ASP.NET Core Essentials

Develop and deploy modern cross-platform web applications with ASP.NET Core

Shahed Chowdhuri

ASP.NET Core Essentials

Table of Contents

ASP.NET Core Essentials

ASP.NET Core Essentials

Credits

Author	Copy Editor
Shahed Chowdhuri	Safis Editing
Reviewer	**Project Coordinator**
Steve Albers	Ulhas Kambali
Commissioning Editor	**Proofreader**
Dipika Gaonkar	Safis Editing
Acquisition Editor	**Indexer**
Sonali Vernekar	Rekha Nair
Content Development Editor	**Graphics**
Prashanth G	Kirk D'Penha
Technical Editor	**Production Coordinator**
Sushant S Nadkar	Melwyn Dsa

About the Author

Shahed Chowdhuri has over 18 years of experience in the field of professional software development, and is currently a Senior Technical Evangelist at Microsoft Corporation. He began his career with what is now known as Classic ASP, and worked with ASP.NET as it continued to evolve over the years. He is a public speaker in the DC metro area and along the East Coast of the United States. He serves as a mentor for startups and indie developers at 1776 DC, and also manages multiple developer groups on Facebook and Meetup. You can find him blogging at WakeUpAndCode.com or on Twitter via @shahedC.

About the Reviewer

Steve Albers is a software developer and speaker living in Northern Virginia.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Preface

Web applications have flourished in enterprise application development, even as mobile applications continue to rise in popularity. Moreover, the use of web APIs provides greater flexibility when it comes to building a backend that can serve both web apps and mobile apps. With ASP.NET Core 1.0, you get the benefits of a mature programming language such as C# with the performance of an all-new .NET Core that allows your web apps and web APIs to run on Windows, Mac, and Linux.

ASP.NET Core lets you choose your operating system during development, whether you're writing your code on a Surface Book, MacBook Air, or on a popular distribution of Linux. While this book primarily covers development on Visual Studio 2015 on a Windows system, you will find links to online guides if you wish to set up your environment on a different operating system.

ASP.NET Core 1.0 Essentials will introduce you to Microsoft's latest revision of ASP.NET, with everything you need to get started today. If you have already worked with ASP.NET MVC before, you will get a refresher of a few things you already know, followed by what's new. If you've only worked with ASP.NET Web Forms or other competing web frameworks, you will find the initial chapters very useful in laying the groundwork for what you need to know.

What this book covers

Chapter 1, *Getting Started with ASP.NET Core*, teaches you about ASP.NET Core 1.0, including MVC and web API. This chapter will explain the differences between the full .NET Framework 4.6 and .NET Core, while introducing Visual Studio 2015 and the all-new cross-platform Visual Studio Code.

Chapter 2, *Building Your First ASP.NET Core Application*, shows you how to create a new ASP.NET Core web application. This chapter will dissect the parts of a modern ASP.NET web application, while explaining what's new and what's changed.

Chapter 3, *Understanding MVC*, teaches you all about ASP.NET Core MVC by going deeper into controllers, views, and models. This chapter will explain how to create all the parts of a modern MVC application and then bring it all together.

Chapter 4, *Using Web API to Extend Your Application*, is about using the web API to extend your web application to support web and mobile applications. This chapter will explain how to create routes and configure a web API application and then consume it from a client application.

Chapter 5, *Interacting with Web API using JavaScript*, teaches you how to interact with the ASP.NET web API using straight JavaScript as well as popular frameworks such as AngularJS and KnockoutJS. This chapter will also cover developer tools, which automate important tasks and help during the development process.

Chapter 6, *Using Entity Framework to Interact with Your Database in Code*, teaches you how to use Entity Framework in the data layer of your web application. This chapter will introduce object-relational mapping (ORM) tools and the use of EF Code First Migrations .

Chapter 7, *Dependency Injection and Unit Testing for Robust Web Apps*, shows you how to implement dependency injection by using Inversion of Control . This chapter will cover DI, IoC containers, and Microsoft's new built-in support for dependency injection. The chapter will also cover the basics of unit testing.

Chapter 8, *Authentication, Authorization, and Security*, is about enabling authentication in ASP.NET web applications while implementing authorization for specific application features. The chapter will also cover guidelines for protecting user data and strategies to combat common security vulnerabilities in web applications.

Chapter 9, *Deploying Your Application*, covers how to deploy your applications to IIS on a web server or on Microsoft's Azure cloud platform. The chapter will also cover the use of continuous integration to run unit tests and deploy successfully compiled applications.

What you need for this book

To learn ASP.NET Core and run the code examples for this book, the following software is recommended:

- Windows 7 or higher (Windows 10 recommended)
- Visual Studio 2015 Update 3 or higher (the free Community Edition is OK)
- Any modern web browser, such as Microsoft Edge, Google Chrome, or Mozilla Firefox
- Optional: Visual Studio Code (Windows, Mac, Linux)

For more information on developing for and running ASP.NET Core on systems other than Windows, please consult the *Running ASP.NET Core on Windows, Mac OS X, Linux* section in Chapter 1, *Getting Started with ASP.NET Core*. There, you will find a high-level overview of instructions for Mac and Linux, with links to online guides that are frequently updated.

Who this book is for

This book is for software developers who have experience in .NET, preferably with C# or some other object-oriented programming language, which is required in order to build ASP.NET Core web applications. A basic understanding of web application development is also essential.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Uses the HTTP GET method with optional querystring parameters"

A block of code is set as follows:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

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

```
dotnet restore
dotnet build
dotnet ef migrations add Initial
dotnet ef database update
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Enter some values and click the **Create** button."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/ASPdotNET-Core-Essentials. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us
at questions@packtpub.com, and we will do our best to address the problem.

Chapter 1. Getting Started with ASP.NET Core

Active Server Pages was first made available on Microsoft's IIS web server in the mid-1990s. Fast-forward about 20 years, and **ASP.NET Core** (that is, ASP.NET 5 while in development) is now open source and runs on Windows, Linux, and OS X. Moreover, **Visual Studio** is now available as a cross-platform lightweight code editor in addition to the full-featured IDE on Windows.

As a **.NET** developer, you can take advantage of all the recent improvements, while building on the experience that you already have. The **.NET Framework** has evolved beyond the full framework and is now available as a cross-platform runtime called **.NET Core**.

In this chapter, we will discuss the following principles and concepts to get you started with ASP.NET Core:

- .NET architecture
- ASP.NET unified programming model
- New Visual Studio IDEs
- Cross-platform runtime

ASP.NET Core - Unifying MVC, Web API, and Web Pages

When developers hear the term *ASP.NET*, some associate it with ASP.NET **Web Forms**, while others think of **ASP.NET MVC** or even **Web API**. Many developers have started their journey with Web Forms, and have migrated to **Model-View-Controller** (**MVC**) in recent years.

The adoption of ASP.NET MVC has been on the rise for years, due to its cleaner code and testability. While you can still develop Web Forms on .NET 4.x, you will be using MVC on .NET Core 1.0. Fortunately, ASP.NET Core runs on both .NET 4.5.1+ and .NET Core, so you can choose which runtime you would like to target.

Going forward, Microsoft has unified MVC, Web API, and Web Pages in ASP.NET Core. What does this mean for you, the developer? It means that you will enjoy more consistency when building your application with MVC, Web API, and Web Pages. In the past, MVC and Web API were implemented separately, leading to duplication and inconsistencies, but with ASP.NET Core, they have been merged into a single programming model.

High-level overview

If you're new to any of the preceding terms, here's a high-level overview:

- **ASP.NET MVC:** Microsoft's implementation of the MVC architectural pattern, used for building test-friendly web applications that are robust yet lightweight.
- **Web API:** Microsoft's answer to RESTful APIs, Web API allows developers to build HTTP services that can serve as a backend to web applications and mobile apps.
- **Web Pages:** Microsoft's solution for creating lightweight dynamic websites. At the time of writing, Web Pages is not available in ASP .NET Core 1.0 so it will not be covered in this book.

The following figure illustrates a typical ASP.NET web application that uses both MVC and Web API. Note that Web API can be used to serve clients other than web browsers, such as mobile apps. It's important to mention that browsers here can include mobile web browsers as well:
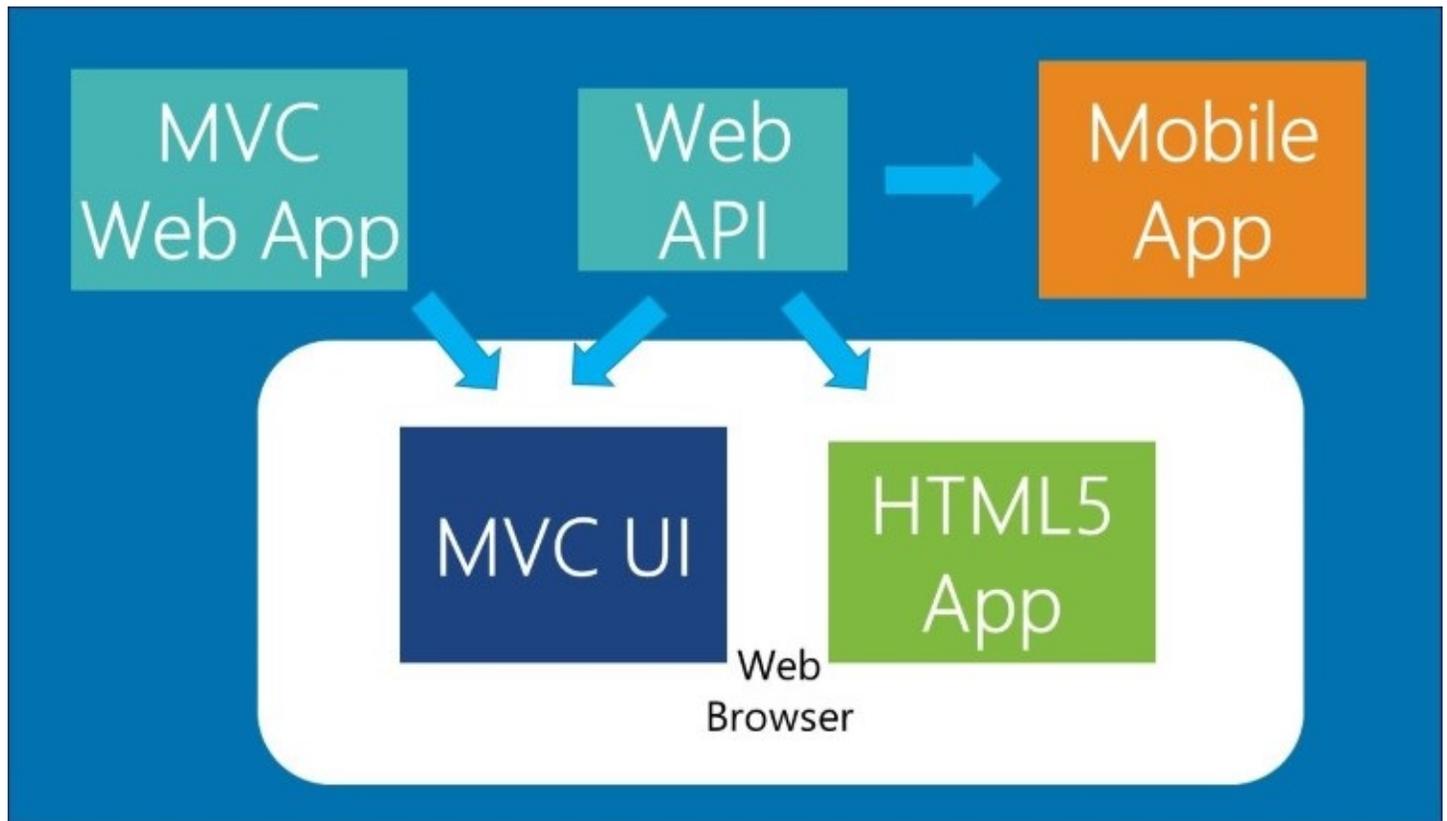
Version numbers

You may have noticed that I mentioned MVC 6 while talking about ASP.NET Core. ASP.NET MVC 6 has been renamed ASP .NET Core 1.0 MVC. To keep track of all the new version numbers, the following is a handy list of what you need to know:

- C# 6.0
- ASP.NET Core:
  - Runs on .NET Core 1.0 or .NET Framework 4.5.1+

- .NET Framework 4.6.2 (at the time of writing)
- .NET Core 1.0
- ASP.NET Core 1.0 MVC
- SignalR 3 (not released at the time of writing)
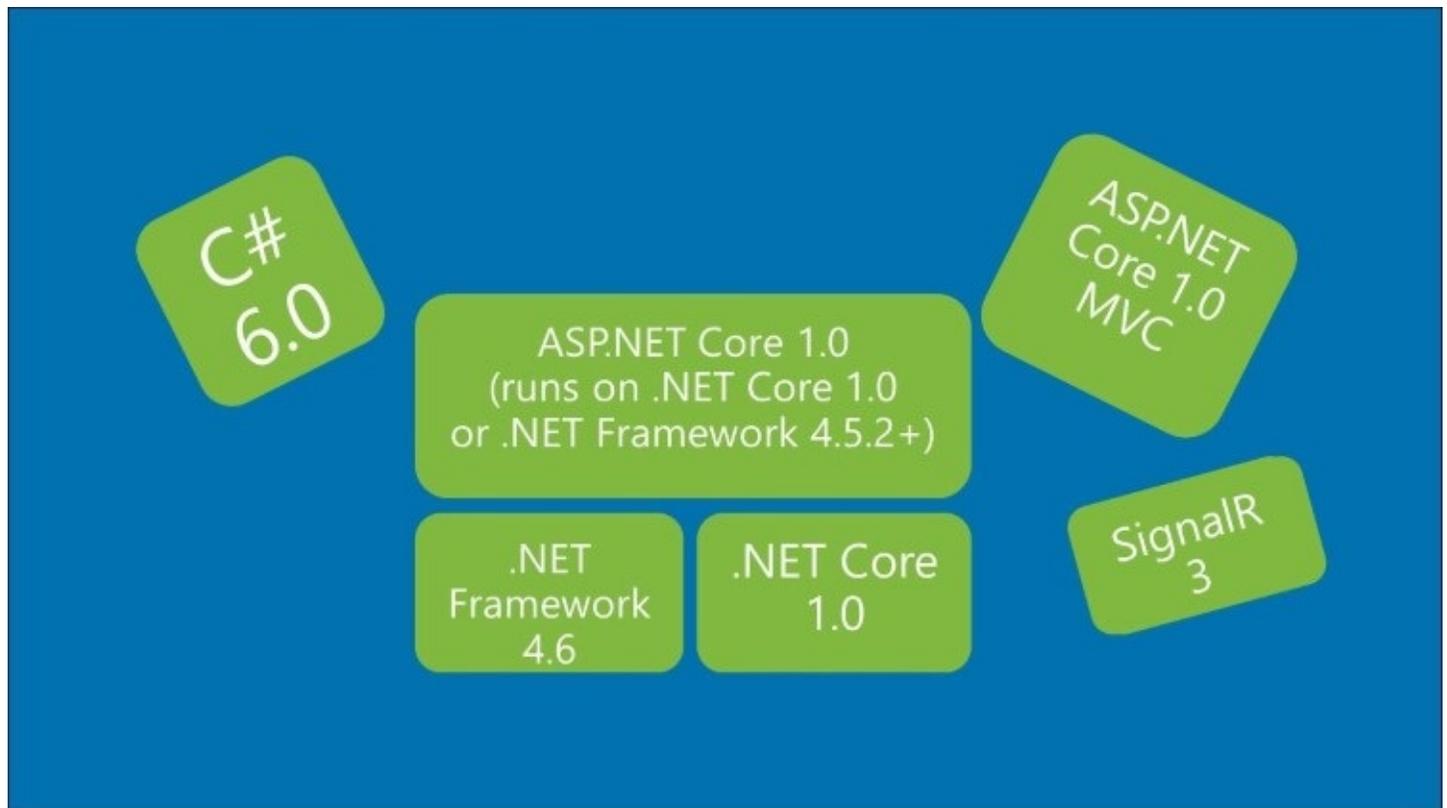
Putting it all together

Now that you're familiar with the moving parts of ASP.NET Core and its various version numbers, how will you decide which parts to use? The good news is that you won't have to figure it all out before you get started on your project.

Think about the requirements of your web software project and the needs of your customers. Choose the technologies that make sense for you and your team, and get started on a **Minimum Viable Product** (**MVP**).

If you need to support web browsers, go with ASP.NET MVC and build responsive web applications that look good on any screen size. If you need to extend your application with Web API, expose only what needs to be exposed.

If you need real-time functionality, **SignalR** is a great place to start. At the time of writing, Microsoft has announced that SignalR 3 will not be included with the 1.0 release of ASP.NET Core. As a result of this, SignalR 3 will not be covered in this book. For more information, stay tuned to the SignalR website: http://www.asp.net/signalr .

Differences between .NET Framework and .NET Core

Since the inception of .NET Framework, there has always been one release at a time. Whenever a newer version was released, you could upgrade to the latest version or install multiple versions side by side. But there was always one single latest version.

Going forward, there are now two distinct versions: the full .NET Framework and the all-new .NET Core. While ASP.NET Core web applications can run on either one of them, you will decide which is more suitable for your needs. You can change the runtime during development or during release. Best of all, you can deploy the runtime alongside your released product, which allows multiple versions of .NET to be deployed to the same server.

The following screenshot illustrates how ASP.NET Core runs on both .NET Framework 4.6 and .NET Core 1.0:
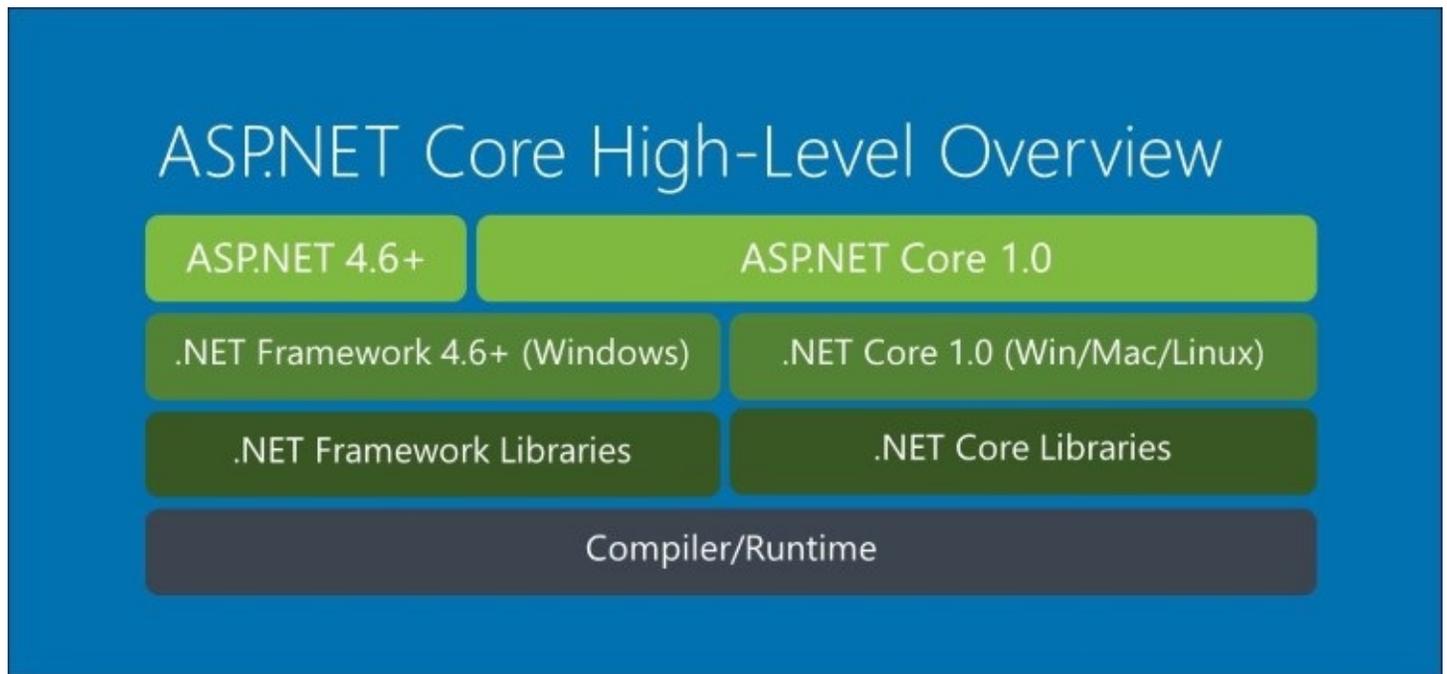
Full .NET Framework 4.6

Even with the release of .NET Core, there is still a place for the full .NET Framework. It will continue to be the framework of choice for rich Windows desktop applications, created with **Windows Presentation Foundation** (**WPF**) or Windows Forms. It will be one of two choices for ASP.NET Core developers.

For ASP.NET 4 developers, .NET Framework 4.6 will be an *in-place* replacement for .NET 4.x runtimes. This includes .NET 4, 4.5, 4.5.1, and 4.5.2. One good reason to upgrade to .NET 4.6 is the added benefit of new improvements such as better compilation and added language features.

Lightweight .NET Core

The new .NET Core is a lightweight cross-platform subset of the full .NET Framework that makes its home on Windows, Linux, and OS X. It is expected to leap past the .NET Framework in new features that may make their way back to the .NET Framework.

It is worth noting that .NET Core is not an option for Windows desktop developers or ASP.NET 4 developers. In addition to supporting ASP.NET Core web applications with the **CoreCLR** runtime, .NET Core also includes the .NET native runtime, which is specifically used for **Universal Windows Applications** on Windows 10.

Running ASP.NET Core on .NET Framework versus .NET Core

To recap, let's focus on the following benefits of running an ASP.NET Core web application on .NET Core, in addition to its cross-platform support:

- **Flexibility**: With .NET Core, web applications can be deployed with a specific version of the .NET Core framework, which will allow you to deploy each application with only the version that it needs
- **Performance**: With .NET Core, you'll enjoy performance benefits due to its lower memory footprint and faster start-up times

Whether you run ASP.NET on the full .NET Framework or the new .NET Core, you'll enjoy a modern application framework that eases you into cloud deployment and facilitates faster development with dynamic compilation.

What's new with Visual Studio 2015 and Visual Studio Code?

There are several different SKUs of Visual Studio available at each release cycle. Starting with Visual Studio 2013, Microsoft added a Community Edition as a free alternative to the Professional Edition for students, open source developers, and small teams. Alongside Visual Studio 2015, Microsoft has also added a cross-platform code editor named **Visual Studio Code**.

For developers, the following are the current offerings of Visual Studio 2015:

- Community Edition
- Professional Edition
- Enterprise Edition

You may recall that previous versions of Visual Studio also included a Premium Edition and Ultimate Edition. These two have been merged into the Enterprise Edition.

For a high-level overview of the various editions of Visual Studio, you may refer to the comparison table at the following URL:

https://www.visualstudio.com/products/vs-2015-product-editions

Community Edition

Before the Community Edition, many students, independent developers, and small teams used the free Express Edition. However, the Express Edition was missing many Pro features, such as extensions. Now, with the Community Edition, non-enterprise developers can get a full-featured IDE for cross-platform application development.

Better yet, the Community Edition opens the door to thousands of Visual Studio extensions. If you don't find what you need in the **Visual Studio Gallery**, you can also create your own extensions.

Professional and Enterprise Editions

For enterprise developers and professional teams with more needs, you can choose from the Professional or Enterprise Editions. Beyond the full-IDE features of the Community Edition, the Professional Edition also gives you access to the **CodeLens**, **Team Foundation Server** (**TFS**), and **Agile** project planning tools. CodeLens offers a deeper look into your code history, while TFS can be used for source control and continuous integration.

The Enterprise Edition gives you all of the above, plus enterprise-grade tools for software architecture, modeling, testing, and code coverage.

For a detailed look at the various editions of Visual Studio, you may refer to the comparison table at https://www.visualstudio.com/products/compare-visual-studio-2015-products-vs .

Visual Studio Code

To provide even more choice to developers today, Microsoft has started offering a cross-platform code editor aptly called Visual Studio Code. This new offering is free and available for your platform of choice: Windows, Mac OS X, and Linux.

Visual Studio Code is primarily made for web and cloud applications, such as ASP.NET web applications or Node.js backend code. But you are free to use it for other uses, such as Unity game development and cross-platform application development with most popular languages such as C++, Java, PHP, Python, Ruby, and many others.

For those of you who are wondering, Visual Studio Code bears a striking resemblance to the popular **Sublime Text** and is a good alternative to it. The following screenshot shows Visual Studio Code:
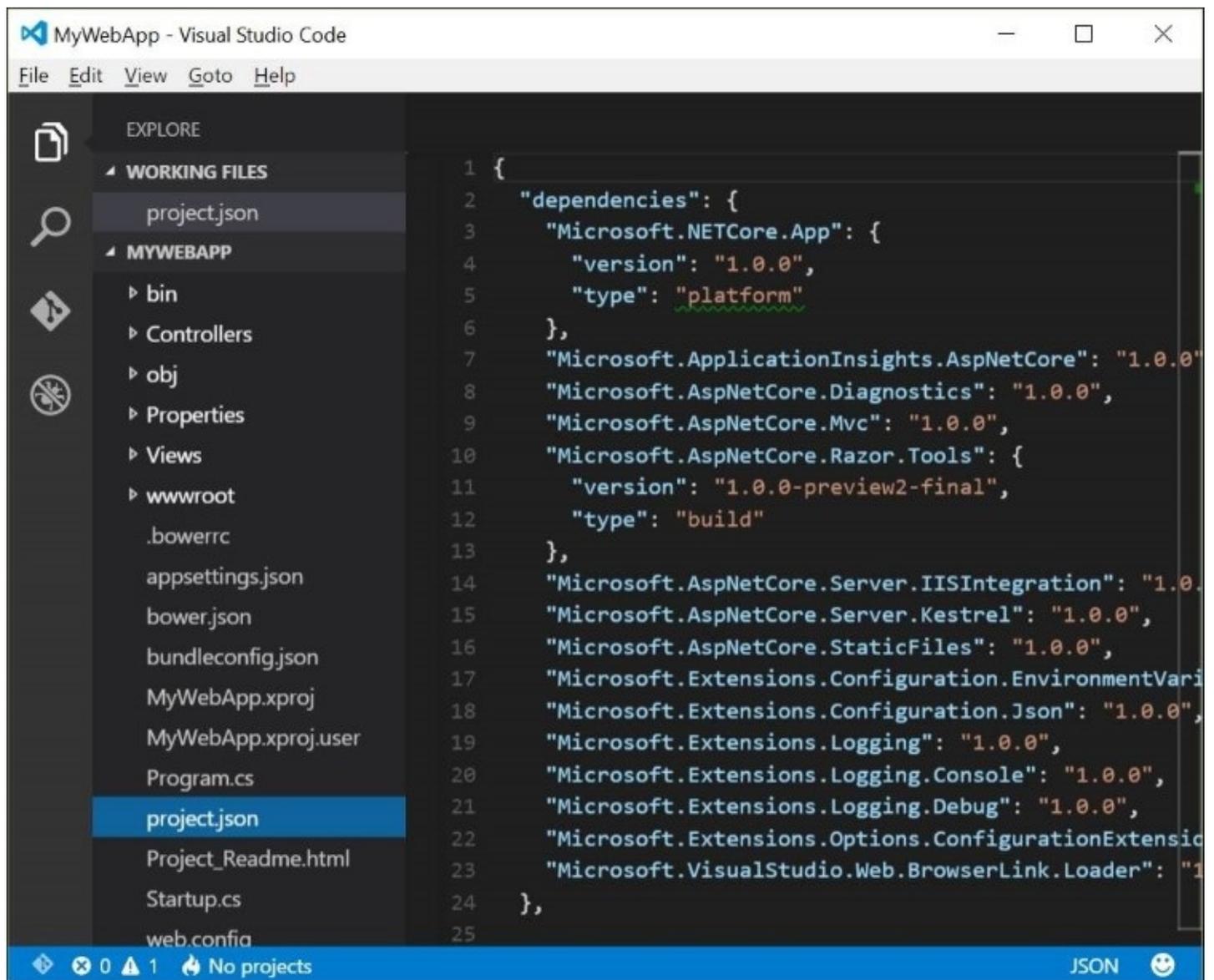
Visual Studio Code:project.json

For more information on other languages, you may refer to https://code.visualstudio.com/Docs/languages .

All of these versions of Visual Studio 2015 and Visual Studio Code can be used for ASP.NET Core web application development. When you create a new project in Visual Studio 2015, you'll see project templates for an **Empty** project, a **Web API** project, and a **Web Application** project, as shown in the following screenshot. Visual Studio Code, on the other hand, is file/folder-based (as opposed to project/solution-based), so you can open a project's files by opening a folder that contains a supported project:
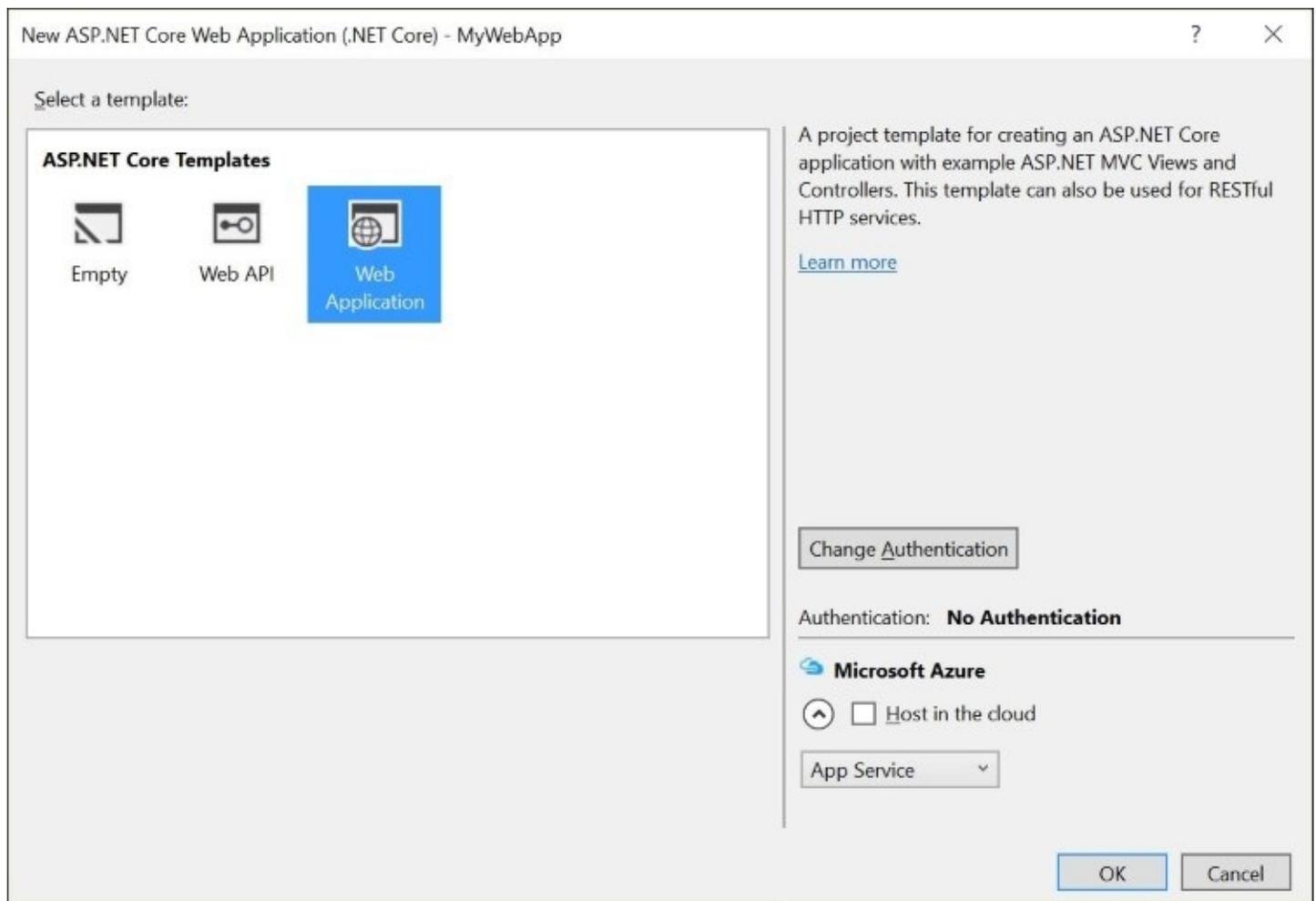
In addition to Visual Studio, you could also use **Yeoman** to create a new ASP.NET project to work on. Yeoman is a scaffolding tool that can help you generate various modern web apps, including ASP.NET. For more information on Yeoman, check out their website at http://yeoman.io .

This book will focus primarily on Visual Studio 2015. All the screenshots and instructions will use Visual Studio 2015 Community Edition and the Edge web browser running on

Windows 10. If you use a different editor, operating system, or web browser, you will have to perform the equivalent steps for your own environment. The application code should remain the same, regardless of the platform.

Running ASP.NET Core on Windows, Mac OS X, and Linux

With ASP.NET Core, you can now develop on, and deploy your applications to, Mac OS X and Linux, in addition to Windows. If you worked with **Active Server Pages** (**ASP**) a long time ago, you may remember **Chili!Soft ASP**, a third-party solution web server plugin that allowed ASP to run on Solaris, Linux, and other Unix platforms. The new .NET Core is much more than that.

With a cross-platform runtime that can be deployed along with your web application, .NET Core provides official support from Microsoft to enjoy the benefits of ASP.NET on your favourite platform. With the aforementioned Visual Studio Code, you can also easily build ASP.NET Core applications on those platforms.

Although this book will primarily cover ASP.NET Core development on Windows, the following instructions will help you get your environment set up on OS X and Linux as well.

ASP.NET Core on Windows

Setting up your development environment for ASP.NET Core on Windows is pretty straightforward. If you have Visual Studio 2015 Update 3 with the latest Web Developer tools installed, you already have what you need. The new ASP.NET Core templates are enough to get you started.

If you prefer, you can also use Visual Studio Code on Windows if you just need a basic code editor on one or more development machines. Be aware that the Community Edition is only for non-enterprise customers, so you cannot install it alongside the new Professional or Enterprise Editions.

ASP.NET Core on Mac OS X

To set up your development environment on Mac OS X, you can start by installing the latest version of Visual Studio Code. You can also install ASP.NET Core from a command line. Here is a high-level three-step process to get you started:

1. Install Open SSL.
2. Install .NET Core SDK.
3. Install Visual Studio Code for OS X.

For up-to-date detailed instructions (and command-line instructions) for setting up ASP.NET Core on Mac OS X, you may refer to Microsoft's official guide at https://www.microsoft.com/net/core#macos .

ASP.NET Core on Linux

Setting up your development environment on Linux is a little more complicated. As you may expect, you can get started by installing Visual Studio Code on your Linux machine.

To summarize the steps to set up the runtime, you will have to do the following:

1. Install prerequisites, which may vary for your version of Linux.
2. Install .NET Core SDK.
3. Install Visual Studio Code for Linux.

For up-to-date detailed instructions on setting up ASP.NET Core on Linux, you may refer to Microsoft's official guide at the following URL. Select a Linux distribution to see more details about that particular distro, such as Red Hat, Ubuntu, Debian, and many others, at https://www.microsoft.com/net/core .

Summary

In this chapter, we've taken an introductory look at ASP.NET Core, the .NET Framework, and the various versions of Visual Studio. We also learned about the cross-platform nature of .NET Core, and provided a quick overview of how you can set up your development environment on various operating systems.

In the next chapter, we will learn about how you can build your very first ASP.NET Core web application in Visual Studio 2015 running on Windows 10. We will go through project templates and also dissect a basic web application to better understand its parts and its configuration.

Chapter 2. Building Your First ASP.NET Core Application

Whether you're a seasoned ASP.NET developer or a little rusty on controller methods, the best way to get up-to-speed on ASP.NET Core is with a **Hello World** application. In addition to new configuration files, there are also new project types to learn about.

Going beyond its traditional reliance on IIS or IIS Express, your new ASP.NET web application can also be self-hosted without a web server. This new paradigm is the basis for its cross-platform ambitions.

In this chapter, we will discuss the following principles and concepts to help you become familiar with the basic structure of an ASP.NET Core application:

- Project templates
- Models, views, and controllers
- Web configuration

Project templates in Visual Studio 2015

The quickest way to expose yourself to the new project templates in Visual Studio 2015 is to start using them. For ASP.NET Core, there are currently three project templates that you can use.

Launch Visual Studio 2015, then click **File** | **New** | **Project**. Within **Visual C#** | **Web** templates, select a template for ASP.NET Core Web Application to proceed. As of the Core 1.0 release, there are two different versions: one that uses the all-new .NET Core runtime, and another that uses the full .NET Framework, as shown in the following screenshot:
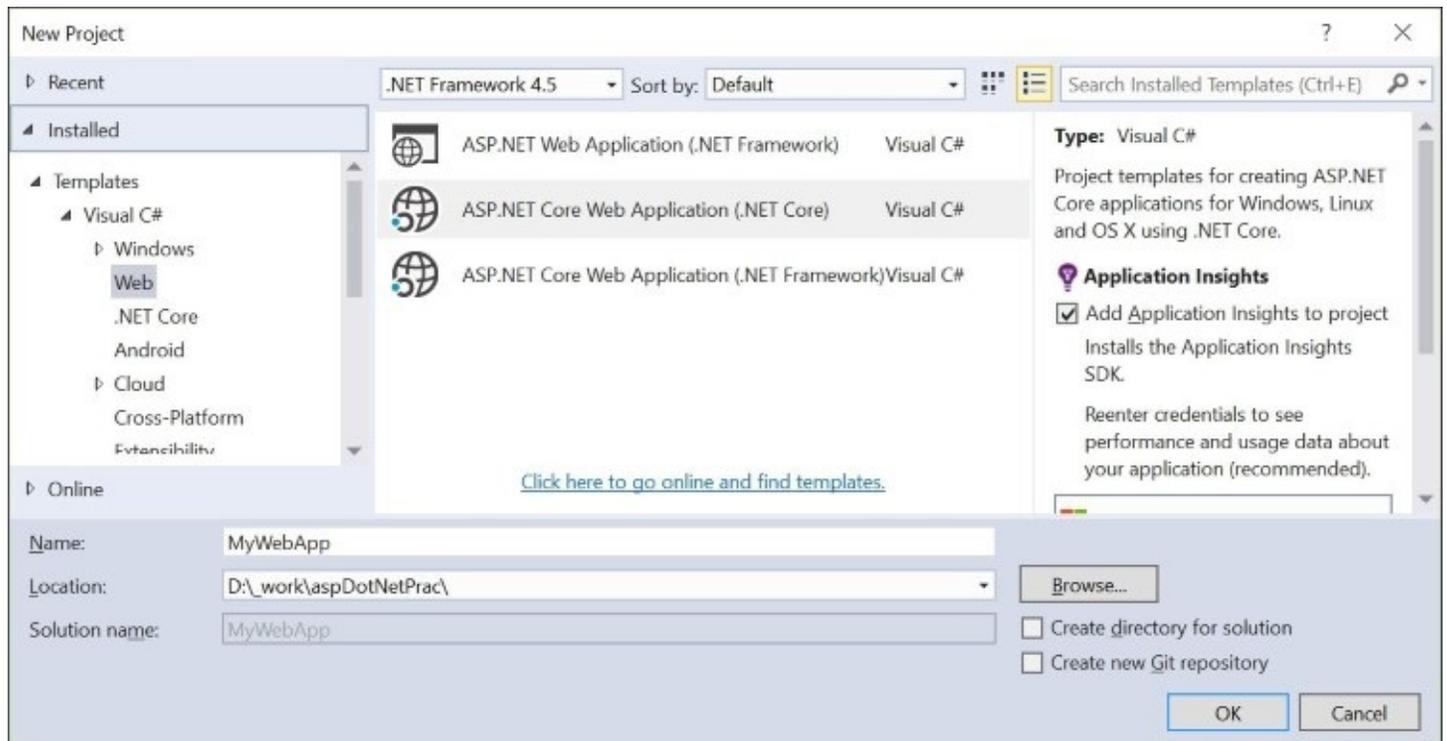
In pre-release versions, there were other project types available as the **Web** templates. These are now listed under .NET Core instead, and are as follows:

- ASP.NET Web Application
- Class Library (Package)
- Console Application (Package)

The project template for ASP.NET Core 1.0 creates a console application that has a `Main` method in a `Program.cs` file as its entry point. This is similar to other console applications you may already be familiar with. When you proceed with any of the ASP.NET Core options, you will be greeted with the all-new ASP.NET Core templates. The following screenshot shows the available project templates under **ASP.NET Core Templates**:
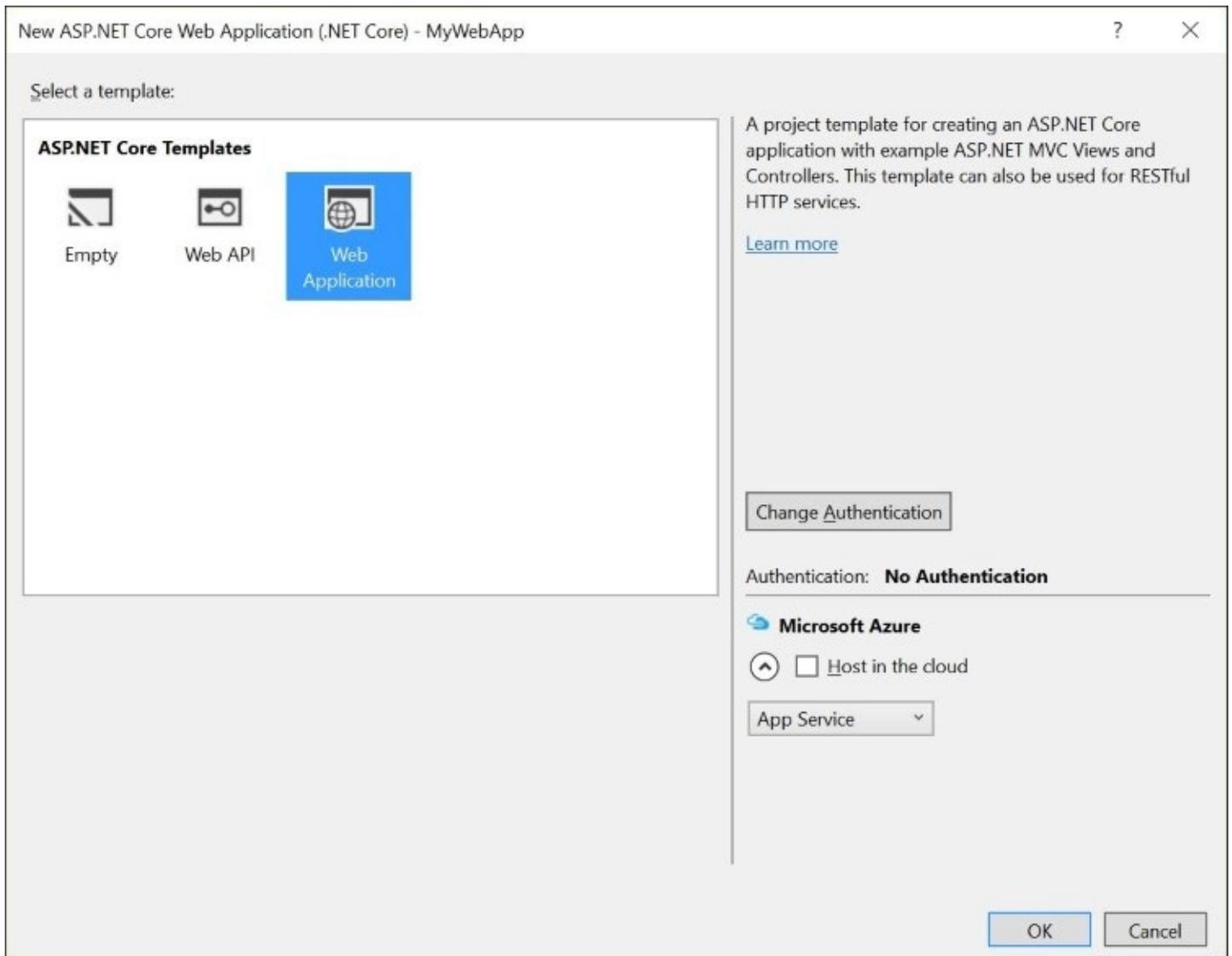
The new templates are available as follows:

- **Empty**
- **Web API**
- **Web Application**

You'll note the absence of Web Forms in the list of new templates. As mentioned in Chapter 1 , *Getting Started with ASP.NET Core,* the new ASP.NET favors MVC over Web Forms, which it eschews in its cross-platform implementations.

Empty template

One common complaint from developers using some older versions of Visual Studio has been that the **Empty** template wasn't empty enough. As a result, those developers would start with the **Empty** template, only to start stripping out components of the newly created project.

This was improved in Visual Studio 2013 and continues to work as expected in Visual Studio 2015. Selecting the **Empty** option ensures that you will have a barebones project with just enough to get started, as shown in the following screenshot:
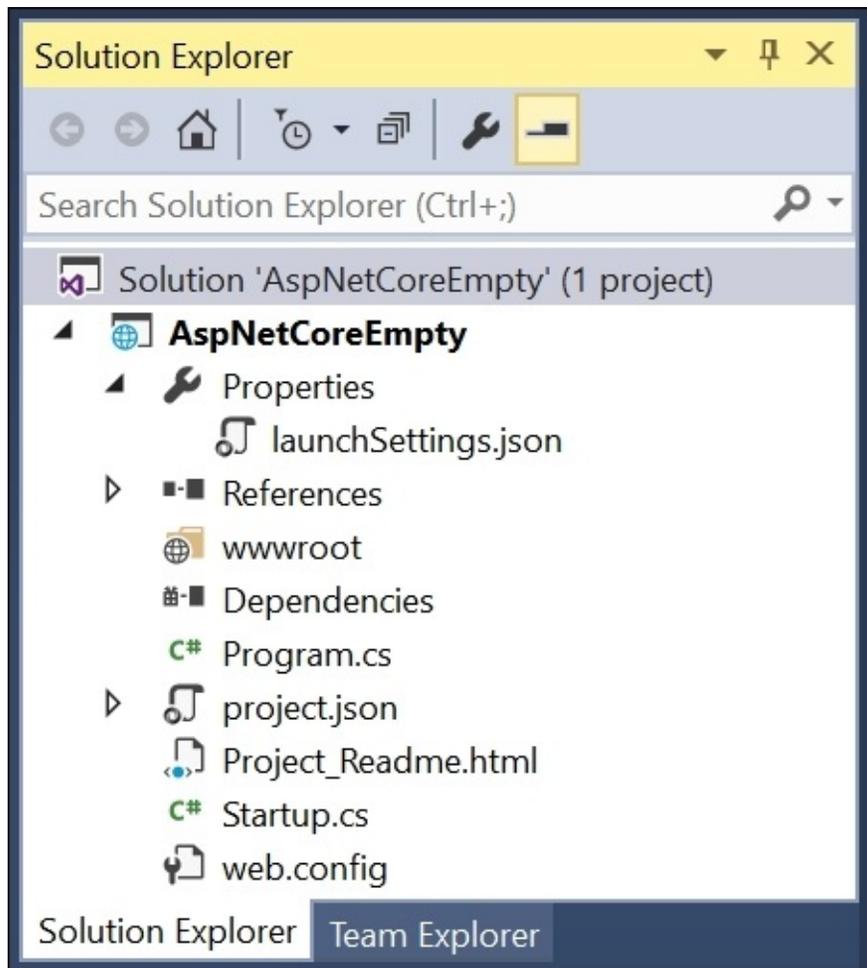
Within an **Empty** template project, you can identify the following items:

- (Project) **Properties**
- **References**
- **wwwroot** (web root folder)
- **Dependencies** (initially empty)
- **Program.cs** (contains the `Main` method for the entry point)
- **project.json** (minimal settings)
- **Project_Readme.html** (information about ASP.NET Core)

- **Startup.cs**:
  - `ConfigureServices()`
  - `Configure()`
  - `Main()`

- **web.config** (for IIS launch only; use JSON-based files for configuration)

Web API template

The ASP.NET Web API template was released with ASP.NET MVC 4 back in 2012. It allowed ASP.NET developers to easily build HTTP services to be consumed by web or mobile clients. With ASP.NET Core MVC, developers can enjoy a more streamlined experience while building a Web API in an MVC project.

The Web API template is a great starting point for creating RESTful HTTP services for your web project. Selecting the Web API project template will give you a little more than the Empty template. The structure of a Web API project is shown in the following screenshot:
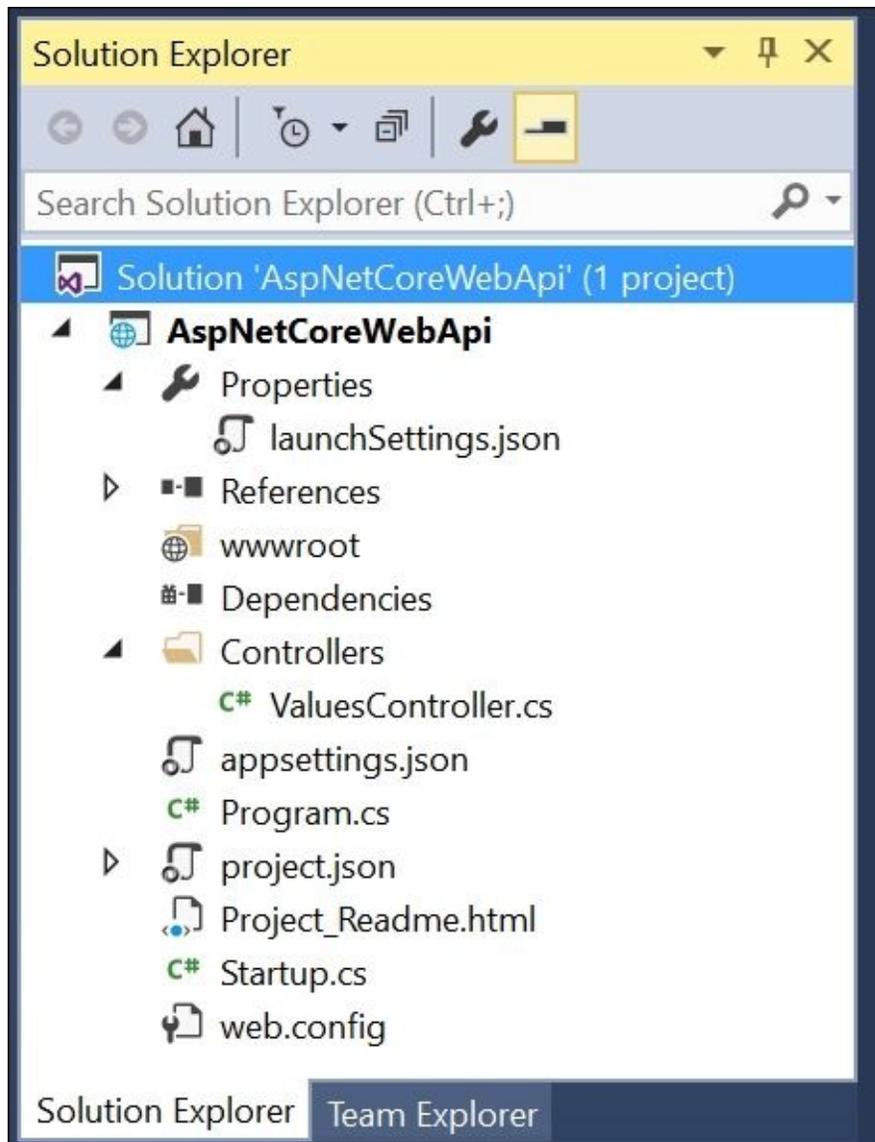
Within a Web API template project, you can identify the following items:

- (Project) **Properties**
  - **launchSettings.json**

- **References**
- **wwwroot** (web root folder)
- **Dependencies** (initially empty)
- **Controllers** folder
  - **ValuesController.cs**

- **appsettings.json** (minimal settings)
- **Program.cs** (contains the `Main` method for the entry point)
- **project.json** (basic settings and references)
- **Project_Readme.html** (information about ASP.NET Core)
- **Startup.cs**
  - `Startup()` constructor
  - `ConfigureServices()`
  - `Configure()`

Going beyond the Empty template, the Web API template has the following additional items:

- **ValuesController.cs**
- **appsettings.json**

Web Application template

The Web Application template provides a lot more initial code and content than any of the other templates. This is a good starting point for learning about the ins and outs of ASP.NET Core. When using this web template, click on the **Change Authentication** button to include **Individual User Accounts** as the type of authentication for your starter application. The structure of a Web Application project is shown in the following screenshot:
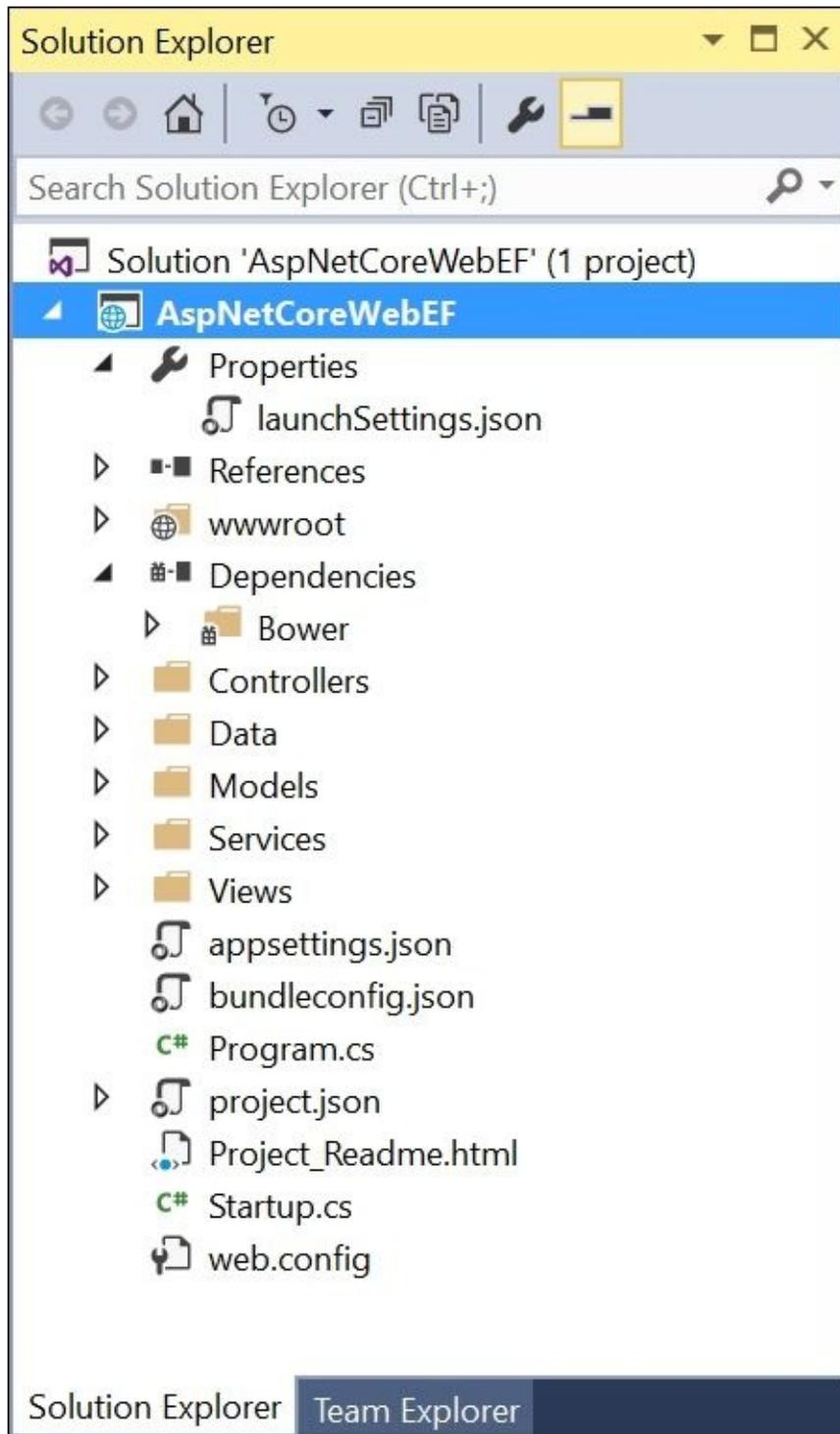
Within a Web Site template project, you can identify the following items. First, there are some top-level items above the folders in your project:

- (Project) **Properties**
  - **launchSettings.json**

- **References**
- **wwwroot**
  - **css**
  - **images**
  - **js**
  - **lib**
  - **_references.js**
  - **favicon.ico**

- **Dependencies**
  - **Bower** (for **Bootstrap**, **jQuery**, and others)

Next, there are several folders for your code, MVC components, and DB migrations, as follows:

- **Controllers** folder
  - Class files for each controller

- **Data**
  - Migrations subfolder for initial migration
  - DB context snapshot

- **Models**
  - `Models` and `ViewModels`

- **Services**
  - Service classes, such as for messaging and e-mail

- **Views**
  - Views organized in subfolders per controller
  - Shared views
  - `_ViewImports.cshtml` (shared imports for views)
  - `_ViewStart.cshtml` (share view header for views)

Finally, there are several JSON configuration files and a couple of `.cs` code files, such as `appsettings.json`, `bundleconfig.js`, `Program.cs`, `project.json`, `Project_Readme.html`, and `Startup.cs`:

- `appsettings.json` (app settings, such as connection strings)
- `bundleconfig.json` (bundling and minification configuration)
- `Program.cs` (entry point with `Main` method)

- `project.json` (project settings, server-side references)
- `Project_Readme.html` (information about ASP.NET Core)
- Startup.cs
  - `Startup()` constructor
  - `ConfigureServices()`
  - `Configure()`

- `web.config` (for IIS launch only; use JSON-based files for configuration)

You may have to click on the **Show All Files** icon in Solution Explorer to see additional files that may be initially hidden.

Hello, ASP.NET - your first ASP.NET application

Most programming books and tutorials help you get started with a Hello World application. This refers to a simple application that has all the basic elements of an application to get it up-and-running in your environment. If the words Hello World can be displayed on a screen by the application, you have succeeded in setting it up correctly.

Create a new ASP.NET web application project and select the Empty template as described earlier this chapter. This should provide a basic template for your Hello World application.

Writing a response

By default, the empty template includes a `Startup.cs` class with a `Configure()` method in it. This method should have a line of code that writes *Hello World* to the HTTP response output stream, to be displayed in your browser. The following code is for the `Configure()` method:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

The `Configure()` method can be used to configure the HTTP request pipeline by enabling static files, Identity, MVC, and others. It can be used to set a default route inside the `Startup.cs` class, replacing the need to have a `global.asax` file.

Note that the `Configure()` method is called after the `ConfigureServices()` method, which can be used to add various services, such as Entity Framework, Identity, and MVC. Both of these methods are called by the runtime.

To change the text that is being displayed, simply replace `Hello World!` with `Hello, ASP.NET Core!`. This text change doesn't do anything functionally different than what the Empty template provides, but it does expose you to the `Startup` class and its methods.

Launching the application

To launch your application, press *F5* on your keyboard, or click **Start Debugging** in the **Debug** menu. This should launch the application in your default web browser, as shown in the following screenshot:
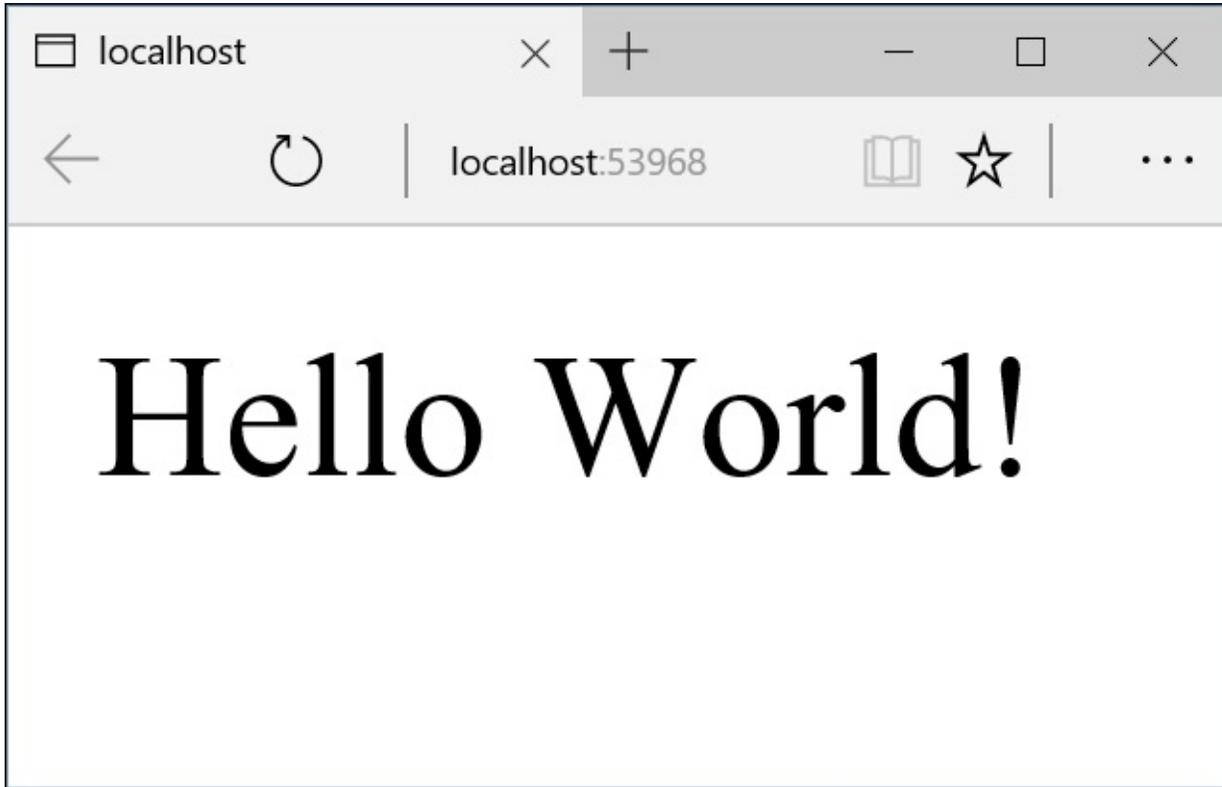
You may also be familiar with the green *Play* button, which launches your application, on the toolbar in Visual Studio. In Visual Studio 2015, this button offers some new choices, as shown in the following screenshot:
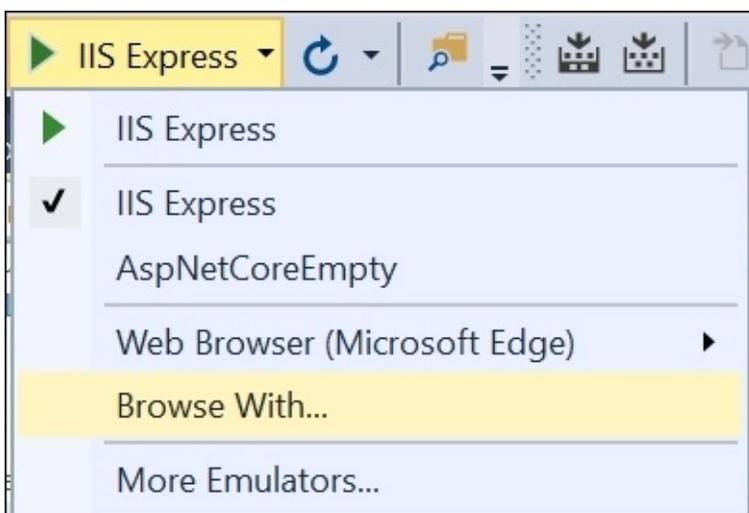
The choices are as follows:

- **IIS Express**: The default option for launching in the default web browser
- **Internal Web Host**: Named after the project's name
- **web, gen, ef, and others**: Various commands, configurable through the `project.json` file
- **Browse With**: Change the default browser or add new browsers

You may have noticed that you can select a different web browser while debugging. Simply browse the list of web browsers to select a different browser for debugging purposes, as shown in the following screenshot:
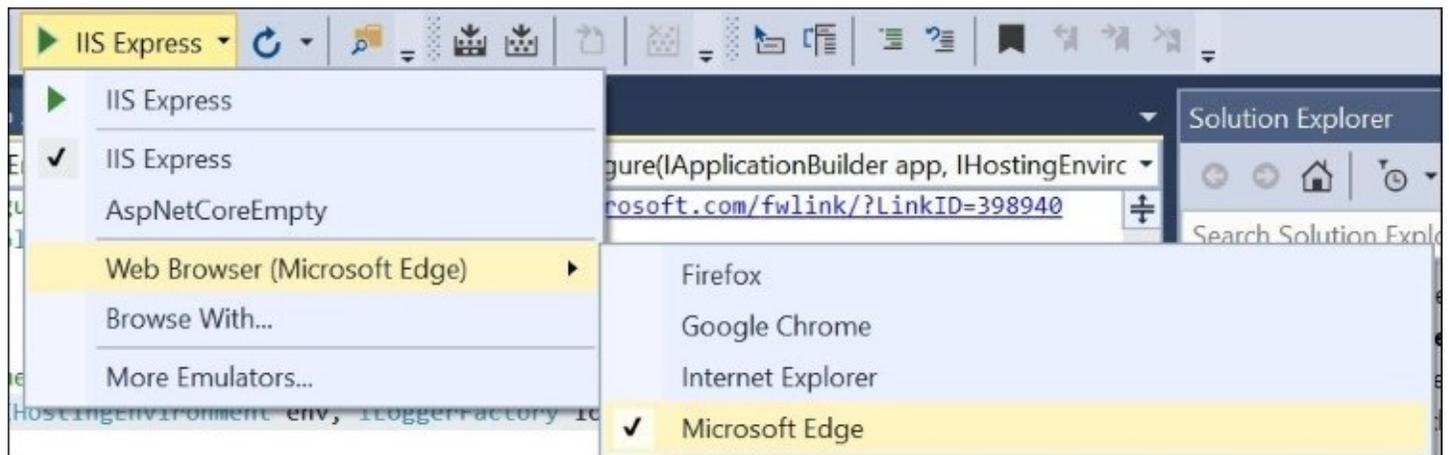
You may also right-click your project in Solution Explorer to access its properties. In the project properties panel, click on the **Debug** tab to further customize each debugging profile, as shown in the following screenshot:

AspNetCoreEmpty	project.json	web.config	Startup.cs	Welcome to ASP.NET Core

Application

Build

Debug

Configuration: N/A   Platform: N/A

Profile   AspNetCoreEmpty

Launch: Project

Application Arguments: *Arguments to be passed to the application*

Web files and folders

Now let's take a deeper look at all of these files and folders in the Web Application template project to understand their purpose. First, we'll focus on the configuration files, starting with `bower.json`.

This JSON configuration file is used to manage **Bower** dependencies. It starts off with a list of dependencies, followed by an optional detailed configuration for each dependency, as shown in the following screenshot:

```
bower.json  ⊟ ✕
Schema: http://json.schemastore.org/bower
1    ⊟ {
2          "name": "asp.net",
3          "private": true,
4    ⊟     "dependencies": {
5              "bootstrap": "3.3.6",
6              "jquery": "2.2.0",
7              "jquery-validation": "1.14.0",
8              "jquery-validation-unobtrusive": "3.2.6"
9          }
10    }
11
```

You will notice that this is the same list of Bower dependencies that appears in the Solution Explorer panel, as shown in the following screenshot:
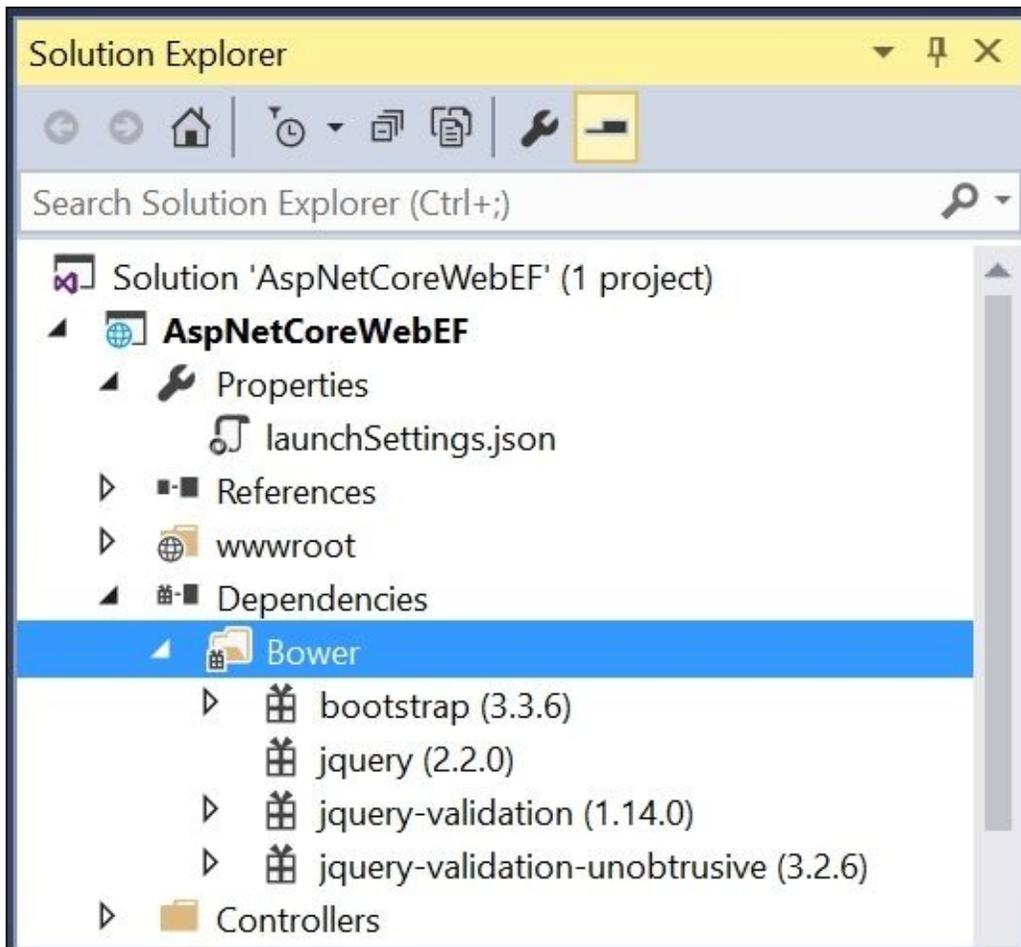
Next, let's take a look at `appsettings.json`, which is your application configuration file. With ASP.NET Core, this new JSON file (temporarily known as `config.json` while in Beta) replaces the need to store your settings in the XML-based `web.config` file. You can specify your configuration file in your `Startup.cs` code.

Your settings file may look like this:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "<YourDbConnString>"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Last but not least, we have `project.json`, a JSON file that contains project-specific

information and server-side dependencies. See the *Bundling and publishing* section in this chapter for more information.

Models, views, and controllers - an MVC refresher

For those of you not familiar with MVC, here's a quick refresher that also includes information that is new to ASP.NET Core. The following figure shows a simple architecture diagram that represents the MVC software architectural pattern:
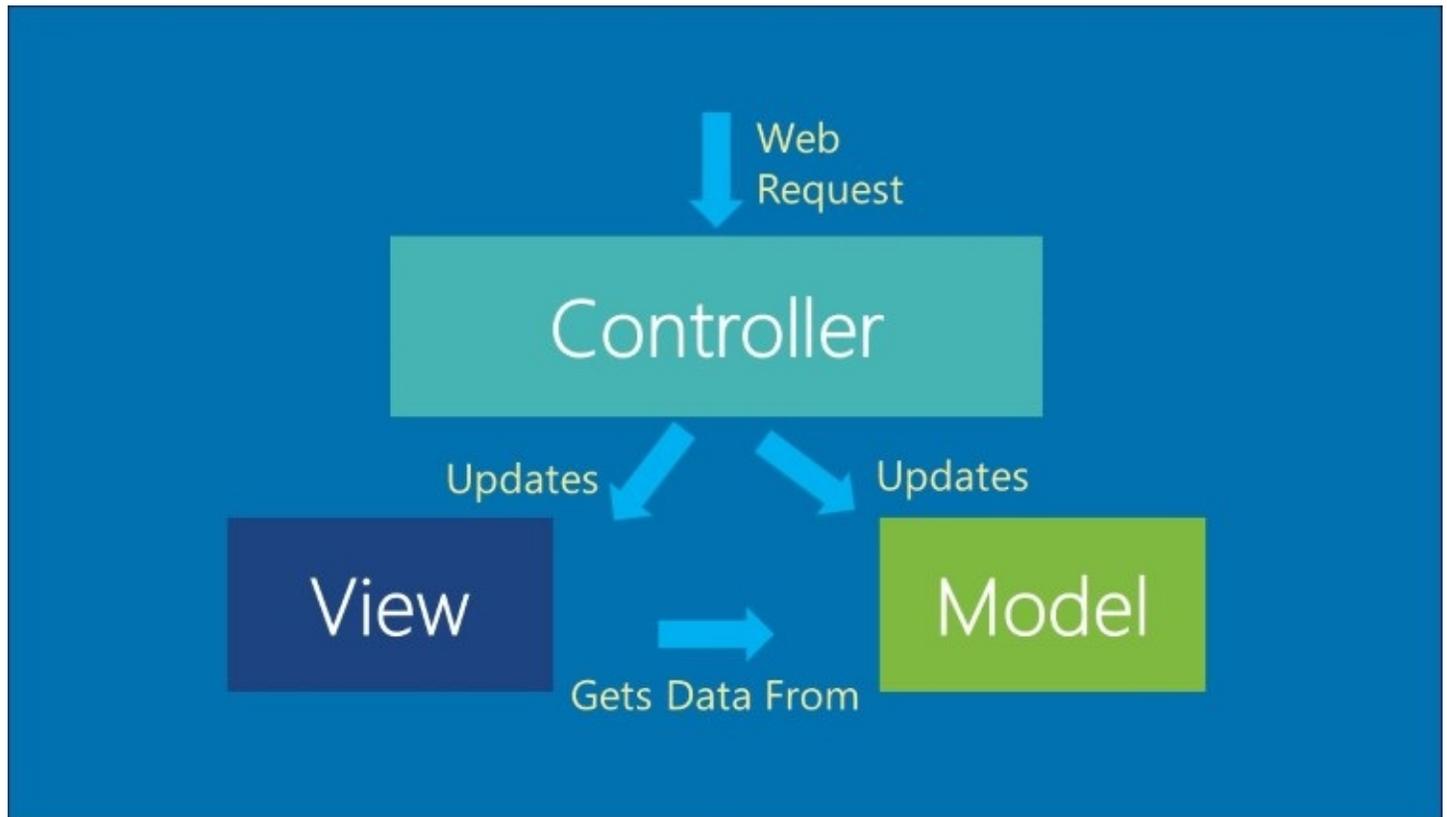
Even though the figure shows one of each item, you can have multiple models, views, and controllers in your figure. Each request gets routed through a specific controller to determine the result that will be displayed in the user's web browser.

MVC itself wasn't a new concept when Microsoft first released ASP.NET MVC. But it immediately introduced .NET developers to a new way of developing web applications. The benefits over Web Forms became apparent to early adopters: clearer separation of concerns, better testability, and lightweight client output that can be customized.

Some developers weren't tempted away from Web Forms for a variety of reasons: a new learning curve, personal preference, corporate requirements, and many others. Going forward, ASP.NET MVC becomes more necessary since Web Forms won't be supported by the cross-platform .NET Core. Moreover, ASP.NET Core does not depend on `System.Web` like its predecessors.

So, if you haven't looked at MVC yet, now is a great time to do so.

Controllers

In a way, controllers are the heart of your ASP.NET MVC application. Each controller is responsible for handling user requests, based on a matching route. Controllers can update data in a model, and then select a view to return back to the user.

Every controller is a subclass of the base `Controller` class, which lives in the `Microsoft.AspNetCore.Mvc` namespace. This is different from the `Controller` class from prior versions of ASP.NET, which lived in the `System.Web.Mvc` namespace. In fact, the new `Controller` base class is also used by Web API controllers.

In MVC, each controller typically returns an **IActionResult** from its action methods. For web application projects, this could be a view. For Web API projects, this could be a set of data. It is possible to return both views and results from a controller.

Models

As before, your models will embody your project's data domain. ==Each model class can represent entities in your code,== while decorated by attributes surrounded by square brackets. This kind of declarative syntax also allows you to add validation rules inside your models.

A model's data can be affected by the controller operating on it. By binding your models to your views, each view will automatically determine what to display and how to display it.

For a cleaner architecture, you can use a view-specific model (or `ViewModel`) to bind to a view. You can use a repository pattern with a service layer for models that reflect your database entities through Entity Framework. This will let you have UI elements that don't have to rely on the structure of your database entities.

See [Chapter 6](#) , *Using Entity Framework to Interact with Your Database in Code*, for more information on Entity Framework, an **Object-Relational Mapping** (**ORM**) framework for .NET applications.

Views

The views in your application are probably the simplest part of MVC. Each view represents the UI layer, resulting in client-side HTML, CSS, and JavaScript that will be displayed to the end user.

Views are stored in `.cshtml` files, with the ability to include server-side code and client-side code in the same file. A built-in object called `ViewBag` allows you to store your own properties and display them in the view. The `ViewBag` object and its properties can be manipulated by your controller code.

Instead of switching back and forth between server-side code and client-side code, you can now use tag helpers in your views for smoother syntax. A tag helper allows you to use custom attributes within your HTML tags that may be familiar to developers who use **AngularJS**.

See Chapter 3, *Understanding MVC,* for more information on tag helpers and views.

Web configuration with project.json ←

In the `project.json` file, you may find the following information:

- **userSecretsId**: A GUID-like value used for working with user secrets
- **dependencies**: A list of server-side dependencies
- **version**: The project version
- **frameworks**: .NET Core framework name and version
- **build/runtime/publish options**: Build/runtime configuration and a list of files/folders to include when publishing the web app
- **scripts**: A list of scripts for actions that can be triggered by specific events, such as `prepublish`, `postpublish`, and others

The following screenshot of the `project.json` file shows the contents of a typical project configuration file, collapsed to fit all top-level items:
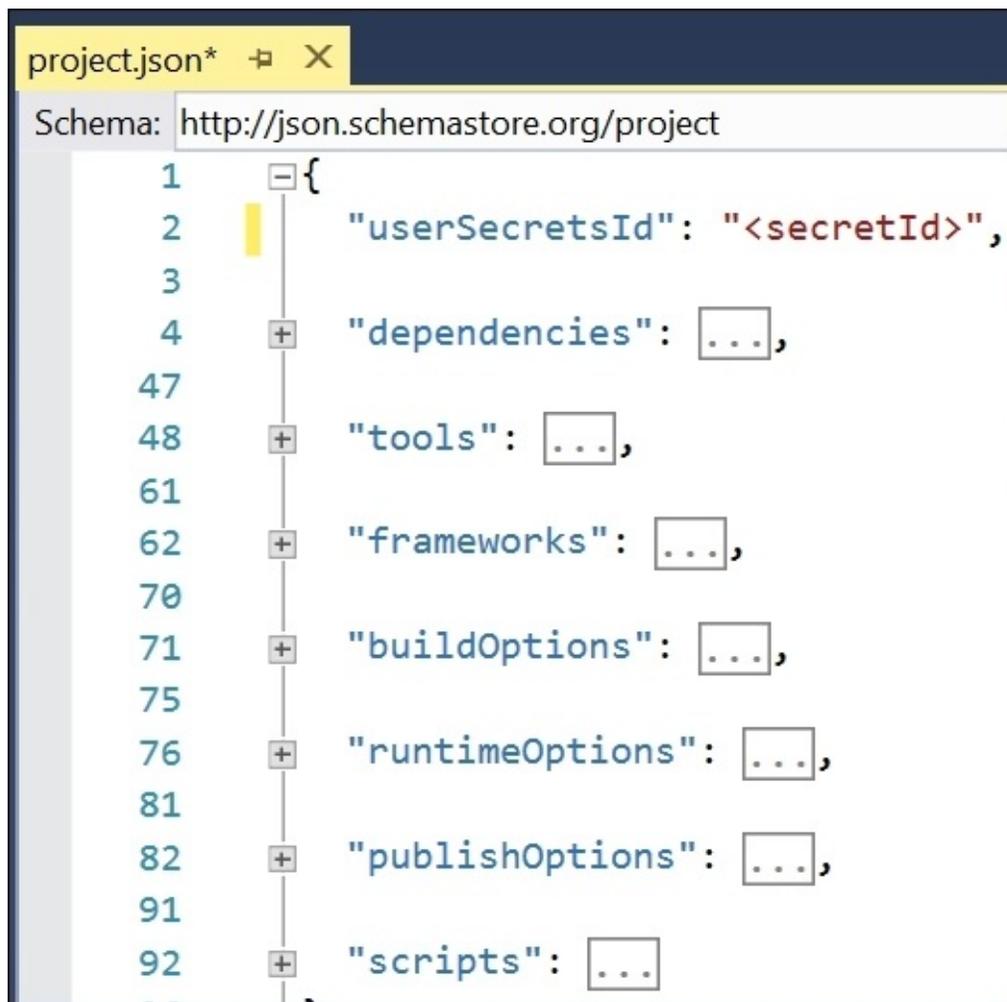
Dependencies and frameworks

Within the dependencies section of your configuration file, you will find a list of all your server-side dependencies, such as ASP.NET MVC, Entity Framework, and many others. These are also visible as your project references in Solution Explorer.
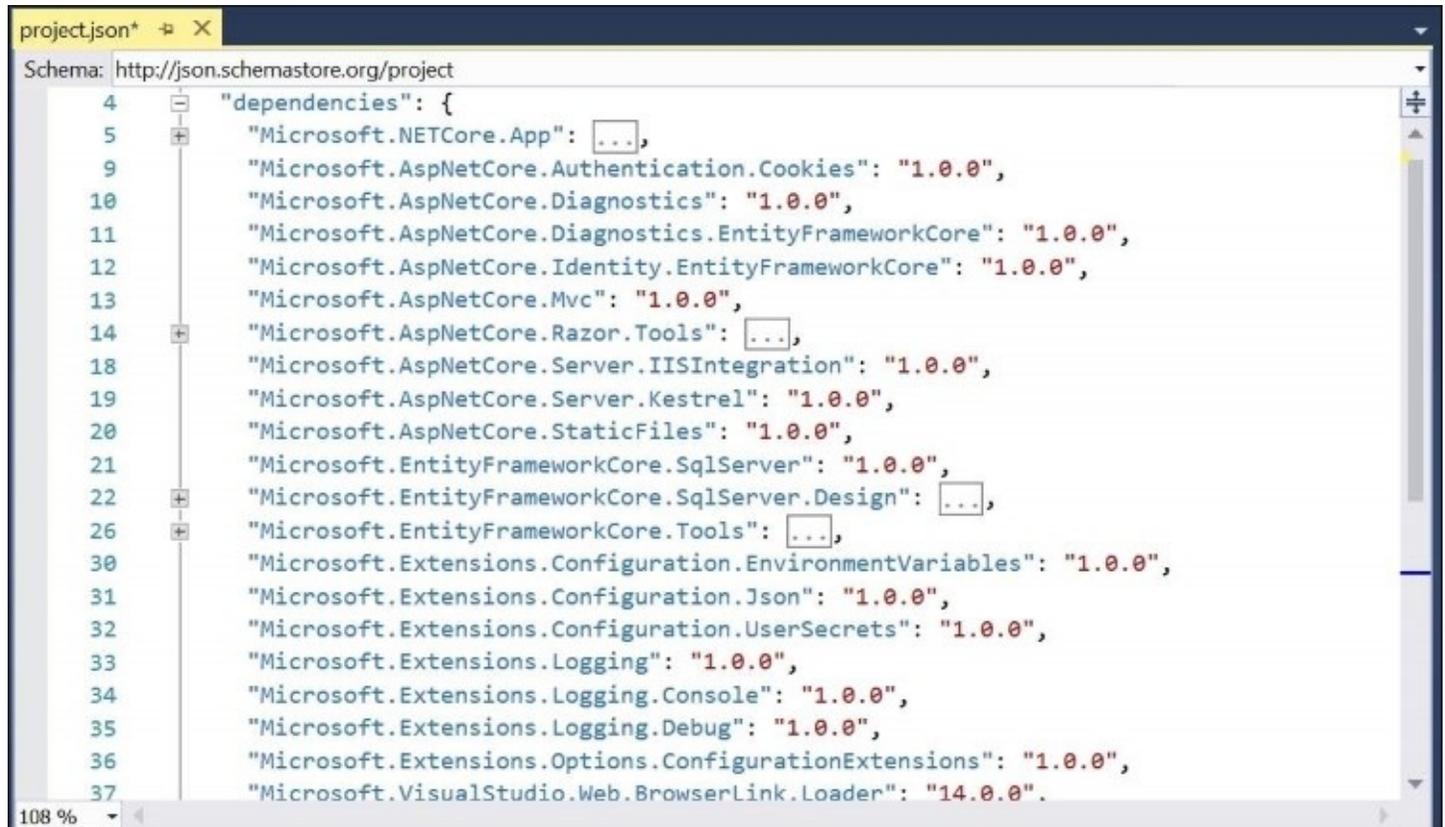
```
project.json*   ⊣ ✕
Schema: http://json.schemastore.org/project
    4   ⊟    "dependencies": {
    5   ⊞       "Microsoft.NETCore.App": [...],
    9           "Microsoft.AspNetCore.Authentication.Cookies": "1.0.0",
   10           "Microsoft.AspNetCore.Diagnostics": "1.0.0",
   11           "Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore": "1.0.0",
   12           "Microsoft.AspNetCore.Identity.EntityFrameworkCore": "1.0.0",
   13           "Microsoft.AspNetCore.Mvc": "1.0.0",
   14   ⊞       "Microsoft.AspNetCore.Razor.Tools": [...],
   18           "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
   19           "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
   20           "Microsoft.AspNetCore.StaticFiles": "1.0.0",
   21           "Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",
   22   ⊞       "Microsoft.EntityFrameworkCore.SqlServer.Design": [...],
   26   ⊞       "Microsoft.EntityFrameworkCore.Tools": [...],
   30           "Microsoft.Extensions.Configuration.EnvironmentVariables": "1.0.0",
   31           "Microsoft.Extensions.Configuration.Json": "1.0.0",
   32           "Microsoft.Extensions.Configuration.UserSecrets": "1.0.0",
   33           "Microsoft.Extensions.Logging": "1.0.0",
   34           "Microsoft.Extensions.Logging.Console": "1.0.0",
   35           "Microsoft.Extensions.Logging.Debug": "1.0.0",
   36           "Microsoft.Extensions.Options.ConfigurationExtensions": "1.0.0",
   37           "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0".
108 %  ▼
```

The preceding screenshot shows the **dependencies** section expanded to reveal the references and their version numbers. The following screenshot shows the same set of references in Solution Explorer:

Solution Explorer

Search Solution Explorer (Ctrl+;)

- ◢ ▥ .NETCoreApp,Version=v1.0
    - ▷ 🗗 Microsoft.AspNetCore.Authentication.Cookies (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Diagnostics (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Identity.EntityFrameworkCore (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Mvc (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Razor.Tools (1.0.0-preview2-final)
    - ▷ 🗗 Microsoft.AspNetCore.Server.IISIntegration (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.Server.Kestrel (1.0.0)
    - ▷ 🗗 Microsoft.AspNetCore.StaticFiles (1.0.0)
    - ▷ 🗗 Microsoft.EntityFrameworkCore.SqlServer (1.0.0)
    - ▷ 🗗 Microsoft.EntityFrameworkCore.SqlServer.Design (1.0.0)
    - ▷ 🗗 Microsoft.EntityFrameworkCore.Tools (1.0.0-preview2-final)
    - ▷ 🗗 Microsoft.Extensions.Configuration.EnvironmentVariables (1.0.0)
    - ▷ 🗗 Microsoft.Extensions.Configuration.Json (1.0.0)
    - ▷ 🗗 Microsoft.Extensions.Configuration.UserSecrets (1.0.0)
    - ▷ 🗗 Microsoft.Extensions.Logging (1.0.0)

Solution Explorer | Team Explorer

Commands and tools

Pre-release versions of ASP.NET Core included commands that could be configured in the `project.json` configuration file and triggered from the Visual Studio IDE. The following screenshot show the **commands** section of the configuration file:
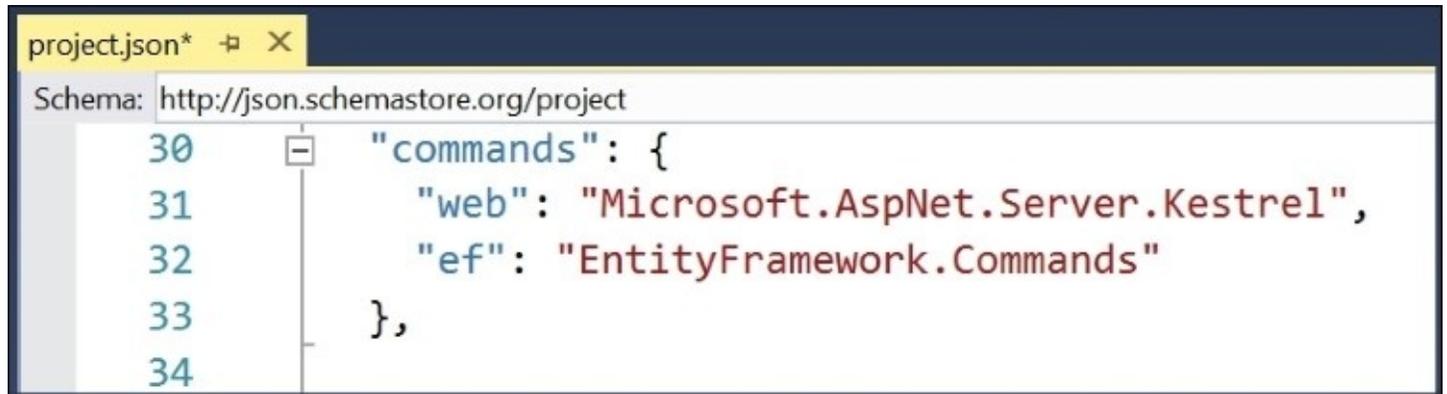
```
project.json*  ⊞  ✕
Schema: http://json.schemastore.org/project
    30      ⊟      "commands": {
    31                  "web": "Microsoft.AspNet.Server.Kestrel",
    32                  "ef": "EntityFramework.Commands"
    33              },
    34
```

The **commands** section is no longer supported in ASP.NET Core 1.0. Instead, there is a new `tools` section in the `project.json` file, which can be used to specify packages containing tools for your project.

Bundling and publishing

Finally, we have sections to include specific components for publishing when it's time to deploy your application. The following screenshot shows the **publishOptions** section, expanded to display the components to be included:
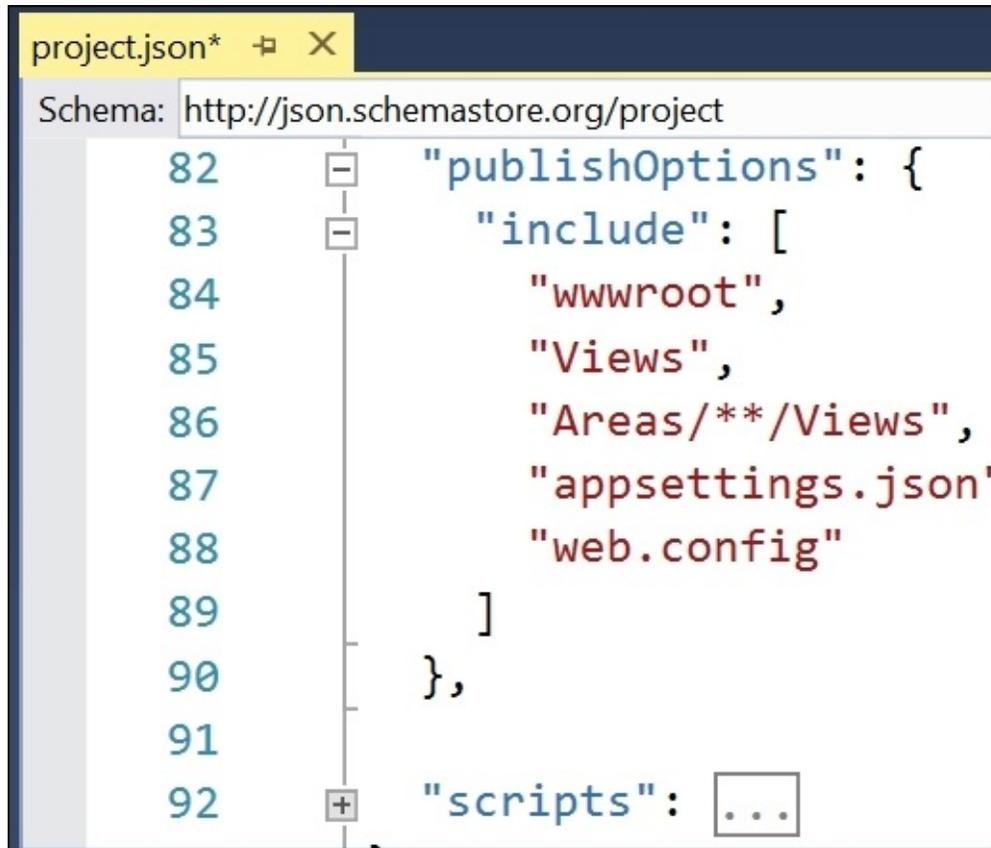
```
project.json*    X
Schema: http://json.schemastore.org/project
82        "publishOptions": {
83            "include": [
84                "wwwroot",
85                "Views",
86                "Areas/**/Views",
87                "appsettings.json"
88                "web.config"
89            ]
90        },
91
92        "scripts": |...|
```

Summary

In this chapter, we've taken a look at the basic structure of an ASP.NET Core 1.0 web application. We delved deeper into the parts that make up the application, including the all-new configuration files. We also covered the basics of MVC and its components.

In the next chapter, we will take a deep dive into ASP.NET Core MVC. We will build a sample application from scratch to illustrate the features of models, views, and controllers in an ASP.NET Core MVC web application.

Chapter 3. Understanding MVC

In previous chapter, we covered a quick refresher of MVC and its components in ASP.NET. In this chapter, we will go deeper into MVC in ASP.NET Core, which is MVC 6 while in beta.

Two common questions from developers are: *How do I upgrade my older MVC projects to ASP.NET Core MVC? Is there an automatic migration process?* Migrating from MVC 5 to the new MVC in ASP.NET Core involves a few manual steps, since there is no automatic migration process. You can copy all of your static client-side files into the `wwwroot` location and adjust any references to these files to refer to the correct location.

For server-side code, you can migrate over your models, views, and controllers without many changes. This chapter will cover what you need to know about each of the following areas:

* Controllers
* Views
* Models

Parts of this chapter will be familiar to developers who have already worked with previous versions of ASP.NET MVC. We will go through some familiar material, while revealing newer additions to MVC along the way.

Building controllers

Your MVC controller is where the magic happens. Requests come in from the end-user, then content and data get returned. What happens in between is up to you-the developer.

Controller methods

By default, controller methods can be used for HTTP `GET` requests using the `[HttpGet]` attribute or omitting this action verb attribute altogether, as it is the default behavior. Most likely, you have already been using the following HTTP `GET` and `POST` verbs:

- `HttpGet`: Uses the HTTP `GET` method with optional `querystring` parameters
- `HttpPost`: Uses the HTTP `POST` method for form submissions to create an entity

In addition to the preceding, you should also be aware of additional HTTP verbs that can be used as controller attributes; they are as follows:

- `HttpPut`: Uses the HTTP `PUT` method to edit an existing entity
- `HttpDelete`: Uses the HTTP `DELETE` method to delete an existing entity
- `HttpPatch`: Allows partial model updates instead of a full `PUT` request
- `AcceptVerbs`: Allows multiple action verbs to be specified

Basic controllers

Using Visual Studio, you can add a basic controller and gradually add code to it. You can also add a more complete controller with scaffolding, which includes model binding and action methods to manipulate the model. But we'll cover that later in this chapter.

First, let's create a sample project in which we will add a basic controller, as follows:

1. Create a new ASP.NET web project by clicking on **File** | **New** | **Project**.
2. Select **ASP.NET Web Application (.NET Core)**.
3. Name your project *PatientRecords*.
4. Select **Web Application** under the list of **ASP.NET Core Templates**.
5. Click **Change Authentication** to switch to **Individual User Accounts**.
6. Click **OK** to create your web project.

We could have started with an **Empty** template as well, but then we would have to add a lot more dependencies and configuration just to get things up and running. Let's proceed with adding our basic controller.

To add a basic controller, follow these steps:

1. In Solution Explorer, right-click the **Controllers** folder.
2. In the context menu, click **Add** | **New Item**.
3. In the **Add New Item** dialog, select **MVC Controller Class** under .NET Core and ASP.NET.
4. Name your controller `PatientController` and click **Add** to proceed.

The following screenshot shows the **Add New Item** window:
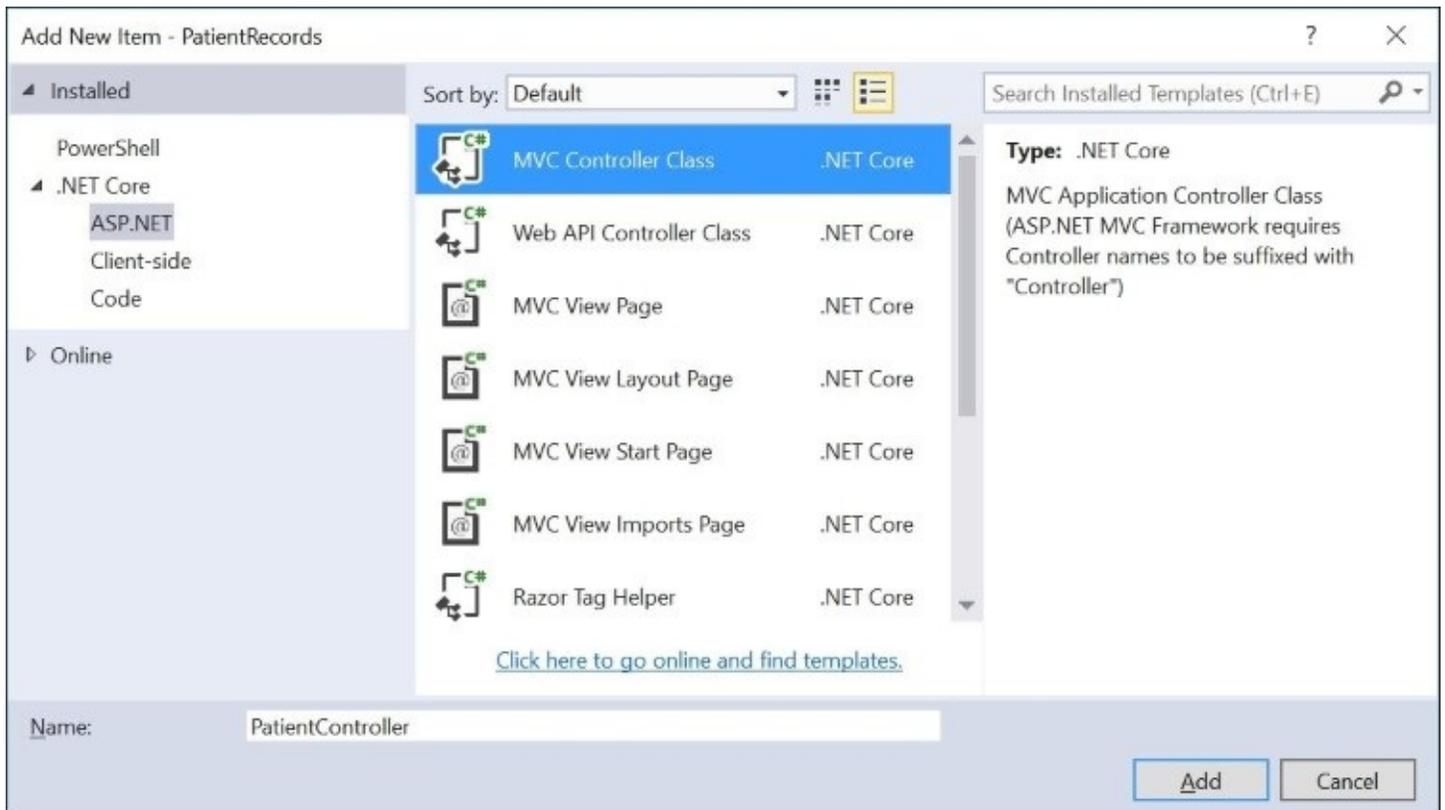
At this point, you should have a basic controller with one `Index()` method that returns a simple `View`. This should already be familiar to experienced ASP.NET MVC developers. If we run the project now, the web application should launch in a web browser, using an arbitrary port number such as `12345`. Your port number will vary. Your link for your web application will be something like `http://localhost:12345`.

With your browser still open, you could try to access the `PatientController` by adding the controller name `Patient` to the end of the URL: `http://localhost:12345/Patient`.

This should result in an unhandled `InvalidOperationException`, since we haven't added an `Index` view yet for the `Patient` controller. As with previous versions of MVC, this view would typically be located in a subfolder named `Patient`, within the `Views` folder. If it's missing, the fallback view would be located in a `Shared` subfolder with the `Views` folder. The exception occurs when neither of these conventional locations contain the expected view.

To bypass the expected route, let's replace the `Index` method of the `PatientController` class with the following code:

```
public string Index()
{
    return "Patient Info";
}
```

Note that we've changed the return type to `string` so that we can return a string literal to test

our controller. If you run the application again, you should now be able to see the placeholder text in your web browser when you access the `Patient` controller at `http://localhost:12345/Patient`:

URL routes and conventions

If you take a look at the `Configure()` method in `Startup.cs`, you can see the default route for your web application:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

You may recall from Chapter 2 , *Building Your First ASP.NET Core Application,* that the `Configure()` method in ASP.NET Core is used for applying components and services that are added in the `ConfigureServices()` method. As you can see from the sample code, the controller and action values are being defaulted to `Home` and `Index` respectively.

Optionally, there is an `id` parameter that can be added to the URL after a trailing slash. Note that this is different from adding an `id` parameter to the `QueryString`, but both approaches can be used to pass the value to the controller method through method parameters.

In order to use the `Encode()` method, make sure that you have added the following `using` statement to include the `WebEncoders` namespace in your code:

```
using System.Text.Encodings.Web;
```

In order to make use of the default convention, update the `Index` method in the `PatientController` class as follows to accept parameters:

```
public string Index(int id, string name = "Unknown")
{
    return "Patient Info: " + HtmlEncoder.Default.Encode(
        "ID: " + id + ", " +
        "Name: " + name);
}
```

The `id` and `name` parameters refer to a patient's ID and name, respectively. The `name` parameter is being defaulted to `"Unknown"` in case no name has been specified. The passed values are wrapped around a call to `Encode()` to ensure that no malicious tags or scripts are being passed through the `QueryString`.

You should now be able to use `http://localhost:12345/Patient?id=1&name=John` as the `QueryString` parameters. The following screenshot is the output for the same:

Patient Info: ID: 1, Name: John
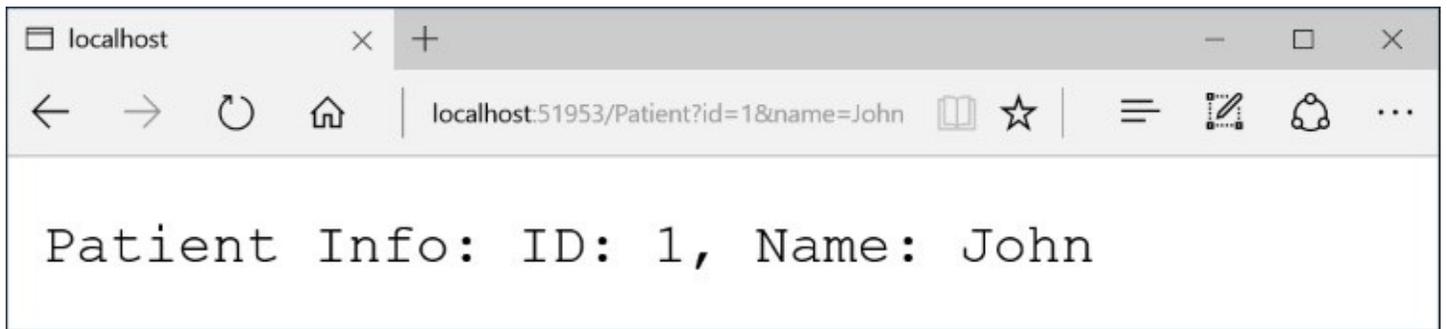
Implementing views

Views in MVC make up the presentation layer, the user interface that holds on-screen elements for the user to interact with. The UI in any web application can get busy really quickly, either filled with too many items at the same time, or burdened with business logic and unnecessary code. The goal of MVC's *separation of concerns* aims to help developers create well-architected applications where the views are lightweight and validation is performed in the model layer, which is connected to the view with binding.

Basic views

Similar to how we added a basic controller, we will create a basic view and gradually add code to it. Just as you can create a more complete controller with scaffolding, you can also create a more complete view attached to a template and model. But once again, we'll cover that later in this chapter.

First, let's create a subfolder for views that will be used by our `Patient` controller, so that we can add our new view to it. Follow the these steps:

1. In Solution Explorer, right-click the `Views` folder, and add a new folder to it.
2. Name the subfolder `Patient` to take advantage of naming conventions.
3. Right-click the `Patient` subfolder, then click **Add** | **New Item**.
4. In the **Add New Item** dialog, select **MVC View Page** under **ASP.NET**.
5. Name your view `Index.cshtml` and click **Add** to proceed.

We are using the default view name of `Index.cshtml` so that this view will be returned by the `Index()` controller method in the `Patient` controller. At this point, the view has no actual content, just some placeholder symbols for comments and server-side code. The code is as follows:

```
@* server-side comments *@
@{ // server-side code }
```

Replace the contents of the view with the following content:

```
@{
    ViewData["Title"] = "Patient Index";
}
<h2>Patient Index</h2>
```

The `ViewData` value for `"Title"` is used by the default `_Layout.cshtml` layout file, which can be found in the `Shared` subfolder in your `Views` folder. The following code is for `Patient` controller's `Index` view:

```
<title>@ViewData["Title"] - PatientRecordsWebApp</title>
```

Replace the `Patient` controller's `Index` view with the following code:

```
public IActionResult Index()
{
    return View();
}
```

To browse to this page, run the application and add the `Patient` controller name to the end of the URL after a trailing slash, like `http://localhost:12345/Patient`.

You should see the contents of the view in your browser now, including the title and page

header text. This is shown in the following screenshot:
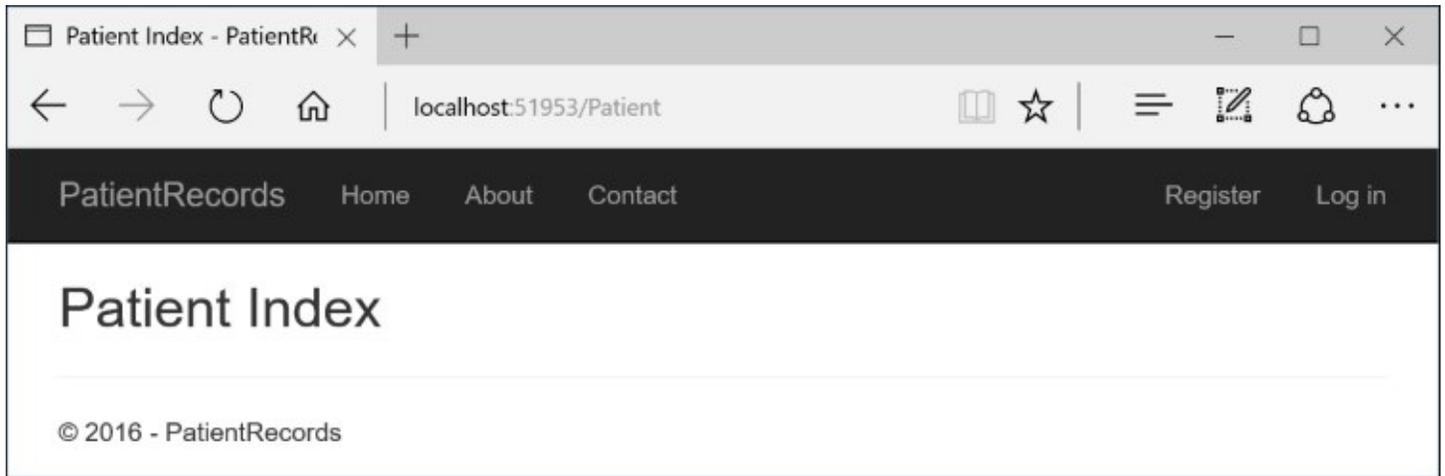
Tag helpers in views

To make it easier to browse to this page, let's update its layout file to include a link to the `controller` method that will return this view. First, locate the set of clickable links in the layout file, `<li>` list items within a `<ul>` unordered list.

Note that the `<a>` tags each have two attributes that act as tag helpers:

- `asp-controller`
- `asp-action`

Tag helpers were newly introduced in ASP.NET Core MVC, and are similar to `HtmlHelpers` you may have used in the past. The values of these two attributes will determine the controller name and the subsequent action method that will be triggered upon clicking the link.

Let's add an additional clickable link for the `Patient` controller, just after the `Contact` link, using the following code:

```
<li><a asp-controller="Patient" asp-action="Index">Patients</a></li>
```

Run the web application to see the new link in the top header area, which will allow you to jump to the patient's index view much more quickly:
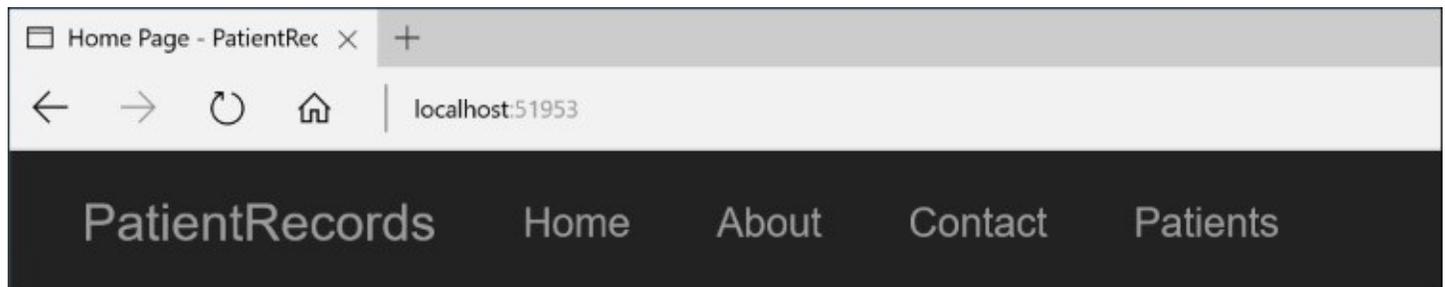
Later in this chapter, we will learn how to use tag helpers in your views for validation purposes. We will also add specific attributes to our models to help validate specific fields when a form is submitted.

ViewData, ViewBag, and TempData

You can pass data from your controllers to your views by setting the values using `ViewData` items in a controller, and then retrieving the values in a view. These values are stored as key-value dictionary items with string-based keys. `ViewBag` is a more dynamic alternative to using `ViewData`, allowing you to use complex data types without typecasting. The assigned value will be reset upon a redirect.

The following are two examples on how to use `ViewData` and `ViewBag` in your controllers:

```
ViewData["PatientId"] = id;
ViewBag.PatientData = "someData";
```

`TempData` is a little different, as it retains its data during a redirect, which allows you to pass data between controllers during a redirect operation. Its syntax is similar to that of `ViewData` so it can be used as a dictionary item with the following string-based key:

```
TempData["UserToken"] = userTokenData;
```

Let's add the following `Details()` method in our `Patient` controller to assign some values to some `ViewData` objects:

```
public IActionResult Details(int id, string name = "Unknown")
{
    ViewData["PatientId"] = id;
    ViewData["PatientName"] = name;
    return View();
}
```

Here, we are setting two `ViewData` values for the patient's ID and name, respectively. To display the data in a view, we will add a new `Details` view as follows:

1. In Solution Explorer, expand the `Views` folder.
2. Right-click the `Patient` subfolder, then click **Add** | **New Item**.
3. In the **Add New Item** dialog, select **MVC View Page** under **ASP.NET**.
4. Name your view `Details.cshtml` to proceed.

Replace the contents of the `Details` view with the following code:

```
@{
    ViewData["Title"] = "Patient Details";
}
<h2>Patient Details</h2>
<ul>
    <li>ID: @ViewData["PatientId"]</li>
    <li>Name: @ViewData["PatientName"]</li>
</ul>
```

In order to provide an easy link between the `Index` view and the `Details` view, let's also

update the `Index` view located in the `Patient` subfolder, inside the `Views` folder.

Replace the contents of the `Patient` controller's `Index` view with the following code:

```
@{
    ViewData["Title"] = "Patient Index";
}
<h2>Patient Index, with Tag Helpers</h2>

<ul>
    @for (int i = 0; i < 10; i++)
    {
        <li><a asp-controller="Patient" asp-action="Details"
               asp-route-id="@i" asp-route-name="Patient @i">
               Patient # @i</a></li>
    }
</ul>
```

In the preceding code, the `for` loop is used to generate a list of clickable links. Each link can be clicked to show the details for a particular patient. Once again, you can see the use of tag helpers as the following attributes within the <a> tag:

- `asp-controller`
- `asp-action`
- `asp-route-id`
- `asp-route-name`

The last two attributes are open-ended enough to reference named parameters such as ID and name. The `id` attribute will automatically follow the URL routing convention, while the `name` attribute will be added as a QueryString parameter.

The links will appear in the following format:

- `http://localhost:12345/Patient/Details/0?name=Patient%200`
- `http://localhost:12345/Patient/Details/1?name=Patient%201`
- `http://localhost:12345/Patient/Details/2?name=Patient%202`

The numeric value just before the question mark is the ID value as defined by the URL route defined for the application. Note that the space in the parameter value is URL-encoded to a `%20`.

Run the web application and click on the `Patients` link in the top toolbar to navigate to the `Patient` index page. You should see a list of 10 items in the following screenshot, each with its own link. Click one of the links to go to the `Details` page for that particular patient. The details page will display the ID and name of the selected patient:

Designing models and ViewModels

Instead of passing around individual values one by one from controllers to views, you can use a model to store a set of data. The controller is responsible for updating the model, which can be associated with a view to get its data.

Creating models

A model is just a class file with a `.cs` file extension. In the Solution Explorer panel, you can add new model class files to the `Models` folder. You may have noticed that there are choices available for MVC `Controller` class and MVC `View` page, but none for the `Model` class.

To create a model class, we can select the **Class** option when adding a new item as follows:

1. In Solution Explorer, right-click the `Models` folder, and click **Add** | **New Item**.
2. In the `Add New Item` dialog, select **Class** under `Code`.
3. Name the file `Human.cs` to create a new model.
4. Verify that you have an empty class named `Human`.

Add the following fields to the `Human` class:

```
public class Human
{
    public int ID { get; set; }
    public string SocialSecurityNumber { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

You may have also noticed in the following screenshot that there are `ViewModels` subfolders within the `Models` folder. When you bind a view to a model, the fields defined in the model represent the UI elements and form fields in the associated view. These subfolders are a great place to organize these models, whose names match your controller names:
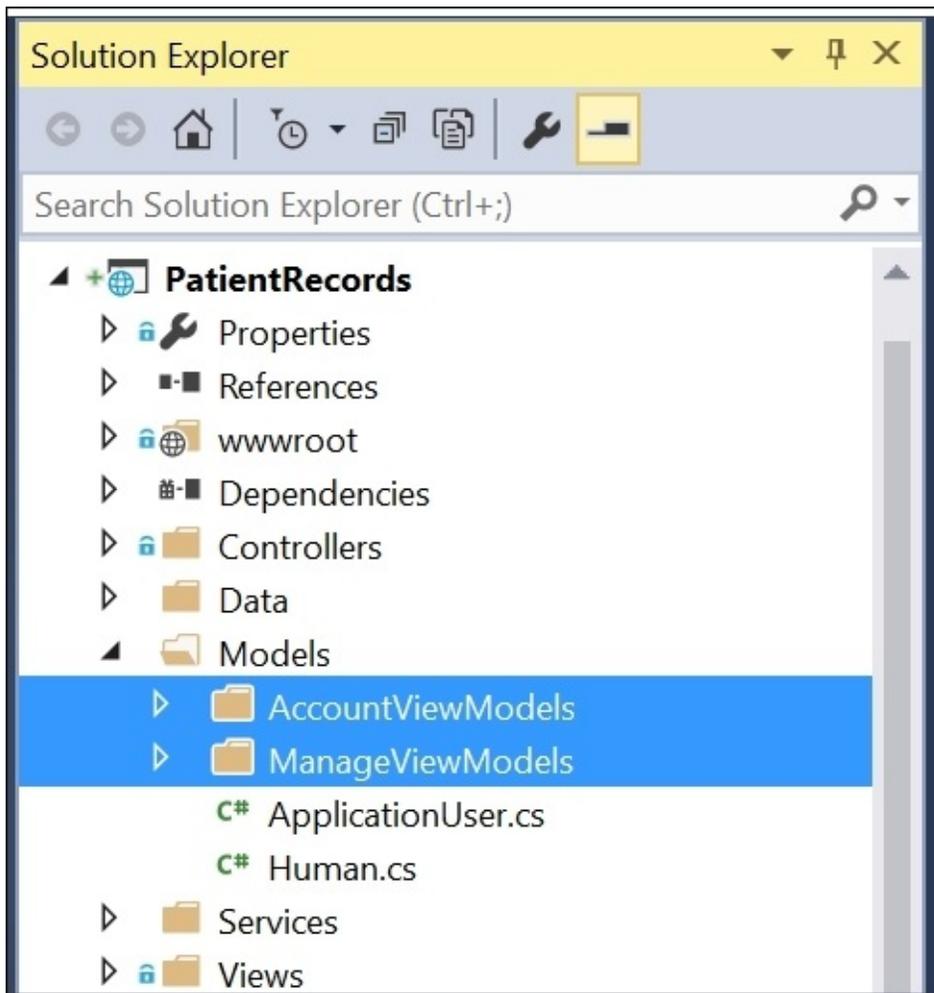
Binding models to views

To bind a view to a model, the following syntax can be seen inside a view's `.cshtml` file:

```
@model MyModelName
```

The rest of the view should contain a server-side form, with form elements associated with the model's fields. While it is possible to have more than one form in a view, MVC views typically tend to have one form if multiple forms can be avoided. In any case, the model used for the view should contain all the necessary fields to cover all the forms in the view.

In previous versions of ASP.NET MVC, you may have used the following syntax for a server-side form:

```
@using (Html.BeginForm("action", "controller", FormMethod.Post, new {}))
{
    @Html.LabelFor(m => m.Field1)
    @Html.TextBoxFor(m => m.Field1)

    @Html.LabelFor(m => m.Field2)
    @Html.TextBoxFor(m => m.Field2)

    <input type='Submit' value='Submit' />
}
```

In ASP.NET Core MVC, the following syntax has been introduced, to use tag helpers:

```
<form asp-controller="controller" asp-action="action" method="post">
    <label asp-for="Field1"></label>
    <input asp-for="Field1" />
    <label asp-for="Field2"></label>
    <input asp-for="Field2" />
    <input type="submit" value="Submit" />
</form>
```

This new approach makes the code cleaner and more efficient as well. When the form is submitted, the controller methods can accept a model's data type as a parameter to receive all its fields:

```
[HttpPost]
public IActionResult MyAction(MyModelName model)
{
    return View(model);
}
```

ViewModels and mapping

When you organize your `ViewModels` in their own folders, you would typically bind each view to a specific `ViewModel`. But your database design may require you to have entities defined in your code as model files that match your database objects. You'll need to figure out how to map your database-specific models to your `ViewModels`.

There are a couple of different ways for you to map your `ViewModels` and database entity models:

- When your controller methods accept a `ViewModel`, you could map each field to their corresponding entity model field, one by one
- You could use a library such as `AutoMapper`, which is available for free on GitHub/NuGet and described on http://automapper.org as a *convention-based object-based mapper*

To dig deeper into entity models, we will cover *Object-relational mapping and entity framework* in [Chapter 6](#) , *Using Entity Framework to Interact with Your Database in Code*.

Bringing it all together

To bring it all together, let's make use of Visual Studio's built-in code generation features to take advantage of scaffolding and binding. In this section, we will cover field attributes that assist in model validation.

Before we wrap up this chapter, we will also learn about the use exception handling to catch errors in your code.

Scaffolding, validation, and model binding

Using the `PatientRecords` project we've built so far, let's add a new controller to it following the following steps, to make use of scaffolding. Instead of clicking on **Add Item**, we will choose the **Controller** option:

1. In Solution Explorer, right-click the `Controllers` folder.
2. In the context menu, click **Add** | **Controller**.
3. Select the option to add a new MVC controller with views, using EF.

You will be prompted for additional information while adding your controller. Select/enter the following information:

- **Model class**: `Human`, from your models namespace
- **Data context class**: `ApplicationDbContext`
- **Controller name**: `HumanController`

For all the following checkboxes, you may leave the defaults as they are:

- **Generate views**: checked by default, used to generate corresponding views for each action.
- **Reference script libraries**: checked by default, includes references to script libraries in the client-side code.
- **Use a layout page**: checked by default, allows entry of layout name or can be left empty to allow `_viewstart.cshtml` to be used. This page typically includes a reference to the project's default layout page, set to `_Layout` by default.

You may have noticed that the controller name defaults to `HumenController` with an e in it. This is because the scaffolding tool attempts to pluralize entities in the code, so it incorrectly assumes that the plural form of *Human* is *Humen*. To fix this, make sure you rename it `HumanController` with an a before you add the controller.

Once the scaffolded controller is added, you can check the contents of your `HumanController` to verify that it has auto-generated some code for you. It should start with a database context object, followed by a constructor that makes use of the context.

The rest of the class should have a series of action methods for basic CRUD operations, that is, operations used to create, read, update, and delete entries:

- **Index**: list of `Human` objects
- **Details(id)**: details of specific `Human` object
- **Create**: GET/POST methods to display a creation form and create a new entity
- **Edit**: GET/POST methods to display an edit form and edit an existing entity
- **Delete**: GET/POST methods to display a confirmation screen and perform a delete operation

Take a look at the `Views` folder and verify that there is a `Human` subfolder with separate views for all of the preceding action methods: create, delete, details, edit, and index. If you open any of these `.cshtml` files, you will find a reference to each corresponding model in the first line of each view.

The validation included in these views use the new tag helper syntax. These tags can be used for validation, which we saw in the preceding section. To build upon this, we can edit the `Human` model to include a few attributes to aid in validation.

The attributes you will need are defined in the `DataAnnotations` namespace, so you should ensure that the following using statement is added to your `Human.cs` file:

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

Edit the `Human.cs` file in the `Models` directory to include the following attributes for your fields:

```
public class Human
{
    public int ID { get; set; }
    [Required]
    [StringLength(11)]
    [Display(Name = "SSN")]
    public string SocialSecurityNumber { get; set; }
    [Display(Name = "DOB")]
    [DataType(DataType.Date)]
    public DateTime DateOfBirth { get; set; }
    [Display(Name = "First Name")]
    public string FirstName { get; set; }
    [Display(Name = "Last Name")]
    public string LastName { get; set; }
}
```

The following attributes have been used here:

- `Required`: indicates that this field is required
- `StringLength(n)`: limits the field's value to n characters
- `Display(Name ="Alt Name")`: an alternate name to display in the UI
- `DataType(DataType.Date)`: limits the field to a specific data type, for example, date

Note

If you would like to experiment with additional attributes, refer to the ASP.NET documentation to look for other validation attributes in this namespace.

Before we launch the web app again, let's add a new link to the `Human` controller's `Index` view. We can add another clickable link to the top menu defined in the shared layout page, `_Layout.cshtml` in the `Views/Shared` directory:

```
<li><a asp-controller="Human" asp-action="Index">Humans</a></li>
```

The preceding line can be added right after the `Patients` link we added earlier. If you click on the `Humans` link in the top toolbar at this time, you will get an error because our database doesn't exist yet.

Database setup and data entry

To set up the initial database, we will have to run the initial migration from a command prompt. The current folder must be set to the project's root location, where `Startup.cs` resides. This is the same location where your `Models`, `Views`, and `Controllers` folders exist.

Once you open a command prompt to your folder's location, you may run the following commands. Your DNVM version may vary:

```
dotnet restore
dotnet build
dotnet ef migrations add Initial
dotnet ef database update
```

The preceding commands are responsible for the following:

- `restore`: restores packages for your project
- `build`: compiles and builds your project
- `ef migrations add Initial`: creates an initial snapshot of your database models
- `ef database update`: creates/updates your database

Visual Studio should install the **dotnet** tools for you, so you may refer to Chapter 1 , *Getting Started with ASP.NET Core,*for initial setup instructions if you are missing anything. For deeper coverage of Entity Framework, see Chapter 6 , *Using Entity Framework to Interact with Your Database in Code.*

Once the commands are done running, you should have an initial version of your database, with the necessary database tables that match your entity models, for example, a `Human` table. The columns in the `Human` table should match the fields in the `Human.cs` model file.

To view the SQL Server Object Explorer in Visual Studio, select **View** | **SQL Server Object Explorer** from the top menu, as shown in the following screenshot:
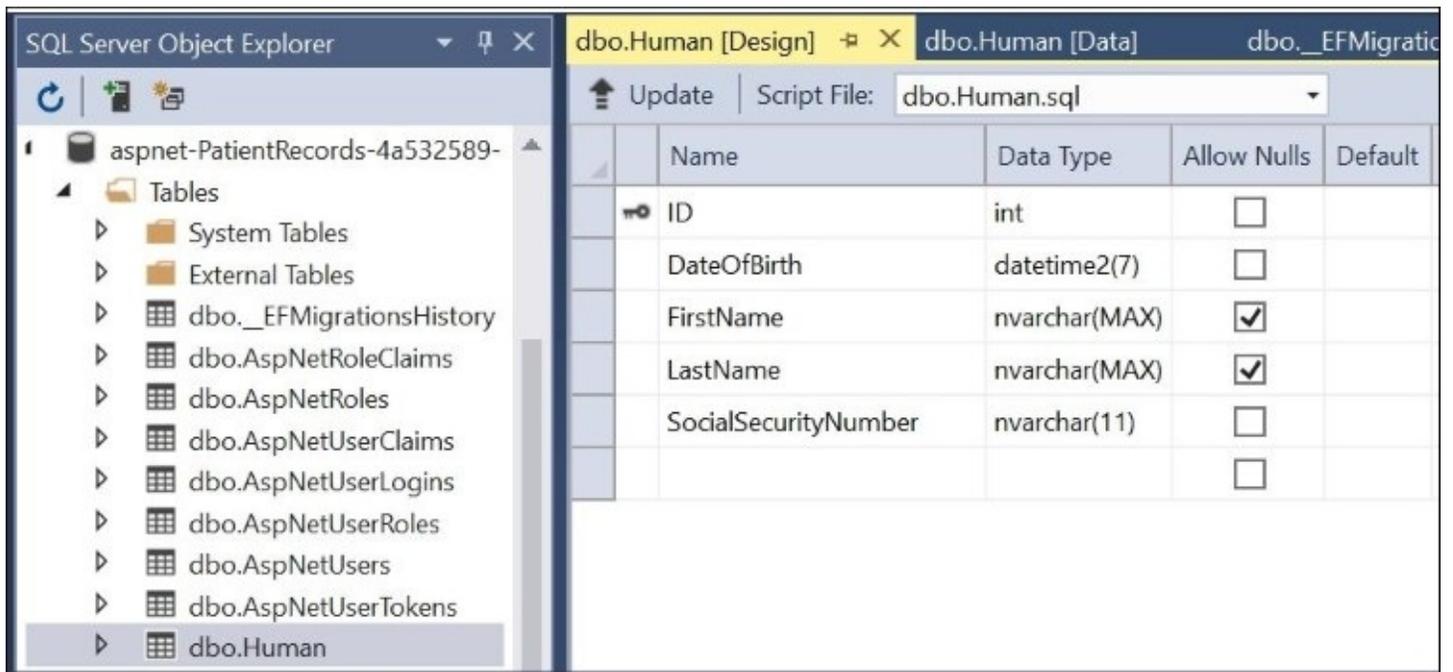
Now, you are ready to run the application again and click on the `Humans` link the top toolbar. This should take you to the `Index` view, showing you that there are no `Humans` yet, as shown in the following screenshot:
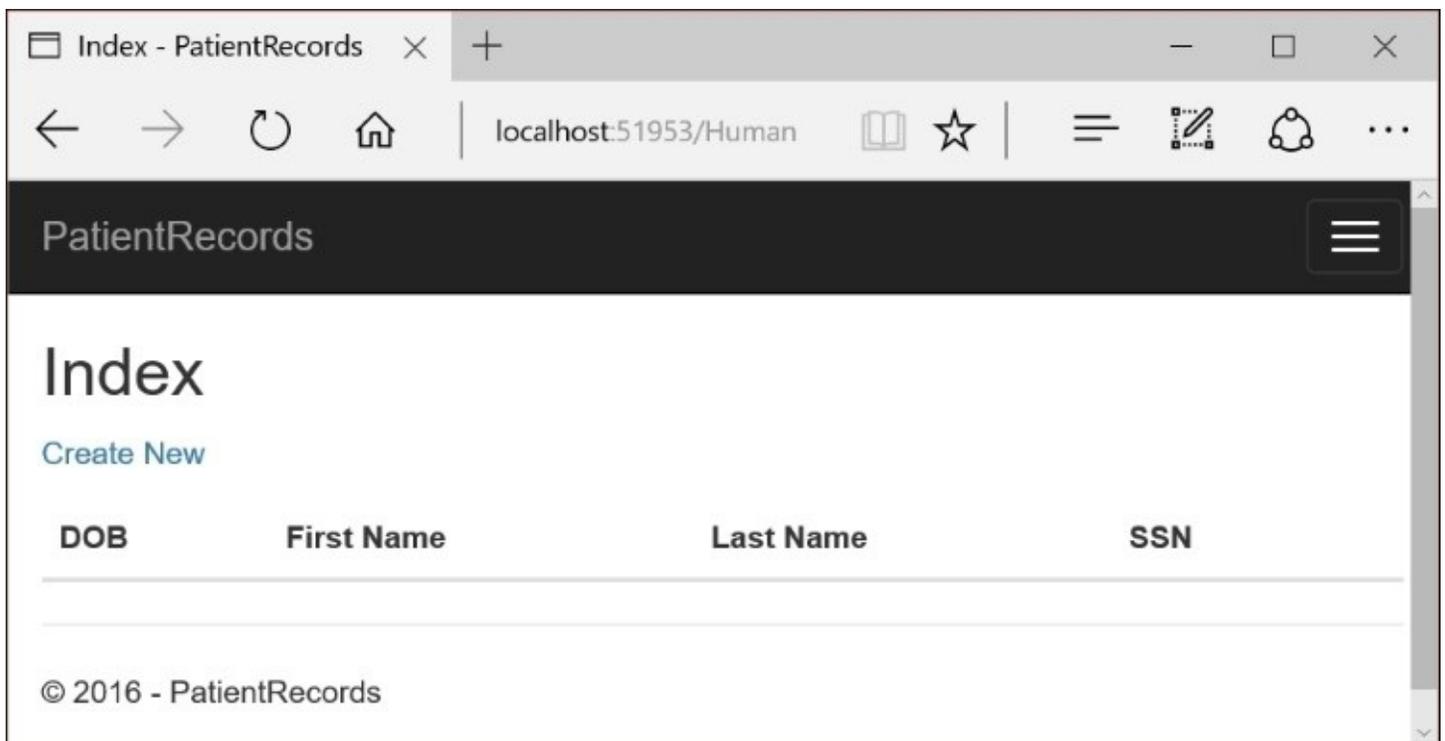
Click the **Create New** link to create a new `Human` entry. Clicking this link should take you to

the `Create` view of the `Human` controller. Enter some values and click the **Create** button, as shown in the following screenshot:
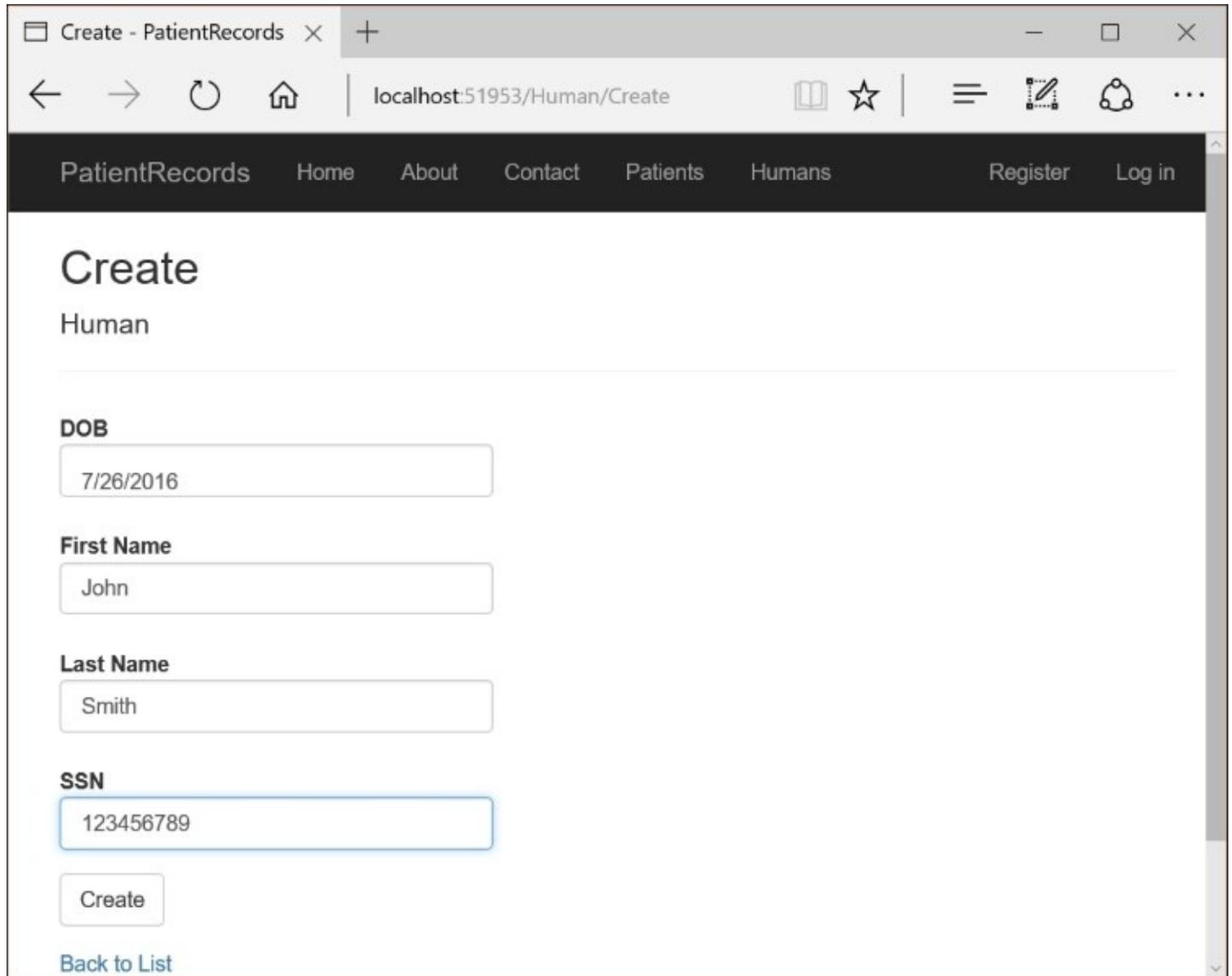
Once the new `Human` entry is created, your submission will be processed by the post-version of the `Create` method. If you leave out any required values or enter the wrong number of characters, the validation messages will automatically display in the UI. Once submitted correctly, you will be taken back to the `Index` view, where you should see the new `Human` entry you just created, as shown in the following screenshot:

Exception handling

In ASP.NET Core MVC, exception handling has evolved and there are new ways to handle exceptions. The use of `UseDeveloperExceptionPage()` and `UseExceptionHandler()` can be seen in ASP.NET web apps. To find the location of these method calls, look no further than your `Startup.cs` file. Inside the `Configure()` method, you may find calls to the following methods:

```
app.UseDeveloperExceptionPage();
app.UseExceptionHandler("/Home/Error");
```

The first method call is relevant when you are running the application in development, so that you can get additional debug information. When in production, the actual exception handler will be used to redirect to the appropriate error page. The preceding code triggers the `Home` controller's `Error` action method and subsequently, its `Error` view. In our sample application, there is an `Error.cshtml` file in the `Shared` folder that will be used in case of an exception in production.

In order to simulate development and production environments, you can change the value of the environment variable `ASPNETCORE_ENVIRONMENT` to `Development` or `Production`. This change be changed in your `launchSettings.json` file or in the **Debug** tab of your project's properties screen.

As with previous versions of ASP.NET, you should use try/catch blocks to anticipate various exceptions that may occur at runtime and handle them appropriately. In some cases, you may have to throw an exception in your application code to let the calling method handle the exception.

Summary

In this chapter, we've taken a look at how models, views, and controllers come together to form a working ASP.NET Core MVC web application. We touched on database setup and wrapped up with exception handling.

In the next chapter, we will learn about Web APIs and how you can go beyond traditional web applications with ASP.NET. We will take a look at various clients that can take advantage of a Web API backend.

Chapter 4. Using Web APIs to Extend Your Application

If you've used the ASP.NET Web API with prior versions of ASP.NET, you may recall that a Web API controller typically inherits from the `APIController`. This has changed in the ASP.NET Core, as part of the unification of MVC and the Web API. As a result, each Web API controller inherits from the `Controller` class, which is also the base class for each MVC controller.

In this chapter, we will work through a working sample of a Web API project. During this chapter, we will cover the following:

- Web APIs
- Web API configuration
- Web API routes
- Consuming Web API applications

Understanding a Web API💬

Building web applications can be a <u>rewarding</u> experience. The satisfaction of reaching a broad set of potential users can trump the frustrating nights spent fine-tuning an application and fixing bugs. But some mobile users demand a more streamlined experience that only a native mobile app can provide.

Mobile browsers may experience performance issues in low-bandwidth situations, where HTML5 applications can only go so far with a heavy server-side backend. Enter Web API, with its RESTful endpoints, built with mobile-friendly server-side code.

The case for Web APIs

In order to create a piece of software, years of wisdom tell us that we should build software with users in mind. Without use cases, its features are literally useless. By designing features around user stories, it makes sense to reveal public endpoints that relate directly to user actions. As a result, you will end up with a leaner web application that works for more users.

If you need more convincing, here's a recap of features and benefits of a Web API:

- It lets you build modern lightweight web services, which are a great choice for your application, as long as you don't need SOAP
- It's easier to work with than any past work you may have done with ASP.NET **Windows Communication Foundation** (**WCF**) services
- It supports RESTful endpoints
- It's great for a variety of clients, both mobile and web
- It's unified with ASP.NET MVC and can be included with/without your web application

Creating a new Web API project from scratch

Let's build a sample web application named *Patient Records*. In this application, we will create a Web API from scratch to allow the following tasks:

- Adding a new patient
- Editing an existing patient
- Deleting an existing patient
- Viewing a specific patient or a list of patients

These four actions make up the so-called CRUD operations of our system: to Create, Read, Update, or Delete patient records.

Following the steps below, we will create a new project in Visual Studio 2015:

1. Create a new Web API project.
2. Add an API controller.
3. Add methods for CRUD operations.

The preceding steps have been expanded into detailed instructions with the following screenshots:

1. In Visual Studio 2015, click **File** | **New** | **Project**. You can also press *Ctrl + Shift + N* on your keyboard.
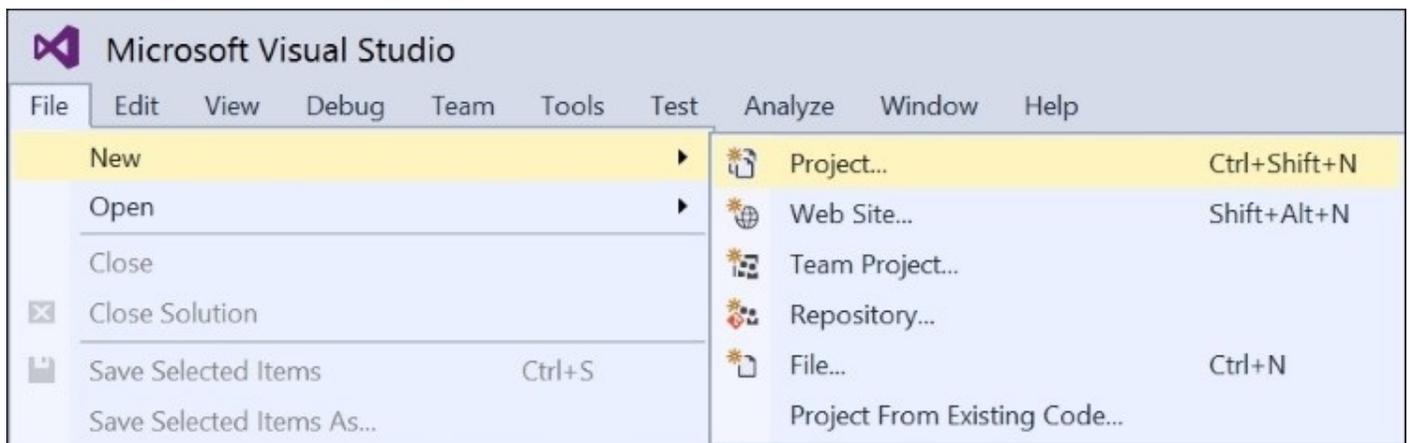
2. On the left panel, locate the **Web** node below **Visual C#**, then select **ASP.NET Core Web Application (.NET Core)**, as shown in the following screenshot:
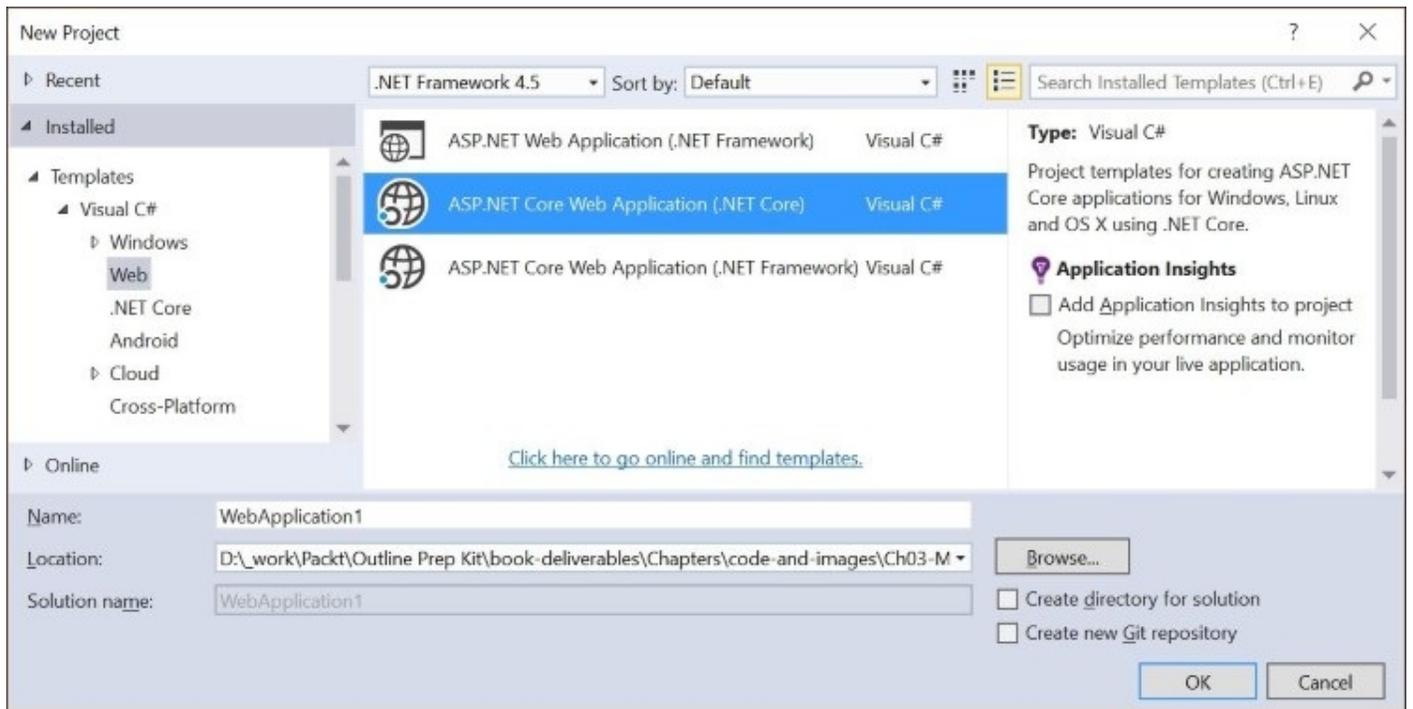
3. With this project template selected, type in a name for your project, for example **PatientRecordsApi,** and choose a location on your computer, as shown in the following screenshot:
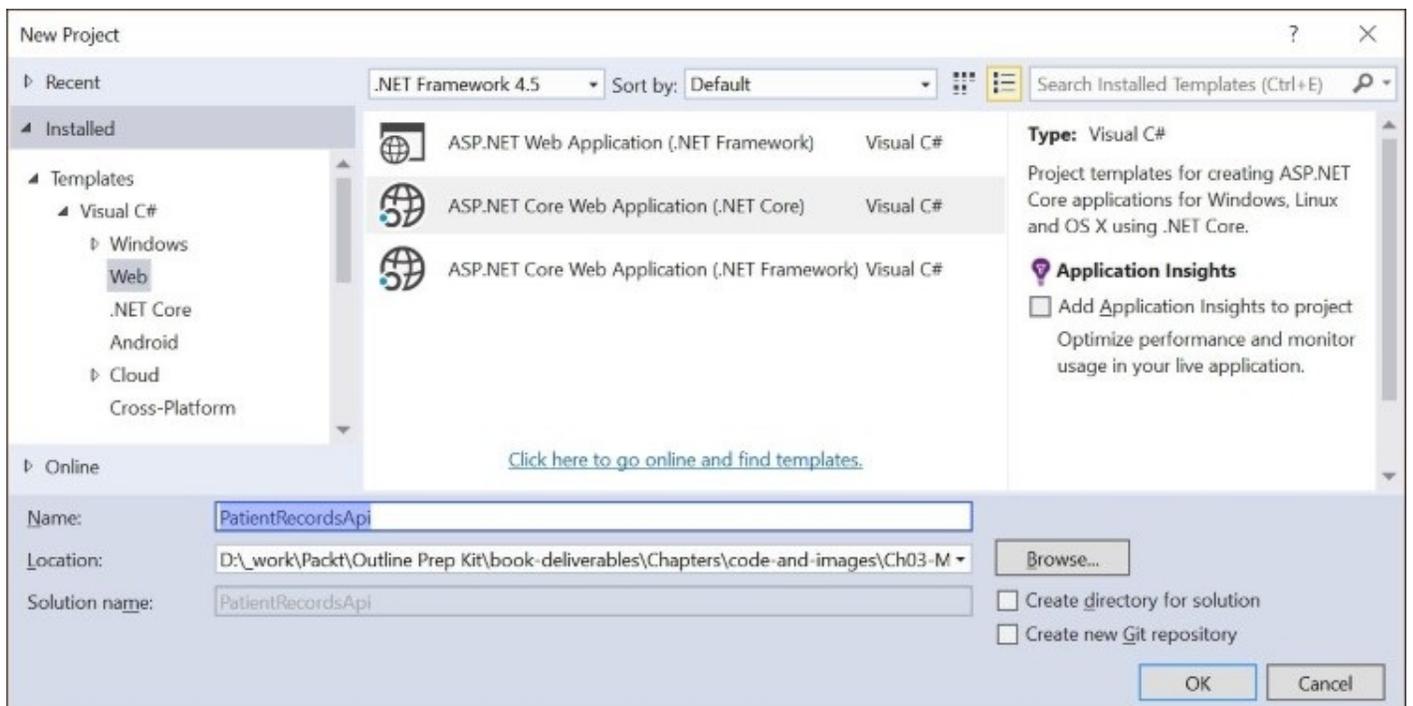
4. Optionally, you may select the checkboxes on the lower right to create a directory for

your solution file and/or add your new project to the source control. Click **OK** to proceed.

5. In the dialog that follows, select **Empty** from the list of the **ASP.NET Core Templates**, then click **OK**, as shown in the following screenshot:
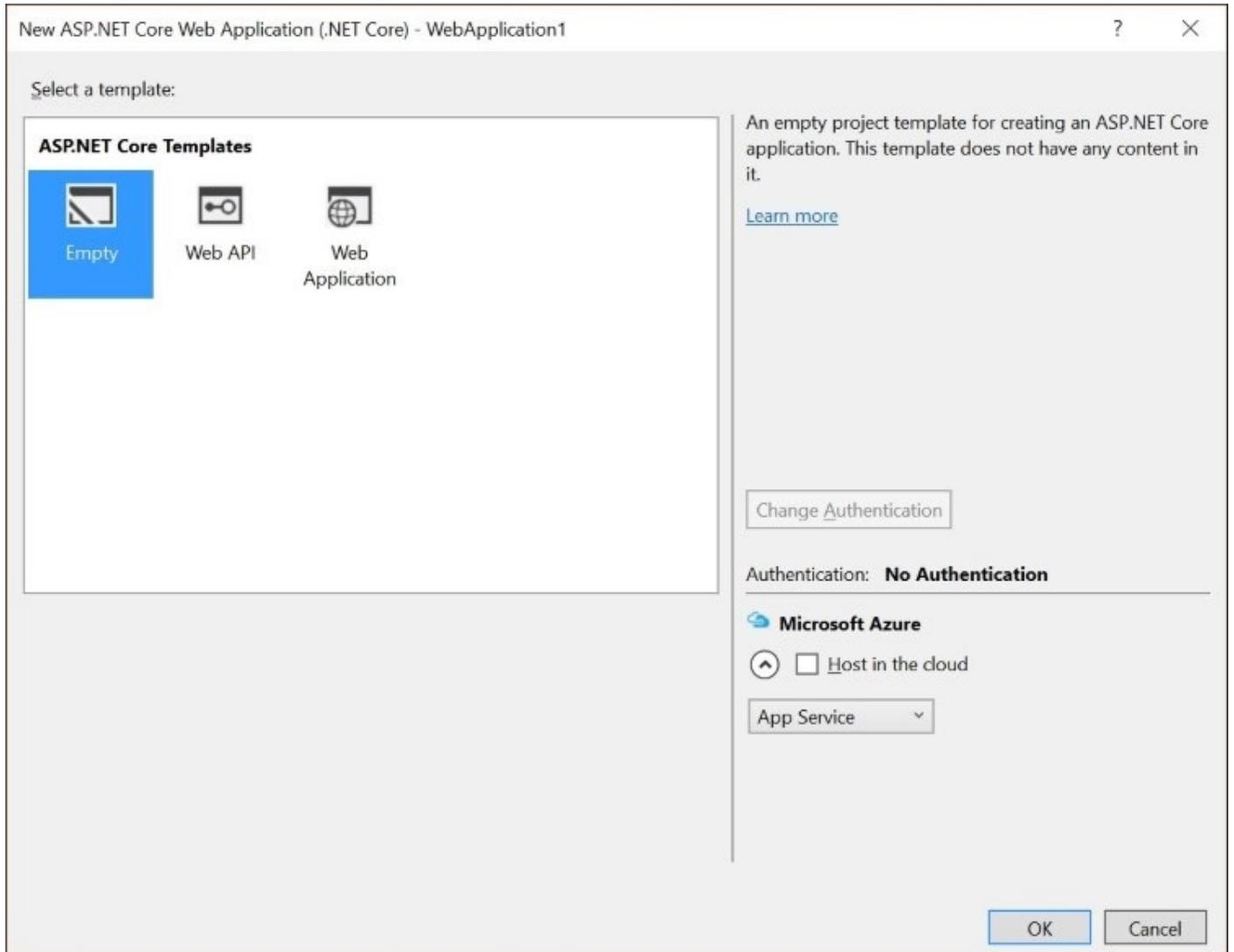
6. Optionally, you can check the checkbox for **Microsoft Azure** to host your project in the cloud. Click **OK** to proceed.

Building your Web API project

In the Solution Explorer, you may observe that your **References** are being restored. This occurs every time you create a new project or add new references to your project that have to be restored through **NuGet**, as shown in the following screenshot:
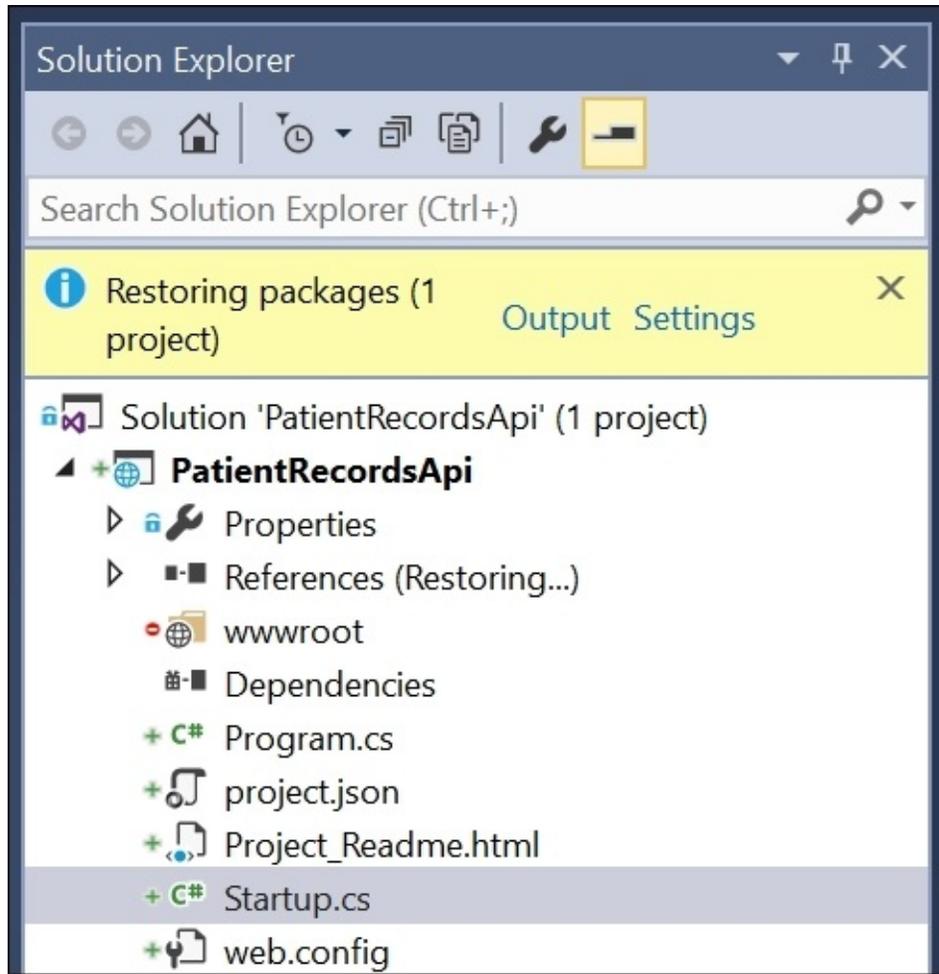
Follow these steps, to fix your references, and build your Web API project:

1. Right click on your project, and click **Add** | **New Folder** to add a new folder, as shown in the following screenshot:
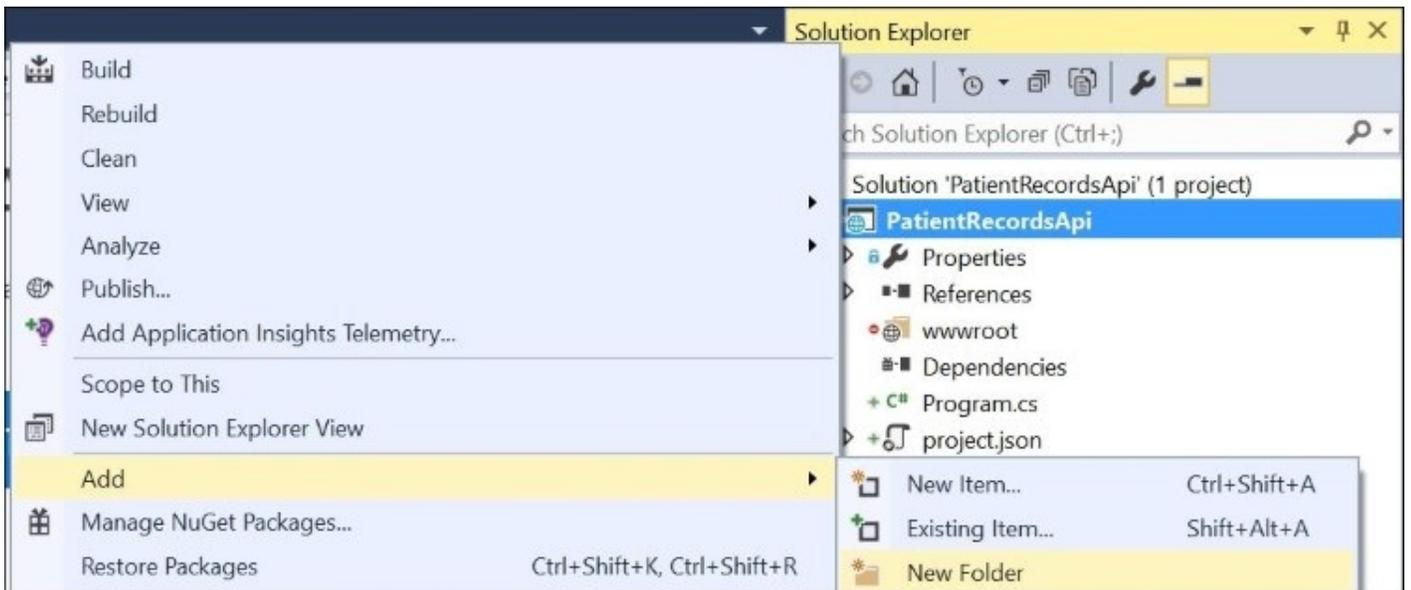
2. Perform the preceding step three times to create new folders for your **Controllers**, **Models**, and **Views**, as shown in the following screenshot:
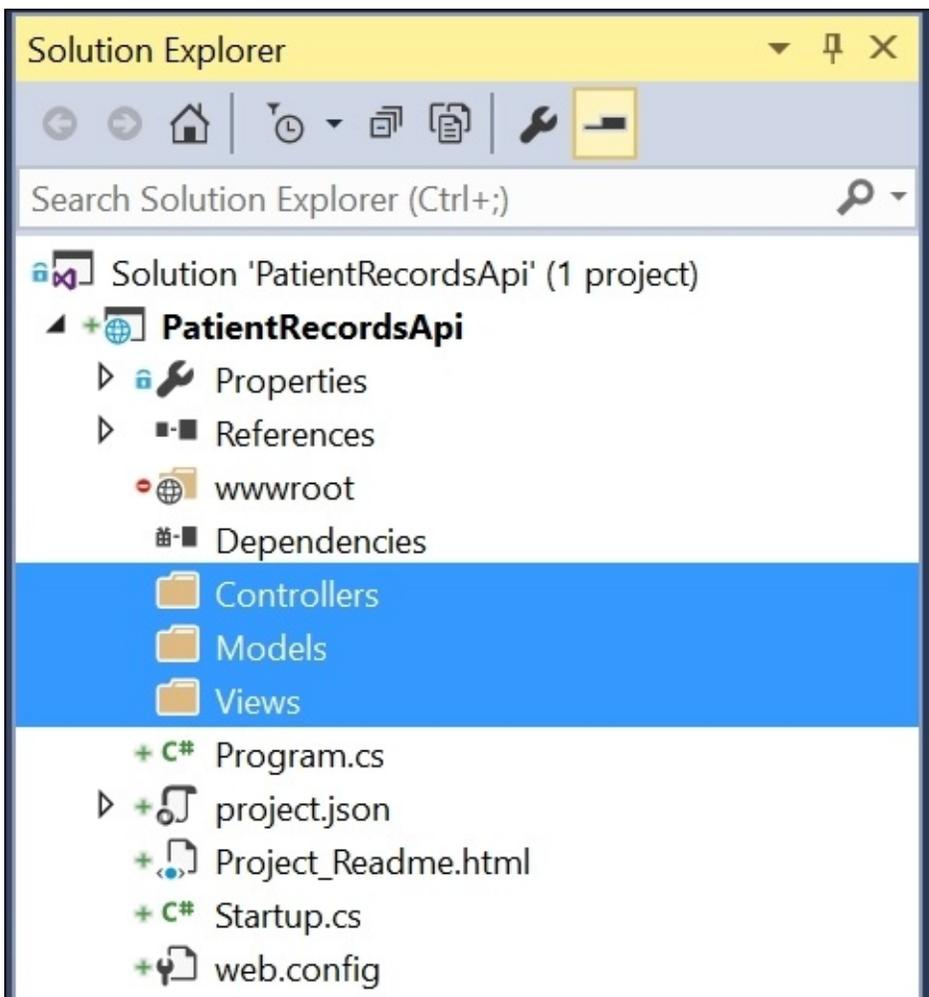
3. Right click on your `Controllers` folder, then click **Add** | **New Item** to create a new API controller for patient records on your system, as shown in the following screenshot:
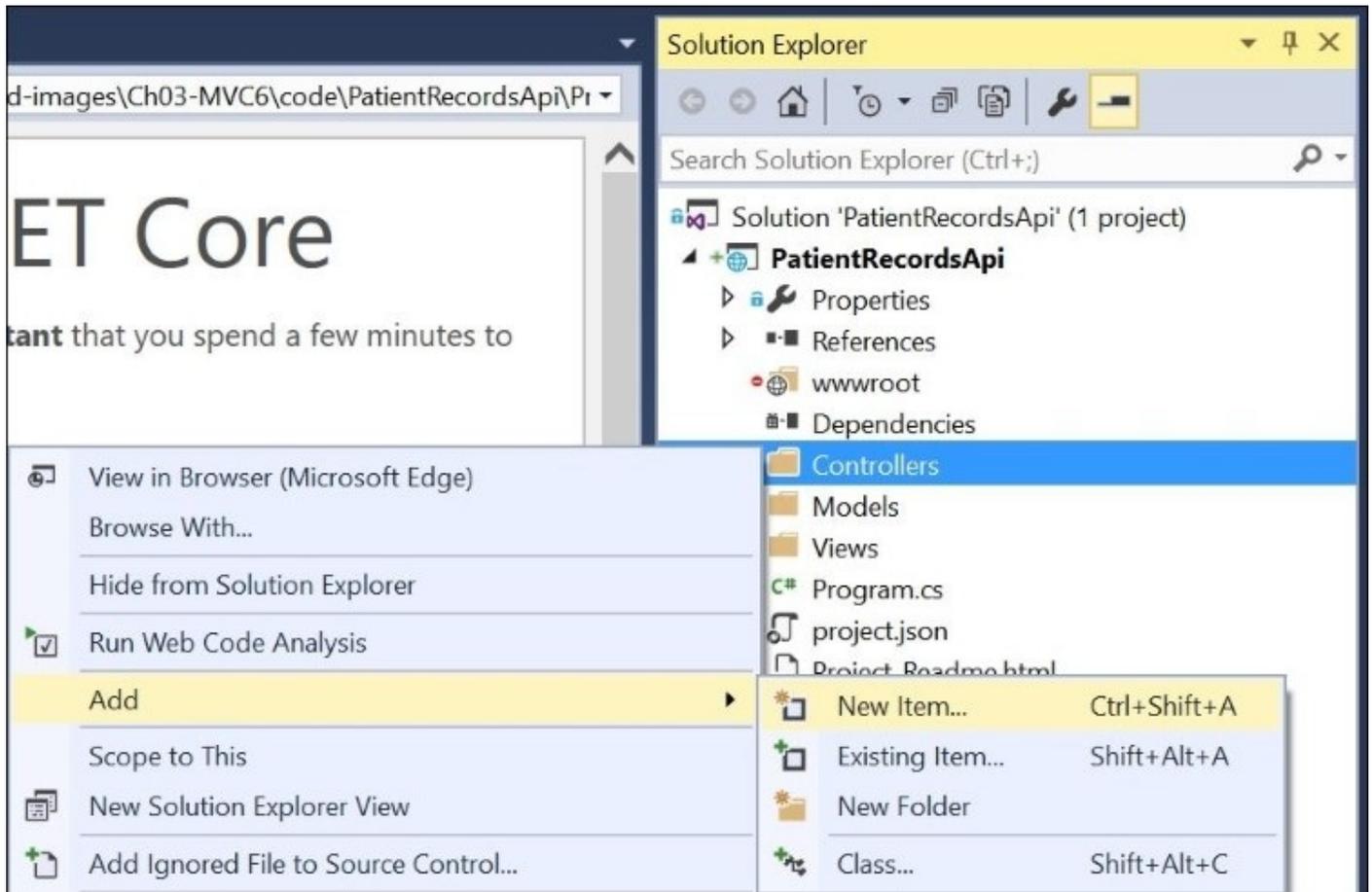
4. In the dialog box that appears, choose **Web API Controller Class** from the list of options under **.NET Core**, as shown in the following screenshot:

Add New Item - PatientRecordsApi

▲ Installed		
PowerShell	Interface	.NET Core
▲ .NET Core	Assembly Information File	.NET Core
ASP.NET	MVC Controller Class	.NET Core
Client-side	**Web API Controller Class**	.NET Core
Code	MVC View Page	.NET Core
▷ Online	MVC View Layout Page	.NET Core
	MVC View Start Page	.NET Core

Sort by: Default

Search Installed Templates (Ctrl+E)

Click here to go online and find templates.

Type: .NET Core

Web API Controller Class

Name: PatientController.cs

Add   Cancel

5. Name your new API controller, for example `PatientController.cs`, then click **Add** to proceed.

6. In your new `PatientController`, you will most likely have several areas highlighted with red squiggly lines due to a lack of necessary dependencies, as shown in the following screenshot. As a result, you won't be able to build your project/solution at this time:

```
PatientController.cs  ⌐□ ✕

C# PatientRecordsApi..NETCoreApp,Version=v1  ▼    ⚡ PatientRecordsApi.Controllers.PatientContro  ▼   ⊕  G

   1 💡   ⊟using System;
   2       using System.Collections.Generic;
   3       using System.Linq;
   4       using System.Threading.Tasks;
   5       using Microsoft.AspNetCore.Mvc;
   6
   7       // For more information on enabling Web API for
             http://go.microsoft.com/fwlink/?LinkID=397860
   8
   9     ⊟namespace PatientRecordsApi.Controllers
  10       {
  11           [Route("api/[controller]")]
  12     ⊟     public class PatientController : Controller
```

In the next section, we will learn about how to configure your Web API so that it has the
proper references and dependencies in its configuration files.

Configuring the Web API in your web application

How does the web server know what to send to the browser when a specific URL is requested? The answer lies in the configuration of your Web API project.

Setting up dependencies

In this section, we will learn how to set up your dependencies automatically using the IDE, or manually by editing your project's configuration file:

1. To pull in the necessary dependencies, you may right-click on the using statement for `Microsoft.AspNet.Mvc` and select **Quick Actions and Refactorings....** This can also be triggered by pressing *Ctrl + .* (period) on your keyboard or simply by hovering over the underlined term, as shown in the following screenshot:
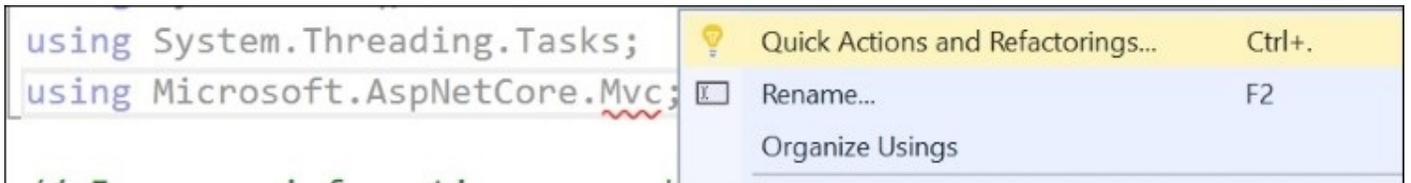
2. Visual Studio should offer you several possible options, from which you can select the one that adds the package `Microsoft.AspNetCore.Mvc.Core` for the namespace `Microsoft.AspNetCore.Mvc.` For the `Controller` class, add a reference for the `Microsoft.AspNetCore.Mvc.ViewFeatures` package, as shown in the following screenshot:

Adding the Microsoft.AspNetCore.Mvc.Core 1.0.0 package

3. If you select the latest version that's available, this should update your references and remove the red squiggly lines, as shown in the following screenshot:

Updating your references and removing the red squiggly lines

4. The preceding step should automatically update your `project.json` file with the correct dependencies for the `Microsoft.AspNetCore.Mvc.Core`, and `Microsoft.AspNetCore.Mvc.ViewFeatures`, as shown in the following screenshot:

5. The **"frameworks"** section of the `project.json` file identifies the type and version of the .NET Framework that your web app is using, for example `netcoreapp1.0` for the 1.0 version of .NET Core. You will see something similar in your project, as shown in the following screenshot:

```
project.json    X
Schema: http://json.schemastore.org/project
   19        ⊟        "frameworks": {
   20        ⊟            "netcoreapp1.0": {
   21        ⊟                "imports": [
   22                            "dotnet5.6",
   23                            "portable-net45+win8"
   24                        ]
   25                    }
   26                },
```

6. Click the **Build Solution** button from the top menu/toolbar. Depending on how you have your shortcuts set up, you may press *Ctrl + Shift + B* or press *F6* on your keyboard to build the solution. You should now be able to build your project/solution without errors, as shown in the following screenshot:
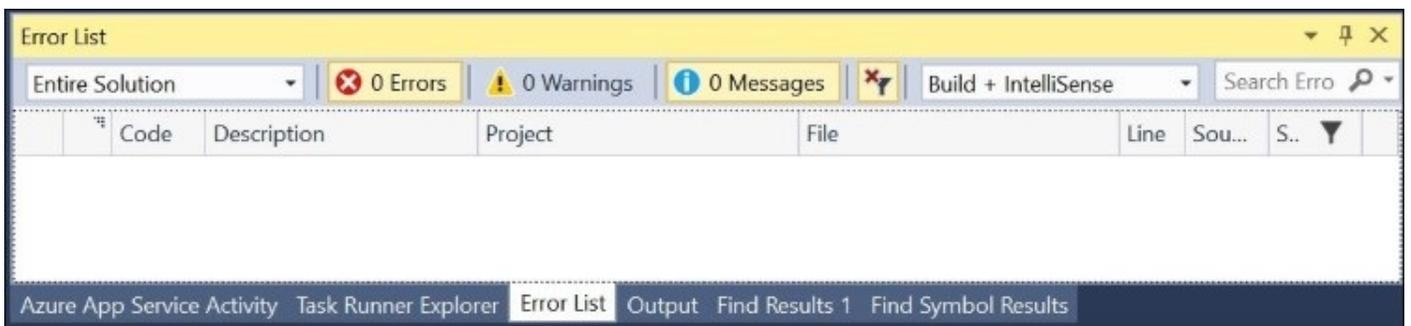
Before running the Web API project, open the `Startup.cs` class file, and replace the `app.Run()` statement/block (along with its contents) with a call to `app.UseMvc()` in the `Configure()` method. To add the `Mvc` to the project, add a call to the `services.AddMvcCore()` in the `ConfigureServices()` method. To allow this code to compile, add a reference to `Microsoft.AspNetCore.Mvc`.

Parts of a Web API project

Let's take a closer look at the `PatientController` class. The auto-generated class has the following methods:

- `public IEnumerable<string> Get()`
- `public string Get(int id)`
- `public void Post([FromBody]string value)`
- `public void Put(int id, [FromBody]string value)`
- `public void Delete(int id)`

The `Get()` method simply returns a `JSON` object as an enumerable string of values, while the `Get(int id)` method is an overridden variant that gets a particular value for a specified ID.

The `Post()` and `Put()` methods can be used for creating and updating entities. Note that the `Put()` method takes in an ID value as the first parameter so that it knows which entity to update.

Finally, we have the `Delete()` method, which can be used to delete an entity using the specified ID.

These operations will work best with an **Object-Relational Mapping** (**ORM**) framework such as an **Entity Framework**, which we will cover in Chapter 6, *Using Entity Framework to Interact with Your Database in Code* . In the meantime, we will create placeholder objects and values in the API controller code.

Running the Web API project

You may run the Web API project in a web browser that can display JSON data.

If you use Google Chrome, I would suggest using the **JSONView Extension** (or other similar extension) to properly display JSON data.

The aforementioned extension is also available on GitHub at the following URL:

https://github.com/gildas-lormeau/JSONView-for-Chrome

If you use **Microsoft Edge**, you can view the raw JSON data directly in the browser. Once your browser is ready, you can select your browser of choice from the top toolbar of Visual Studio. Click on the tiny triangle icon next to the **Debug** button, then select a browser, as shown in the following screenshot:
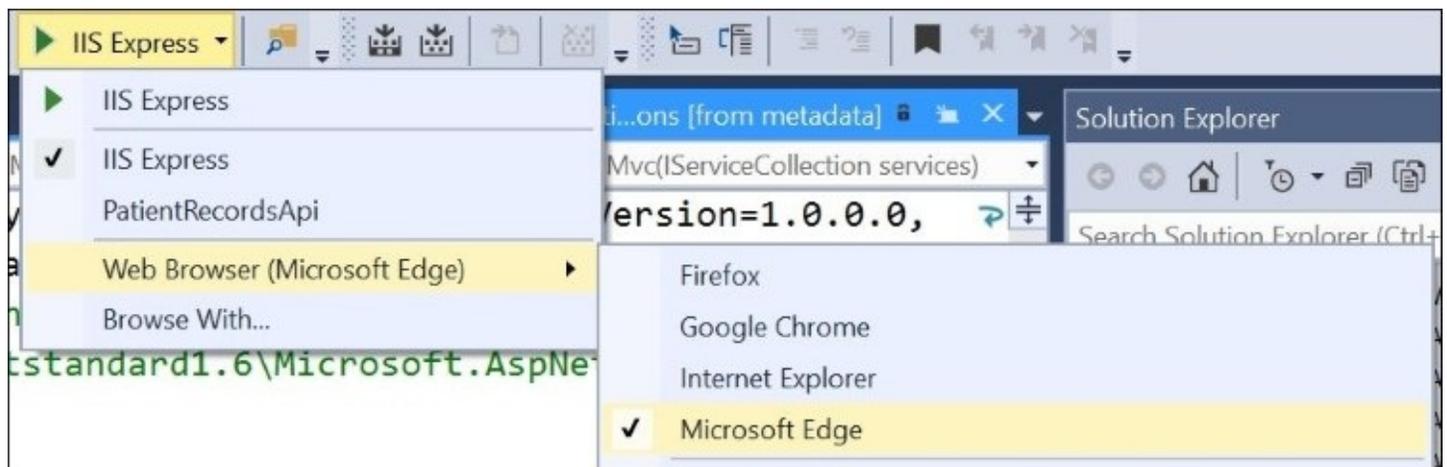
In the preceding screenshot, you can see that multiple installed browsers are available, including Firefox, Google Chrome, Internet Explorer, and Edge. To choose a different browser, simply click on **Browse With...**, in the menu to select a different one.

Now, click the **Debug** button (that is the green *play* button) to see the Web API project in action in your web browser, as shown in the following screenshot. If you don't have a web application set up, you won't be able to browse the site from the root URL:
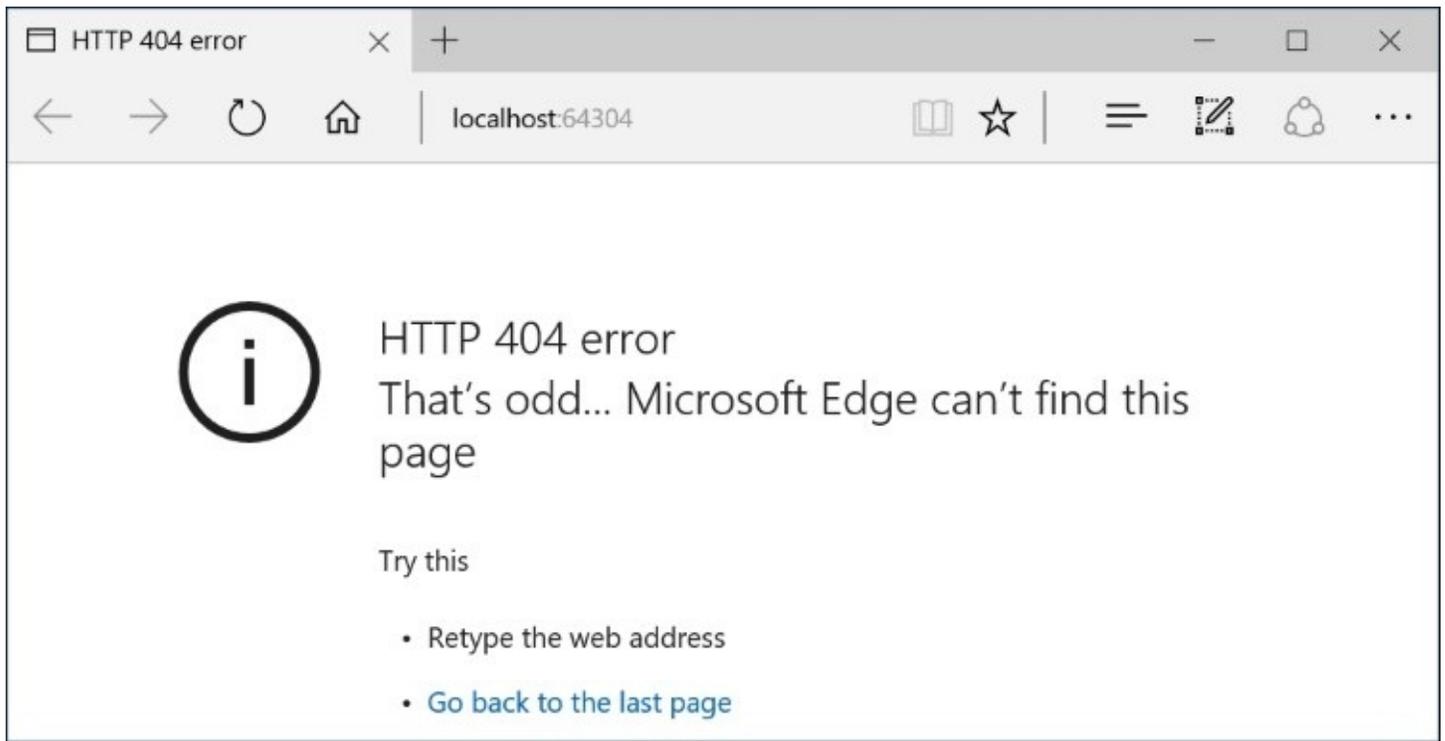
Don't worry if you see this error, you can update the URL to include a path to your API controller; for an example, see `http://localhost:12345/api/Patient`.

Note that your port number may vary. Now, you should be able to see a list of views that are being spat out by your API controller, as shown in the following screenshot:

Adding routes to handle anticipated URL paths

Back in the days of classic ASP, application URL paths typically reflected physical file paths. This continued with ASP.NET web forms, even though the concept of custom URL routing was introduced. With ASP.NET MVC, routes were designed to cater to functionality rather than physical paths.

ASP.NET Web API continues this newer tradition, with the ability to set up custom routes from within your code. You can create routes for your application using fluent configuration in your startup code or with declarative attributes surrounded by square brackets.

Understanding routes

To understand the purpose of having routes, let's focus on the features and benefits of routes in your application. <mark>This applies to both the ASP.NET MVC and ASP.NET Web API:</mark>

- By defining routes, you can introduce <u>predictable</u> patterns for URL access
- This gives you more control over how URLs are mapped to your controllers
- Human-readable route paths are also SEO-friendly, which is great for **Search Engine Optimization**
- It provides some level of obscurity when it comes to revealing the underlying web technology and physical file names in your system

Setting up routes

Let's start with this simple class-level attribute that specifies a route for your API controller, as follows:

```
[Route("api/[controller]")]
public class PatientController : Controller
{
    // ...
}
```

Here, we can dissect the attribute (seen in square brackets, used to affect the class below it) and its parameter to understand what's going on:

- The `Route` attribute indicates that we are going to define a route for this controller.
- Within the parentheses that follow, the route path is defined in double quotes.
- The first part of this path is the string literal `api/`, which declares that the path to an API method call will begin with the term `api` followed by a forward slash.
- The rest of the path is the word `controller` in square brackets, which refers to the controller name. By convention, the controller's name is part of the controller's class name that precedes the term `Controller`. For a class `PatientController`, the controller name is just the word `Patient`.
- This means that all API methods for this controller can be accessed using the following syntax, where `MyApplicationServer` should be replaced with your own server or domain name: `http://MyApplicationServer/api/Patient`

For method calls, you can define a route with or without parameters. The following example illustrates both types of route definitions:

```
[HttpGet]
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

In this example, the `Get()` method performs an action related to the HTTP verb `HttpGet`, which is declared in the attribute directly above the method. This identifies the default method for accessing the controller through a browser without any parameters, which means that this API method can be accessed using the following syntax:

```
http://MyApplicationServer/api/Patient
```

To include parameters, we can use the following syntax:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return "value";
}
```

Here, the `HttpGet` attribute is coupled with an `"{id}"` parameter, enclosed in curly braces within double quotes. The overridden version of the `Get()` method also includes an integer value named `id` to correspond with the expected parameter.

If no parameter is specified, the value of `id` is equal to `default(int)` which is zero. This can be called without any parameters with the following syntax:

```
http://MyApplicationServer/api/Patient/Get
```

In order to pass parameters, you can add any integer value right after the controller name, with the following syntax:

```
http://MyApplicationServer/api/Patient/1
```

This will assign the number `1` to the integer variable `id`.

Testing routes

To test the aforementioned routes, simply run the application from Visual Studio and access the specified URLs without parameters.

The preceding screenshot show the results of accessing the following path:

```
http://MyApplicationServer/api/Patient/1
```

Consuming a Web API from a client application

If a Web API exposes public endpoints, but there is no client application there to consume it, does it really exist? Without getting too philosophical, let's go over the possible ways you can consume a client application.

You can do any of the following:

- Consume the Web API using external tools
- Consume the Web API with a mobile app
- Consume the Web API with a web client

Testing with external tools

If you don't have a client application set up, you can use an external tool such as **Fiddler**. Fiddler is a free tool that is now available from Telerik, at http://www.telerik.com/download/fiddler , as shown in the following screenshot:
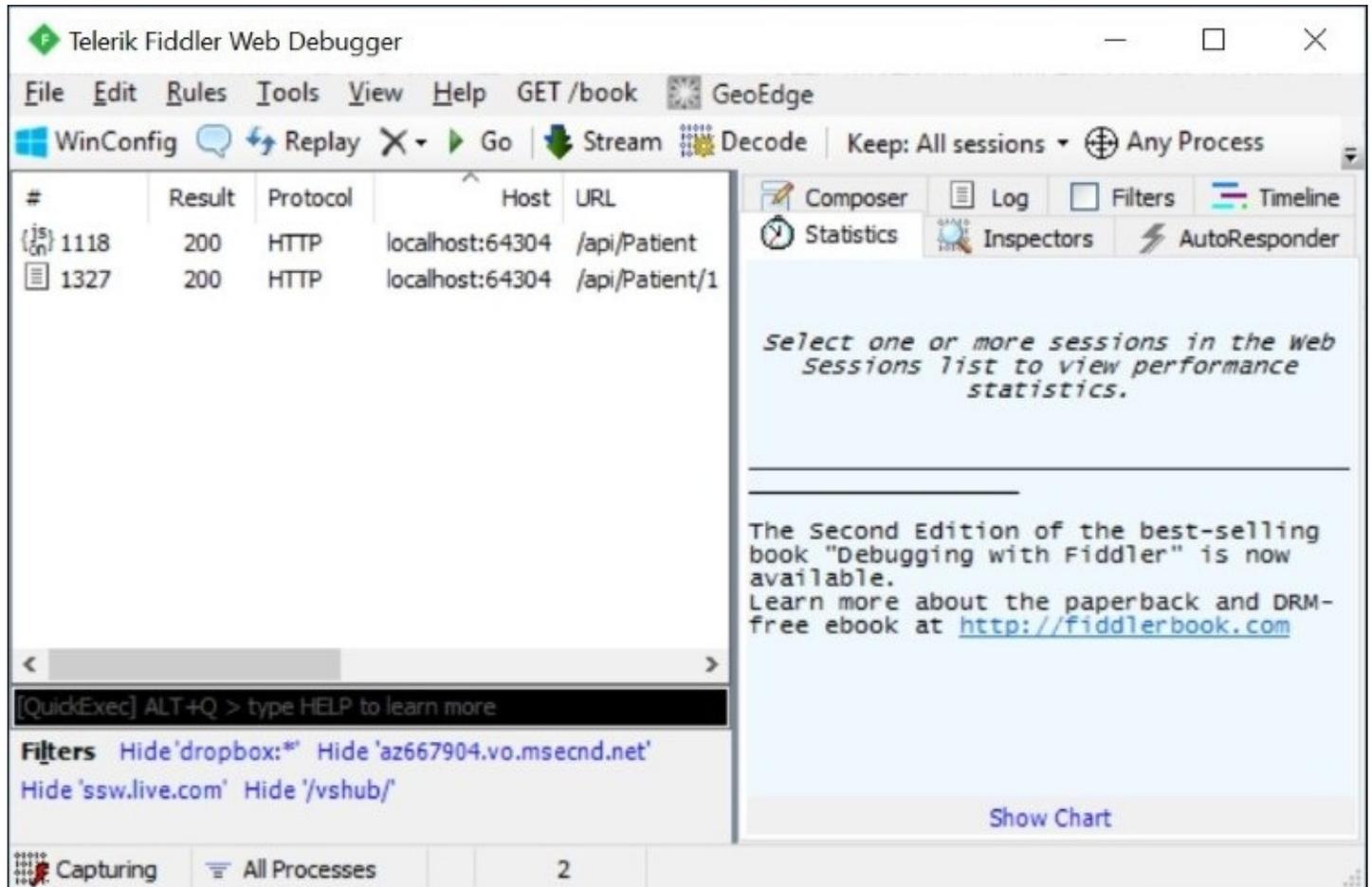
You can use Fiddler to inspect URLs that are being retrieved and submitted on your machine. You can also use it to trigger any URL, and change the request type (Get, Post, and others).

Consuming a Web API from a mobile app

Since this chapter is primarily about the ASP.NET core Web API, we won't go into detail about mobile application development. However, it's important to note that a Web API can provide a backend for your mobile app projects.

Mobile apps may include Windows Mobile apps, iOS apps, Android apps, and any modern app that you can build for today's smartphones and tablets. You may consult the documentation for your particular platform of choice, to determine what is needed to call a RESTful API.

Consuming a Web API from a web client

A web client, in this case, refers to any HTML/JavaScript application that has the ability to call a RESTful API. At the very least, you can build a complete client-side solution with straight JavaScript to perform the necessary actions. For a better experience, you may use jQuery and also one of many popular JavaScript frameworks.

A web client can also be a part of a larger ASP.NET MVC application or a **Single-Page Application** (**SPA**). As long as your application is spitting out JavaScript that is contained in HTML pages, you can build a frontend that works with your backend Web API.

Summary

In this chapter, we've taken a look at the basic structure of an ASP.NET Web API project, and observed the unification of a Web API with MVC in an ASP.NET core. We also learned how to use a Web API as our backend to provide support for various frontend applications.

In the next chapter, we will learn more about developing an interactive frontend UI with HTML5, JavaScript, and CSS3. We will focus on using JavaScript to consume a Web API, while getting an introduction to client-side tools, such as Bower, Grunt, and Gulp.

Chapter 5. Interacting with Web API using JavaScript

A browser-based web application wouldn't be complete without client-side code to complement the server-side code. Regardless of the technology and language used on the web server, all web application developers should learn how to use JavaScript with their HTML on the client.

In this chapter, we will start off by building an HTML page filled with JavaScript code to develop an interactive user interface. We will cover the following topics:

- Working with ASP.NET Web API using JavaScript
- JavaScript frameworks
- Client-side tools such as **Bower**, **Grunt**, and **Gulp**

Using JavaScript to interact with Web API

There have been a lot of JavaScript frameworks fighting for the affections of web developers worldwide, and a few of them have caught on. In the next few sections, we will learn about **AngularJS** and **KnockoutJS**. But first, let's focus on using some basic JavaScript to read and write data to/from your Web API.

The example for this section will use the following files:

- HTML file with client-side JavaScript to use the Web API
- Server-side controller class
- Server-side model class

The following screenshot shows the preceding three files that we need:
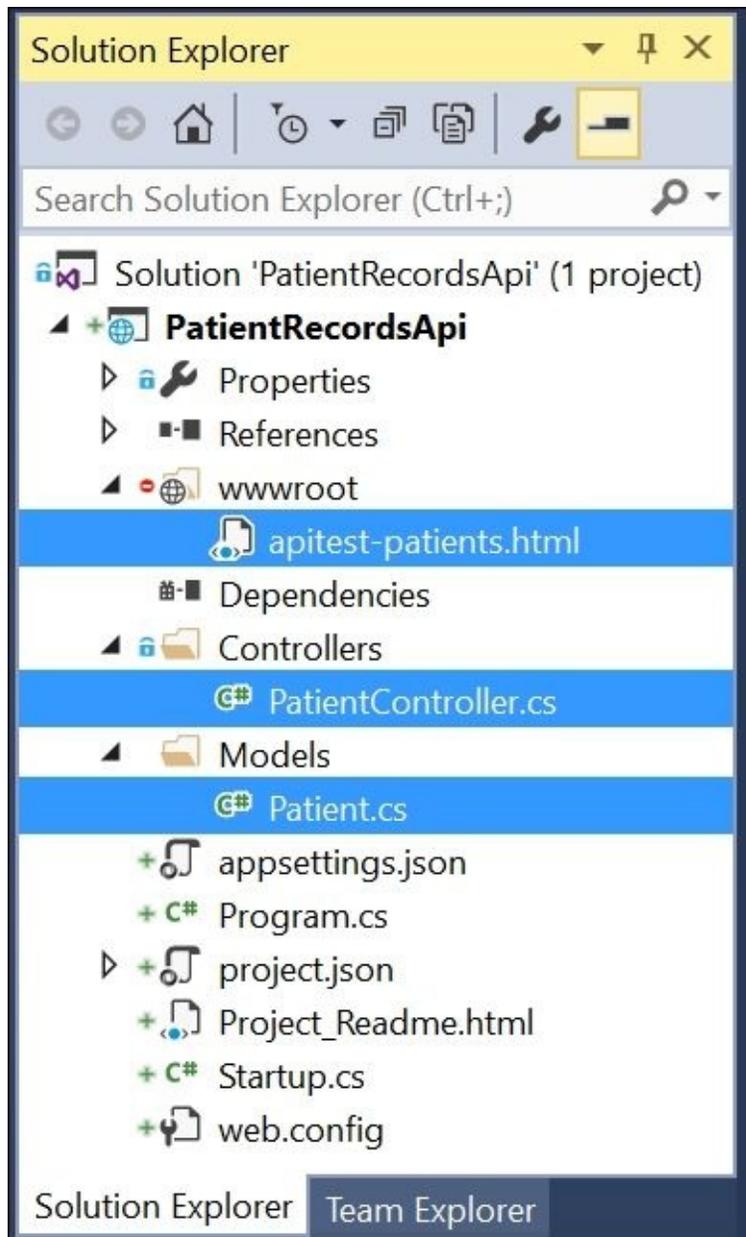
We will be dealing with these files in the example in this chapter:

- The static HTML file is in the `wwwroot` location of the web application
- The controller is in the `Controllers` folder (by convention)
- The patient model is in the `Models` folder (by convention)

To visualize what the application looks like, see the following screenshot of the **Patient Records App**:

Preparing the server-side code

Create a new class file in the `Models` folder and call it `Patient.cs` with the following code:

```
public class Patient
{
    public int Id { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string SocialSecurityNumber { get; set; }
}
```

The preceding code gives us just the right amount of controller and model code to get started on the client-side JavaScript.

The following API controller has at least two controller methods to return a list of patients or just one patient when given a patient ID. This controller should have a using statement to reference the models namespace.

It begins with a class-level `[Route]` attribute to indicate how the API controller will be accessed from a web browser's GET request or through JavaScript code in the browser, as shown in the following code:

```
[Route("api/[controller]")]
public class PatientController : Controller
```

Inside the class itself, there is a class-level instance variable that holds a data structure to define an array of patients with a few sample values, as shown in the following code. In a real-world application, this data would typically come from a database:

```
    Patient[] patients = new Patient[]
    {
        new Patient
        {
            Id = 1,
            FirstName = "John",
            LastName = "Smith",
            SocialSecurityNumber = "123550001"
        },
        new Patient
        {
            Id = 2,
            FirstName = "Jane",
            LastName = "Doe",
            SocialSecurityNumber = "123550002"
        },
        new Patient
        {
            Id = 3,
            FirstName = "Lisa",
            LastName = "Smith",
            SocialSecurityNumber = "123550003"
```

```
        }
    };
```

Next, we have a simple `Get()` method to return an enumerable list of patient objects:

```
    // GET: api/patient
    [HttpGet]
    public IEnumerable<Patient> Get()
    {
        return patients;
    }
```

Finally, we have a simple `Get(id)` method to return a specific patient when a numeric ID value is provided to perform a lookup:

```
    // GET api/patient/5
    [HttpGet("{id}")]
    public Patient Get(int id)
    {
        var patient = patients.FirstOrDefault((p) => p.Id == id);
        if (patient == null)
        {
            return null;
        }
        return patient;
    }
}
```

Client-side JavaScript

The following HTML/JavaScript code can be included <mark>in a sample HTML file to obtain and display patient records.</mark> It starts off with a typical HTML opening:

```
<!DOCTYPE html>
<html>
<head>
    <title>Patient Records App</title>
</head>
```

The body contains a header and a `<div>` to display the data:

```
<body>

<div>
    <h2>Patient Records</h2>
    <ul id="patients"></ul>
</div>
<div>
    <h2>Find Patient By ID</h2>
    <input type="text" id="patientId" size="5" />
    <input type="button" value="Search" onclick="find();" />
    <p id="patient" />
</div>
```

<mark>Note the following:</mark>

- <mark>An empty list is defined with an ID of `patients` to display a list of patient records when the page first loads</mark>
- <mark>A search textbox is defined with an ID of `patientId` to allow the user to search by ID</mark>
- <mark>The **Search** button has an `onclick` event, which calls a `find()` method</mark>
- <mark>An empty paragraph is defined with an ID of `patient` to display the results of a call to the `find()` method</mark>

Next is a reference to jQuery, which can be a specific (or the latest) version, as needed:

```
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.min.js"></script>
```

Then, we begin the main `<script>` to define functions to handle API calls:

```
<script>
```

At the beginning of the script, let's define a variable to hold the URI path for the API call:

```
var uri = 'api/Patient';
```

Next, let's ensure that the patient list is populated immediately after the page loads:

```
$(document).ready(function () {
    $.getJSON(uri)
        .done(function (data) {
```

```
            $.each(data, function (key, patient) {
                $('<li>', {
                    text: formatPatientInfo(patient)
                }).appendTo($('#patients'));
            });
        });
    });
```

The preceding code has a reference to a function named `formatPatientInfo()` to format the patient information returned by the API call. This method is defined as follows:

```
function formatPatientInfo(patient) {
    if (patient && patient.lastName)
        return patient.lastName +
            ', ' + patient.firstName +
            ': ' + patient.socialSecurityNumber;
    else
        return 'No patient information found.';
}
```

To enable search functionality, a `find()` method is defined as follows:

```
function find() {
    var id = $('#patientId').val();
    $.getJSON(uri + '/' + id)
        .done(function (data) {
            $('#patient').text(formatPatientInfo(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#patient').text('Error: ' + err);
        });
}
```

Here, an `id` variable is defined to obtain the numeric ID value from the search box provided to the end user. This ID is then appended to the end of the previously defined URI to form a URL used to call `getJSON().`

Once the API call has been triggered, JavaScript handles successful and failed calls with `.done` and `.fail` blocks, respectively. The `.done` block updates.

At this point, we can close out the `<script>`, `<body>`, and `<html>` tags:

```
</script>
</body>
</html>
```

To verify that you can display static files in your web app, perform the following steps:

1. Verify that `Main()` includes `.UseContentRoot(Directory.GetCurrentDirectory())` in `Program.cs`.
2. Add a reference to `Microsoft.AspNetCore.StaticFiles(1.0.0)`.
3. Add `app.UseStaticFiles()` to `Startup.cs`.

The following screenshot shows the results displayed when the user searches for a patient with an ID of 1:

JavaScript frameworks

When you start adding more and more JavaScript to your code, your web application can get unwieldy and difficult to maintain. To take advantage of providing JavaScript frameworks, you may choose from many JavaScript frameworks.

These are four popular frameworks:

- **AngularJS**: https://angularjs.org
- **KnockoutJS**: http://knockoutjs.com
- **BackboneJS**: http://backbonejs.org
- **EmberJS**: http://emberjs.com

In this chapter, we will focus on AngularJS and KnockoutJS.

Single-page applications with AngularJS

The official ASP.NET documentation mentions AngularJS as an option for building a SPA-style ASP.NET application. The examples in this chapter are derived from Microsoft's documentation at: http://docs.asp.net/en/latest/client-side/angular.html .

Getting started with AngularJS

AngularJS is an open source JavaScript framework, which is officially maintained by Google. At the time of writing, the latest stable release is at version 1.5.x, and version 2.0 is going through a few release candidate (RC) stages as of July 2016.

AngularJS uses a subset of jQuery called **jqLite**, which allows it to manipulate the DOM of an HTML page across multiple browsers. If you want to use jQuery in your AngularJS application, you must ensure that jQuery is loaded before the `angular.js` file.

The easiest way to set up AngularJS to your web application in Visual Studio 2015 is to update your `bower.json` file in a new web application. Launch Visual Studio 2015, click on **File** | **New** | **Project**, and then create a new **Web Application** using the new **ASP.NET Template** (not the empty one).

This should automatically create a `bower.json` configuration file for you. You may have to click on the `Show All Files` icon in the Solution Explorer to view the `bower.json` file. Simply edit this JSON file and add two entries for `"angular"` and `"angular-route"` under `"dependencies"`. Use **IntelliSense** to help you decide which version numbers to use, as shown in the following screenshot:
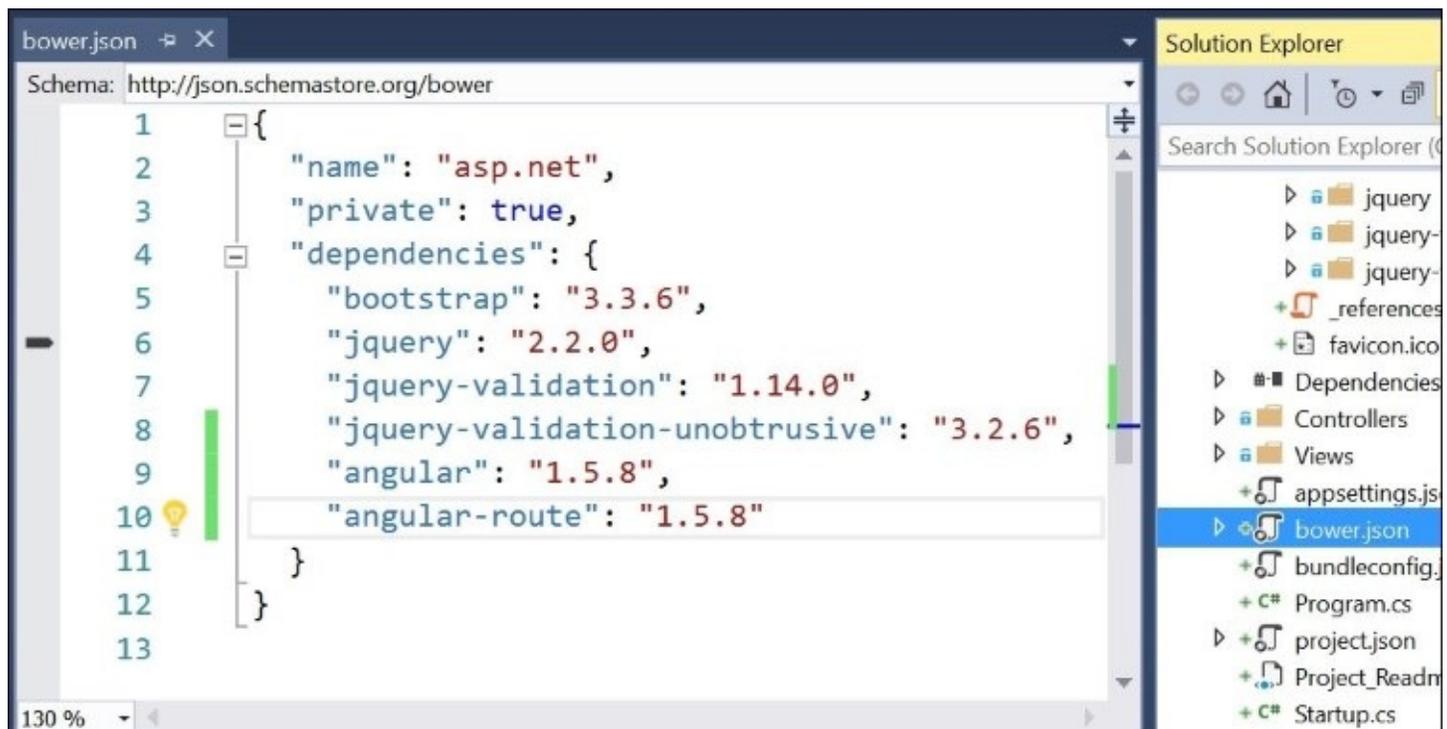
The bower.json file

The `angular-route` module enables your web app to use routing and deeplinking services, as

These terms (and more) are explained in the following section.

Almost immediately after you add the two entries, the necessary JavaScript files will be added to your `Bower` folder under `/Dependencies/Bower/`, as shown in the following screenshot. You can configure Grunt or Gulp to always copy the necessary files to the `lib` subfolder within the `wwwroot` web root location.
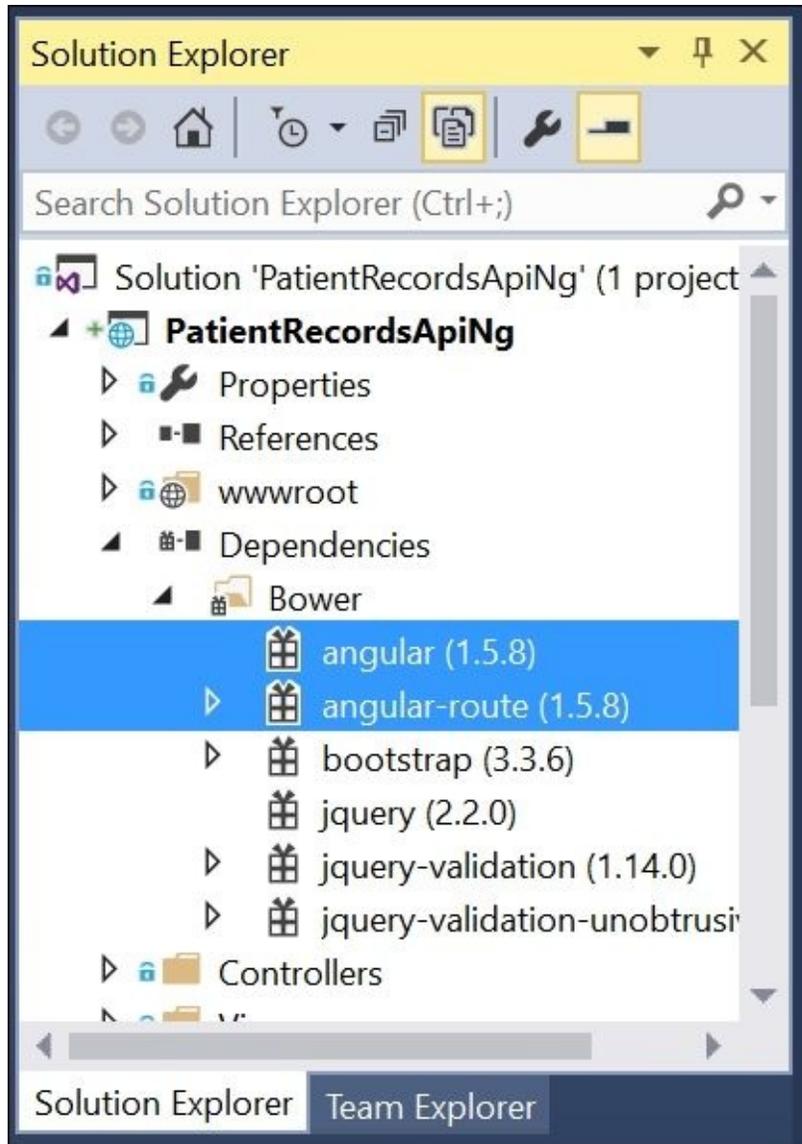
Angular (1.5.8) and Angular route (1.5.8)

The following screenshot shows how your `lib` folder should look after the necessary modules are placed into it:
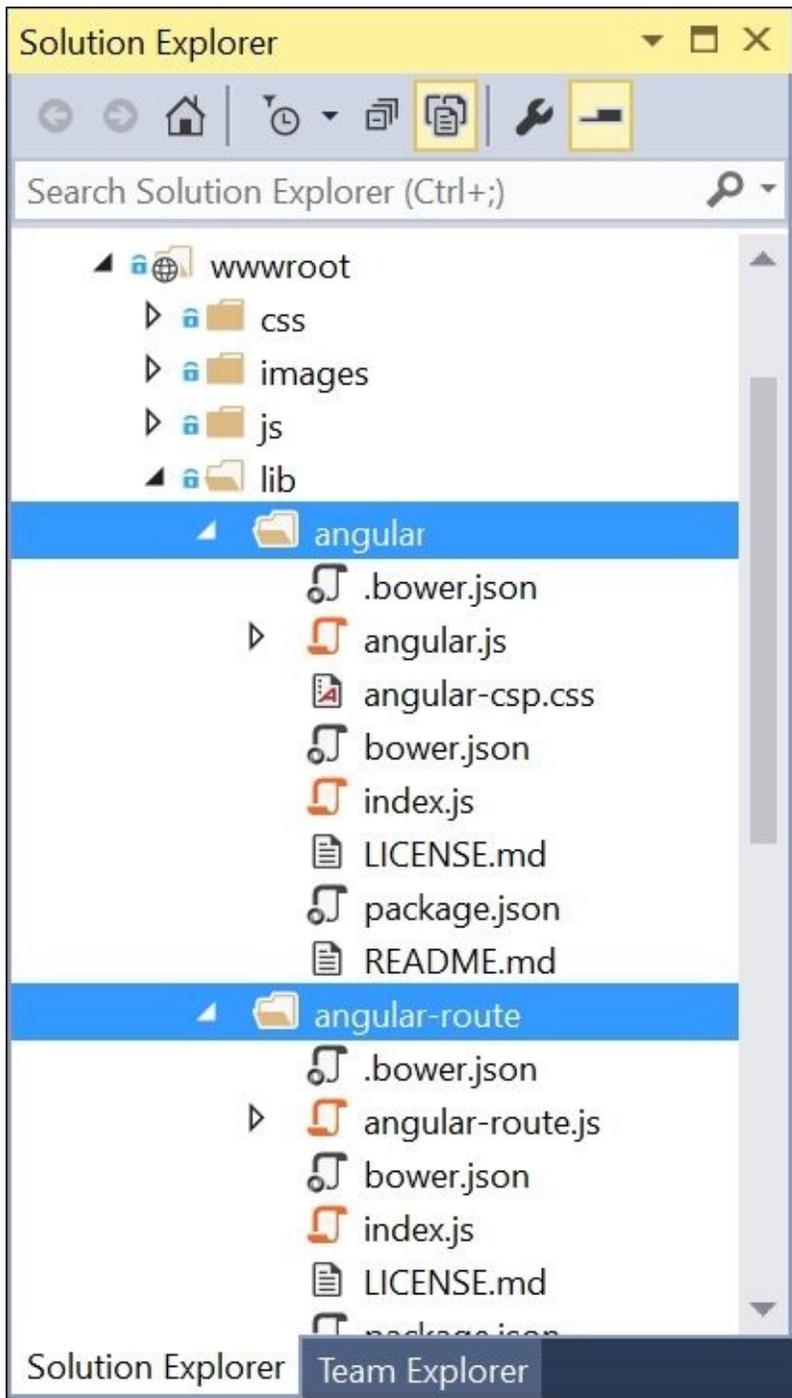
In the last section of this chapter, we will learn more about using Bower to manage your packages for your client-side code.

AngularJS syntax and features

AngularJS can be explained by illustrating its syntax and features with a few examples. AngularJS syntax in an HTML file can be easily identified by recognizing the following:

- Directives that appear within HTML tags, such as `ng-app` and `ng-init`
- Expressions that appear in double curly braces, such as `{{1+2}}`
- Data binding with `ng-bind`, such as `<span ng-bind="ssn"></span>`
- Repeaters to enumerate data, such as `<li ng-repeat="patient in patients">`
- Event handlers to handle user input, such as `<button ng-click="handleClick()">`
- All of the preceding can be contained in HTML files that are known as templates.

Within JavaScript code files, you may recognize some of the following syntax:

- App modules defined with `angular.module()`
- Model factories defined with `.factory()`
- Services defined with `.service()`
- Controllers defined with `.controller()`

This is not an exhaustive list, and this chapter is not meant to be a substitute for AngularJS documentation or tutorials. To learn more about AngularJS, read through the official documentation at https://docs.angularjs.org .

Building a web application with AngularJS

Once you have your dependencies set up in your project, <u>add a script reference to AngularJS in your HTML code</u>. This can be added in the following ways:

> - In your `_Layout.cshtml` file under `/Views/Shared`
> - In a static HTML page, just before the closing `</body>` tag

The layout file has different sections for `Development` and `Staging/Production`. While you may use a local reference for development, you should use a shared URL from a **Content Delivery Network** (**CDN**) for staging and production as follows:

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
</environment>
```

For staging and production, common CDN servers used for AngularJS include Google's CDN at ajax.googleapis.com or Microsoft's AJAX CDN at ajax.aspnetcdn.com.

A static HTML file is much simpler, as it just refers to your local file. You may also use IntelliSense to type out the path to your `angular.js` file, as Visual Studio will locate the file on your system while you type. The following code does this:

```
    <script src="../lib/angular/angular.js"></script>
</body>
</html>
```

The following static HTML files can be saved to the `wwwroot` web folder of your project, while additional JavaScript files can be saved there as well:
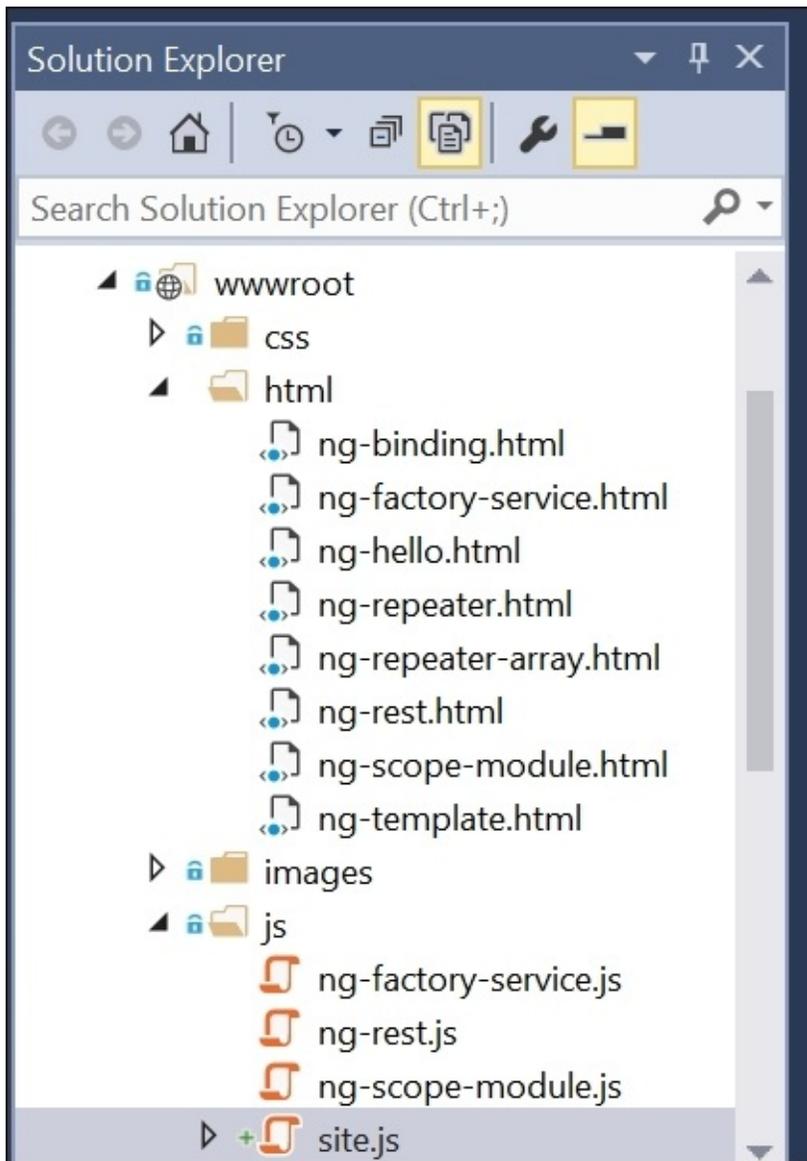
The following code shows the contents of a simple HTML file named `ng-hello.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Hello, Angular!</title>
</head>
<body ng-app>
    <h1>Hello, Angular!</h1>

    Calculate (2016 - 1990) = {{2016-1990}}
    <script src="../lib/angular/angular.js"></script>
</body>
</html>
```

The following screenshot is the output of `ng-hello.html`:
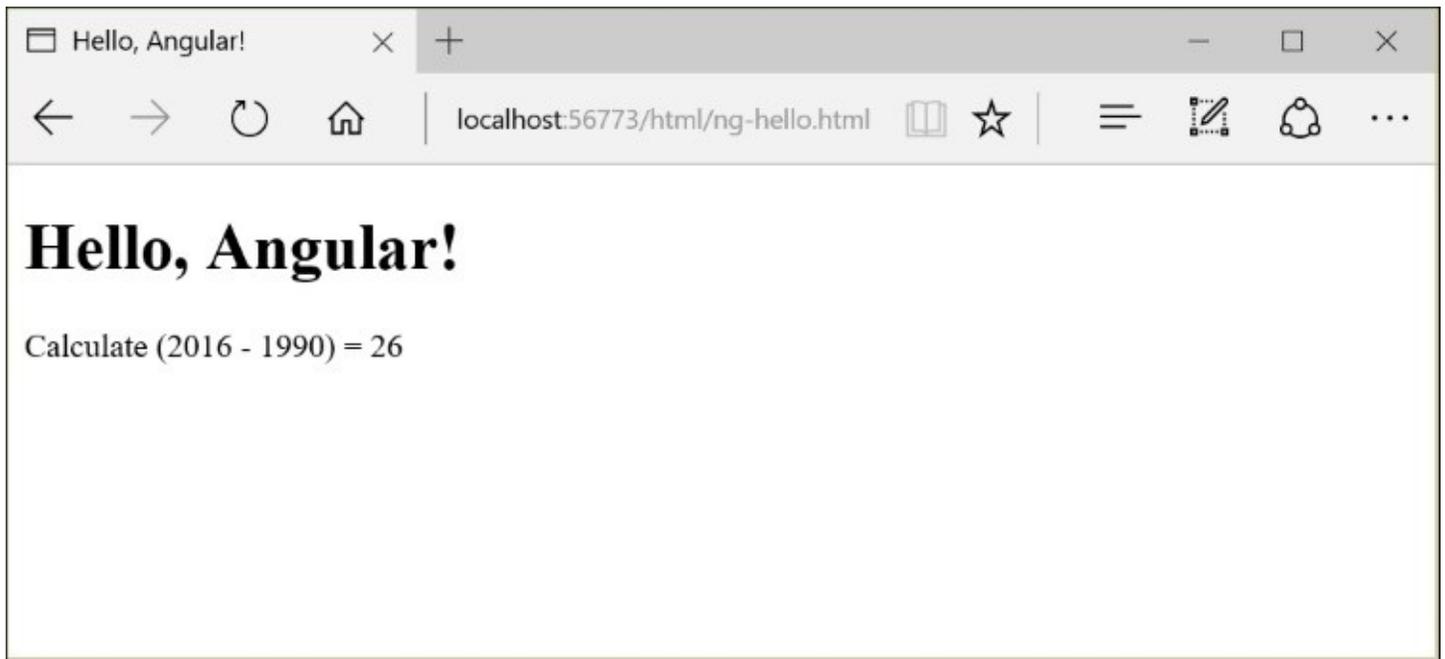
Note that the expression {{2016-1990}} is actually calculated as an arithmetic operation, and the result (26) is displayed in the browser.

The following code shows data binding in action in the sample file ng-binding.html:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Data Binding in Angular</title>
</head>
<body ng-app>
    <h1>Data Binding in Angular</h1>
    <h2>Patient Details</h2>

    <ul ng-init="lastName='Smith';
firstName='John';socialSecurityNumber='123550001'">
        <li>Name: {{lastName}}, {{firstName}}</li>
        <li>SSN: <span ng-bind="socialSecurityNumber"></span></li>
    </ul>

    <script src="../lib/angular/angular.js"></script>
</body>
</html>
```

The following screenshot is the output of ng-binding.html:

Note the following:

- The `ng-app` directive within the `<body>` tag indicates the root element of the Angular application
- The `ng-init` directive within the `<ul>` tag initializes the expression with the scope of the `<ul>` block
- The double curly braces within the first `<li>` element evaluate variables initialized within the `<ul>`
- The `ng-bind` directive in the second `<li>` element is another way to bind to a variable
- Both ways of data binding are valid

The following code shows a repeater in action in the sample file `ng-repeater.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Angular Repeaters</title>
</head>
<body ng-app>
    <h1>Angular Repeaters</h1>

    <h2>Patient Records</h2>
    <div ng-init="patients=['Smith, John','Doe, Jane', 'Smith, Lisa']">
        <ul>
            <li ng-repeat="patient in patients">
                {{patient}}
            </li>
        </ul>
    </div>
```

```
        <script src="../lib/angular/angular.js"></script>
</body>
</html>
```

The following screenshot is the output of `ng-repeater.html`:

Here, note the following:

- The `ng-init` directive initializes a simple array of patient names, each stored in a string
- The `ng-repeat` directive enumerates through each patient in the array
- The double curly braces appear inside an `<li>` element, which is repeated in the eventual HTML that is generated in the DOM

The following code shows a basic example of using the Web API controller that we created in the previous chapter, in the sample file `ng-rest.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>RESTful Endpoints from Angular</title>
</head>
<body ng-app="patientRecordsApp">
    <h1>RESTful Endpoints from Angular</h1>

    <div ng-controller="patientController">
        <ul>
            <li ng-repeat="patient in patients">
                {{patient.lastName}}, {{patient.firstName}}
            </li>
        </ul>
```

```
        </div>

        <script src="../lib/angular/angular.js"></script>
        <script src="../js/ng-rest.js"></script>
</body>
</html>
```

It uses the **Model View Controller** pattern on the client side by letting a controller update the model data, which is then displayed in the HTML. In the following code, you will notice that there is a reference to a JavaScript file named ng-rest.js:

```
(function () {
    'use strict';
    var prApp = angular.module('patientRecordsApp', []);
})();

(function () {
    'use strict';

    var pService = 'patientFactory';

    angular.module('patientRecordsApp').factory(pService,
        ['$http', patientFactory]);

    function patientFactory($http) {

        function getPatientsFromApi()
        {
            // NOTE: the port number should be changed as necessary
            return $http.get('http://localhost:50915/api/Patient');
        }

        var patientService = {
            getPatientsFromApi: getPatientsFromApi
        };
        return patientService;
    }
})();

(function () {
    'use strict';

    var pController = 'patientController';

    angular.module('patientRecordsApp').controller(pController,
        ['$scope', 'patientFactory', patientController]);

    function patientController($scope, patientFactory) {
        $scope.patients = [];

        patientFactory.getPatientsFromApi().success(function (patientData) {
            $scope.patients = patientData;
            console.log("Data obtained successfully.");
        }).error(function (error) {
            console.log("An error has occured.");
        });
```

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

In the preceding code, the `getPatientsFromApi()`method is responsible for calling the API method in the server-side code to display the following results in the web browser:

To get the preceding Angular sample to work, run the previous Web API project first and refer to the API URL at its specific port number before performing an API call. You could also add a new server-side API controller in your Angular project if you prefer.

Model-View-ViewModel (MVVM) with KnockoutJS

The official ASP.NET documentation mentions KnockoutJS as an option for building a Model-View-ViewModel (**MVVM**) web application; refer to http://docs.asp.net/en/latest/client-side/knockout.html .

Getting started with KnockoutJS

KnockoutJS is a popular JavaScript framework that you can use by itself, or with jQuery, and other JavaScript libraries. It uses the MVVM pattern on the client side and facilitates data binding between HTML elements and JavaScript variables.

Similar to AngularJS, the easiest way to set up KnockoutJS to your web application in Visual Studio 2015 is to update your `bower.json` file in a new web application. Once again, launch Visual Studio 2015, click on **File** | **New** | **Project**, and then create a new **Web Application** using the new **ASP.NET Template** (not the empty one).

Just like previously, this should automatically create a `bower.json` configuration file that you can edit. Add an entry for `"knockout"` under `"dependencies"`. You may use IntelliSense to help you decide which version numbers to use. Once again, you may have to click on **Show All Files** in the Solution Explorer to see the `bower` file. The following screenshot shows the `bower` file:
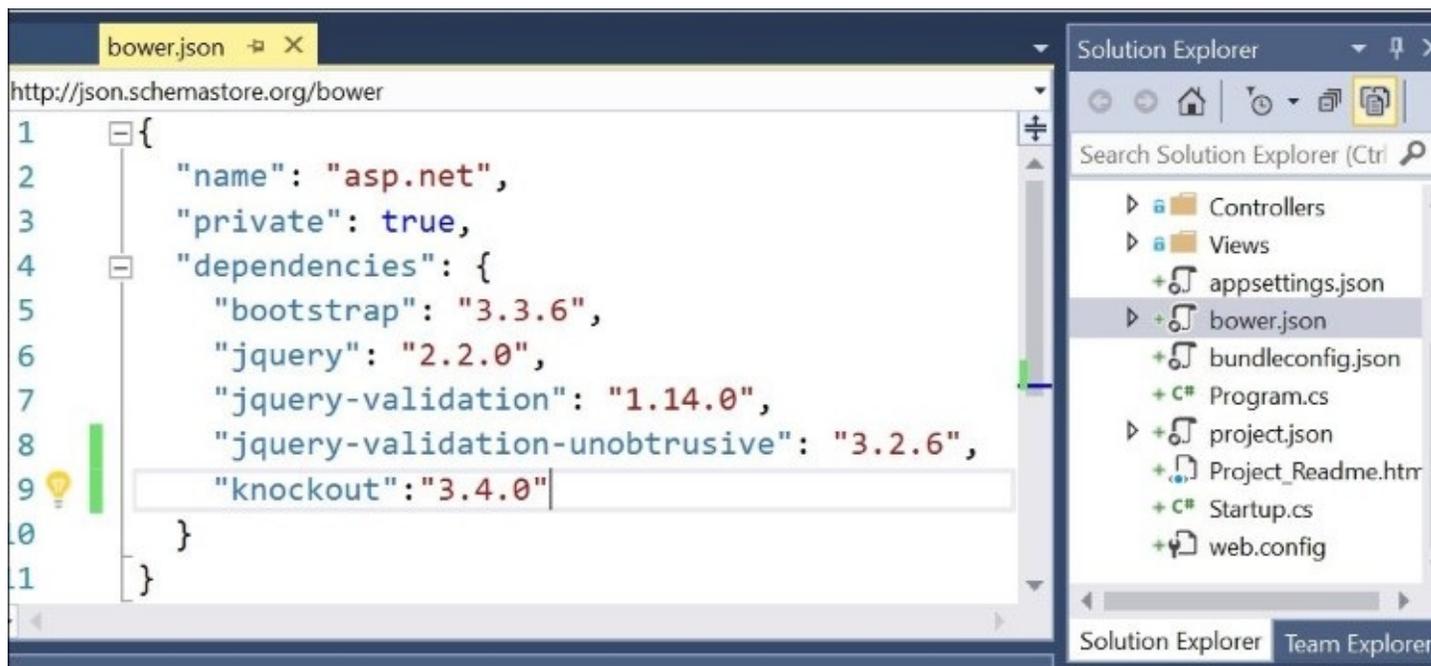
The bower.json file "Knockout" : "3.4.0"

Almost immediately after you add the entry, the necessary JavaScript dependency will be added to your `Bower` folder under `/Dependencies/Bower/`, as shown in the following screenshot:

Knockout (3.4.0)

The following screenshot shows how your `lib` folder should look after the Knockout module is placed into it:

Solution Explorer   ▼ □ ✕

[◀ ▶ ⌂ | 🕐 ▾ 🗗 🗐 | 🔧 ▬]

Search Solution Explorer (Ctrl+;)   🔍 ▾

    ▷ ■·■ References   ▲
    ▲ 🔒🌐 wwwroot
        ▷ 🔒📁 css
        ▷ 🔒📁 images
        ▷ 🔒📁 js
        ▲ 🔒📁 lib
            ▷ 🔒📁 bootstrap
            ▷ 🔒📁 jquery
            ▷ 🔒📁 jquery-validation
            ▷ 🔒📁 jquery-validation-unobtrusive
            ▲   📁 knockout
                ▷   📁 build
                ▷   📁 dist
                ▷   📁 src
                      🗍 .bower.json
                      🗍 bower.json
                      🗍 Gruntfile.js
                      🗋 LICENSE
                      🗍 package.json
                      🗎 README.md   ▼

Solution Explorer   Team Explorer
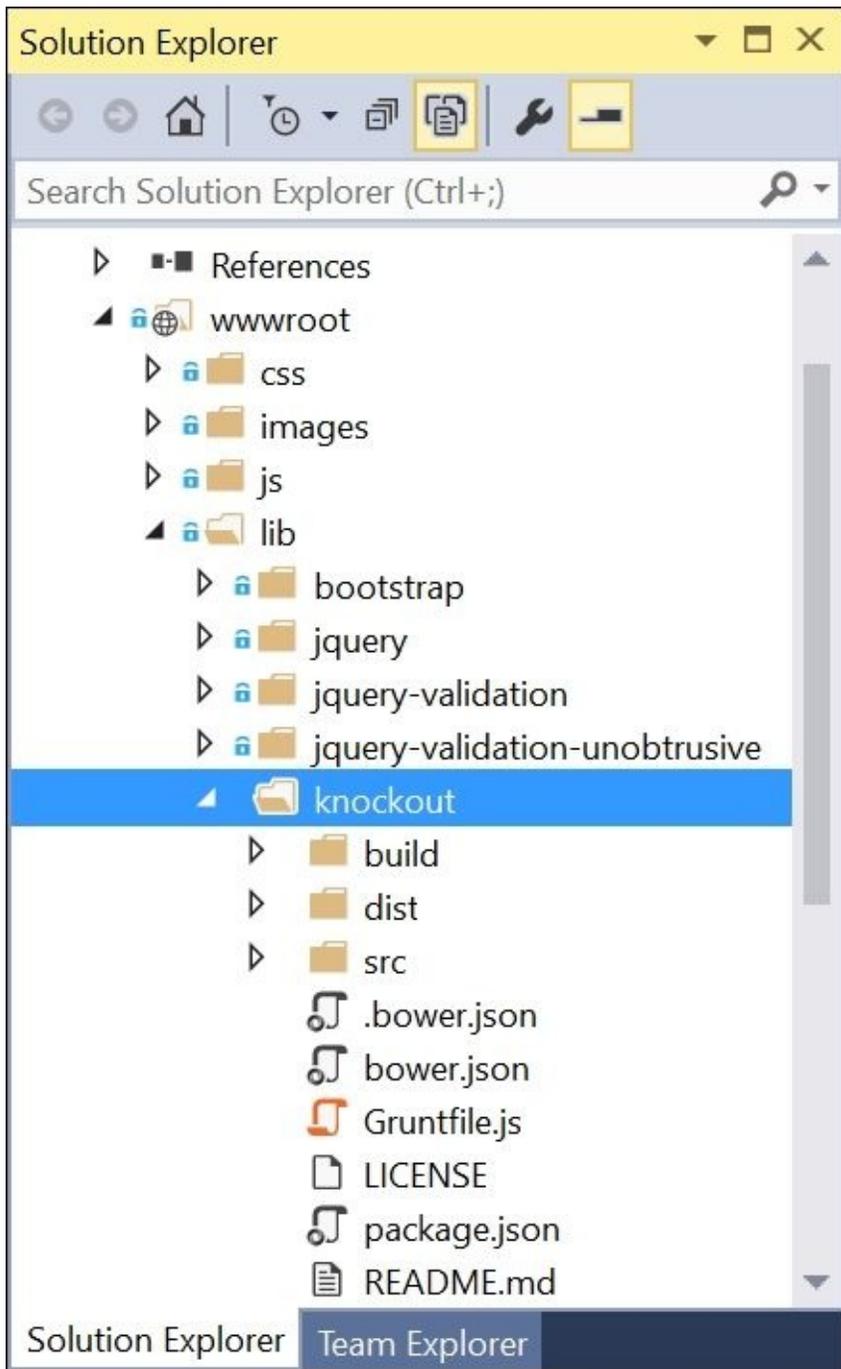
KnockoutJS syntax and features

KnockoutJS syntax can be understood by recognizing the following:

- Data-binding setup using `data-bind` attribute within HTML tags
- Associating bindings with a data model using `.applyBindings()`
- Auto-updates using `observable()` for objects
- Auto-updates using `observableArray()` for collections
- Custom bindings using `computed()` for one or more observables

To learn more about KnockoutJS, read through the official documentation at http://knockoutjs.com/documentation/introduction.html .

Building a web application with KnockoutJS

Once you have your dependency set up in your project, you may add a script reference to KnockoutJS in your HTML code.

The following code shows a basic example of binding a `<span>` element to a simple data value in the sample file `ko-hello.html`:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Hello, Knockout!</title>
    <script type="text/javascript" src="../lib/knockout/dist/knockout.js">
</script>
</head>
<body>
    <h1>Hello, Knockout!</h1>

    <h2>Patient Details</h2>

    Name: <span data-bind="text: patientName"></span>

    <script type="text/javascript">
        var patientViewModel = {
            patientName: 'John Smith'
        };
        ko.applyBindings(patientViewModel);
    </script>

</body>
</html>
```

The following screenshot shows the output of `ko-hello.html`:
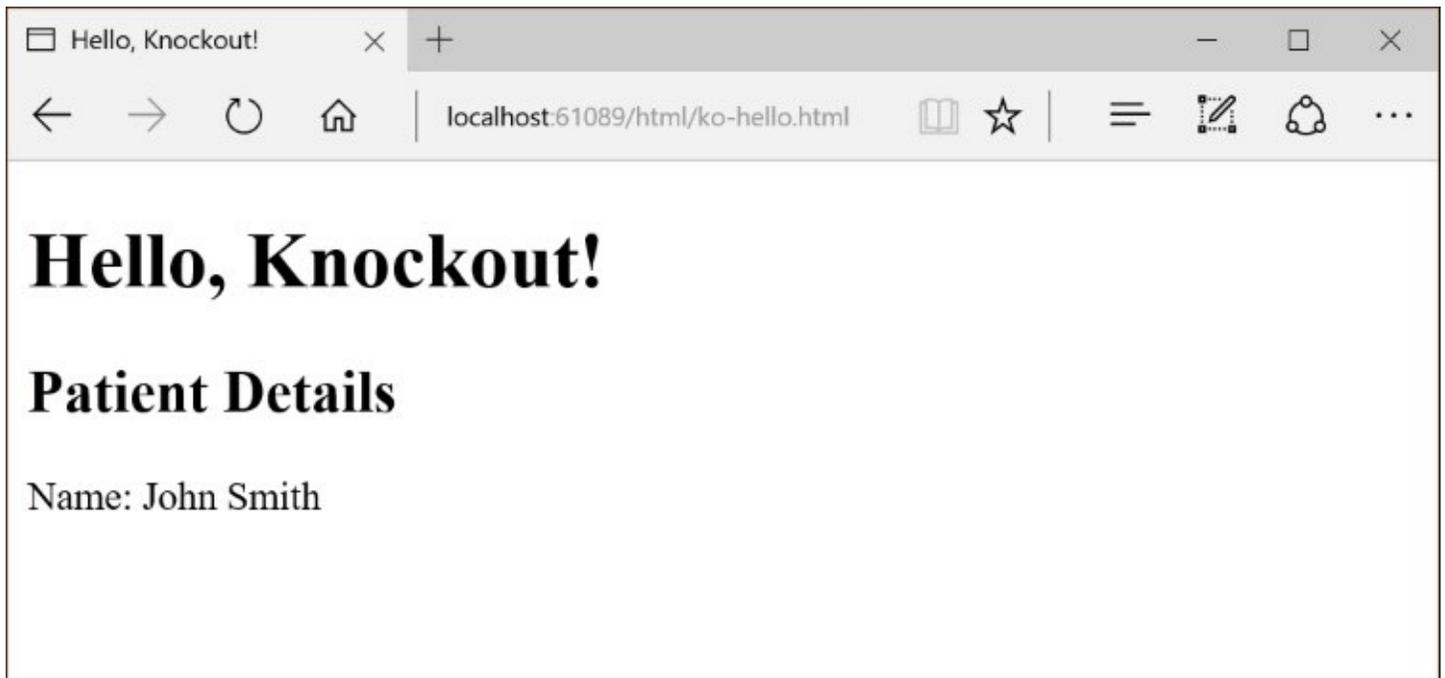
You will notice the following:

- A `<script>` tag identifies the location of KnockoutJS
- A `<span>` tag uses the `data-bind` attribute marked as a text value to display a `patientName`
- A view model is defined with one property, `patientName`
- `ko.applyBindings()` is called by passing the data model to establish the connection

To jump right into a Web API example, the following code makes a call to a Web API method to get a list of patient records in the sample file `ko-rest.html`:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Web API from KnockoutJS</title>
    <script type="text/javascript" src="../lib/jquery/dist/jquery.js"></script>
    <script type="text/javascript" src="../lib/knockout/dist/knockout.js">
</script>
</head>
<body>
    <h1>Web API from KnockoutJS</h1>

    <h2>Patient Details</h2>


    <ul data-bind="foreach: Patients">
        <li>
            <span data-bind="text: lastName"></span>,
            <span data-bind="text: firstName"></span>
        </li>
    </ul>
```

```
    <script type="text/javascript">
    var patientViewModel = function () {
        var self = this;
        self.Patients = ko.observableArray();
        LoadPatientData();

        // NOTE: the port number below should be changed as necessary
        function LoadPatientData() {
            $.ajax({
                type: "GET",
                url: "http://localhost:50915/api/Patient",
                contentType: "application/json; charset=utf-8",
                dataType: "json",
                success: function (data) {
                    self.Patients(data);
                },
                error: function (error) {
                    console.log("An error has occurred.");
                }
            });
        }
    };
    ko.applyBindings(new patientViewModel());
    </script>

</body>
</html>
```

The following screenshot shows the output of `ko-rest.html`:
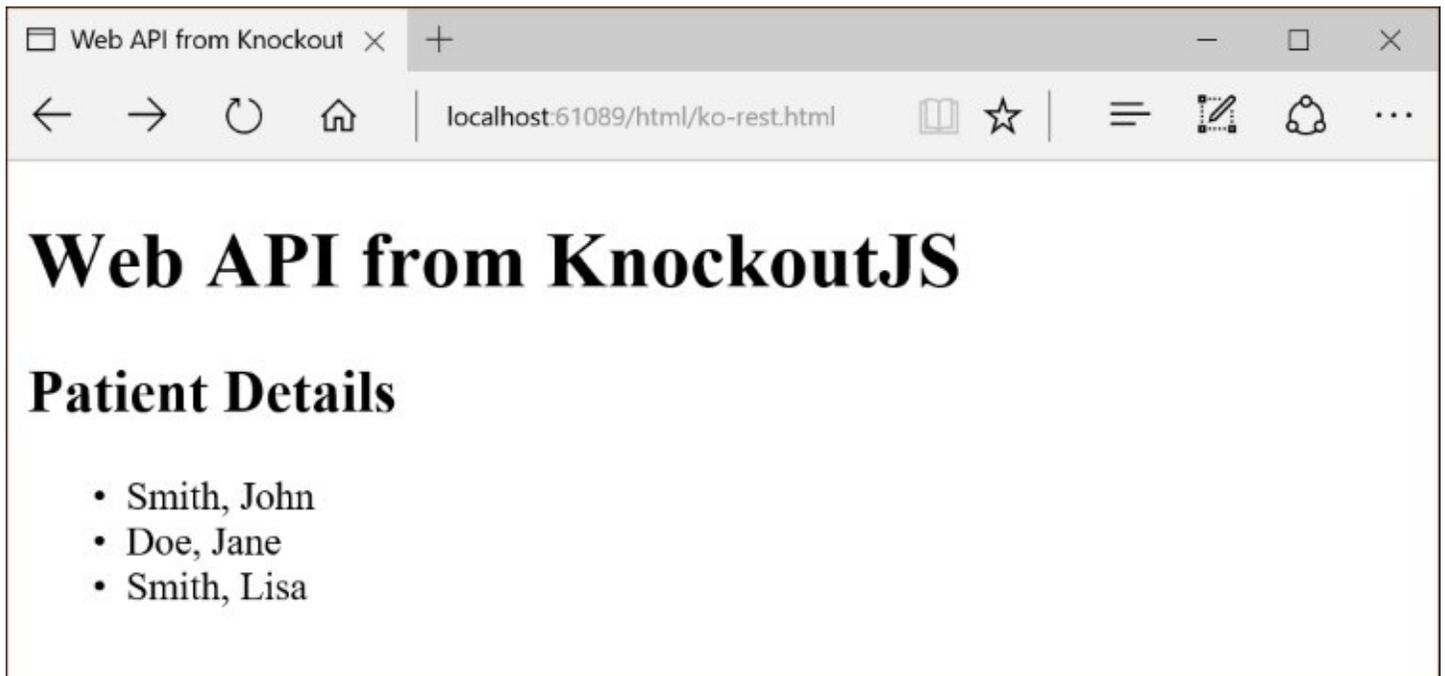
In the preceding example, you will notice the following:

- A `<script>` tag identifies the location of jQuery
- A second `<script>` tag identifies the location of KnockoutJS
- A `<ul>` tag uses the `data-bind` attribute to enumerate a list of patients
- Each `<li>` contains `<span>` tags to bind to `lastName` and `firstName`
- A basic `ViewModel` is defined to hold an `observableArray()` of patients
- A function named `LoadPatientData` calls the Web API
- Finally, `ko.ApplyBindings()` is called with the `ViewModel` for binding

Once again, you must make sure that the Web API is up-and-running if it is in a different project.

Task runners, bundling, and minification using Bower, Grunt, and Gulp

With Visual Studio 2015, Microsoft has introduced tighter integration with client-side package managers and task runners. In order to make the most of these tools, it is advisable to learn about what they do and when to use them.

Why do we need task automation?

You could write code line by line in a basic text editor without any IntelliSense, but you would soon be itching for Visual Studio or a rich IDE to get more work done. This is analogous to performing certain tasks manually, when you could be using automated tools to make your life easier.

Using Bower as your package manager

To work with client-side dependencies, you need to work with a package manager that works well with your development environment. Bower is such a tool.

Instead of using **NuGet** to install client-side packages such as jQuery, you will be using Bower to obtain JavaScript libraries and CSS frameworks. Instead of waiting for the latest versions of packages to be available on NuGet, you'll benefit from getting these packages using Bower, which is already being used by developers around the world.

If you create a new web project using the starter template, you will have a Bower configuration file already setup. To set it up from scratch, create a new empty web project, and a new configuration file to it, and follow these steps:

1. In Visual Studio 2015, click on **File** | **New** | **Project**.
2. Select **ASP.NET Core Web Application** and enter a project name and location.
3. Select the **Empty** template for ASP.NET Core.
4. Right-click on the web project then and click on **Add** | **New Item**.
5. Under the **Client-side** category, add new Bower configuration file, typically named `bower.json`.

In `bower.json`, add a new dependency for bootstrap as follows:

```json
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7"
  }
}
```

Use IntelliSense to decide which version of bootstrap you would like to use. Almost immediately, the `lib` subfolder within `wwwroot` will be populated with the necessary files for **bootstrap** (the dependency you added) and **jquery** (which bootstrap is dependent on), as shown in the following screenshot:
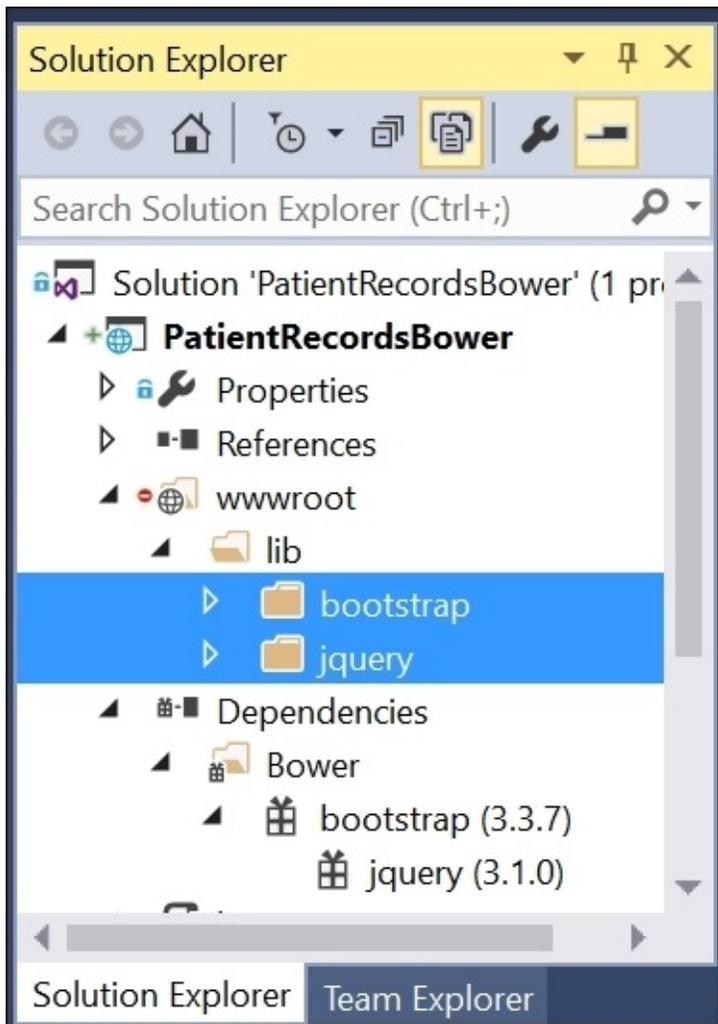
The location has been configured based on the entry in the `.bowerrc` file, which sits alongside `bower.json`, but usually tucked underneath it, in Solution Explorer. The following code is from `.bowerrc`:

```
{
    "directory": "wwwroot/lib"
}
```

For more information on using Bower as a client-side package manager for ASP.NET Core projects, take a look at the official documentation at http://docs.asp.net/en/latest/client-side/bower.html .

Using Gulp and Grunt as task runners

To set up Gulp and Grunt in your project, add a new item at the project level of type **NPM Configuration File** with the name `package.json`. Then, update the `devDependencies` section to include the following references to Gulp and Grunt:

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-less": "3.1.0",
    "grunt": "1.0.1",
    "grunt-contrib-clean": "1.0.0"
  }
}
```

In the Solution Explorer, verify that these dependencies are now visible in the `NPM` folder within your dependencies. Then, add a **LESS** file named `styles.less` into the `less` subfolder at your project's root, with the following LESS code:

```
@light-gray: #C0C0C0;
@darker-gray: @light-gray - #222;

#banner {
  color: @darker-gray;
}
```

This LESS code will auto-generate a darker shade of gray for an HTML element that uses an ID value of `banner`, such as `<div id="banner">`.

Add two new items to your project of types **Gulp Configuration File** and **Grunt Configuration File**, typically named `gulpfile.js` and `gruntfile.js`, respectively.

Add the following code to `gulpfile.js` to watch your LESS files and compile CSS output when there are changes. The directive in the first line also instructs Visual Studio to run each task based on specific conditions, such as after a build or when the project is opened:

```
/// <binding AfterBuild='stylemaker' ProjectOpened='lesswatcher' />
var gulp = require('gulp');
var gulpless = require('gulp-less')

gulp.task('stylemaker', function () {
    gulp.src("./less/styles.less")
        .pipe(gulpless({ compress: true }))
        .pipe(gulp.dest("./wwwroot/css"));
});

gulp.task('lesswatcher', function () {
    gulp.watch("./less/*.less", ["stylemaker"]);
});
```

Next, add the following code to `gruntfile.js` to ensure that the CSS output folder is cleaned:

```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/css/*"],
    });
    grunt.loadNpmTasks("grunt-contrib-clean");
};
```

You can use the **Task Runner Explorer** window to run the preceding tasks, or configure bindings to enable the tasks to run based on specific conditions. If the **Task Runner Explorer** window is not visible, click on **View** | **Other Windows** | **Task Runner Explorer** from the top menu of Visual Studio. The following screenshot shows the **Task Runner Explorer** window:
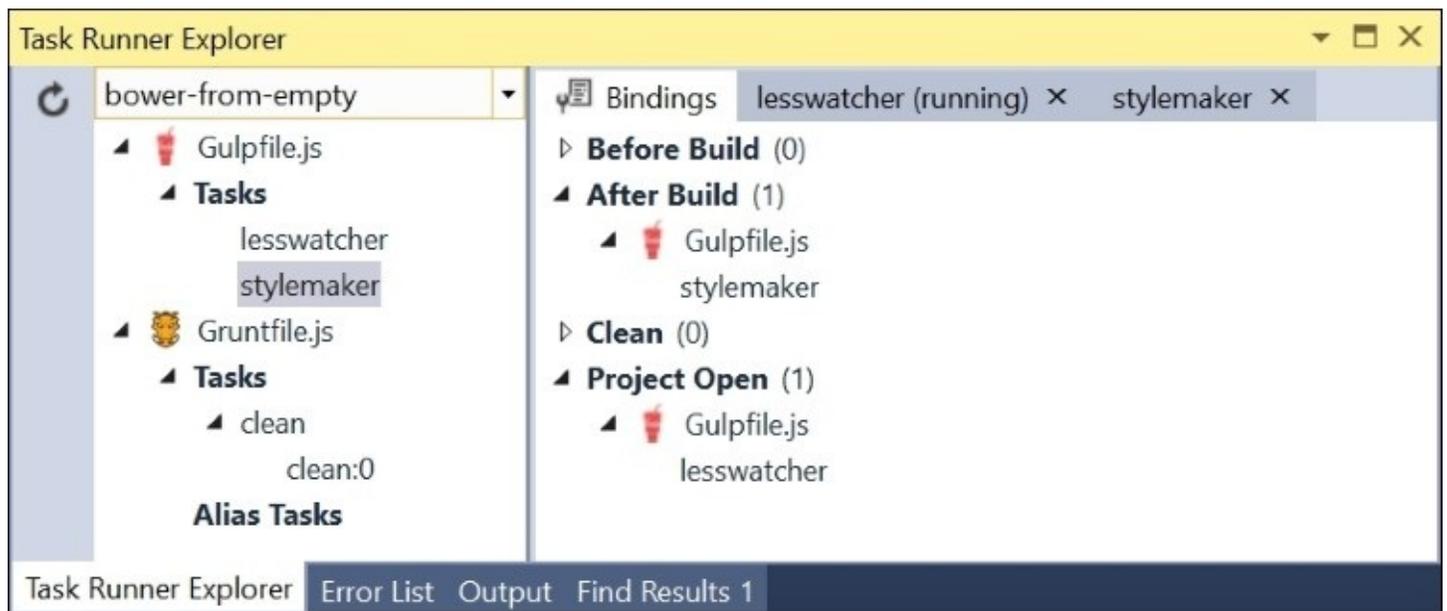
For more information on using Grunt and Gulp as task runners for ASP.NET Core projects, take a look at the official documentation at the following URLs:

- http://docs.asp.net/en/latest/client-side/using-gulp.html
- http://docs.asp.net/en/latest/client-side/using-grunt.html

Summary

In this chapter, you learned how to use a Web API with JavaScript, after which you covered the basics of AngularJS and KnockoutJS. You also learned about client-side tools that help automate some common tasks while building an ASP.NET web application in Visual Studio 2015.

In the next chapter, you will learn how to build database-driven web applications without having to write SQL code. Instead, you will use **Entity Framework**, an object-relational mapper that allows you to write code to represent your database entities.

Chapter 6. Using Entity Framework to Interact with Your Database in Code

In [Chapter 3](#) , *Understanding MVC,* we mentioned **Entity Framework** (**EF**) while building a functional ASP.NET Core MVC web application. In this chapter, we'll go deeper into EF Core 1.0 and cover what you need to know about EF Core, that is, EF7 during development.

It's worth noting that EF Core can be used with more than just ASP.NET Core and relational databases. In fact, it has also been designed to be used with **Universal Windows Platform** (**UWP**) apps on Windows 10 and Windows desktop apps (**Windows Presentation Foundation** (**WPF**) or **WinForms**). Going beyond SQL Server, EF Core can also be used with **Azure Table Storage**, **PostgreSQL**, **SQLite**, and even **NoSQL** databases.

For the purposes of this book, we will focus primarily on ASP.NET Core, SQL Server, and the new in-memory provider.

Object-relational mapping in .NET

If you've used EF or any **object-relational mapper** (**ORM**) earlier, you may skim this section and move on to the next section. But if you're new to ORM frameworks, this section is for you.

An ORM such as EF makes it easy for you to interact with your database from within your application code. The following screenshot illustrates ORM:
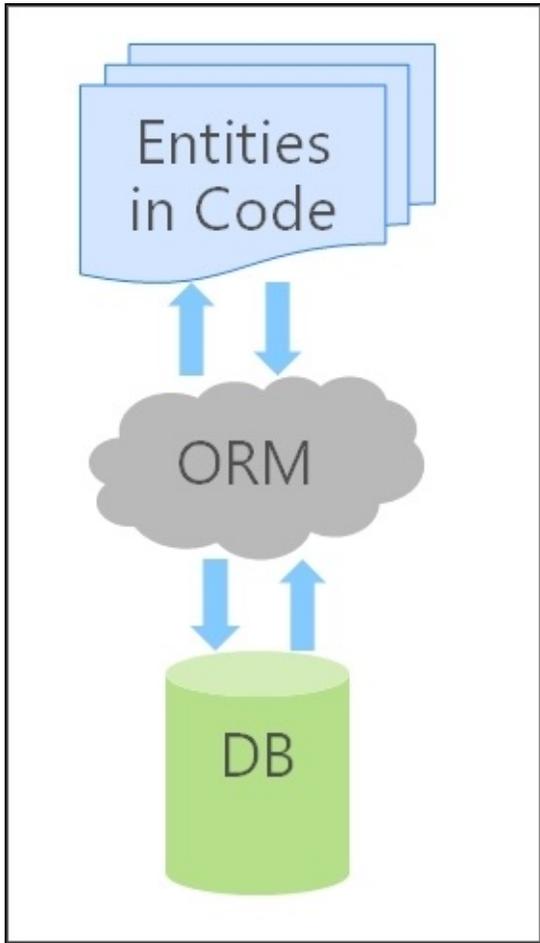
Why use an ORM?

It's perfectly feasible to build a fully functional web application without any ORM at all. You could use straight ADO.NET or raw SQL to talk to your database from your application code. You could write SQL code to create your database objects and relationships. You could keep all your database logic in the database by heavy use of stored procedures and functions or you could use an ORM such as EF and take advantage of a **Code First** approach.

While database administrators might scoff at the use of an ORM, developers should feel confident that they will gain more power and control of how their database entities are created, manipulated, and maintained over time. The following is a list of pros and cons you may consider.

Pros of using an ORM:

- **Common codebase**: Instead of having disconnected database creation code outside your application codebase, you can define all your entities in your code
- **Compile-time benefits**: If you make certain mistakes in your model code, the compiler will catch them
- **Easier updates**: If you need to update/rename anything in your models, your IDE makes it a breeze
- **Worry-free constraints**: You can easily set up your keys, relationships, and other constraints using attributes and fluent code
- **Manageable maintenance**: With the help of migrations, you and your team can upgrade/downgrade your database as needed and keep track of revisions.

Cons of using an ORM:

- **Learning curve**: If you haven't used an ORM before, you can't just add it to an existing application in a day. Start small use it for newer projects and take your time when updating existing projects.
- **Complex queries**: If you or someone in your team is already proficient in SQL queries, you may have difficulties rebuilding the results of complex queries using **Language-Integrated Query** (**LINQ**) syntax. Over time, it may get easier for you or you may decide that ORMs are not right for a specific application.
- **Performance (arguable)**: When ORMs were first introduced, you might have considered the performance implications of adding yet another layer between your application code and the database. But with modern ORM frameworks such as EF Core, the performance improvements should make it worth the effort, with less cause for concern.

Now that we've got the pros and cons out of the way, let's focus on why you would pick EF as the ORM of choice for your web application.

Why Entity Framework?

If you've worked with ORMs earlier, you may have used **NHibernate** or an earlier version of EF. You may have used **Hibernate** within a Java application. Regardless of your background, there are plenty of reasons why EF is a good choice for your ASP.NET Core applications.

First of all, the new EF goes with the new ASP.NET like no other version before it. Earlier versions of the EF runtime and tooling were released alongside the .NET framework, but more recent versions separated out the runtime, which enabled out-of-band releases through NuGet. The latest versions of the runtime can be obtained through NuGet, while major releases are synced with Visual Studio releases.

The evolution of Entity Framework

EF has improved a lot over time. The introduction of a Code First approach and EF Migrations made many developers take note. At the same time, there are plenty of developers who have yet to work with EF.

Over the years, the EF team has continued to collect feedback from developers who have been garnering experience with their product. As a result, the team has put together a mature product that has only gotten better with time.

If you are interested in providing feedback or seeing input from other developers, head on over to https://data.uservoice.com .

Some of the new features that emerged include batch updates, shadow properties, an in-memory provider, and improved methods of working with disconnected data. We will discuss new features in more detail in this chapter, with a sample project to illustrate how it all comes together.

EF 6.x for .NET Framework versus EF Core 1.0

If you've been using EF 6 until now, you should take the time to learn about what has changed in EF Core. The basic concepts and usage of EF should be familiar, but there are some new additions (and some removals), which will affect your learning curve for picking up EF Core.

If you have been doing Code First development, you should already be familiar with building your own models in your application code. If you have already used Migrations, you should also be familiar with generating/updating your database by triggering a migration. However, if you have gotten used to visual data modeling tools and EDMX files, you may have to unlearn a few things. EDMX files are Entity Data Model files that can be used to represent your data model visually, using earlier versions of EF.

What's different in EF Core

First of all, it's worth noting that EDMX files are no longer supported as a means of creating your data model. This doesn't mean that there are no ways to visually model your database or start with a database-first approach. On the contrary, you can still start off with an existing database and use external tools to generate your model classes if you choose.

That being said, we will focus on a Code First approach with EF Core. This is important because it will help you maintain better control of your model classes, which will make code merges much easier.

Prior versions of EF relied on `ObjectContext` as a means of working with your database entities as objects in your code. Even with the **DbContext API** in EF 4.1, there was still continued reliance on the original `ObjectContext` underneath. In EF Core, `ObjectContext` will leave us, but `DbContext` remains. `DbContext` represents a combination of design patterns that allow you to work with your database easily from your application code.

Although EF6 was available through NuGet, EF Core takes its dependency management to another level in keeping with ASP.NET Core. You can choose to use just the pieces you need, including separate packages for Core, Commands, Relational (includes migrations), and SQL Server. If you include a specific package as a reference, Visual Studio 2015 will automatically include additional dependencies, as needed.

Getting started with EF Core

If you create a new Web Application project in Visual Studio 2015 using the standard Web Application template, you should already have EF Core set up. You may recall the Patient Records web app from [Chapter 3](#) , *Understanding MVC*, in which we set up a basic web project with database connectivity.

If you start with an empty project, you should type in your EF references in your `project.json` configuration file. In my Patient Records app, I have the following reference:

`"Microsoft.EntityFrameworkCore.SqlServer": "1.0.0"`

In the Solution Explorer panel, you can expand each reference to identify related references that were pulled in automatically as dependencies, as shown in the following screenshot:
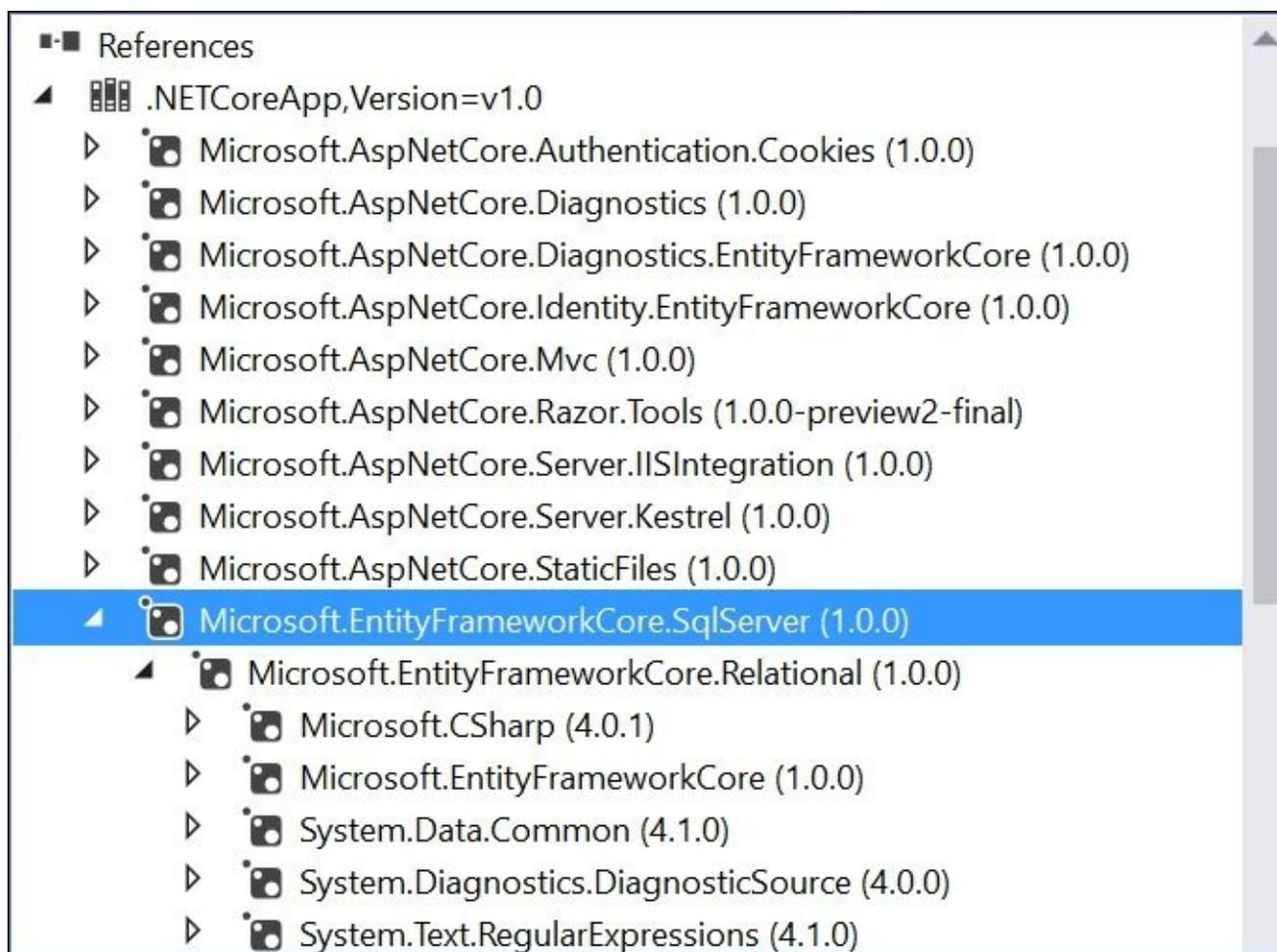
Depending on when you add your references, your version numbers may vary. Although you can type in any current or earlier version in your configuration file, you will most likely let Visual Studio's IntelliSense feature guide you with little pop-up tooltip suggestions.

In the `Startup.cs` file, you should have a call to `AddEntityFrameworkStores()` within the `ConfigureServices()` method, followed by calls to add a specific database provider and a database context to work with. To use EF Core, first ensure that the following using statement appears in your `Startup.cs` file:

```
using Microsoft.EntityFrameworkCore;
```

In my Patients Record application, the `ConfigureServices()` method reads as follows:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // rest of method removed for brevity
}
```

The preceding code contains the following terms and method calls:

- `Microsoft.EntityFrameworkCore`: This namespace provides access to EF Core
- `AddDbContext<>()`: This adds a db context called `ApplicationDbContext`
- `UseSqlServer()`: This configures an SQL Server database connection

The database context lives in its own class file, `ApplicationDbContext.cs`, in your `Data` folder. This is where you will add one or more `DbSet` entries to represent your entities.

The database connection string can exist as a text value in your `appsettings.json` file, which lives in the root of your project location. This connection string can also exist as an environment variable for your specific environment, whether it's your development machine, an on-premise server, or a cloud environment.

Once your references and your configuration files have been set up, you are now ready to use EF in your code.

What else is new?

Before we get into the rest of the code, let's take a look at what else is new with EF Core:

- **Migrations History**: In prior versions of EF, each new migration (that is, database revision) would generate a new entry into a special table in the database. Such a table still exists with EF Core and is called `__EFMigrationsHistory`, but it has been simplified to include only an alphanumeric migration ID, in addition to the product version.
- **Snapshots**: The migration history table used to include snapshots of each migration, but this has been moved to a code file in your `Migrations` subfolder within a `Data` folder, such as `ApplicationDbContextModelSnapshot.cs`. The actual filename may vary depending on the name of your database context.
- **Batch updates**: EF Core generates SQL statements to run database commands based on your application code. You can configure a maximum batch size when setting up your database context.
- **Shadow properties**: You can used shadow properties to dynamically extend your model from within your `OnModelCreating()` method in your database context. These changes will make it to your database, but will not affect your actual model files.
- **In-memory providers**: Instead of interacting directly with your actual database, you can choose to use an in-memory provider. This will also come in handy when setting up and running unit tests.
- **Disconnected data**: There are improved ways to work with disconnected data, which should make it easier for developers to add and update data in disconnected-data scenarios.

Code First approach to database design and relationships

So what exactly is a Code First approach? It is exactly what it sounds like. You can model your database objects as entity classes in your code. To establish relationships between those objects, you can define a class to include other classes as member variables. If you already have an existing database, you can create an entity model in your code to represent some (or all) of your database objects.

From our Patient Records example from Chapter 3 , *Understanding MVC*, we already have a model class that represents a human. In this chapter, we will add a `RobotDoctor` class to our project to build a computerized system for a futuristic hospital with robot doctors. Then, we will establish a relationship between `Humans` and `RobotDoctors` in the code so that each robot doctor can be assigned to one or more human patients in the database.

Updating the models

In the `Models` folder of the sample project from [Chapter 3](#) , *Understanding MVC,* let's add a new model class to represent a `RobotDoctor` and then update the existing `Human` class to add fields to recognize each human's relationship with a robot doctor. We will also have to update our context file, `ApplicationDbContext.cs`, with a separate `DbSet` for each set of entities.

To add the `RobotDoctor` class, follow these steps:

1. In the Solution Explorer panel, right-click on the `Models` folder.
2. Click on **Add** | **New Item** in the context menu.
3. Name the new class `RobotDoctor.cs`.
4. Add the following code for the `RobotDoctor` class:

```
using System.ComponentModel.DataAnnotations;

namespace PatientRecords.Models
{
    public class RobotDoctor
    {
        [Display(Name = "Robot Doctor ID")]
        public int RobotDoctorId { get; set; }

        [Display(Name = "Model Number")]
        public int ModelNumber { get; set; }

        [Display(Name = "Preferred Name")]
        public string PreferredName { get; set; }
    }
}
```

The `RobotDoctorId` integer field will be used as the primary key. The remaining fields will store and display the model number and preferred name, respectively. The `Display` attributes will be used as friendly text labels for the fields.

To update the Human class, follow these steps:

1. Open the `Human.cs` class file from the `Models` folder.
2. Add the following fields to the bottom of the class:

```
[Display(Name = "Robot Doctor")]
public int RobotDoctorId { get; set; }
public RobotDoctor RobotDoctor { get; set; }
```

The `RobotDoctorId` integer field in the `Human` class will be used as a foreign key to refer to the corresponding field in the `RobotDoctor` class. The `RobotDoctor` object will allow each `Human` to be assigned to a specific `RobotDoctor`. The `Display` attribute will be used as a friendly text label to be used for the field.

To update the database context, follow these steps:

1. Open the `ApplicationDbContext.cs` class file from the `Data` folder.
2. Add a new `DbSet` for `RobotDoctors` at the bottom of the class.
3. Update the `DbSet` for `Humans` by pluralizing the word `Human` to `Humans`, shown in the following code:

```
public DbSet<RobotDoctor> RobotDoctors { get; set; }
public DbSet<Human> Humans { get; set; }
```

The `DbSet` named `RobotDoctors` will represent the set of robot doctors stored in the database. The `DbSet` named `Humans` will represent the set of `Humans` stored in the database. While it is not necessary to pluralize it, this minor change should help clarify its purpose.

Updating the controllers

When we updated `DbSet` named `Human` to `Humans` in the `ApplicationDbContext` class, our `HumanController` will still retain references to the old name. There are two ways to fix this: you could rename all `Human` references to `Humans` manually in the controller file or you could use Visual Studio's built-in rename feature.

To use the rename feature, you can click on `DbSet` named `Human` in your code (before renaming `Human` to `Humans`) and then follow up with a rapid set of keystrokes: *Ctrl + R + R*. This would allow you to type in the new value everywhere it is being referenced. You will get a chance to preview your changes by checking the appropriate checkbox that appears, as shown in the following screenshot:
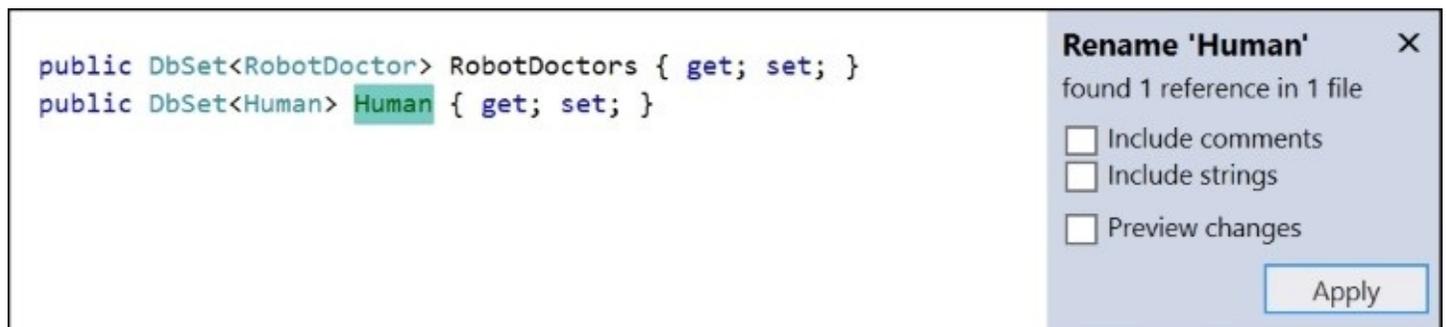
Before we update the rest of the controller methods, ensure that the following using statements are at the top of the `HumanController` class:

```
using System.Threading.Tasks;
using System.Collections.Generic;
using System;
```

The following namespaces are useful for the tasks we will perform in the controller:

- The `Threading.Tasks` namespace will allow us to use asynchronous methods.
- The `Collections.Generic` namespace will allow us to use an **IEnumerable** list.
- The `System` namespace will allow us to use `String` methods.

To ensure that we include `RobotDoctor` data for each `Human` in a list of `Humans`, update the `Index()` method of the `HumanController` class to include the following code:

```
public async Task<IActionResult> Index()
{
    var humans = _context.Humans.Include(h => h.RobotDoctor);
    return View(humans);
}
```

In the preceding code, the call to the `DbSet` named `Humans` within the database context is pulled

along with `RobotDoctor` data by the `Include()` method. The method then returns the set of humans to the corresponding view. The view is then responsible for iterating through a list of `Humans` to display to the user.

To ensure that we include `RobotDoctor` data for a specific `Human` in its `Details` page, update the `Details()` method to include the following code:

```
public async Task<ActionResult> Details(int? id)
{
    Human human = await _context.Humans
        .Include(h => h.RobotDoctor)
        .SingleOrDefaultAsync(h => h.ID == id);
    if (human == null)
    {
        return NotFound();
    }
    return View(human);
}
```

Note that the `DbSet` named `Humans` is followed by a call to `Include()` to make sure that `RobotDoctor` data is also included. While we're at it, we are also taking advantage of asynchronous method calling by including the following:

- async keyword in the method definition of the `Details()` method
- `Task<ActionResult>` return type instead of just `ActionResult`
- `SingleOrDefaultAsync()` instead of just `Single()`
- `await` keyword in async method call

Next, let's update the `HttpGet` version of the `Create()` method to get a list of `RobotDoctors` before displaying an entry form to create new `Humans`. Since `HttpGet` is the default verb for controller methods, the method does not need an attribute to indicate its **HttpVerb**. Replace `Create()` with the following code:

```
public IActionResult Create()
{
    ViewBag.RobotDoctors = GetListOfRobotDoctors();
    return View();
}

private IEnumerable<SelectListItem> GetListOfRobotDoctors(int selected = -1)
{
    var tmp = _context.RobotDoctors.ToList();

    // Create authors list for <select> dropdown
    return tmp
        .OrderBy(rb => rb.ModelNumber)
        .Select(rb => new SelectListItem
        {
            Text = String.Format("{0}: {1}", rb.ModelNumber, rb.PreferredName),
            Value = rb.RobotDoctorId.ToString(),
            Selected = rb.RobotDoctorId == selected
        });
```

}

In the preceding code, the `Create()` method is followed by the addition of the new private method to do the work of getting the data. The `Create()` method then returns the default view, `Create.cshtml`, to display an entry form.

Next, we also need to update the `HttpPost` version of the `Create()` method. This method will specifically have the `HttpPost` attribute above the method definition. Update this `Create()` method with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind("SocialSecurityNumber",
"DateOfBirth", "FirstName", "LastName", "RobotDoctorId")] Human human)
{
    if (ModelState.IsValid)
    {
        _context.Humans.Add(human);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(human);
}
```

Another thing to note is the `Bind` attribute within the `Create()` method's argument list. This attribute helps us identify exactly which model properties we would like to bind to the corresponding form fields.

Finally, let's update the `Edit()` methods with the following code, starting with the `HttpGet` version. As with the `Create()` method, the `HttpGet` version won't necessarily have any attribute above it to indicate the HttpVerb used for the controller method:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Human human = _context.Humans.Single(m => m.ID == id);
    if (human == null)
    {
        return NotFound();
    }

    ViewBag.RobotDoctors = GetListOfRobotDoctors();
    return View(human);
}
```

The primary change to this `Edit()` method is the addition of a dynamic `ViewBag` property named `RobotDoctors`, similar to the one seen in the `Create()` method. It uses the same private method to obtain a list of `RobotDoctors`. It returns the `Human` model to the view, `Edit.cshtml`,

which displays an entry form to edit an existing `Human` entry.

Finally, let's update the `HttpPost` version of the `Edit()` method. Replace this method with the following code to ensure that the `Human` model is updated with the passed data:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("SocialSecurityNumber",
"DateOfBirth", "FirstName", "LastName", "RobotDoctorId")] Human human)
{
    if (id != human.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        human.ID = id;
        _context.Humans.Attach(human);
        _context.Entry(human).State = EntityState.Modified;
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(human);

}
```

To notify EF Core of the modified state, the `State` property of the edited entry is set to `EntityState.Modified`. If there are any validation errors, the user is directed to the same `Edit.cshtml` view by passing in the updated `Human` model. If there are no validation errors, the user is redirected to the `Index.cshtml` view to display the current list of `Human` entries.

The `Delete()` methods do not need any additional changes, as the entity code generation takes care of the proper deletion of each entity. When a selected `Human` is deleted using the appropriate controller methods, it displays a confirmation screen prepared by the `HttpGet` version of the `Delete()` method. Once confirmed, the `HttpPost` version of the `Delete()` method takes care of the actual removal before calling `SaveChanges()` to update the database accordingly. After the deletion operation, the user is then redirected to the `Index` method to view a list of `Human` entries.

Updating the views

To complete the coding changes, you must update the `Views` for the `Human` controller. These include the following `.cshtml` files in your `Human` subfolder within the `Views` folder:

- Index
- Edit
- Details
- Create

In the `Index.cshtml` file, add the following table header `<th>` block to display a label for the robot doctor's preferred name, right after the SSN label:

```
<th>
    @Html.DisplayNameFor(model => model.RobotDoctor.PreferredName)
</th>
```

Within the `foreach` loop iterating through the items in the `Model`, add a table data `<td>` block to display the preferred name, right after the SSN field, which is used to store a unique **Social Security Number**. This should be just before the links for **Edit**, **Details**, and **Delete**:

```
<td>
    @Html.DisplayFor(modelItem => item.RobotDoctor.PreferredName)
</td>
```

In the `Edit.cshtml` file, update the `<form>` tag to include additional tag helper attributes:

```
<form
    asp-controller="Human"
    asp-action="Edit"
    method="post"
    asp-route-id="@Model.ID">
```

The attributes are useful for various reasons:

- `asp-controller` and `asp-action`: This indicates the controller name and method to submit the form to
- `method`: This indicates the method to submit your form with, usually `POST`
- `asp-route-id`: This indicates the value of ID to associate with the submission

Below the validation summary within the form, remove the hidden ID field:

```
<input type="hidden" asp-for="ID" />
```

Instead, replace it with the following code:

```
<div class="form-group">
    <label
        asp-for="RobotDoctorId"
        class="col-md-2 control-label"></label>
    <div class="col-md-10">
```

```
            <select
                asp-for="RobotDoctorId"
                asp-items="@ViewBag.RobotDoctors"></select>
    </div>
</div>
```

The preceding code displays a label for the `RobotDoctorId` field, followed by a drop-down of `RobotDoctors`. You may recall that the value of `ViewBag.RobotDoctors` is being set in multiple controller methods.

In the `Details.cshtml` file, add the following description list entries within pairs of `<dt>`/`<dd>` blocks to display labels and fields for each robot doctor's ID, model number, and preferred name, right after the SSN label:

```
<dt>
    @Html.DisplayNameFor(model => model.RobotDoctor.RobotDoctorId)
</dt>
<dd>
    @Html.DisplayFor(model => model.RobotDoctor.RobotDoctorId)
</dd>
<dt>
    @Html.DisplayNameFor(model => model.RobotDoctor.ModelNumber)
</dt>
<dd>
    @Html.DisplayFor(model => model.RobotDoctor.ModelNumber)
</dd>
<dt>
    @Html.DisplayNameFor(model => model.RobotDoctor.PreferredName)
</dt>
<dd>
    @Html.DisplayFor(model => model.RobotDoctor.PreferredName)
</dd>
```

The preceding entries should be added just before the `</dl>` list is closed.

Similar to the `Edit` view, you must update the `Create` view with a new label and a drop-down to display the list of `RobotDoctors`. Below the validation summary within the form, add the following form-group:

```
<div class="form-group">
    <label
        asp-for="RobotDoctorId"
        class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <select
        asp-for="RobotDoctorId"
        asp-items="@ViewBag.RobotDoctors"></select>
  </div>
</div>
```

Once again, the dynamic value of `ViewBag.RobotDoctors` is being set in the appropriate controller method. Now that all the necessary changes have been made to the models,

controllers, and views, you are ready to update the database using EF Migrations.

EF Code First migrations for database versioning and maintenance

In EF, you can use migrations to facilitate the creation, upgrade, and downgrade of your database. You can also use the automatically generated version history to keep track of changes and stay in sync with the rest of your development team.

We have previously set up migrations in our sample project to get the ball rolling. In this section, we will add a new migration to reflect the changes we made in this chapter.

Setting up migrations

The following is a recap of how we previously set up our migrations for EF. Once again, keep in mind that your DNX version may vary:

1. Open a command prompt to your project folder's location.
2. Run the following commands:

```
>dotnet restore
>dotnet build
>dotnet ef migrations add Initial
>dotnet ef database update
```

The preceding commands do not need to be run again, as your initial migration has already been created back in Chapter 3 , *Understanding MVC*. Instead, we will run additional commands to create a new migration to represent the changes to your models and database context.

Adding and removing migrations

When you add a new migration, several things will happen:

- A new class file will be generated in the `Migrations` folder, below the initial migration file. The filename will usually be prefixed with a date/timestamp, followed by the name of the migration, such as `2016MMDDXXYY_RobotDoctors.cs`.
- The contents of the migration file will contain a single class that bears the name of the migration, such as `RobotDoctors`. This class will contain two methods, `Up()` and `Down()`, to assist in the upgrade and downgrade of the database, respectively.
- An auto-generated snapshot file will be updated to reflect the current state of the database models and relationships. The filename will typically be prefixed with the name of the database context, such as `ApplicationDbContextModelSnapshot.cs`.
- After the update command has been run, the physical database will be updated with the appropriate changes. If the changes cannot be processed successfully for some reason, one or more error messages will be returned.

To add a new migration, run the following commands in a command prompt within the project folder's location:

```
>dotnet restore
>dotnet build
>dotnet ef migrations add RobotDoctors
>dotnet ef database update
```

This generates a new migration, as described previously. If you get any database conflict errors, you may have to delete records in you `Humans` table first and then rerun the database updated command. You can now inspect the database to check for the new changes. To find the database quickly, use the **SQL Server Object Explorer** panel to drill down to your database and inspect the tables and fields.

To take a step back, you can use the `remove` command to remove the previous migration. The full list of commands is as follows:

- `add`: This will add a new migration
- `apply`: This will apply migrations to the database
- `list`: This will display a list of migrations
- `script`: This will generate a list of SQL scripts without actually updating the database
- `remove`: This will remove the previous migration

It is worth noting that the script command is similar to a script parameter in previous versions of EF. Instead of hiding this feature within a parameter (and possibly risking accidental overwriting of your database if you forgot to include it), you now have separate commands to either update your database or just generate SQL scripts without changing anything.

Running your application

Now that you have completed all the necessary database changes with the use of a new migration, you are ready to run your application. Build your solution and run the application from Visual Studio 2015. This should launch your default web browser with the main `Index` page from the `Home` controller, as shown in the following screenshot:
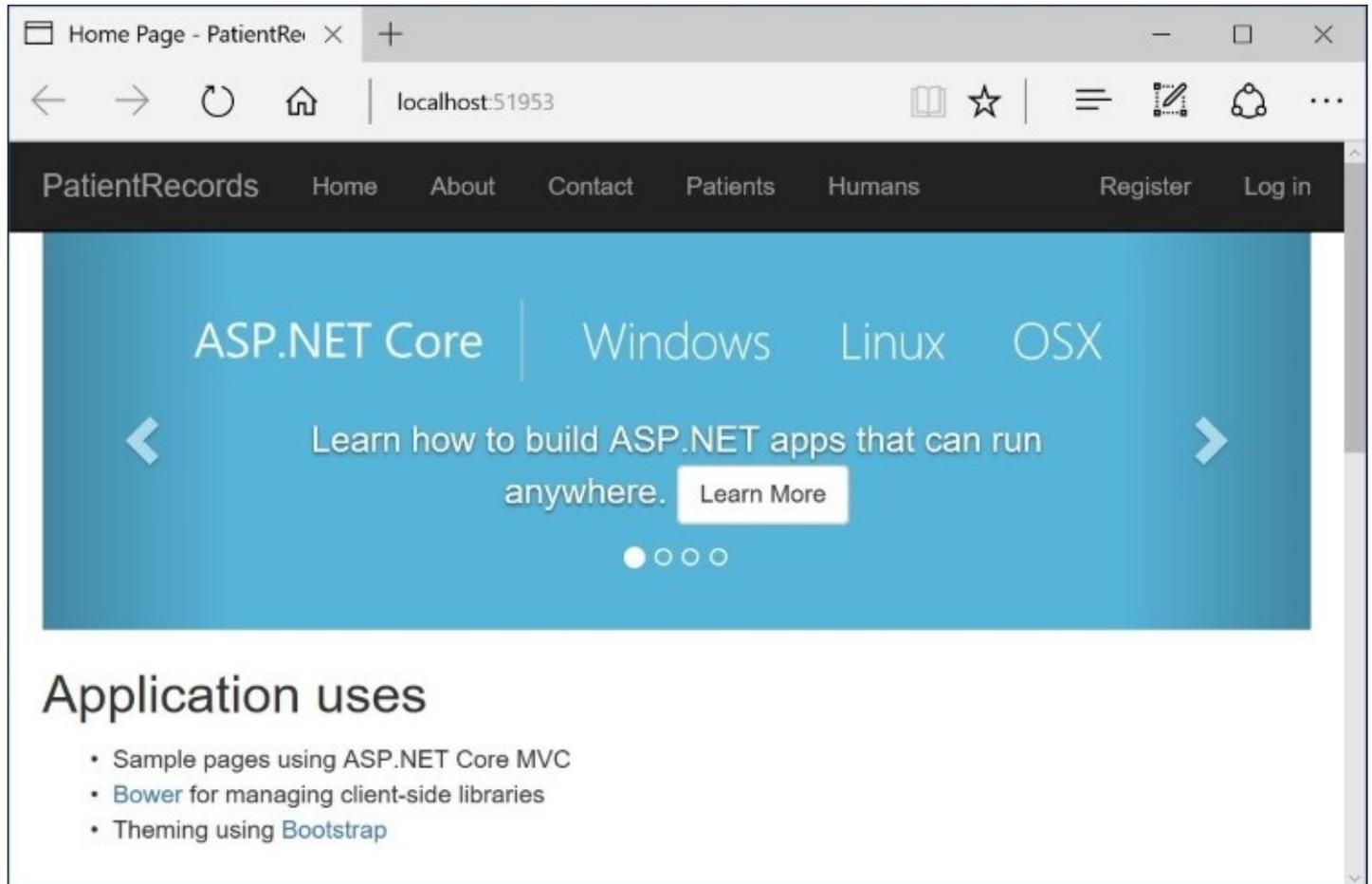
On the top menu, click on the **Humans** link to view the `Index` page of the `Human` controller. This should show you an empty list, as we have not created any new `Humans` at this time. If you already created your own entries prior to reaching this step, please delete your entries before you proceed.

The following screenshot is the `Index` page of the `Human` controller:
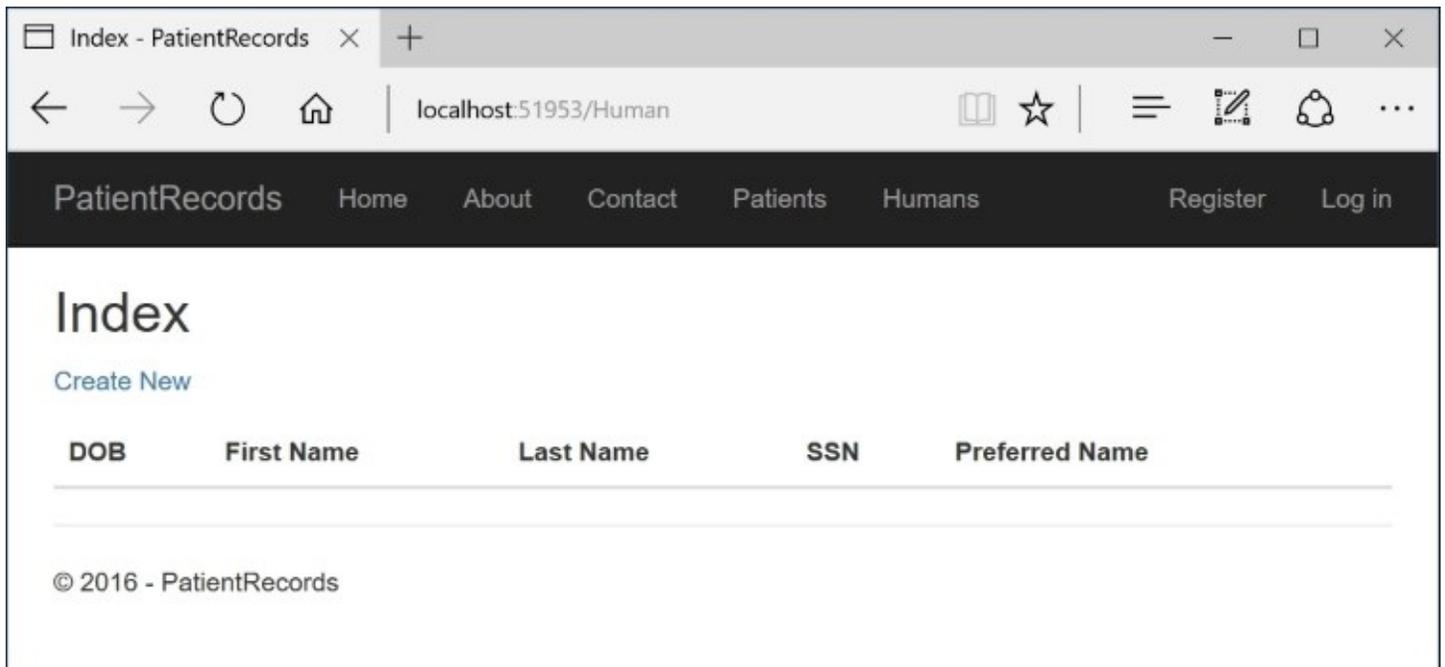
To ensure that we have some sample data, open the `RobotDoctors` table and add a few entries manually into the database using the **SQL Server Object Explorer** in Visual Studio. The following screenshot shows some sample values for `RobotDoctorId`, `ModelNumber`, and `PreferredName`:

RobotDoctorId (auto)	ModelNumber	PreferredName
1	90210	Dr. X
2	12345	Dr. Chicago

We could write some additional code to add seed data to your database, but we will save this step for Chapter 7 , *Dependency Injection and Unit Testing for Robust Web Apps*. It is better to use a testing framework to add seed data instead of adding it to your application code.

Back in your web browser, click on the **Create New** link in the top-left area to view the `Create` view for `HumanController`. Select a **Robot Doctor**, and then complete and submit the form to create a new **Human** entry. If there are any validation errors, you will be prompted to correct them before the submission goes through. Once submitted, you should be redirected to the `Index` page. The following screenshot shows the `Create` view:
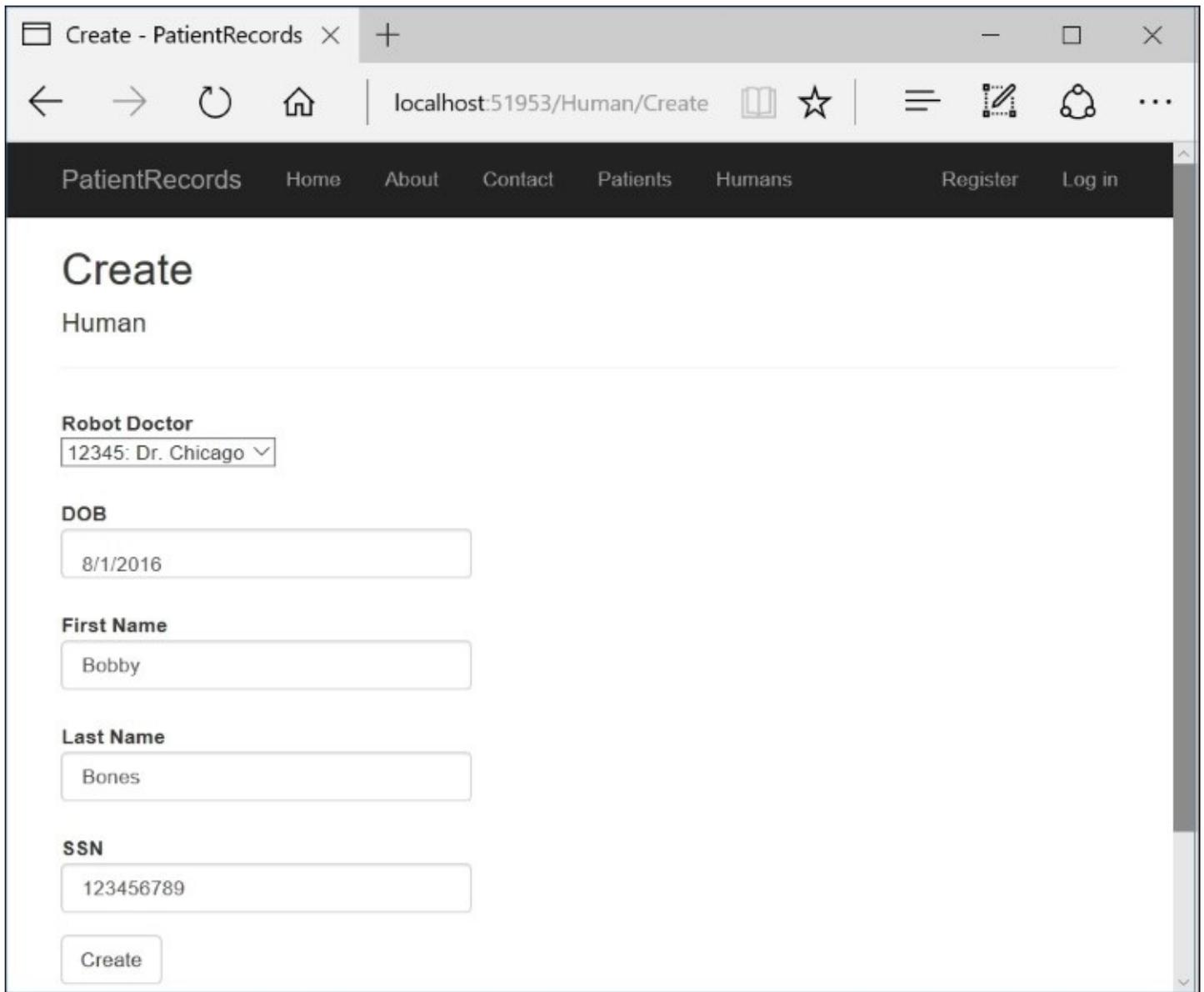
On the `Index` page in the following screenshot, you should see links labeled **Edit**, **Details**, and **Delete.** Let's try out each of these links to edit an entry, view the details of an entry, or delete an existing entry:

If you click the `Edit` link, you should now see the editable details of a previously added entry, as shown in the following screenshot. Change a few values and submit the form. Similar to the creation of a new entry, you will either be prompted to fix any validation errors, or taken back to the `Index` page upon submission:
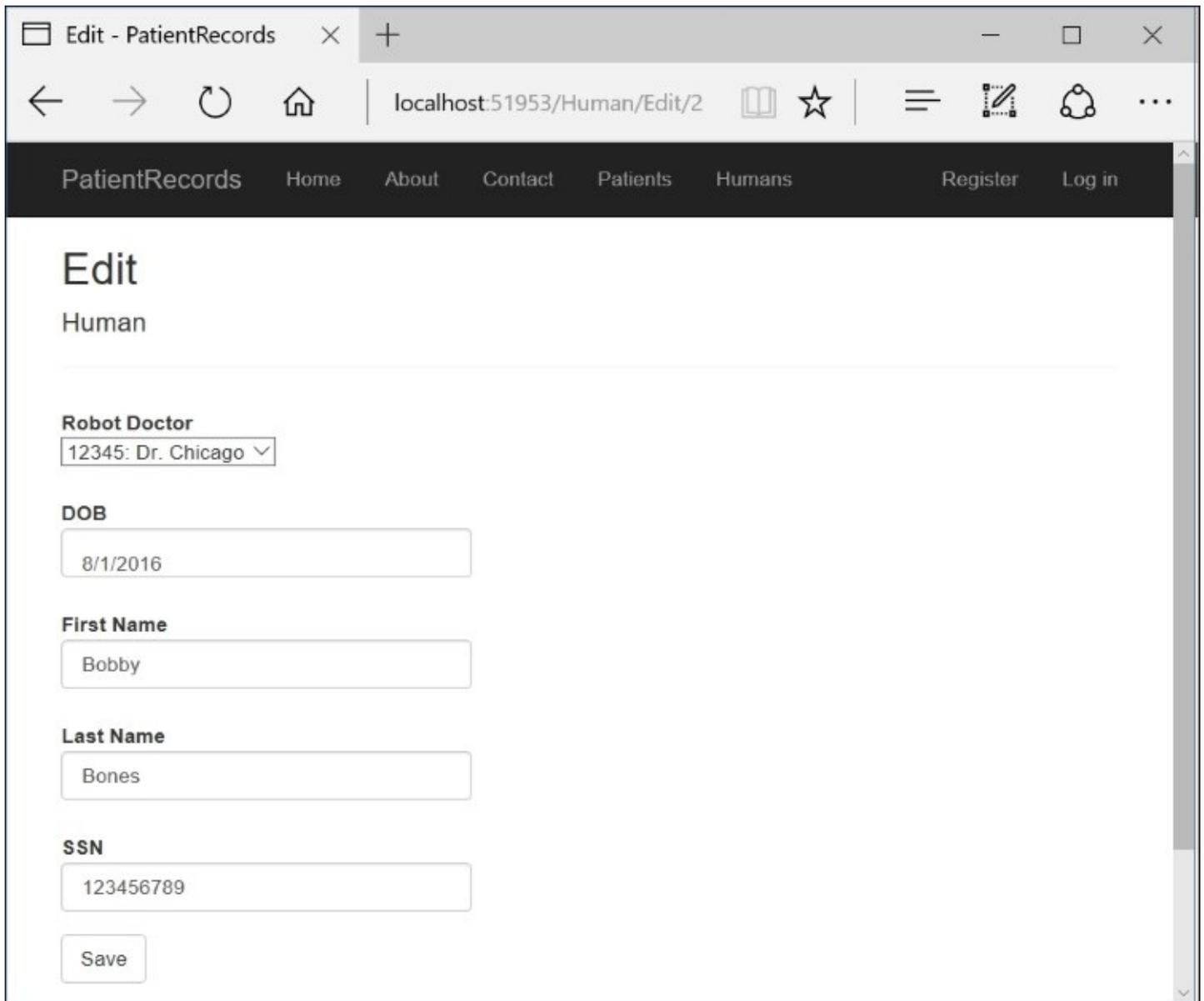
If you click the `Details` link, you should see the noneditable details of a previously added entry, as shown in the following screenshot. You have the choice of returning to the `Index` page once again or clicking `Edit` to see the editable view:
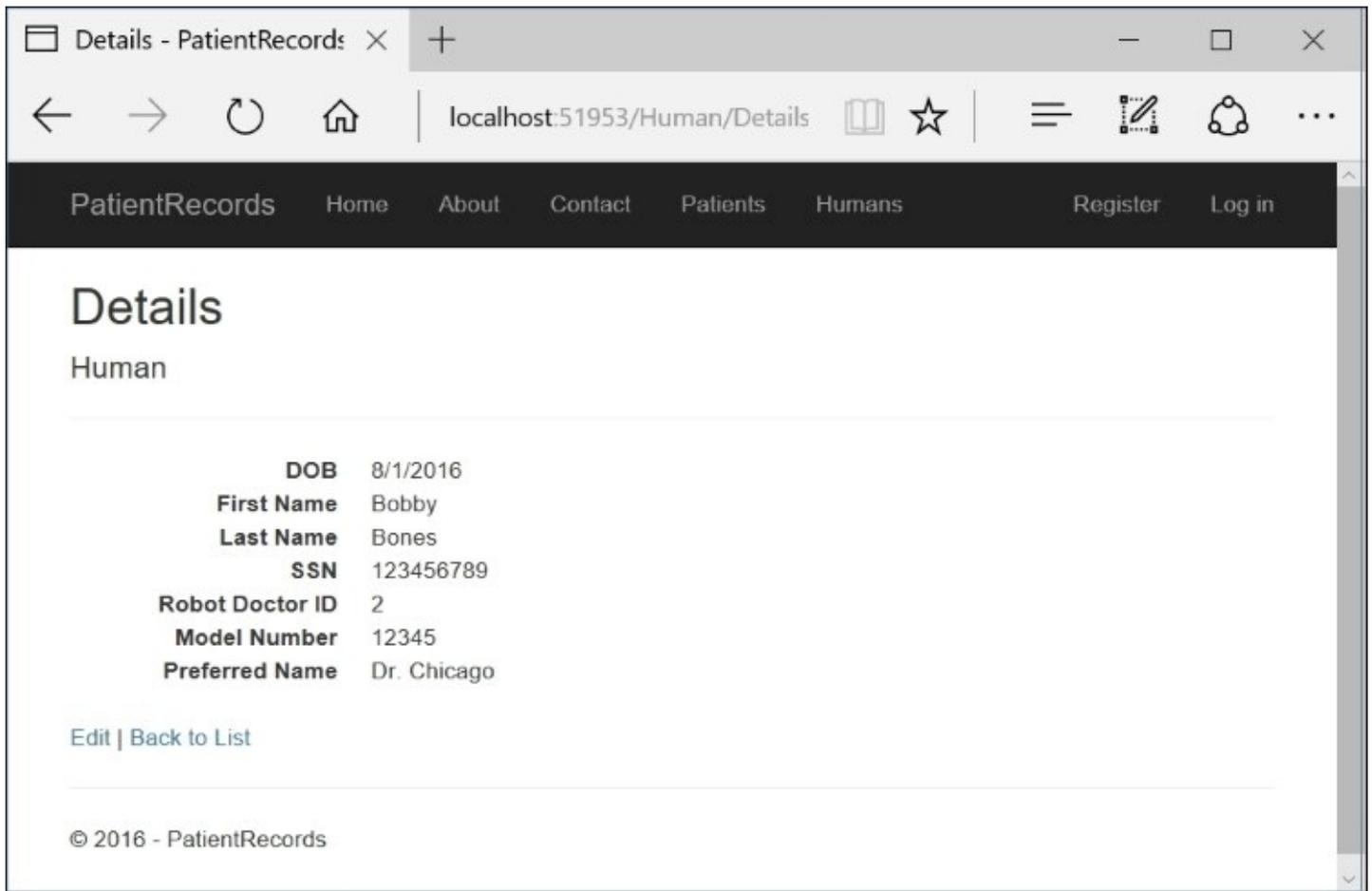
If you click the `Delete` link, you should see a confirmation page asking you if you want to delete the entry, as shown in the following screenshot. Click on the **Delete** button to proceed or return to the `Index` page:

PatientRecords   Home   About   Contact   Patients   Humans   Register   Log in

Delete

Are you sure you want to delete this?

Human

DOB	8/1/2016
First Name	Bobby
Last Name	Bones
SSN	123456789

[Delete] | Back to List

© 2016 - PatientRecords

Summary

In this chapter, we discussed EF Core, which is a modern ORM tool for web applications and more. For the purposes of this book, we only covered EF Core as it pertains to ASP.NET Core. For more information about EF Core, you may check out the official documentation at https://ef.readthedocs.org .

In the next chapter, you will cover **Dependency Injection**, **Inversion of Control**, and **Unit Testing**. This will give you a good application-design foundation to ensure that you have robust testable applications that can be easily updated with confidence.

Chapter 7. Dependency Injection and Unit Testing for Robust Web Apps

Inversion of Control (**IoC**) is one of those topics that tends to get dismissed by some developers as an advanced concept that they may never need. In the past, developers have had the choice of either rolling out their own code or using one of the many IoC containers to introduce **Dependency Injection** (**DI**) in their code. With ASP.NET Core, you will have the choice of using the built-in DI features or making use of existing IoC containers that you may already be familiar with.

In this chapter, we will start off with an introduction to IoC and how DI can help you build better applications. We will implement DI in a sample project and then learn the benefits of unit testing. Finally, we will wrap up with various DI options you have available to you as an ASP.NET Core web application developer.

Understanding IoC

IoC is a well-known pattern that invents the control of how dependent objects are created within software components. Using DI allows us to implement IoC in our software applications. Let's take a look at an actual scenario to illustrate what this means.

If you are already familiar with DI and IoC, feel free to skip this section and jump right into the code in the next section. If you're not familiar with this topic, or have not had a chance to use it in your real-life projects, this section aims to help you become more comfortable with it.

Pros and cons of DI

Both experienced developers and beginners have opinions on whether DI is right for them or not. As a result, there are arguably many pros and cons for and against the use of DI and IoC in your projects.

Here are some reasons why DI is a good choice:

- Helps with adhering to the **Dependency Inversion Principle** (**DIP**)
- Allows objects to be easily swapped with replacements
- Facilitates the use of the **Strategy Design Pattern** (**SDP**)
- Improves the testability of applications
- Enables loose coupling of software components

For more information on DIP and the **Strategy Pattern**, take a look at the following Wikipedia articles: https://en.wikipedia.org/wiki/Dependency_inversion_principle

https://en.wikipedia.org/wiki/Strategy_pattern

On the other hand, you may not agree due to the following reasons:

- DI introduces a learning curve for some developers
- DI may require a significant overhaul of existing projects
- Project timelines may not allow DI

Let's address the list of cons first. Every technology has a learning curve if you're not accustomed to it. Once you recognize the benefits, it will make the extra effort worth it. If your project requires too much work to retrofit DI into it, you may be happy to leave the architecture as is, and avoid DI for that particular project. Since ASP.NET Core is new, you have the luxury of starting from a clean slate, and it also makes it easier to start off with DI.

As for the list of pros, the benefits may not be immediately clear, so the terms will be explained in more detail in the rest of this chapter.

SOLID principles and Gang of Four patterns

You may not be familiar with **SOLID** principles and the so-called **Gang of Four** (**GoF**) design patterns. To realize the benefits of DI, it helps to have some knowledge of these patterns and principles.

The SOLID principles refer to the following principles that make up the word SOLID. This acronym should help you remember the list:

- **(S)ingle Responsibility Principle**: Each class should only be responsible for one primary function. This encourages better class naming and discourages developers from making a class more than it needs to.
- **(O)pen-Closed Principle**: Objects should be open for extension, while remaining closed for modification. This encourages the creation of subclasses for added functionality without breaking existing code.
- **(L)iskov Substitution Principle**: Objects should be replaceable with appropriate objects, for example, other objects that share the same parent class or common interface. This enables loose coupling of related objects.
- **(I)nterface Segregation Principle**: Instead of overusing one generic interface, it is better to have more interfaces, well-suited for specific purposes. This encourages you to keep each interface lightweight.
- **(D)ependency Inversion Principle**: Objects should be decoupled or loosely coupled. This forces classes to depend on the abstract definition of another object instead of a concrete implementation.

Note

For more information on SOLID principles, there are plenty of reference material, blog posts, and articles online. A good source to start with may be the following Wikipedia page, which includes links to more detail on each principle, at: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

GoF design patterns refer to the patterns described in the classic book *Design Patterns: Elements of Reusable Object-Oriented Software,* written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They are frequently referred to as the Gang of Four.

There are a total of 23 design patterns, so covering all of them is way beyond the scope of this chapter, but it is worth mentioning the Strategy Pattern, which is relevant to the benefits of DI. This pattern defines a design that dictates interchangeable objects, which can be followed while adhering to the aforementioned SOLID principles.

Loose coupling

To demonstrate loose coupling, we can start with a controller with an action method that is responsible for a complex action. Ideally, our controller methods should be as lean as possible. If we create complex objects in our controller methods with the new keyword, these methods will become responsible for the creation of those objects. In other words, they will become tightly coupled with those objects.

Here is a code snippet that shows a controller method using a complex object:

```
public class MyController: Controller
{
        public MyComplexObject ComplexObject { get;  set; }

        public MyController()
        {
                ComplexObject = new MyComplexObject();
        }

        public IActionResult MyActionMethod()
        {
                ComplexObject.GetStuffDone();
        }
}
```

Here, we can see that the object has to be initialized with the `new` keyword in the constructor. This creates a tightly-coupled dependency that requires the controller code to be updated if we need to swap out a different object. In order to avoid such a dependency, we could pass in an object that has already been initialized elsewhere. This will allow us to call the methods on that object without having to worry about creating instances of it and disposing of them afterward. Here's a code example:

```
public class MyController: Controller
{
        public MyComplexObject ComplexObject { get;  set; }

        public MyController(MyComplexObject complexObject)
        {
                ComplexObject = complexObject;
        }

        public IActionResult MyActionMethod()
        {
                ComplexObject.GetStuffDone();
        }
}
```

In the preceding code, the object is already being passed in through the constructor, so we can call its methods elsewhere in the controller class. This is a good start, but we can improve it further by changing the method parameter from a class to an abstracted interface instead:

```
public class MyController: Controller
{
        public IMyComplexObject ComplexObject { get;  set; }

        public MyController(IMyComplexObject complexObject)
        {
                ComplexObject = complexObject;
        }

        public IActionResult MyActionMethod()
        {
                ComplexObject.GetStuffDone();
        }
}
```

Now, the constructor can accept any concrete class that implements the expected interface. But how will this object get initialized before it's passed in? That's where IoC containers come in. To get going, we will look at an example of the minimalistic built-in DI container that is provided with ASP.NET Core.

Implementing DI in ASP.NET Core

Before we proceed, you may be surprised to know that we already have DI in the sample project that we have been working with in earlier chapters. Take a look at the `Startup.cs` file in your web `project` folder. You should see the following code near the end of the `ConfigureServices()` method:

```
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
```

This code adds a couple of instances of the `AuthMessageSender` service class, which implements both the `IEmailSender` and `ISmsSender` interfaces. The code for this class and its interfaces can be found in the `Services` subfolder of your project, as shown in the following screenshot:
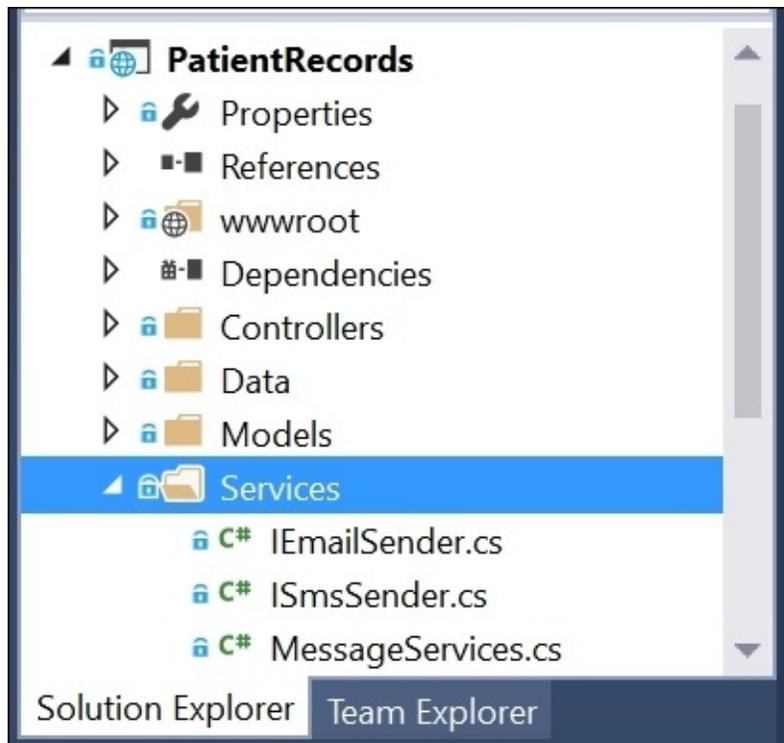
The call to `AddTransient` ensures that the object is properly created and destroyed, so that you don't have to worry about instantiation and disposal. There are multiple ways to indicate what kind of life cycle you want. Life cycle management is an integral part of using DI in your application.

Using our Patient Records project from [Chapter 6](#) , *Using Entity Framework to Interact with Your Database in Code,* let's add a new service called `ApptService` to help us make appointments for our fictitious hospital:

1. Right-click the `Services` folder in Solution Explorer.

2. Select **Add** | **Class** from the pop-up menu.
3. Name your class `ApptService.cs`:
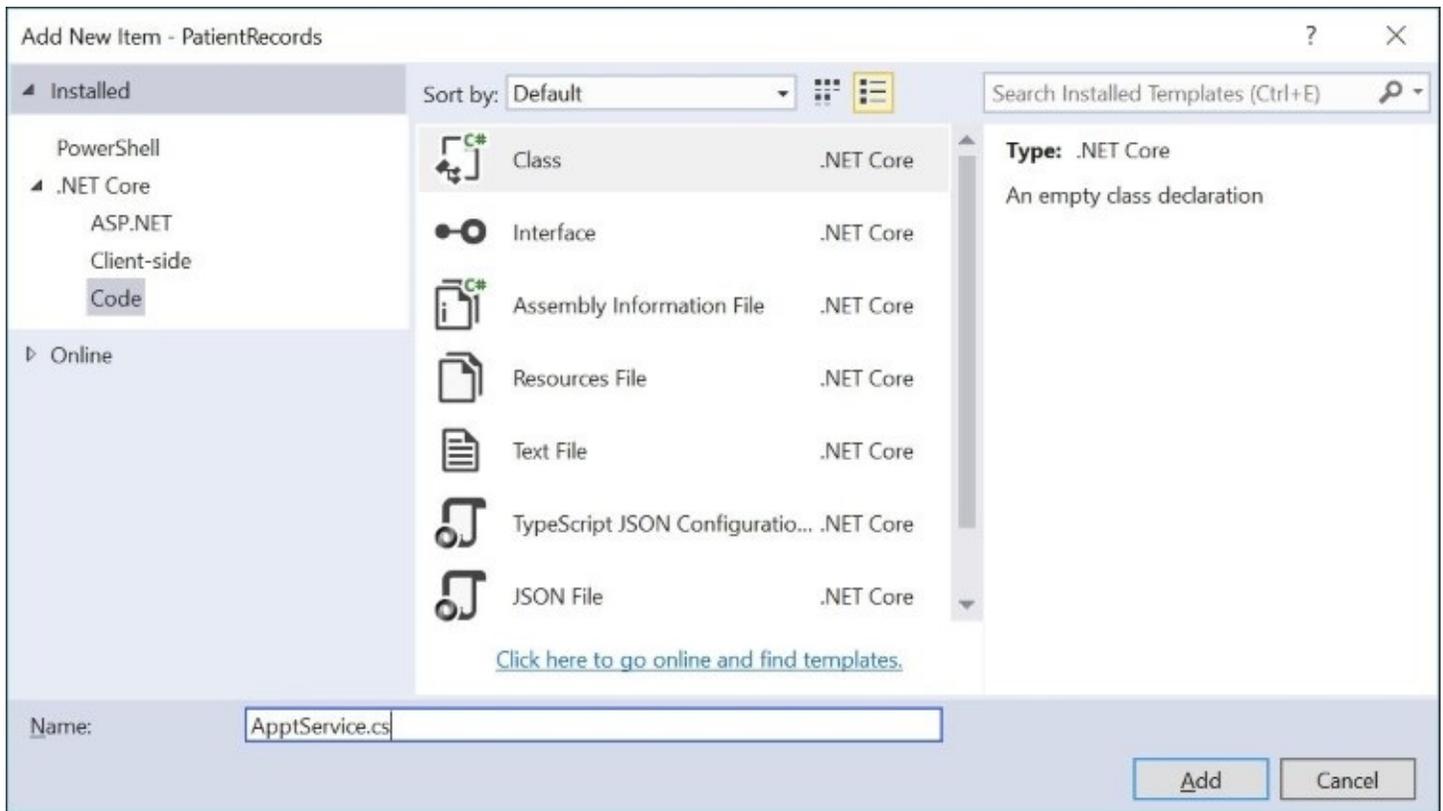
Next, add a method to simulate the scheduling of an appointment. For simplicity's sake, the following sample code returns true to indicate that the appointment was scheduled successfully:

```csharp
public class ApptService
{
    public bool ScheduleAppt()
    {
        bool isSuccess = false;
        // scheduling code goes here
        isSuccess = true;
        return isSuccess;
    }
}
```

Next, extract an interface from this service class. You could create another class file and name it `IApptService`, but it is easier to let Visual Studio 2015 do the extra work:

1. In `ApptService.cs`, right-click the name of the class.
2. Choose **Quick Actions...** in the pop-up menu that appears.
3. Choose **Extract Interface** in the next pop-up menu.
4. In the dialog box that appears, leave the defaults and click **OK**:
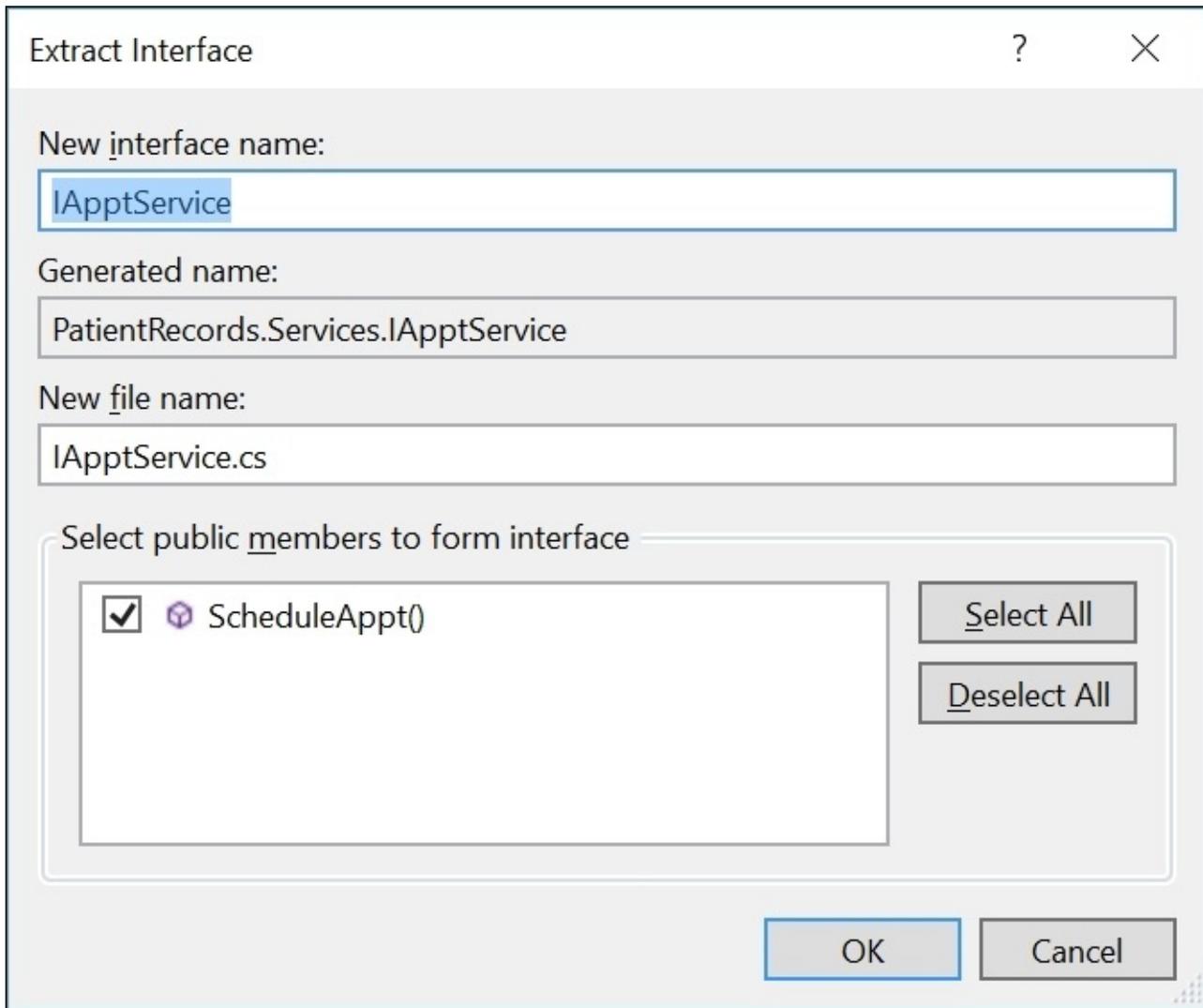
If Visual Studio doesn't automatically add the new interface to your `Services` folder, you may have to locate it within the project's root folder in Solution Explorer. If the `IApptService.cs` file is not in your `Services` folder, you should drag it into the `Services` folder manually:
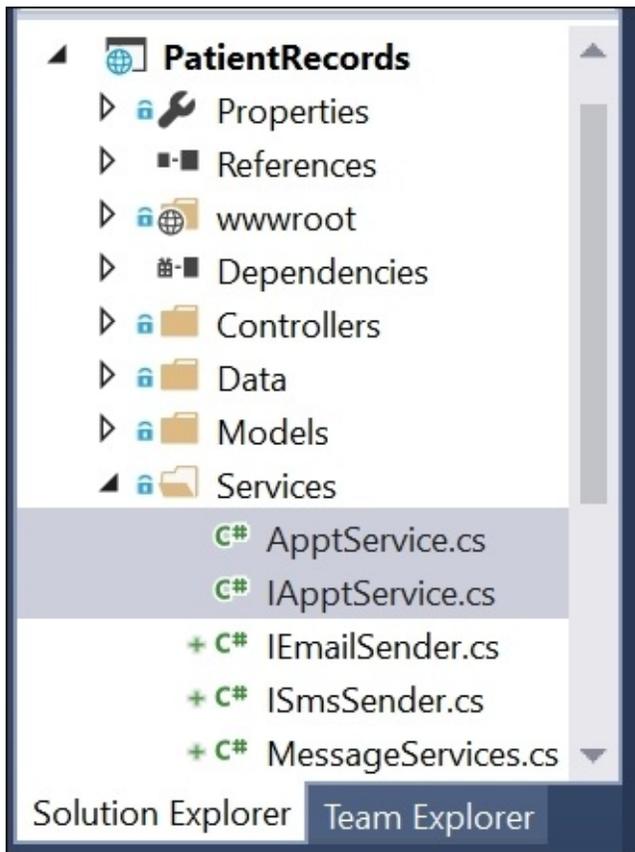
You should now have a new interface and service class in your `Services` folder. We are now ready to instantiate the service class and make use of it. We need to choose a lifetime for the object when we set up its DI. We will make use of it by injecting it inside a controller class.

Lifecycle management

In order to manage your dependency's lifecycle in ASP.NET, here is a list of lifetimes you can choose from:

- **Transient**: A new instance will be created each time the object is needed
- **Scoped**: A new instance will be created for each web request
- **Singleton**: A new instance will be created only once at application startup
- **Instance (special case of Singleton)**: Use `AddSingleton()` and create an instance yourself

In each case, the object will be disposed only after it is no longer needed. To use one or more lifetime settings, you would typically add the your code to the `ConfigureServices()` method of your `Startup.cs` file. This is explained in more detail later in this chapter.

Constructor injection versus action injection

When you inject a dependency through the constructor, this method is known as **constructor injection**. If you inject it in an action method, that is known as **action injection**. You can use either approach in your application.

First, let's add a new controller to manage the hospital's activities. Create a new `HospitalController` class in the `Controllers` subfolder:

1. Right-click the `Controllers` subfolder in Solution Explorer.
2. In the pop-up menu, click **Add** | **New Item**.
3. Add a new **Class** named `HospitalController.cs`.

The following screenshot shows the creation of `HospitalController.cs`:
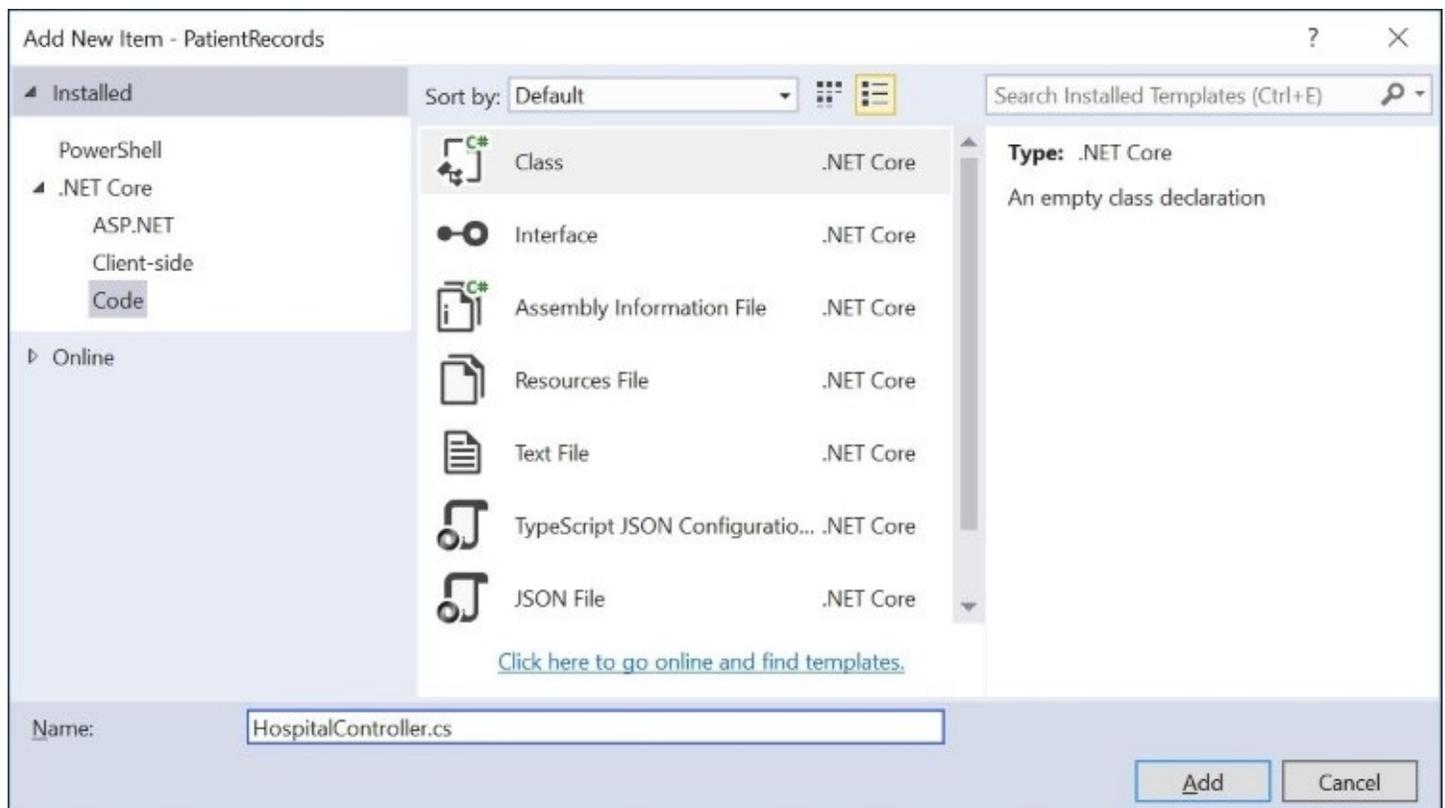
To use constructor injection, let's add a reference to our controller's constructor. For this option, add the following code to the body of the `HospitalController` class:

```
public IApptService ApptService { get; set; }
public HospitalController(IApptService apptService)
{
    ApptService = apptService;
}
```

In order to use the aforementioned service class, make sure you add the proper using statement to the top of the class:

```
using PatientRecords.Services;
```

To use action injection, the injection is performed within the parameter section of your controller's action method:

```
public IActionResult ActionMethod([FromServices] IApptService apptService)
{
    ViewData["Message"] = "Scheduled: " + apptService.ScheduleAppt();

    return View();
}
```

If you happen to read older blog posts or outdated documentation, it may be useful to know that earlier pre-release versions of ASP.NET Core allowed **setter injection** to inject a dependency at the property level.

Once again, make sure you include proper using statements above the class definition:

```
using Microsoft.AspNetCore.Mvc;
using PatientRecords.Services;
```

The first using statement ensures that we will be able to use the `[FromServices]` attribute (among other things). The second using statement ensures that we can use the `IApptService` interface class.

Finally, let's add a controller action method to call our service class. Add the following code below the constructor of our `HospitalController` class:

```
public string ProcessAppointment()
{
    bool isSuccess = ApptService.ScheduleAppt();
    if (isSuccess)
        return "Success!";
    else
        return "Failed...";
}
```

For simplicity, the preceding code returns a string value instead of an actual view. But if we were to run the application now and invoke the new controller method, we would get an error message. The error message would be something like the following:

```
An unhandled exception occurred while processing the request.
InvalidOperationException: Unable to resolve service for type
'PatientRecords.Services.IApptService' while attempting to activate
'PatientRecords.Controllers.HospitalController'.
```

This error occurs because we still haven't told the DI engine how to create our service object. To fix this problem, we need to update our `Startup.cs` file and use the following line of code

at the bottom of the `ConfigureServices()` method, as seen in the sample code for this project:

```
services.AddTransient<IApptService, ApptService>();
```

You can add the preceding line below the existing calls to `services.AddTransient` in the sample application. Make sure you add an interface file for `IApptService` that will be implemented by the `ApptService` class. Your application is now ready for some quick testing.

Verifying the expected behavior

Run the application then manually invoke the controller method by typing in the controller name and method name in your web browser.

```
http://localhost:12345/Hospital/ProcessAppointment
```

Your port number may vary, but the rest of the URL should be similar to the preceding URL. You should now see a **Success!** message, as shown in the following screenshot. You may recall that our service method has been hard-coded to always return a successful result. If you wish to test out an unsuccessful result, you can edit the method in your service class to return false instead of true:
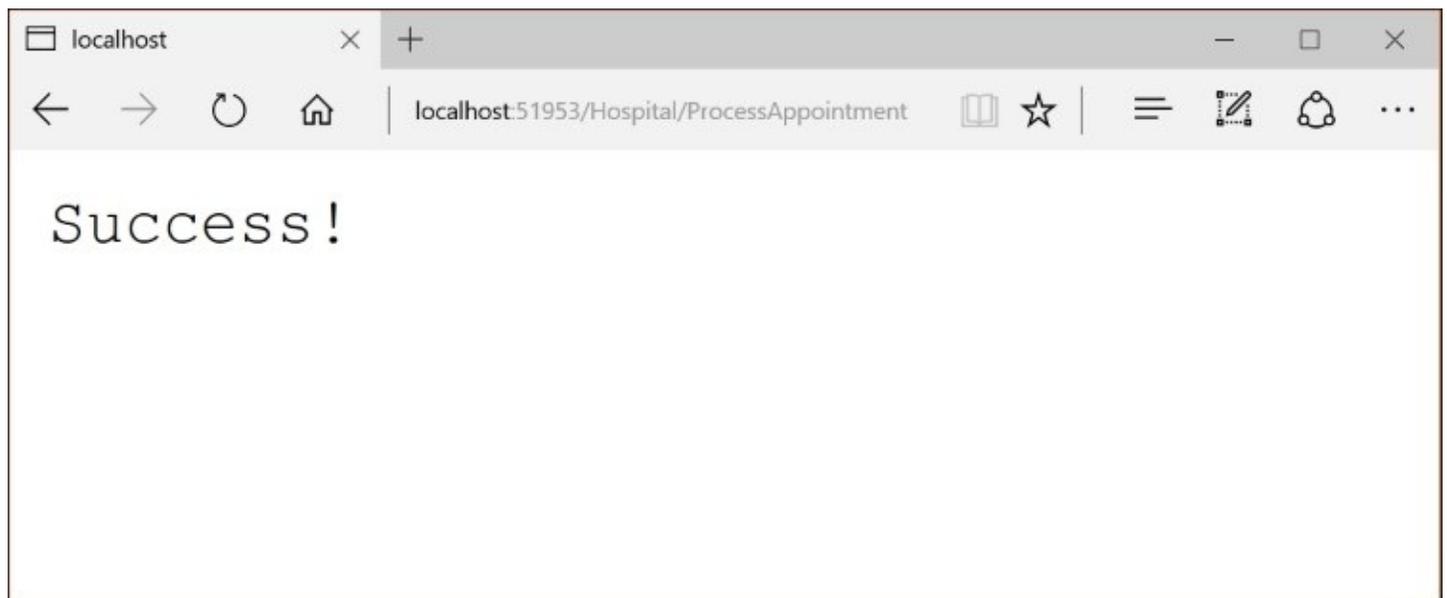
Congratulations! You have now successfully set up DI in ASP.NET Core with the built-in IoC features.

In order to test the remaining life cycle options for the injected service object, we can replace the call to `AddTransient()` in our `ConfigureServices()` method in the `Startup.cs` class. Try using the following options to swap out the single line of code with another. Do *not* use more than one of the following options:

```
// Option 1: Transient
services.AddTransient<IApptService, ApptService>();
// Option 2: Scoped
//services.AddScoped<IApptService, ApptService>();
// Option 3: Singleton
//services.AddSingleton<IApptService, ApptService>();
// Option 4: Instance (Singleton, with explicit instance)
//services.AddSingleton<IApptService>(new ApptService());
```

Note that each of the options displays very similar syntax, and they all make use of both `IApptService` and `ApptService`. However, the last call to `AddSingleton()` is different from the rest. With this option, we are responsible for creating the instance ourselves, so we need to use the `new` keyword in this case. Based on the requirements of your application, you can use one of the preceding options to set the life cycle of your service object.

DI options in ASP.NET Core

When it comes to DI, you have a few choices to select from. When it comes to DI in ASP.NET Core, you already know that you have the benefit of selecting from the built-in IoC container.

But what other options do you have and how should you decide when to select one over another?

Built-in IoC

If you're just getting started with ASP.NET Core, I would highly recommend going with the simplest choice for DI. Go with what's included out-of-the-box. If you're especially new to DI and IoC containers, you should definitely stick with the default IoC container.

If you're already very familiar with other alternatives, you could start with what you know. For example, if you have used **Autofac** extensively as your IoC container in past projects, you may want to stick with Autofac for new ASP.NET Core projects.

Autofac

Autofac is already used by many ASP.NET developers, and was available for ASP.NET Core long before its official release. In this section, we will take a look at how we can use Autofac in our project.

Create a new standard web project and add the following references to Autofac in your application's `project.json` file, in the dependencies section:

```
"Autofac": "4.0.0-rc3-316",
"Autofac.Extensions.DependencyInjection": "4.0.0-rc3-309"
```

The two references to the main Autofac package and its DI-specific package will get you started. As with other references in your configuration file, your version numbers may vary.

Next, update your `ConfigureServices()` method in your `Startup.cs` file to initialize Autofac as needed. You may have noticed that this method returns void in a newly created web project. In order to use Autofac, you should change the return type to `IServiceProvider`.

Update your `ConfigureServices()` method to return an `IServiceProvider` and add the following block of code:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // other code in method

    // Prepare Autofac
    var containerBuilder = new ContainerBuilder();
    containerBuilder.RegisterModule<DefaultModule>();
    containerBuilder.Populate(services);
    var container = containerBuilder.Build();
    return container.Resolve<IServiceProvider>();
}
```

In order for the preceding code to work, make sure you add the proper using statements to the top of your `Startup.cs` class:

```
using Autofac;
using Autofac.Extensions.DependencyInjection;
```

The first using statement allows you to use Autofac in your code, as you would expect. The second using statement brings in additional extension methods such as the `Populate` method.

So what is all this code doing? Autofac is being set up in five lines of code:

1. First, a new `ContainerBuilder` object is created. This is Autofac's DI container, which has methods in it to register service modules.
2. Next, we are registering a module that will contain the instructions for service objects that need to be created. You may notice that an undefined module named

`DefaultModule` is being registered, but we have yet to create this module. We'll take care of it shortly.

3. With a call to the `Populate()` method of `ContainerBuilder`, it passes in the services object, which has already been set up earlier in the method.
4. Before we return, a call to the `Build()` method of `ContainerBuilder` will create a new container using the components that have just been registered.
5. Finally, we can return the service provider to kick things off.

Now, let's take care of creating the module that the preceding code is attempting to register:

1. Right-click the web project in Solution Explorer.
2. Click **Add** | **New Folder** in the pop-up menu.
3. Name this folder `DependencyInjection` (for convenience).
4. Right-click the new folder you just created.
5. Click **Add** | **Class** to create a new class.
6. Name your class `DefaultModule.cs`.

The following screenshot shows the creation of `DefaultModule.cs`:
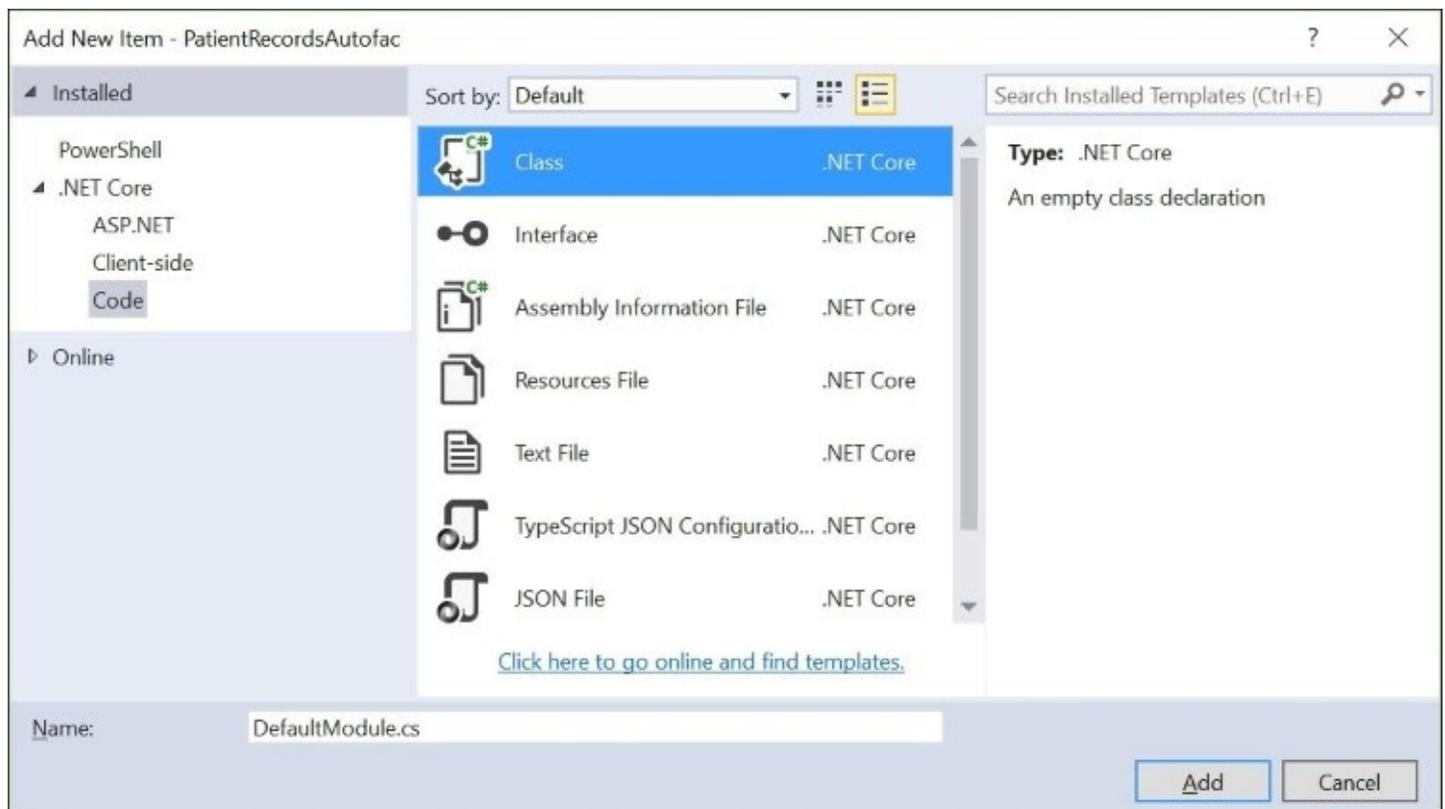
Replace the contents of this new class with the following code:

```
public class DefaultModule: Module
{
```

```
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<SomeClass>().As<ISomeInterface>();
    }
}
```

Note that we are deriving `DefaultModule` from `Module` and overriding its `Load()` method. In order for this code to work, we must add the following using statements to the top of the class:

```
using Autofac;
using Autofac.Extensions.DependencyInjection;
```

In the call to `RegisterType()`, there is a placeholder class called `SomeClass` and a placeholder interface called `ISomeInterface`. You should replace these with a class and an interface that you would like Autofac to resolve for use in your code. In fact, you can use `IApptService` and `ApptService` from the example used earlier in this chapter. Just make sure you include a using statement that includes the namespace for your service classes.

Other alternatives

If you would like to explore other alternatives, take a look at the following IoC containers available for .NET. Make sure you read each provider's documentation to ensure that their packages are appropriate for the latest version of ASP.NET Core and MVC:

- **StructureMap**: http://structuremap.github.io
- **Ninject**: http://www.ninject.org
- **Castle Windsor**: http://www.castleproject.org/projects/windsor
- **Unity**: https://github.com/unitycontainer/unity

You may also use IntelliSense in your `project.json` file to determine the latest Stable/Beta version available through NuGet.

Writing unit tests

In addition to DI, it is also important to write unit tests to build robust software. By automating your tests, you can minimize the chances of bugs in your code and generate confidence when adding new features.

Many articles and a few books have been written on unit testing, so this section is by no means a comprehensive reference on it. Instead, its intent is to introduce you to the possibilities of automated testing in your code.

Setting up a test project

ASP.NET developers have been using various unit testing frameworks over the years. The list of testing frameworks includes **nUnit**, **xUnit.net**, and Microsoft's own **MSTest**. All of these products have the same basic capabilities and some advanced capabilities as well. To extend the functionality, developers can use mocking frameworks such as **Moq** (pronounced Mock-You or Mock).

Since the introduction of ASP.NET Core through various Beta versions, xUnit.net has been made available and is ready for use. Although MSTest was not initially available during early betas, the ASP.NET team announced MSTest in the RC2 version of ASP.NET Core. In this chapter, we will cover xUnit.net for our unit testing samples.

Tip

If you are interested in comparing the syntax of xUnit.net with its alternatives, take a look at the comparison chart available on GitHub at https://xunit.github.io/docs/comparisons.html.

To set up xUnit.net for your project, we should create a new project in our solution, just for our test code. Although this is not technically required, it is good practice to separate the test code from the application code. The test project will have a reference to the application project, but the application project should not have any knowledge of the test project.

You could create a new folder for your test project:

1. Right-click your solution in Solution Explorer.
2. Click **Add** | **New Solution Folder** in the pop-up menu.
3. Name your new folder test, as shown in the following screenshot:
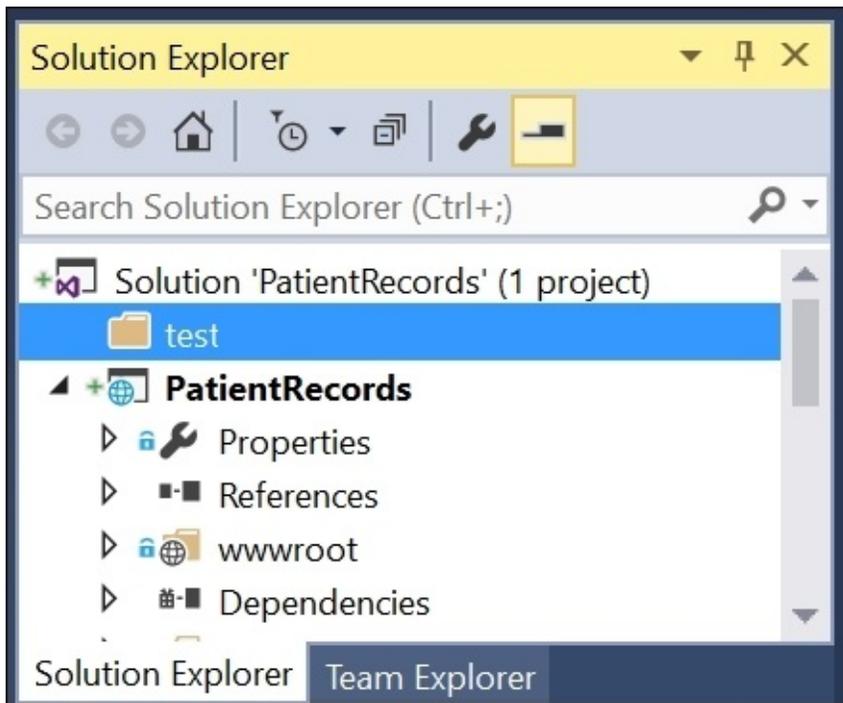
To create a test project, follow these steps:

1. Right-click your `test` folder.
2. Click **Add** | **New Project** in the pop-up menu.
3. Choose **Class Library (.NET Core)** for the project type.
4. Enter `PatientRecords.Tests` as the project name, as shown in the following screenshot:
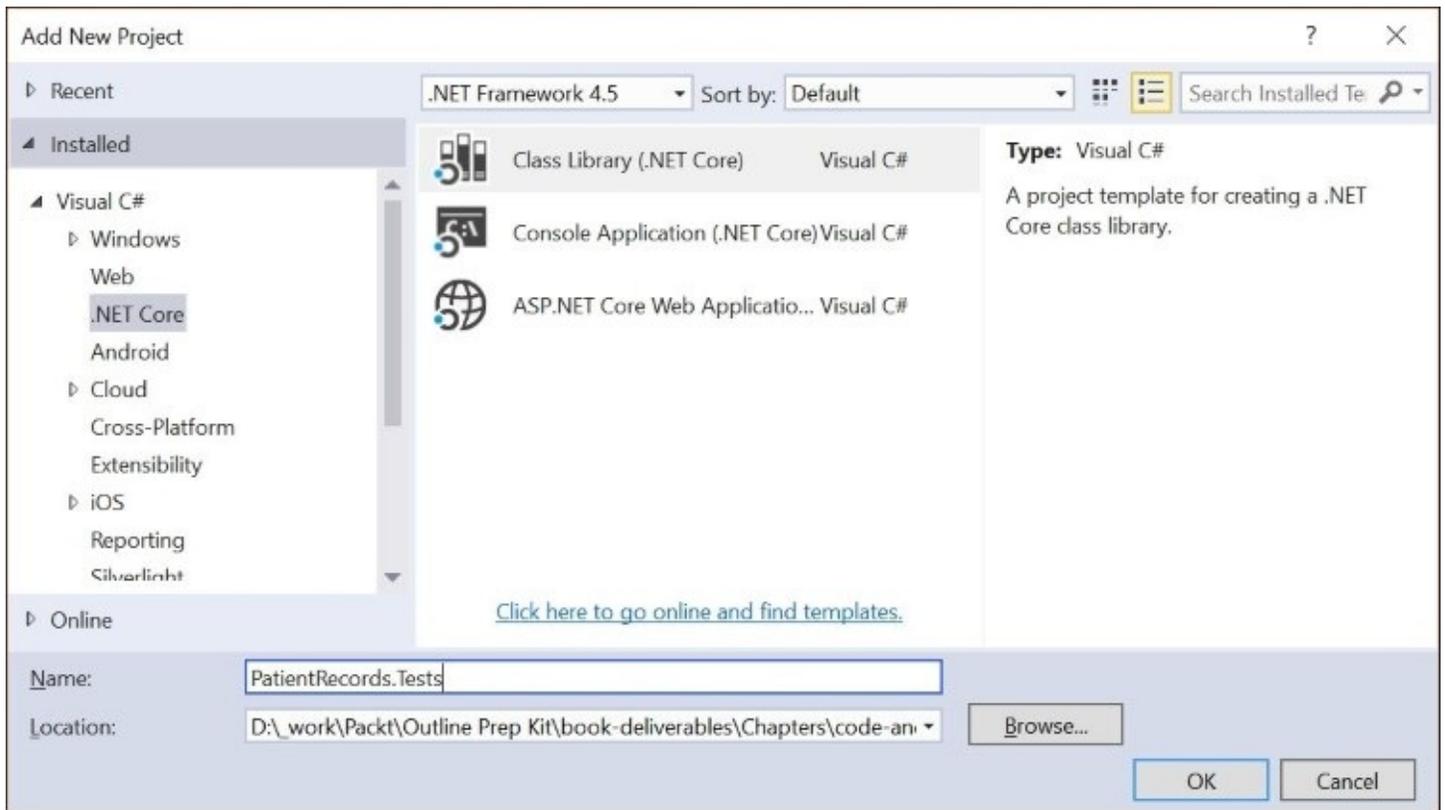
Note that the name of the test project does not have to include the name of your application project. However, it is a common convention to start with the app project name when naming the test project, then end with `.Test` or `.Tests`.

To include the proper references in our test project, edit your `project.json` file from within your test project. Make sure that you have opened this configuration file from the test project and not the app project, to make the following changes.

In the "dependencies" section of the `project.json` file, add references to `xunit`, `xunit.runner.dnx`, and your web project:

```
"dependencies": {
  "xunit": "2.2.0-beta2-build3300",
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "PatientRecords": "1.0.0-*"
},
```

The first `xunit` reference allows you to use xunit in your test project, while the runner reference allows you to run the unit tests from a command line or from within Visual Studio. Finally, the reference to `PatientRecords` is a reference to your web application project, and may vary depending on how you named your project. Once again, your version numbers may vary so you should use IntelliSense popups while typing to decide which Beta/Stable version may be suitable for your project.

Just below the project version number, identify `xunit` as your test runner:

```
"testRunner": "xunit",
```

The new project should already have a placeholder class in it. To create your first test class, you could rename the placeholder class name and its filename. You could also delete the placeholder class and create a new class instead.

To create a new test class, follow these steps:

1. Right-click your test project in Solution Explorer.
2. Click **Add** | **Class** in the pop-up menu.
3. Name the test class `HospitalServiceTests.cs`, as shown in the following screenshot:
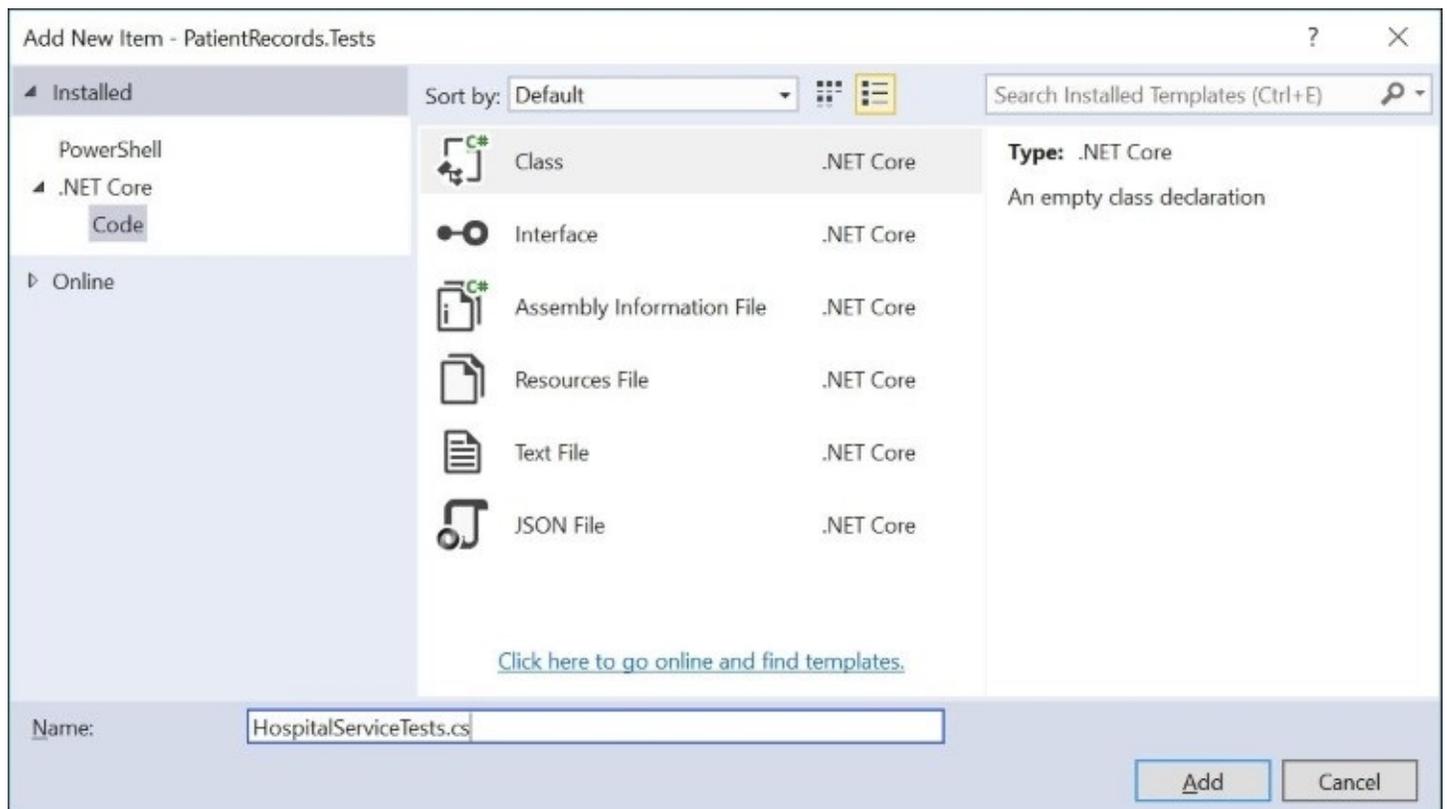
In order to write your first unit test, you need to add a using statement to make use of xUnit.net. Add the following using statement to the top of your test class:

```
using Xunit;
```

Next, add the following placeholder method in your test class:

```
[Fact]
public void VerifySuccess()
{
    bool isSuccess = false;
```

```
    isSuccess = true;
    Assert.True(isSuccess);
}
```

In the preceding code, the `[Fact]` attribute identifies the method as a test method without any parameters. The `public` keyword ensure that the test can be called from outside the assembly, by test runners and continuous integration systems. The call to `Assert.True()` will verify whether the `isSuccess` variable is set to true or not. For simplicity, we are hard-coding its value. You are now ready to run this sample test.

Running unit tests

We can run our unit tests from with in Visual Studio 2015 or from a command line. To run your tests from within a Visual Studio, make sure the **Test Explorer** panel is open and visible. To run your tests from a command line, you must open a command prompt window and run the commands at the project's folder.

To use the Test Explorer from Visual Studio 2015, follow these steps:

1. In the top menu, click **Test** | **Windows** | **Test Explorer**.
2. Verify that the Test Explorer panel is visible in Visual Studio.
3. In the top menu, click **Build** | **Build Solution**.
4. Verify that your unit test appears in the Test Explorer panel.
5. Click on **Run All** in the Test Explorer panel.
6. Verify that your test has passed successfully, as shown in the following screenshot:
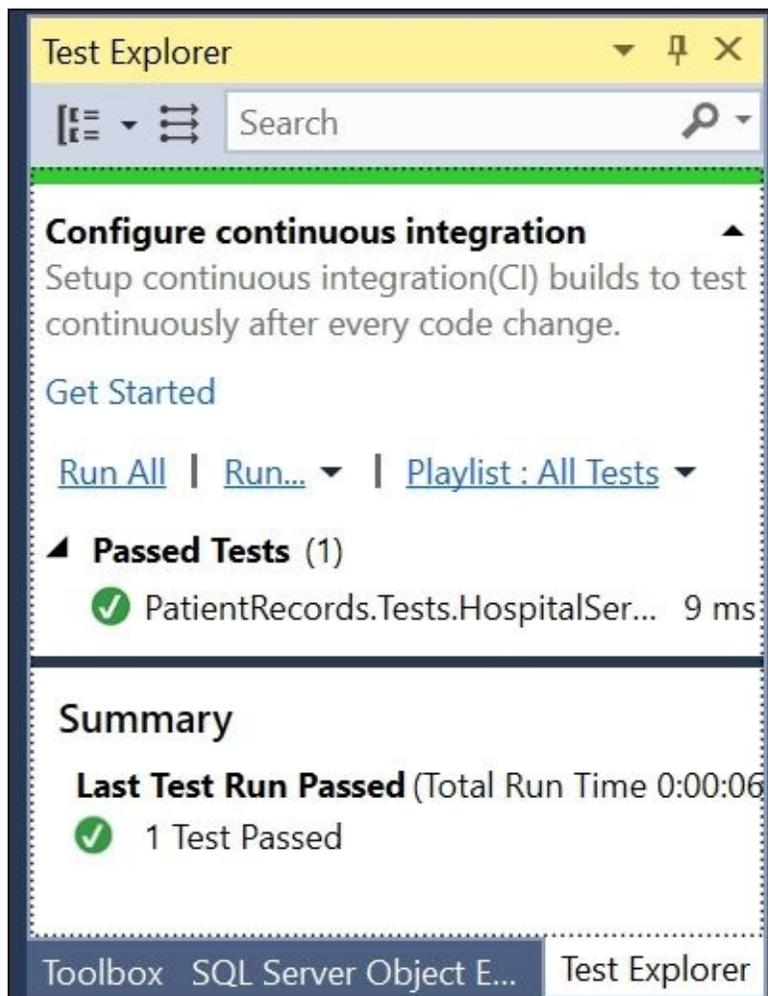
To use a command prompt to run the test, follow these steps:

1. Open a command prompt window.
2. Change the current directory to point to the test project folder.
3. Enter the following command to run your sample test:

```
>dotnet test
```

This command will trigger the test system to discover your unit tests, and run all available tests. Once the tests have been run, you should see a test execution summary that identifies the number of tests, how many succeeded, and how many failed, as shown in the following screenshot:

If you want to include parameters in your unit test methods, you can use the `[Theory]` and `[InlineData]` attributes together to replace the `[Fact]` attribute. The parameters passed in the attributes should match the parameters in the test method signature.

To include parameters, add the following method to your test class:

```
[Theory]
[InlineData(7, 5)]
public void VerifySuccessWithParams(int n1, int n2)
{
    Assert.True(n1 > n2);
}
```

Here, the `[Theory]` attribute defines the test method to accept some test data as parameters. The `[InlineData]` attribute accepts one or more parameters that match the test method's parameters. The **Assert** statement can take many forms, so you can discover its various features by typing in a period after the word `Assert`, and observing the list of possible methods you can call on it. In this case, we are verifying that the number *n1* is greater than *n2*.

Finally, let's take a look at how you can call your application's methods to test out your application code. Add the following using statement to your test class:

```
using PatientRecords.Services;
```

Next, add the following method to your test class:

```
[Fact]
public void VerifyApptService()
{
    ApptService apptService = new ApptService();
    var isSuccess = apptService.ScheduleAppt();
    Assert.True(isSuccess);
}
```

This test method is responsible for creating a new instance of the `ApptService` class, after which it calls the `ScheduleAppt()` method. Depending on what is returned by the application code, the test with either pass or fail.

Whether your test method is a `[Fact]` or a `[Theory]`, you may want to temporarily skip the test for some reason. It may be because the test is failing and you need some extra time to troubleshoot it. In order to skip a test method, you may use the `Skip` parameter and provide a reason:

```
[Fact(Skip = "skip this for now")]
[Theory(Skip = "skip this too")]
```

The next time you run all your tests, any test methods marked with this parameter will be skipped in that test run. Keep in mind that this practice should not be used to hide your test methods for long periods of time. If some tests are being skipped for any period of time, you should figure out a way to fix the tests if necessary. If the tests have become obsolete or irrelevant, you should consider deleting them instead of skipping them every time.

Going beyond the basics

Before wrapping up this section, it is important to mention that there is more to automated testing than just the basic unit tests covered in this chapter. Earlier in this section, there was a brief mention of alternative frameworks, such as nUnit and MSTest. If you have the time to learn more, take a look at what else is available before you decide which testing framework is right for your project.

If you also choose to use the Moq mocking framework, you can use it along with existing testing frameworks such as xUnit.net. In the earlier testing example, you could have replaced the concrete class with an abstract interface, which will allow you to replace the interface with mock objects. Mocking frameworks is beyond the scope of this book, but you can learn more about Moq on the product's GitHub page at https://github.com/Moq .

At a minimum, you should be aware of the varieties of mocked objects you can create in your test project: **Mocks**, **Fakes**, and **Stubs**:

- Mocks are pre-programmed with the expected results, don't actually connect to a real database, but can be used for behavior verification
- Fakes are working examples, but not production-ready (for example, an in-memory database)
- Stubs are placeholders that provide canned answers, which are useful for specific limited scenarios

If these terms are entirely clear to you, you should refer to Martin Fowler's classic write-up of Mocks, Fakes, and Stubs at http://martinfowler.com/articles/mocksArentStubs.html .

Summary

In this chapter, we discussed IoC, DI, and unit testing. We covered multiple ways of implementing DI in an ASP.NET Core project and then learned the basics of unit testing in ASP.NET Core.

If you want to learn more about DI and IoC containers, a great place to start is a classic article on the topic by software engineer and public speaker Martin Fowler at http://www.martinfowler.com/articles/injection.html .

In the next chapter, we will cover authentication, authorization, and security in ASP.NET Core web applications.

Chapter 8. Authentication, Authorization, and Security

Authentication, authorization, and security are all important topics to be aware of when it comes to building secure web applications. There is a lot to cover, so we will focus on what's important and relevant to ASP.NET Core web apps.

But first, let's define these three terms:

- **Authentication**: Instead of allowing any website visitor to access your web application, you can use authentication to restrict who can use your application. This can be useful for any application that needs to identify each user before allowing any interaction with it.
- **Authorization**: Once inside your application, you can use authorization to restrict specific parts of the application. This can be useful for allowing some users to perform some tasks not accessible to other users, for example, administrative tasks, editing of data, and so on.
- **Security**: In this context, the term security refers to security vulnerabilities that may affect web applications. This can include, but is not limited to **Anti-Request Forgery**, **Cross-Site Scripting**, and **SQL Injection**.

Enabling authentication in ASP.NET

You may be pleased to hear that ASP.NET Core includes built-in authentication methods in its web template. In fact, creating the sample projects mentioned in previous chapters will result in a functioning web application with authentication included. If you start a brand new project, this section will also help you identify what you need to do to include authentication.

High-level overview

Here is a high-level overview of how you can enable authentication in a new ASP.NET Core application using the web template:

1. Create a new project with a web template.
2. Select the authentication method.
3. Verify the packages and references.
4. Verify the code in the `Startup.cs` class.
5. Add additional identity options as needed.
6. Add external providers as needed.

If you want to start with an empty project, you can perform the following steps:

1. Create a new empty web project.
2. Do not select an authentication method.
3. Add packages and references for authentication.
4. Add code to the `Startup.cs` class.
5. Add additional identity options as needed.
6. Add external providers as needed.

For obvious reasons, it is much easier to start with a web template. Note the differences in the two preceding lists. If you decide to start with an empty web template, you will have to do more work to get things up and running with authentication.

When you choose a project type, Visual Studio displays a dialog box that includes a button labeled **Change Authentication** in the right panel, as shown in the following screenshot:
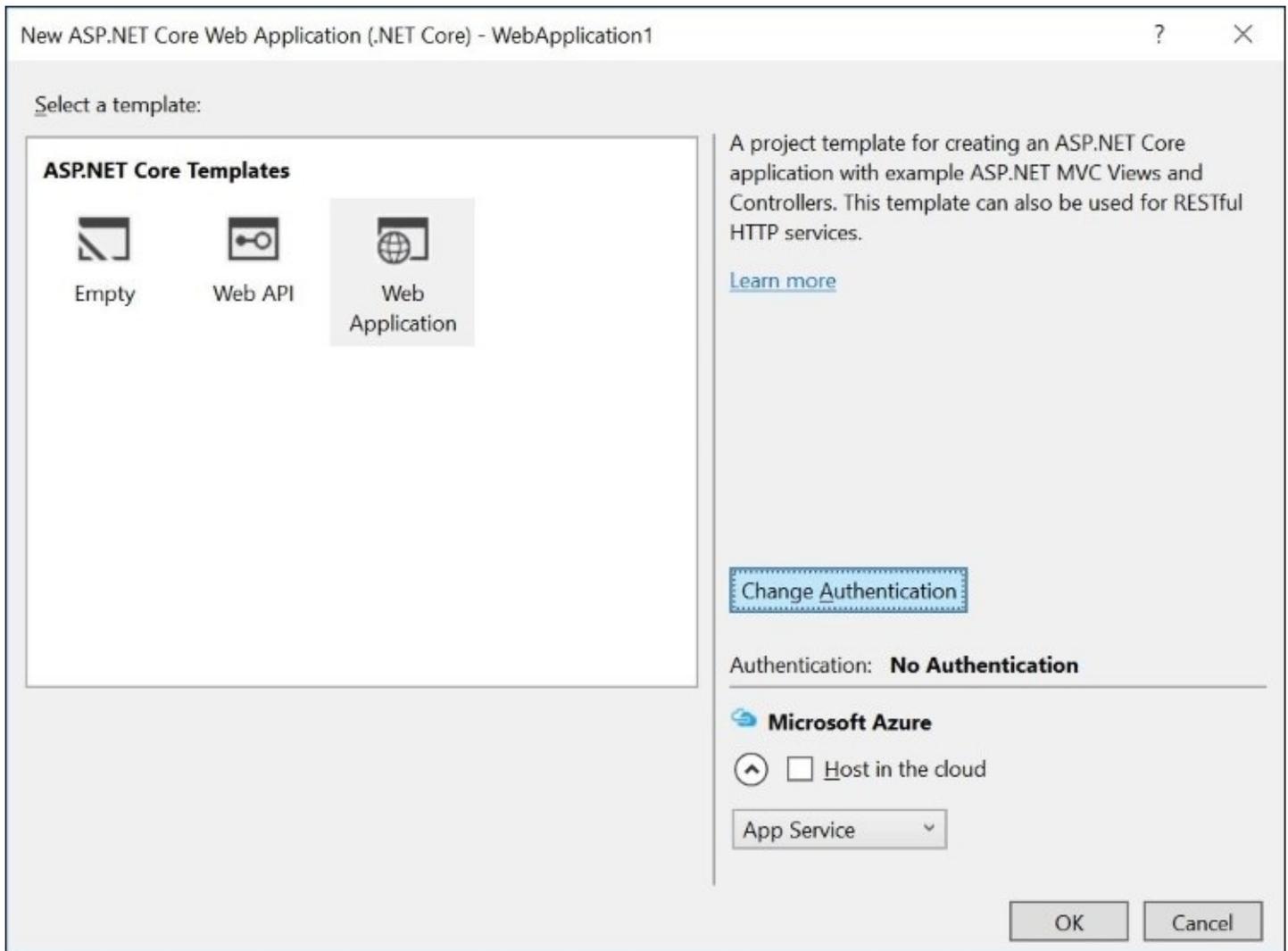
If you click the **Change Authentication** button, Visual Studio will display another dialog box that displays the following options:

- **No Authentication**: This is self-explanatory. Your project won't include any authentication automatically, and so should work for web applications that don't require any authentication. If you choose to add it later, you will have to take additional steps to add it manually.
- **Individual User Accounts**: This is quite useful for a starter web project. All user data is stored in the application database and can be accessed using the ASP.NET Core framework. Note that **ASP.NET Identity** has shifted away from **Windows Identity Foundation** in ASP.NET Core.
- **Work and School Accounts**: This option includes organizational accounts such as corporate or education institutional accounts that use **Windows Server Active Directory** or **Azure Active Directory**.
- **Windows Authentication**: This option is well suited to applications that will be used in an intranet environment, where the credentials on a Windows computer will be used to authenticate the user on a web application.

In this chapter, we will cover the authentication process using **Individual User Accounts**.

Authentication configuration

By using the Hospital Records project from prior chapters, you should already have a functional web project with the proper authentication method selected by default. Pay attention to the following verification steps to identify what you would need if you use an empty template for a future web project.

In your `project.json` file, verify that you have a reference to the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` package, as shown in the following screenshot:
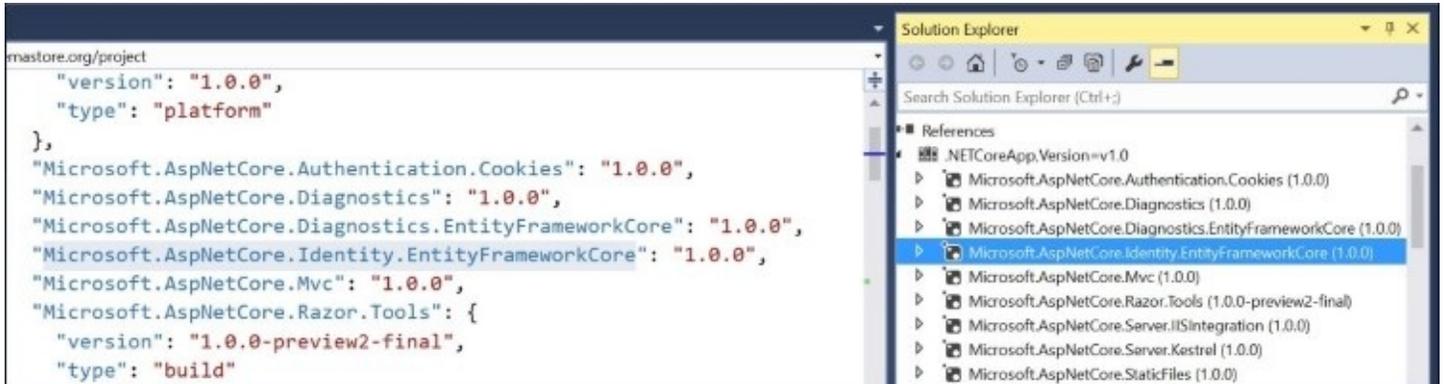
This EF-specific **Identity** package has a couple of other dependencies that you can see in the list of references in the Solution Explorer panel. The dependencies include these additional packages:

- Microsoft.AspNetCore.Identity
- Microsoft.EntityFrameworkCore.Relational

The `Relational` package pulls in the `Core` package for `EntityFramework` as a dependency, while the ASP.NET Identity package includes what we need to use authentication in our project. Entity Framework will be used to interact with the database, where our user credentials will be stored in a table named `AspNetUsers`.

To complete the configuration, the `Startup.cs` class needs the proper code to add and use authentication. As you may have guessed, this code will go into the `ConfigureServices()` and `Configure()` methods respectively.

In the `ConfigureServices()` method of your `Startup` class, verify that the following code is there to enable the use of the Identity features of ASP.NET Core:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

In the `Configure()` method of your `Startup` class, verify that the following code is there to enable the use of Identity features in ASP.NET Core:

```
app.UseIdentity();
```

There are several configuration flags that allow us to customize additional options for the use for Identity features. These include, but are not limited to, password restrictions. The following code shows how this works. For example, you should set `RequireDigit` to `true` if you want to require that digits should be used in a user's password during registration:

```
services.AddIdentity<ApplicationUser, IdentityRole>(
    options =>
    {
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireLowercase = false;
        options.Password.RequireNonAlphanumeric= true;
        options.Password.RequireUppercase = false;
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

If you want to explore additional options, you can enter a period after the word options in the list of options. This will allow you to use IntelliSense to discover what else is available for you. The password-related options provide these restrictions on top of any attributes you may have already assigned in the password field of an MVC model.

Run your application and click the **Register** link in the top panel. Once registered, you will be automatically redirected to the application as the registered user. You can log out and then log back in with the credentials you created.

External service providers

Many users may already have accounts provided by external services that they want to reuse. Social media giants such as Twitter and Facebook provide authentication services that you can easily integrate with your ASP.NET Core application.

The hard part is actually setting up an app on the provider's respective website, which gets easier once you have it set up.

Here is an overview of the steps required to set up an external service provider:

1. Visit the external provider's website to set up an app on their system.
2. Obtain the necessary ID, key, secret, or token values.
3. Install the **Secret Manager** tool in your development environment.
4. Assign secret values using the values you obtained earlier.
5. Update your code to use the user secret store during development.
6. Update your code to use a third-party provider, for example, Facebook.
7. Verify the authentication by logging in to your application through Facebook.

In an Azure (cloud) environment, you can configure secret values under **Application Settings** for your web app. We will cover Azure deployment in Chapter 9 , *Deploying Your Application.*

To set up an app in an external provider's system, visit their specific website:

- **Facebook**: https://developers.facebook.com/
- **Twitter**: https://apps.twitter.com/
- **Microsoft**: https://account.live.com/developers/applications/
- **Google**: https://developers.google.com/identity/

The instructions and user interface for each provider may change over time, so you should always visit each particular website for the latest instructions. The overall process usually involves creating a named app and then obtaining a set of values that uniquely identify your app.

Let's explore one particular process by integrating Facebook authentication in our application:

1. Go to https://developers.facebook.com/.
2. Register as a new developer.
3. Create a new app.
4. Choose the website platform if asked.
5. Create an **App Id** with a display name of your choice
6. Proceed to the **Settings** screen once your app has been created.
7. Make a note of your **App Id** and **App Secret** values.
8. Click the **Add Platform** button.
9. Select the website platform from the list of selections.

10. Add your website's URL.
11. In the **Advanced** section, add your URL as a **Redirect URI**.

When testing your app during development, you can use a localhost URL with the appropriate port number. To test a deployed website on a server with a domain name, you should use the fully-qualified domain name. Keep in mind that the preceding instructions may be subject to change, so you should refer to the instructions on the Facebook developer website in case something has changed.

Verify the following:

- The project's `userSecretsId` value is defined in your `project.json` file
- `Microsoft.Extensions.Configuration.UserSecrets` is under dependencies
- `SecretManager` is listed in the tools section of the `project.json` file

Open up a command prompt with administrative privileges. Switch to your project directory and then test the Secret Manager tool using the following DNU command:

```
>dotnet user-secrets -h
```

You may have to run `dotnet restore` before the preceding command if you have just added the `SecretManager` to your configuration file. After the `SecretManager` has been verified, follow up with these two additional commands to store your app ID and secret values you obtained from Facebook:

```
dotnet user-secrets set Authentication:Facebook:AppId <Value>
dotnet user-secrets set Authentication:Facebook:AppSecret <Value>
```

Note

Note that the placeholder values above must be replaced by actual values for the commands to work. If the values change in Facebook, you must rerun these two commands to ensure that the latest values are stored in your development environment.

Verify that the constructor in your `Startup` class has the following line of code in it:

```
builder.AddUserSecrets();
```

This line of code must only be run when `IsDevelopment()` returns `true` to ensure that this scenario only exists in a development environment. To verify whether your environment is correctly configured for this, make sure that the environment variable for hosting is set to **Development**, as shown in the following screenshot. You can check this in the **Debug** tab of your web project's **Properties** panel. You can also see it in `launchSettings.json` file under the project's `Properties` folder:
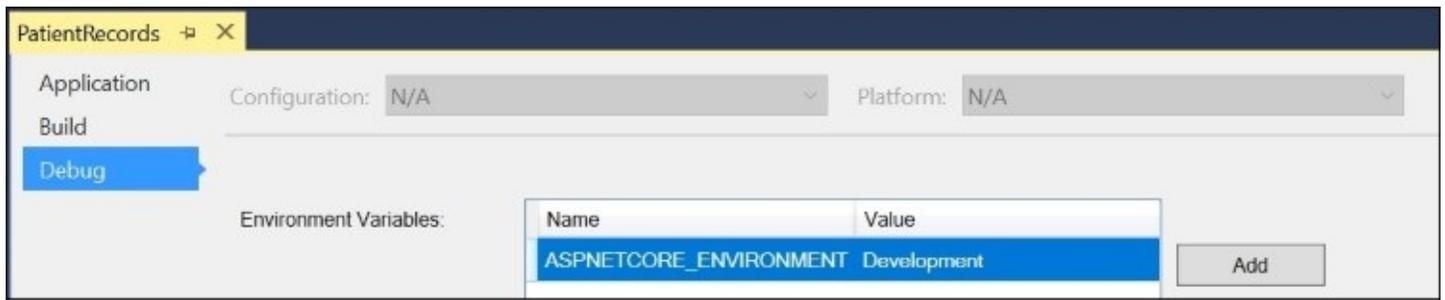
Finally, update your `Configure()` method in `Startup.cs` to use Facebook authentication. Add the following code just after the call to `UseIdentity()`:

```
app.UseFacebookAuthentication(new FacebookOptions()
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});
```

The next time you run your application and attempt to log in, you should see an option for Facebook authentication in the list of other services. In case you're wondering how this Facebook button is displayed, take a look at the `Login.cshtml` view for the `Account` controller in your code, as shown in the following screenshot:
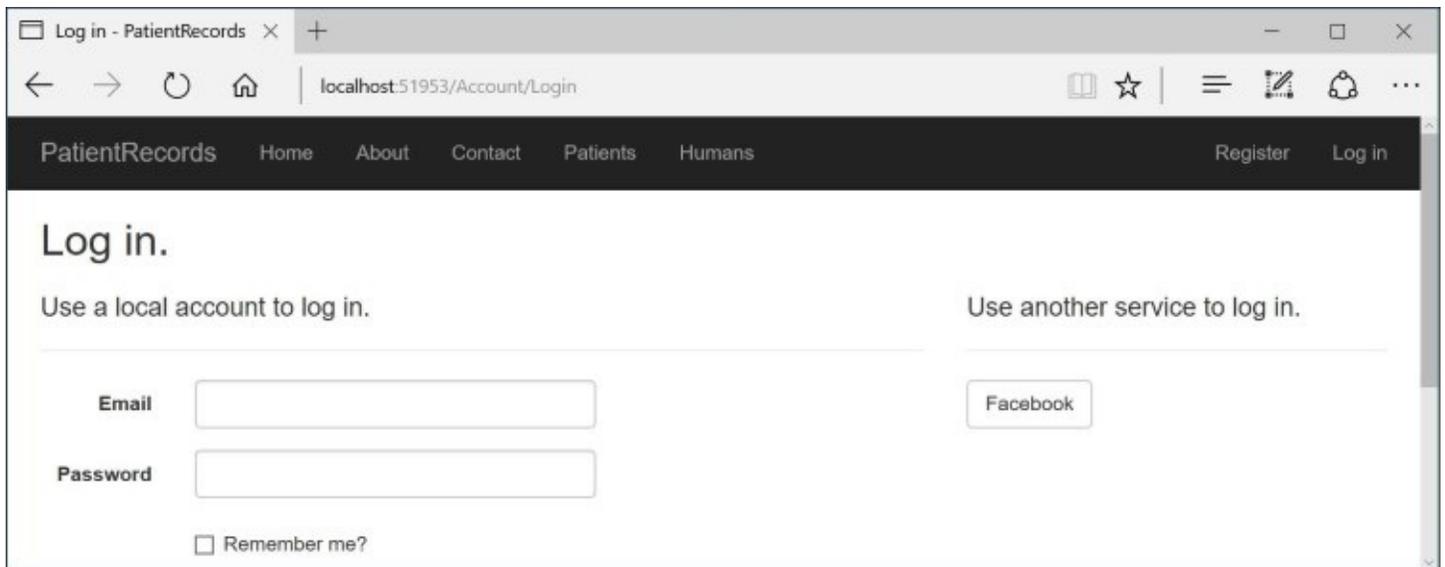
In the `Login.cshtml` view (under the `Account` subfolder of the `Views` folder), there is a block of code surrounded by `@{}` delimiters that checks for external authentication providers. If any providers are found, a button is displayed for each provider, within a `foreach` loop. A simplified version of this server-side code is as follows:

```
var loginProviders = SignInManager.GetExternalAuthenticationSchemes().ToList();
```

```
if (loginProviders.Count == 0)
{
    <p>None configured</p>
}
else
{
    <form
        asp-controller="Account"
        asp-action="ExternalLogin"
        asp-route-returnurl="@ViewData["ReturnUrl"]"
        method="post" class="form-horizontal" role="form">

    @foreach (var provider in loginProviders)
    {
        <button
            type="submit"
            class="btn btn-default"
            name="provider"
            value="@provider.AuthenticationScheme"
            >
        @provider.AuthenticationScheme
        </button>
    }
    </form>
}
```

After you log in, your authentication should be processed by Facebook and you may be prompted to log in to Facebook. If you're already logged in to Facebook in the current window, your current session will be used. In both cases, you should be redirected back to your application and your username will be the login name (for example, your e-mail address) that you may have used to log in to Facebook.

By default, your Facebook account will be the administrator of the Facebook app. When you log in to your web app, you will be asked to associate your Facebook account with a new account in your web app's user system. You can register for a new account by entering a new e-mail address in the association screen. If you want to add additional users during testing, you must first go to the **Roles** tab for your app on the Facebook developers' website, where you can add additional administrators, developers, and testers.

Using authorization for application features

As explained in the introduction to this chapter, authorization can be used to exclude a user from specific parts of an application once they have already been authenticated. In our Hospital Records application, we could restrict certain features so they are accessible to doctors but not nurses or patients.

High-level overview

Here is a high-level overview of how you can implement basic authorization techniques in an ASP.NET Core application:

1. Use the `Authorization` namespace in your controller code.
2. Grant authorization at the controller class level.
3. Grant authorization at the controller action method level.
4. Grant anonymous access at the controller class level.
5. Grant anonymous access at the controller action method level.

Although the first step is required to use authorization, the rest of the suggestions do not have to be followed in any particular order. In fact, you can authorize either a controller class or a method, both, or none at all. In all cases, if you grant anonymous access, it takes precedence over any other authorization. For example, a class with both authorization and anonymous access set will grant anonymous access to any user.

The easiest ways to implement authorization are with **Role-Based Authorization** and **Claims-Based Authorization**. We will cover both of these techniques in this section. For a more complete list of more complex authorization techniques, please refer to the official documentation at the following URL:

http://docs.asp.net/en/latest/security/authorization

Basic authorization

To configure authorization, the quickest way is to simply add the `Authorize` attribute to a specific controller class. The following code shows how this can be applied to the `HumanController`:

```
[Authorize]
public class HumanController : Controller
{
}
```

In order to use the `Authorize` attribute, make sure you have the proper using statement above the class definition:

```
using Microsoft.AspNetCore.Authorization;
```

Run the application from Visual Studio and click on the **Humans** link in the top toolbar to access the `HumanController`. Since the controller has been restricted at the class level, you will be redirected to the **Login** page if you try to access the `HumanController` without logging in first.

To try out anonymous access, add the `[AllowAnonymous]` attribute to the `Index()` action method of the `HumanController` class:

```
[AllowAnonymous]
public async Task<IActionResult> Index()
```

Run the application again, then click on the **Humans** link once again. Even though the controller has been restricted at the class level, the `Index()` method will let you access the `Index` view without having to log in, but trying to access the **Edit**/**Delete** links will prompt you to log in. This overriding behavior works even when you have an `[Authorize]` attribute on an action method along with an `[AllowAnonymous]` attribute.

Roles and claims

Going beyond basic authentication, we can implement Role-Based Authorization and Claims-Based Authorization for even more precise access control. These types of authorization rely on the following ASP.NET user-related tables:

- `AspNetUsers`: User table with user ID, e-mail, password, and so on
- `AspNetRoles`: Master list of roles
- `AspNetUserRoles`: Users mapped to specific roles
- `AspNetUserClaims`: Users mapped to specific claims
- `AspNetRoleClaims`: Roles (and thus, users) mapped to specific claims

To populate the database with some sample data, go through the following steps to create new users, roles, and claims. It is OK to use fake e-mail addresses for testing purposes. To interact with the database, use the **SQL Server Object Explorer** panel in Visual Studio to expand the contents of the Patient Records database.

You can right-click any table and select **View Data** to view (and edit) its data:

1. From Visual Studio, run the application.
2. In your browser, click **Register** in the top right menu.
3. Register a few users (for example, doctor, nurse, patient) in your browser.
4. In the database, verify the users in the `AspNetUsers` table.
5. Add a few roles to the `AspNetRoles` table, with sequential ID values.
6. Add a few records to the `AspNetUserRoles` table to map a user to a role.
7. Add a few records to the `AspNetUserClaims` table to map a user to a claim.
8. Leave the `AspNetRoleClaims` table empty for now.

If you need help with adding data to the `AspNetRoles` table, here are some sample values:

Id	ConcurrencyStamp	Name	NormalizedName
1	NULL	Admins	Admins
2	NULL	Doctors	Doctors
3	NULL	Nurses	Nurses

If you need help with adding data to the `AspNetUserRoles` table, here are some sample values:

UserId	RoleId

cb09fda1-1458-4da1-bee8-3e831d68ca8c	3
e8104b13-72e6-497f-baac-0619548c1e70	2

You should not copy the preceding `UserId` values into your `AspNetUserRoles`. Instead, you should check your own `AspNetUsers` table to copy the **Globally Unique ID** (**GUID**) values assigned to each user's `UserId` field. These are automatically generated by the built-in registration tool. Here is a sample of what the `AspNetUsers` table may look like:

UserId	Email	. . .
cb09fda1-1458-4da1-bee8-3e831d68ca8c	RobotNurse1@fakedomain.com	
e8104b13-72e6-497f-baac-0619548c1e70	RobotDoctor1@fakedomain.com	

Finally, if you need help adding data to the `AspNetUserClaims` table, here are some sample values. In this table, the ID values are automatically generated in sequence:

Id	ClaimType	ClaimValue	UserId
1	DoctorCred	123456789	e8104b13-72e6-497f-baac-0619548c1e70
2	NurseCred	987654321	cb09fda1-1458-4da1-bee8-3e831d68ca8c

Now that your data is all set up, decorate your action methods with the attributes shown here:

```
[Authorize(Roles = "Doctors,Nurses")]
public async Task<IActionResult> Index()

[Authorize(Roles = "Doctors")]
public IActionResult Create
```

The preceding code ensures the following:

- Both doctors and nurses will be able to access the main `Index` view
- Only doctors will be able to create a new entry

This covers the setup and usage of Role-Based Authorization. To use Claims-Based Authorization, you'll have to update your startup configuration and use additional attributes and parameters in your controller code.

Within the `ConfigureServices()` method of the `Startup.cs` file, add a call to `AddAuthorization()` right after the call to `AddMvc()`, as shown here:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("DoctorsOnly",
        policy => policy.RequireClaim("DoctorCred"));

});
```

This will add a new policy to your application called `DoctorsOnly` and requiring the claim `DoctorCred`. You may recall that `DoctorCred` is the `ClaimType` we added to the `AspNetUserClaims` table earlier in this section. To make use of this claim, simply add the following attribute to the `Details()` method, as shown here:

```
[Authorize(Policy = "DoctorsOnly")]
public async Task<IActionResult> Details(int? id)
```

This ensures that only users with the `DoctorCred` claim can access the details of the list of human patients. To go one step further, redefine the claim to include a specific claim number. This value can be a set of one or more comma-separated values:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("DoctorsOnly",
        policy => policy.RequireClaim("DoctorCred", "123456789"));

});
```

Run your application and log in as the various users you created earlier. Verify that you are observing the behavior you expect based on the authorization privileges for each user. Without the proper permissions, you can expect to see an **Access Denied** error page.

Protecting your data

ASP.NET Core includes a new data protection system that can be used in web applications and console applications. You may recall that the list of templates for ASP.NET Core include a console application that we have not needed to use in previous chapters and shown in the following screenshot. However, this project type would be a great choice for illustrating framework features, such as the use of data protection in ASP.NET Core:
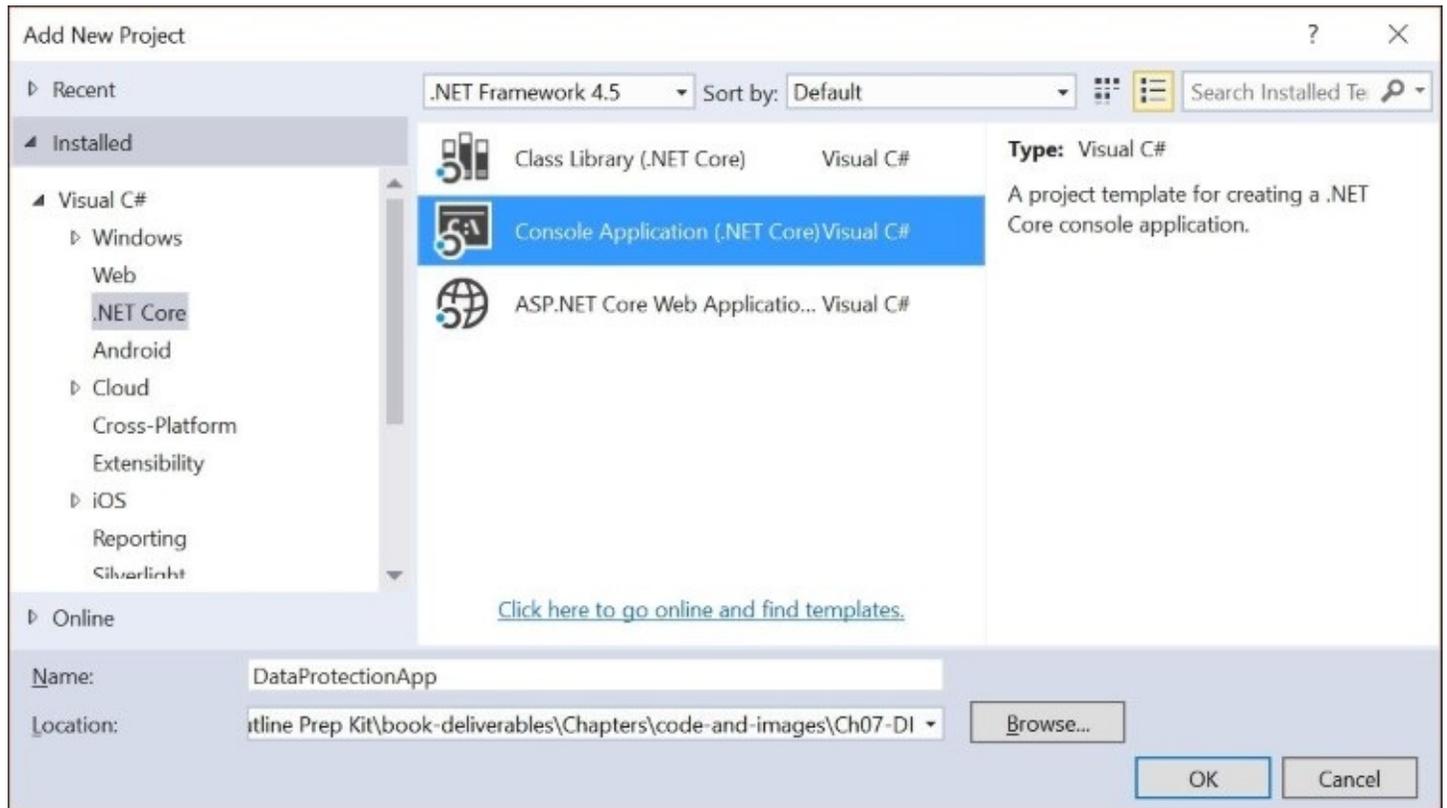
The `<machinekey>` element was used in prior versions of ASP.NET, and the new data protection stack is intended to be its replacement. To make things easier, the new system encourages its use with minimal configuration effort. At the same time, there are extensibility APIs that allow more customization as needed.

Data protection in ASP.NET Core

When a user communicates with a web application, there are many ways to persist and transfer data. Some methods are more persistent than others, while security and encryption may vary greatly. A token provided to an authenticated user needs to be trusted when used later on. That's where data protection comes in.

The ASP.NET team has identified a few requirements when building the new data protection system: authenticity, confidentiality, and isolation. These requirements are all satisfied by ensuring that we can vouch for the integrity of the protected data, while keeping the data safe from an untrusted client. Note that this does not prevent a malicious app from misusing the API to get access to protected data where it shouldn't.

Implementing data protection

Setting up your project with the new **Data Protection API** is fairly simple. This is intentional and allows a developer to get up and running quickly. Keep in mind that the protected data should not be something that you would like to store for an indefinite period of time. It would be technically feasible, but not recommended.

To get started, create a new **Console Application** using the following steps:

1. Click **File** | **New** | **Project** in Visual Studio 2015.
2. From the list of **Installed** templates, choose **Visual C#** | **.NET Core** | **Console Application (.NET Core)**.
3. Enter `DataProtectionApp` as the project name and click **OK**.
4. Open the `project.json` file for editing.
5. Add the following references to the list of dependencies:

```
"Microsoft.AspNetCore.DataProtection":"1.0.0",
"Microsoft.Extensions.DependencyInjection": "1.0.0"
```

As you type in the version numbers, you will most likely use the most stable version available. Within the `DataProtection` namespace, you can get access to a data protection provider and its methods for protecting and unprotecting your data. The `DependencyInjection` namespace will allow you to easily set up your data protection using DI. You will use a so-called purpose string to tie it all together.

Edit the `Main()` method of the `Program.cs` file to include the following code:

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddDataProtection();
var services = serviceCollection.BuildServiceProvider();

var instance = ActivatorUtilities.CreateInstance<DataProtector>(services);
instance.ProtectAndRelease();
```

To use the `ServiceCollection` and `ActivatorUtilities` classes, make sure that you add the following using statement to the top of the `Program` class.

```
using Microsoft.Extensions.DependencyInjection;
```

The preceding code creates an instance of a `DataProtector` class that we will create next. It will have one public `ProtectAndRelease()` method. As you may have guessed, this method will be responsible for protecting some input data, and then releasing it as unprotected data. In a real-world scenario, you probably wouldn't unprotect the data in the same method.

To create the `DataProtector` class, follow these steps:

1. Right-click your project in the Solution Explorer panel.
2. Choose **Add** | **Class** from the pop-up menu.

3. Name the class `DataProtector.cs` and click **Add**, as shown in the following screenshot:
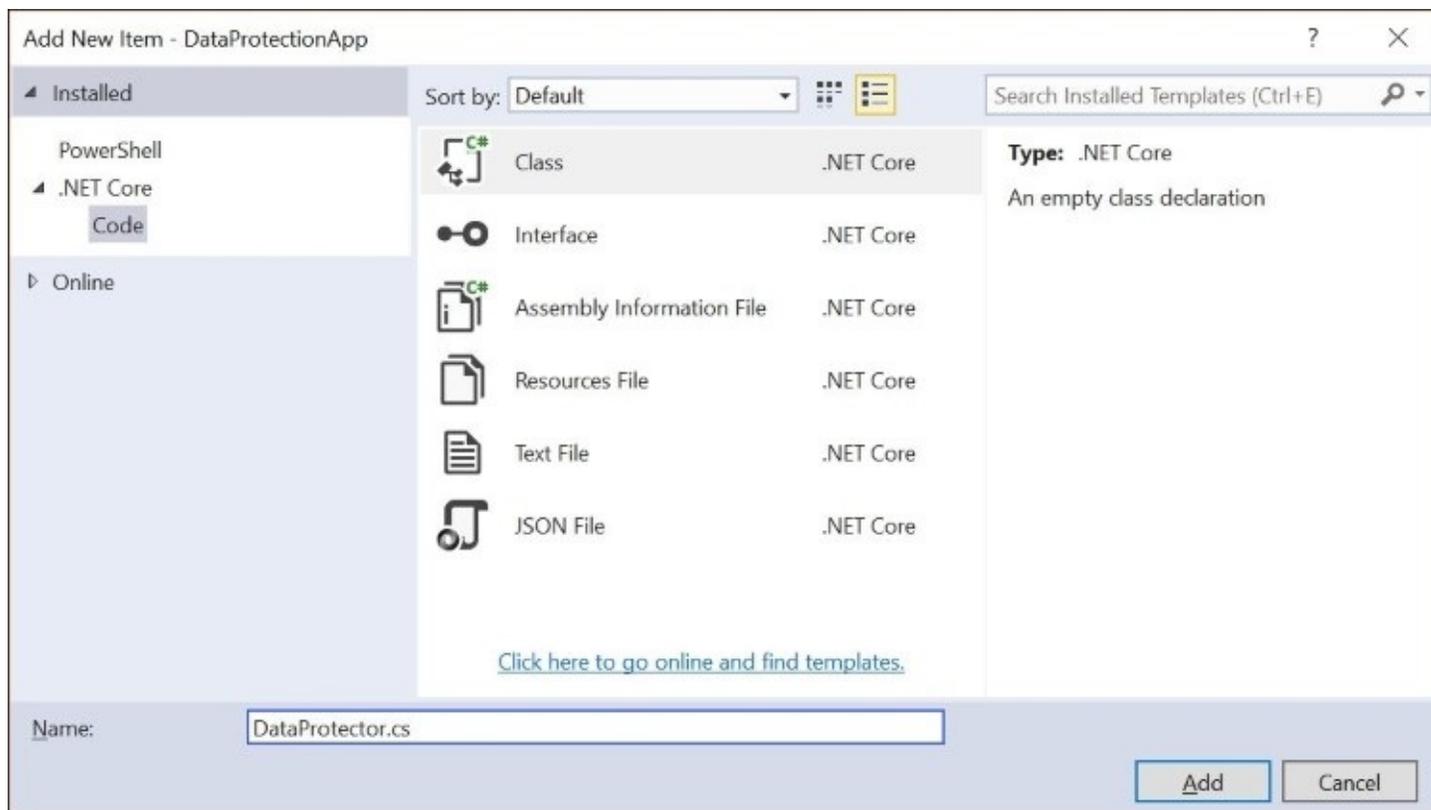
Add the following instance variable and constructor to the `DataProtector` class:

```
IDataProtector _protector;
public DataProtector(IDataProtectionProvider provider)
{
    _protector = provider.CreateProtector("Company.Project.v1");
}
```

In order to use the `IDataProtector` and `IDataProtectionProvider` interfaces, make sure you add the following namespace to the top of your `DataProtector` class:

```
using Microsoft.AspNetCore.DataProtection;
```

The constructor is responsible for creating a new data protection provider object, which in turn is used to create a new protector. The `Company.Project.v1` purpose string is used to create the protector used for this sample. This string could be anything you want it to be, but it is recommended that you make it unique for your application. Elsewhere in the same application, creating a protector with the same purpose string will allow that protector to unprotect the data that was previously protected by the first protector.

Finally, add the following method to take care of protecting and unprotecting your data. The `Protect()` method can take in a string value or a byte array as its input parameter:

```
public void ProtectAndRelease()
{
    Console.Write("Enter input: ");
    string userToken = Console.ReadLine();

    string protectedToken = _protector.Protect(userToken);
    Console.WriteLine($"Protected token: {protectedToken}");

    string unprotectedToken = _protector.Unprotect(protectedToken);
    Console.WriteLine($"Unprotected result: {unprotectedToken}");

    Console.ReadKey();
}
```

You are now ready to run your application. Run the application from within Visual Studio and enter some sample text, for example, a user token in the form of a text string. You should immediately see the output from the `ProtectAndRelease()` method, which shows the protected and unprotected versions of the text you entered:
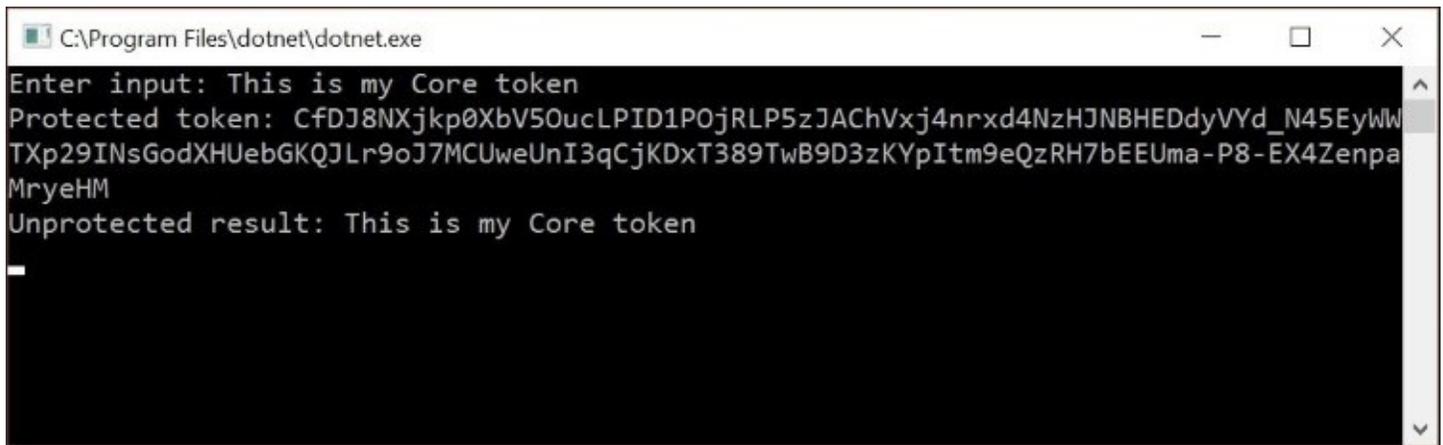
You may be tempted to add this data protection code to your ASP.NET Core controllers in your web project. Instead, you should consider abstracting away service classes to handle specific part of your application to keep your controllers lean.

How it all works

To understand how all of this works, let's go back to the purpose string. The purpose can be provided as a string, a list of strings, or even a daisy-chain of method calls. You could use any one of the formats shown here:

```
CreateProtector("Company.Project.v1");
CreateProtector(new List<string> { "Comp", "Proj", "v1" });
CreateProtector("Comp.Proj").CreateProtector("v1");
```

As long as two provider objects are considered equivalent, one provider can unprotect data that has been protected by another provider. Keep in mind the following points:

- Purpose strings are considered equivalent if they contain the same strings in the same order
- Provider objects are considered equivalent if they were created with equivalent purpose string values
- Protector objects are considered equivalent if they were created from equivalent provider objects
- Internally, the system uses a unique identifier in the purpose chain that is unique for the application

Here are some considerations you should be aware of:

- If any user input is used when building a purpose string, append something to the string to prevent the user from controlling how a purpose string may look in its entirety.
- You should catch a `CryptographicException` where it occurs. If a different data protector is used to attempt an unprotect operation or if a payload has been tampered with, such an exception will be thrown.
- This method of data protection doesn't protect malicious application code from itself. If the same purpose string is reused where it shouldn't be, this may result in equivalent provider objects where you aren't expecting them.

This is by no means a complete picture of all that you need to know about data protection in ASP.NET Core. To learn more, read through the official documentation available at the following URL:

https://docs.asp.net/en/latest/security/data-protection

Implementing web application security

Soon after the web became accessible to the world, web applications started to pop up everywhere. Along with web applications came vulnerabilities that could be exploited by malicious users. Fortunately, security experts and framework developers are constantly providing ongoing advice and better safeguards.

Some common security vulnerabilities over the years have included the following:

- **SQL Injection**: SQL is executed against a database by injecting malicious SQL scripts through HTML form fields whose values are used to build a text string of SQL. By using **LINQ** to entities with an ORM such as EF Core, you can avoid the risk of SQL Injection. If you find yourself using parameterized queries, make sure you sanitize any user input (by HTML-encoding the values) before using them in a query parameter.
- **Sensitive Data Exposure**: Information about the server, file system, database, and operating system may be unnecessarily exposed in a production setting, especially during error conditions. It is good practice to avoid revealing too much information to the end user. Instead, take advantage of built-in logging features to store the details elsewhere if an error occurs, while displaying a numeric error code and friendly message to the user.
- **Cross-Site Scripting (XSS)**: Malicious script code is injected into the body of an HTML page, usually through `QueryString` parameters. Fortunately, there are multiple ways to fend off XSS attacks.

The rest of this section explores some techniques for fighting XSS, preventing forgery, and enabling cross-origin requests.

Cross-site scripting

To prevent cross-site scripting attacks from malicious users, you may have used the **AntiXSS** library with a previous version of ASP.NET. However, AntiXSS is has been considered end-of-life as of late 2015. As a result, the AntiXSS library is not compatible with .NET Core 1.0.

The `Sanitizer` object from the most recent version of the AntiXSS library included a method to convert potentially unsafe HTML code to a safer alternative. As a replacement, you can let the **Razor** engine in ASP.NET MVC automatically encode the values of variables from untrusted sources. The end goal is to ensure that the users of your application can't inject malicious client-side scripts into your HTML body.

For more details on HTML Encoding in ASP.NET MVC web apps, check out the section on HTML Encoding on the XSS section of the ASP.NET documentation:

https://docs.asp.net/en/latest/security/cross-site-scripting.html

Anti-forgery

The `[ValidateAntiForgeryToken]` attribute provides built-in anti-forgery support for your web applications. It generates an HTTP-only cookie with a value that is also written to the HTML form. If there is a mismatch in the value after the HTTP `POST` operation occurs, this signals a red flag that a **Cross-Site Request Forgery** (**CSRF**) may have occurred; that is, the form being submitted has been altered from what was originally provided by the server.

You may have already noticed that this `[ValidateAntiForgeryToken]` attribute has been applied to several action methods in the `AccountController` provided by the web project template:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> MethodName(...)
{
        // ...
}
```

If you need to disable this feature for a particular form, you can set the tag helper `asp-anti-forgery` attribute to false in the desired view:

```
<form asp-controller="controllerName"
      asp-antiforgery="false"asp-anti-forgery="false"
      asp-action="methodName">
</form>
```

Cross-origin requests

Due to the so-called *same-origin policy,* web browsers typically prevent web pages from making AJAX requests to a page on a different domain. However, your web app may have a need to accept an AJAX request from a different domain. Fortunately, the **World Wide Web Consortium** (**W3C**) organization defines a standard named **Cross-Original Resource Sharing** (**CORS**) that enables cross-origin requests.

To determine whether two URLs have the same origin or not, compare the domain, subdomain, port number, and scheme (HTTP/HTTPS) of each URL. If they are different, that's where CORS becomes a useful tool. To enable CORS in ASP.NET Core, follow the simple steps outlined here:

1. Add a reference to the CORS package to your `project.json` file.
2. Update the `ConfigureServices()` method to add `Cors`.
3. Update the `Configure()` method to use `Cors`.
4. Customize CORS behavior with attributes in your controller code.

To add a reference to your project, add the following package in the dependencies section of your `project.json` file:

```
"Microsoft.AspNetCore.Cors": "1.0.0"
```

To set up your configuration, add the following code to `AddCors()` in the `ConfigureServices()` method of your `Startup` class. You may add it right after the call to `AddMvc()`:

```
services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin",
        builder =>
        {
            builder.WithOrigins("http://fakedomain.com");
        });
});
```

The preceding code adds CORS functionality to your application while explicitly specifying the named `AllowSpecificOrigin` policy. It allows a cross-origin request from fakedomain.com to come through to your website to be processed.

To enable CORS at the controller class level, use the `[EnableCors]` attribute:

```
[EnableCors("AllowSpecificOrigin")]
public class MyController: Controller
```

To enable CORS at the controller method level, use the same attribute:

```
[EnableCors("AllowSpecificOrigin")]
```

```
public IActionResult ActionMethod()
```

In both cases, add the proper using statement to include the CORS namespace:

```
using Microsoft.AspNetCore.Cors;
```

The setting at the method level takes precedence over the setting at the controller level, which takes precedence over the setting at the application level. This allows us to disable CORS at any level we choose, even if it has been enabled at the application level. For example, you can add the `[DisableCors]` attribute to a specific method, if you need to disable CORS for just that particular method.

To add the `AllowSpecificOrigin` policy in your application at a global level, add the following code to your `ConfigureServices()` method. You may add this code after the call to `AddMvc()`:

```
services.Configure<MvcOptions>(options =>
{
    options.Filters.Add(
        new CorsAuthorizationFilterFactory("AllowSpecificOrigin"));
});
```

In order to use the `MvcOptions` and `CorsAuthorizationFilterFactory` classes in your `Startup` class, you will have to add the following namespaces to the top of the class:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Cors.Internal;
```

Summary

In this chapter, we discussed some ways of providing access to your application and other ways of restricting access to it. You can use authentication to verify a user's identity while you can use authorization to enable access to specific parts of the application. You can protect small bits of data using the new data protection system, while you can take several different actions to protect your application from well-known security risks.

In the next and final chapter we will cover the process of deployment. A web application is no good if it only sits in your development environment. By deploying your application to a public website, you can reach a global audience. By deploying to the cloud, you can scale your website to meet public demand.

Chapter 9. Deploying Your Application

There are plenty of ways for you to get your web application to the world out there. But there are additional considerations beyond deploying just the web project. Typically, you would also deploy your database but you wouldn't want your test data in production. You should maintain separate configuration settings so that your project can point to the correct database in each environment.

Visual Studio and ASP.NET Core help you maintain your project files and settings across different environments with simple techniques and conventions. With the help of Microsoft's Azure cloud platform, you can easily deploy your website within seconds and scale it up and down to meet demand.

Deployment options

This chapter will focus on deploying your ASP.NET Core application to **Internet Information Services** (**IIS**) (Microsoft's web server product) and Azure (Microsoft's cloud service). If you plan on deploying to operating systems other than Windows, please consult the documentation for your specific operating system. Before you proceed with deployment, you should be familiar with the environment configuration settings.

Environment configuration

In [Chapter 8](#) , *Authentication, Authorization, and Security,* we took a quick peek at the environment variable that defines the hosting environment, for example, development, staging, or production. You may recall that a list of environment variables appears in the **Debug** tab of your web project's **Properties** panel.

Open the `launchSettings.json` file to see the same variable in a configuration file:

```
"environmentVariables": {
  "ASPNETCORE_ENVIRONMENT": "Development"
}
```

You may have to expand the **Properties** node in the Solution Explorer panel to find the `launchSettings.json` file:
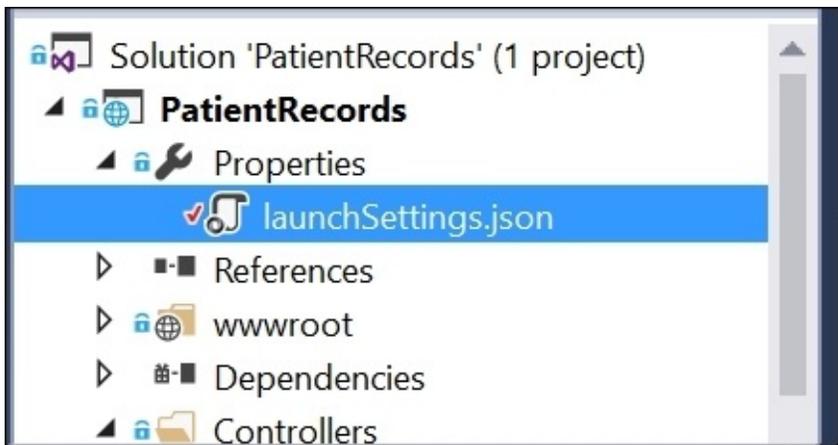
The `Configure()` method of the `Startup.cs` file takes in an `IHostingEnvironment` parameter named `env`. This can be used to check the environment type, for example:

```
if (env.IsDevelopment()) { /* do something */ }
```

There are several useful properties and methods, including the following:

- `env.EnvironmentName`: Read/write property used to get or set the name of the environment
- `env.IsDevelopment()`: Returns `true` if the `EnvironmentName` is `Development`
- `env.IsStaging()`: Returns `true` if the `EnvironmentName` is `Staging`
- `env.IsProduction()`: Returns `true` if the `EnvironmentName` is `Production`
- `env.IsEnvironment(string environmentName)`: Returns `true` if the passed argument `environmentName` matches the current `EnvironmentName`

On Windows, the value of `env.EnvironmentName` is set by the `ASPNETCORE_ENVIRONMENT` variable. On Linux or OS X, the environment setup will vary, so please consult the ASP .NET

documentation for non-Windows deployments, as the documentation is still evolving. The suggested values are merely a convention, but are recommended as a starting point.

If you need to display different elements in your MVC views based on the environment, you can use the environment tag helper, as shown in the following snippet. Multiple environment names can be used in a comma-delimited list:

```
<environment names="Development">
    <p>For Development Only</p>
</environment>
<environment names="Staging,Production">
    <p>For Staging and Production Only</p>
</environment>
```

To simulate these different environments while testing your application, you may create a different profile for each environment in the **Debug** tab of your project's **Properties**. Each time you add a new **Profile**, you can add the corresponding environment variable and assign the appropriate value. The related configuration file will be updated automatically. When you deploy your application to other environments, such as staging and production, simply choose the appropriate profile for a particular build configuration.

To help your application find your database, the database connection string can be stored in your `appSettings.json` file, located in the root location of your web project:

```
"ConnectionStrings": {
  "DefaultConnection": "<full connection string goes here>"
}
```

In a server environment, you can define environment variables directly on the server. In an Azure-hosted environment, you can set up environment variables through the **Azure Portal**.

Deploying your web app

There are several ways you can deploy your web app. You could simply right-click your web project in Solution Explorer and select the **Publish** option, as shown in the following screenshot. This will present you with the following publish targets:

- **Microsoft Azure App Service**: You can either deploy to an existing web app on Azure (formerly known as **Azure Website**) or create a new web app through Visual Studio.
- **Import**: You can import a **Publish Profile** to prepopulate the fields that identify your target location. This can come from a hosting provider, network administrator, or development team. It can also be exported from an Azure web app.
- **File System (under Custom)**: This option is fairly rudimentary but gives you the option of exporting your deployed application directly to your file system. You can copy the exported files to a target destination of your choice or compress them to transfer them elsewhere. The **Custom** section also offers other options such as **Web Deploy**, **Web Deploy Package**, and **FTP**. Choose an option that suits your needs and complete the fields as necessary.
- **Other Hosting Options**: You can visit a public gallery of web hosting options, with the ability to choose from an assortment of hosting partners at the following URL: http://hosting.asp.net/hosting.

Publish   ?   ✕

🌐 Publish

┌─────────────┐
│ Profile   │   Select a publish target
└─────────────┘

                       ┌───┐
                       │ 🖼   Microsoft Azure App Service   │
                       └───┘

                       ┌───┐
                       │ ➡   Import   │
                       └───┘

                       ┌───┐
                       │ 📄   Custom   │
                       └───┘

                       Find other hosting options at our web hosting gallery

                                      < Prev   Next >   Publish   Close

Deploying your database

SQL Server is a common choice for many ASP.NET developers, and it works well with **Entity Framework**. There are many different ways for you to deploy your database to staging and production environments. For some developers, this responsibility may be handed off to database administrators or system administrators.

Here are a few options that you may consider:

- Write SQL scripts to generate the database objects on the server
- Use **Entity Framework Migrations** to generate SQL scripts to run against the server
- Use Entity Framework Migrations to generate the database

You may end up using a combination of these options, but the best option is one that you and your team can agree on. In some cases, you may have to discuss the database creation method with your database administrator, customer, or IT manager. You may decide that the best option is to generate the SQL scripts without running them right away. This will allow you to inspect the scripts and make modifications if necessary, before actually affecting your database.

Depending on your scenario, you may have to automate the insertion of seed data through code or scripts. Ideally, your application code should be prepared to handle the absence of data.

Deploying to IIS

Microsoft's IIS web server predated even the first release of **Active Server Pages**. IIS supports a wide variety of web frameworks and languages, including ASP.NET with C#. You can run IIS on your own Windows server or in a hosted environment.

To deploy to IIS, you can choose any of the following options:

- Copy web application files directly to the web application root on the server
- Use Visual Studio's aforementioned publishing feature
- Use a Continuous Integration system to automate web deployments

The following screenshot shows the IIS Manager:
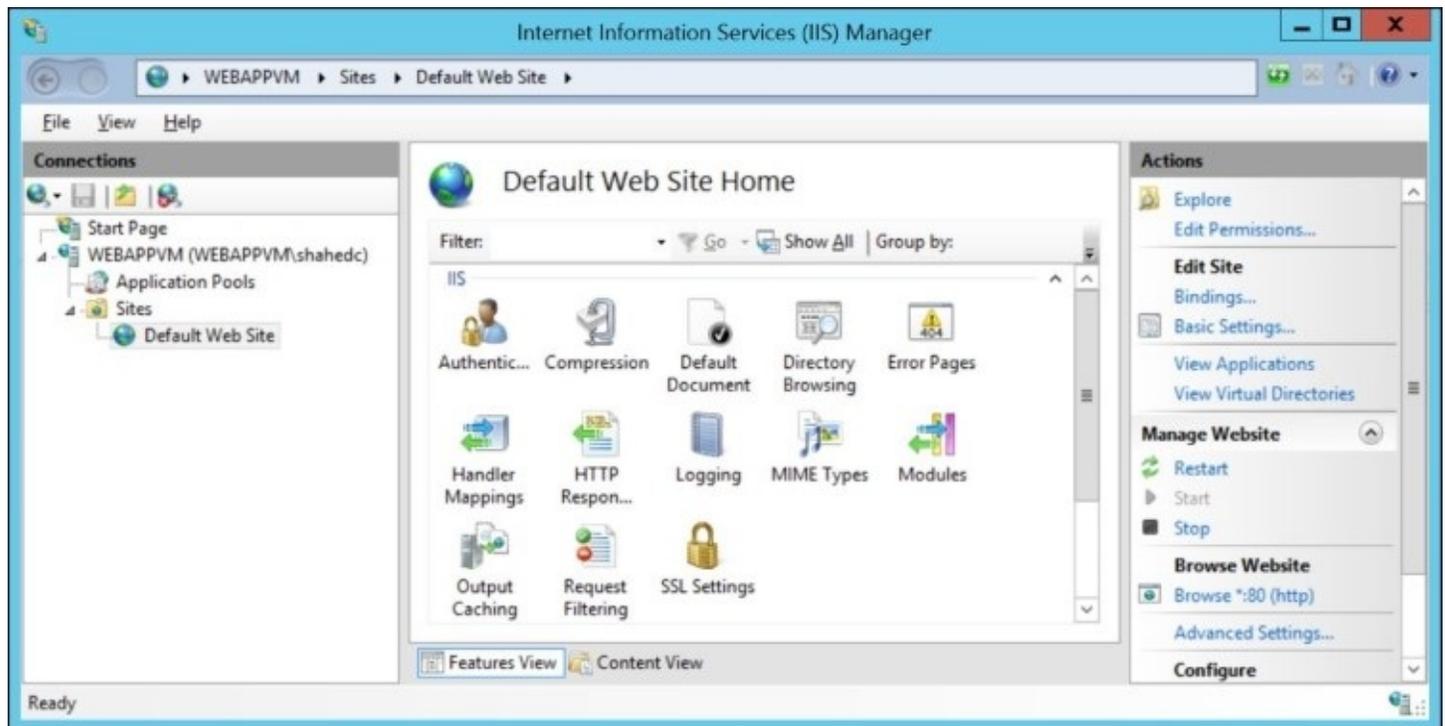
Setting up IIS

Before you can deploy to IIS, you must ensure that IIS is properly installed and configured on your server. Typically, a website will run on port 80 for HTTP access and port 443 for HTTPS access. To enable **Secure Sockets Layer** (**SSL**), you will also have install a certificate.

If you're not familiar with setting up IIS, please consult the official tutorials at the following URL:

http://www.iis.net/learn

Setting up IIS involves the following steps:

1. In **Server Manager** on Windows, enable the **Web Server (IIS)** role.
2. In IIS manager, set up a web application to run on the desired port(s).
3. Set the application pool to **No Managed Code**, as ASP.NET Core manages its own runtime.

You may need to consult your administrator to set up IIS in your work environment. For additional information, please consult the official documentation at the following URL:

https://docs.asp.net/en/latest/publishing/iis.html

Using the filesystem

Once the web server has been set up to run ASP.NET Core applications, you can copy the files from your web application directly to the web root location on your web server. From Visual Studio, you can use the publishing feature to export your files to a folder destination you can access through the filesystem.

To publish to a filesystem, follow these steps:

1. Right-click your project in Solution Explorer, then click **Publish**.
2. Select the **File System** option under **Custom**.
3. Name your publishing profile.
4. Select a target location, then click **Next**.
5. Preview the connection information, then click **Next**.
6. Update the settings and configuration as desired, then click **Next**.
7. Preview the files to be published, then click **Publish**.

The settings screen includes a checkbox that can be enabled to remove additional files that are already present in the target location. This can be useful if the file list changes between deployments. Note that this option is present whether you are deploying to Azure or to a file system destination, as shown in the following screenshot:
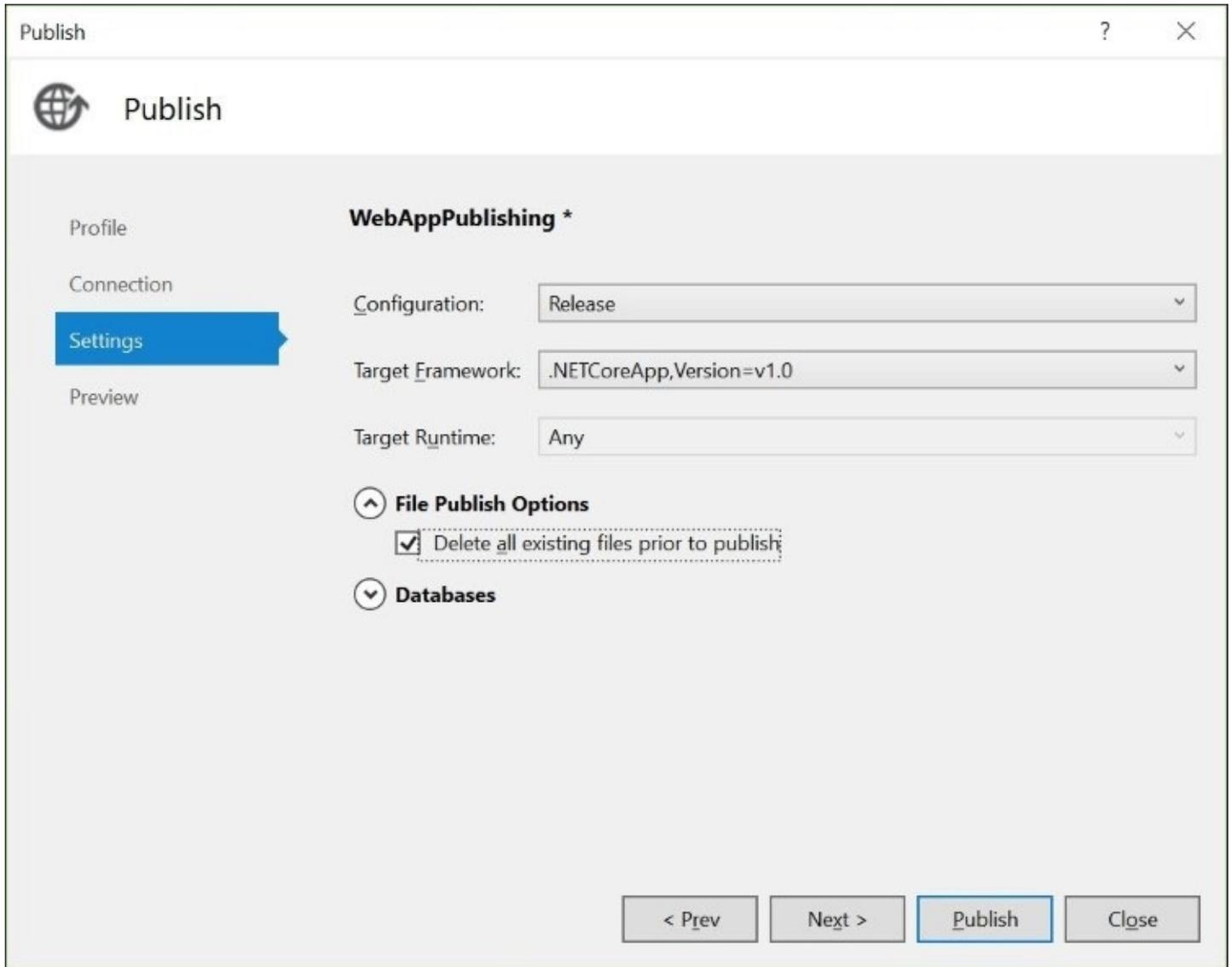
Importing a publish profile

In order to import a publish profile, you will need a publish profile file that you generated yourself, received from a team member or obtained from a web host. You can then choose the **Import** option in the **Publish** dialog box. If you're not sure how to generate your own publish profile, follow the instructions in the next section to create your own web app. Then, browse to the web app's *blade* in the Azure portal and click the **Get publish profile** button in the web app's toolbar.

To publish using an imported publish profile, follow these steps:

1. Right-click your project in Solution Explorer, then click **Publish**.
2. Select the **Import** option.
3. Browse to a publish profile file.
4. Preview the connection information, then click **Next**.
5. Update the settings and configuration as desired, then click **Next**.
6. Preview the files to be published, then click **Publish**.

Deploying to Azure, Microsoft's cloud platform

Azure has become a popular cloud platform for many developers across the globe, whether the application is using a Microsoft framework or an open-source alternative. For ASP.NET developers, Azure is a no-brainer. Under the **Free** plan, you can create up to 10 free websites on Azure, which provides a playground for web apps to live out in the wild.

To get started with Azure, you may sign up for a free trial at http://azure.com .

Creating a web app

Once you have signed up for Azure, you may create a web app using the Azure portal. This process takes only a few seconds. Once kicked off, the web app should be up and running within a minute.

To create a web app in Azure, follow these steps:

1. Log in to the Azure portal at http://portal.azure.com.
2. Click on the **New** button on the top left, marked with a plus sign.
3. Select **Web + Mobile** from the list of options.
4. Select **Web App** from the list of options.
5. Fill out the required information, then click **Create**.

Allow Azure a few seconds to generate your new web app. Once your app has been created, you should be redirected to the **Dashboard**. If you pin the web app to the **Dashboard**, you should see a tile that updates you on the status of your web app. Once completed, you should be redirected to the **Settings** blade of the web app you just created.

The following screenshot shows the creation of **Web App**:
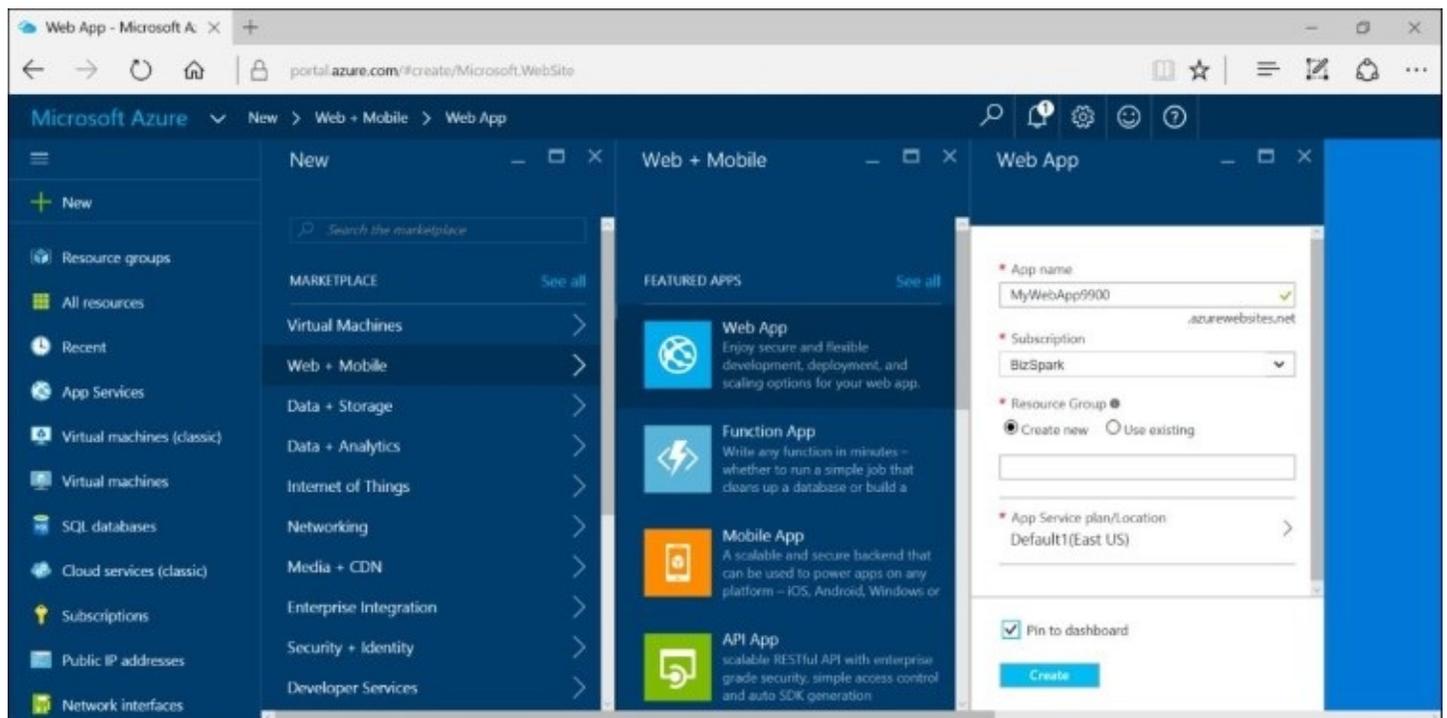
Note that the Azure UI will continue to evolve over time, so the instructions in this chapter may not reflect exactly what you see on your screen. However, the basic flow of instructions can be used as a close approximation of what you need to do. For the latest instructions,

please refer to the official documentation at the following URL:

https://azure.microsoft.com/documentation

Creating a virtual machine

If you need the full control of a **virtual machine** (**VM**), you can also create a VM on Azure to deploy your applications. Note that a VM will typically cost more than a simpler web app. It will also take more effort to create and configure the machine and the web server.

To create a VM in Azure, follow these steps:

1. Log in to Azure if you haven't done so already.
2. Click on the **New** button on the top left, marked with a plus sign.
3. Select **Virtual Machines** from the list of options.
4. Select a featured item, for example, **Windows Server 2012 R2 Datacenter**.
5. Select a **Deployment Model** (if available), then click **Create**.

Although you could choose a **Classic deployment model**, it is recommended that you choose the **Resource Manager** option as your deployment model. This will allow you to easily manage related resources in Azure, along with your VM.

In the next blade that appears, you must enter additional information about your VM:

- **Basics**: Enter a machine name and admin credentials then select a subscription, resource group, and location. If you don't have any resource groups yet, you may create a new one before selecting it.
- **Size**: Choose one of the many preconfigured machine sizes. Your selection will determine the number of cores, amount of RAM, amount of disk space, auto-scaling capabilities, and load balancing features. Keep in mind that it will cost you more to create VMs with more features and capabilities.
- **Settings**: Choose your storage account, virtual network, subnet, public IP, default security group, diagnostics storage account (if diagnostics enabled), and availability set. An availability set allows you to add multiple machines to the same set to avoid all the machines going down at the same time, enabled by placing the VMs across different physical servers/racks.

Allow Azure about 8 to 10 minutes to generate your new VM. Once your VM has been created, you should be redirected to the **Dashboard**, where you should see a tile that updates you on the status of your VM. Once completed, you should be redirected to the **Settings** blade of the VM you just created.

In the main blade of the VM, the toolbar of icons should have a **Connect** icon that allows you to download a `.rdp` file. This file can be opened in remote desktop in Windows (or an other operating system) to interact with the VM directly, as if you were in front of it.

To connect to the VM directly through the remote desktop, use the following format for your credentials:

- **Login name**: `<machinename\username>`

- **Login password**: `<password>`

The following screenshot shows the creation of the VM:
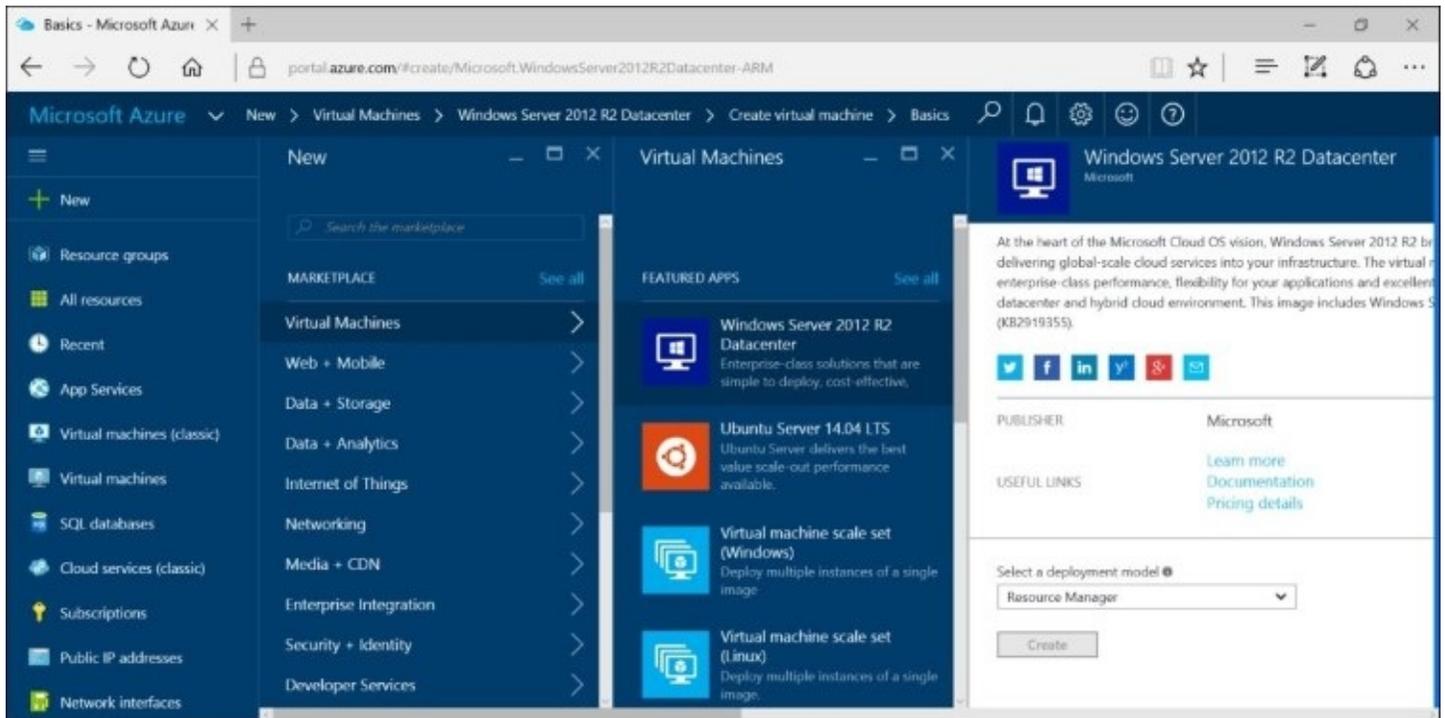
Once you access the machine, you will have to set up IIS manually and configure the VM in Azure to enable the desired ports. These additional settings in Azure can be accessed through the **Network Security Group** that your VM is associated with. If you cannot locate it, click on **All Resources** on the left menu, where you can filter the list of all your resources, and select the network security group that shares the same name as your VM. In the **Settings** blade for this group, you should be able to manage the group's properties and inbound/outbound security rules. By default, an RDP port should be enabled for you.

Deploying to Azure

In order to deploy to Azure, you can either deploy your web project manually, or set up automated deployment through continuous integration. Visual Studio 2015 makes it really easy to deploy your web application to your web app in Azure.

To deploy to Azure from Visual Studio, follow these steps:

1. Right-click your project in Solution Explorer, then click **Publish**.
2. Select the **Microsoft Azure App Service** option.
3. Log in to Azure if you haven't done so already.
4. Select a **Web App** or create a new one.
5. If applicable, select a specific **Deployment Slot**.
6. Preview the connection information (if applicable), then click **Next**.
7. Update the settings and configuration as desired, then click **Next**.
8. Preview the files to be published, then click **Publish**, as shown in the following screenshot:
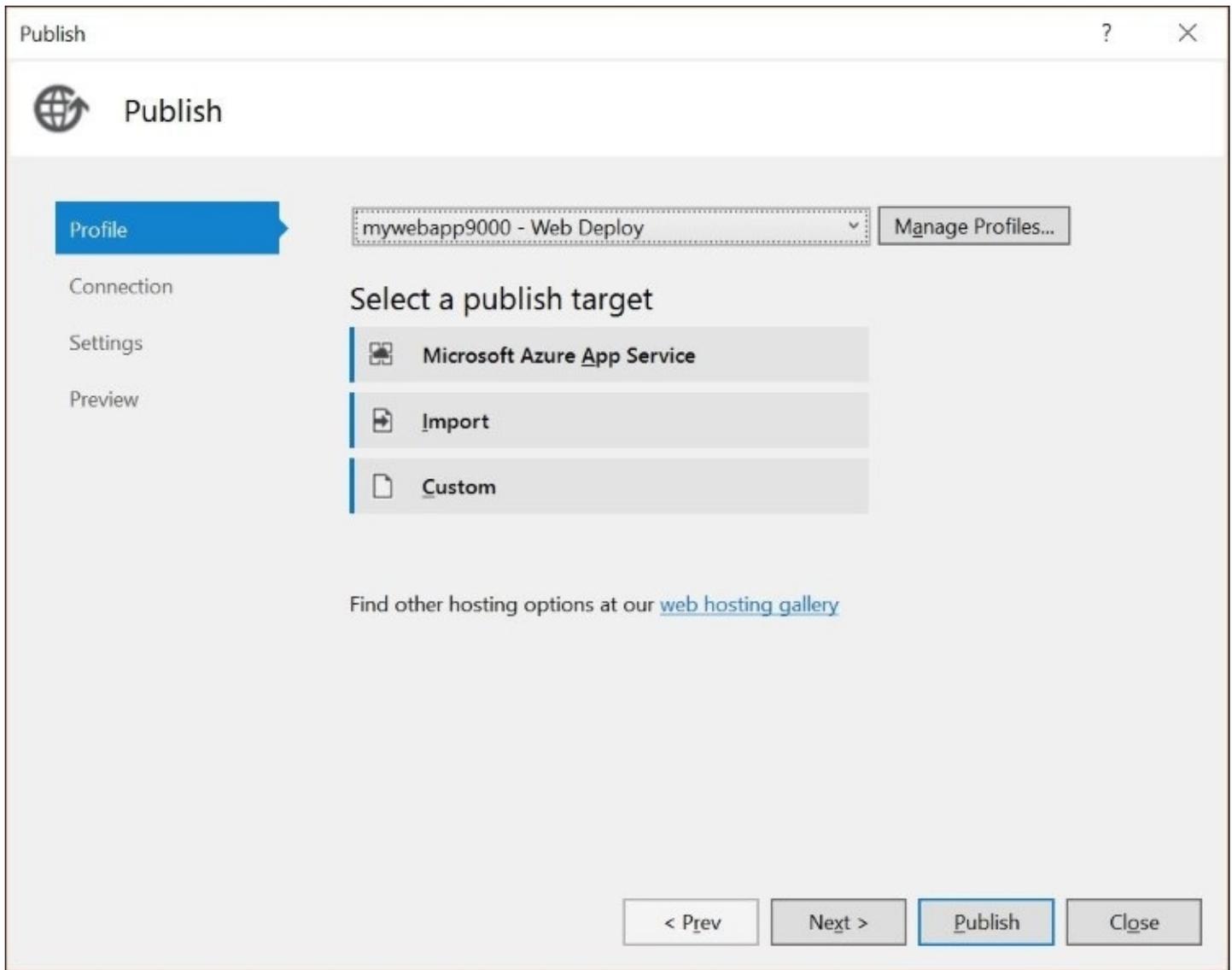
This process usually takes a couple of minutes for a small application, but the publishing time may vary depending on the quantity and size of your files. Once deployed, your web browser should pop up with the website loaded from the URL of your web app on Azure. The first time the website loads, it should take a few seconds for the app to start up on the web server for the first time. Subsequent loads should be almost instantaneous.

If your website doesn't get any visitors for a while, it is considered idle. If this happens, the web app is unloaded by Azure until the next visitor requests the site. The first visitor to reach an idle site will trigger the site to load up again, which will take a few seconds. Although this helps you conserve resources, you may want to leave the web app loaded at all times. In the **Application Settings** blade for your web app in Azure, you can enable a feature called **Always On** to ensure that it is always loaded.

Continuous integration

Whether you're working on a project alone or with a team of developers, you will most likely use a source control system such as **Git**, **Subversion**, **Mercurial**, or **TFS**. To help automate the deployment process, it can be very efficient to deploy the web app directly from source control through a **continuous integration** (**CI**) system.

There are a few different options to consider when implementing CI:

- **Team Foundation Server** from Microsoft
- **Visual Studio Team Services**, a hosted option from Microsoft
- **TeamCity** from JetBrains

To get started with TeamCity, you may download it from here:

https://www.jetbrains.com/teamcity

To get started with TFS, you may download it from here:

https://visualstudio.com/products/tfs-overview-vs.aspx

To get started with VSTS, you may sign up here:

https://visualstudio.com/products/visual-studio-team-services-vs.aspx

Preparing for CI

Not everyone on your team may be ready for CI. In fact, some team members and even project managers may be opposed to it. It is important for you to discuss the pros and cons of your CI decisions and ensure that you have a successful strategy to implement it for your project.

Consider the following concerns:

- *We don't have time for CI*: If you hear this, you should consider the time you will lose when trying to deploy an application to a staging server in the middle of the night, when your IT administrator is on vacation. Automating your deployments will allow you to save more time in the long term.
- *We can't afford to do CI*: If you hear this complaint, you should consider the cost of not having a CI system. A CI system will allow you to run unit tests in addition to performing deployments. You could spend costly hours (or even days/weeks) trying to troubleshoot complex problems that get introduced when fixing bugs or adding new features. A CI system will help pinpoint these problems early and save time in the long-term.
- *We won't get buy-in from our clients or upper management*: If you are exposed to this type of fear, you should convince your team to talk to your clients about the benefits of CI. Having a CI system in place could mean that key stakeholders will have access to a live version of the web application at all times, even during development iterations. This will enable an agile development cycle, which will encourage early feedback.

Setting up Continuous Deployment

Setting up **Continuous Deployment** (**CD**) with Azure is fairly simple. You can choose from a list of options in the Azure portal. Once CD has been set up, every check-in into your source control system will trigger a web deployment to Azure.

To get started, complete the following steps:

1. Log in to Azure if you haven't done so already.
2. Select your web app from the **Dashboard**.
3. Browse to the settings for your web app.
4. In the **Settings** blade, look for **Deployment** options to configure your deployment source.
5. In the list of sources, choose a source.

Here is an example of what you may expect to see in the list of sources:

- Visual Studio Online
- OneDrive
- Local Git Repository
- GitHub
- Bitbucket
- Dropbox
- External Repository (such as Git, Mercurial, and others), as shown in the following screenshot:
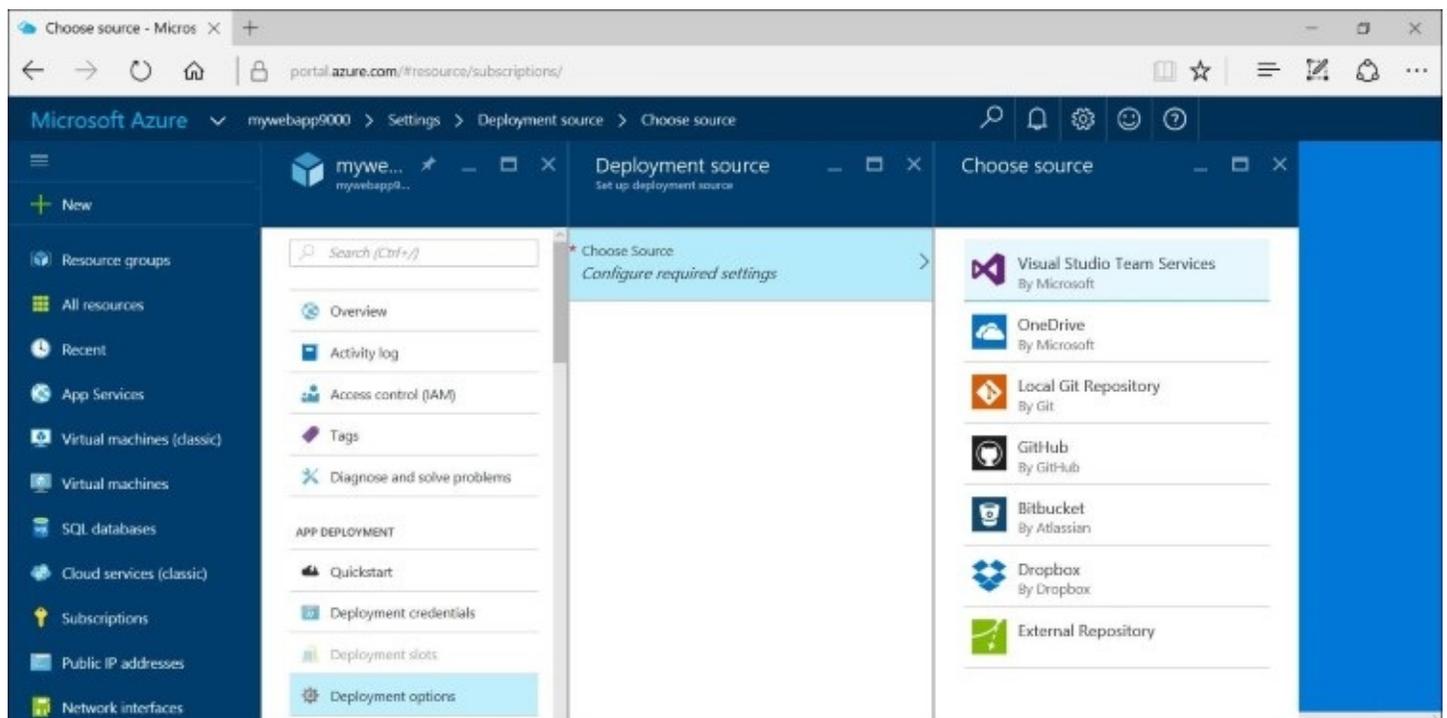
For the option you have selected, fill in the required information to connect to the corresponding service to establish a connection. Once the connection is established, your credentials will be used for future deployments. Azure allows you to set up deployment slots for staging and production environments that can be swapped as needed. This should help you speed up the time taken to get the latest stable build into production.

If you're interested in learning more about running unit tests in the cloud with **Visual Studio Team Services**, please refer to the official documentation:

https://www.visualstudio.com/docs/test/developer-testing/developer-testing

The pitfalls of CI

Now that you have set up CD and CI, what could go wrong? Years of experience have taught many developers the pitfalls to watch out for. As long as you are aware of what to expect, you will be well prepared to handle challenges in your CI journey.

Here is a list of challenges that you may expect to encounter:

- **Learning curve**: If your team is new to CI, there will be a learning curve for your team members. There is no way around this. Do what it takes to get your team up to speed. Rest assured that the benefits will soon outweigh the initial hurdles.
- **Upfront cost**: As you get over the learning curve, you will face some upfront costs in terms of work hours and possibly server expenses. Once again, do remind yourself that the extra costs up front will save you money in the long run.
- **False positives**: Once everything has been set up, some of your unit tests may result in some false positives. This may be caused by poorly-written tests, or some obscure runtime scenarios that may not have to be addressed.
- **False reassurance**: You may also have poorly-written tests that result in false reassurance. A bad test could signal that everything is OK, while ignoring an error condition that is waiting for some user to discover it at a future date.
- **Long-running deployments**: If you have a lot of code and a lot of heavy tests, you may have long-running deployments. The trick is to refactor your codebase to reduce unnecessary code and to optimize unit tests so that they run quickly and efficiently.

Once you get through the initial setup and have optimized your code and tests, you should be well on the way to a smooth deployment process. As you continue to make changes to your code and infrastructure, you may occasionally have to revisit your continuous deployment strategy with your team. This will help your team bring up any concerns that need to be addressed.

Summary

In this chapter, we covered various deployment options and focused on deploying an ASP.NET web application to Azure. We wrapped up with a quick introduction to CI and CD.

Going forward, you should stay tuned to announcements from Microsoft to learn more about ASP.NET Core and Azure. There are several websites, blogs, and videos online that may be useful to you.

Here is a list of online resources that you may find useful:

- Official ASP.NET website (http://asp.net)
- Downloads (https://www.microsoft.com/net)
- Documentation (https://docs.asp.net)
- ASP.NET Community Standup (https://live.asp.net)
- Microsoft Virtual Academy (https://mva.microsoft.com)
- MSDN Channel 9 (https://channel9.msdn.com)