Blazor Revealed

Building Web Applications in .NET

Peter Himschoot

Apress®

Blazor Revealed

Building Web Applications in .NET

Peter Himschoot

Apress®

Blazor Revealed: Building Web Applications in .NET

Peter Himschoot
Melle, Belgium

ISBN-13 (pbk): 978-1-4842-4342-8 ISBN-13 (electronic): 978-1-4842-4343-5
https://doi.org/10.1007/978-1-4842-4343-5

Library of Congress Control Number: 2019932722

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

Cover designed by eStudioCalamar

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 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 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/9781484243428. For more detailed information, please visit 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 is also a Microsoft Regional Director, a group of trusted advisors to the developer and IT professional audiences, and to Microsoft.

About the Technical Reviewer

Gerald Versluis is a developer and Microsoft MVP from Holland with years of experience working with Xamarin, Azure, ASP.NET, and other .NET technologies. He has been involved in numerous projects, in various roles. A great number of his projects are Xamarin apps. Not only does Gerald like to code, but he is keen on spreading his knowledge as well as gaining some in the bargain. He speaks, provides training sessions, and writes blogs and articles in his spare time.

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 Training, 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.

Introduction to WebAssembly and Blazor

I was attending the *Microsoft Most Valued Professional and Regional Directors Summit* when 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! Blazor is a framework that allows you to build single-page applications (SPAs) using C# and allows you to run any standard .NET library in the browser. Before Blazor, your options for building a SPA were JavaScript or one of the other higher-level languages like TypeScript, which get compiled into JavaScript anyway. In this introduction, I will look at how browsers are now capable of running .NET assemblies in the browser using WebAssembly, Mono, and Blazor.

Blazor is, at the time of writing, an EXPERIMENTAL framework. I hope by the time you are reading this book that it has been made official by Microsoft.

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 this very well and are bidding for you to use their browser. In the beginning of mainstream Internet, everyone was using *Netscape*. Microsoft wanted a share of the market, so in 1995 it 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 marque 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, and Opera. HTML5 not only defines a series of standard HTML elements but also rules on how they should render, making it a lot easier to build a web site that looks the same in all modern browsers.

But let's go back to 1995, when *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, thus avoiding full page reloads and allowing for client-side web applications (applications written in JavaScript that run completely in the browser). One of the first companies to apply Ajax was Microsoft, when it built *Outlook Web Access (OWA)*. OWA is a web application almost identical to the Outlook desktop application but providing 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 would display the map like any other image by downloading a bitmap from a server.

Building an Ajax web site was a major undertaking, which only big companies like Microsoft and Google could afford. This soon changed with the introduction of JavaScript libraries like jQuery and knockout.js. Today we can build rich web apps with Angular, React, and Vue.js. All of them use JavaScript or higher-level languages like TypeScript, which get complied into JavaScript. Which brings us back to JavaScript and the second browser war. JavaScript performance is paramount in modern browsers. Chrome, Edge, Firefox, and Safari 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 Just-in-Time (JIT) compilation where JavaScript gets converted into native code, as illustrated by Figure 1.

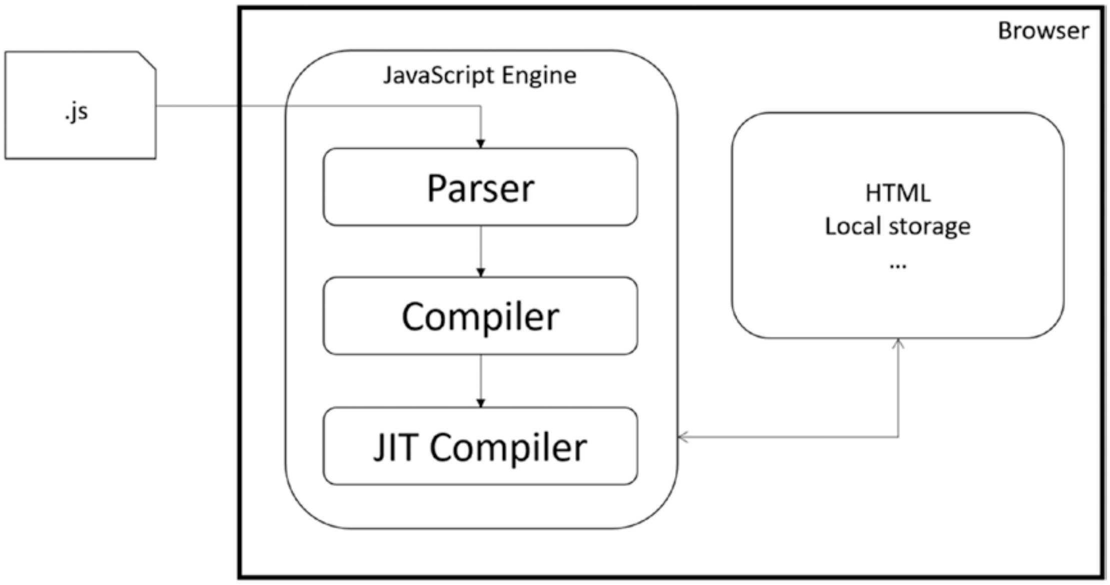


Figure 1. *The JavaScript execution process*

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 JIT 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 JIT compiled into native code, as shown in Figure 2. Open your browser and google "*webassembly demo zen garden.*" One of the links is `https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html` where you can see an impressive ray-trace demo of a Japanese Zen garden, shown in Figure 3.

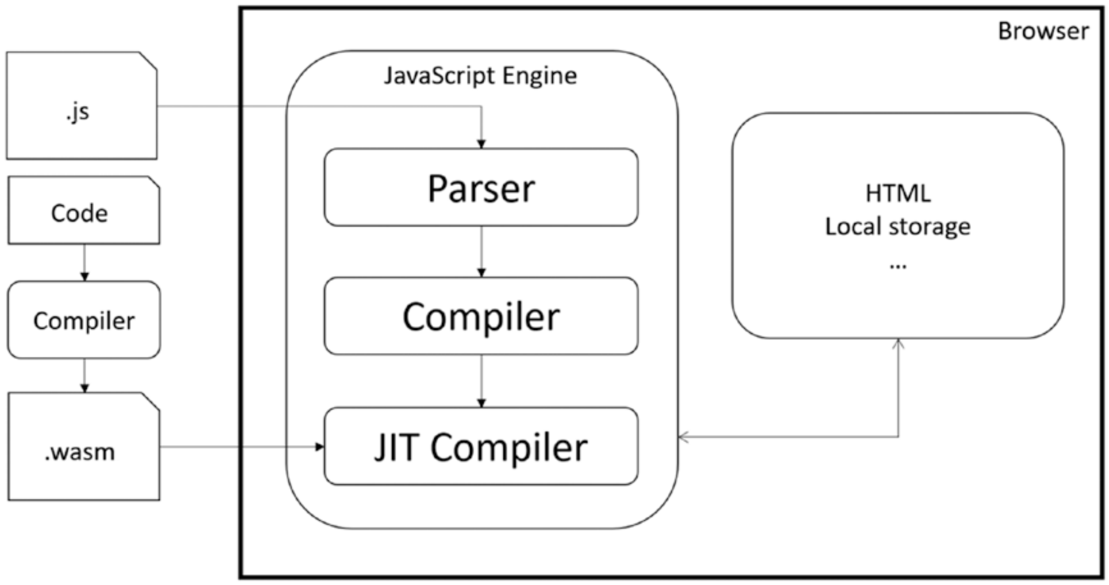


Figure 2. *The WebAssembly execution process*

Figure 3. *Japanese Zen Garden*

From the official site, `www.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 is a new binary format optimized for browser execution; it is NOT JavaScript. There are compilers for languages like C++ and Rust that compile to WASM.

Which Browsers Support WebAssembly?

WebAssembly is supported by all major browsers: Chrome, Edge, Safari, and Firefox, including their mobile versions. As WebAssembly becomes more and more important, we will see other modern browsers follow suit, but don't expect Internet Explorer to support 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 the Windows, Android, and iOS mobile operating systems. Mono also allows you to run .NET on Linux (its original purpose) and is written in C++. This last part is important because you 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, which is not so practical for day-to-day development. The other approach takes the Mono runtime, 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, the *Ooui* project 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. These tools are not yet available, but they are in the pipeline.

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) that execute inside the 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. See Listing 1, which you can find when you create a new Blazor project.

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>

@functions {
    int currentCount = 0;

    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.

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.

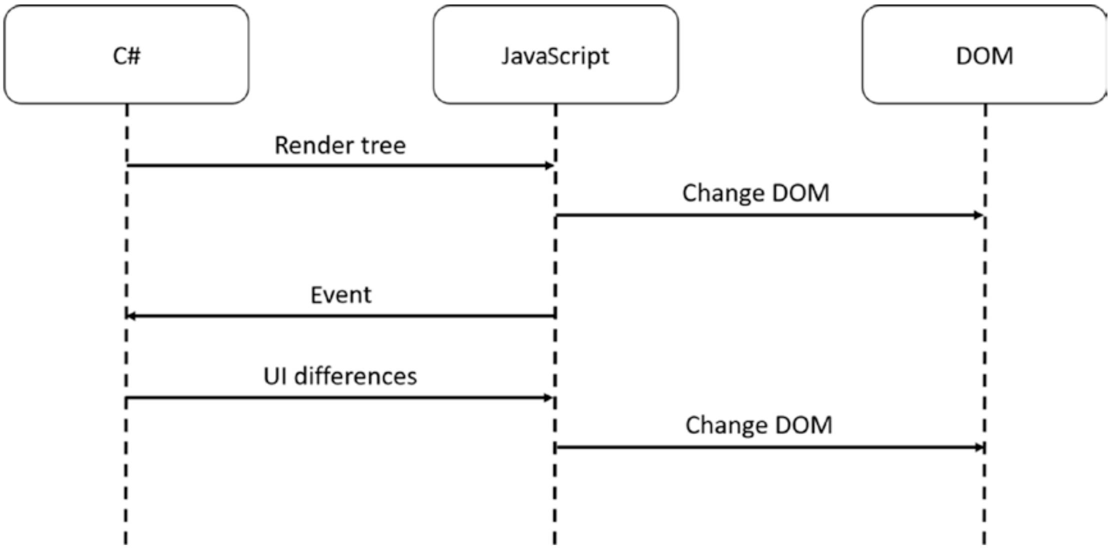


Figure 4. *The Blazor DOM generation process*

Server-Side Blazor

On August 7, 2018, Daniel Roth introduced a new execution model for Blazor called server-side Blazor at the *ASP.NET community standup*. In this model, your Blazor site runs on the server, resulting in a much smaller download for the browser.

The Server-Side Model

You just saw that client-side Blazor 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 gets 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, which is 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. You can see 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!

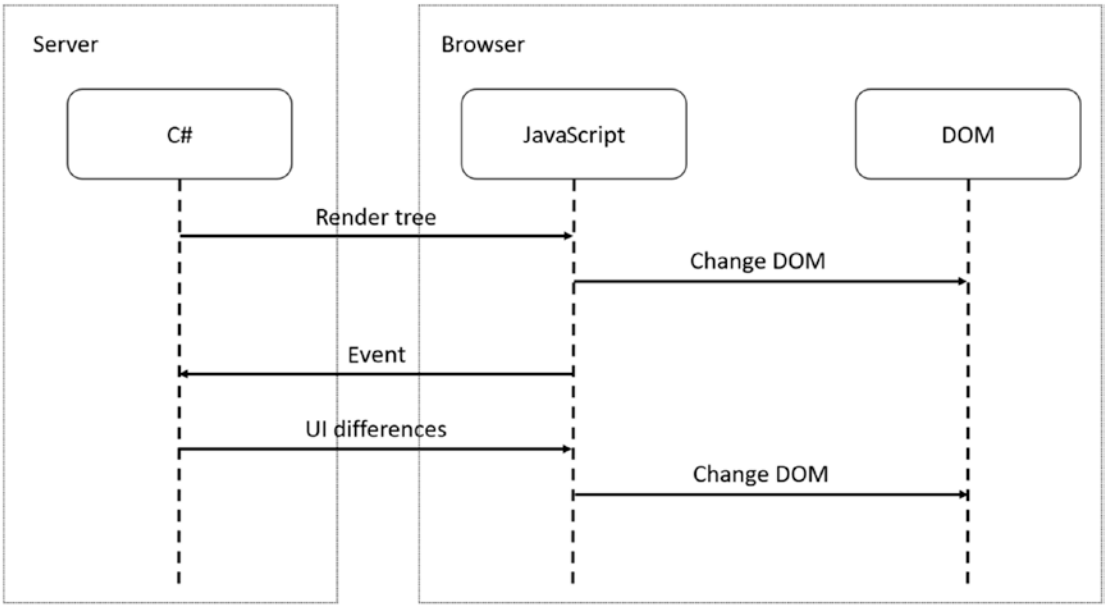


Figure 5. *Server-side Blazor*

Pros and Cons of the Server-Side Model

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

Smaller Downloads

With server-side Blazor, your application does not need to download `mono.wasm` nor all your .NET assemblies. This means that the application will start a lot faster.

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 MVC applications, for example accessing the database directly. Note that doing this will stop you from being able to quickly convert it to 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; you also can't 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.

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. Also, Blazor uses a state-full model, which means you must keep track of every user's state on the server.

Summary

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

CHAPTER 1

Your First Blazor Project

Getting a hands-on experience is the best way to learn. In this chapter, you'll install the prerequisites to developing with Blazor, which includes 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 inspect the different aspects of the project to get a "lay of the land" view for how Blazor applications are developed.

Installing Blazor Prerequisites

Working with Blazor requires you to install some prerequisites, so let's get to it.

.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 OSX.

You can find the installation files at `www.microsoft.com/net/download`. Look for the latest version of the .NET Core SDK. Download the installer, 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 have the correct version installed. The version number should be at least 2.1.300.

Should the command's output show an older version (for example 2.1.200), you must download and install a more recent version of .NET Core SDK.

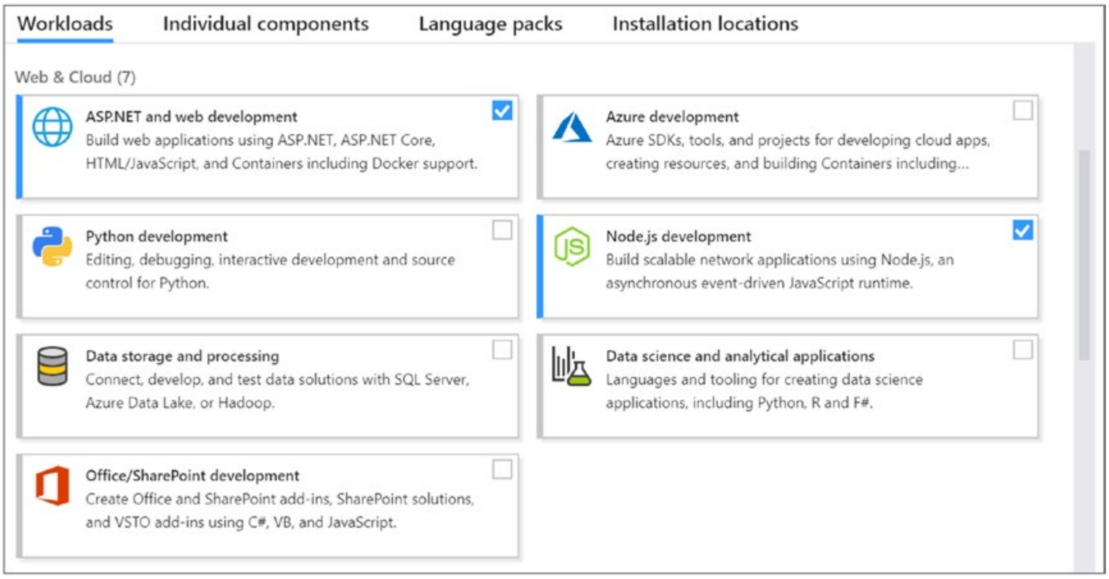
P. Himschoot, *Blazor Revealed*, https://doi.org/10.1007/978-1-4842-4343-5_1

Visual Studio 2017

Visual Studio 2017 (from now on I will refer to Visual Studio as VS) is one of the integrated development environments (IDEs) you will use throughout this book. The other IDE is Visual Studio Code. 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 on Visual Studio Code.

Download the latest version of Visual Studio 2017 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.



ASP.NET and web development ✓
Build web applications using ASP.NET, ASP.NET Core, HTML/JavaScript, and Containers including Docker support.

Azure development
Azure SDKs, tools, and projects for developing cloud apps, creating resources, and building Containers including...

Python development
Editing, debugging, interactive development and source control for Python.

Node.js development ✓
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

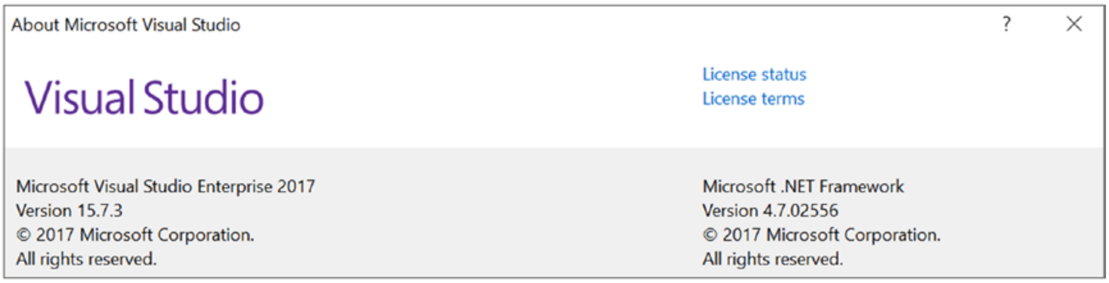
Data storage and processing
Connect, develop, and test data solutions with SQL Server, Azure Data Lake, or Hadoop.

Data science and analytical applications
Languages and tooling for creating data science applications, including Python, R and F#.

Office/SharePoint development
Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins using C#, VB, and JavaScript.

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 15.7.3, as illustrated in Figure 1-2.


About Microsoft Visual Studio

Visual Studio

License status
License terms

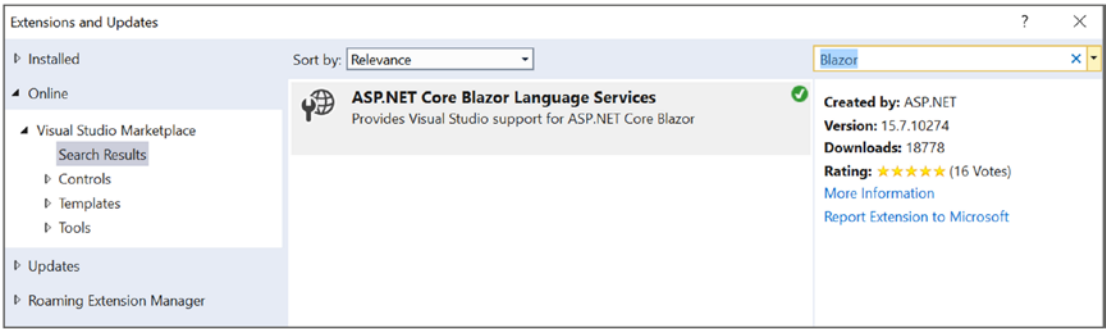
Microsoft Visual Studio Enterprise 2017
Version 15.7.3
© 2017 Microsoft Corporation.
All rights reserved.

Microsoft .NET Framework
Version 4.7.02556
© 2017 Microsoft Corporation.
All rights reserved.

Figure 1-2. *About Microsoft Visual Studio*

ASP.NET Core Blazor Language Services

The Blazor Language Services plugin for Visual Studio will aid you when typing Blazor files and will install the Blazor VS project templates. Installation of the plugin is done directly from Visual Studio. Open Tools ➤ Extensions and Updates. Click the Online tab and enter Blazor in the search box. You should see the ASP.NET Core Blazor Language Services listed as shown in Figure 1-3. Select it and click the Download button to install.



ASP.NET Core Blazor Language Services
Provides Visual Studio support for ASP.NET Core Blazor

Created by: ASP.NET
Version: 15.7.10274
Downloads: 18778
Rating: ★★★★★ (16 Votes)

Figure 1-3. *Installing Blazor Language Services from the Extensions and Updates menu*

Visual Studio Code

Visual Studio Code is a free, modern, cross-platform development environment with 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 Visual Studio 2017 (because you're running a non-Windows operating system or you don't want to use it), use Code.

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

After installation I do advise you install a couple of extensions for Code, especially the C# extensions. Start Code, and on the left side, select the Extensions tab, as shown in Figure 1-4.

Figure 1-4. *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.

Another extension you want to search for is Razor+, as shown in Figure 1-5. This extension will give you nice syntax coloring for the kind of Razor files you will use in Blazor.

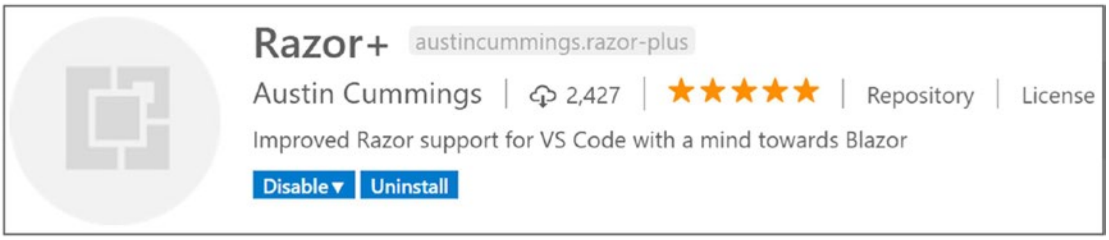

Razor+
austincummings.razor-plus
Austin Cummings | 2,427 | ★★★★★ | Repository | License
Improved Razor support for VS Code with a mind towards Blazor
Disable ▾
Uninstall

Figure 1-5. *Razor+ for Visual Studio Code*

Installing the Blazor Templates for VS/Code

Throughout this book you 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.Blazor.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:

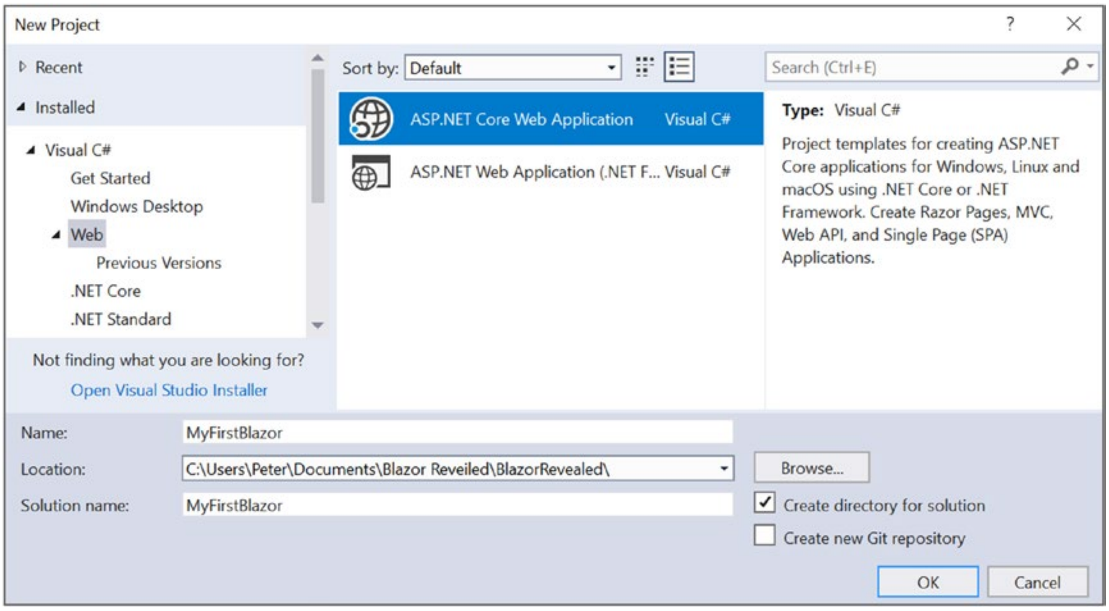
```
Blazor (hosted in ASP.NET server)          blazorhosted
Blazor Library                             blazorlib
Blazor (Server-side in ASP.NET Core)       blazorserverside
Blazor (standalone)                        blazor
```

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 `blazor` template) that has no need for server-side code. This kind of project 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 (using the `blazorhosted` template) with client, server, and shared code. This kind of project will require you to host it where there is .NET core 2.1 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 `blazorserverside` template). 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, you will use the second option, but the concepts you will learn in this book are the same for all three options.

Creating a Project with Visual Studio

For your first project, start Visual Studio and select File ➤ New ➤ Project. On the left side of the New Project dialog, select C# ➤ Web, and then select ASP.NET Core Web Application, as illustrated by Figure 1-6.



Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: MyFirstBlazor

Location: C:\Users\Peter\Documents\Blazor Reveiled\BlazorRevealed\

Solution name: MyFirstBlazor

☑ Create directory for solution

☐ Create new Git repository

Figure 1-6. *Visual Studio New Project dialog*

Name your project *MyFirstBlazor*, leave the rest to the preset defaults, and click OK. On the next screen, you can select what kind of ASP.NET Core project you want to generate. From the top drop-downs, select .NET Core and ASP.NET Core 2.1 (or higher), as shown in Figure 1-7. Then select Blazor (ASP.NET hosted) and click OK.