 6-12. Adding a loading UI to the `PizzaList` component

```
<h1>@Title</h1>

@if (Menu == null || Menu.Pizzas == null || Menu.Pizzas.Count == 0)
{
  <div style="height:20vh;" class="pt-3">
    <div class="mx-left pt-3" style="width:200px">
      <div class="progress">
        <div class="progress-bar bg-danger
                    progress-bar-striped
                    progress-bar-animated w-100"
             role="progressbar"
             aria-valuenow="100" aria-valuemin="0"
             aria-valuemax="100"></div>
      </div>
    </div>
  </div>
}
else
{
  @foreach (var pizza in Menu.Pizzas)
  {
    <PizzaItem Pizza="@pizza"
               ButtonClass="btn btn-success"
               ButtonTitle="Order"
               Selected="@Selected" />
  }
}

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

  [Parameter]
  public Menu Menu { get; set; }
```

```
    [Parameter]
    public EventCallback<Pizza> Selected { get; set; }
}
```

If the menu has not been loaded yet, it will display a progress bar like in Figure 6-9.

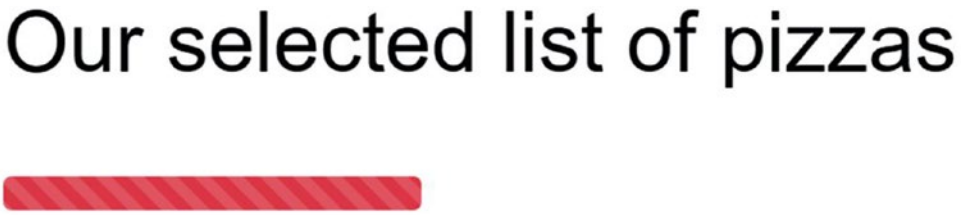

Figure 6-9. *Showing a loading progress bar while loading the menu*

Storing Changes

Now onto storing the order from the customer. Because you don't have a microservice yet for storing the order, you will build this first, and then you will implement the client service to send the order to the server.

Updating the Database with Orders

What is an order? Every customer has an order, and each order has one or more pizzas. A pizza can belong to more than one order, which can result in a specific problem: you need a many-to-many relation between pizzas and orders. In relational databases, this is done by adding a table between orders and pizzas, which you will map using a `PizzaOrder` class, as shown in Figure 6-10.

Entity Framework Core 3.1 does not have support for hiding this extra table, so you need to do this manually. In future versions, Microsoft might add this feature.

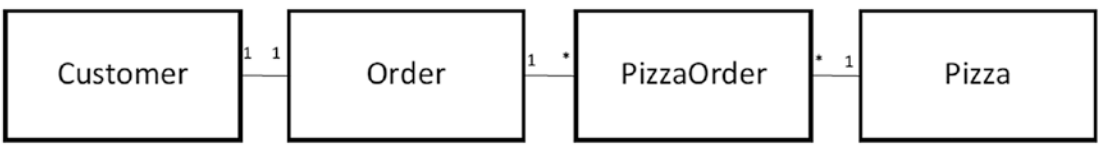

Figure 6-10. *Modeling the relationships*

Add a new class to the `PizzaPlace.Shared` project called `PizzaOrder`, as shown in Listing 6-13. This class keeps track of which pizzas belong to an order. For every pizza in the order, there will be an instance of this class, referring to the order and the pizza.

Listing 6-13. The `PizzaOrder` class

```
namespace PizzaPlace.Shared
{
  public class PizzaOrder
  {
    public int Id { get; set; }
    public Order Order { get; set; }
    public Pizza Pizza { get; set; }
  }
}
```

Next, add a new class named `Order` to the `PizzaPlace.Shared` project, as shown in Listing 6-14. This class tracks the customer's order, which pizzas are on the order, and the total price of the order.

Listing 6-14. The `Order` class

```
using System.Collections.Generic;

namespace PizzaPlace.Shared
{
  public class Order
  {
    public int Id { get; set; }

    public Customer Customer { get; set; }

    public int CustomerId { get; set; }

    public List<PizzaOrder> PizzaOrders { get; set; }

    public decimal TotalPrice { get; set; }

  }
}
```

Update the Customer class from the PizzaPlace.Shared project by adding an Order to it, as in Listing 6-15. We need to add this to find the customer's order. For the moment, we have a new customer instance for each new order.

Listing 6-15. The Customer class

```
using System.ComponentModel.DataAnnotations;

namespace PizzaPlace.Shared
{
  public class Customer
  {
    public int Id { get; set; }
    ...
    public Order Order { get; set; }
  }
}
```

Now you can add these tables to the PizzaPlaceDbContext class, which can be found in Listing 6-16.

Listing 6-16. The updated PizzaPlaceDbContext class

```
using Microsoft.EntityFrameworkCore;
using PizzaPlace.Shared;

namespace PizzaPlace.Server
{
  public class PizzaPlaceDbContext : DbContext
  {
    public PizzaPlaceDbContext(DbContextOptions<PizzaPlaceDbContext> options)
      : base(options) { }

    public DbSet<Pizza> Pizzas { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders{ get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);
```

```
        var pizzaEntity = modelBuilder.Entity<Pizza>();

        pizzaEntity.HasKey(pizza => pizza.Id);
        pizzaEntity.Property(pizza => pizza.Price)
          .HasColumnType("money");

        var customerEntity = modelBuilder.Entity<Customer>();
        customerEntity.HasKey(customer => customer.Id);
        customerEntity.HasOne(customer => customer.Order)
                      .WithOne(order => order.Customer)
                      .HasForeignKey<Order>(
                         order => order.CustomerId);

        var orderEntity = modelBuilder.Entity<Order>();
        orderEntity.HasKey(order => order.Id);
        orderEntity.HasMany(order => order.PizzaOrders)
                   .WithOne(pizzaOrder => pizzaOrder.Order);

        modelBuilder.Entity<PizzaOrder>()
          .HasOne(po => po.Pizza)
          .WithMany();
    }
  }
}
```

Here you have added the `Customers`, `Orders`, and `PizzaOrders` tables, and in the `OnModelCreating` method, you explain to Entity Framework Core how things should be mapped.

A `Customer` has a primary key `Id` and a one-to-one relation with an `Order`. When using a one-to-one relation, Entity Framework Core needs to know which side is the *master* in the relation, and that is why you need to add a foreign key to the `Order` class with the `HasForeignKey<Order>` method.

An `Order` has a primary key `Id`, and it has a many-to-one relationship with a `PizzaOrder` (one `Order` can have many `PizzaOrders`, and each `PizzaOrder` belongs to one `Order`).

Finally, you indicate that a PizzaOrder has one pizza and that one pizza can belong to many orders. Since we don't need to find the orders for a pizza, we don't need to store the link to the PizzaOrder, and that is why we call the parameterless `WithMany()`.

Build your project and fix any compiler error(s) you might have.

Now it is time to create another migration. This migration will update your database with your new tables. In Visual Studio, open the Package Manager Console (which you can find via View ➤ Other Windows ➤ Package Manager Console). With Code, open the integrated terminal.

Change the directory to the `PizzaPlace.Server` project. Now type the following command. This will create a migration for your new database schema.

```
dotnet-ef migrations add HandlingOrders
```

Apply the migration to your database by typing the following command:

```
dotnet-ef database update
```

This concludes the database part.

Building the Order Microservice

Time to build the microservice for taking orders. With Visual Studio, right-click the `Controllers` folder of the `PizzaPlace.Server` project and select New ➤ Controller. Select an Empty API Controller and name it `OrdersController`. With Code, right-click the `Controllers` folder of the `PizzaPlace.Shared` project and select New File, naming it `OrdersController`. This class can be found in Listing 6-17.

Listing 6-17. The `OrdersController` class

```
using Microsoft.AspNetCore.Mvc;
using PizzaPlace.Shared;
using System.Collections.Generic;
using System.Linq;

namespace PizzaPlace.Server.Controllers
{
  [Route("api/[controller]")]
  [ApiController]
  public class OrdersController : ControllerBase
  {
    private readonly PizzaPlaceDbContext db;
```

```
    public OrdersController(PizzaPlaceDbContext db)
      => this.db = db;

    [HttpPost("/orders")]
    public IActionResult CreateOrder([FromBody] Basket basket)
    {
      Customer customer = basket.Customer;
      var order = new Order()
      {
        PizzaOrders = new List<PizzaOrder>()
      };
      customer.Order = order;

      foreach (int pizzaId in basket.Orders)
      {
        Pizza pizza = this.db.Pizzas.Single(p => p.Id == pizzaId);
        order.PizzaOrders.Add(new PizzaOrder { Pizza = pizza, Order = order });
      }

      order.TotalPrice = order.PizzaOrders.Sum(po => po.Pizza.Price);

      this.db.Customers.Add(customer);
      this.db.SaveChanges();
      return Ok();
    }
  }
}
```

Your `OrdersController` needs a `PizzaPlaceContextDb`, so you add a constructor taking the instance and you let dependency injection take care of the rest. To create a new order, you use the POST method for the `CreateOrder` method taking a `Basket` instance (which can be found in Listing 2-29) in the request body. Upon receipt of a basket instance, you create the customer and order. You then set the customer's order. There is no need to set the order's `Customer` property; Entity Framework Core will take care of the inverse relationship for you. Next, you fill up the order's `PizzaOrders` collection with pizzas. Then you calculate the total price for the order, and you save the whole Customer ➤ Order ➤ PizzaOrders ➤ Pizza chain by adding the root entity `Customer` to `PizzaPlaceDbContext` and calling `SaveChanges`. That's it. Entity Framework Core does all the work of storing the data!

Please note that this method returns with the Ok status code and not the Created status result. This is by design because you normally return the URI to retrieve the order from the service. This service does not support retrieval of an order, so we also don't return its URI.

Talking to the Order Microservice

Add a new class called `OrderService` to the `Services` folder of the `PizzaPlace.Client` project. This `OrderService` uses a POST request to the server, as shown in Listing 6-18.

Listing 6-18. The `OrderService` class

```
using Microsoft.AspNetCore.Components;
using PizzaPlace.Shared;
using System.Net.Http;
using System.Threading.Tasks;

namespace PizzaPlace.Client.Services
{
  public class OrderService : IOrderService
  {
    private readonly HttpClient httpClient;

    public OrderService(HttpClient httpClient)
      => this.httpClient = httpClient;

    public async Task PlaceOrder(Basket basket)
      => await this.httpClient.PostAsJsonAsync("/orders", basket);
  }
}
```

First, you add a constructor to the `OrderService` class, taking the `HttpClient` dependency, which you store in the `httpClient` field of the `OrderService` class. Next, you implement the `IOrderService` interface by adding the `PlaceOrder` method, taking a `Basket` as a parameter. Finally, you invoke the asynchronous `PostAsJsonAsync` method using the `await` keyword.

Now open the `Program` class from the `PizzaPlace.Client` project and replace the `ConsoleOrderService` class with your new `OrderService` class, as shown in Listing 6-19.

Listing 6-19. Configuring dependency injection to use the `OrderService` class

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;
using System.Threading.Tasks;

namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService, MenuService>();
      builder.Services.AddTransient<IOrderService, OrderService>();
      builder.Services.AddTransient(sp => new HttpClient { BaseAddress =
      new Uri(builder.HostEnvironment.BaseAddress) });
      builder.Services.AddSingleton<State>();
      await builder.Build().RunAsync();
    }
  }
}
```

Run your `PizzaPlace` application and place an order for a couple of pizzas. Now open SQL Server Object Explorer in Visual Studio (or Azure Data Studio) and examine the `Customers`, `Orders`, and `PizzaOrders` tables. They should contain your new order.

Summary

In this chapter, you learned that in Blazor you talk to the server using the `HttpClient` class, calling the `GetAsJsonAsync` and `PostAsJsonAsync` extension methods. You also learned that you should encapsulate calling the server using a client-side service class so you can easily change the implementation by switching the service type using dependency injection.

CHAPTER 7

Single-Page Applications and Routing

Blazor is a .NET framework you use for building single-page applications (SPA), just like you can use popular JavaScript frameworks such as Angular, React, and Vue.js. But what is a SPA? In this chapter, you will use routing to jump between different sections of a SPA and send data between different components.

What Is a Single-Page Application?

At the beginning of the Web, there were only static pages. A *static page* is an HTML file somewhere on the server that gets sent back to the browser upon request. Here the server is really nothing but a file server, returning HTML pages to the browser. The browser renders the HTML. The only interaction with the browser then was that you could click a link to get another page from the server. Later came the rise of dynamic pages. When a browser requests a *dynamic page*, the server runs a program to build the HTML in memory and sends the HTML back to the browser (this HTML never gets stored to disk; of course, the server can store the generated HTML in its cache for fast retrieval later). Dynamic pages are flexible in the way that the same code can generate thousands of different pages by retrieving data from a database and using it to construct the page. Lots of commercial websites like amazon.com use this. But there is still a usability problem. Every time your user clicks a link, the server must generate the next page from scratch and send it to the browser for rendering. This results in a noticeable wait period, and of course, the browser re-renders the whole page.

Then web pages started to use JavaScript to retrieve parts of the page when the user interacts with the UI. One of the first examples of this technique was Microsoft's Outlook Web Application. This web application looks and feels like Outlook, a desktop application, with support for all user interactions you expect from a desktop application.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_7

Google's Gmail is another example. They are now known as *single-page applications*. SPAs use certain sections of the web page that are replaced at runtime depending on the user's interaction. If you click an email, the main section of the page is replaced by the email's view. If you click your inbox, the main section gets replaced by a list of emails, and so on.

A SPA is a web application that replaces certain parts of the UI without reloading the complete page. SPAs use JavaScript (or C# when you're using Blazor) to implement this manipulation of the browser's control tree (also known as the DOM), and most of them consist of a fixed UI and a placeholder element where the contents are overwritten depending on where the user clicks. One of the main advantages of using a SPA is that you can make a SPA state-full. This means that you can keep information loaded by the application in memory. You will look at an example of a SPA, built with Blazor, in this chapter.

Using Layout Components

Let's start with the fixed part of a SPA. Every web application contains UI elements that you can find on every page, such as a header, footer, copyright, menu, and so on. Copy-pasting these elements to every page would be a lot of work and would require updating every page if one of these elements needed to change. Developers don't like to do that so every framework for building websites has had a solution for this. For example, ASP.NET WebForms uses master pages; ASP.NET MVC has layout pages. Blazor also has a mechanism for this called layout components.

Blazor Layout Components

Layout components are Blazor components. Anything you can do with a regular component, you can do with a layout component, like dependency injection, data binding, and nesting other components. The only difference is that they must inherit from the `LayoutComponentBase` class.

The `LayoutComponentBase` class defines a `Body` property as in Listing 7-1.

Listing 7-1. The LayoutComponentBase class

```
// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the Apache License, Version 2.0. See License.txt in the
project root for license information.

namespace Microsoft.AspNetCore.Components
{
  /// <summary>
  /// Optional base class for components that represent a layout.
  /// Alternatively, components may implement <see cref="IComponent"/>
      directly
  /// and declare their own parameter named <see cref="Body"/>.
  /// </summary>
  public abstract class LayoutComponentBase : ComponentBase
  {
    internal const string BodyPropertyName = nameof(Body);

    /// <summary>
    /// Gets the content to be rendered inside the layout.
    /// </summary>
    [Parameter]
    public RenderFragment Body { get; set; }
  }
}
```

As you can see from Listing 7-1, the LayoutComponentBase class inherits from the ComponentBase class. This is why you can do the same things as with normal components. Let's look at an example. Open the MyFirstBlazor solution from previous chapters. Now, look at the MainLayout.razor component in the MyFirstBlazor.Client's Shared folder, which you'll find in Listing 7-2.

Listing 7-2. MainLayout.razor

```
@inherits LayoutComponentBase

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

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

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

On the first line, the `MainLayout` component declares that it inherits from `LayoutComponentBase`. Then you see a `sidebar` and `main` `<div>` element, with the `main` element data-binding to the inherited `Body` property.

In Figure 7-1, you can see the `sidebar` on the left side (containing the links to the different components) and the `main` area on the right side with the `@Body` emphasized with a black rectangle (which I added to the figure). Clicking the `Home`, `Counter`, or `Fetch Data` link in the `sidebar` will replace the `Body` property with the selected component, updating the UI without reloading the whole page.

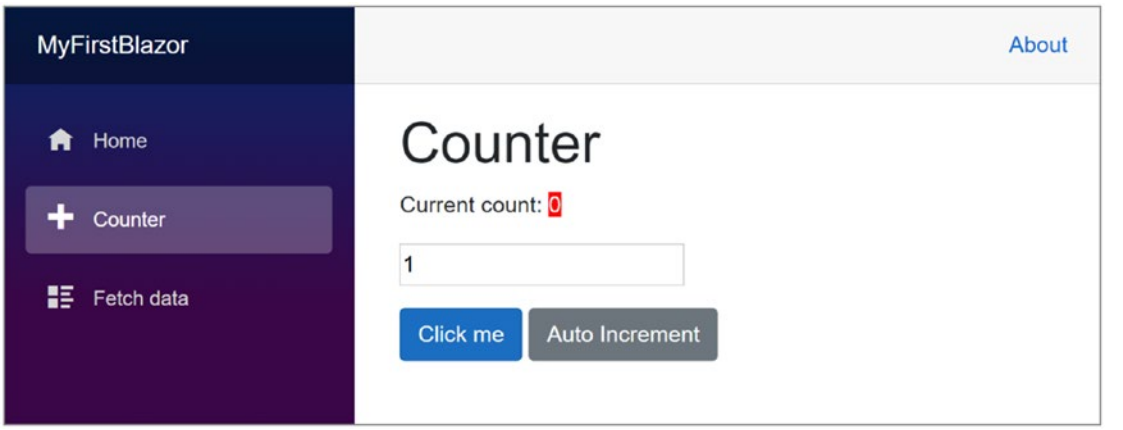

Figure 7-1. *The `MainLayout` component*

Selecting a @layout Component

Every component can select which layout to use by stating the name of the layout component with the `@layout` directive. For example, start by copying the `MainLayout.razor` file to `MainLayout2.razor`. This will generate a new layout component called `MainLayout2`, inferred from the file name. Change the `About` link's text to `Layout` as in Listing 7-3.

Listing 7-3. A second layout component

```
@inherits LayoutComponentBase

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

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

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

Now open the `Counter` component and add a `@layout` as in Listing 7-4.

Listing 7-4. Choosing a different layout with `@layout`

```
@page "/counter"
@layout MainLayout2

<h1>Counter</h1>

  ...
}
```

Run the application and watch the layout change (the text of the link in the top right corner) as you alternate between `Home` and `Counter`.

You can also use the `LayoutAttribute` if you're building your component completely in code.

So where is the default layout defined? Find and open `App.razor` from the client projects. As you can see in Listing 7-5, the `RouteView` defines the default layout.

Listing 7-5. Default layout pages are defined in App.razor

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

_Imports.razor

Most components will use the same layout. Instead of copying the same `@layout` directive to every page, you can also add a `_Imports.razor` file to the same folder as your components. Open the `Pages` folder from the `MyFirstBlazor.Client` project and add a new `_Imports.razor` file, which can be found in Listing 7-6.

Listing 7-6. `_Imports.razor`

```
@layout MainLayout2
```

Any component in this folder (or subfolder) that does not explicitly declare a `@layout` component will use the `MainLayout2` component.

Anything that is shared between all your components can be put in `_Imports.razor`, especially `@using` statements. A component can always override the `@layout` by explicitly adding the layout as in Listing 7-6.

Nested Layouts

Layout components can also be nested. You could define the `MainLayout` to contain all the UI that is shared between all components, and then define a nested layout to be used by a subset of these components. For example, add a new `Razor View` called `NestedLayout.razor` to the `Shared` folder and replace its contents with Listing 7-7.

Listing 7-7. A simple nested layout

```
@inherits LayoutComponentBase
@layout MainLayout

<div class="container-fluid">
  <div class="row bg-primary text-white">
    <div class="col-sm-12">
      <h2>This is a nested layout</h2>
    </div>
  </div>
  <div class="row">
    <div class="col-sm-12">
      @Body
    </div>
  </div>
</div>
```

To build a nested layout, you `@inherit` from `LayoutComponentBase` and set its `@layout` to another layout, for example, `MainLayout`. Now make the `Counter` component use this nested layout as in Listing 7-8.

Listing 7-8. The `Counter` component is using the nested layout

```
@page "/counter"
@layout NestedLayout

<h1>Counter</h1>

...
```

Run your application and select the `Counter` component, as shown in Figure 7-2.

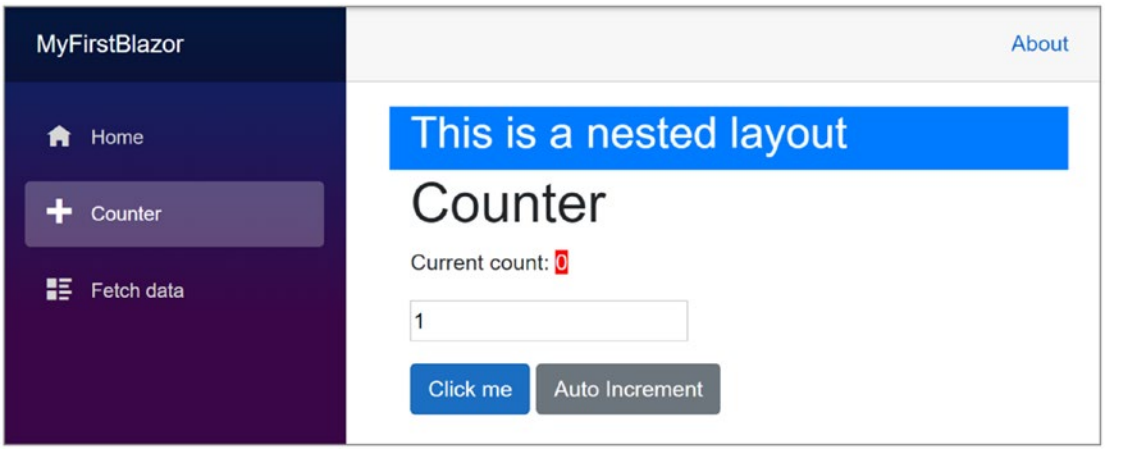

Figure 7-2. *The Counter component using the nested layout*

Understanding Routing

Single-page applications use routing to select which component gets picked to fill in the layout component's `Body` property. Routing is the process of matching the browser's URI to a collection of *route templates* and is used to select the component to be shown on screen. That is why every component in as Blazor SPA uses a `@page` directive to define the route template to tell the router which component to pick.

Installing the Router

When you create a Blazor solution from scratch, the router is already installed, but let's have a look at how this is done. Open `App.razor`. This `App` component only has one component, the `Router` component, as shown in Listing 7-9.

Listing 7-9. The App component containing the router

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

The Router component is a *templated component* with two templates. The Found template is used for known routes, and the NotFound is shown when the URI does not match any of the known routes. You can replace the contents of the last to show a nice error page to the user.

The Found template uses a RouteView component which will render the selected component with its layout (or default layout).

The router will look for all components that have the RouteAttribute (the @page directive gets compiled into a RouteAttribute) and pick the component that matches the current browser's URI. You will look at setting this RouteAttribute a little later in this chapter, but first, you need to look at the NavMenu component.

The NavMenu Component

Review the MainLayout component from Listing 7-2. On the fourth line, you will see the NavMenu component. This component contains the links to navigate between components. Open the MyFirstBlazor solution and look for the NavMenu component in the Shared folder, which is repeated in Listing 7-10.

Listing 7-10. The NavMenu component

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

```
    <span class="navbar-toggler-icon"></span>
  </button>
</div>

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

@code {
  private bool collapseNavMenu = true;

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

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

The first part of Listing 7-10 contains a toggle button which allows you to hide and show the navigation menu. This button is only visible on displays with a narrow width (e.g., mobile displays). If you want to look at it, run your application and make the browser width smaller until you see the *hamburger button* in the top right corner, as in Figure 7-3. Click the button to show the navigation menu and click it again to hide the menu again.

Figure 7-3. *Your application on a narrow display shows the toggle button*

The remaining markup contains the navigation menu, which consists of `NavLink` components. Let's look at the `NavLink` component.

The NavLink Component

The `NavLink` component is a specialized version of an anchor element `<a/>` used for creating navigation links. When the browser's URI matches the `href` property of the `NavLink`, it applies a CSS style (the `active` CSS class if you want to customize it) to itself to let you know it is the current route. For example, look at Listing 7-11.

Listing 7-11. The counter route's `NavLink`

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="counter">
    <span class="oi oi-plus" aria-hidden="true"></span> Counter
  </NavLink>
</li>
```

When the browser's URI ends with `/counter` (ignoring things like query strings), this `NavLink` will apply the `active` style. Let's look at another one in Listing 7-12.

Listing 7-12. The default route's `NavLink`

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
    <span class="oi oi-home" aria-hidden="true"></span> Home
  </NavLink>
</li>
```

When the browser's URI is empty (except for the site's URL), the `NavLink` from Listing 7-12 will be active. But here you have a special case. Normally, `NavLink` components only match the end of the URI. For example, `/counter/55` matches the `NavLink` from Listing 7-11. But with an empty URI, this would match everything! This is why in the special case of an empty URI you need to tell the `NavLink` to match the whole URI. You do this with the `Match` property, which by default is set to `NavLinkMatch.Prefix`. If you want to match the whole URI, use `NavLinkMatch.All` as in Listing 7-12.

Setting the Route Template

The `Routing` component from Blazor examines the browser's URI and searches for a component's route template to match. But how do you set a component's route template? Open the `counter` component shown in Listing 7-4. At the top of this file is the `@page "/counter"` directive. It defines the route template. A route template is a string matching a URI, and that can contain parameters, which you can then use in your component.

Using Route Parameters

You can change what gets displayed in the component by passing parameters in the route. You could pass the `id` of a product, look up the product's details with the `id`, and use it to display the product's details. Let's look at an example. Change the counter component to look like Listing 7-13 by adding another route template which will set the `CurrentCount` parameter.

Listing 7-13. Defining a route template with a parameter

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

@layout NestedLayout

<h1>Counter</h1>

<p>
  Current count:
  <span>@CurrentCount</span>
</p>
```

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

@code {
  [Parameter]
  public int CurrentCount { get; set; } = 0;

  private void IncrementCount()
  => CurrentCount += 1;
}
```

Listing 7-13 adds route template `@page "/counter/{CurrentCount:int}"`. This tells the router component to match a URI like `/counter/55` and to put the number in the `CurrentCount` parameter of your `Counter` component. You encase parameters in curly brackets. The `Router` component will put the value from the route in the property with the same name. You can also specify multiple parameters. Blazor does not allow you to specify default parameters, and therefore you need to specify two route templates. The first route template will pick the `Counter` component, with the `CurrentCount` set to its default value of `0`. The second route template will pick the `Counter` component and set the `CurrentCount` parameter to an `int` value. It must be `int` because of the `int` route constraint.

Filter URIs with Route Constraints

Just like routes in ASP.NET MVC Core, you can use *route constrains* to limit the type of parameter to match. For example, if you were to use the `/counter/Blazor` URI, the route template would not match because the parameter does not hold an integer value and the router would not find any component to match.

Constraints are even mandatory if you're not using string parameters; otherwise, the router does not cast the parameter to the proper type. You specify the constraint by appending it using a colon, for example, @page, `"/counter/CurrentCount:int"`.

A list of other constraints can be found in Table 7-1. Each of these maps to the corresponding .NET type.

Table 7-1. *Routing Constraints*

Route Constraints
Bool
Datetime
Decimal
Double
Float
Guid
Int
Long

If you are building your components as pure C# components, apply the `RouteAttribute` to your class with the route template as an argument. This is what the `@page` directive gets compiled into.

Redirecting to Other Pages

How do you navigate to another component using routing? You have three choices: use a standard anchor element, use the `NavLink` component, and use code. Let's start with the normal anchor tag.

Navigating Using an Anchor

Using an anchor (the `<a/>` element) is effortless if you use a relative `href`. For example, add Listing 7-14 below the button of Listing 7-13.

Listing 7-14. Navigation using an anchor tag

```
<a class="btn btn-primary" href="/">Home</a>
```

This link has been styled as a button using Bootstrap 4. Run your application and navigate to the `Counter` component. Click the Home button to navigate to the `Index` component whose route template matches `"/"`.

Navigating Using the NavLink Component

The `NavLink` component uses an underlying anchor, so its usage is similar. The only difference is that a `NavLink` component applies the `active` class when it matches the route. Generally, you only use a `NavLink` in the `NavMenu` component, but you are free to use it instead of anchors.

Navigating with Code

Navigating in code is also possible, but you will need an instance of the `NavigationManager` class through dependency injection. This instance allows you to examine the page's URI and has a helpful `NavigateTo` method. This method takes a string that will become the browser's new URI.

Let's try an example. Modify the `counter` component to look like Listing 7-15.

Listing 7-15. Using the `NavigationManager`

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

@layout NestedLayout

@inject NavigationManager uriHelper

<h1>Counter</h1>
```

```
<p>
  Current count:
  <span>@CurrentCount</span>
</p>

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

<a class="btn btn-primary" href="/">Home</a>

<button class="btn btn-primary" @onclick="StartFrom50">
  Start From 50
</button>

@code {
  [Parameter]
  public int CurrentCount { get; set; } = 0;

  private void IncrementCount()
  => CurrentCount += 1;

  private void StartFrom50()
  => uriHelper.NavigateTo("/Counter/50");
}
```

You tell dependency injection with the `@inject` directive to give you an instance of the `NavigationManager` and put it in the `uriHelper` field. Then you add a button that calls the `StartFrom50` method when clicked. This method uses the `NavigationManager` to navigate to another URI by calling the `NavigateTo` method. Run your application and click the "Start from 50" button. You should navigate to `/counter/50`.

Understanding the Base Tag

Please don't use absolute URIs when navigating. Why? Because when you deploy your application on the Internet, the base URI will change. Instead, Blazor uses the `<base/>` element and all relative URIs will be combined with this `<base/>` tag. Where is the `<base/>` tag? Open the `wwwroot` folder of your Blazor project and open `index.html`, shown in Listing 7-16.

Listing 7-16. Index.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>MyFirstBlazor</title>
  <base href="/" />
  <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
  <link href="css/site.css" rel="stylesheet" />
  <link href="_content/MyFirstBlazor.Components/styles.css"
  rel="stylesheet" />
</head>

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

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

</html>
```

If you are using Server-Side Blazor, the base tag can be found in _Host.cshtml.

When you deploy in production, all you need to do is to update the base tag. For example, you might deploy your application to https://online.u2u.be/selfassessment. In this case, you would update the base element to <base href="/selfassessment" />. So why do you need to do this? If you deploy to https://online.u2u.be/selfassessment, the counter component's URI becomes https://online.u2u.be/selfassessment/counter. Routing will ignore the base URI so it will match the counter as expected. You only need to specify the base URI once, as shown in Listing 7-16.

Sharing State Between Components

When you navigate between different Blazor components with routing, you will probably encounter the need to send information from one component to another. One way to accomplish this is by setting a parameter in the destination component by passing it in the URI. For example, you could navigate to `/pizzadetail/5` to tell the destination component to display information about the pizza with id 5. The destination component can then use a service to load the information about pizza #5 and then display this information. But in Blazor, there is another way. You can build a `State` class (most developers call this `State`, but this is just a convention and you can call it anything you want; `State` just makes sense) and then use dependency injection to give every component the same instance of this class. This is also known as the *Singleton Pattern*. This way, all your components share the same State instance, and this makes it easy to communicate between components that are not in a parent-child relationship. Your PizzaPlace application is already using a `State` class, so it should not be too much work to use this pattern.

Start by opening the Pizza Place solution from previous chapters. Open the `Index` component from the `Pages` folder (in the `PizzaPlace.Client` project) and look for the private `State` field. Remove this field (I've made it a comment) and replace it with an `@inject` directive as in Listing 7-17.

Listing 7-17. Using dependency injection to get the `State` Singleton instance

```
@page "/"

...

@inject IMenuService menuService
@inject IOrderService orderService
@inject State State

@code {

  // private State State { get; } = new State();

  protected override async Task OnInitializedAsync()
  {
    State.Menu = await menuService.GetMenu();
  }
  ...
}
```

Now configure dependency injection in `Program.cs` to inject the `State` instance as a singleton, as in Listing 7-18.

Listing 7-18. Configuring dependency injection for the `State` singleton

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using PizzaPlace.Client.Services;
using PizzaPlace.Shared;
using System.Threading.Tasks;

namespace PizzaPlace.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.RootComponents.Add<App>("app");
      builder.Services.AddTransient<IMenuService, MenuService>();
      builder.Services.AddTransient<IOrderService, OrderService>();
      builder.Services.AddTransient(sp => new HttpClient { BaseAddress =
      new Uri(builder.HostEnvironment.BaseAddress) });
      builder.Services.AddSingleton<State>();
      await builder.Build().RunAsync();
    }
  }
}
```

Run the application. Everything should still work! What you've done is to use the *Singleton Pattern* to inject the `State` singleton into the `Index` component. Let's add another component that will use the same `State` instance.

You want to display more information about a pizza using a new component, but before you do this, you need to update the `State` class. Add a new property called `CurrentPizza` to the `State` class, as shown in Listing 7-19.

Listing 7-19. Adding a `CurrentPizza` property to the `State` class

```
using System.Linq;

namespace PizzaPlace.Shared
{
  public class State
  {
    public Menu Menu { get; set; } = new Menu();

    public Basket Basket { get; set; } = new Basket();

    public UI UI { get; set; } = new UI();

    public decimal TotalPrice
      => Basket.Orders.Sum(id => Menu.GetPizza(id).Price);

    public Pizza CurrentPizza { get; set; }
  }
}
```

Now when someone clicks a pizza on the menu, it will display the pizza's information. Update the `PizzaItem` component by wrapping the pizza name in an anchor, like in Listing 7-20. Here we added a new `ShowPizzaInformation` parameter, and if this is non-null, we wrap it in an anchor which invokes the `ShowPizzaInformation` action.

Listing 7-20. Adding an anchor to display the pizza's information

```
<div class="row">
  <div class="col">
    @if (ShowPizzaInformation is object)
    {
      <a href=""
         @onclick="@(() => ShowPizzaInformation?.Invoke(Pizza))">
        @Pizza.Name
      </a>
    }
    else
```

```
    {
      @Pizza.Name
    }
  </div>
  <div class="col">
    @Pizza.Price
  </div>
  <div class="col">
    <img src="@SpicinessImage(Pizza.Spiciness)"
         alt="@Pizza.Spiciness" />
  </div>
  <div class="col">
    <button class="@ButtonClass"
            @onclick="@(async () => await Selected.InvokeAsync(Pizza))">
      @ButtonTitle
    </button>
  </div>
</div>

@code {
  [Parameter]
  public Pizza Pizza { get; set; }

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

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

  [Parameter]
  public EventCallback<Pizza> Selected { get; set; }

  private string SpicinessImage(Spiciness spiciness)
      => $"images/{spiciness.ToString().ToLower()}.png";

  [Parameter]
  public Action<Pizza> ShowPizzaInformation { get; set; }
}
```

When someone clicks this link, it should set the `State` instance's `CurrentPizza` property. But you don't have access to the `State` object. One way to solve this would be by injecting the `State` instance in the `PizzaItem` component. But you don't want to overburden this component, so you add a `ShowPizzaInformation` callback delegate to tell the containing `PizzaList` component that you want to display more information about the pizza. Clicking the pizza name link simply invokes this callback without knowing what should happen.

You are applying a pattern here known as "Dumb and Smart Components." A dumb component is a component that knows nothing about the global picture of the application. Because it doesn't know anything about the rest of the application, a dumb component is easier to reuse. A smart component knows about the other parts of the application (such as which service to use to talk to the database) and will use dumb components to display its information. In our example, the `PizzaList` and `PizzaItem` are dumb components because they receive all their data through data binding, while the `Index` component is a smart component which talks to services.

Update the `PizzaList` component to set the `PizzaItem` component's `ShowPizzaInformation` parameter as in Listing 7-21.

Listing 7-21. Adding a `PizzaInformation` callback to the `PizzaList` component

```
<h1>@Title</h1>

@if (Menu == null || Menu.Pizzas == null || Menu.Pizzas.Count == 0)
{
  <div style="height:20vh;" class="pt-3">
    <div class="mx-left pt-3" style="width:200px">
      <div class="progress">
        <div class="progress-bar bg-danger
                    progress-bar-striped
                    progress-bar-animated w-100"
             role="progressbar"
             aria-valuenow="100" aria-valuemin="0"
```

```
            aria-valuemax="100"></div>
      </div>
    </div>
  </div>
}
else
{
  @foreach (var pizza in Menu.Pizzas)
  {
    <PizzaItem Pizza="@pizza"
               ButtonClass="btn btn-success"
               ButtonTitle="Order"
               Selected="@Selected"
               ShowPizzaInformation="@ShowPizzaInformation"/>
  }
}

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

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

  [Parameter]
  public EventCallback<Pizza> Selected { get; set; }

  [Parameter]
  public Action<Pizza> ShowPizzaInformation { get; set; }

}
```

You added a `ShowPizzaInformation` callback to the `PizzaList` component, and you simply pass it to the `PizzaItem` component. The `Index` component will set this callback, and the `PizzaList` will pass it to the `PizzaItem` component.

Update the `Index` component to set the `State` instance's `CurrentPizza` and navigate to the `PizzaInfo` component, as shown in Listing 7-22.

Listing 7-22. The Index component navigates to the PizzaInfo component

```
@page "/"
<!-- Menu -->

<PizzaList Title="Our selection of pizzas"
           Menu="@State.Menu"
           Selected="@( async (pizza)
             => AddToBasket(pizza))"
           ShowPizzaInformation="ShowPizzaInformation"/>

<!-- End menu -->
<!-- Shopping Basket -->

<ShoppingBasket Title="Your current order"
                Basket="@State.Basket"
                GetPizzaFromId="@State.Menu.GetPizza"
                Selected="@( (pos) => RemoveFromBasket(pos))" />

<!-- End shopping basket -->
<!-- Customer entry -->

<CustomerEntry Title="Please enter your details below"
               ButtonTitle="Checkout"
               ButtonClass="btn btn-primary"
               @bind-Customer="@State.Basket.Customer"
               Submit="@PlaceOrder" />

<!-- End customer entry -->

<p>@State.ToJson()</p>

@inject IMenuService menuService
@inject IOrderService orderService
@inject State State
@inject NavigationManager uriHelper

@code {

    // private State State { get; } = new State();
```

```
  protected override async Task OnInitializedAsync()
  {
    State.Menu = await menuService.GetMenu();
  }

  private void AddToBasket(Pizza pizza)
  {
    Console.WriteLine($"Added pizza {pizza.Name}");
    State.Basket.Add(pizza.Id);
  }

  private void RemoveFromBasket(int pos)
  {
    Console.WriteLine($"Removing pizza at pos {pos}");
    State.Basket.RemoveAt(pos);
  }

  private async Task PlaceOrder()
  {
    await orderService.PlaceOrder(State.Basket);
  }

  private void ShowPizzaInformation(Pizza selected)
  {
    this.State.CurrentPizza = selected;
    Task.Run(() =>  this.uriHelper.NavigateTo("/pizzainfo"));
  }

}
```

The `Index` component tells the `PizzaList` component to call the `ShowPizzaInformation` method when someone clicks the information link from the `PizzaItem` component. The `ShowPizzaInformation` method then sets the `State`'s `CurrentPizza` property and navigates using the `NavigateTo` method to the `/PizzaInfo` route.

If you call `NavigateTo` as part of a callback, Blazor returns to the original route. That is why I use a background Task so Blazor will navigate after the callback…

Right-click the `Pages` folder and add a new `Razor Component` called `PizzaInfo`, as shown in Listing 7-23 (to save you some time and to keep things simple, you can copy

most of the PizzaItem component). The PizzaInfo component shows information about the State's CurrentPizza. This works because you share the same State instance between these components.

Listing 7-23. Adding a PizzaInfo component

```
@page "/PizzaInfo"

<h2>Pizza @State.CurrentPizza.Name Details</h2>

<div class="row">
  <div class="col">
    @State.CurrentPizza.Name
  </div>
  <div class="col">
    @State.CurrentPizza.Price
  </div>
  <div class="col">
    <img src="@SpicinessImage(State.CurrentPizza.Spiciness)"
         alt="@State.CurrentPizza.Spiciness" />
  </div>
  <div class="col">
    <a class="btn btn-primary" href="/">Menu</a>
  </div>
</div>

@inject State State

@code {
  private string SpicinessImage(Spiciness spiciness)
    => $"images/{spiciness.ToString().ToLower()}.png";
}
```

At the bottom of the markup, you add an anchor (and make it look like a button using bootstrap styling) to return to the menu. It's an example of changing the route with anchors. Of course, in a real-life application, you would show the ingredients of the pizza, a nice picture, and other information. I leave this as an exercise for you.

Summary

In this chapter, you looked at two things, layouts and routing.

Layouts allow you to avoid replicating markup in your application and help keep your application's look consistent. You also saw that layouts can be nested.

Routing is an important part of building single-page applications and takes care of picking the component to show based on the browser's URI. You define route templates using the `@page` syntax where you use route parameters and constraints. Navigation in your single-page application can be done using anchor tags and from code using the `NavigationManager` class. You then modified the PizzaPlace application to show how to share information between different routes in a Blazor application.

CHAPTER 8

JavaScript Interoperability

Sometimes there is just no escape from using JavaScript. For example, Blazor itself uses JavaScript to update the browser's DOM from your Blazor components. You can, too. In this chapter, you will look at interoperability with JavaScript and, as an example, you will build a Blazor component library to display a line chart using a popular open source JavaScript library for charts. This chapter does require you to have some basic JavaScript knowledge.

Calling JavaScript from C#

Browsers have a lot of capabilities you might want to use in your Blazor website. For example, you might want to use the Browser's *local storage* to keep track of some data. Thanks to Blazor's JavaScript interoperability, this is easy.

Providing a Glue Function

To call JavaScript functionality, you start by building a *glue function* in JavaScript. I like to call these functions glue functions (my own naming convention) because they become the glue between .NET and JavaScript.

Glue functions are regular JavaScript functions. A JavaScript glue function can take any number of arguments, on the condition that the arguments are JSON serializable (meaning that you can only use types that are convertible to JSON, including classes whose properties are JSON serializable). This is required because the arguments and return type are sent as JSON between .NET and JavaScript runtimes.

You then add this function to the *global scope* object, which in the browser is the `window` object. You will look at an example a little later, so keep reading. You can then call this JavaScript glue function from your Blazor component, as you will see in the next section.

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3_8

Using IJSRuntime to Call the Glue Function

Back to .NET land. To invoke your JavaScript glue function from C#, you use the .NET `IJSRuntime` instance provided through dependency injection. This instance has the `InvokeAsync<T>` generic method, which takes the name of the glue function and its arguments and returns a value of type `T`, which is the .NET return type of the glue function. If your JavaScript method returns nothing, there is also the `InvokeVoidAsync` method. If this sounds confusing, you will look at an example right away.

The `InvokeAsync` method is asynchronous to support all asynchronous scenarios, and this is the recommended way of calling JavaScript. If you need to call the glue function synchronously, you can downcast the `IJSRuntime` instance to `IJSInProcessRuntime` and call its synchronous `Invoke<T>` method. This method takes the same arguments as `InvokeAsync<T>` with the same constraints.

Using synchronous calls for JavaScript interop is not recommended! Server-side Blazor requires the use of asynchronous calls because the calls will be serialized over SignalR to the client.

Storing Data in the Browser with Interop

It's time to look at an example, and you will start with the JavaScript glue function. Open the `MyFirstBlazor` solution you used in previous chapters. Open the `wwwroot` folder from the `MyFirstBlazor.Client` project and add a new subfolder called `scripts`. Add a new JavaScript file to the `scripts` folder called `interop.js` and add the glue functions from Listing 8-1. These glue functions allow you to access the `localStorage` object from the browser, which allows you to store data on the client's computer so you can access it later, even after the user has restarted the browser.

Listing 8-1. The `getProperty` and `setProperty` glue functions

```
(function () {
  window.blazorLocalStorage = {
    get: key => key in localStorage ? JSON.parse(localStorage[key]) : null,
```

```
    set: (key, value) => { localStorage[key] = JSON.stringify(value); },
    delete: key => { delete localStorage[key]; },
  };
})();
```

Your Blazor website needs to include this script, so open the `index.html` file from the `wwwroot` folder and add a script reference after the Blazor script, as shown in Listing 8-2.

Listing 8-2. Including the script reference in your HTML page

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>MyFirstBlazor</title>
  <base href="/" />
  <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
  <link href="css/site.css" rel="stylesheet" />
  <link href="_content/MyFirstBlazor.Components/styles.css"
  rel="stylesheet" />
  <script src="scripts/interop.js"></script>
</head>

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

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

</html>
```

Now let's look at how to call these `set/get` glue functions. Open the `Index.razor` Blazor component and modify it to look like Listing 8-3. The `Counter` component now will use local storage to remember the last value of the counter. Even restarting your browser will not lose the value of the counter because local storage is permanent. To do this, you use a `Counter` property, which invokes your glue functions in the property setter to store the last value.

Listing 8-3. Invoking the glue functions from a Blazor component

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

@layout NestedLayout
@inject NavigationManager uriHelper
@inject IJSRuntime JSRuntime

<h1>Counter</h1>

<p>
  Current count:
  <span>@CurrentCount</span>
</p>

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

<a class="btn btn-primary" href="/">Home</a>

<button class="btn btn-primary" @onclick="StartFrom50">
  Start From 50
</button>

@code {

  protected override async Task OnInitializedAsync()
  {
    try
    {
```

```
      int? c = await JSRuntime
        .InvokeAsync<int?>("blazorLocalStorage.get", nameof(CurrentCount));
      if (c.HasValue)
      {
        currentCount = c.Value;
      }
    }
    catch { }
  }

  private int currentCount = 0;

  [Parameter]
  public int CurrentCount
  {
    get => currentCount;
    set
    {
      if (currentCount != value)
      {
        currentCount = value;
        JSRuntime.InvokeAsync<string>("blazorLocalStorage.set",
        nameof(CurrentCount), currentCount);
      }
    }
  }

  private void IncrementCount()
  => CurrentCount += 1;

  private void StartFrom50()
  => uriHelper.NavigateTo("/Counter/50");
}
```

The `Counter` component overrides the `OnInitializedAsync` to retrieve the last stored value from local storage using the `window.blazorLocalStorage.get` glue function. It is possible that there is no value yet, and that is why the `InvokeAsync` uses a nullable `int`.

Run the solution and modify the `Counter`'s value. Now when you refresh your browser, you will see the last value of `Counter`. The `Counter` is now persisted between sessions! You can exit your browser, open it again, and you will see the `Counter` again with the last value.

Passing a Reference to JavaScript

Sometimes your JavaScript needs to access one of your HTML elements. You can do this by storing the element in an `ElementReference` and then pass this `ElementReference` to the glue function.

Never use JavaScript interop to modify the DOM because this will interfere with the Blazor rendering process! If you need to modify the browser's DOM, use a Blazor component.

You should use this `ElementReference` as an opaque handle, meaning you can only pass it to a `JavaScript` glue function, which will receive it as a `JavaScript` reference to the element. You cannot even pass the `ElementReference` to another component.

Let's look at an example by setting the focus on an input element using interop. Start by adding a property of type `ElementReference` to the `@code` area in `Index.html` as in Listing 8-4.

Listing 8-4. Adding an `ElementRef` property

```
private ElementReference inputElement;
```

Then add an input element with a `@ref` attribute to set the `inputElement` field as in Listing 8-5.

Listing 8-5. Setting the `inputElement`

```
<p>
  Current count:
  <span>@CurrentCount</span>

  <input @ref="inputElement" @bind="CurrentCount"/>

</p>
```

Now add another JavaScript file setFocus.cs with the glue function from Listing 8-6. Don't forget to add the script reference to index.html.

Listing 8-6. Adding the setFocus glue function

```
(function () {
  window.blazorFocus = {
    set: (element) => { element.focus(); }
  };
})();
```

Now comes the "tricky" part. Blazor will create your component and then call the life cycle methods such as OnInitialized. If you invoke the setFocus glue function in OnInitialized, the DOM has not been updated with the input element so this will result in a runtime error because the glue function will receive a null reference. You need to wait for the DOM to be updated, which means that you should only pass the ElementReference to your glue function in the OnAfterRender/OnAfterRenderAsync method!

Override the OnAfterRenderAsync method as in Listing 8-7.

Listing 8-7. Passing the ElementReference in OnAfterRenderAsync

```
protected override async Task OnAfterRenderAsync(bool first)
{
  await JSRuntime.InvokeVoidAsync("blazorFocus.set", inputElement);
}
```

Run your solution and you should see that the input element receives focus automatically, as in Figure 8-1.

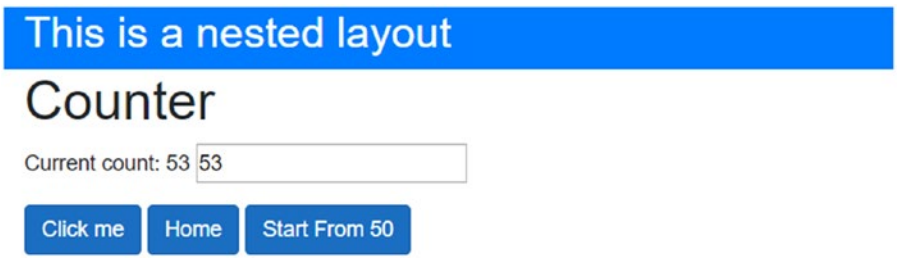

Figure 8-1. *The Counter input element receives focus automatically*

Calling .NET Methods from JavaScript

You can also call .NET methods from JavaScript. For example, your JavaScript might want to tell your component that something interesting has happened, like the user clicking something in the browser. Or your JavaScript might want to ask the Blazor component about some data it needs. You can call a .NET method, but with a couple of conditions. First, your .NET method's arguments and return value need to be JSON serializable, the method must be `public`, and you need to add the `JSInvokable` attribute to the method. The method can be a `static` or instance method.

To invoke a static method, you use the JavaScript `DotNet.invokeMethodAsync` or `DotNet.invokeMethod` function, passing the name of the assembly, the name of the method, and its arguments. To call an instance method, you pass the instance wrapped as a `DotNetObjectRef` to a JavaScript glue function, which can then invoke the .NET method using the `DotNetObjectRef`'s `invokeMethodAsync` or `invokeMethod` function, passing the name of the .NET method and its arguments.

Adding a Glue Function Taking a .NET Instance

Let's continue with the previous example. When you make a change to local storage, the storage triggers a JavaScript `storage` event, passing the old and new value (and more). This allows you to register for changes in other browser tabs or windows and use it to update the page with the latest data in `localStorage`.

Open `interop.js` from the previous example and add a `watch` function, as in Listing 8-8.

Listing 8-8. The `watch` function allows you to register for local storage changes

```
(function () {
  window.blazorLocalStorage = {
    get: key => key in localStorage ? JSON.parse(localStorage[key]) : null,
    set: (key, value) => { localStorage[key] = JSON.stringify(value); },
    delete: key => { delete localStorage[key]; },
    watch: async (instance) => {
```

```
      window.addEventListener('storage', (e) => {
        instance.invokeMethodAsync('UpdateCounter');
      });
    }
  };
})();
```

The watch function takes a reference to a DotNetObjectRef instance and invokes the UpdateCounter method when storage changes.

Adding a JSInvokable Method to Invoke

Open Counter.razor and add the UpdateCounter method to the @code area, as shown in Listing 8-9.

Listing 8-9. The UpdateCounter method

```
[JSInvokable]
public async Task UpdateCounter()
{
  Console.WriteLine("Update");
  int? c = await JSRuntime.InvokeAsync<int?>("blazorLocalStorage.get",
                                              nameof(CurrentCount));
  if (c.HasValue)
  {
    currentCount = c.Value;
  }
  this.StateHasChanged();
}
```

This method triggers the UI to update with the latest value of Counter. Please note that this method follows the .NET async pattern returning a Task instance. To complete the example, add the OnInitializedAsync life cycle method shown in Listing 8-10.

Listing 8-10. The OnInitialized method

```
protected override async Task OnInitializedAsync()
{
  var objRef = DotNetObjectReference.Create(this);
  await JSRuntime.InvokeVoidAsync("blazorLocalStorage.watch", objRef);

  int? c = await JSRuntime.InvokeAsync<int?>("blazorLocalStorage.get",
                                             nameof(CurrentCount));
  if (c.HasValue)
  {
    currentCount = c.Value;
  }
}
```

The OnInitialized method wraps the Counter component's this reference in a DotNetObjectRef and passes it to the blazorLocalStorage.watch glue function.

To see this in action, open two browser tabs on your website. When you change the value in one tab, you should see the other tab update to the same value automatically!

Using Services for Interop

The previous example is not the way I would recommend doing interop with JavaScript. There is a better way, and that is encapsulating the JSRuntime code in a service. This will hide all the dirty details of interacting with JavaScript and allow for easier maintenance.

Building the ILocalStorage Service

Start by adding a new interface to the Services folder of the MyFirstBlazor.Client project. Name it ILocalStorage and add the three methods from Listing 8-11 to it.

Listing 8-11. Building the ILocalStorage service interface

```
using System.Threading.Tasks;

namespace MyFirstBlazor.Client.Services
{
  public interface ILocalStorage
```

```
  {
    Task<T> GetProperty<T>(string propName);
    Task<object> SetProperty<T>(string propName, T value);
    Task<object> WatchAsync<T>(T instance) where T : class;
  }
}
```

These methods correspond with the glue functions from interop.js.

Now add a new class to the same `Services` folder and name it `LocalStorage`. This class should implement the `ILocalStorage` interface from Listing 8-12. See how this class hides away all the details of performing JavaScript interop? And this is a simple case!

Listing 8-12. Implementing the ILocalStorage interface

```
using Microsoft.JSInterop;
using System.Threading.Tasks;

namespace MyFirstBlazor.Client.Services
{

  public class LocalStorage : ILocalStorage
  {
    private readonly IJSRuntime jsRuntime;

    public LocalStorage(IJSRuntime jsRuntime)
    => this.jsRuntime = jsRuntime;

    public async Task<T> GetProperty<T>(string propName)
      => await this.jsRuntime
                   .InvokeAsync<T>("blazorLocalStorage.get", propName);

    public async Task<object> SetProperty<T>(string propName, T value)
      => await this.jsRuntime
                   .InvokeAsync<object>("blazorLocalStorage.set", propName,
                                        value);
```

```
    public async Task<object> WatchAsync<T>(T instance) where T : class
      => await this.jsRuntime
                   .InvokeAsync<object>("blazorLocalStorage.watch",
                      DotNetObjectReference.Create<T>(instance));
  }
}
```

Components will receive this service through dependency injection, so add it as a Singleton as in Listing 8-13.

Listing 8-13. Registering the LocalStorage service in dependency injection

```
using Microsoft.AspNetCore.Blazor.Hosting;
using Microsoft.Extensions.DependencyInjection;
using MyFirstBlazor.Client.Services;
using System.Threading.Tasks;

namespace MyFirstBlazor.Client
{
  public class Program
  {
    public static async Task Main(string[] args)
    {
      var builder = WebAssemblyHostBuilder.CreateDefault(args);
      builder.Services.AddSingleton<ILocalStorage, LocalStorage>();
      builder.RootComponents.Add<App>("app");
      await builder.Build().RunAsync();
    }
  }
}
```

The Counter with the LocalStorage Service

Go back to the `Counter` component and replace each call of `JSRuntime` using `blazorLocalStorage` with the `LocalStorage` service. Start by adding the inject directive for the `ILocalStorage` service as in Listing 8-14.

Listing 8-14. Injecting the ILocalStorage service into the Counter component

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

@layout NestedLayout
@inject NavigationManager uriHelper
@inject IJSRuntime JSRuntime

@using MyFirstBlazor.Client.Services
@inject ILocalStorage LocalStorage
```

Now onto the `OnInitialiazedAsync` method, where we retrieve the value from local storage. Replace the `JSRuntime` calls with `LocalStorage` calls, as in Listing 8-15.

Listing 8-15. Implementing OnInitializedAsync

```
protected override async Task OnInitializedAsync()
{
  await LocalStorage.WatchAsync(this);
  int? c = await LocalStorage.GetProperty<int?>(nameof(CurrentCount));
  if (c.HasValue)
  {
    currentCount = c.Value;
  }
}
```

Do the same for the `UpdateCounter` method from Listing 8-16.

Listing 8-16. The UpdateCounter method using the LocalStorage service

```
[JSInvokable]
public async Task UpdateCounter()
{
  int? c = await LocalStorage.GetProperty<int?>(nameof(CurrentCount));
  if (c.HasValue)
  {
    currentCount = c.Value;
  }
  this.StateHasChanged();
}
```

And finally, the setter for the `CurrentCount` property as in Listing 8-17.

Listing 8-17. Remembering the Counter's value

```
[Parameter]
public int CurrentCount
{
  get => currentCount;
  set
  {
    if (currentCount != value)
    {
      currentCount = value;
      LocalStorage.SetProperty<int>(nameof(CurrentCount), currentCount);
    }
  }
}
```

This was not so hard, was it?

Building a Blazor Chart Component Library

In this section, you will build a *Blazor component library* to display charts by using a popular open source JavaScript library called Chart.js (`www.chartjs.org`). However, wrapping the whole library would make this chapter way too long, so you'll just build a simple line-chart component.

Creating the Blazor Component Library

Open Visual Studio and start by creating a new Blazor Server App project called `ChartTestProject`. This project will only be used for testing the chart component. Now right-click the solution and add a new `Razor Class Library` called `U2U.Components.Chart`.

If you are using Visual Studio Code, open a command prompt and type

```
dotnet new blazorserver -o ChartTestProject
To create the component library, type
```

```
dotnet new razorclasslib -o U2U.Components.Chart
```

Add the component library to the solution:

```
dotnet sln add U2U.Components.Chart/U2U.Components.Chart.csproj
```

Adding the Component Library to Your Project

Now you have the Blazor component library project. Let's use it in the test project. Look for `Component1.razor` in the `U2U.Components.Chart` project and rename it to `LineChart.razor`. Add a reference to the component library in the client project. In Visual Studio, right-click the `ChartTestProject` and select Add ➤ Reference. Check the `U2U.Components.Chart` project, shown in Figure 8-2, and click `OK`.

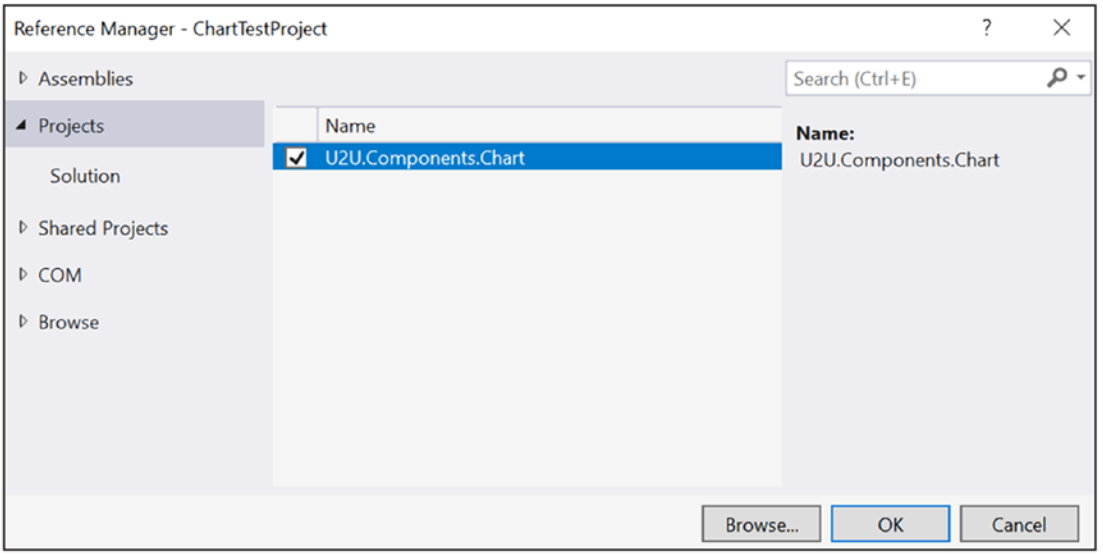

Figure 8-2. *Adding a reference to the component library*

With Visual Studio Code, use the integrated terminal, change the current directory to `ChartTestProject.Client`, and type this command:

```
dotnet add reference ../U2U.Components.Chart/U2U.Components.Chart.csproj
```

This will add a reference to the `U2U.Components.Chart` component library.

Look for the `_Imports.razor` file (the one next to `App.razor`) in the `ChartTestProject` and open it in the editor.

Add a `@using` to the component library's namespace as in Listing 8-18.

Listing 8-18. Adding the component library namespace to the Blazor project

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

@using U2U.Components.Chart
```

Open the `Index.razor` file from the `Pages` folder and add the `LineChart` component shown in Listing 8-19.

Listing 8-19. Adding the `LineChart` component

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<LineChart/>
```

Build and run this project. You should see the new component as in Figure 8-3, but the styling is wrong!

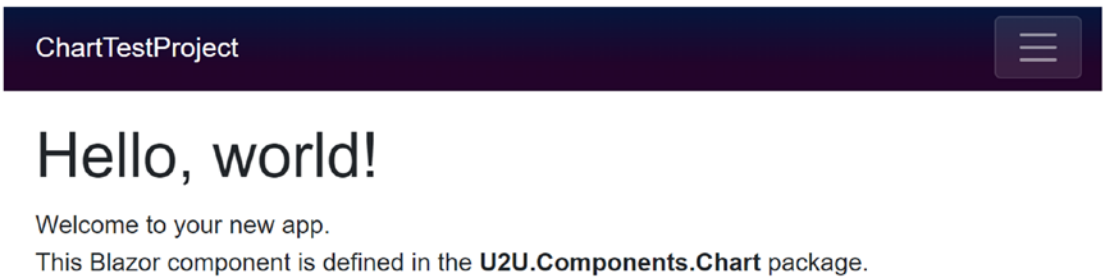

Figure 8-3. *Using the component library, but styling is missing*

Let's fix the styling. In the ChartTestProject, look for _Host.cshtml. This time we are using Server-Side Blazor, which means that your Blazor App component is rendered by the ASP.NET Core runtime. There is no Index.html! Instead, add it here in the <head> section, as in Listing 8-20. You should only have to add a single <link> element to the bottom of the <head> section.

Listing 8-20. Add the style to the _Host file

```
@page "/"
@namespace ChartTestProject.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
  Layout = null;
}

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0" />
  <title>ChartTestProject</title>
  <base href="~/" />
  <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
  <link href="css/site.css"
            rel="stylesheet" />
  <!-- Add the style from the component library here -->
  <link href="_content/U2U.Components.Chart/styles.css"
            rel="stylesheet" />
</head>
<body>
  <app>
    <component type="typeof(App)" render-mode="ServerPrerendered" />
  </app>

  <div id="blazor-error-ui">
    <environment include="Staging,Production">
```

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

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

Build and run your application. It should now look like Figure 8-4.

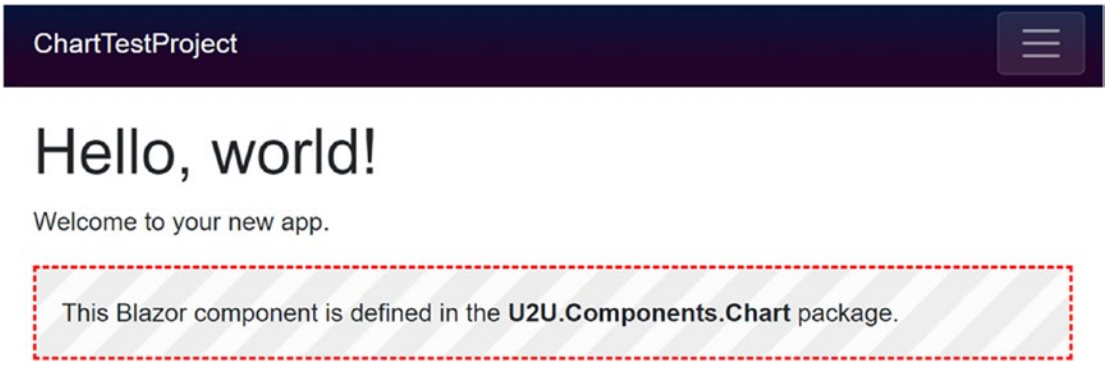

Figure 8-4. *Testing if the component library has been added correctly*

Adding Chart.js to the Component Library

The `LineChart` component doesn't look like a chart, so it's time to fix this! First, you need to add the `Chart.js` JavaScript library to the component library project. Go to `www.chartjs.org/`. This is the main page for `Chart.js`. Now click the GitHub button, shown in Figure 8-5, to open the project's GitHub page.

Figure 8-5. *The Chart.js main page*

Look for the GitHub releases link (or just enter this URL: `https://github.com/chartjs/Chart.js/releases/tag/v2.9.3`). The release page will open, as shown in Figure 8-6.

Since it takes some time between writing a book and you reading it, there is a big chance that the version number will have incremented. Make sure you select a version starting with 2 since version 3 will contain breaking changes.

Figure 8-6. *GitHub releases page for Chart.js*

Click `Chart.bundle.min.js` to download it.

After it has been downloaded, copy this file to the `wwwroot` folder of the `U2U.Components.Chart` project, as shown in Figure 8-7.

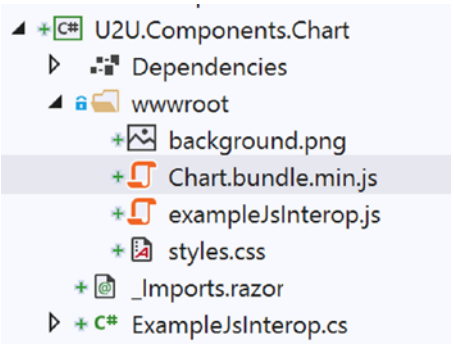

Figure 8-7. *Copying Chart.bundle.min.js into the wwwroot folder*

You will also need to add this as a `<script>` tag to `_Host.cshtml` (or `index.html` with `Blazor WebAssembly`) as in Listing 8-21. You only have to add one `<script>` element to the bottom of the `<body>` section.

Listing 8-21. Add the Chart.js script to the host page

```
@page "/"

...

  <script src="_framework/blazor.server.js"></script>
  <!-- Add the script file(s) from the component library here -->
  <script src="_content/U2U.Components.Chart/Chart.bundle.min.js"></script>
</body>
</html>
```

Verifying If the JavaScript Library Loaded Correctly

Do you know about Murphy's law? It states, "Anything that can possibly go wrong, does." Let's make sure that the `Chart.js` library gets loaded by the browser. Run your Blazor project and open the browser's debugger. Check if `Chart.bundle.min.js` has been loaded correctly. The easiest way to do this is to see if the `window.Chart` has been set (`Chart.js` adds one constructor function called `Chart` to the `window` global object). You can do this from the `Console` tab of the debugger by typing `window.Chart`, as shown in Figure 8-8.

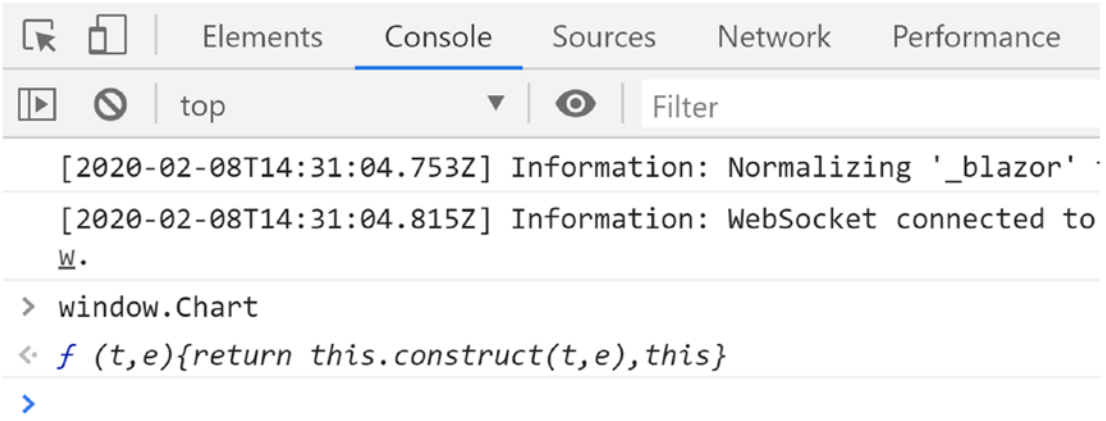

Figure 8-8. *Using the browser's console to check the value of window.Chart*

If this returns `undefined`, rebuild the `U2U.Components.Chart` project. Then you can try refreshing the browser after emptying the browser's cache. When the browser's debugger is shown, right-click the refresh button and you'll get a drop-down menu, as shown in Figure 8-9. Select the Empty Cache and Hard Reload menu item.

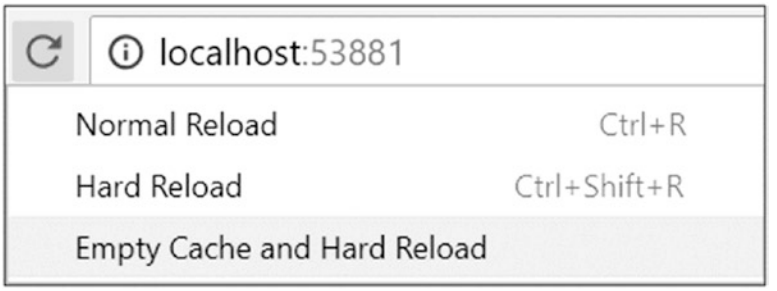

Figure 8-9. *Reloading the page after clearing the cache*

Adding Chart.js Data and Options Classes

Open your browser and type in `www.chartjs.org/docs/latest/`. Here you can see a sample of using `Chart.js` in JavaScript. This library requires two data structures to be passed to it: one containing the chart data and one containing the options. This section will add these classes to the Blazor component library, but now using C#. Again, I am not going for full coverage of all the features of `Chart.js` to keep things crisp.

The ChartOptions Class

Let's start with the options class. Right-click the `U2U.Components.Chart` library and add a new class called `ChartOptions` as in Listing 8-22.

This is a fair amount of code. You might consider copying it from the sources provided with this book. I've also left out comments describing each property for conciseness.

Listing 8-22. The `ChartOptions` class

```
namespace U2U.Components.Chart
{
  public class ChartOptions
  {
    public class TitleOptions
    {
```

```
  public static readonly TitleOptions Default
  = new TitleOptions();

  public bool Display { get; set; } = false;
}

public class ScalesOptions
{
  public static readonly ScalesOptions Default
  = new ScalesOptions();

  public class ScaleOptions
  {
    public static readonly ScaleOptions Default
    = new ScaleOptions();

    public class TickOptions
    {
      public static readonly TickOptions Default
      = new TickOptions();

      public bool BeginAtZero { get; set; } = true;


      public int Max { get; set; } = 100;
    }

    public bool Display { get; set; } = true;

    public TickOptions Ticks { get; set; }
    = TickOptions.Default;
  }

  public ScaleOptions[] YAxes { get; set; }
  = new ScaleOptions[] { ScaleOptions.Default };
}

public static readonly ChartOptions Default
= new ChartOptions { };
```

```
    public TitleOptions Title { get; set; }
    = TitleOptions.Default;

    public bool Responsive { get; set; } = true;

    public bool MaintainAspectRatio { get; set; } = true;

    public ScalesOptions Scales { get; set; }
    = ScalesOptions.Default;
  }
}
```

This C# class, with nested classes, reflects the JavaScript options object (partially) from `Chart.js`. You don't have to bind to all the things in the library, just the bits you need. Note that I've added `Default` static properties to each class to make it easier for developers to construct the options hierarchy.

The LineChartData Class

`Chart.js` expects you to give it the data it will render. For this, it needs to know a couple of things, like the color of the line, the color of the fill beneath the line, and, of course, the numbers to plot the graph. So how will you represent colors and points in your Blazor component? As it turns out, there are classes in .NET to represent colors and points: `System.Drawing.Color` and `System.Drawing.Point`. Unfortunately, you cannot use `Color` because it doesn't convert into a JavaScript color, but you can allow users to use it in their code. I'll discuss how to do this a little later.

Right-click the `U2U.Components.Chart` project and select Manage NuGet Packages…. Select the Browse tab and look for the `System.Drawing.Common` package as in Figure 8-10. Install this package to add support for `System.Drawing` classes.

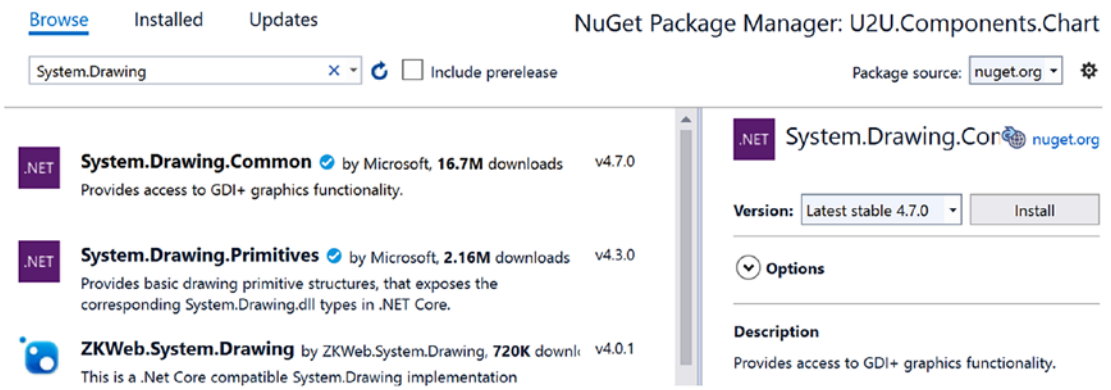

Figure 8-10. *Add the System.Drawing.Common package*

Add a new class `LineChartData` to the component library, as shown in Listing 8-23.

Listing 8-23. The `LineChartData` class

```
using System;
using System.Collections.Generic;
using System.Drawing;

namespace U2U.Components.Chart
{
  public class LineChartData
  {
    public class DataSet
    {
      public string Label { get; set; }

      public List<Point> Data { get; set; } = null;

      public string BackgroundColor { get; set; }

      public string BorderColor { get; set; }

      public int BorderWidth { get; set; } = 2;
    }
```

```
    public string[] Labels { get; set; }
    = Array.Empty<string>();

    public DataSet[] Datasets { get; set; }
  }
}
```

Most of this class should be clear, except maybe for `Array.Empty<string>()`. This method returns an empty array of the generic argument. But why is this better? You cannot modify an empty array, so you can use the same instance everywhere (this is also known as the *Flyweight Pattern*). This is like `string.Empty`, and using it puts less strain on the garbage collector.

Registering the JavaScript Glue Function

To invoke the `Chart.js` library, you need to add a little JavaScript of your own. Open the `wwwroot` folder of the component library project and start by renaming the `exampleJsInterop.js` file to `JsInterop.js` and replacing the code with Listing 8-24.

Listing 8-24. Registering the `JavaScript` glue class

```
window.components = (function () {
  return {
    chart: function (id, data, options) {
      var context = document.getElementById(id)
        .getContext('2d');
      var chart = new Chart(context, {
        type: 'line',
        data: data,
        options: options
      });
    }
  };
})();
```

This adds a `window.components.chart` glue function that when invoked calls the `Chart` function (from `Chart.js`), passing in the graphics `context` for the canvas, `data`, and `options`. It is very important that you pass the `id` of the canvas because someone might want to use the `LineChart` component several times on the same page. By using a unique id for each `LineChart` component, you end up with canvasses with unique ids.

Add a `<script>` element to the `_Host.cshtml` (or `index.html` with Blazor WebAssembly) after the `chartjs` script as in Listing 8-25.

If your Blazor project needs to reference a static resource from a component library, always use a relative URL that starts with `_content/<<LibraryName>>`. This way the runtime knows it needs to read this from the component library.

Listing 8-25. Add the JsInterop.js script to the host page

```
...
  <script src="_framework/blazor.server.js"></script>
  <!-- Add the script file(s) from the component library here -->
  <script src="_content/U2U.Components.Chart/Chart.bundle.min.js"></script>
  <script src="_content/U2U.Components.Chart/JsInterop.js"></script>
</body>
</html>
```

Providing the JavaScript Interoperability Service

Your `LineChart` component will need to call the `Chart.js` library using your `window.components.chart` glue function. But putting all this logic in the `LineChart` component directly is something you want to avoid. Instead, you will build a service encapsulating this logic and inject the service into the `LineChart` component. Should the Blazor team at Microsoft decide to change the way JavaScript interoperability works (they have done that before), then you will only need to change one class (again, the *single responsibility principle*). Start by adding a new interface to the `U2U.Component.Chart` library project called `IChartInterop` with the code from Listing 8-26.

Listing 8-26. The `IChartInterop` interface

```
namespace U2U.Components.Chart
{
  public interface IChartInterop
  {
    void CreateLineChart(string id, LineChartData data,
                                ChartOptions options);
  }
}
```

As you can see, this interface's `CreateLineChart` method closely matches the `window.components.chart` glue function. Let's implement this service. Add a new class called `ChartInterop` to the component library project and implement it as in Listing 8-27.

Listing 8-27. Implementing the `ChartInterop` class

```
using Microsoft.JSInterop;

namespace U2U.Components.Chart
{
  /// <summary>
  /// It is always a good idea to hide specific implementation
  /// details behind a service class
  /// </summary>
  public class ChartInterop : IChartInterop
  {
    public IJSRuntime JSRuntime { get; }

    public ChartInterop(IJSRuntime jsRuntime)
      => JSRuntime = jsRuntime;

    public void CreateLineChart(string id, LineChartData data,
                                ChartOptions options)
      => JSRuntime.InvokeAsync<string>("components.chart",
                                    id, data, options);
  }
}
```

This `CreateLineChart` method invokes the JavaScript `components.chart` function you added in Listing 8-24.

Time to configure dependency injection. You could ask the user of the library to add the `IChartInterop` dependency directly, but you don't want to put too much responsibility in the user's hands. Instead, you will provide the user with a handy *C# extension method* that hides all the gory details from the user. Add the new class called `DependencyInjection` to the component library project with the code from Listing 8-28.

Listing 8-28. The `AddCharts` extension method

```
using Microsoft.Extensions.DependencyInjection;

namespace U2U.Components.Chart
{
  public static class DependencyInjection
  {
    public static IServiceCollection AddCharts(
      this IServiceCollection services)
    => services.AddTransient<IChartInterop, ChartInterop>();
  }
}
```

This class provides you with the `AddCharts` extension method that the user of the `LineChart` component can now add to the client project. Let's do this. Make sure everything builds first, and then open `Startup.cs` (since we are using a Blazor Server project, dependency injection is configured in the `Startup` class's `ConfigureServices` method) in the `ChartTestProject` and add a call to `AddCharts` as in Listing 8-29.

Listing 8-29. Convenient dependency injection with `AddCharts`

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

The user of the component does not need to know any implementation details to use the LineChart component. Mission accomplished!

Implementing the LineChart Component

Now you are ready to implement the LineChart component. Chart.js does all its drawings using an HTML5 canvas element, and this will be the markup of the LineChart component. Update LineChart.razor to match Listing 8-30.

Listing 8-30. The LineChart component

```
@inject IChartInterop JsInterop

<canvas id="@Id" class="@Class">
</canvas>

@code {

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

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

  [Parameter]
  public LineChartData Data { get; set; }

  [Parameter]
  public ChartOptions Options { get; set; } = ChartOptions.Default;

  protected override void OnAfterRender(bool firstRender)
  {
    string id = Id;
    JsInterop.CreateLineChart(Id, Data, Options);
  }
}
```

The LineChart component has a couple of parameters. The Id parameter is used to give each LineChart's canvas a unique identifier; this way you can use LineChart several times on the same page. The Class parameter can be used to give the canvas one or

more CSS classes to add some style (and you can never have enough style). Finally, the `Data` and `Options` parameters get passed to JavaScript to configure the chart.

Now comes the tricky part (this is like the earlier section where you wanted to set the focus on the input). To call the JavaScript `chart` function, the `canvas` needs to be in the browser's DOM. When does that happen? Blazor creates the component hierarchy, calls each component's `OnInitialized`, `OnInitializedsAsync`, `OnParametersSet`, and `OnParametersSetAsync` methods, and then uses the component hierarchy to build its internal tree, which then is used to update the browser's DOM. Then Blazor calls each component's `OnAfterRender` method. Because the `canvas` element should already be part of the DOM, you need to wait for the `OnAfterRender` method before calling `JsInterop.CreateLineChart`.

Using the LineChart Component

With everything in place, you can now complete the `LineChart` component from the `Index` page in your `ChartTestProject`. Update the `Index.razor` file to match Listing 8-31. You will add the `toJS()` extension method later, so ignore any errors till then.

Listing 8-31. Completing the `Index` component

```
@page "/"

@using U2U.Components.Chart
@using System.Drawing

<h1>Hello, world!</h1>

Welcome to your new app.

<LineChart Id="test" Class="linechart" Data="@Data" Options="@Options" />

@code {

  private LineChartData Data { get; set; }

  private ChartOptions Options { get; set; }

  protected override void OnInitialized()
```

```
  {
    this.Options = ChartOptions.Default;

    this.Data = new LineChartData
    {
      Labels = new string[] { "", "A", "B", "C" },
      Datasets = new LineChartData.DataSet[]
      {
        new LineChartData.DataSet
        {
          Label = "Test",
          BackgroundColor = Color.Transparent.ToJs(),
          BorderColor = Color.FromArgb(10, 96, 157, 219)
                             .ToJs(),
          BorderWidth = 5,
          Data = new List<Point>
          {
            new Point(0, 0),
            new Point(1, 11),
            new Point(2, 76),
            new Point(3, 13)
          }
        }
      }
    };
  }
}
```

You start by adding two `@using` directives for the `U2U.Components.Chart` and `System.Drawing` namespaces. Then you add the `Id`, `Class`, `Data`, and `Options` parameters. You give these parameters values in the `OnInitialized` method (should you get this data asynchronously from the server, you would use the `OnInitializedAsync` method). One more thing before you can build and run the project and admire your work: add a new class called `ColorExtensions` to the `U2U.Component.Chart` project. Implement it as shown in Listing 8-32.

Listing 8-32. The `ColorExtensions` class with the `toJS` extension method

```
using System.Drawing;

namespace U2U.Components.Chart
{
  public static class ColorExtensions
  {
    public static string ToJs(this Color c)
    => $"rgba({c.R}, {c.G}, {c.B}, {c.A})";
  }
}
```

Build and run your project. If all is well, you should see Figure 8-11.

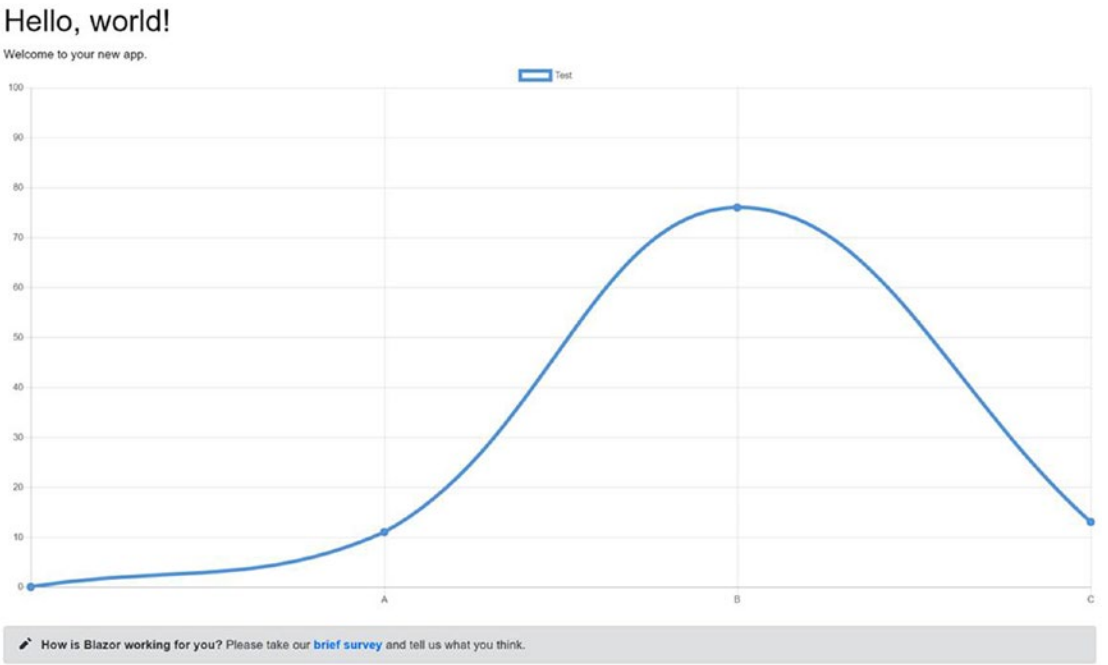

Figure 8-11. *The finished chart example*

Summary

In this chapter, you saw how you can call JavaScript from your Blazor components using the `IJSRuntime.InvokeAsync<T>` method. This requires you to register a JavaScript glue function by adding this function to the browser's window global object.

You can also call your .NET static or instance method from JavaScript. Start by adding the `JSInvokable` attribute to the .NET method. If the method is static, you use the JavaScript `DotNet.invokeMethodAsync` function (or `DotNet.invokeMethod` if the call is synchronous), passing the name of the assembly, the name of the method, and its arguments. If the method is an instance method, you pass the .NET instance wrapped in a `DotNetObjectRef` to the glue function, which can then use the `InvokeMethodAsync` function to call the method, passing the name of the method and its arguments.

Finally, you applied this knowledge by wrapping the `Chart.js` open source library to draw a nice line chart. You built *a Blazor component library*, added some classes to pass the data to the `Chart` function, and then used a glue function to draw the chart.

Index

A

B

C

P. Himschoot, *Microsoft Blazor*, https://doi.org/10.1007/978-1-4842-5928-3

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z