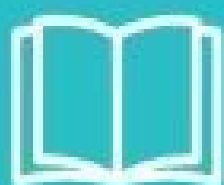Mugilan T. S. Ragupathi,Valerio De Sanctis,
James Singleton

ASP.NET Core: Cloud-ready, Enterprise Web Application Development

Learning Path

Create fast, scalable, and high performance applications with C# and ASP.NET Core

Packt>

ASP.NET Core: Cloud-ready, Enterprise Web Application Development

Table of Contents

ASP.NET Core: Cloud-ready, Enterprise Web Application Development

ASP.NET Core: Cloud-ready, Enterprise Web Application Development

Create fast, scalable, and high performance applications with C# and ASP.NET Core

A course in three modules

BIRMINGHAM - MUMBAI

ASP.NET Core: Cloud-ready, Enterprise Web Application Development

Credits

Authors

Mugilan T. S. Ragupathi

Valerio De Sanctis

James Singleton

Reviewers

Anuraj Parameswaran

Mustafa Erhan Ersoy

Anand Ranjan Pandey

Vincent Maverick Durano

Jason De Oliveira

Content Development Editor

Kinnari Sanghvi

Graphics

Abhinash Sahu

Production Coordinator

Shraddha Falebhai

Preface

If your website is dynamic and you have to use some server-side scripting, you can use ASP.NET, and the fact that you are using AngularJS won't make your server-side code more MVC. If you want to be able to structure your server-side application as MVC, you'll rather pick ASP.NET MVC. The learning path aims to help you learn the fundamentals of ASP.NET Core MVC and apply that knowledge to building applications using ASP.NET Core. This book also aims to serve as a solid guide for beginners who want to learn ASP.NET MVC.

The learning path is to demonstrate that it is still possible to develop great web applications from scratch, providing that we use the proper tools and have the patience to understand how we can blend them together to achieve what we want. Luckily enough, the insane amount of additional work required nowadays can be mitigated by a lot of powerful frameworks that make their way through the development scene: we're talking of serverside runtime environments like ASP.NET and Node.js and also of many client-side frameworks such as Bootstrap, jQuery, React, AngularJS and Angular 2. We chose to focus on two of them – ASP.NET Core to cover the server-side aspects, Angular 2 to deal with the client-side not only for their stunning set of features and elegant design, but also because they happen to have something revolutionary in common: they are both a complete rewrite of their massively-popular previous installments, who played a leading role in their respective field.

It will make you realize the importance for web applications. We will introduce performance improvements along with the trade-offs that they entail. We will strike a balance between premature optimization and inefficient code by taking a scientific and evidence-based approach, focusing on the big problems and avoiding changes that have little impact. This book addresses web application performance-improvement techniques from both a general standpoint (HTTP, HTTPS, HTTP/2, TCP/IP, database access, compression, I/O, asset optimization, caching, message queuing, and other concerns) and from a C#, ASP.NET Core, and .NET Core perspective. This includes delving into the details of the latest frameworks and demonstrating software design patterns that improve performance. We will highlight common performance pitfalls, which can often occur unnoticed on developer workstations, along with strategies to detect and resolve these issues early. By understanding and addressing challenges upfront, you can avoid nasty surprises when it comes to deployment time.

What this learning path covers

[Module 1](#), *Learning ASP.NET Core MVC Programming*, introduces you to fundamentals of ASP.NET and how it fits in the ASP.Net ecosystem. Further, it helps you to set up the development environment. You will learn what an action method and a Controller does from the perspective of an overall ASP.NET MVC application. It explains various basic programming constructs(conditionals, loops and so on) with the examples using Razor view engine. You will also learn role of models and how to use javaScript and JQuery libraries for validation. Further, you will learn routing module and deployment of ASP.NET Core application. You will use HTTP-based services and learn how to implement them using the Web API. It explains the approaches to analyzing of performance and measures for improvement in various layers of your application and the security aspects of your application .

[Module 2](#), *ASP.NET Core and Angular 2*, introduces the ASP.NET Core and Angular 2 frameworks. The reader will learn how to handle the HTTP request-response cycle and also how to configure the improved routing logic built upon the .NET Core pipeline. We will also explain how the client-side routing works in a SPA and what we need do to implement a viable routing pattern for our project using the PathLocationStrategy. You will learn to build proper, DBMS-based Data Model using Entity Framework Core and update the application code to make full usage of the EF Core entities defined in the data model. You will learn applying styles and the most relevant auth-related concepts for building a web application. It further expands the authentication and authorization topics introducing OpenIddict. It discusses account-related features such as user registration and edit account mechanism. Further, it describes the most common tasks to publish a potentially shippable web application onto a production server.

[Module 3](#), *ASP.NET Core 1.0 High Performance*, shows you why you need to care about the performance of your software. We will highlight a few ways of manually monitoring performance and some helpful tools that you can use to measure statistics. It looks at some of the most frequent performance mistake and show you how to fix simple issues across a range of different application areas. It digs into the networking layer that underpins all web applications. You will learn how to batch your requests and optimizing database usage with aggregates or sampling, aiming to reduce the data and time required. You will learn caching and message queuing along with discussing the negatives of performance- enhancing tools. It will explain how to measure performance from an automation and Continuous Integration (CI) perspective. Lastly, it briefly sums up the lessons of the module and then has a look at some advanced topics that you may like to read more about.

What you need for this learning path

To start programming the ASP.NET MVC applications, you will need Visual Studio Community 2015 IDE. This is a fully featured IDE available for building desktops and web applications. You will also need various packages and frameworks, such as NuGet, Bootstrap, and project.json, the installation and configuration of which will be explained in the course.

You should acquire :

1. Windows 7 SP1 or newer, up to and including Windows 10.
2. Visual Studio 2015 with Update 3 (or newer): any version will work, including the freely available Community Edition.
3. Microsoft SQL Server 2014 (o newer) for Chapter 10, Finalization and Deployment only: any version will work, including the freely available Express Edition.
4. Windows Server 2008 R2 (or newer) for Chapter 10, Finalization and Deployment only.
5. All ASP.NET, Angular 2, JavaScript and CSS packages used throughout the book are open-source and freely available for download using Visual Studio package managers such as NuGet, MyGet, NPM and Bower.

You will also need a development environment to follow the code examples in this module, either Visual Studio Community 2015 or Visual Studio Code if you're not on Windows. You can also use your text editor of choice and the .NET command line tool. If you use Visual Studio, then you should also install the .NET Core SDK and tooling and the latest NuGet extension. For some of the chapters, you will also need SQL Server 2014 Express. You can use 2016 too, particularly if you are on Linux. However, you can also use Azure and run against a cloud database.

There are other tools that we will cover, but we will introduce these as they are used. The detailed software/hardware list is uploaded along with the code files.

Who this learning path is for

This book is for developers who want to learn to build web applications using ASP.NET Core, developers who want to make a career building web applications using Microsoft technology, and developers who are working on other web frameworks and want to learn ASP.NET Core.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—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 course'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 course, 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 course from your account at
http://www.packtpub.com. If you purchased this course 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 course in the Search box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the
course's webpage at the Packt Publishing website. This page can be accessed by
entering the course's name in the Search box. Please note that you need to be logged
in to your Packt account.

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 course is also hosted on GitHub at
https://github.com/PacktPublishing/ASP-NET-Core-Cloud-ready-Enterprise-Web-Application-Development. We also have other code bundles from our rich catalog of
books, videos, and courses 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 courses—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 course. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your course, 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 course 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 course, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part 1. Module 1

Learning ASP.NET Core MVC Programming

Learn the fundamentals of ASP.NET MVC and build realworld web applications using ASP.NET Core MVC

Chapter 1. Introduction to ASP.NET Core

ASP.NET Core, the latest version of ASP.NET MVC from Microsoft, is the server-side web application development framework which helps you to build web applications effectively. This runs on top of the ASP.NET 5 platform, which enables your application to be run on a wide variety of platforms, including Linux and Mac OS X. This opens up heaps of opportunities and it is exciting to be a .NET developer in these times.

In this chapter, you'll learn about the following topics:

- Fundamental concepts about web applications—HTTP, client-side, and server-side
- Three programming models of ASP.NET—ASP.NET Web Forms, ASP.NET Web Pages, and ASP.NET MVC
- Philosophy of ASP.NET MVC
- Features of ASP.NET Core and ASP.NET 5

Before discussing the ASP.NET Core and its features, let us understand the fundamentals of web applications development. I strongly believe the principle that if you want to be an expert at something, you need to be very good at the fundamentals. It will be helpful in debugging the issues and fixing them.

Having said that we are going to discuss the following key fundamentals:

- How web applications work, and a bit about HTTP
- Client-side and server-side
- HTTP methods

Just three key concepts. No big deal!

How web applications work

All web applications, irrespective of whether they are built using ASP.NET MVC, Ruby on Rails, or any other new shiny technology, work on the HTTP protocol. Some applications use HTTPS (a secure version of HTTP), where data is encrypted before passing through the wire. But HTTPS still uses HTTP.

So what is an HTTP protocol?

HTTP stands for **Hyper Text Transfer Protocol** and is an application protocol which is designed for distributed hypermedia systems. "Hyper Text" in Hyper Text Transfer Protocol refers to the structured text that uses hyperlinks for traversing between the documents. Standards for HTTP were developed by the **Internet Engineering Task Force** (**IETF**) and the **World Wide Web Consortium**(**W3C**). The current version of HTTP is HTTP/2 and was standardized in 2015. It is supported by the majority of web browsers, such as Internet Explorer, Chrome, and Firefox.

The HTTP protocol (a protocol is nothing but a set of rules which govern the communication) is a stateless protocol that follows the request-response pattern.

Request-response pattern

Before talking about the request-response pattern, let us discuss a couple of terms: Client and server. A server is a computing resource that receives the requests from the clients and serves them. A server, typically, is a high-powered machine with huge memory to process many requests. A client is a computing resource that sends a request and receives the response. A client, typically, could be a web server or any application that sends the requests.

Coming back to the request-response pattern, when you request a resource from a server, the server responds to you with the requested resource. A resource could be anything—a web page, text file, an image , or another data format.
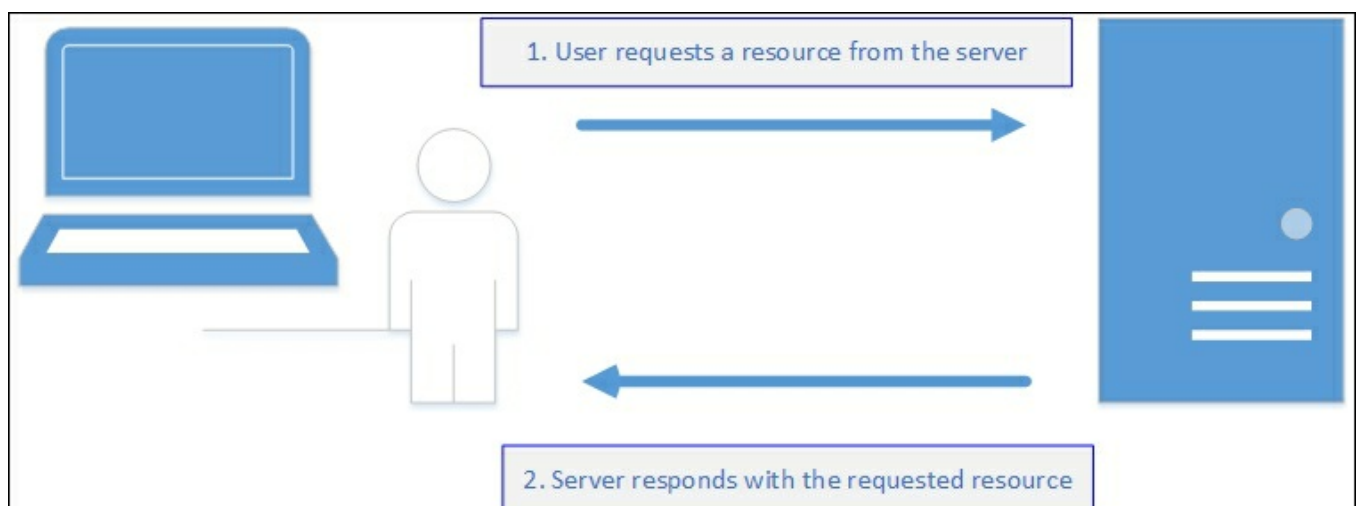
You fire a request. The server responds with the resource. This is called a **request-response pattern**.

Stateless nature of HTTP

When you request for the same resource again, the server responds to you with the requested resource again without having any knowledge of the fact that the same was requested and served earlier. The HTTP protocol inherently does not have any knowledge of the state knowledge of any of the previous requests received and served. There are several mechanisms available that maintain the state, but the HTTP protocol by itself does not maintain the state. We will explain the mechanisms to maintain the state later.

Let me explain to you about the statelessness and the request-response pattern to you with a simple practical example:

1. You type the following URL: https://en.wikipedia.org/wiki/ASP.NET_MVC. This is a Wikipedia web page about ASP.NET MVC.
2. From the preceding URL, the browser fires a request to the Wikipedia server.
3. The web server at Wikipedia serves you the ASP.NET MVC web page.
4. Your browser receives that web page and presents it.
5. You request the same page again by typing the same URL again (https://en.wikipedia.org/wiki/ASP.NET_MVC) and press *Enter*.
6. The browser again fires the request to the Wikipedia server.
7. Wikipedia serves you the same ASP.NET MVC web page without being aware of the fact that the same resource was requested previously from the same resource.

Note

As mentioned earlier, there are several mechanisms to maintain the state. Let us assume, for the time being, that no such mechanism is implemented here. I know that I am being too simplistic here, but this explains the point.

Client-side and server-side

It is necessary to understand the client-side and server-side of web applications and what can be done either side. With respect to web applications, your client is the browser and your server could be the web server/application server.

The browser side is whatever that happens in your browser. It is the place where your JavaScript code runs and your HTML elements reside.

The server-side is whatever happens at the server at the other end of your computer. The request that you fire from your browser has to travel through the wire (probably across the network) to execute some server-side code and returns the appropriate response. Your browser is oblivious to the server-side technology or the language your server-side code is written in. The server-side is also the place where your C# code resides.

Let us discuss some of the facts to make things clear:

- **Fact 1**: All browsers can only understand HTML, CSS, and JavaScript, irrespective of the browser vendor.
  - You might be using Internet Explorer, Firefox, Chrome, or any other browser. Still, the fact that your browser can understand only HTML, CSS, and JavaScript holds true. It cannot understand C#. It cannot understand Java. Nor Ruby. Only HTML, CSS, and JavaScript. This is the reason why you can access the web applications, built using any technology could be accessed by the same browser.
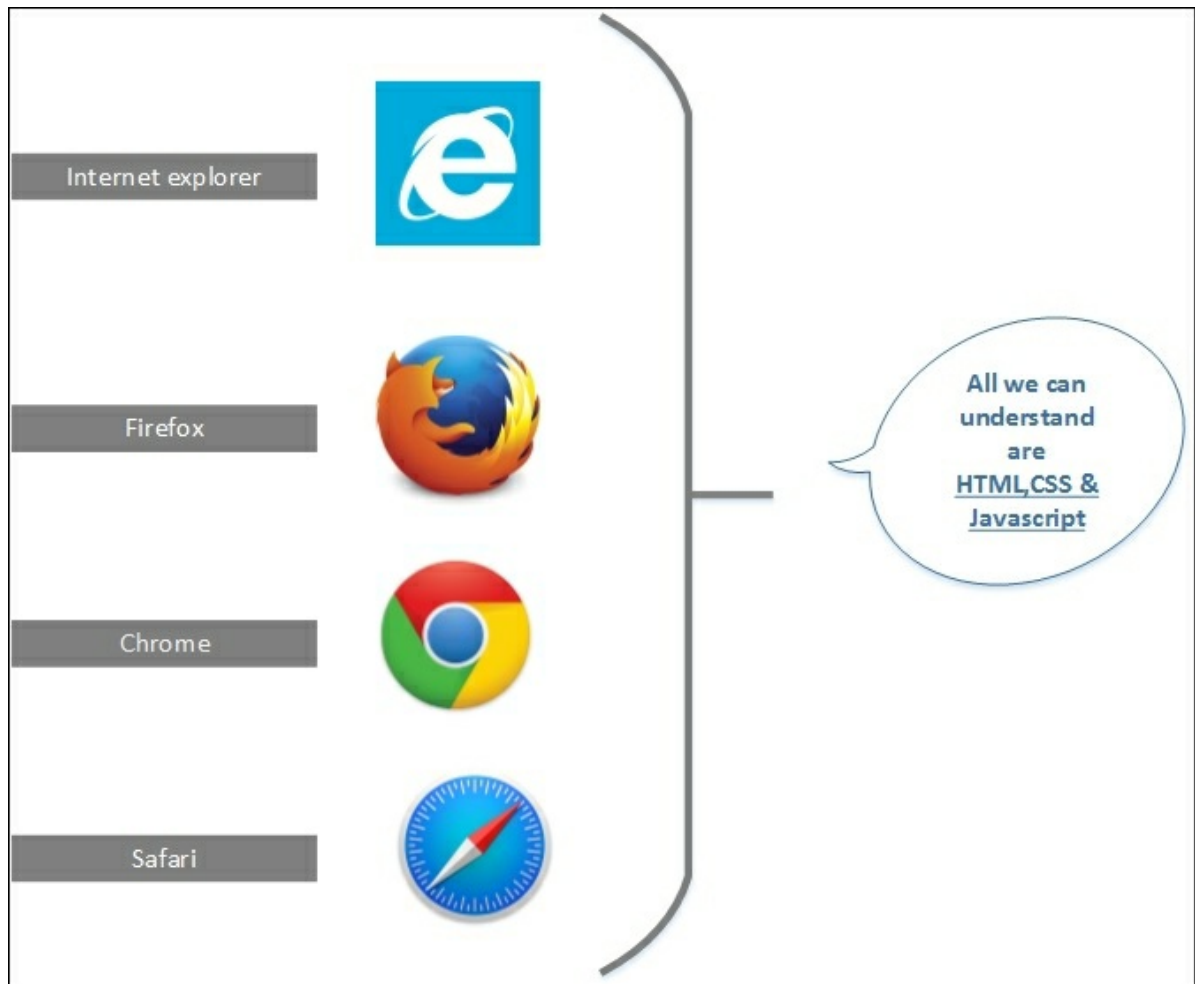
- **Fact 2**: The purpose of any web development framework is to convert your server-side code to HTML, CSS, and JavaScript.
  - This is related to the previous point. As browsers can only understand HTML, CSS, and JavaScript, all the web development technologies should convert your server-side code to HTML, CSS, and JavaScript so that your browser can understand. This is the primary purpose of any web development framework. This is true for whether you build your web applications using ASP.NET MVC, ASP.NET Web Forms, Ruby on Rails, or J2EE. Each web development framework may have a unique concept/implementation regarding how to generate the HTML, CSS, and JavaScript, and may handle features such as security performance differently. But still, each framework has to produce the HTML, because that's what your browsers understand.

HTTP methods

Even though all the requests of the HTTP protocol follow the request-response pattern, the way the requests are sent can vary from one to the next. The HTTP method defines how the request is being sent to the server.

The available methods in HTTP are `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, and `PATCH`. In most of the web applications, the `GET` and `POST` methods are widely used. In this section, we will discuss these methods. Later, we will discuss other HTTP methods on a need-to-know basis.

GET method

`GET` is a method of the HTTP protocol which is used to get a resource from the server. Requests which use the `GET` method should only retrieve the data and should not have any side effect. This means that if you fire the same `GET` request, again and again, you should get the same data, and there should not be any change in the state of the server, as a result of this GET request.

In the `GET` method, the parameters are sent as part of the request URL and therefore will be visible to the end user. The advantage of this approach is that the user can bookmark the URL and visit the page again whenever they want. An example is www.yourwebsite.com?tech=mvc6&db=sql.

We are passing a couple of parameters in the preceding `GET` request. `tech` is the first parameter with the value `mvc6` and `db` is the second parameter with the value `sql`. Assume your website takes the preceding parameters with values and searches in your database to retrieve the blog posts that talk about `mvc6` and `sql` before presenting those blog posts to the user.
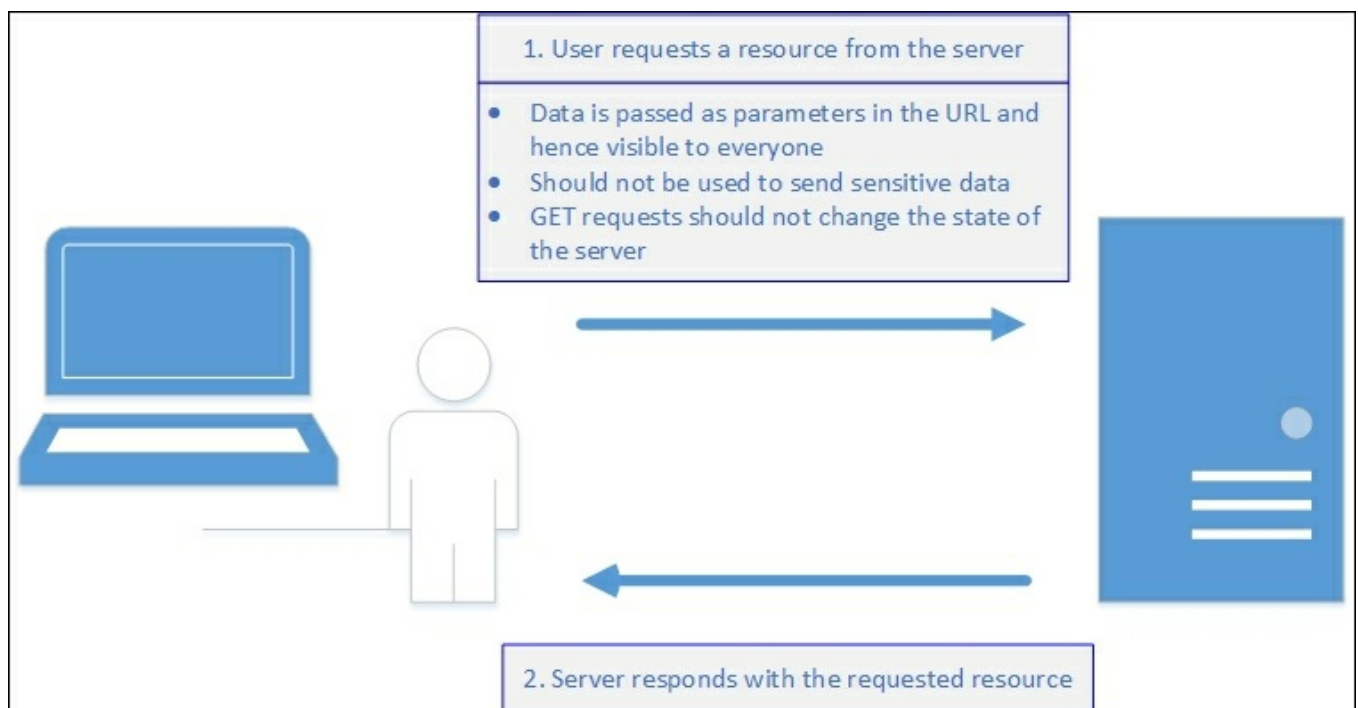
The disadvantage of the `GET` method is that, as the data is passed in clear text in the URL as parameters, it cannot be used to send the sensitive information.

Moreover, most browsers have limitations on the number of characters in the URL,

so, when using `GET` requests, we cannot send large amounts of data.

POST method

The POST request is generally used to update or create resources at the server.

Data is passed in the body of the request. This has the following implications:

- You can send sensitive information to the server, as the data is embedded in the body of the request and it will not be visible to the end user in the URL.
- As the data is not sent through the request URL, it does not take up space in the URL and therefore it has no issues with the URL length limitations.
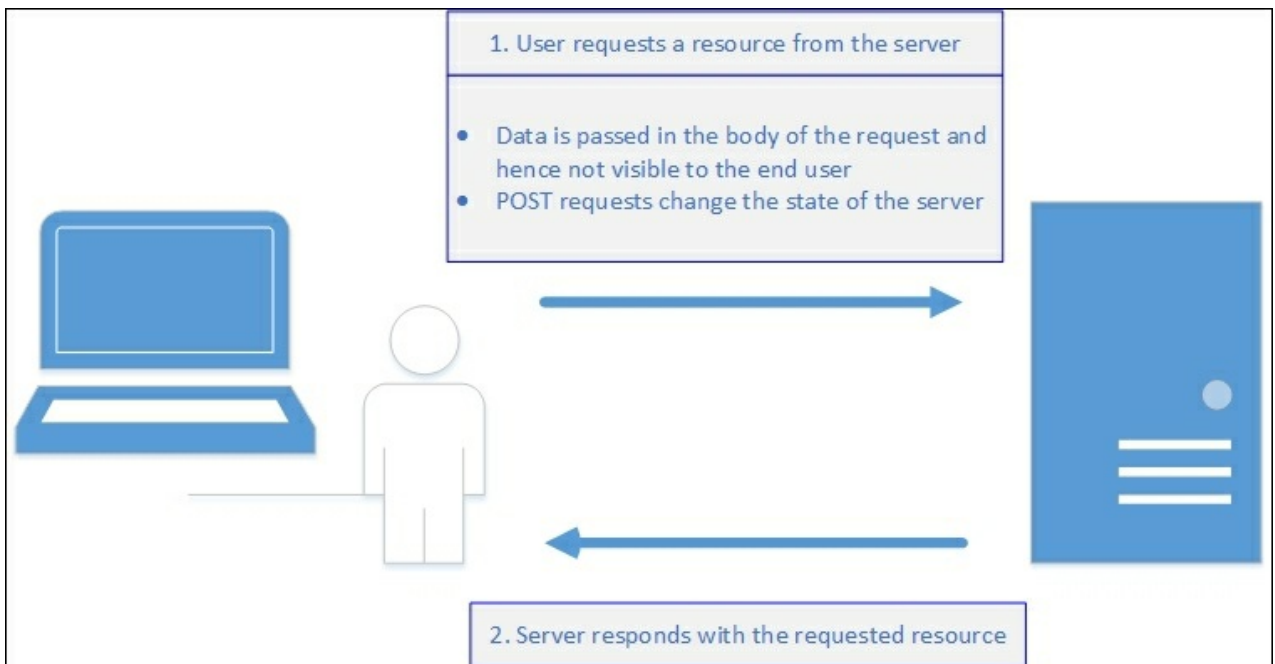
As we have covered the fundamentals, we can now proceed to discuss ASP.NET.

What is ASP.NET?

ASP.NET is a server-side web application development framework allowing developers to build web applications, websites, and web services. It was first introduced by Microsoft in early 2002, and in these 14 years, it has undergone a lot of changes.

Basically, ASP.NET has three programming models:

- ASP.NET Web Forms
- ASP.NET Web Pages
- ASP.NET MVC

Even though the end result of all of the preceding programming models is to produce the dynamic web pages effectively, the methodologies that they follow differ from each other. Let us discuss each one of these programming models to understand their principles.

ASP.NET Web Forms

Historically, when ASP.NET was first introduced, ASP.NET Web Forms was the only programming model available to programmers to develop web applications in ASP.NET.

The ASP.NET Web Forms model abstracted the web so that it can maintain the state even though the web is inherently stateless.

It also supports the event-driven programming model at the server-side. This has helped desktop application developers to have a smooth transition in moving into web application development.

Like PHP and several other web application frameworks, ASP.NET Web Forms is a file-based framework where users access the web page by means of accessing a file at the server. The server will process your request, convert all of your server-side components in that file to HTML, and send it back to the requesting client.

Each web page in ASP.NET Web Forms is represented by two files: `.aspx` and `.aspx.cs` or `.aspx.vb`. The `.aspx` file contains your front end components-all of your ASP controls and your HTML elements. The `.aspx.cs` (if you are using C# as the code-behind language) or `.aspx.vb` (if you are using Visual Basic as the code-behind programming language) contains the code for events which are happening at the web page.

This was the predominant programming model prior to the arrival of ASP.NET MVC, and this programming model is still being used to maintain the production applications that were written using this model.

ASP.NET Web Pages

ASP.NET Web Pages are primarily targeted at small web applications where the data-processing logic is written directly on the web page.

ASP.NET MVC

ASP.NET MVC is the implementation of the MVC pattern in ASP.NET. The disadvantages of ASP.NET Web Forms, such as limited control over the generation of HTML are resolved in ASP.NET MVC. As most of the modern applications are controlled by client-side JavaScript libraries/frameworks, such as **jQuery**, **KnockoutJS,** and **AngularJS**, having complete control over the generated HTML is of paramount importance.

Let us talk a bit about the Model-View-Controller pattern and how it benefits the web application development.

Model-View-Controller (MVC) pattern: This is a software architectural pattern which helps in defining the responsibility for each of the components and how they fit together in achieving the overall goal. This pattern is primarily used in building user interfaces, and is applicable in many areas including developing desktop applications and web applications. But I am going to explain the MVC pattern from the context of web development.

Primarily, the MVC pattern has three components:

- **Model**: This component represents your domain data. Please note that this is not your database. This model component can talk to your database, but the model only represents your domain data. For example, if you are building an e-commerce web application, the model component may contain classes such as Product, Supplier, and Inventory.
- **View**: This component is responsible for what to present to the user. Usually, this component would contain your HTML and CSS files. This may also include the layout information governing how your web application looks to the end user.
- **Controller**: As the name implies, the controller is responsible for interacting with different components. It receives the request (through the routing module), talks to the model, and sends the appropriate view to the user.

1. User request the web page by typing the URL in the browser of his computer
2. Based on the URL, appropriate controller is selected and request is forwarded to that controller
3. Then, Controller talks to the Model to query/update the data
4. Controller passes the data to the view and view is responsible for displaying the data
5. View is returned to the requested user
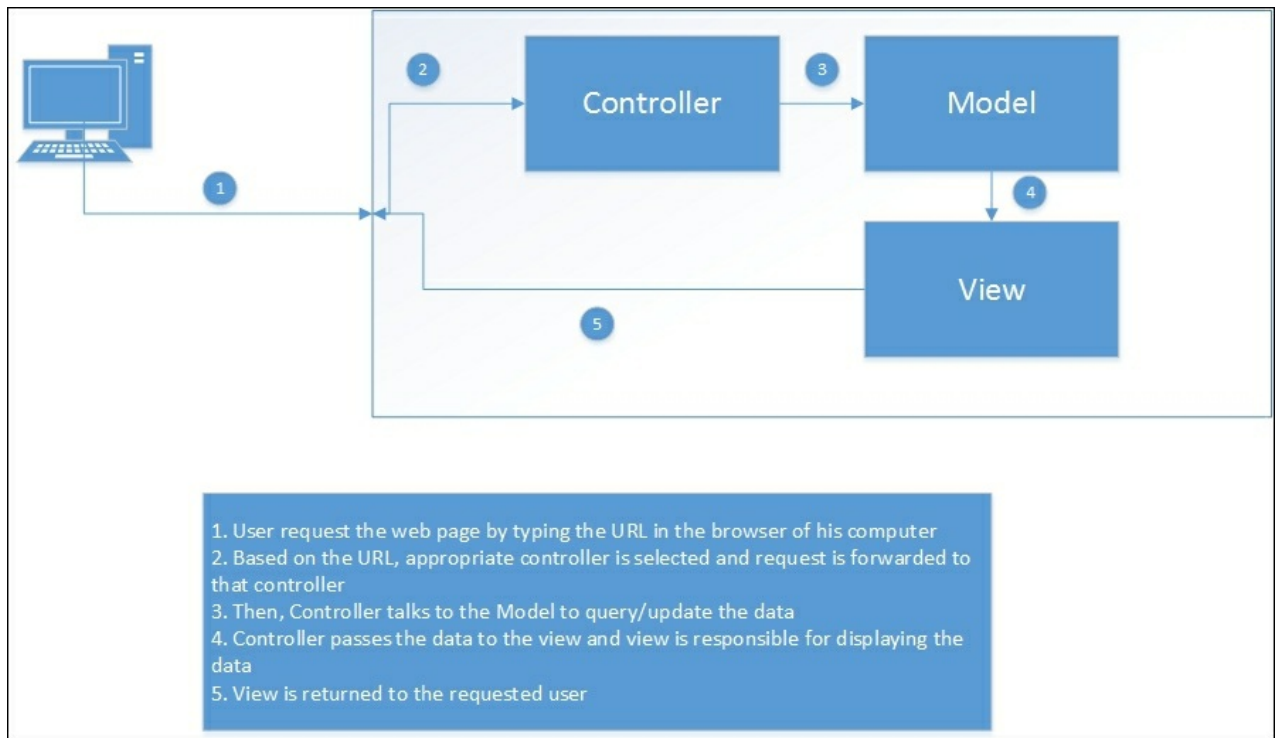
This separation of responsibilities brings great flexibility to the web application development, allowing each area to be managed separately and independently.

Features of ASP.NET MVC

ASP.NET MVC is an opinionated application development framework that prefers some functionality to be handled in a certain unique way. Let us discuss each of the features of ASP.NET MVC, along with the benefits they bring to the table.

Convention over configuration

This is a design methodology that significantly reduces the number of decisions while developing the application, and thus making it simpler.

If you have built any application using any technology, you might be using some kind of XML file where you have to configure everything in it. Even for the simpler straightforward things, we might have to configure the things over there.

ASP.NET MVC embraces *convention over configuration* completely. It is the philosophy where you can be certain of how it is going to work without ever configuring same.

Let me give you a simple example. All Controller code resides in the `Controller` folder, and Views have a separate folder for each of the Controllers. Whenever a request comes, ASP.NET MVC knows where to find the Controller and its associated View without any configuration. This methodology results in less configuration and less time in debugging.

Separation of concerns

As discussed earlier, ASP.NET MVC has three major components—Model, Controller, and Views. This clearly separates the responsibilities so that the UI designer or UI developer can work on the View while backend developers can work on the Model to build a data domain for the application or to talk to the database. As the duties of each of the components are clearly defined and separated, the work can be done in parallel.

Control over the generated HTML

If you have any experience in building an ASP.NET Web Forms application, you might have used ASP controls such as `asp:textbox`. Even though these controls have a lot of benefits, they have their cons as well. Developers cannot have complete control over the generated HTML when using these controls. Of course, you can set some properties in ASP control which in turn set some attributes in your generated HTML. But complete control is not possible. ASP.NET MVC HTML helpers and Tag helpers in ASP.NET Core provide better control over the generated HTML.

Better support for unit testing

As each of the components is separated and compartmentalized, creating the unit test cases becomes easier to achieve:

- **Unified MVC and Web API Controller in ASP.NET Core**: In earlier versions of ASP.NET MVC, different controllers were used for MVC (`System.Web.MVC.Controller`) and Web API (`System.Web.Http.ApiController`). In ASP.NET Core, there is only one base controller that supports creating both MVC controllers and Web API controllers. With respect to routing, all the controllers use the same routes. Of course, you can use convention-based routing or attribute-based routing depending on your needs.
- **Note about Web API**: Web API is the Microsoft technology for building web services over the HTTP protocol. HTTP is not only limited to serving web pages. Web API could be used for building API services and data. The advantage of this approach is that the services which are built using Web API could be consumed by a wide range of clients such as, browsers, mobile applications, and desktop applications.

The code for the earlier version of ASP.NET MVC (till ASP.NET MVC 5) is as follows:

```
publicclassValuesController : ApiController
{
  // GET api/values
  publicIEnumerable<string>Get()
  {
    returnnewstring[] { "value1","value2"};
  }
}
Code for ASP.NET Core:
publicclassValuesController:Controller
{
  //GET api/values
  [HttpGet]
  publicIEnumerable<string>Get()
  {
    returnnewstring[] { "value1","value2"};
  }
}
```

ASP.NET 5

ASP.NET 5 is the latest framework from Microsoft for building modern cloud-based applications using .NET. It is a cross-platform framework so that you can run your applications built on ASP.NET 5 on any platform, such as Linux or Mac OS X and also on Microsoft Windows, obviously. ASP.NET 5 is open source, and the complete source code is available on GitHub at https://github.com/aspnet/home .

The latest version of ASP.NET MVC, ASP.NET Core—runs on the ASP.NET 5 platform.

Features of ASP.NET 5

- **Cross-platform support**: Applications that are built on top of ASP.NET 5 can run on any platform where ASP.NET 5 is installed. This means that the applications that you build on ASP.NET 5 can run on Apple OS X and Linux machines. Deploying ASP.NET Core on a Linux machine will be explained in a later chapter.
- **Better support for client-side development**: ASP.NET 5 is designed to work seamlessly with a range of client-side frameworks, such as **AngularJs**, **Knockout**, **Bootstrap**, and **React.js**.

Summary

In this chapter, we have learned the basics of web development, including what constitutes the server-side and client-side. We have even discussed the features of ASP.NET Core and ASP.NET 5.

Chapter 2. Setting Up the Environment

In any development project, it is vital to set up the right kind of development environment so that you can concentrate on the developing the solution rather than solving environment issues or configuration problems. With respect to .NET, Visual Studio is the defacto standard **IDE (Integrated Development Environment)** for building web applications in .NET.

In this chapter, you'll be learning about the following topics:

- Purpose of IDE
- Different offerings of Visual Studio
- Installation of Visual Studio Community 2015
- Creating your first ASP.NET MVC 5 project and project structure

Purpose of IDE

First of all, let us see why we need an IDE, when you can type the code in Notepad, compile it, and execute it.

When you develop a web application, you might need the following things to be productive:

- **Code editor**: This is the text editor where you type your code. Your code editor should be able to recognize different constructs such as the `if` condition, `for` loop of your programming language. In Visual Studio, all of your keywords would be highlighted in blue color.
- **Intellisense**: Intellisense is a context aware code-completion feature available in most modern IDEs including Visual Studio. One such example is when you type a dot after an object; this *Intellisense* feature lists out all the methods available on the object. This helps the developers to write code faster and easier.
- **Build/Publish**: It would be helpful if you could build or publish the application using a single click or single command. Visual Studio provides several options out-of-the-box to build a separate project or to build the complete solution in a single click. This makes the build and deployment of your application easier.
- **Templates**: Depending on the type of the application, you might have to create different folders and files along with the boilerplate code. So, it'll be very helpful if your IDE supports the creation of different kinds of template. Visual Studio generates different kinds of templates with the code for ASP.NET Web Forms, MVC, and Web API to get you up-and-running.
- **Ease of adding items**: Your IDE should allow you to add different kinds of items with ease. For example, you should be able to add an XML file without any issues. And if there is any problem with the structure of your XML file, it should be able to highlight the issue and provide information to help you to fix the issues.

Visual Studio offerings

There are different versions of Visual Studio 2015 available to satisfy the various needs of developers/organizations. Primarily, there are four versions of Visual Studio 2015:

- **Visual Studio Community**
- **Visual Studio Professional**
- **Visual Studio Enterprise**
- **Visual Studio Test Professional**

System requirements

Visual Studio can be installed on computers running Windows 7 Service Pack 1 operating system and above. You can get to know the complete list of requirements from the following URL:

https://www.visualstudio.com/en-us/downloads/visual-studio-2015-system-requirements-vs.aspx

Visual Studio Community 2015

This is a fully featured IDE available for building desktops, web applications, and cloud services. It is available free of cost for individual users.

You can download Visual Studio Community from the following URL:

https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

Throughout this book, we will be using the Visual Studio Community version for development as it is available free of cost to individual developers.

Visual Studio Professional

As the name implies, Visual Studio Professional is targeted at professional developers and contains features such as **Code Lens** for improving your team's productivity. It also has features for greater collaboration within the team.

Visual Studio Enterprise

Visual Studio Enterprise is the full-blown version of Visual Studio with a complete set of features for collaboration, including a team foundation server, modeling, and testing.

Visual Studio Test Professional

Visual Studio Test Professional is primarily aimed for the testing team or people who are involved in the testing, which might include developers. In any software development methodology, either the waterfall model or agile developers need to execute the development suite test cases for the code they are developing.

Installing Visual Studio Community

Follow the given steps to install Visual Studio Community 2015:

1. Visit the following link to download Visual Studio Community 2015:

    https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

2. Click on the **Download Community 2015** button. Save the file in a folder where you can retrieve it easily later:
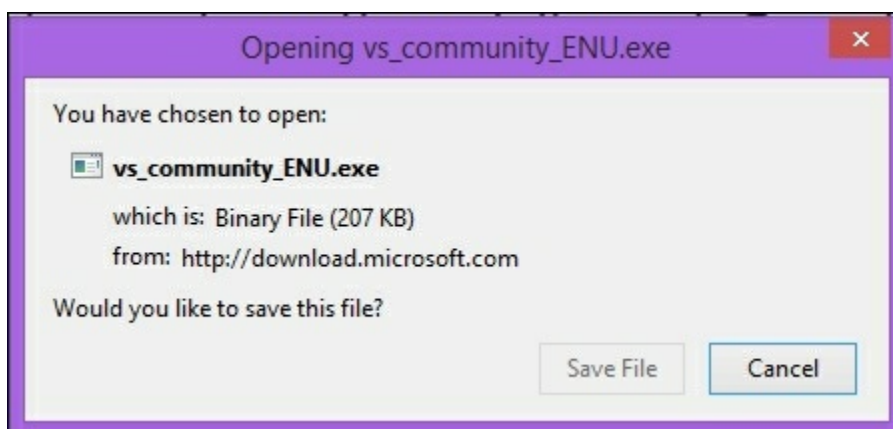
3. Run the downloaded executable file:
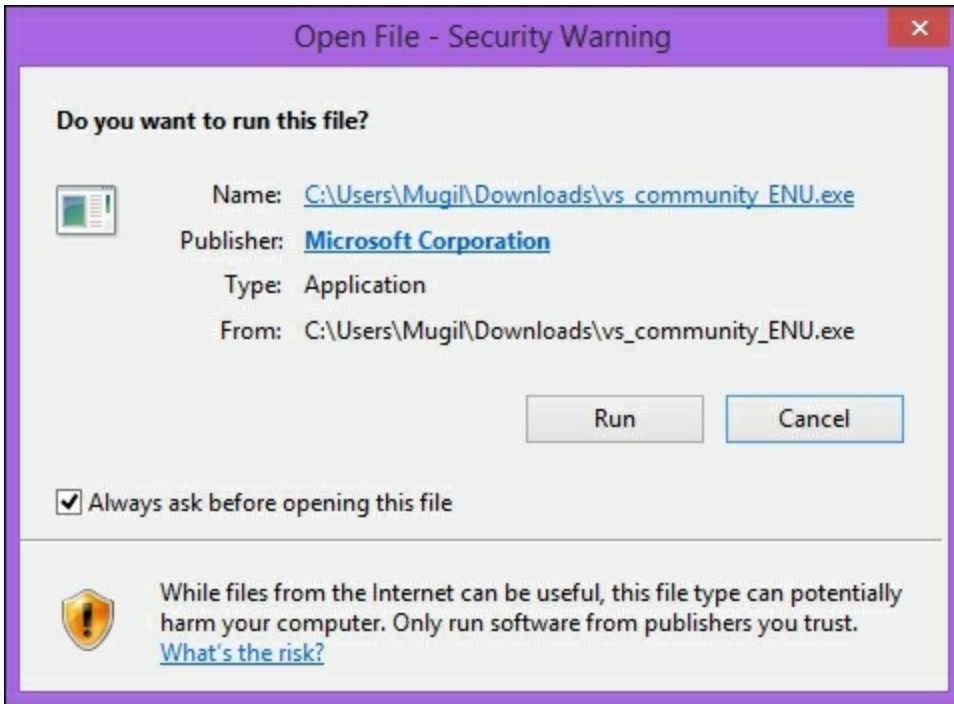
4. Click on **Run** and the following screen will appear:
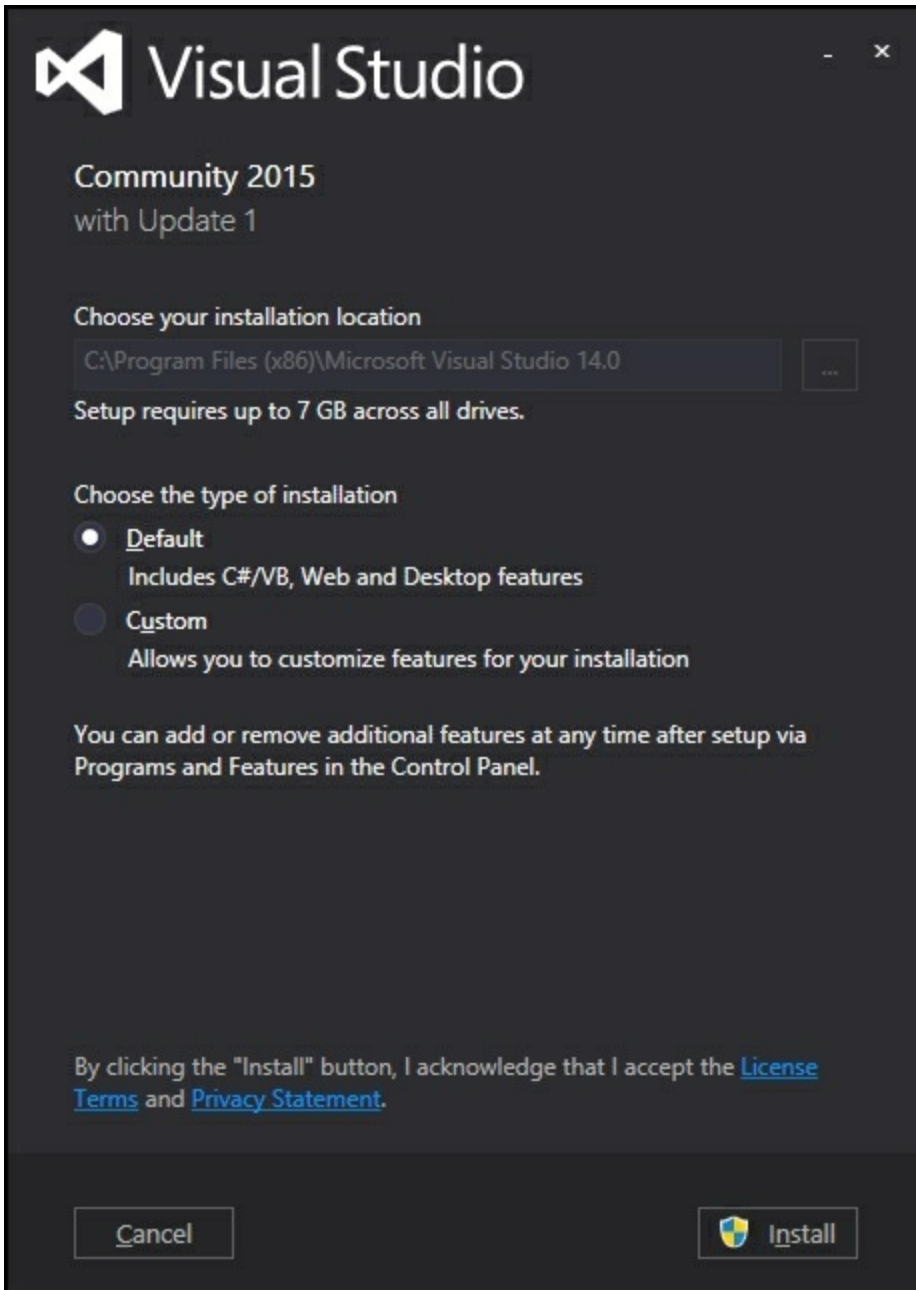
There are two types of installation—default and custom installation. The default installation installs the most commonly used features and this will cover most developer use cases of the developer. Custom installation helps you to customize the components that you want to install:

1. Click on the **Install** button after selecting the installation type.
2. Depending on your memory and processor speed, the installation will take 1 to 2 hours to install.
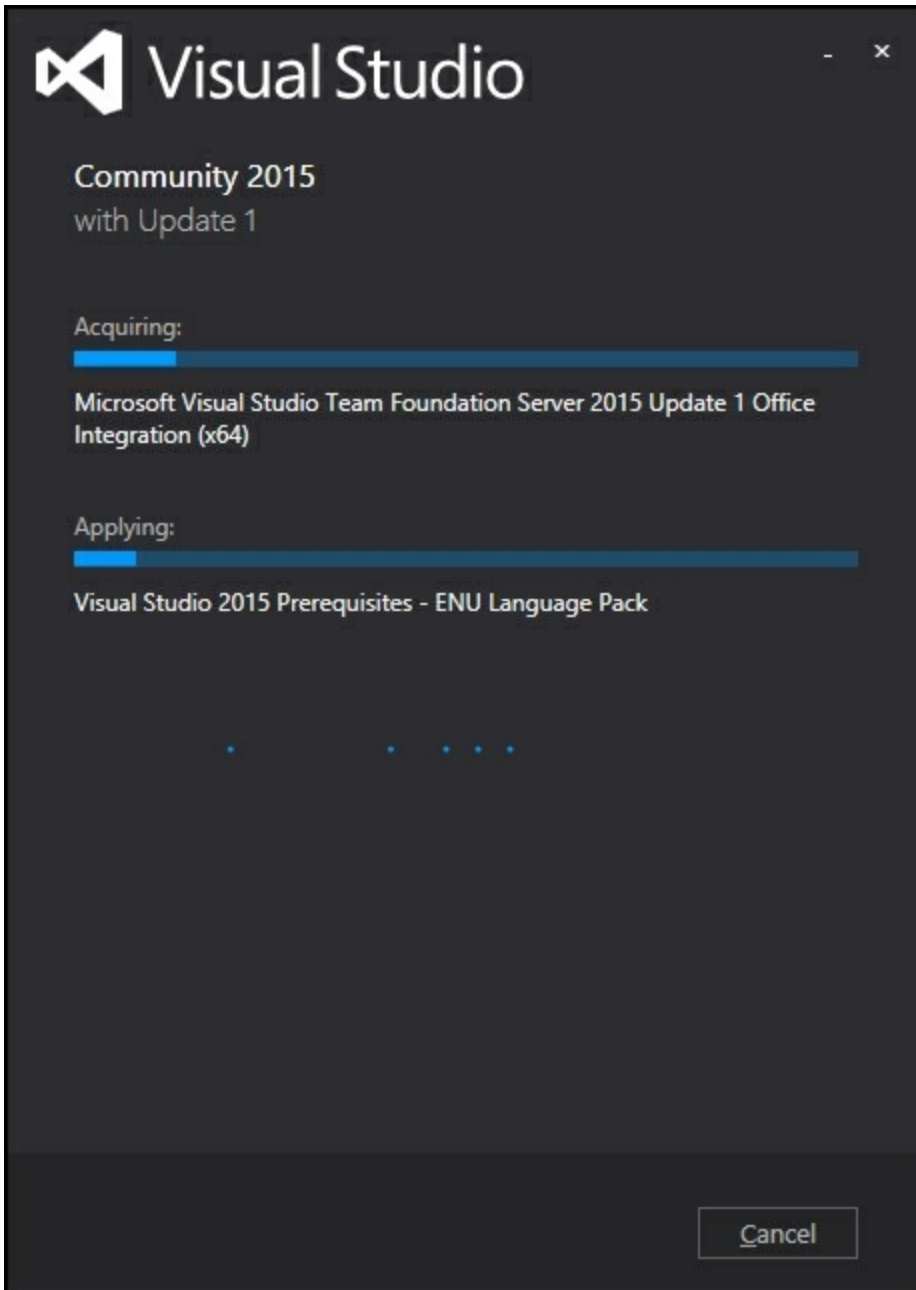
3. Once all the components are installed, you will see the following **Setup Completed** screen:

Installing ASP.NET 5

When we install the Visual Studio Community 2015 edition, ASP.NET 5 will be installed by default. As the ASP.NET Core application runs on top of ASP.NET 5, we need to install ASP.NET 5. There are a couple of ways to install ASP.NET 5:

- Get ASP.NET 5 from https://get.asp.net/
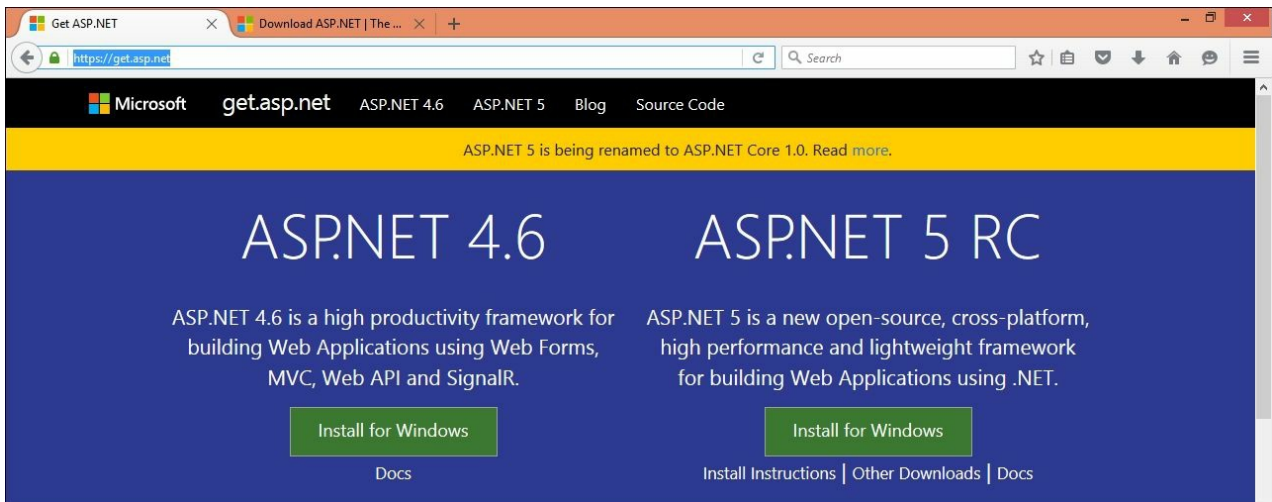
- Another option is to install from the **New Project** template in Visual Studio

This option is a bit easier as you don't need to search and install.

The following are the detailed steps:

1. Create a new project by selecting **File** | **New** | **Project** or using the shortcut *Ctrl + Shift + N*:
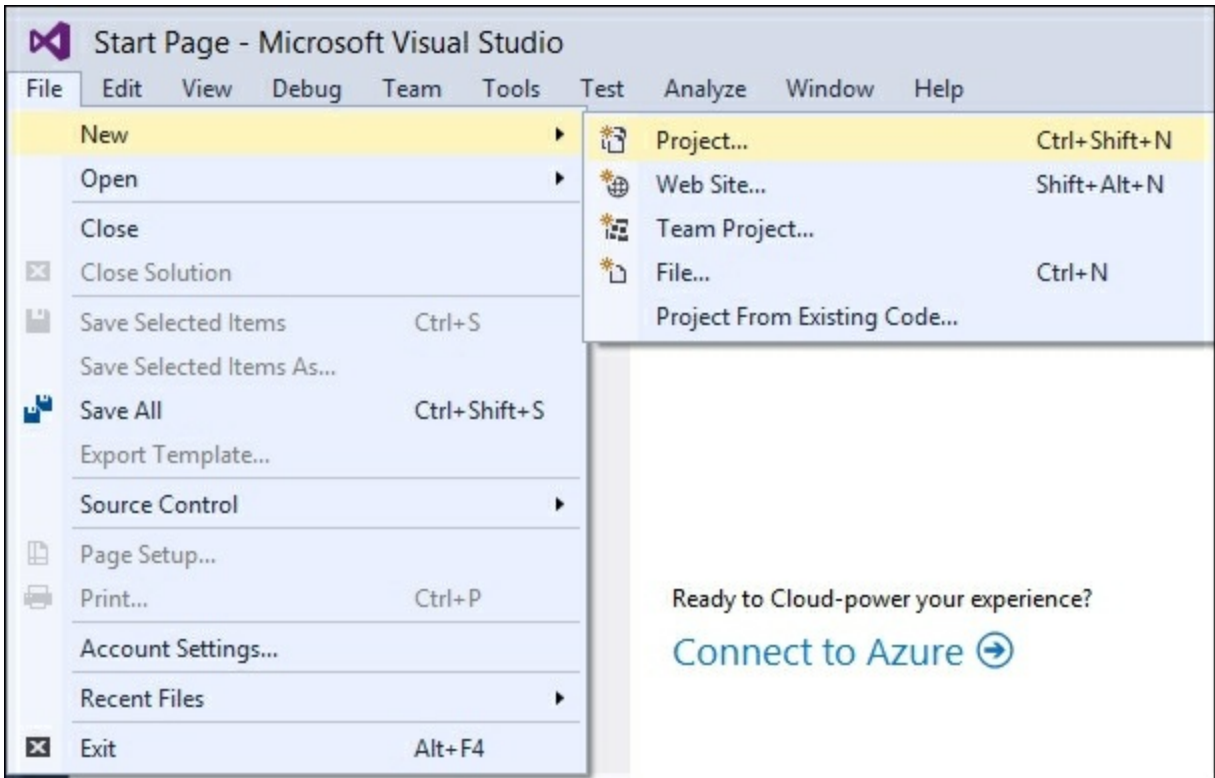
2. Select **ASP.NET Web Application** and enter the project name and click on **OK**:
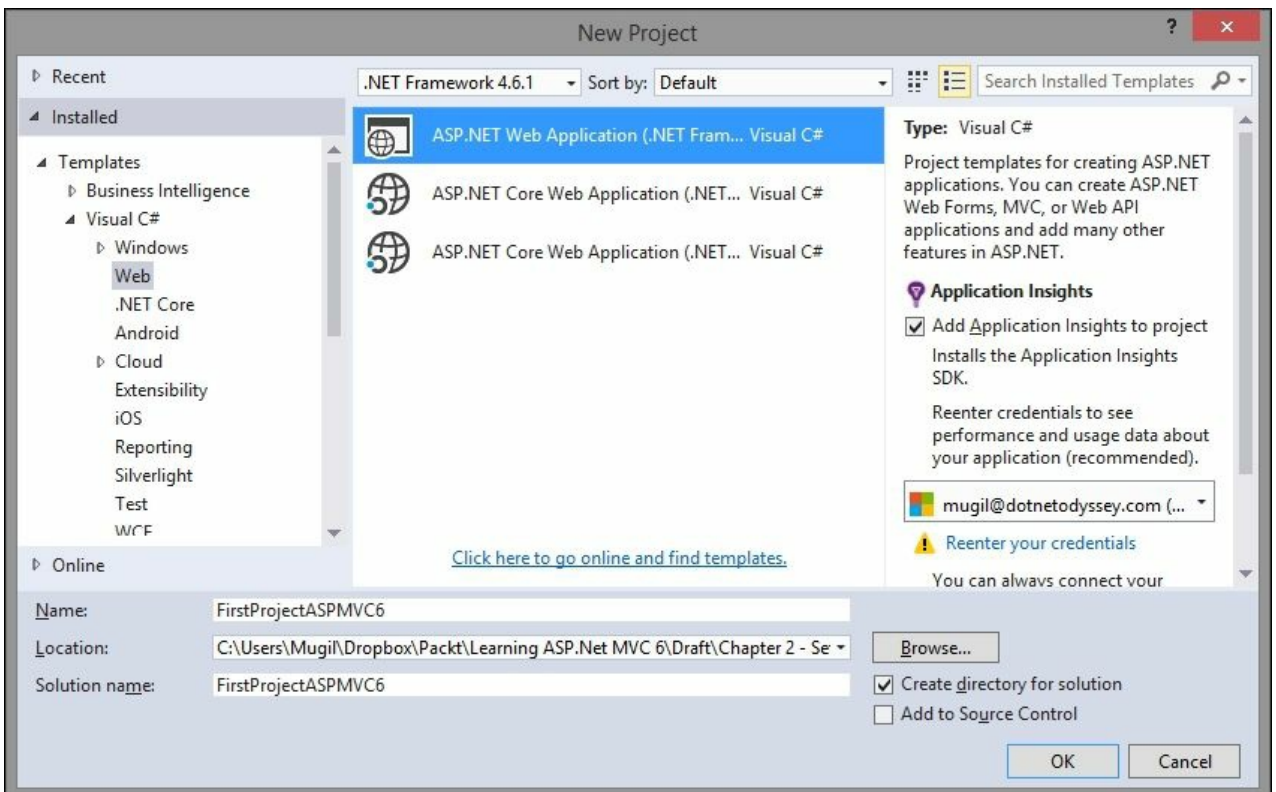
3. The following window will appear to select a template. Select the **Get ASP.NET 5 RC** option as shown in the following screenshot:
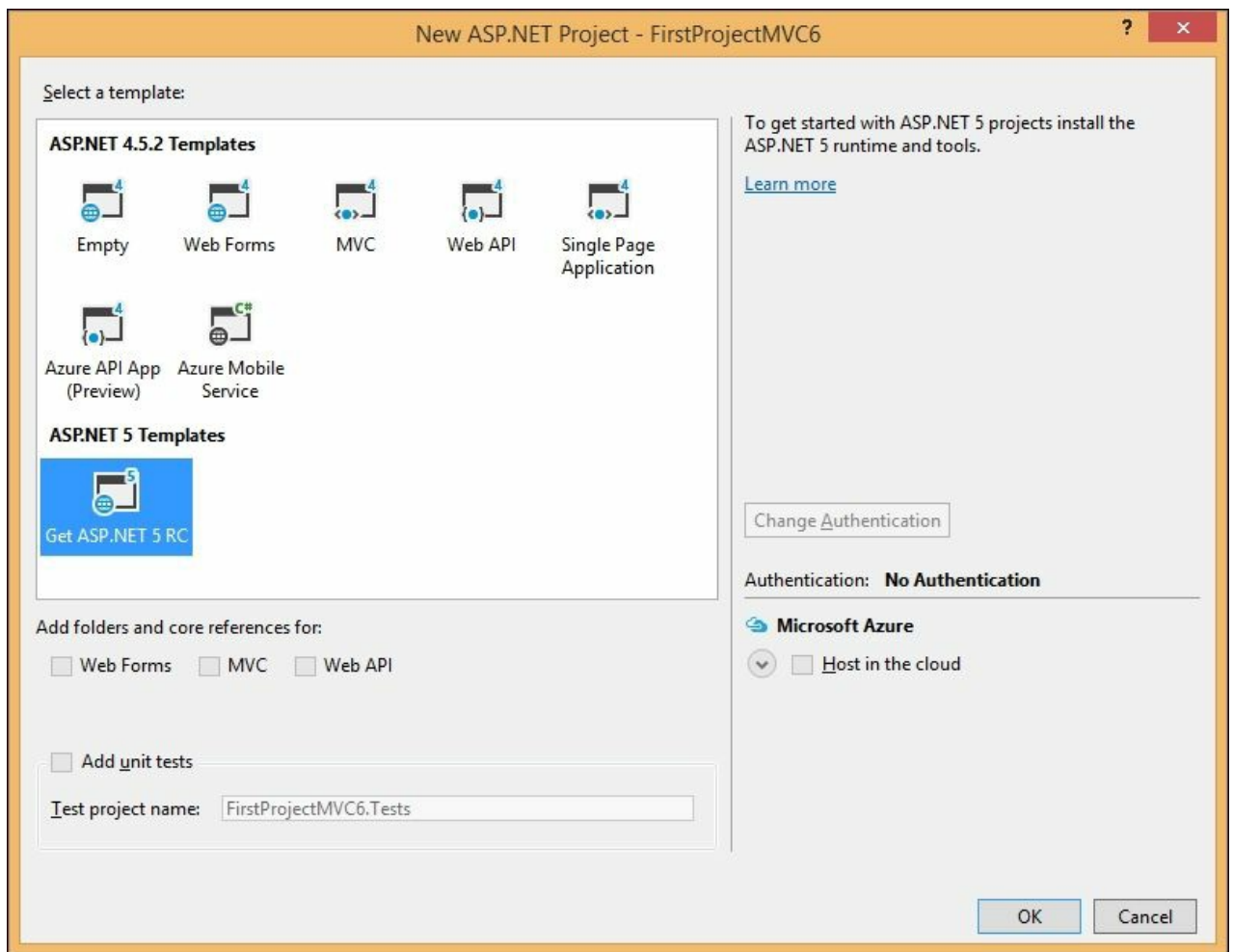
4. When you click on **OK** in the preceding screen, the following window will appear:
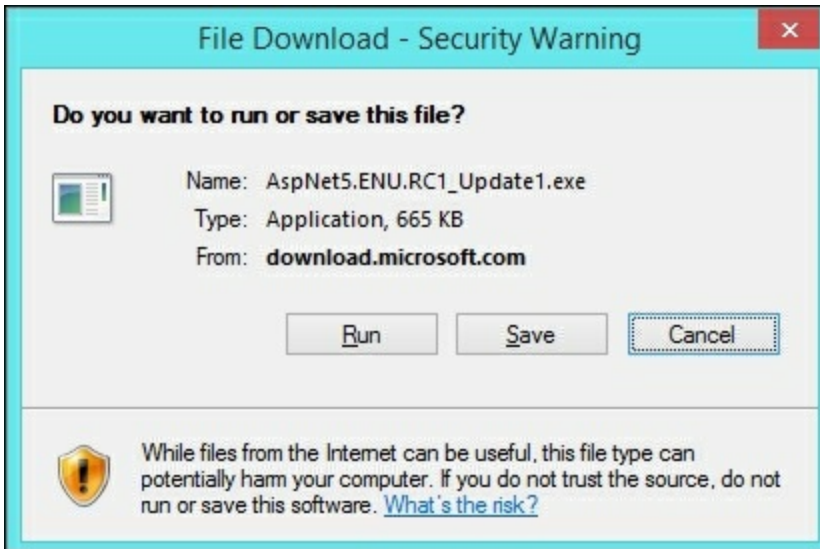
5. When you click on the **Run** or **Save** button in the preceding dialog, you will get the following screen asking for ASP.NET 5 Setup. Select the checkbox, **I agree to the license terms and conditions** and click on the **Install** button:

6. Installing of ASP.NET 5 might take a couple of hours. Once it is completed you'll get the following screen:
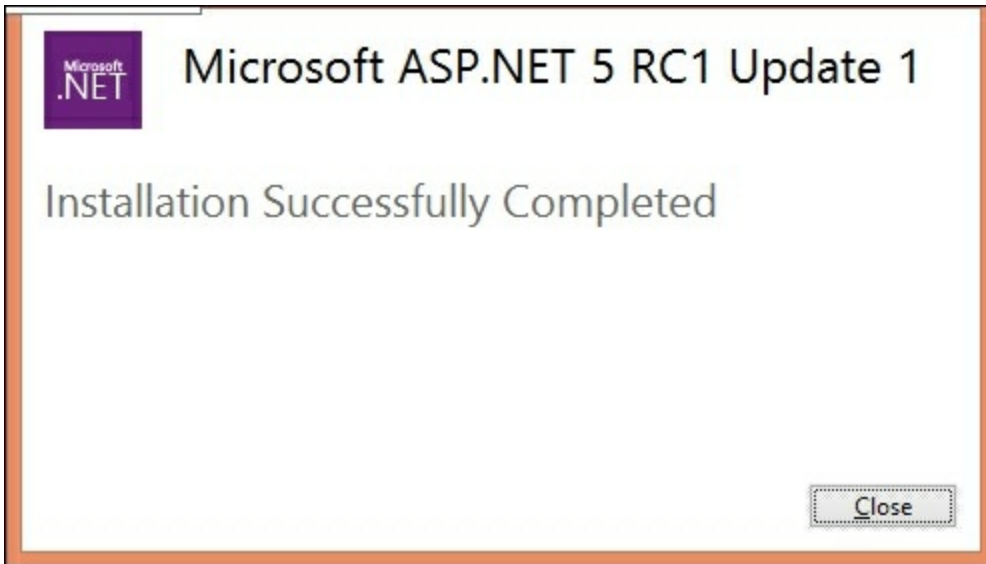
During the process of installing **ASP.NET 5 RC1 Update 1**, it might ask you to close the Visual Studio. If asked, please do so.

Project structure in ASP.NET 5 application

Once ASP.NET 5 RC1 is successfully installed, open the Visual Studio, create a new project and select the ASP.NET 5 **Web Application** as shown in the following screenshot:
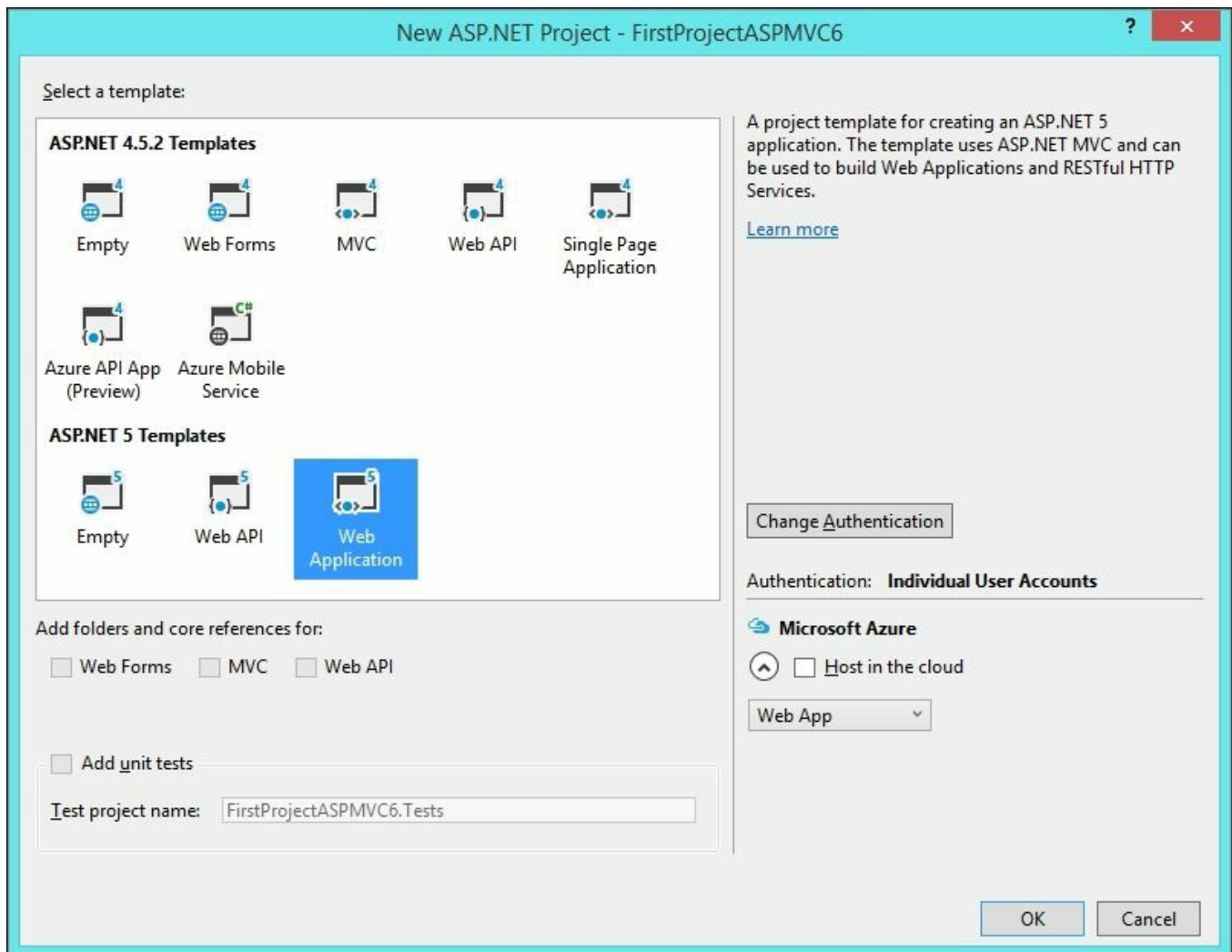
A new project will be created and the structure will be like following:
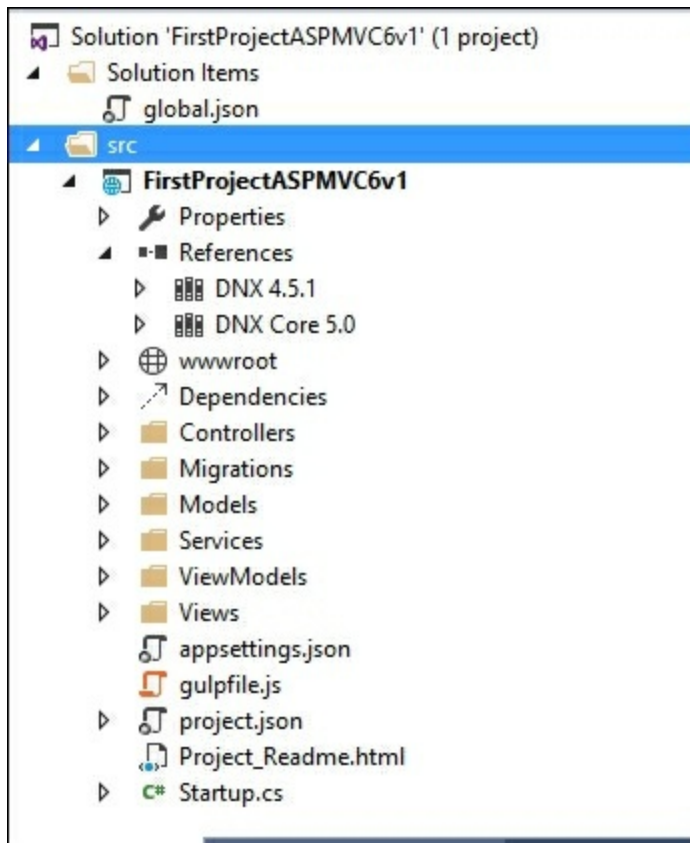
File-based project

Whenever you add a file or folder in your file system (inside the ASP.NET 5 project folder), the changes will be automatically reflected in your application.

Support for full .NET and .NET core

You might have noticed a couple of references in the preceding project: **DNX 4.5.1** and **DNX Core 5.0**. **DNX 4.5.1** provides functionalities of full-blown .NET whereas **DNX Core 5.0** supports only the core functionalities, which would be used if you are deploying the application across cross-platforms such as Apple OS X, Linux. The development and deployment of an ASP.NET Core application on a Linux machine will be explained in a later chapter.

The Project.json package

Usually, in an ASP.NET web application, we would have the assemblies as references and the list of references in a C# project file. But in an ASP.NET 5 application, we have a JSON file by name `Project.json`, which will contain all the necessary configurations with all its .NET dependencies in the form of `NuGet` packages. This makes dependency management easier. `NuGet` is a package manager, provided by Microsoft, which makes package installation and uninstallation easier. Prior to `NuGet`, all dependencies had to be installed manually. The dependencies section identifies the list of dependent packages available for the application. The frameworks section informs us about frameworks, supported by the application. The scripts section identifies the script to be executed during the build process of the application. Include and exclude properties can be used in any section to include or exclude any item.

Controllers

This folder contains all of your controller files. Controllers are responsible for handling requests, communicating models, and generating the views.

Models

All of your classes representing domain data will be present in this folder.

Views

Views are files that contain your frontend components and are presented to the end users of the application. This folder contains all of your **Razor** View files.

Migrations

Any database-related migrations will be available in this folder. Database migrations are the C# files which contain the history of any database changes done through an **Entity Framework** (an ORM framework). This will be explained in detail in a later chapter.

The wwwroot folder

This folder acts as a root folder and it is the ideal container to place all of your static files such as CSS and JavaScript files. All the files which are placed in `wwwroot` folder can be directly accessed from the path without going through the controller.

Other files

The `appsettings.json` file is the config file where you can configure application level settings. **Bower**, **npm** (**Node Package Manager**), and **gulpfile.js** are client-side technologies, supported by ASP.NET 5 applications.

Summary

In this chapter, you learned about the offerings in Visual Studio. Step-by-step instructions are provided for installing the Visual Studio Community version, freely available for individual developers. We have also discussed the new project structure of the ASP.NET 5 application and the changes when compared to the previous versions.

In the next chapter, we are going to discuss the controllers and their roles and functionalities. We'll also build a controller and associated action methods and see how they work.

Chapter 3. Controllers

As discussed in the first chapter, all web applications receive requests from the server and produce a response, that is delivered back to the end user. A Controller does the job of receiving the request and producing the output based on the input data in ASP.NET MVC.

In this chapter, you'll be learning about the following topics:

- Role of the Controller in ASP.NET MVC applications
- Routing introduction and concepts
- Creating your first ASP.NET 5 application
- Installation of the ASP.NET Core `NuGet` packages in your application
- Creation of your first Controller and `action` method, which returns a simple *Hello World*
- Adding a View and making the changes that allow your Controller to use that View
- Adding a Model and passing that Model data to your View

Role of the Controller in ASP.NET MVC applications

At the high level, the Controller orchestrates between the Model and the View, and sends the output back to the user. This is also the place where authentication is usually done through action filters. Action filters will be discussed in detail in the *Filters* section of this chapter. The following figure illustrates the high-level flow of a request (with the steps) in ASP.Net MVC and shows us how the Controller fits into the big picture:
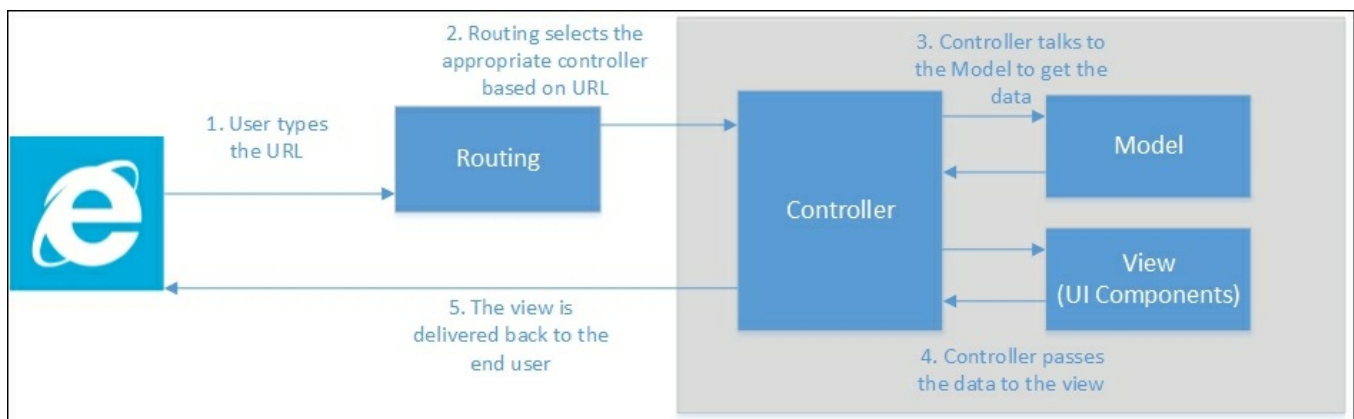
The following is the sequence of events that will happen at high level when the user is accessing the ASP.NET Core application:

1. The user types the URL in the browser.
2. Based on the pattern of the URL, the routing engine selects the appropriate Controller.
3. The Controller talks to the Model to get any relevant data through its action methods. Action methods are methods within a `controller` class.
4. The Controller then passes the data to the View to present it in a viewable format, typically as HTML elements.
5. The View is finally delivered to the user, which he would be viewing in his browser.

Before discussing the controller, let us discuss the fundamentals of routing concepts, as the routing engine only chooses the appropriate `controller` and `action` method at runtime.

Introduction to routing

The routing engine is responsible for getting the incoming request and routing that request to the appropriate Controller based on the URL pattern. We can configure the routing engine so that it can choose the appropriate controller based on the relevant information.

By convention, ASP.NET MVC follows this pattern: **Controller/Action/Id**.

If the user types the URL `http://yourwebsite.com/Hello/Greeting/1`, the routing engine selects the `Hello controller` class and `Greeting action` method within the `HelloController`, and passes the `Id` value as `1`. You can give default values to some of the parameters and make some of the parameters optional.

The following is the sample configuration:

```
The template: "{controller=Hello}/{action=Greeting}/{id?}");
```

In the preceding configuration, we are giving three instructions to the routing engine:

- Use the routing pattern `controller/action/id`.
- Use the default values `Hello` and `Greeting` for the `controller` and `action` respectively, if the values for `controller` or `action` are not supplied in the URL.
- Make the `Id` parameter optional so that the URL does not need to have this information. If the URL contains this `Id` information, it will use it. Otherwise, the `Id` information will not be passed to the `action` method.

Let us discuss how the routing engine selects the `controller` classes, `action` methods, and `Id` values for different URLs:

```
URL1:http://localhost/
Controller: Hello
Action method: Greeting
Id: no value is passed for the id parameter
```

Reasoning: The `Hello` controller is passed as the default value as per the routing configuration, as no value is passed as the Controller in the URL.

The following `action` method will be picked up by the routing handler when the preceding URL is passed:

```
public class HelloController : Controller {
  public ActionResult Greeting(int id) {
    return View();
```

```
    }
}
```

URL2: http://localhost/Hello/Greeting2
Controller: Hello
Action method: Greeting2
Id: no value is passed for the id parameter

Reasoning: The `Hello` controller will be chosen as the URL contains `Hello` as the first parameter, and the `Greeting2` action method will be chosen as the URL contains `Greeting2` as the second parameter. Please note that the default value mentioned in the configuration would be picked only when no value is present in the URL. As the `id` parameter is optional and the URL does not contain the value for `id`, no value is passed to the `id` parameter.

The following action method `Greeting2` will be picked up by the routing handler when the preceding URL is passed:

```
public class HelloController : Controller {
  public ActionResult Greeting(int id) {
    return View();
  }

  public ActionResult Greeting2(int id) {
    return View();
  }
}
```

URL3: http://localhost/Hello2/Greeting2
Controller: Hello2
Action method: Greeting2
Id: no value is passed for the id parameter

Reasoning: As `Hello2` is passed as the first parameter, the `Hello2` controller will be selected, and `Greeting2` is the action method selected since `Greeting2` is passed as the second parameter. As the `id` parameter is optional and no value is passed for the parameter `id`, no value will be passed for the `id`.

The following `action` method will be picked up by the routing handler when the preceding URL is passed:

```
public class Hello2Controller : Controller {
  public ActionResult Greeting2(int id) {
    return View();
  }
}
```
URL4: http://localhost/Hello3/Greeting2/1
Controller: Hello3

```
Action method: Greeting2
Id: 1
```

Reasoning: `Hello3` is the controller selected as it is mentioned as the first parameter, `Greeting4` is the action method, and `1` is the value passed as the `id`.

The following `action` method will be picked up the routing handler when the preceding URL is passed:

```
public class Hello3Controller : Controller {
  public ActionResult Greeting2(int id) {
    return View();
  }
}
```

We will discuss routing in detail in a later chapter.

Once the request reaches the controller, the controller will create a response by talking to the Model and may pass the data to View and the View will then be rendered to the end user.

Creating ASP.NET 5 application

It's time to get our hands dirty. Let us create a simple ASP.NET 5 application. Fire up Visual Studio and follow these steps:

1. Create a project by selecting **File** | **New Project** in Visual Studio. The first option is for creating an earlier version of the ASP.NET Web application. The second option is for creating the ASP.NET Core application using the .NET Core framework. The third option is for creating the ASP.NET Core application using the .NET framework. The difference between the second and third option is that the .NET framework supports all the functionalities of existing .NET frameworks whereas .NET Core supports only the core functionalities. The advantage of using the .NET core library is that it can be deployed on any platform.
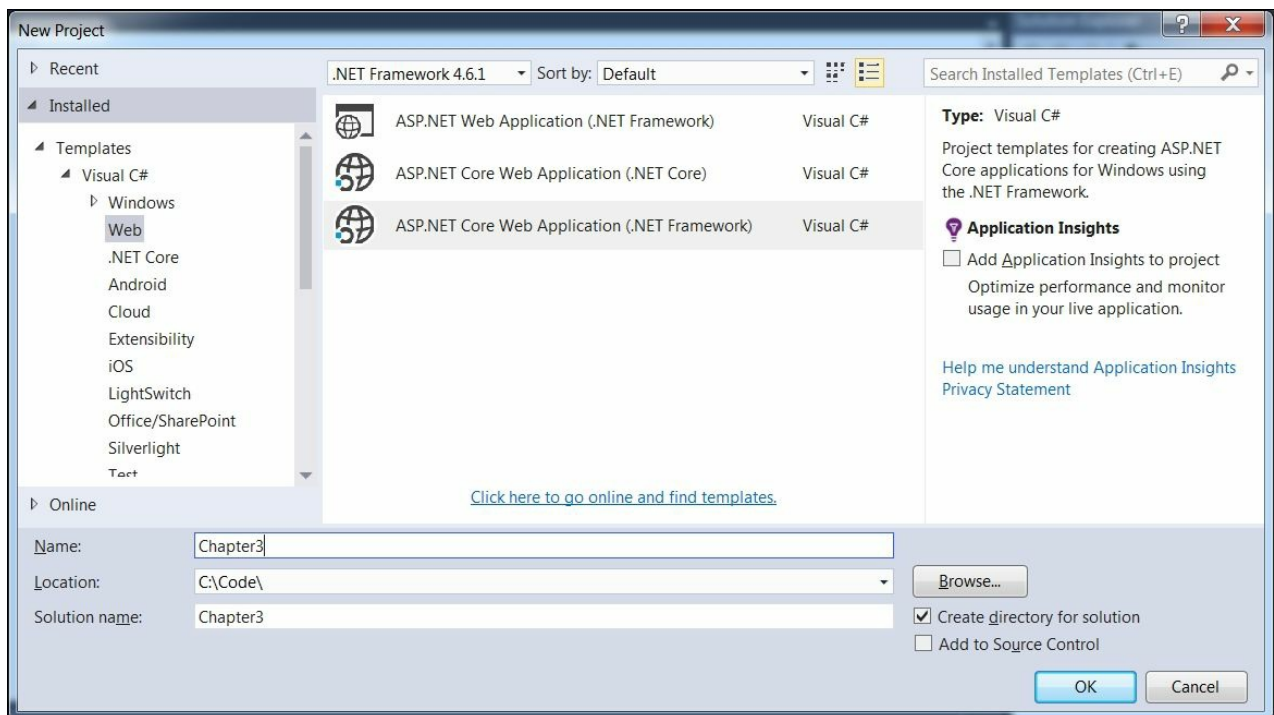
2. Select the **Empty** template from the list of ASP.NET 5 templates. The second option is for creating the Web API application (for building the HTTP-based services) and the third option is for creating a web application containing some basic functionalities which you can run just from out of the box without you ever needing to write anything.
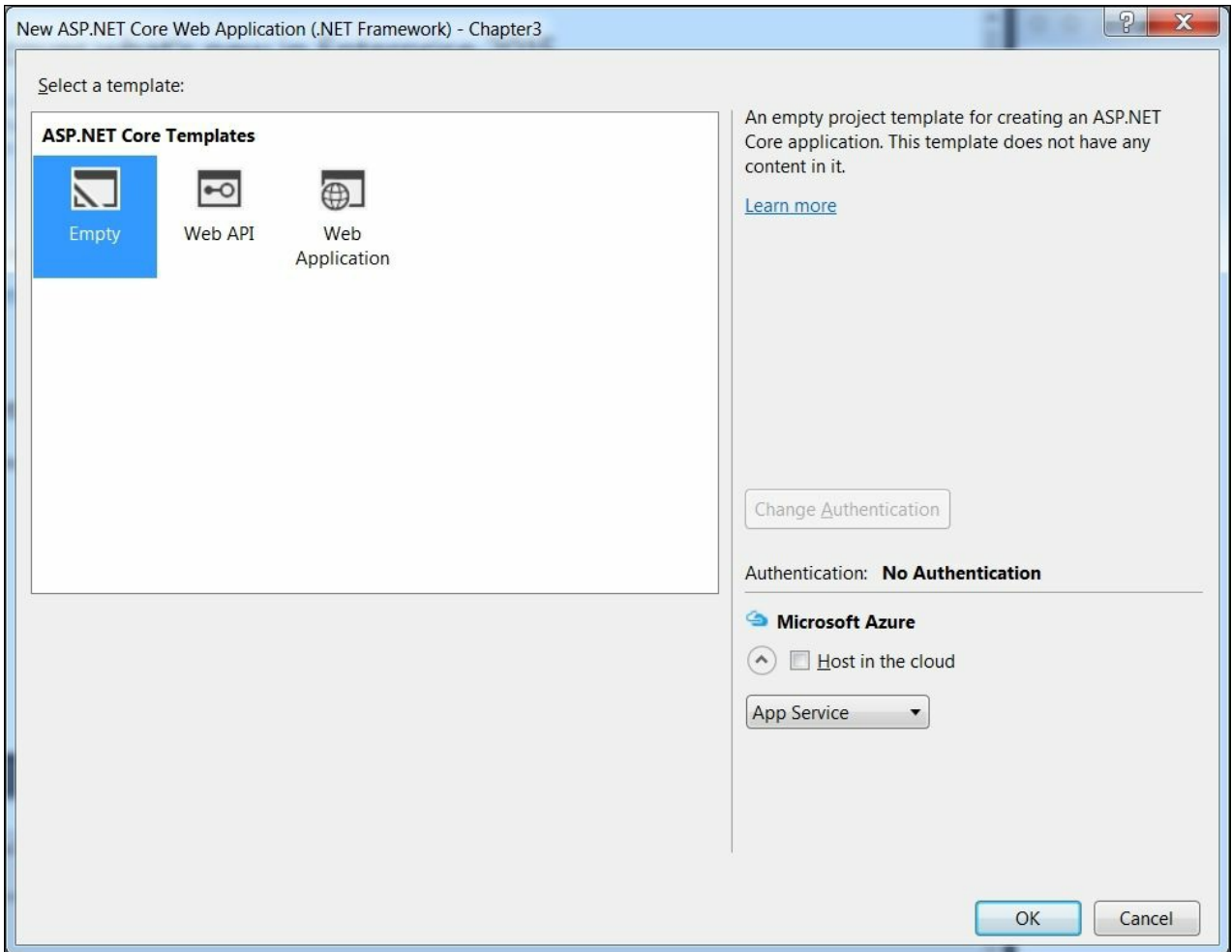
3. Once you click **OK** in the window as shown in the preceding screenshot, (after selecting the Empty template option) a solution will be created as shown in the following screenshot:
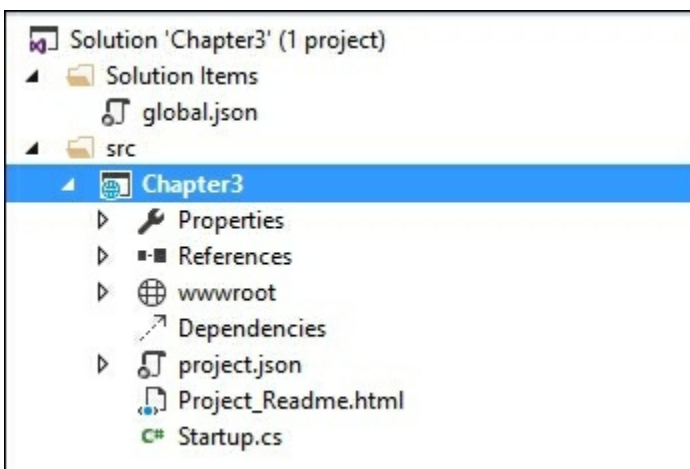
4. When you run the application (by pressing **F5**) without any changes, you'll get the simple **Hello World!** text on your screen as shown in the following screenshot:
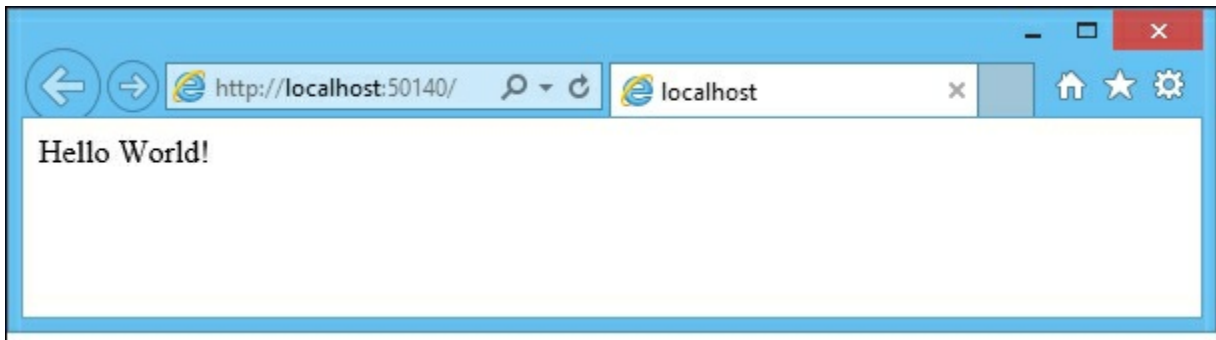
We have not done any coding in this newly created application. So, have you thought about how it displays the text **Hello World!**?

The answer lies in the `Startup.cs` file, which contains a class by the name of `Startup`. This class contains the `Main` method, which acts as the entry point for the web application. If you have used any of the previous versions of ASP.NET MVC, or even ASP.NET Web Forms, this would not be the case.

ASP.NET 5 runtime calls the `ConfigureServices` and `Configure` methods. For example, if you want to configure any service, you can add it here. Any custom configuration for your application can be added to this `Configure` method:

```
public void ConfigureServices(IServiceCollection services) {

}

// This method gets called by the runtime. Use this method to
configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app) {
  app.UseIISPlatformHandler();
  app.Run(async (context) => {
    await context.Response.WriteAsync("Hello World!");
  });

}
```

There are only a couple of statements in the `Configure` method. The first statement tells the run-time to use the `IISPlatformHandler` for handling all the incoming HTTP requests. Let us leave aside `async`, `await`, and `context` for the moment in the

second statement, which we will discuss later. In essence, the second statement tells the run-time to return `Hello World!` for all the incoming requests irrespective of the incoming URL.

When you type the URL `http://localhost:50140/Hello` in your browser, it will still return the same **Hello World!**.

This is the reason we got the **Hello World!** when we ran the application.

As we have chosen the **Empty** template while creating the ASP.NET 5 application, no component will have been installed. Even MVC wouldn't be installed by default when you select the **Empty** template as we did.

You can confirm it by opening the `project.json` file, where you can see no ASP.NET MVC is mentioned in the list of dependencies:

```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final"
},
```

So first, let us install the ASP.Net Core package for our application.

Installing the ASP.NET Core NuGet package in your application

Follow these steps to install the `NuGet` package of ASP.NET MVC:

1. Right click on the project, and select the **Manage NuGet Packages** option:
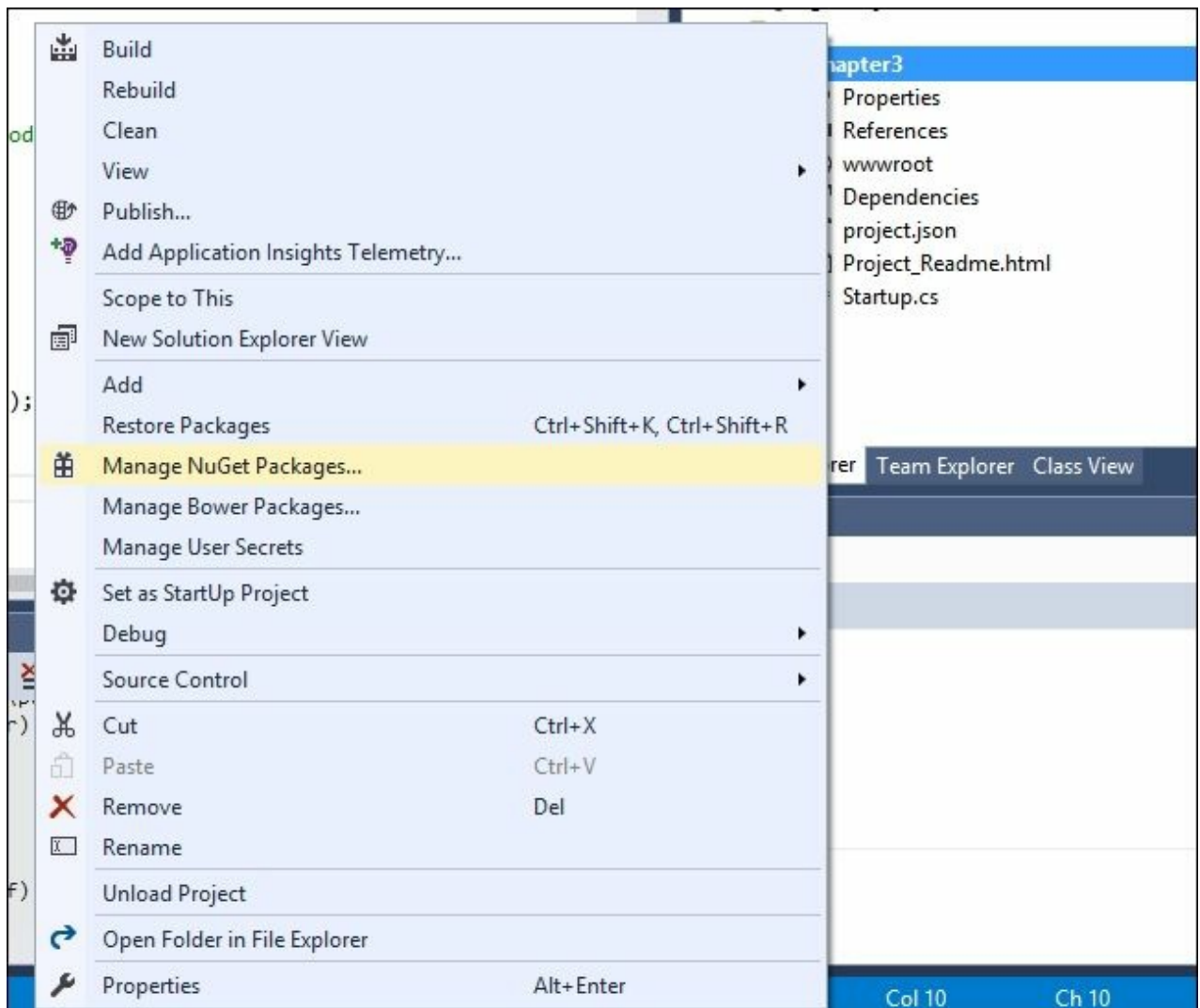
2. Select the **Include Prerelease** checkbox so that the **NuGet Package Manager** will list out all the prerelease packages. Search for `MVC` and you'll get the **Microsoft.AspNet.MVC** package, as shown in the following result, and click on the **Install** button on the right-hand side:
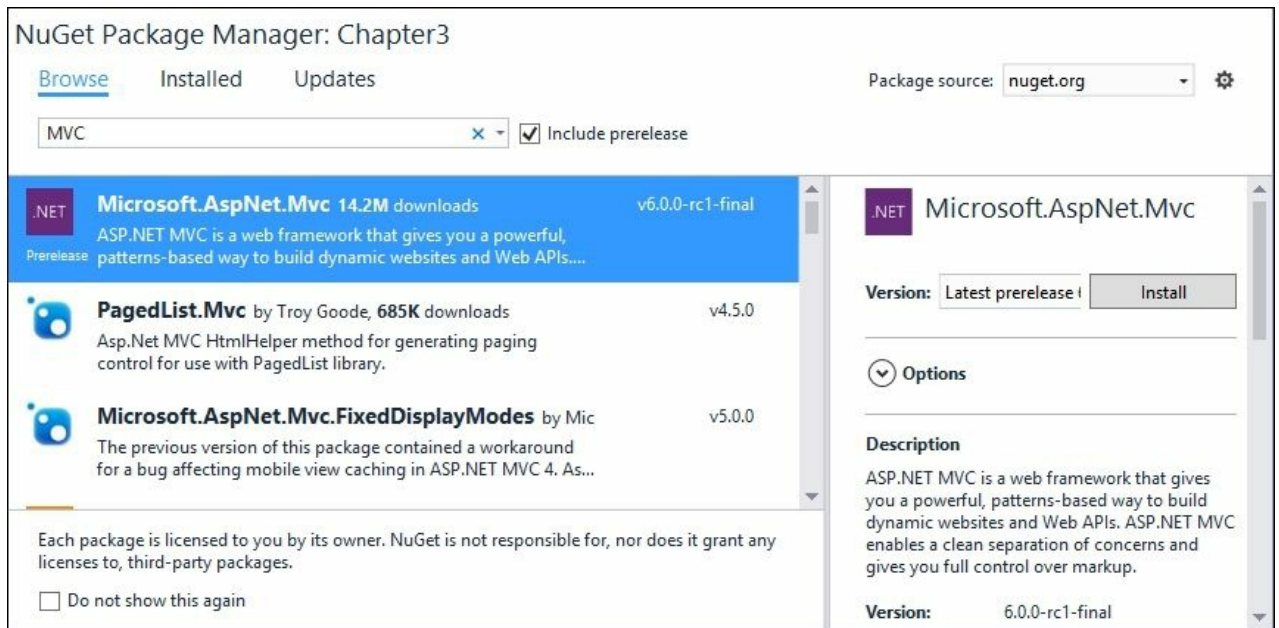
3. Review the changes:
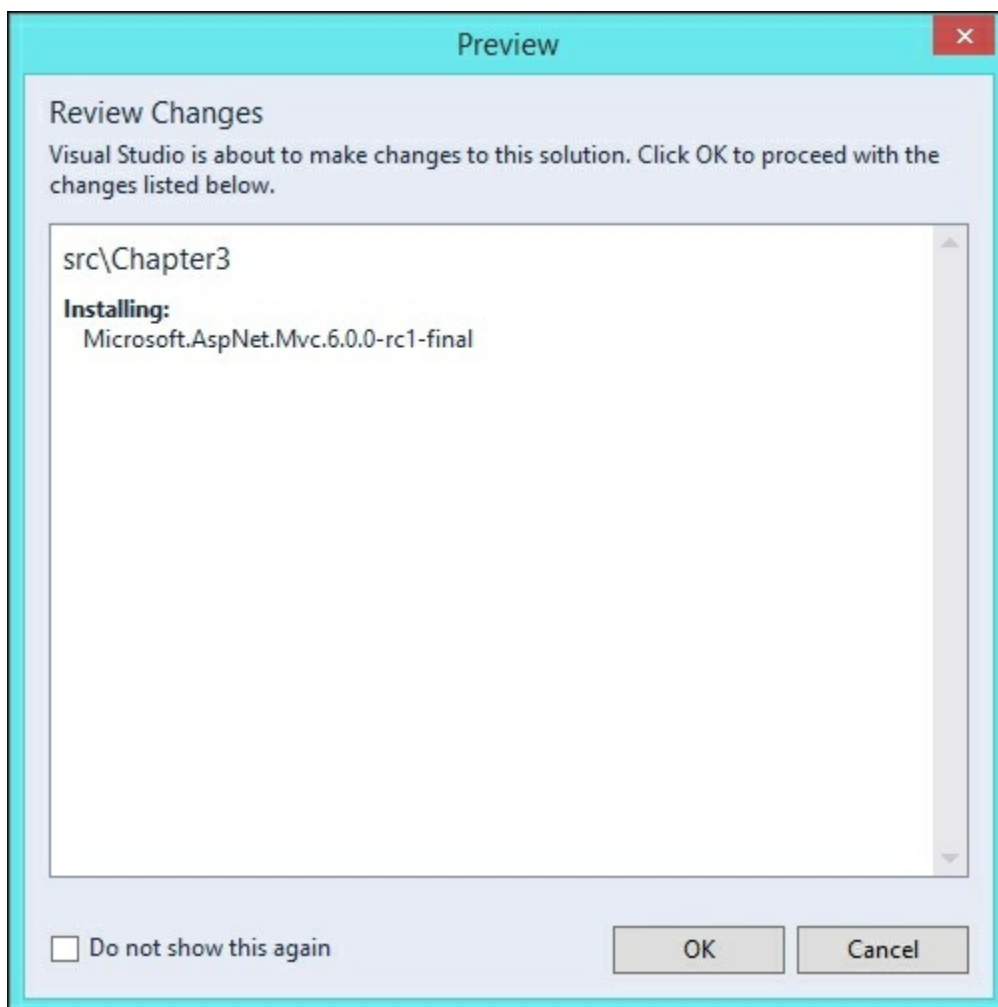
4. Once you click on **Review Changes**, the following dialog box will appear where you need to accept the license terms:
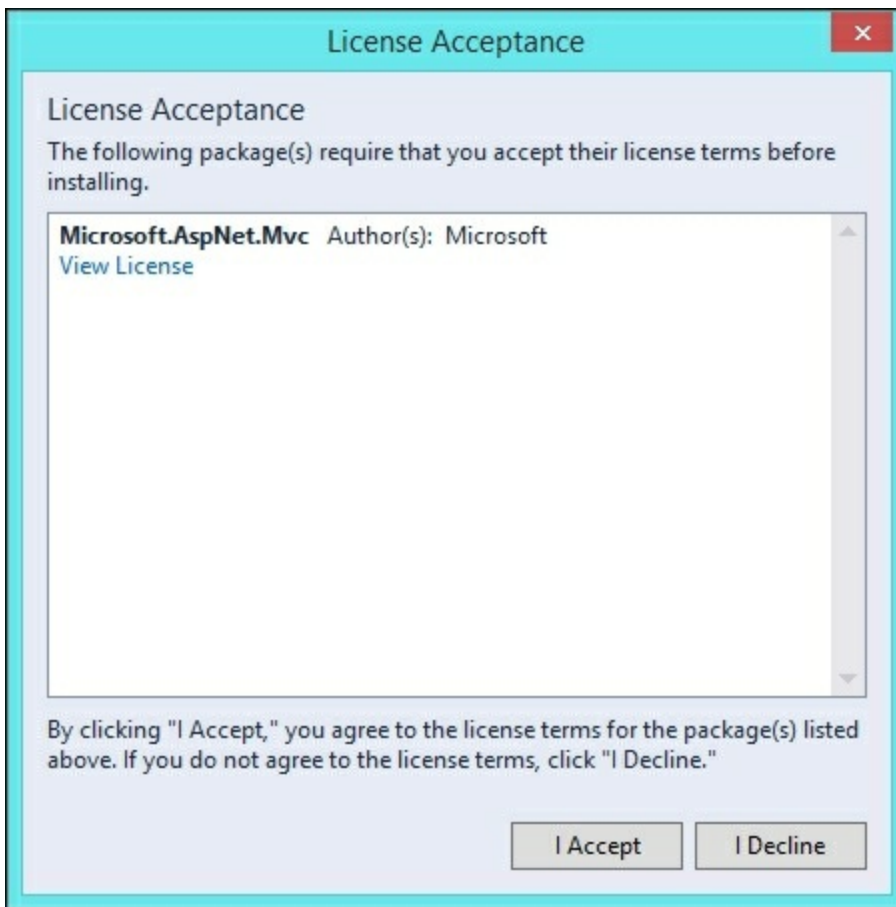
The **NuGet Package Manager** will download and install the ASP.NET Core and will update the `project.json` file and the associated references.

Now, your `project.json` file will have updated dependencies. The second line `Microsoft.AspNet.Mvc` is added:

```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final"
},
```

Alternatively, you can also update the `project.json` with the `NuGet` package along with the version information. The **NuGet Package Manager** will automatically download and install them.

ASP.NET Core is installed in our application. Now, we need to tell our application to use ASP.NET MVC.

This needs a couple of changes to the `Startup.cs` file:

1. Configure the application to add the MVC service. This can be done by adding the following line to the `ConfigureServices` method of the `Startup` class:

   ```
   services.AddMvc();
   ```

2. Configure the routing so that our correct controllers will be picked for the incoming requests based on the URL entered. The following code snippet needs to be updated in the `Configure` method of the `Startup.cs` file:app.UseMvc(routes => {

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

In the preceding statement, we are configuring the routes for our application.

In this chapter and most of the chapters in this book, we will write codes manually or choose an **Empty** template instead of relying on scaffolding templates. For those who are new to the term **scaffolding**, scaffolding is a feature that generates all the necessary boilerplate code for you for the selected item (for example, the Controller) instead of you needing to write everything. Though I agree that scaffolding templates are useful and save time in generating the boilerplate code, they hide many of the details that beginners have to understand. Once you write code manually, you'll know all the intricacies of how each of the components is contributing to the big picture. Once you are strong in the fundamentals, you can use scaffolding templates to save you time in writing the boilerplate code.

Our first Controller

Before creating the Controller, we need to remove the following `app.Run` statement as this will return `Hello World!` for all the incoming requests. As we want incoming requests to be handled by the controllers, we need to remove the following code from the `Configure` method of the `Startup` class:

```
app.Run(async (context) => {
  await context.Response.WriteAsync("Hello World!");
});
```

We have installed the ASP.NET Core in our application. So, we are geared up to creating our first ASP.NET Core controller. Create a folder with the name `Controllers` and add a new Controller by selecting from the context menu as shown in the following screenshot:
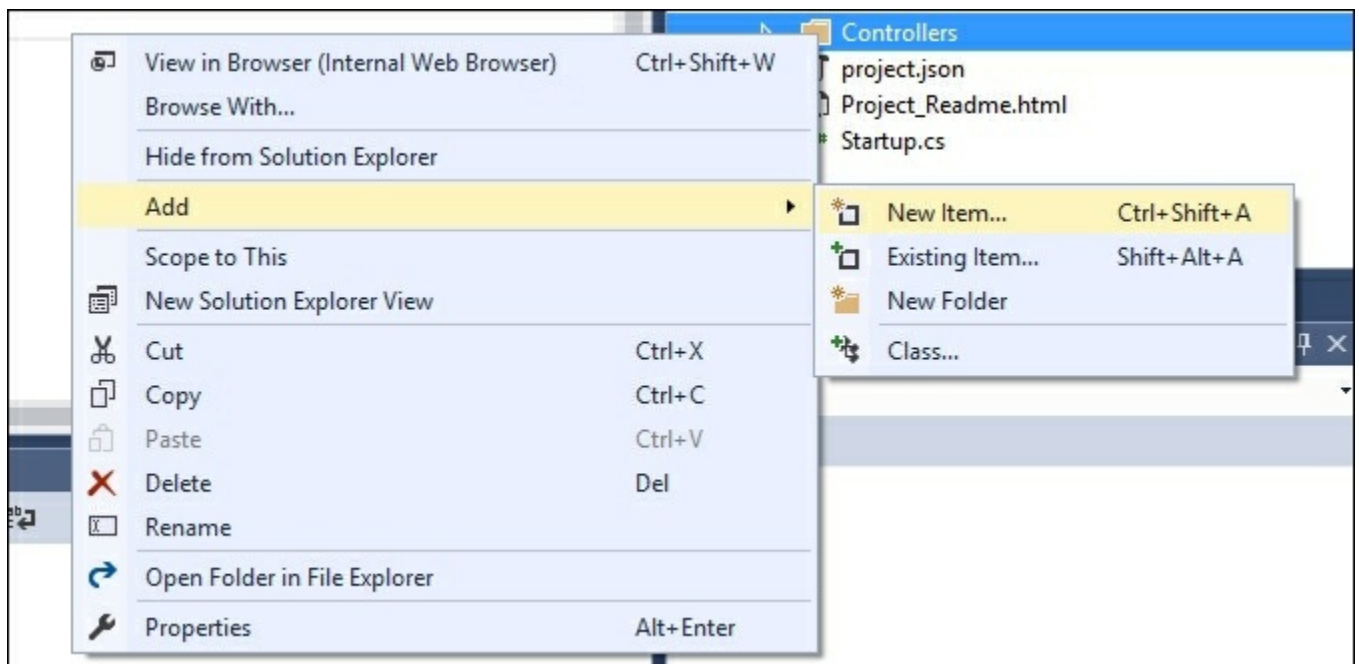
Once you select **Add** | **New Item**, you will be shown the following list of options. We are going to add an MVC controller class to our project:
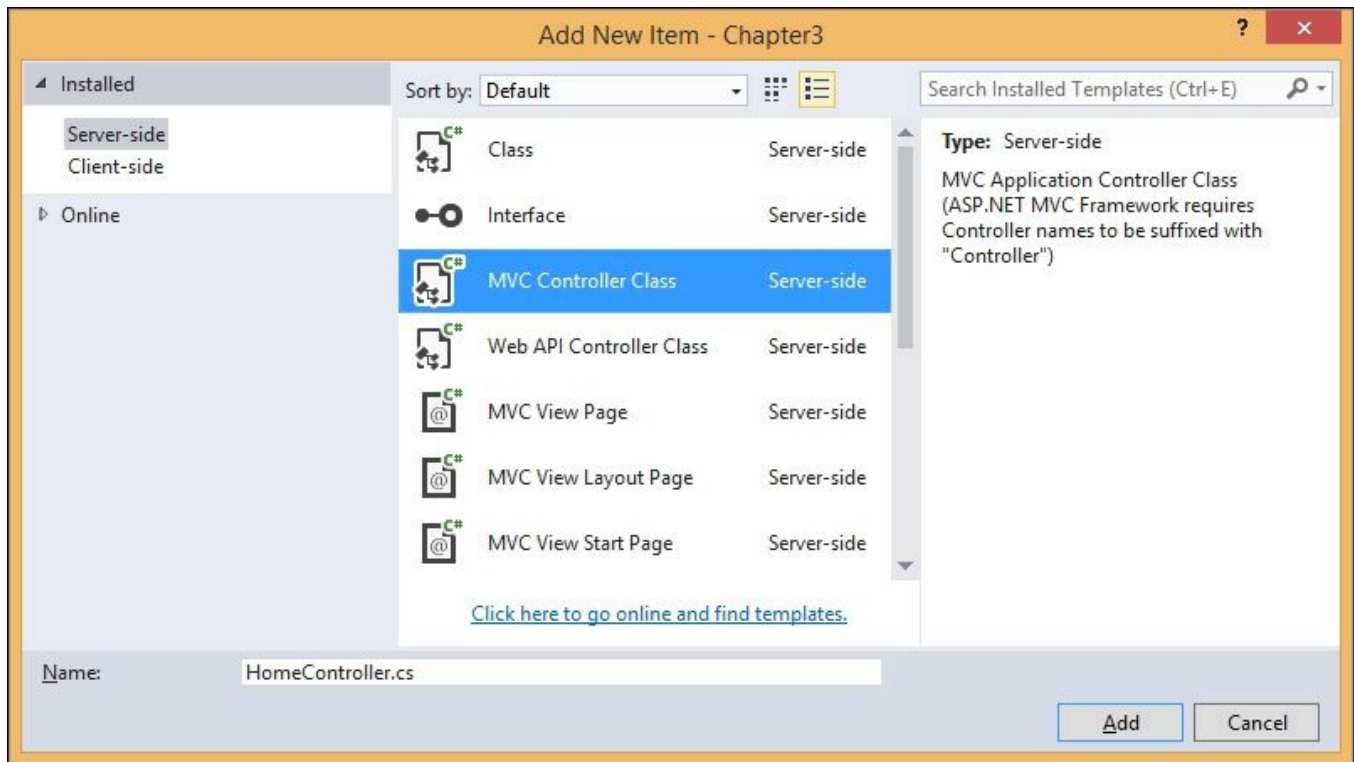
A class will be created with the following content:

```
public class HomeController : Controller {
  // GET: /<controller>/
  public IActionResult Index() {
    return View();
  }
}
```

All controllers, both MVC and Web API controllers, inherit from the `Controller` base class. In earlier versions of ASP.NET MVC, MVC controllers would inherit from the `Controller` class and Web API controllers would inherit from the `APIController` class.

In the preceding `HomeController` class, we have a single action method by `Index` that returns the corresponding View. When you run the application as it is, you'll get a **500 Internal Server Error**. The reason being is that no View has been created for the `Index` action of the `HomeController` and ASP.NET Core tries to search for that View. As the View is not available, it returns a **500 Internal Server Error**.

Instead of creating and returning that View, let us make a simple change to this action method. Let us return a string, `Hello World! I am learning MVC 6!`, and change the return type of `IActionResult`:

```
public string Index() {
  return "Hello World! I am learning MVC 6!";
}
```

Run the application. You'll see the **Hello World! I am learning MVC 6!** in your browser as shown in the following screenshot. Please make sure that you remove the `app.Run` statement in the `Configure` method as mentioned earlier:
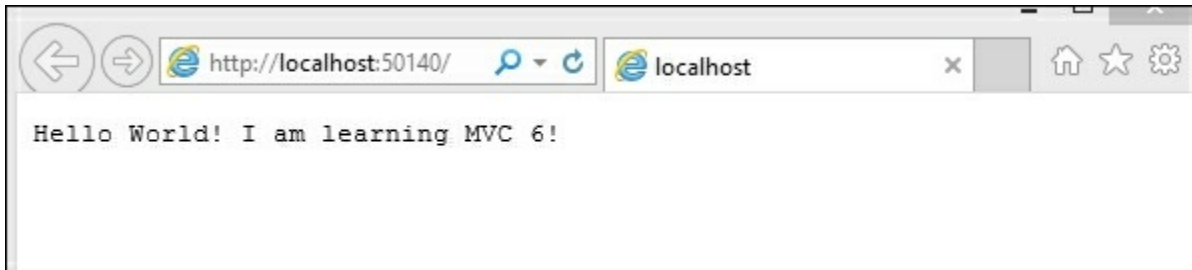
Voila! We have changed the ASP.NET Core application to render the custom content instead of the boring *Hello World*. What we have done may seem like a marginal improvement, but we have used controllers and action methods in our ASP.NET Core application, which has brought a lot of structure and flexibility to the web application development.
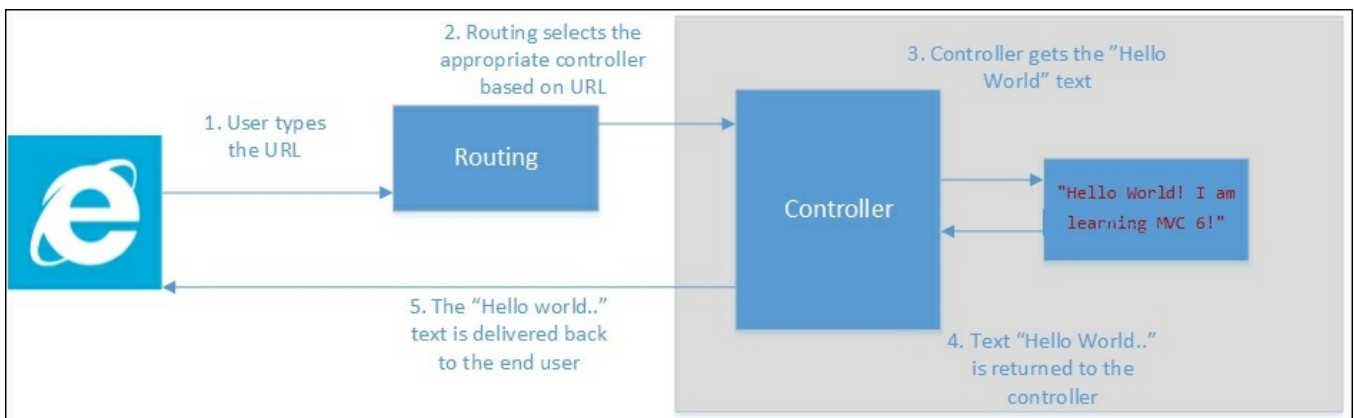
The following is the sequence of steps that occur when we run the application:

1. The application runs on the URL `http://localhost:50140`, where `50140` is the port number selected by IIS Express to run the application on my local system. This number may vary.
2. As we have not passed any parameter, default values for the `Controller` and `action` method will be selected. In our case, `HomeController` will be chosen as the `Controller` and `Index` will be chosen as the `action` method in the

`HomeController`. Since `ID` is the optional value and it is not passed, this `ID` parameter is ignored.

3. After the `Controller` and `action` methods are selected by the routing engine, control is passed to the `action` method of the selected controller. In our case, it will be the `Index` action method of the `HomeController`.

4. In the `Index` action method, we are returning a string, `Hello World! I am learning ASP.Net MVC 6!`. This text is returned from the controller, which would then return back to the user.

IActionResult

If you noticed, the default return type in the `action` method of the controller was `IActionResult` and then we changed the return type to the string in order to return the text `Hello World....`

The `IActionResult` is the interface that we can use to return different types of `ActionResult`, ranging from a simple string to complex JSON data, so, we don't need to change the `return` type of the `action` method to return the string.

In the earlier example, I have changed the `return` type to the string to make things simple. Now, let us make a simple change to return the string by keeping the return type (`IActionResult`) as it is:

```
// GET: /<controller>/
public IActionResult Index() {
  return Content("Hello World! I am learning MVC 6!");
}
```

While returning the string, we are using the `virtual` method, called `Content` from the `Controller` class (the base controller from where `HomeController` is inherited from) in the preceding `action` method. The purpose of this `Content()` method is to convert the string to the type `IActionResult`.

Now, run the application. We should be getting the same result.

`IActionResult` is capable of returning different data types:

- `ContentResult`: Can return a text result.
- `EmptyResult`: Returns a `null` result.
- `FileResult`: Returns a binary output to write to the response.
- `HttpStatusCodeResult`: Provides a way to return.
- `JavaScriptResult`: Returns a script that can be executed from the client side.
- `JSonResult`: When you return a serialized JSON object.
- `RedirectResult`: Redirects to another `action` method.
- `RedirectToRouteResult`: Represents a result that performs a redirection by using a specified route values dictionary.

Adding Views

We were returning a simple string from the controller. Although that explains the concept of how the `Controller` and `action` method works, it is not of much practical use.

Let us create a new `action` method by the name, `Index2`:

```
public IActionResult Index2() {
  return View(); // View for this 'Index2' action method
}
```

Now, we have created the `action` method that returns a View. But we still have not added the View for the same. By convention, ASP.NET MVC would try to search for our View in the folder `Views\{ControllerName}\{ActionMethod.cshtml}`. With respect to the preceding example, it will try to search for `Views\Home\Index2.cshtml`. Please note that the name of the `controller` folder-is `Home` , not `HomeController`. Only the prefix is needed as per convention. As this folder structure and file are not available, you'll get a **500 Internal Server Error** when you try to access this action method through the URL `http://localhost:50140/Home/Index2`.

So, let us create a folder structure. Right-click on the solution, select **Add** | **New Folder** from the context menu, create a folder called `Views`, and then create a subfolder by the name `Home` within the `Views` folder:
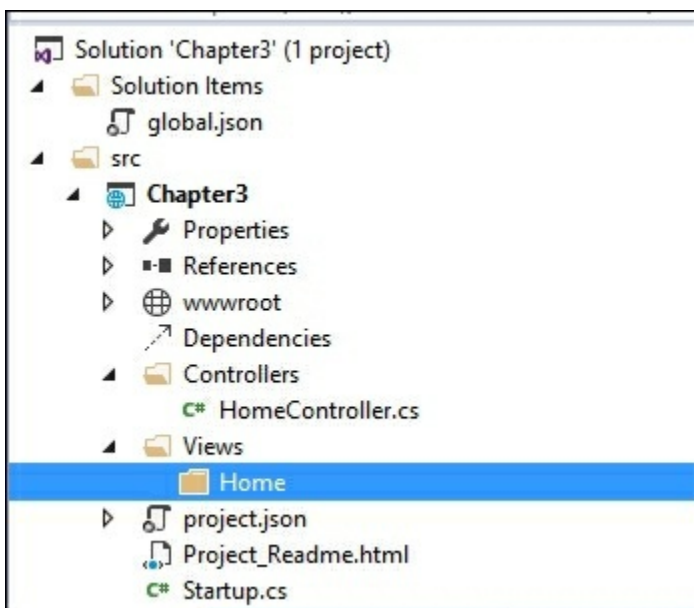
Right click on the `Home` folder, and select **Add | New Item** from the context menu. A dialog will appear as shown in the following screenshot. Give the name of the file as `Index2.cshtml`, as our `action` method name is `Index2.cshtml` is the razor view engine (this will be discussed in detail in the *ViewEngines* section of the *Views* chapter) extension used when you are using C#.
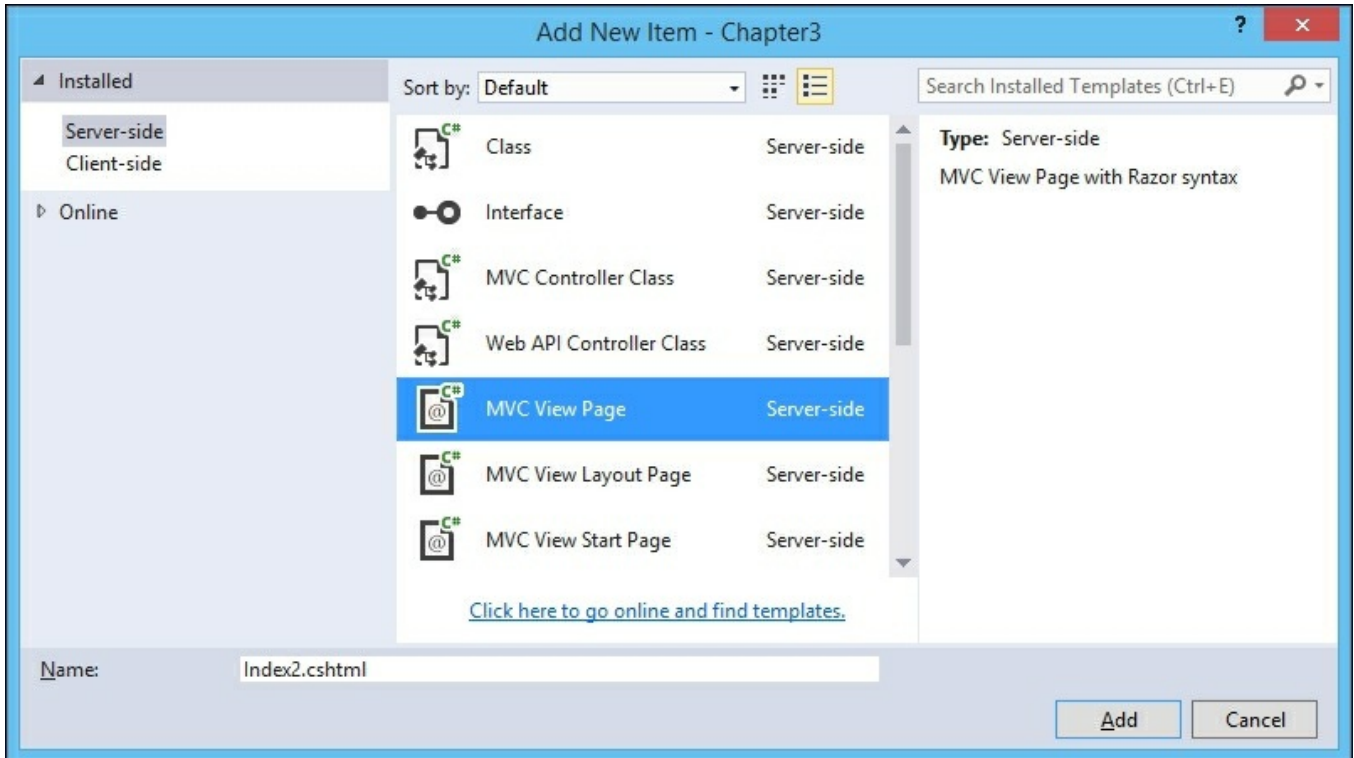
A file by the name `Index2.cshtml` will be created when you click the **Add** button in the preceding screen with the following content:
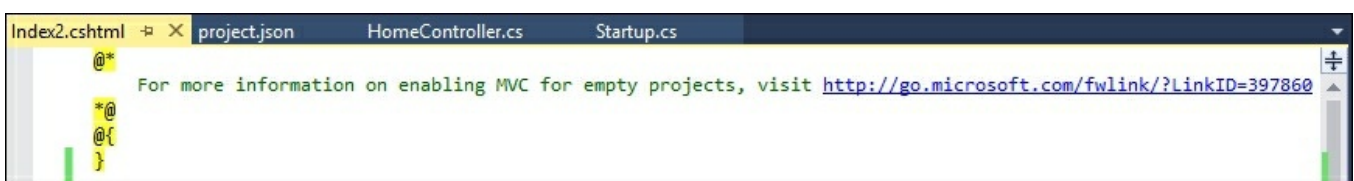
`@*` is the comment syntax in the razor view engine. You can write any C# code within the `@{}` block.

Let us add a simple HTML block after the generated code:

```
<html>
  <body>
```

```
    Hello! This is <b>my first View</b>
  </body>
</html>
```

Now, when you run the application, you will get the following output:
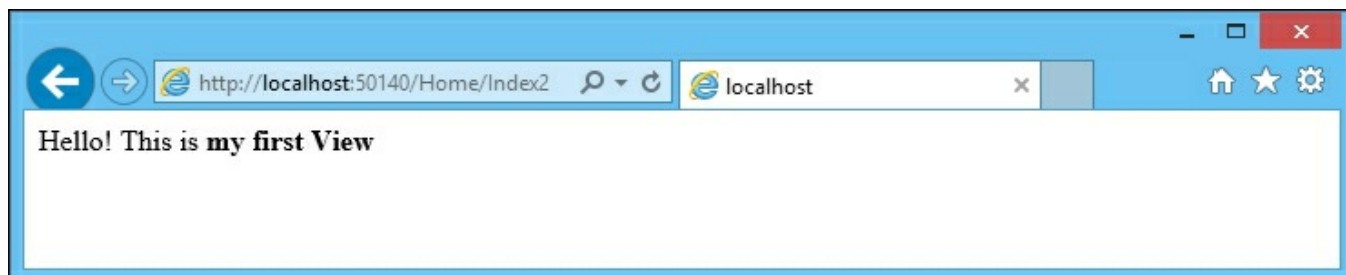
The following diagram explains the request flow and how we generate the response through the View:
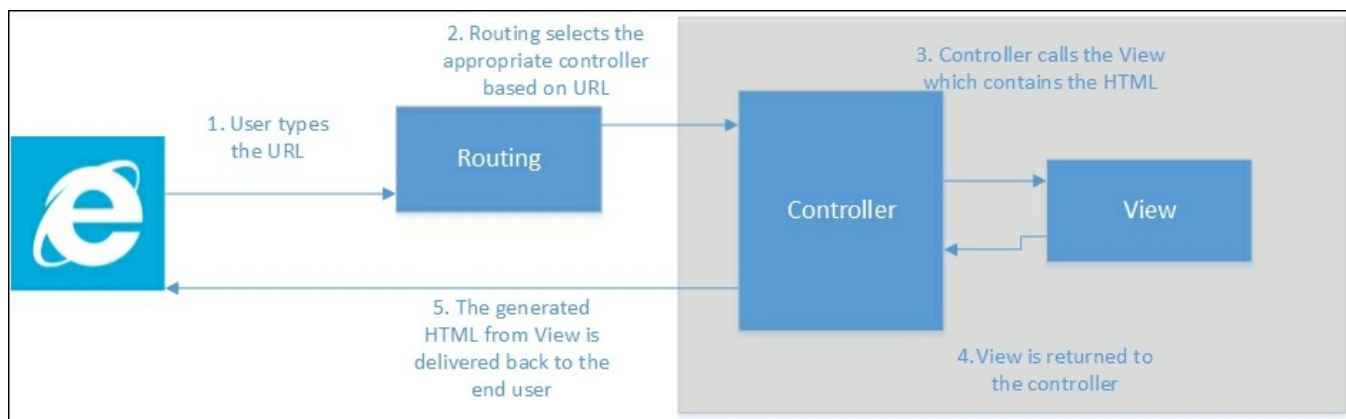
Adding Models

Models represent your business domain classes. Now, we are going to learn about how to use the Models in our controller. Create a `Models` folder and add a simple `Employee` class. This is a just a plain old C# class:

```
public class Employee {
  public int EmployeeId { get; set; }
  public string Name { get; set; }
  public string Designation { get; set; }
}
```

Create a new `action` method, `Employee`, in our `HomeController`, and create an object of the `Employee` Model with some values, and pass the Model to the View. Our idea is to use the Model employee values in the View to present them to the user:

```
using Chapter3.Models;
public IActionResult Employee() {
  //Sample Model - Usually this comes from database
 Employee emp1 = new Employee {
    EmployeeId = 1,
    Name = "Jon Skeet",
    Designation = " Software Architect"
  };
  return View(emp1);
}
```

Now, we need to add the respective View for this `action` method. Add a new Razor view file in the `View\Home folder`.

Add the following code snippet. Whatever comes after the `@` symbol is considered as Razor code. In the following code, we are trying to access the properties of the `Model` object that is passed to our view. In our case, `Model` represents the `employee` object that we have constructed in our `action` method. You can access the object from View using the Model keyword:

```
<html>
  <body>
 Employee Name : @Model.Name <br/>
    Employee Designation: @Model.Designation <br/>
  </body>
</html>
```

When you run the application and type the URL `http://localhost:50140/Home/Employee`, you'll see the following output:

Employee Name : Jon Skeet
Employee Designation: Software Architect
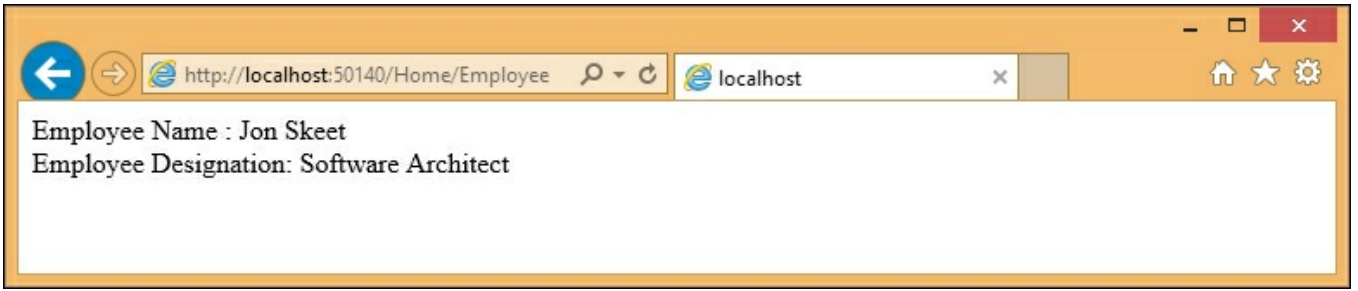
Passing data from Controller to View

We have just discussed how to pass the data from the Controller to the View using the `Model` object. While calling the View, we are passing the model data as a parameter. But there are times when you want to pass some temporary data to the View from the Controller. This temporary data may not deserve a `model` class. In such scenarios, we can use either `ViewBag` or `ViewData`.

`ViewData` is the dictionary and `ViewBag` is the dynamic representation of the same value.

Let us add the company name and company location property using `ViewBag` and `ViewData` as shown in the following code snippet:

```
public IActionResult Employee() {
  //Sample Model - Usually this comes from database
  Employee emp1 = new Employee {
    EmployeeId = 1,
    Name = "Jon Skeet",
    Designation = " Software Architect"
  };

  ViewBag.Company = "Google Inc";
  ViewData["CompanyLocation"] = "United States";

  return View(emp1);
}
```

Make the respective changes in the View file as well so that we can display the `Company` name and `Company location` values:

```
<html>
  <body>
    Employee Name : @Model.Name <br/>
    Employee Designation: @Model.Designation <br/>
    Company : @ViewBag.Company <br/>
    Company Location: @ViewData["CompanyLocation"] <br />
  </body>
</html>
```

Run the application after making the preceding changes:

```
Employee Name : Jon Skeet
Employee Designation: Software Architect
Company : Google Inc
Company Location : United States
```
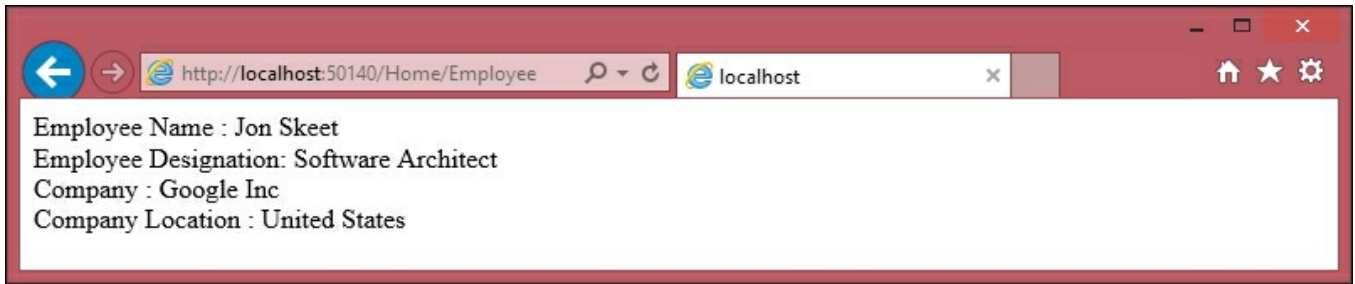
`ViewBag` and `ViewData` represent the same collection, even though the entries in the collection are accessed through different methods. `ViewBag` values are dynamic values and are executed at run-time, whereas the `ViewData` is accessed through the dictionary.

To test this, let us make a simple change to our `view` file:

```
Employee Name : @Model.Name <br/>
Employee Designation: @Model.Designation <br/>
Company : @ViewData["Company"] <br />
Company Location : @ViewBag.CompanyLocation <br />
```

Even though I have stored the `Company` value using `ViewBag` in the `Controller`, I am accessing the same using `ViewData`. The same is the case for the `Company Location` value, we have stored the value using `ViewData` in the Controller, but we are accessing the value using `ViewBag`.

When you run the application after making the preceding changes, you'll see the same result as you have seen before.

Filters

Filters in ASP.NET MVC enable you to run code before or after a particular stage in the execution pipeline. They can be configured globally per-controller or per-action.

There are different kinds of filters, and each filter is executed at a different stage in the pipeline. For example, action filters are executed when the `action` method is executed.

Let us use a simple example to see how an action filter (a type of filter) works.

I have created a simple controller, `DateController`, where I am just displaying the time. In this `action` method, I am using a predefined action filter by the name of `ResponseCache`, that caches the response for the duration specified in seconds. In the following code snippet, we have mentioned the duration as 600 seconds. So, the response will be cached for 10 minutes.

```
public class DateController : Controller {
  [ResponseCache(Duration = 600)]
  public IActionResult Index() {
    return Content(DateTime.Now.ToShortTimeString());
  }
}
```

When I run it for the first time, it displays the time as expected. But when you refresh the browser (which indirectly fires the request again), the time is not updated as the response is cached already by the application. In the following screenshot, even though the time is 7:43, the application is still showing as 7:40:
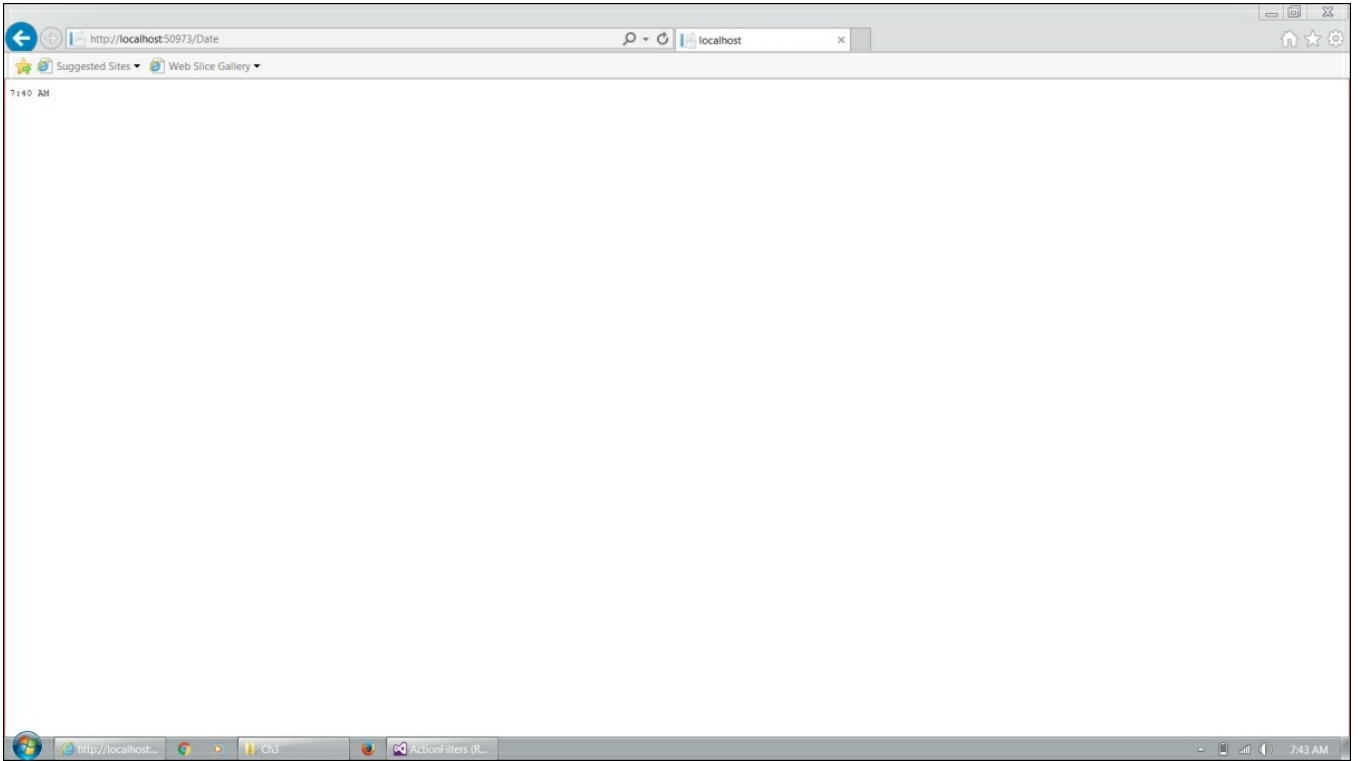
The following are the predefined types of filters available in ASP.NET Core.

Authorization filters

These are used for authorization and are mainly intended to determine whether the current user is authorized for the request being made.

Resource filters

These are the filters that handle the request after authorization and are the last one to handle the request before it leaves the filter pipeline. They are used to implement caching or by passing the filter pipeline.

Action filters

These wrap calls to individual `action` method calls and can manipulate the arguments passed in the action as well as the action result returned from it.

Exception filters

These are used to manage the unhandled exceptions in ASP.NET MVC.

Result filters

This wrap the individual action results and they only run when the `action` method is executed successfully.

Summary

In this chapter, we have built our first ASP.NET 5 application from scratch and we have installed ASP.NET Core in our ASP.NET 5 application. We have learned how the controller fits into the overall ASP.NET MVC application and learned how to build your first controller with the `action` methods. We also learned about how to use Model and View in our Controller. We have also discussed different ways to pass the data from the Controller to the View using `ViewBag` and `ViewData`. We have also learned about filters in ASP.NET MVC and how to make use of predefined filters in ASP.NET Core.

Chapter 4. Views

Views are the actual output of the application that is delivered to the user. It is what they actually see when they access your application from the screen. All the components, be it menus, input elements, dialog boxes, and everything the user sees comes from your Views only. If you do not provide good user experience when accessing your application, users will not care how great your application is. So, Views play a critical role when building an ASP.NET MVC application.

In this chapter, we'll cover the following topics:

- The purpose of View Engine and Razor View Engine
- Programming in Razor View Engine and different programming constructs
- Layout in ASP.NET Core and its features
- HTML Helpers
- Partial Views
- Tag Helpers

The View engine and the Razor View engine

As discussed in [Chapter 1](#), *Introduction to ASP.NET Core*, a browser can only understand HTML, CSS, and JavaScript. The purpose of the View engine is to generate the HTML code from your View and send it to the browser so that it can understand the content. Primarily, there are two different types of View engines— Razor View engine and Webform View engine. Although these two View engines come out of the box with ASP.NET MVC, you can use any custom View engine.

Razor View engine

The Razor View engine is the default and recommended View engine in ASP.NET Core, and going forward, this may be the only View engine coming out of the box when you install ASP.NET MVC.

You can mix a C# code and HTML code in your Razor View and the Razor View engine is intelligent enough to distinguish between these two and generate the expected output. In some scenarios, we may have to give additional information to Razor View to produce the appropriate results. Razor code blocks start with the `@` symbol and do not require a closing `@`.

Programming in Razor View engine

Programming in Razor View engine is just like you program in C#. The difference is that, in Razor View engine, your C# code will get mixed with HTML to produce the desired HTML output.

Variables in Razor View

You can declare a variable inside the razor block and use that variable using the `@` symbol.

Note

For all the examples in this chapter, we will only present the code samples of the view.

Let's discuss this with an example.

1. Create a `Controllers` folder and a Controller called `HomeController`.
2. Create a folder called `Views`, a subfolder called `Home`, and a View file called `Index.cshtml` by right-clicking the context menu and selecting **Add** | **New Item** and then **MVC Razor View** from the list.

The `HomeController.cs` file will have following code:

```
public class HomeController : Controller
{
  // GET: /<controller>/
  public IActionResult Index()
  {
    return View();
  }
}
```

Next is the updated Razor View where we will declare a variable and use it. The first five lines and the last two lines are simple HTML elements.

We will concentrate on the lines that are bold. Then, we will create a Razor block using @ { … } and declaring a variable inside it. The Razor block ends with the closing curly bracket. The snippet `Value:` is considered as simple HTML text. As we would like to use the razor variable value, we will use `@i` to instruct the Razor View engine that `i` is not a normal HTML text; and it is a Razor construct and is to be treated accordingly. The complete HTML code is as follows:

```
<html>
  <head>
    <title> Views demo</title>
  </head>
  <body>
    @{
        int i = 5;

    }
    Value: @i
  </body>
</html>
```

When you run the application, you'll see the following output:
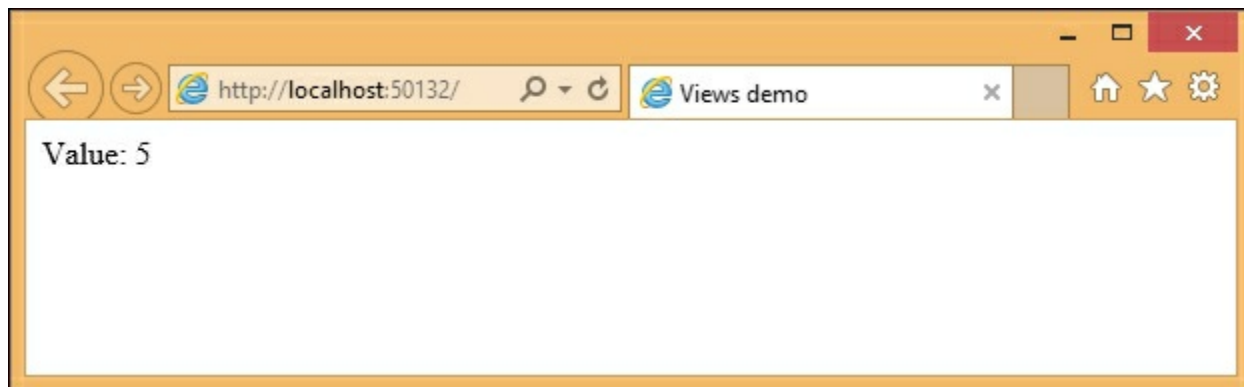
Please note that when you access the razor variable, you will need to use the @ symbol. Without this, Razor View engine sees the `i` as text and not as an expression.

The following screenshot is the result you will get when you access the variable without @ symbol:

Value: i
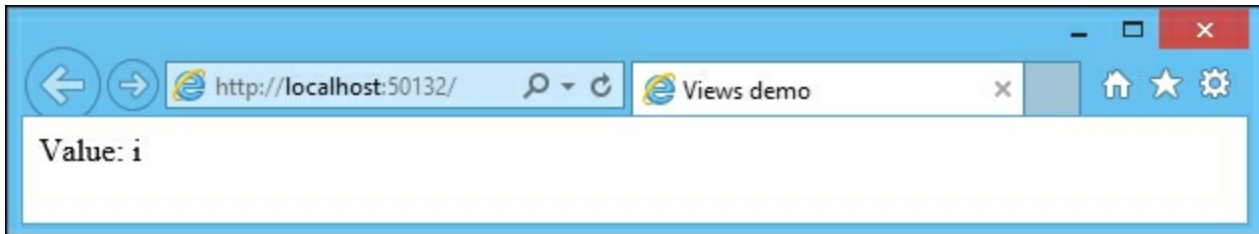
The for loop

You can use most of the programming constructs available in C# in Razor View. The following piece of code is the `for` loop construct where we loop through five times and print the variable name:

```
@{
   for (int i = 0; i < 5; i++)
   {
     <li>@(i+1)</li>
   }
 }
```

The following are a few points to be noted:

- As the for loop is a razor code, we should enclose the loop with the @ symbol to indicate that the code that follows is a Razor code and not normal HTML.
- Whenever we use an HTML element or tag, Razor View engine falls back to HTML mode. If you want to use any Razor expression within the HTML tags, you will want to include the @ symbol again to tell the Razor View engine that whatever follows is a Razor code and not an HTML element. This is the reason we use the @ symbol again in the preceding expression, even within the parent root-level razor code.

The complete code for the View is as follows:

```
<html>
  <head>
    <title> Views demo</title>
  </head>
  <body>
    <ul>
    @{
       for (int i = 0; i < 5; i++)
       {
         <li>@(i+1)</li>
       }
     }
    </ul>
```

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

The while loop

The following piece of code is the `while` loop implementation for the same loop. Please note that the emboldened expressions increment the variable i. We will not use the @ symbol as it is not within the HTML element:

```
@{
    int i = 0;
    while(i<5)
    {
      <li>@(i + 1)</li>
      i++;
    }
  }
```

The foreach loop

The `foreach` loop in Razor View is the same as the `foreach` loop in C#. In the following code, we will initialize a list of integers, iterate through the list and print it as a list item:

```
<ul>
  @{
      List<int> integers = new List<int>
      {
        1,2,3,4,5
      };
      foreach (int i in integers)
      {
        <li>@i</li>
      }
    }
</ul>
```

The if condition

In the following code, we will check if the value of the variable is less than 10. If it is less than 10, we will print `i is less than 10`, otherwise, we will say `i is greater than 10`. You may wonder why we have to include the `text` tag and what its purpose is. As we are inside the Razor View code block, the text `i is less than 10` will be considered as Razor expression, but it is not.

This `text` tag is to instruct the Razor View engine that whatever follows the `text` tag is to be considered as a text and not as a Razor expression:

```
@{
```

```
   int i = 5;
   if (i < 10)
   {
     <text>i is less than 10</text>
   }
   else
   {
     <text>i is greater than 10</text>
   }
}
```

Layout

In all the previous examples we discussed, we have done the complete View coding in a single file. This will result in a lack of flexibility and reduced reusability.

Consider the following web page structure where the **Top Section** contains the company logo or banner and the **Side Section** contains the links to various sections of the site. The **Content Section** changes for every page.
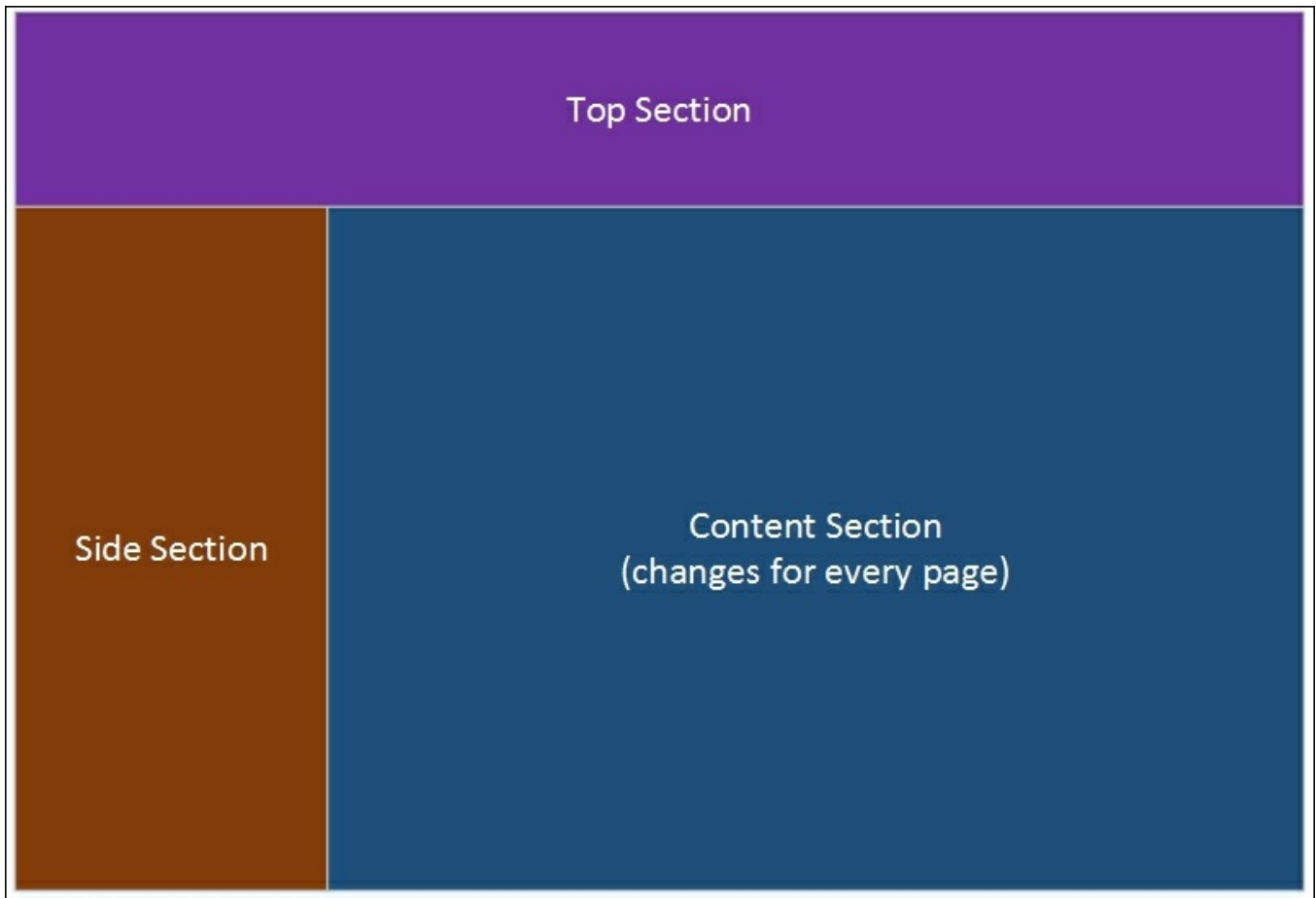
If we code the complete content in a single view, we may have to duplicate the **Top Section** and **Side Section** in every page. If we want to change anything in the **Side Section**, we will have to change all the files. This clearly shows that a single View file is not the best solution.

The layout comes to the rescue in this scenario. The layout defines the site structure that can be reused across all the web pages. The layout does not even need to have something like the top section or side section; it can contain even a simple HTML structure where you can have common content and the body content will be rendered

from individual view.

Let's build our first layout. In order to use the layout, you will need to have the following three things:

1. Inform the name of the layout file—this information should be made available in `_ViewStart.cshtml`. By convention, the names of all the shared files will start with an underscore and this file is located directly under the `Views` folder.
2. Create the Layout file—by convention, the name of the file is `_Layout.cshtml` and it will be located in the `Shared` folder. All the shared content, such as partial views, will also be available here. Partial Views will be discussed later in this chapter.
3. Create the content View file—this View file is almost same as the earlier View files that we created so far with only one difference; only page-specific content will be available in this file, and this means that you'll not have any `html`, `head`, or `title` tags here.
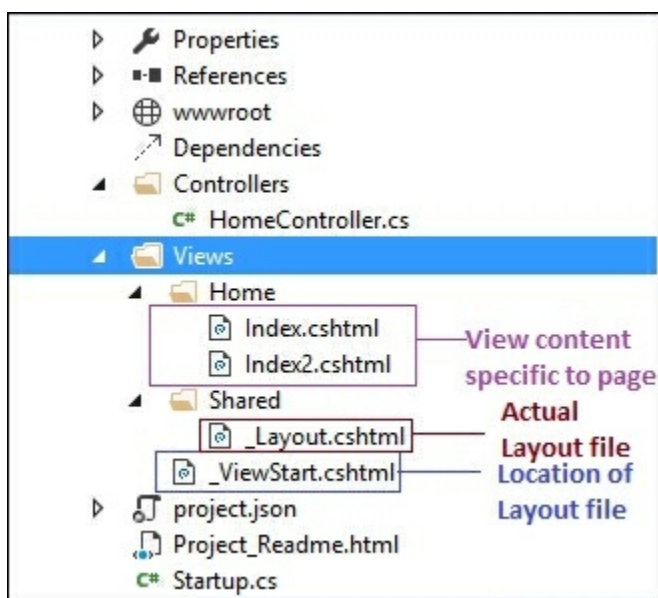
After the creation of `_ViewStart.cshtml`, `_Layout.cshtml`, and page-specific View files, the folder structure will be like the preceding snapshot.

Creating _ViewStart.cshtml

Right-click on the **Views** folder and select **Add New Item** from the **Context** menu. Then, select **MVC View Start Page** from the **Add New Item** dialog box as shown in the following screenshot:
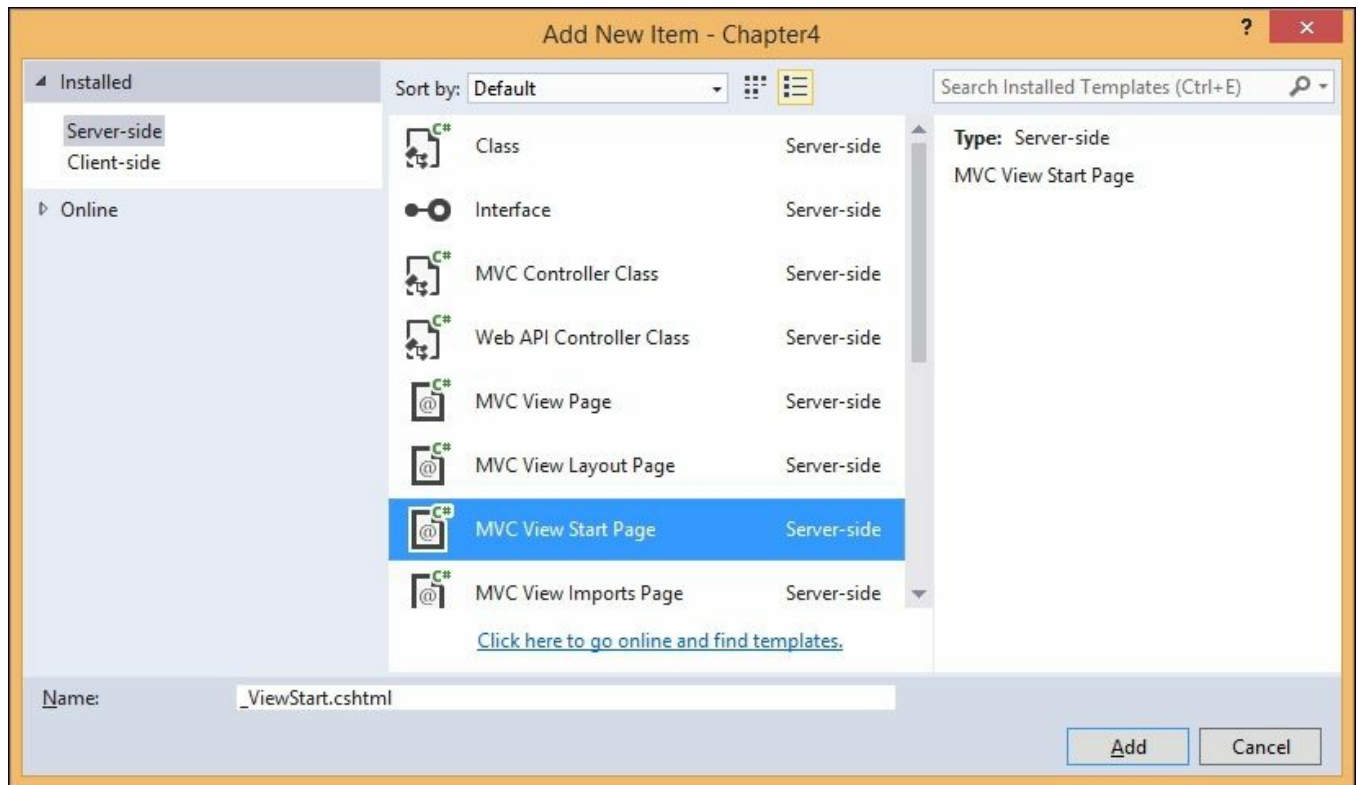
When you click the **Add** button, it will create a file with the following content:

```
@{
    Layout = "_Layout";
}
```

Creating _Layout.cshtml

Create a folder called **Shared** within the **Views** folder. Then, right-click on the **Shared** folder and select **Add New Item** from the **Context** menu as shown in the following screenshot:
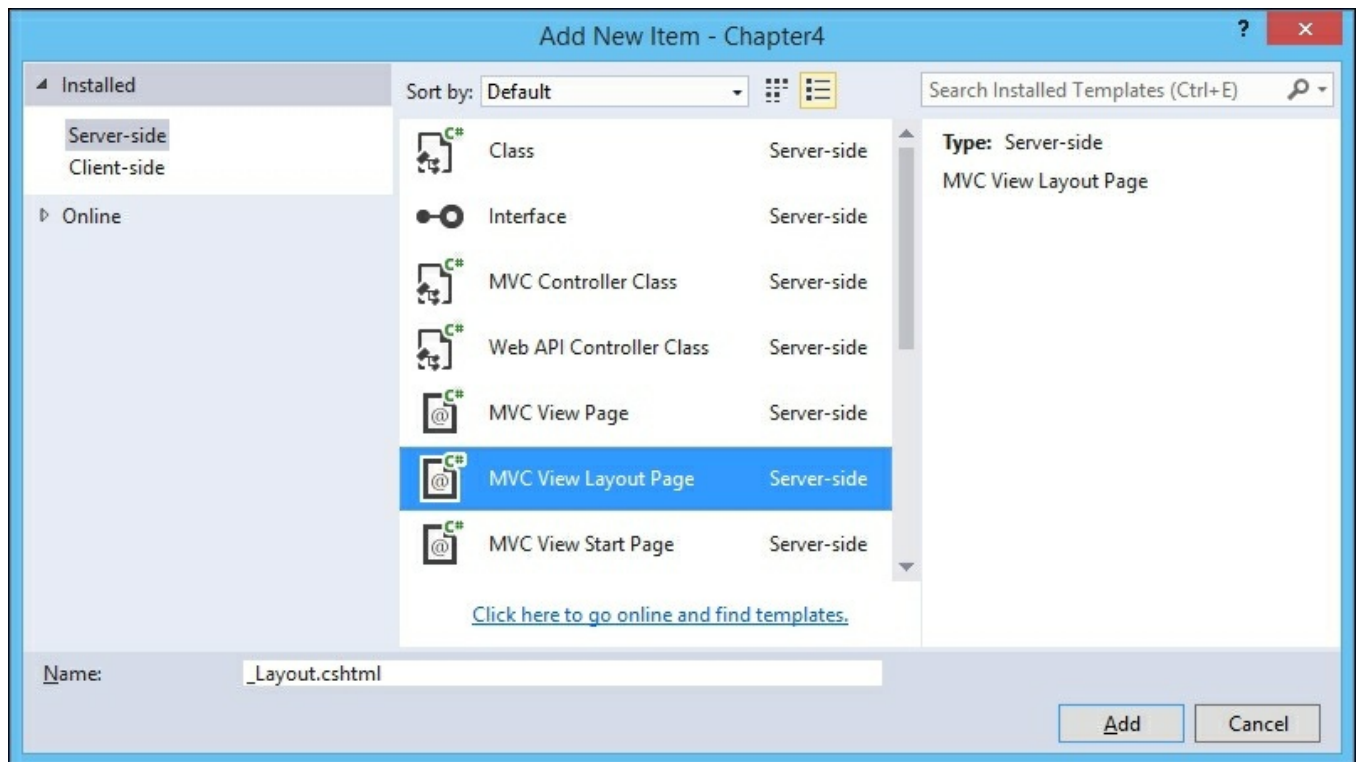
When you click the **Add** button, it will create **_Layout.cshtml** with the following content:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>@ViewBag.Title</title>
</head>
<body>
<div>
@RenderBody()

</div>
</body>
</html>
```

The preceding layout file is a simple HTML content with a couple of Razor expressions. @ViewBag. The title is used to display the title information passed from

the Controller and `@RenderBody` is the Razor expression that calls the page specific View and merges that content over there.

Adding a page-specific View

Before adding the View, we will need to add an action method in our `HomeController` file from which we will be calling our page-specific view.

Let's add an action method named `Index2` as follows:

```
public IActionResult Index2()
{
  ViewBag.Title = "This is Index2";
  return View();
}
```

The `ViewBag` is used to pass information from the Controller to the View. Here, we are passing the `Title` information from the action method to the View.

Now, right-click on the `Views` folder, select **Add** | **New Item**, select **MVC View Page**, and save the file as `Index2.cshtml`.

In the generated view, I have added simple `Hello` text. This text will be rendered in the body of the layout page. The complete code of the View file is as follows:

```
@*
For more information on enabling MVC for empty projects, visit
http://go.microsoft.com/fwlink/?LinkID=397860
*@
@{
 …
 }
 Hello. This text will be rendered in body of the layout page
```

Everything is set now. Run the application and type the URL `http://localhost:50132/Home/Index2` in the browser. Please note that the port number after the local host may vary when you run the application from your PC.

As expected, you'll see the text seen in the preceding picture. However, our point is not about the text. It's about the structure of the generated HTML content.

View the source by pressing *Ctrl + U* (on the Chrome browser in Windows). You'll see the following HTML content:

The top content (the `html`, `head`, `body`, and `div` opening tags) and bottom content (the `html`, `head`, `body`, and `div` closing tags) come from the layout file and the text comes from the View specific to the page.

Generating HTML

As discussed in [Chapter 1](#), *Introduction to ASP.NET Core*, browsers can understand only HTML, CSS, and JavaScript, irrespective of the technology that you use to build the web application. This holds true when building the application in ASP.NET MVC as well.

Most applications get the user input, process the input, and then store the required information in the database to retrieve them later. In the context of web applications, Form HTML elements are used to get the user input.

The following are a couple of ways to generate HTML elements in ASP.NET Core:
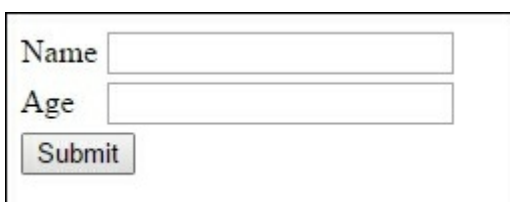
- HTML Helpers
- Tag Helpers

HTML Helpers are server-side methods that aid in generating HTML elements, which can be understood by the browsers. HTML helpers were the primary method of generating the HTML elements up till ASP.NET MVC 5.

Tag Helpers, introduced in ASP.NET Core, also produce HTML elements. Tag helpers, which we will discuss in a later section of this chapter, will look just like HTML elements where you add attributes to identify them as Tag Helpers. The advantage of using Tag helpers over HTML helpers is that the user interfaces designers/engineers do not need to worry about Razor code. They just code with HTML elements and additional attributes.

Before discussing HTML helpers and Tag helpers, let's take a step back and talk about why we need them in the first place.

Let's consider a simple form, as shown in the following picture, where we would like to get the user's name and their age. If the user enters her age, we will display `You are eligible to vote!`. If not, we will display `You are not eligible to vote now`:

The following is the HTML code to show the preceding simple form:

```html
<form>
  <table>
    <tr>
      <td>
        <label for="txtName">Name</label>
      </td>
      <td>
        <input type="text" id="txtName" />
      </td>
    </tr>
    <tr>
      <td>
        <label for="txtAge">Age</label>
      </td>
      <td>
        <input type="text" id="txtAge" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit"  />
      </td>
    </tr>
  </table>
</form>
```

This method of coding HTML elements directly is time-consuming and error-prone. For example, in the preceding form, the label and input HTML elements refer to the same element (`txtName` in the first group and `txtAge` in the second group). If we hand-code the HTML element, there is a possibility of a typo error in building the HTML element.

HTML Helpers

HTML helpers are server-side methods that generate HTML for you. We can generate the same form using HTML helpers as follows (`HTML.BeginForm`, `@Html.Label`, and `@Html.TextBox` generate the HTML `form` element, label, and textbox elements, respectively):

```
@using (Html.BeginForm())
 {
   <table>
     <tr>
       <td>@Html.Label("Name")</td>
       <td>@Html.TextBox("txtName")</td>
     </tr>
     <tr>
       <td>@Html.Label("Age")</td>
       <td>@Html.TextBox("txtAge")</td>
     </tr>
     <tr>
       <td colspan="2"><input type="submit" value="Submit" /></td>
     </tr>
   </table>
}
```
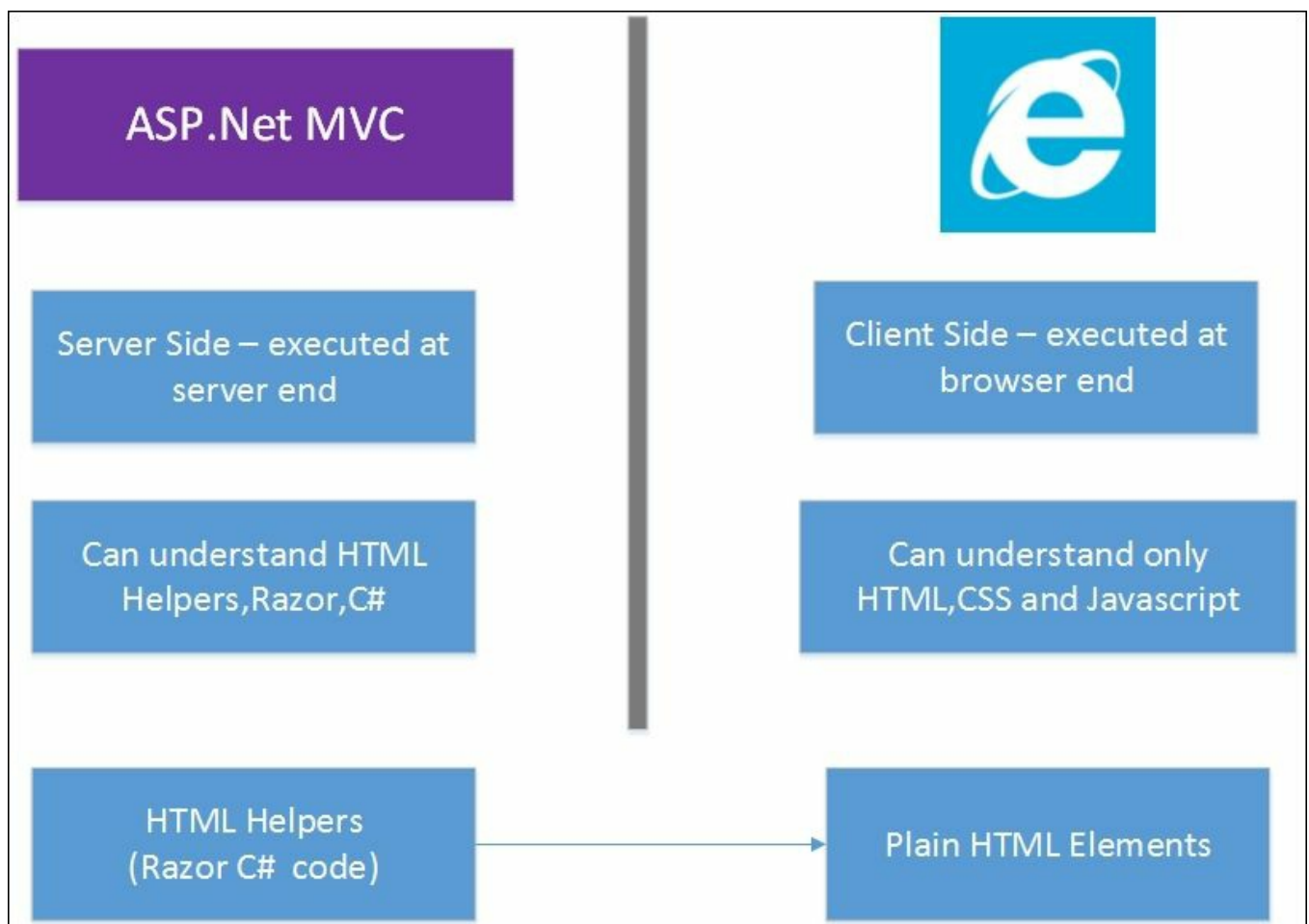
You might wonder why we need to use HTML helpers when we can write the HTML code manually. Things will get more complex when we pass the model from the Controller to the view. Using HTML helpers, we can directly build `form` elements from `Models` files so that they will pick the names from the `Models` that you are using.

For example, let's create a folder called `Models` and a class called `Person`. This class will act as a model as shown in the following screenshot:
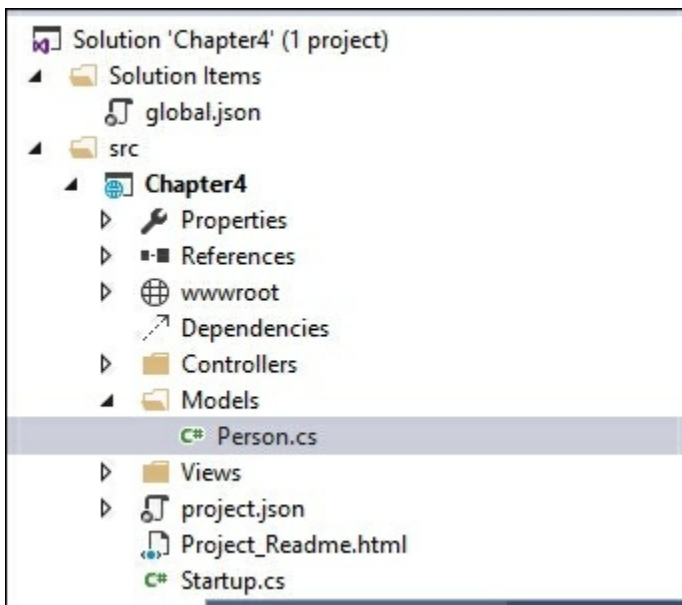
The `Person` class is just a POCO (Plain Old C# Object) class and will act as a model. The complete code for this class is as follows:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public int Age { get; set; }
}
```

Let's create a new action method called `ValidateAge`. In this method, we will create an empty `Person` class and pass the Model to the View. We will also create a dynamic property called `Title` in `ViewBag` so that we can display this value in View:

```
[HttpGet]
public IActionResult ValidateAge()
{
```

```
    ViewBag.Title = "Validate Age for voting";
    Person person1 = new Person();
    return View(person1);
}
```

In the view, create the `form` using the following HTML Helpers:

```
@model Chapter4.Models.Person
@using (Html.BeginForm("ValidateAge", "Home", FormMethod.Post))
  {
    <table>
      <tr>
        <td>@Html.LabelFor(Model => Model.Name) </td>
        <td>@Html.TextBoxFor(Model => Model.Name) </td>
      </tr>
      <tr>
        <td>@Html.LabelFor(Model => Model.Age)</td>
        <td>@Html.TextBoxFor(Model => Model.Age)</td>
      </tr>
      <tr>
        <td colspan="2"><input type="submit" value="Submit" /></td>
      </tr>
    </table>
}
```

In the first line, we are telling the View that we are passing the Model of type `Person` class. This enables you to use the strong type of Model, that is, when you type Model and a dot, **IntelliSense** provides you with all the properties of the `Person` class

In the second line, we are using the overloaded `BeginForm` HTML helpers which accept three parameters—the action method name, the Controller name, and the `Form` method.

Simply, when the user submits the form, the information should be passed to the mentioned action of the Controller.

In the `LabelFor` and `TextBox` For HTML helpers, we are just passing Model properties (name and age); it automatically queries and gets the Model properties and builds the respective HTML elements. This is one of the primary advantages of using HTML helpers. Without using the HTML helpers, this process might become complex.

Now, let's write the respective `POST` action method in the same way. In the following `POST` action method, based on the age entered in the form, we set the dynamic property as `Message`.

```
[HttpPost]
public IActionResult ValidateAge(Person person1)
```

```
{
  if(person1.Age>=18)
  {
    ViewBag.Message = "You are eligible to Vote!";
  }
  else
  {
    ViewBag.Message = "Sorry.You are not old enough to vote!";
  }
  return View();
}
```

It is to be noted that both the `GET` and `POST` action method refer to the same View
—`ValidateAge.cshtml`. Add the following content to the View just above the form
element:

```
@if(ViewBag.Message!=null)
  {
    <b>@ViewBag.Message</b>
  }
```

Once the user submits the form, the `POST` action method sets the dynamic `Message`
property in `ViewBag`. However, the value of this property will be null when the View
is rendered as part of the `GET` request. If the value is not null, insert the message at the
top of the page.

When you run the application, you'll get the following output:
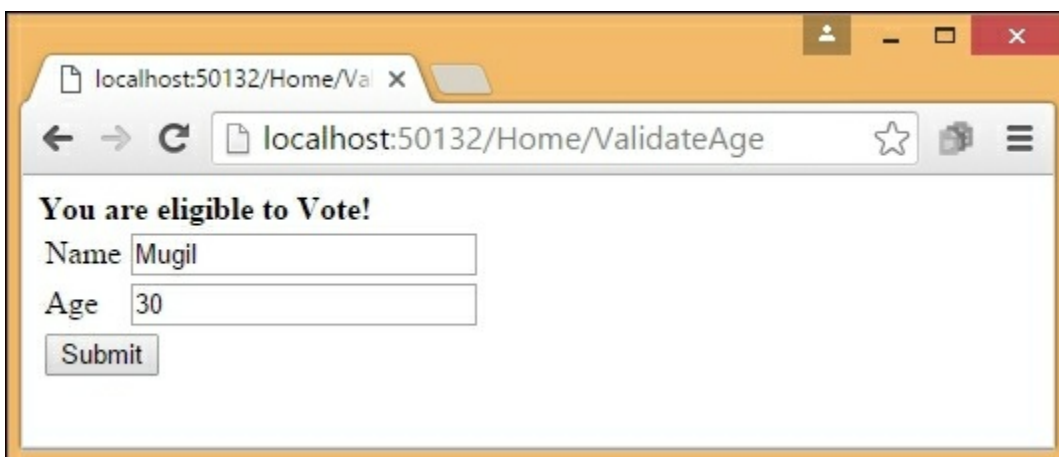
Partial View

Partial Views are just Views that can be reused across your application. Partial Views can be thought of as pluggable reusable blocks that you can call from anywhere and have the content of the partial view displayed.

Consider the following structure of a web page—it's the same layout page that we used earlier, but with a couple of changes. The **Latest News** block is added to the **Side Section** and the **Login** block is added to the **Top Section**. These blocks are not restricted to the **Top Section** or **Side Section** and can be used anywhere in your application, including your **Content Section** as shown in the following figure:
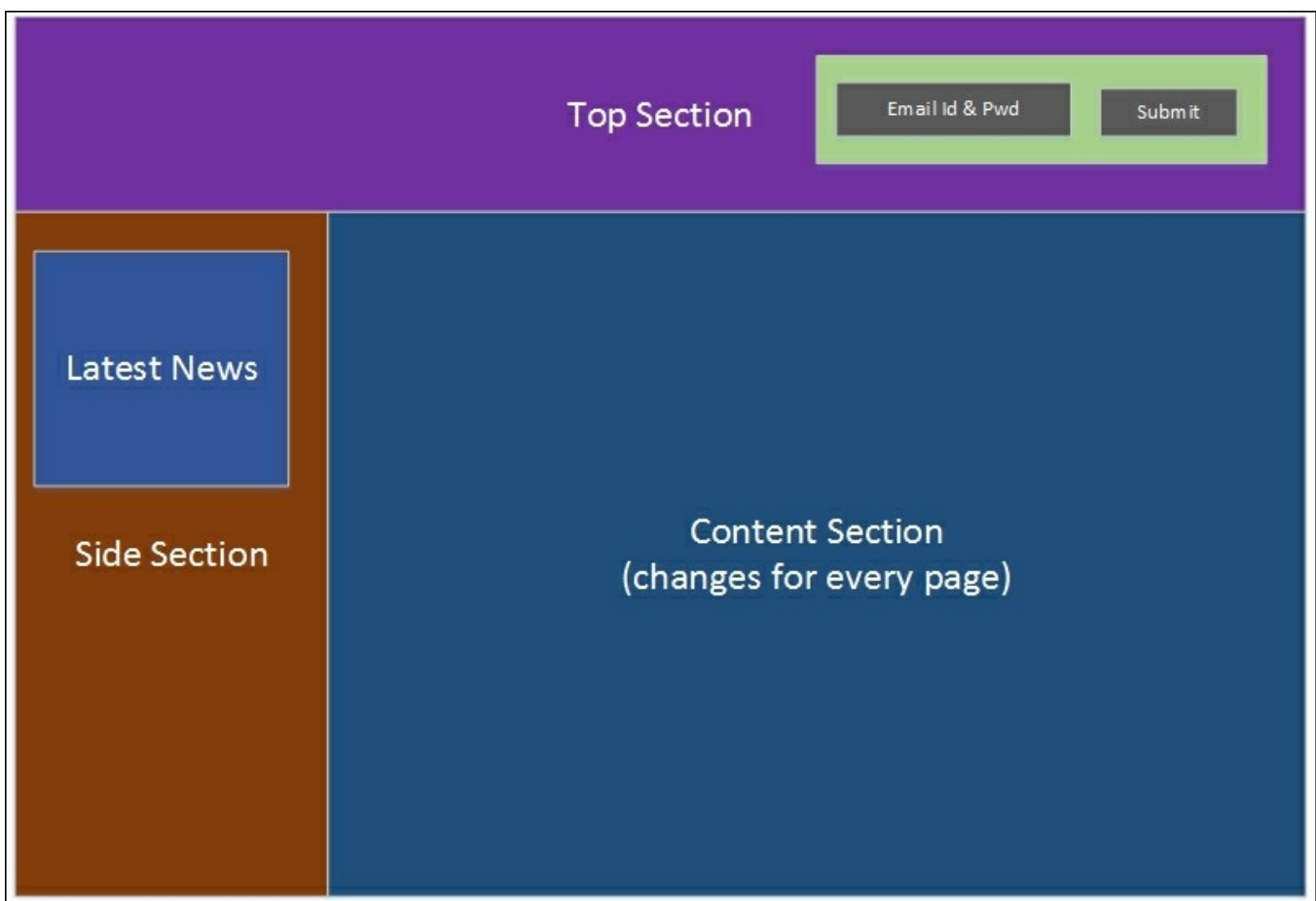
These Partial Views are not restricted to static content and can contain `form` elements. In the preceding screenshot, the **Latest News** Partial View contains the text content and the login Partial View contains `form` elements to get the e-mail ID and password.

Location of Partial Views—Framework does not restrict the location of the Partial

View. However, by convention, if your Partial View will be used only by your Controller, you can create that Partial View in the Controller-specific Views folder. For example, if your Partial View will only be used in `HomeController` file, you can create that Partial View in the `Views\Home` folder.

Let's take look at how to create a Partial View and use it.

As discussed earlier, a Partial View is just like a normal View. So, we will create a Partial View in the same way we create normal View.

Right-click on the `Shared` folder and select **Add** | **New Item**. By convention, like all shared content, the name of the Partial View will also start with "_"(underscore), as shown in the following screenshot:
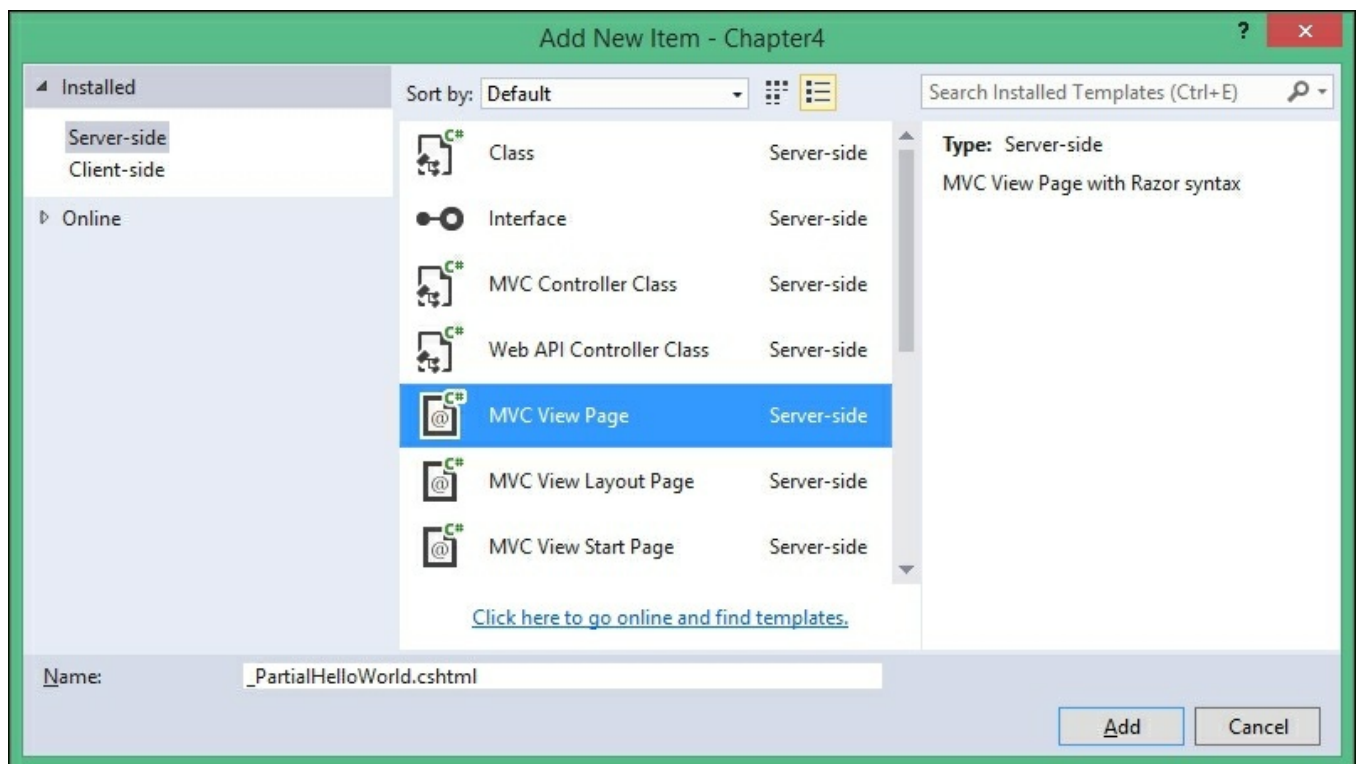
Tip

We are creating this Partial View based on the assumption that it can be used from anywhere in the application.

In the generated Partial View, I have added the following simple static content—a text and a simple table:

```
<b>This content and below table is coming from partial view</b>
<table border="1">
```

```
  <tr>
    <th>Employee No</th>
    <th>Employee Name</th>
  </tr>
  <tr>
    <td>10012</td>
    <td>Jon Skeet</td>
  </tr>
  <tr>
    <td>10013</td>
    <td>Scott Guthrie</td>
  </tr>
</table>
```

Calling the Partial View

A Partial View can be called using the `@Html.Partial` HTML helper.

In our case, we will be calling the Partial View from `Index2.cshtml` file. The parameter that you pass will be the name of the partial file. It will search for the Partial View by that name and render that complete content as part of the `Index2.cshtml` file.

The content of `Index2.html` file will now be as follows:

```
Hello. This text will be rendered in body of the layout page<br/>
<br/> <br/>

@Html.Partial("_PartialHelloWorld")
```

Now, run the application and access the URL `http://localhost:50132/Home/Index2`. You'll see the following output:
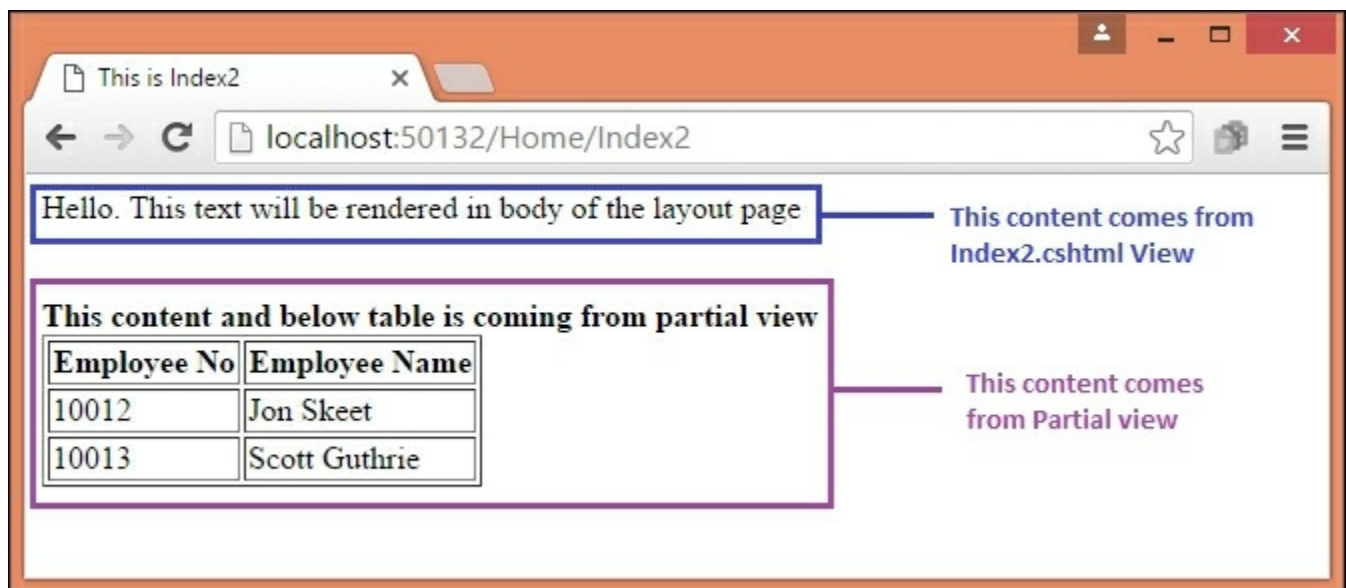
View components

View components are a new feature introduced in ASP.NET Core, they are almost similar to Partial Views but is more powerful. When you use Partial Views, you have dependency over the Controller. However, when you use the `ViewComponent` attribute, you do not have to depend on the Controller, so we will establish separation of concerns and have better testability. Even though the existing Partial View HTML helper is still supported, it is preferable to use the View component whenever you want to show a reusable piece of information when you are using .NET Core.

Creating a View component

You can create a `ViewComponent` using any of the following:

- Create a class by deriving from the `ViewComponent` attribute
- Decorate a class with the `[ViewComponent]` attribute or derive it from the class that has the `[ViewComponent]` attribute
- You can use the convention by creating a class that ends with a suffix `ViewComponent` attribute

Whatever option you choose, this `ViewComponent` should be public, non-nested, and non-abstract classes.

Like Controllers, you can use the Dependency Injection (via a constructor) in the `ViewComponent` attribute as well. As the `ViewComponent` attribute is separate from the Controller lifecycle, you may not be able to use the action filters in `ViewComponents`.

There is a method called `Invoke` (or `InvokeAync`, the asynchronous equivalent of `Invoke`), that will return the `IComponentViewResult` interface. This method is similar to the action method of the Controller that will return the View.

Let's get our hands dirty by creating a `ViewComponent` attribute.

Create a new folder called `ViewComponents` in your project and a new class called `SimpleViewComponent`, as shown in the following screenshot:
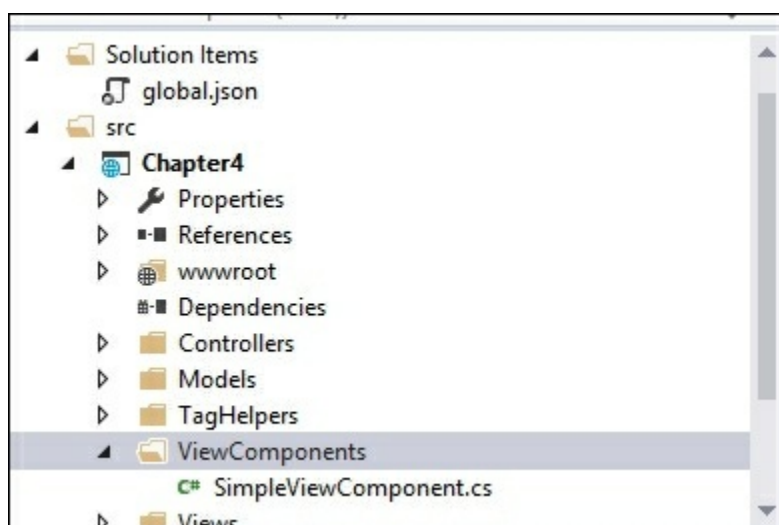
The `SimpleViewComponent` file that we created will look like the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Mvc;

namespace Chapter4.ViewComponents
{
  public class SimpleViewComponent :ViewComponent
  {
    public IViewComponentResult Invoke()
    {
      var data = GetSampleData();
      return View(data);
    }
    /// <summary>
    /// This is a simple private method to return some dummy data
    /// </summary>
    /// <returns></returns>

    private List<string> GetSampleData()
    {
      List<string> data = new List<string>();
      data.Add("One");
      data.Add("Two");
      data.Add("Three");
      return data;
    }
  }
}
```

We just have a couple of methods, one to populate the data and the other is the `Invoke` method where we will render the View.

Once you have created the `ViewComponent` attribute, you will need to include the `ViewComponent` namespace in the `Views\_ViewImports.cshtml` file so that the `ViewComponents` attributes can be available for all the Views. The highlighted code snippet in the following is added to the View:

```
@using Chapter4
@using Chapter4.Models
@using Chapter4.ViewComponents

@addTagHelper "*, Microsoft.AspNet.Mvc.TagHelpers"
```

We have created the `ViewComponent` and made them available to all of the Views. A simple action method in the `HomeController` file just returns the View:

```
public ActionResult Sample()
{
```

```
return View();
}
```

In the associated View, we can just invoke the component as shown in the following code snippet:

```
<p>
  This is a sample web page <br/>
  <div>
    @Component.Invoke("Simple")
  </div>
</p>
```

When you invoke the component, it will search in the following two folders:

- The `Views\<controller_name>\Components\<view component name>\<view name>` folder
- The `Views\Shared\Components\<view_component_name>/<view_name>` folder

The default View name of the View component is `Default`, which makes your file name for the View `Default.cshtml`. So, we will need to create the `Default.cshtml` file in `Views\Shared\Simple\Default.cshtml` folder, as shown in the following screenshot:
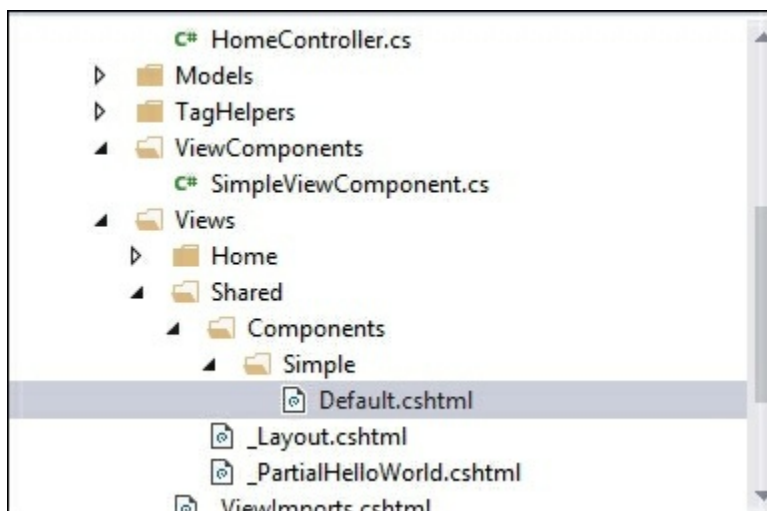
In the the View (`Default.cshtml` file) of the `ViewComponent` file, we are just iterating the items in the model and displaying them as an unordered list item, as shown in the following code:

```
@model IEnumerable<string>
```

```
<h3> Sample list</h3>
<ul>
  @foreach(var item in Model)
  {
    <li>@item</li>
  }
</ul>
```

When you run the application and access the URL
(`http://localhost:50132/Home/Sample`), you should see the following output:
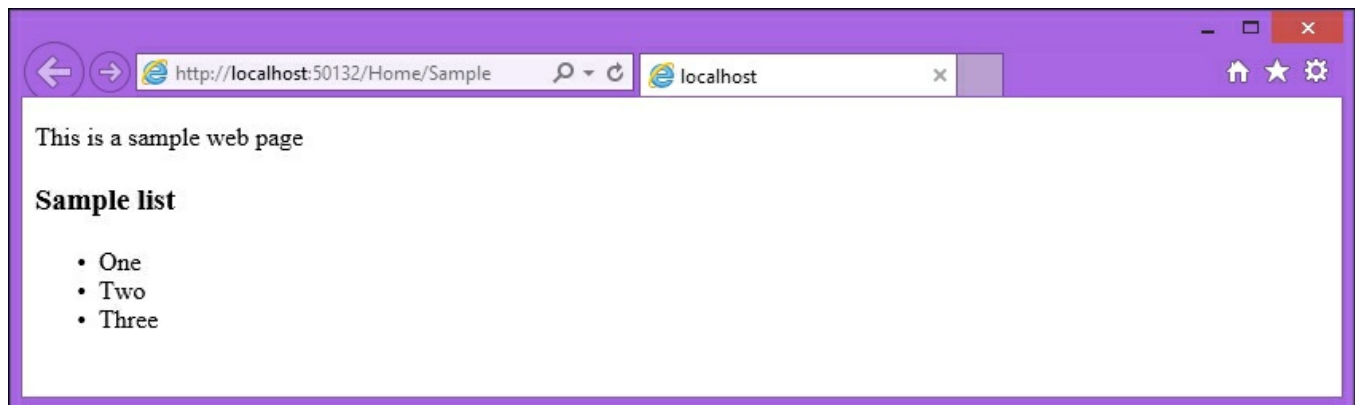
The first line, **This is a sample web page**, comes from the parent View file
(`sample.cshtml`) whereas the subsequent list comes from `ViewComponent` attribute.

The `ViewComponent` attributes are usually referred in the Views. However, if you
want to call the `ViewComponent` directly from your Controller, you can do so.

I have called the `Sample` action method to call the Simple `ViewComponent` directly
instead of calling it through some other View as follows:

```
public ActionResult Sample()
{
  return ViewComponent("Simple");
  //return View();
}
```
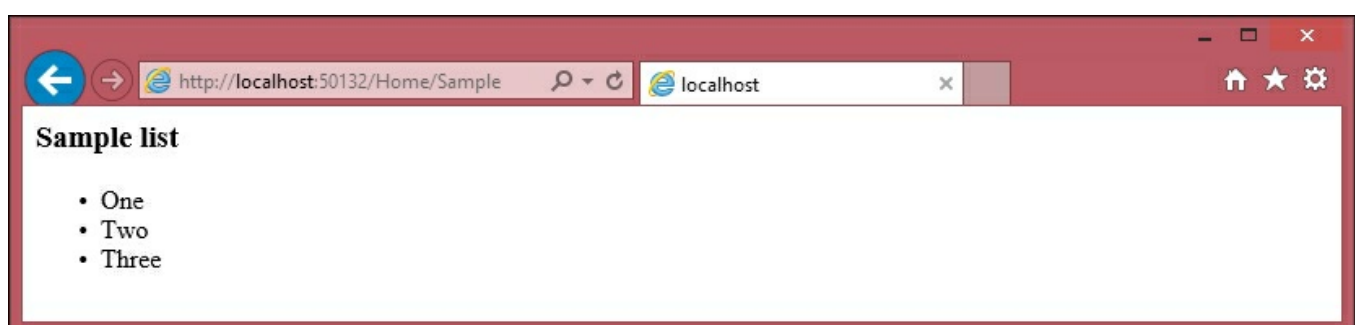
Thus, these `ViewComponents` have far more flexibility and features, such as Dependency Injection, when compared to old HTML Partial Views. This ensures `ViewComponents` are separately tested.

Tag Helpers

Tag Helpers are a new feature in ASP.NET Core; they help generate the HTML elements. In HTML helpers, we will write a C#/Razor code to generate the HTML. The disadvantage associated with this approach is that many frontend engineers will not know C#/Razor code. They work on plain HTML, CSS, and JavaScript. Tag Helpers look just like HTML code but have all the features of server-side rendering. You can even build your custom Tag Helper for your needs.

Let's take a look at how to use a Tag Helper. In order to use the Tag helper, you will need to install the `Microsoft.AspNet.Mvc.TagHelpers` NuGet package.

Open the **Package Manager Console** window by selecting **View** | **Other Windows** | **Package Manager Console**, as shown in the following screenshot:
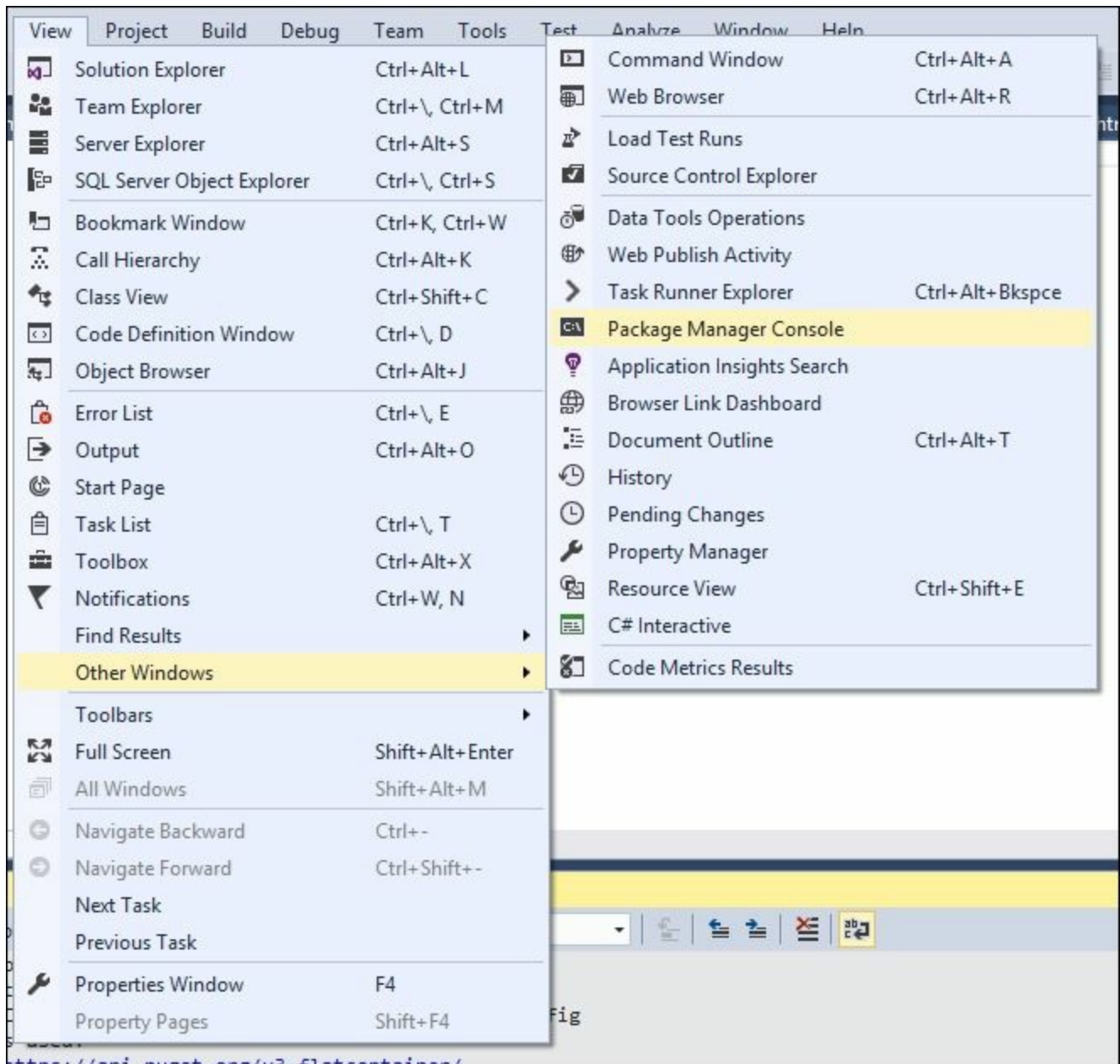
You can install `TagHelpers` methods by entering the following command in the **Package Manager Console** window, the following command:

```
Install-Package Microsoft.AspNet.Mvc.TagHelpers -Pre
```

The following response will appear when you've entered the command:
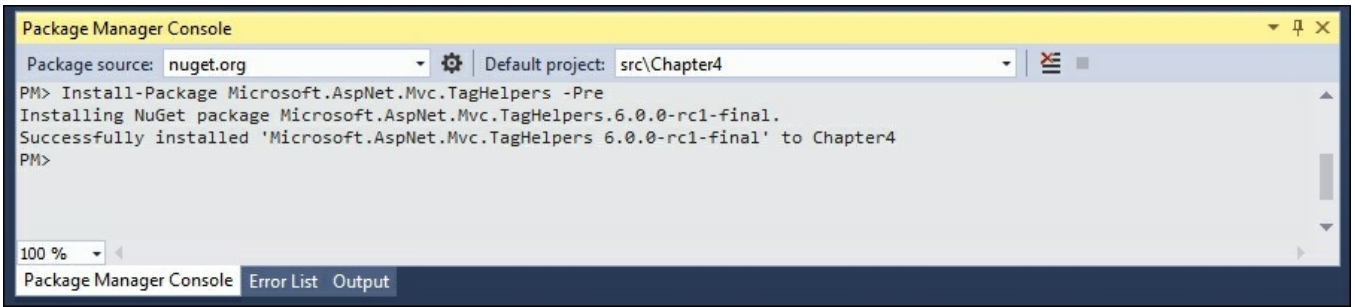
Once the `TagHelpers` package is installed, we will need to call `ViewImports` file, where we will add the `TagHelpers` directive so that Tag Helpers are available to our Views.

Right-click on the `Views` folder and select the **Add New Item** option from the **Context** menu; you'll see the following screen:
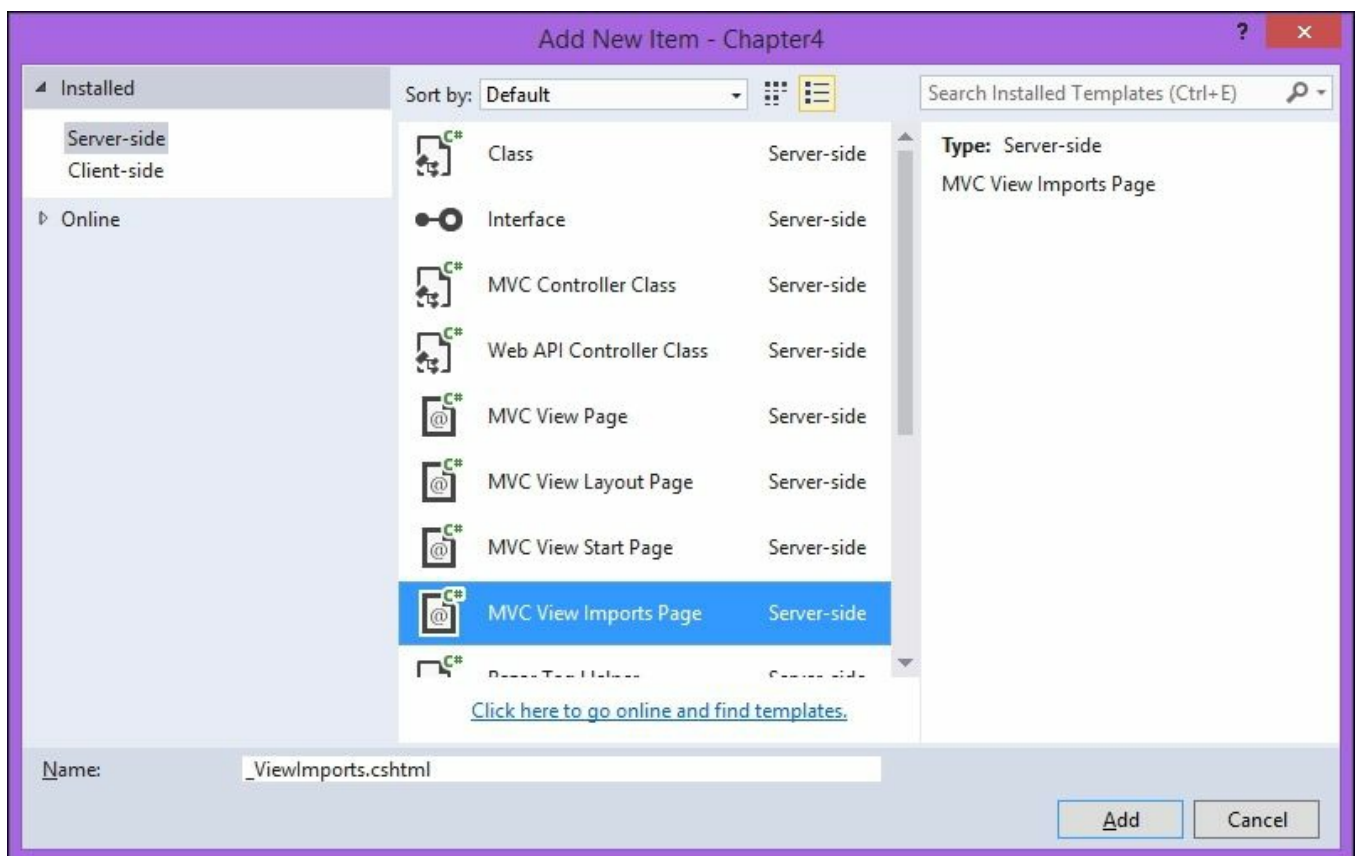
Add the following content to the `_ViewImports.cs` file. The first couple of lines tells ASP.NET MVC to include the necessary namespaces. The last line tells ASP.NET MVC to include all the `TagHelpers` available in `Microsoft.AspNet.Mvc.TagHelpers`. The first parameter indicates the name of

TagHelper. We have used *, which means that we may want to use all the Tag Helpers. The second parameter indicates the assembly where the `TagHelpers` will be available:

```
@using Chapter4
@using Chapter4.Models
@addTagHelper "*, Microsoft.AspNet.Mvc.TagHelpers"
```

As we are creating the `_ViewImports.cshtml` file directly under the `Views` folder, as shown in the following screenshot, the Tag Helpers will be available for all the Views:

Had we included the `_ViewImports.cshtml` file under the `Home` folder, the Tag Helpers would be available only for the Views under the `Home` folder.

Let's add a simple action method called `Index3` in the `HomeController` file, and in the associated View, we will use Tag Helpers as shown in the following code:

```
public IActionResult Index3()
{
  ViewBag.Title = "This is Index3";
  Person person = new Person();
  return View(person);
}
```

Add the corresponding View (`Index3.cshtml` file) for the `Index3` action method with the following code:

```
@model Chapter4.Models.Person
<form asp-controller="Home" asp-action="Index3">
  <table>
    <tr>
      <td><label asp-for="Name"></label></td>
      <td><input asp-for="Name" /></td>
    </tr>
    <tr>
      <td><label asp-for="Age"></label></td>
      <td><input asp-for="Age" /></td>
    </tr>
    <tr>
```

```
      <td colspan="2"><input type="submit" value="Submit" /></td>
    </tr>
  </table>
</form>
```

The following are a few things that you need to note in the preceding code, for the use of Tag Helpers:

- All the form elements look just like standard HTML elements with just a few changes in the attributes. This makes frontend developers work independently, without learning HTML/Razor code and thus more easily achieving the separation which concerns.
- The first line of the preceding view indicates the type of model data passed to the view from the Controller.
- The Form element has a couple of attributes named `asp-controller` and `asp-action` which represent Controller names and action method names respectively.
- The Label and input tag helpers are just like HTML elements, with just an additional `asp-for` attribute. The values for these attributes represent the model properties. You can take advantage of IntelliSense when entering the values for these attributes.

Creating custom Tag Helpers

ASP.NET Core provides many built-in Tag Helpers to help you create the necessary HTML elements for many scenarios. However, this process is not comprehensive and is exhaustive. Sometimes, you may want to make some changes in the generated HTML element, or you may want to create an HTML element with new properties or a new HTML element altogether. You are not restricted to using only the existing Tag Helpers in the ASP.NET Core application. You can create your own Tag Helper if the existing Tag Helpers do not suit your needs. Let's create a simple Tag Helper to create an e-mail link:

```
<a href="mailto:mugil@dotnetodyssey.com">
```

There are a couple of ways to create Tag Helpers to implement the `ITagHelper` interface or inherit the `TagHelper` class. The `TagHelper` class has a `Process` method that you can override to write your custom Tag Helpers. The `TagHelper` class also has the `TagHelperOutput` parameter, which you can use to write and generate the desired output HTML. So, it is preferable to create Tag Helpers by inheriting from the `TagHelper` class.

Our objective is to write a custom e-mail Tag Helper so that when someone uses that Tag Helper, which is `<email mailTo="mugil@greatestretailstore.com"> </email>`, it should be converted to the following line of code:

```
<a href="mailto:mugil@greatestretailstore.com">Drop us a mail</a>
```

The following are the steps that need to be performed to create the custom Tag Helper in the ASP.NET Core application.

Create a folder called `TagHelper` and add a new item named the `EmailTagHelper.cs` file. By convention, all Tag Helpers class should end with `TagHelper`, even though we can override this convention.
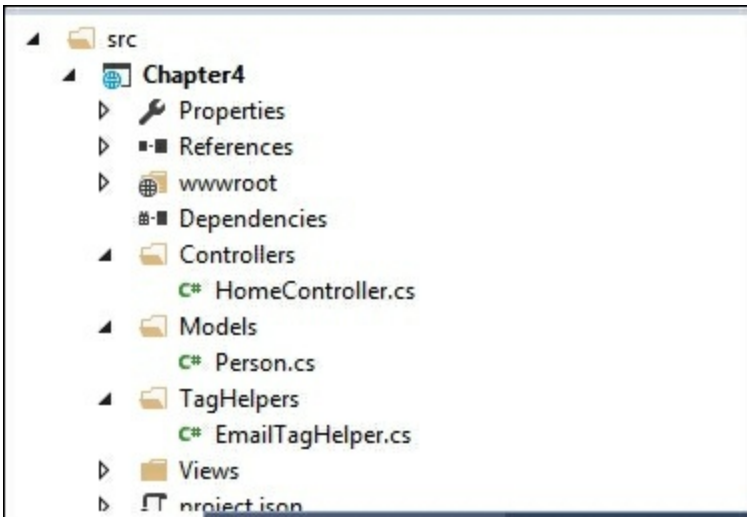
Once you have created the file, you will need to override the `Process` method to generate the desired HTML output:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Razor.TagHelpers;

namespace Chapter4.TagHelpers
{
  public class EmailTagHelper : TagHelper
  {
    public override void Process(TagHelperContext context,
TagHelperOutput output)
    {
      string emailTo=
context.AllAttributes["mailTo"].Value.ToString();
      output.TagName = "a";
      output.Attributes["href"] = "mailto:" + emailTo;
      output.Content.SetContent("Drop us a mail");
    }
  }
}
```

The parameters used in the preceding code are explained as follows:

- The `context` parameter will give you all the information that you supply at Tag Helper. For example, in the `<emailmailTo="mugil@greatestretailstore.com"></email>` Tag Helper, you can get the `mailTo` attribute and its associated value from the `context` parameter. In the first line of the preceding `Process` method, we will get the

`mailTo` attribute value and use that value to create an attribute in the generated HTML (anchor tag).

- The `output` parameter is of type `TagHelperOutput`, which is used to generate the desired HTML output.
- The `output.Content.SetContent` parameter will set the text that is to be displayed for the anchor tag.

We have created the e-mail Tag Helper. Now, we have to make it available to our Views so that we can make use of that Tag Helper in our Views. Edit `Views\_ViewImports.cshtml` to include the namespace of the `TagHelpers` and add the associated `TagHelpers`. In the following `_ViewImports.cshtml` file, we have added the content highlighted in bold:

```
@using Chapter4
@using Chapter4.Models
@using Chapter4.TagHelpers
@addTagHelper "*, Microsoft.AspNet.Mvc.TagHelpers"
@addTagHelper "*, Chapter4"
```

The "`*`" symbol in the following line tells the view engine to include all the TagHelpers in the `Chapter4` namespace:

```
@addTagHelper "*, Chapter4"
```

You can only specific `TagHelpers`, For example, the following line will include only the `EmailTagHelper` so it is available for our Views:

```
@addTagHelper "Chapter4.TagHelpers.EmailTagHelper, Chapter4"
```

Let's create a simple action method in our Home Controller. In the view of the associated action method, we will use the e-mail Tag Helper:

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

The following is the view of the preceding `AboutUs` action method:

```
<h3>About Us</h3>
We are one of the biggest electronics retail store serving millions
of people across the nation. blah.blah. blah <br/>

If you want to hear great offers from us
<email mailTo="mugil@greatestretailstore.com"></email>
```

When you run the application and access the

`http://localhost:50132/Home/AboutUs` URL, you will see the following output:
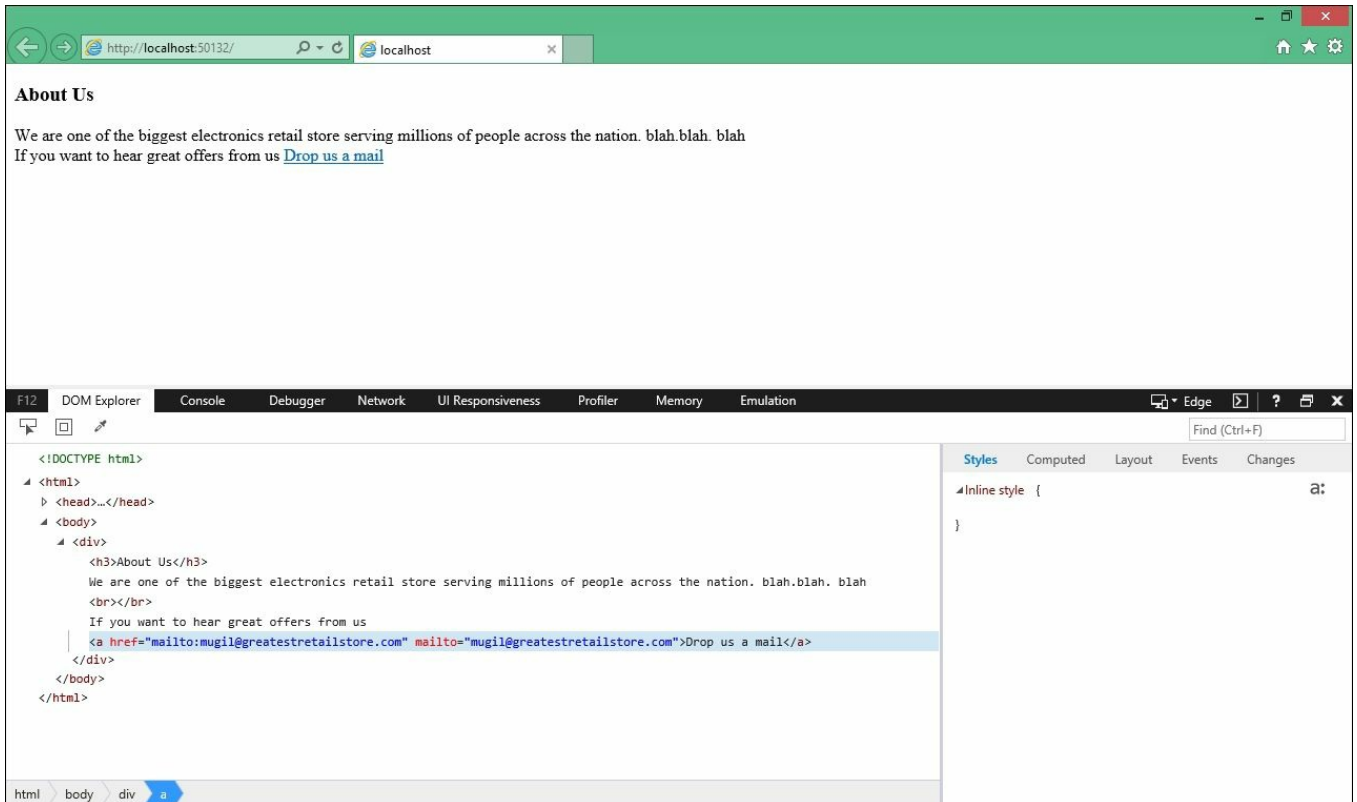
Here, we created an anchor tag with the `mailto` attribute and the email value as the `href` attribute value.

I have opened the **Developer Tools** window (Press *F12* to do this and select the **DOM Explorer** tab) to see the generated HTML.

Summary

In this chapter, you learned what a View engine is and how to build a View using the Razor view engine. We also discussed different programming constructs that you can make use of in Razor to produce the desired HTML output. Then, you learned about Layout and how to provide a consistent site structure across all of the pages in your ASP.NET MVC application. Later, we discussed how to promote re-usability using Partial Views with an example. Finally, you learned how to use Tag Helpers to produce clean HTML.

Chapter 5. Models

Data is the heart of every application. A user enters data into the application, edits the entered data, and searches the data. We can even say that an application that we build is just an interface for the operations that we perform on the application data. So, it is absolutely necessary for any framework to provide a mechanism to handle data operations easier and more manageable. Models in ASP.NET MVC are used to represent the business domain data.

In this chapter, you'll be learning about the following topics:

- Models and their purpose
- Creating a simple model and using it in the controller and views of the ASP.NET MVC application
- Creating a model specific to a View model
- Data flow in an ASP.NET MVC application in the context of models and `ViewModels`
- Purpose of the Entity Framework along with its features and benefits
- Adding, updating, and deleting data using the Entity Framework
- Using the Entity Framework in ASP.NET MVC applications

Models

Models are simple **POCO** (**Plain Old C# Objects**) classes representing your business domain data. For an e-commerce business, model classes would be `Product`, `Order`, and `Inventory`. If you are building an application for a university, model classes would be `Student`, `Teacher`, and `Subject`. Models represent the business domain data in your application and they are not aware of the underlying database that is being used in your application. In fact, you don't even need a database to work with models.

They can represent the data stored in an XML file or CSV file or any other data in your application. Having said that, these models could be used to interact with your database (in most cases) but they don't have any dependency to the database.

The following steps describe how to create an ASP.NET Core application that uses Models:

1. Make sure to create an ASP.NET 5 application with an empty template. Install the ASP.NET Core `NuGet` package and configure this, as discussed in an earlier chapter.
2. Create a `Controllers` folder and create a `HomeController` with a single `Index` action method.
3. Create the following folder/files for the `Views` model:
   - `Views`: This folder is inside your project.
   - `Views\_ViewStart.cshtml`: This identifies the name of the `Layout` file.
   - `Views\Shared` folder: This folder holds all the shared View components for your application.
   - `Shared\_Layout.cshtml`: This file identifies what your web application structure should look like.
   - `Views\Home` folder: This folder contains all of the Views of your `HomeController`.
   - `Views\Home\Index.cshtml`: This is the view corresponding to the `Index` action method of `HomeController`.

Now, we have created an ASP.NET Core application with Controllers and Views.

Let us create a `Models` folder in our application; this folder will contain all of your model files. In a real world application, this folder and the respective model files would reside in separate projects. For the sake of simplicity, we are keeping the `Models` folder and its files in the same project.
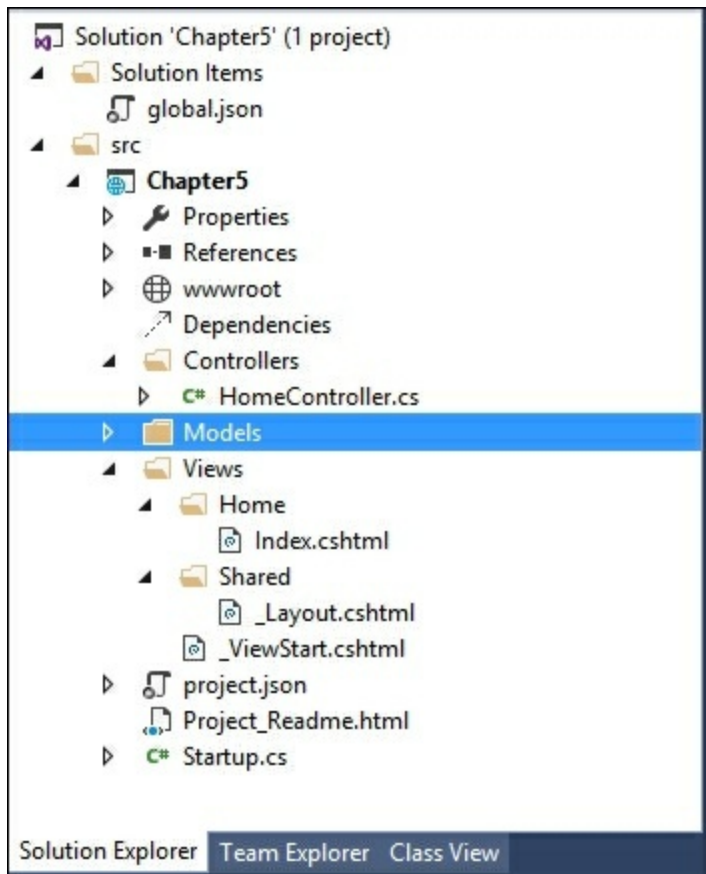
Let us create a simple model class `Product` model, in the `Models` folder:

```
public class Product {
  public int ProductId { get; set; }
  public string Name { get; set; }
  public decimal Price { get; set; }
}
```

This `Product` model class is no different from any other C# class and contains a few properties about the product.

Update the `Index` action method in `HomeController` to use the `Product` model, as shown in the following code snippet. We are building the model data and passing the model data to the View so that it can be shown to the users. However, it is *NOT* recommended to build the Model data in the controller's action methods as it violates the separation of concerns. For the sake of simplicity only, we are building the Model data in an action method.

```
public IActionResult Index() {
  /* Build the products model. It is NOT RECOMMENDED to build
models in Controller action methods  like this. In real world
appication, these models and the    respective Data Access
Layer(DAL) would  be in separate projects. We are creating it here
```

```
to make things    simpler to explain */
  List<Product> Products = new List<Product> {
    new Product {
      Name = "Mobile Phone",
      Price = 300
    },
    new Product {
      Name = "Laptop",
      Price = 1000
    },
    new Product {
      Name = "Tablet",
      Price = 600
    }
  };
  return View(Products);
}
```

Update the corresponding `Index` View method to use the Model data loop through each product and show it as an unordered list item. The `@model` in the first line represents the Model metadata; the type of data being passed to the View. The Model in the `for…each` loop represents the actual data itself, a list of products in our case:

```
@model List<Chapter5.Models.Product>

<ul>
  @foreach (var Product in Model) {
    <li>@Product.Name</li>
  }
</ul>
```

When you run the application, you'll get the following output:
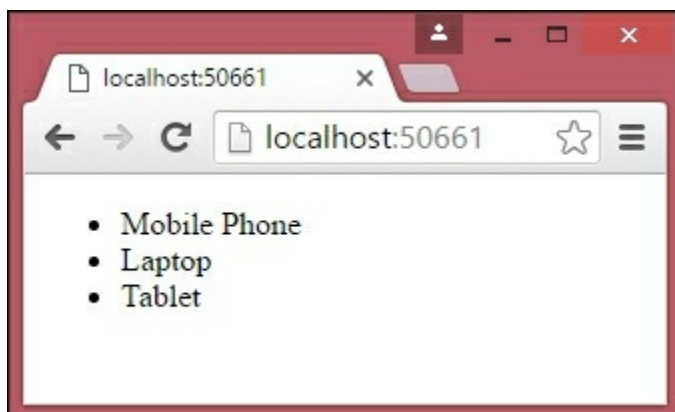
We have successfully created a Model and have used it in our Controller and View.

Let us create a comparatively complex Model class, `Order` (`Order.cs` in the `Models` folder), which contains a list of products and their total amount:

```
public class Order {
  public int OrderId { get; set; }
  public List<Product> Products { get; set; }
  public decimal Total { get; set; }
}
```
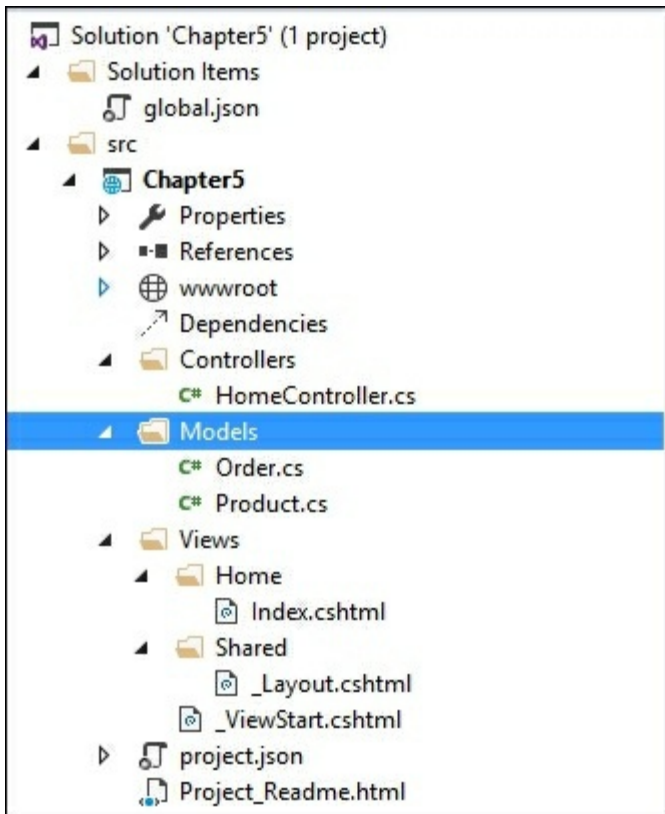
Now, we have to update the `Index` action method to use the `Order` model. Once we build the list of products, we are assigning the products list to the `Order` property and calculating the total cost of the order. These calculations would usually be done as part of the business layer. Again, for the sake of simplicity, we are building the data Model and calculations here in the action; this should never be the case in a real world application.

The code highlighted in bold is the changes that we have made in the action method:

```
public IActionResult Index() {
  /* Build the products model. It is NOT RECOMMENDED to build
models in Controller action methods  like this. In real world
appication, these models and the     respective Data Access
Layer(DAL) would  be in separate projects. We are creating it here
to make things     simpler to explain   */
```

```
    List<Product> Products = new List<Product> {
      new Product {
        Name = "Mobile Phone",
        Price = 300
      },
      new Product {
        Name = "Laptop",
        Price = 1000
      },
      new Product {
        Name = "Tablet",
        Price = 600
      }
    };

    Order order = new Order();
    order.Products = Products;
    order.Total = Products.Sum(product => product.Price);

    return View(order);
}
```

The View is updated to accommodate the Model changes. Model metadata (`@model`) is changed to indicate that the Order information is passed to the View instead of the list of products.

Then, we are showing the list of products in table format. Please note that all of the Model data (`Order` object and its properties, in this case) can be accessed through the Model. For example, the `Products` class can be accessed through `Model.Products` and the value of the `Total` can be obtained through `Model.Total`:

```
@model Chapter5.Models.Order

<table border="1">

  <tr>
    <th>Product Name</th>
    <th>Price</th>
  </tr>

  @foreach (var Product in Model.Products){
    <tr>
      <td>@Product.Name</td>
      <td>@Product.Price</td>
    </tr>
  }
  <tr>
    <td><b>Total</b></td>
    <td><b>@Model.Total</b></td>
  </tr>
```

```
</table>
```
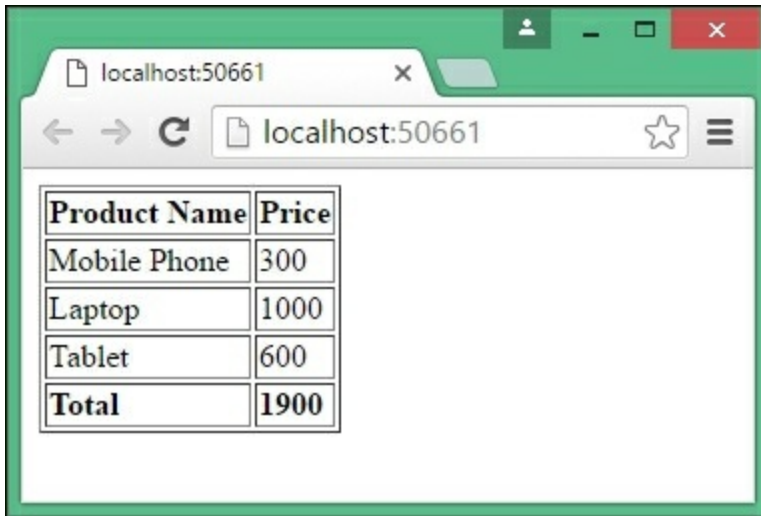
When you run the application, you'll see the following output:

Models specific to a View component

There are scenarios where you might want to update only a few properties in a large Model or you might want to create a new Model based on a few models. In such scenarios, it is better to create a new Model specific to the View.

For example, let us assume that we are building a screen where you update the price of the product. This simple screen may contain only three properties—product ID, product name, and price of the product. But the product's Model may contain more than 30 properties to hold all details of the product such as manufacturer, color, size, and other properties. Instead of sending the complete Model with all the properties, we can create a new Model specific to this view with only a few properties—ID, Name, and Price.

Note on ViewModels

The ViewModels are entities where when you update the Model, your View would get updated automatically and vice versa. In many online articles and even in some books, they are referring to *ViewModels* when they are actually trying to mean *Models specific to the View*.

In ViewModels, binding is two ways—when you update either the Model or the View, the other one would get updated automatically. Let us consider a simple example; you have a form with various fields on the left-hand side and print preview on the right side. In this case, whatever you type in real time in the form will be reflected immediately on the right side. In such cases, you can use pure View models when you type, your ViewModel would be updated and that `ViewModel` would be consumed in the right-hand side print preview. These pure ViewModels are being used in advanced JavaScript frameworks such as **Knockout** or **AngularJS**.

In *Models specific to the View,* we are binding in only one way from the Model to the View. Here, we are sending a Model specific to the View instead of the generic Model (which represents a business domain class).

However, in this book, we will be referring to *Models specific to View* as `ViewModel` for brevity. Unless otherwise specified, you should read all ViewModels as *Models specific to View*. So, I am making the same mistake made by other authors ☺.

Data flow with respect to a Model

The following block diagram shows the data flow in an ASP.NET MVC application:
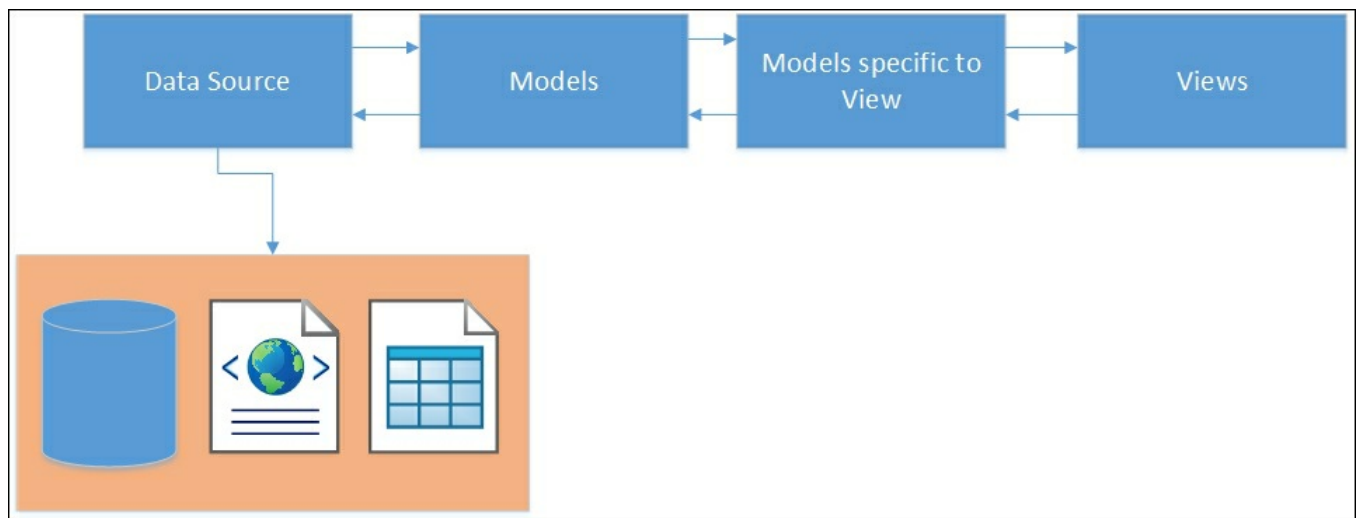
Data Source represents your application data. The application data could reside anywhere—from full-fledged RDBMS such as SQL servers to simple Excel spreadsheets, or anything in between.

Models represent the business domain data for your application and are independent of the data source being used. The same model could be used with different data sources.

We can use the **Model as-is in our views** to get data or to present it. But in some views, you might not need all the properties of the model. So, instead of sending the entire Model to the View, we create models specific to the View and use them in our View. This makes things simpler.

The following is the high-level sequence of events that happens when you store or retrieve a record in ASP.NET Core using the Model:

1. Users enter the data in a form (created using Views) in the application. The fields in the form do not need to represent the complete model as we need only a few properties in the Model.
2. The entered data is passed to the controller where Model binding happens. Model binding is the process where the data entered in the View gets mapped to the Model or ViewModel.
3. If the data is received in the ViewModel, then we would be converting the ViewModel to the Model.

4. Finally, the Model data is stored in the data source.

Till now, we have been handling only in-memory data in our application. In almost all real world applications, some form of the database will be used for data storage, access, and retrieval. In the next section, we will discuss the Entity Framework (ORM framework), which makes data access simpler from a .NET application.

Model binding

Model binding is the process of mapping the Model data coming from the View to the ViewModel parameter of the action method in the Controller.

Let us consider a simple form with a couple of form fields—`Name` and `EmailID`. On the submission of the form, these values would be mapped to the `ViewModel` object of the action method of the Controller. Model binding takes care of this mapping. The Model binder looks for a match in the form fields, query strings, and request parameters.

In the preceding example, any class with these properties would be picked up by ModelBinder without any issues.

As the following `Person` class contains the `Name` and `EmailID` properties, the model binder would not complain about using this model for mapping the entered values in the form:

```
public class Person {
  public string Name { get; set; }
  public string EmailID { get; set; }
}
```

The following code snippet shows how to use the `Person` class in the action method:

```
public ActionResult Add(Person p) {
  return View();
}
```

The Entity Framework

The Entity Framework is the **Object Relational Mapping (ORM)** framework that enables developers to work on domain-specific objects directly for data access instead of working on database queries. This reduces a lot of the code complexity in the data access layer in the application.

Before discussing the Entity Framework and its features, let us pause for a moment and think about the steps that we follow when we try to save some information to the database when using ADO.NET:

1. Construct the business domain object.
2. Create a connection to your database.
3. Open the connection.
4. Create a command object along with the command type.
5. Add the properties of your business domain object to the parameters of the command object.
6. Execute the command that saves the data into the database.

We have to follow the previously mentioned six steps for common operations such as saving a piece of data into the database.

If you are using an ORM framework such as the Entity Framework, you just need three steps:

1. Construct the business domain object.
2. Create the `DbContext` class for your business domain object. The instance of the `DbContext` class represents the session with the database.
3. Save it to the database using the instance of the `DBContext` class.

You might wonder how that is possible.

As a matter of fact, in the background, the Entity Framework creates a connection to the database and executes the query to save the business domain object to the database. To make it simple, the Entity Framework writes all the data access code for you so that you can concentrate on achieving the business functionality of the application rather than writing the database layer code.

The Entity Framework is independent of ASP.NET MVC

As discussed earlier, the Entity Framework is an ORM framework for accessing data and is independent of ASP.NET MVC. The Entity Framework could be used in **Windows Communication Foundation** (**WCF**) services, Web API services, and even in console applications. You could use the Entity Framework in any type of application and make use of it to access data using objects. The concepts and the functionalities of the Entity Framework remain the same, irrespective of the type of application that you use it with.

Now, we are going to use the Entity Framework with the console application. This allows us to concentrate on the task at hand and demonstrate the functionalities of the Entity Framework instead of working on the boilerplate code of the ASP.NET Core application. In a later part of this chapter, we will integrate the Entity Framework with the ASP.NET Core application.

The latest version of the Entity Framework for the SQL server is 7.0.0 and it is still in beta at the time of writing this book. EF7 (Entity Framework 7) brings significant changes when compared to its previous version (Entity Framework 6). However, EF7 is the recommended version when building ASP.NET 5 applications and hence we will be using this version in this book.

Note

We need a database to explain many of the features of the Entity Framework. Please install SQL Server 2014 Express on your PC before continuing further. Step by step instructions for installing SQL Server 2014 Express and SQL Server Management Studio is given in *Appendix A*.

Creating console applications with the Entity Framework

Follow these steps to create a simple console application:

1. Select **File** | **New Project** and select **Console Application**.
2. Name the project and click on **OK**.
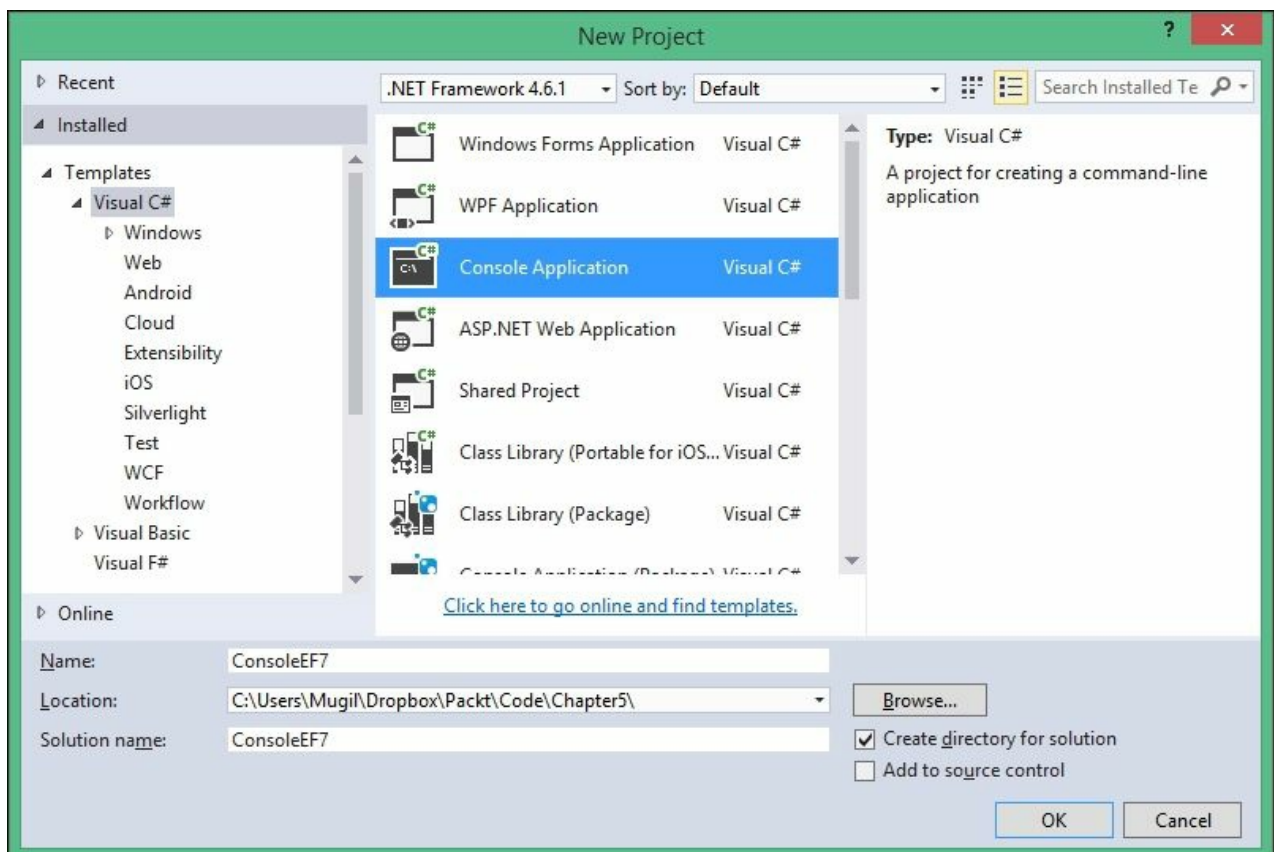
Installing the Entity Framework 7 NuGet package

There are two ways to install any `NuGet` package in your application:

- Using the `NuGet` Package Manager
- Using Package Manager Console

Using the NuGet Package Manager

People who prefer graphical interfaces can use this option:

1. Right-click on the console project and select **Manage NuGet Packages** from the context menu:
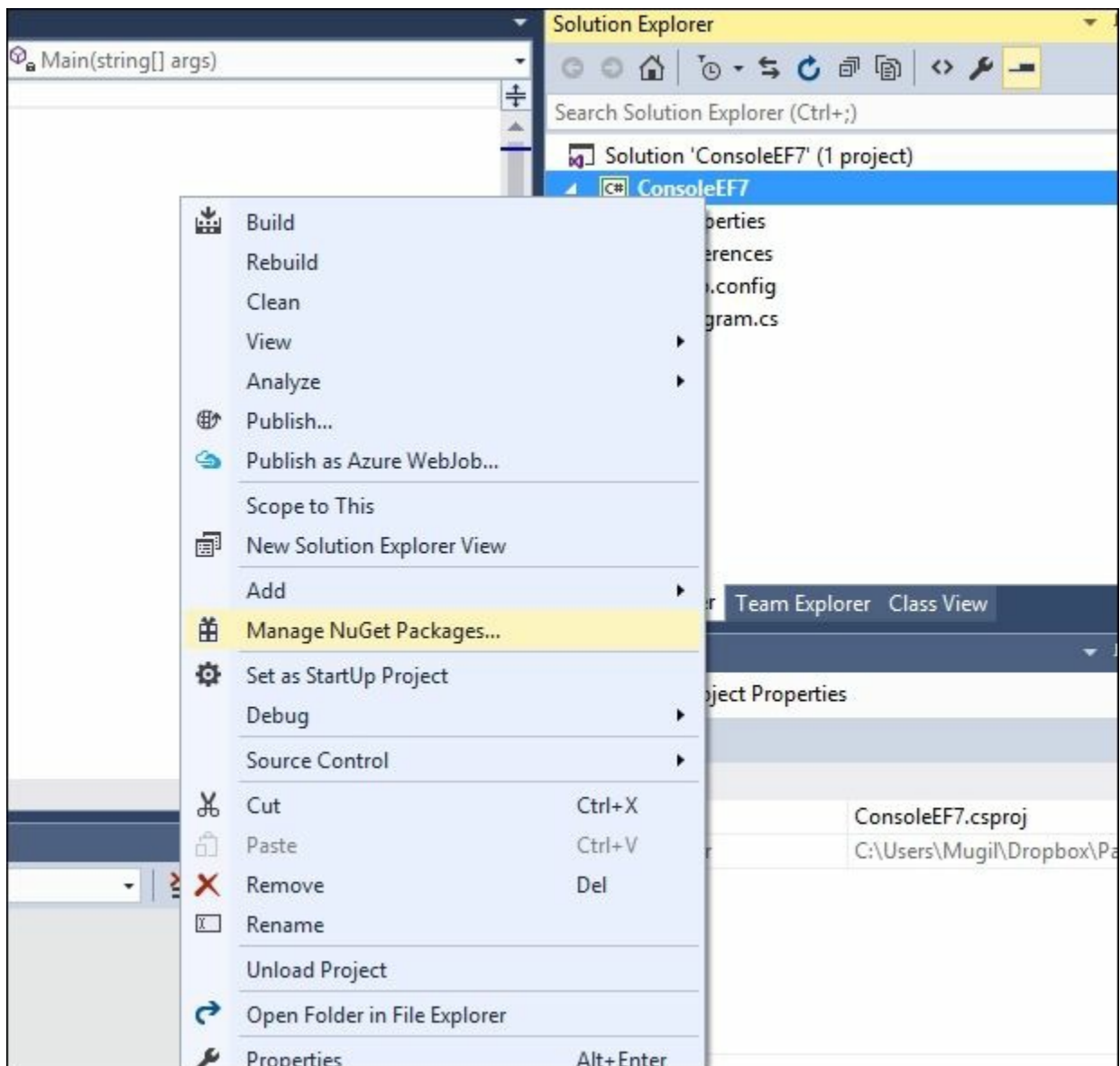
2. Search for `EntityFramework.MicrosoftSqlServer` in the `NuGet` package and make sure the **Include prerelease** checkbox is checked. Click on **Install** once you select **EntityFramework.MicrosoftSqlServer** and select **Latest pre-release 7.0.0-rc1-final** (at the time of writing this book). You can select any latest version of Entity Framework 7:
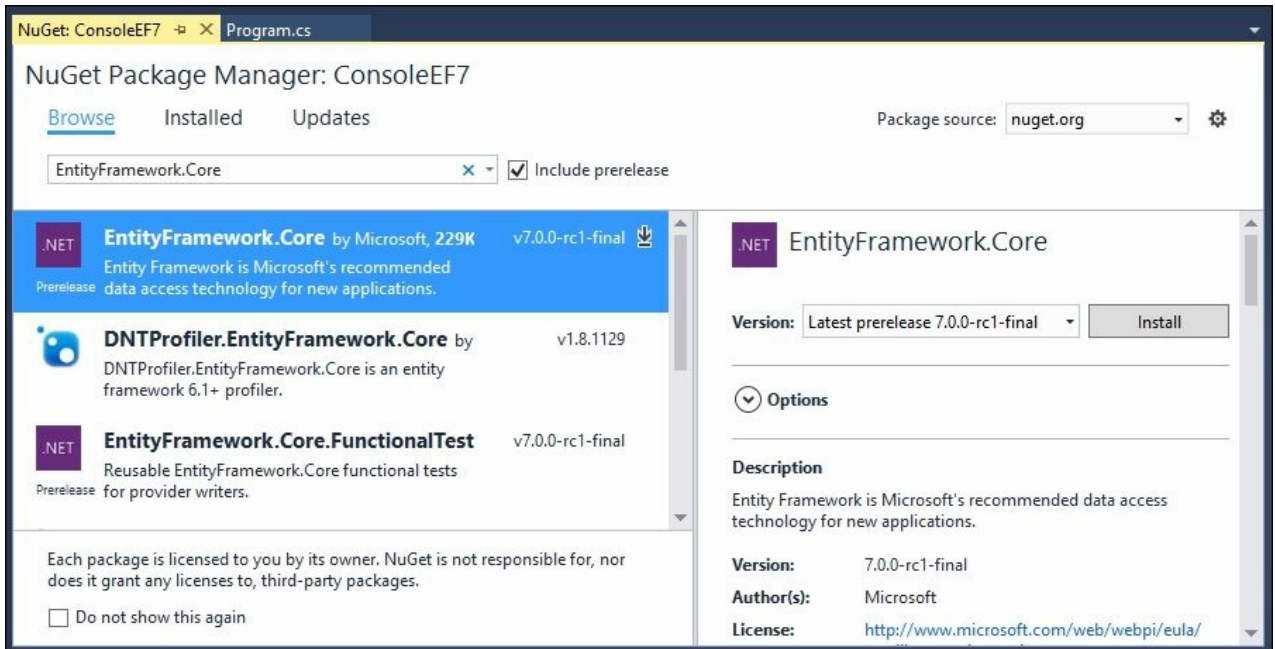
3. Once you click on **Install**, the `NuGet` package manager will ask you to review the changes. Click on **OK**:
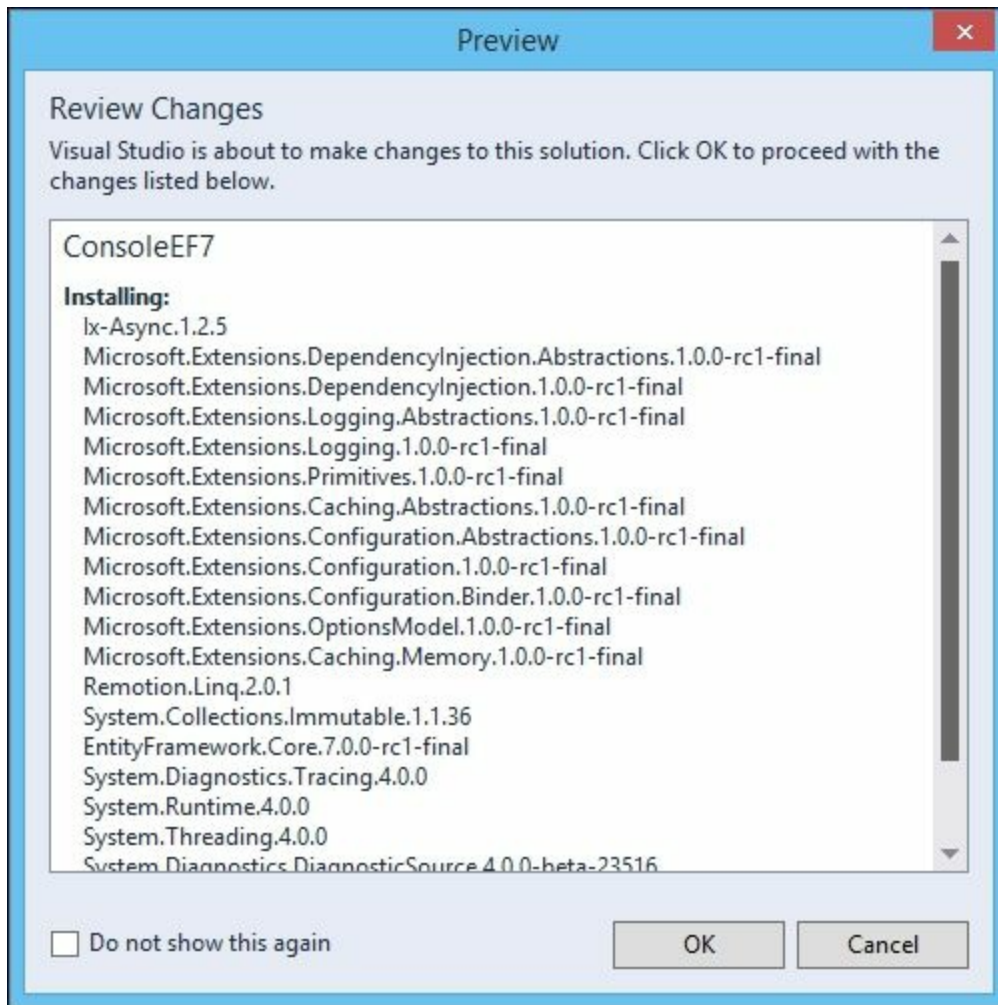
4. Click on **I Accept** in the **License Acceptance** window:
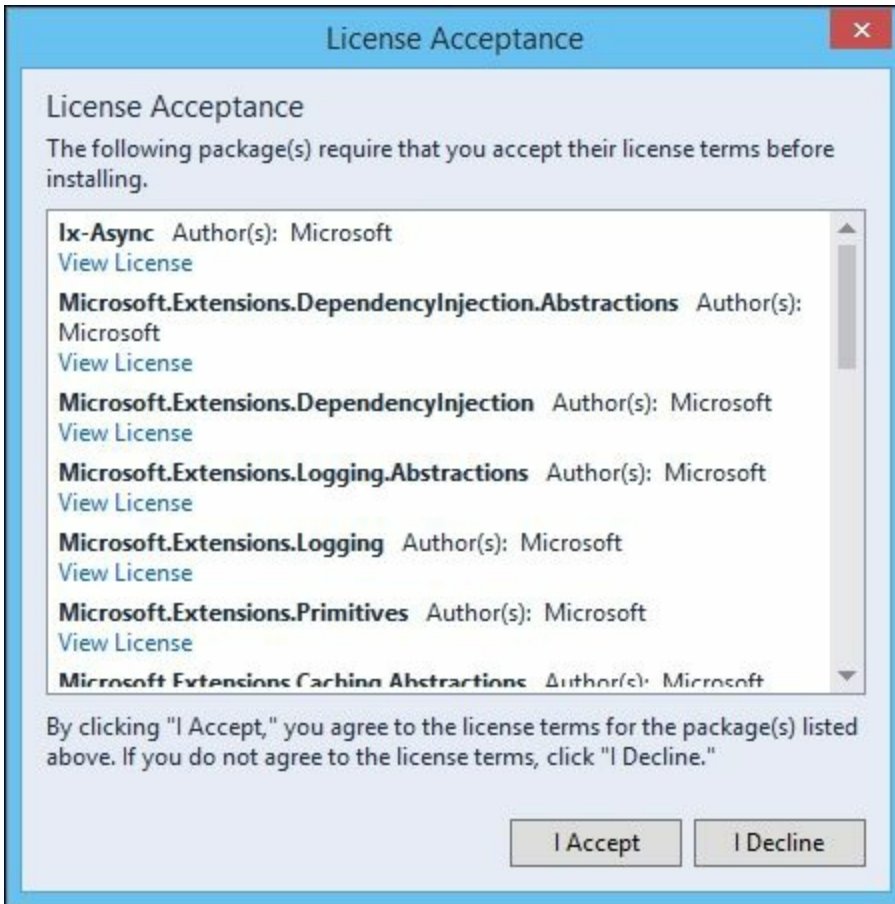
5. Once you click on **I Accept**, it will install the Entity Framework with all its dependencies. In the **Output** window, you will get the **Finished** message once the installation is complete:
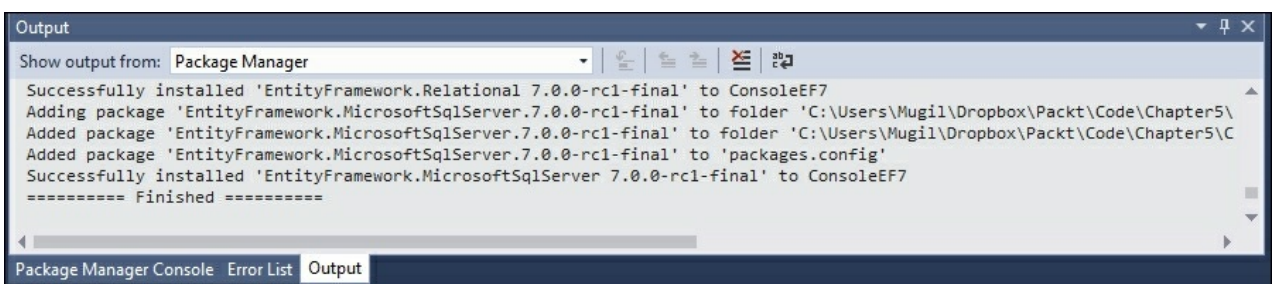
Using the Package Manager Console

To install the `NuGet` package using the Package Manager Console, follow these steps:
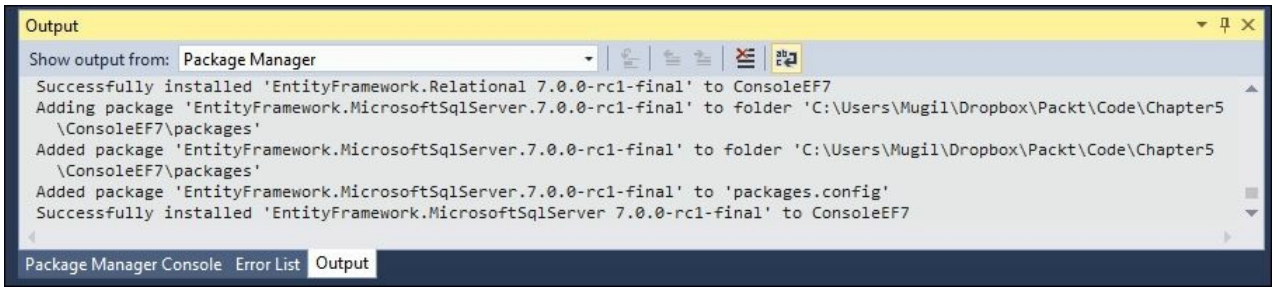
1. Open the **Package Manager Console** window by selecting the menu option **View** | **Other Windows** | **Package Manager Console**.
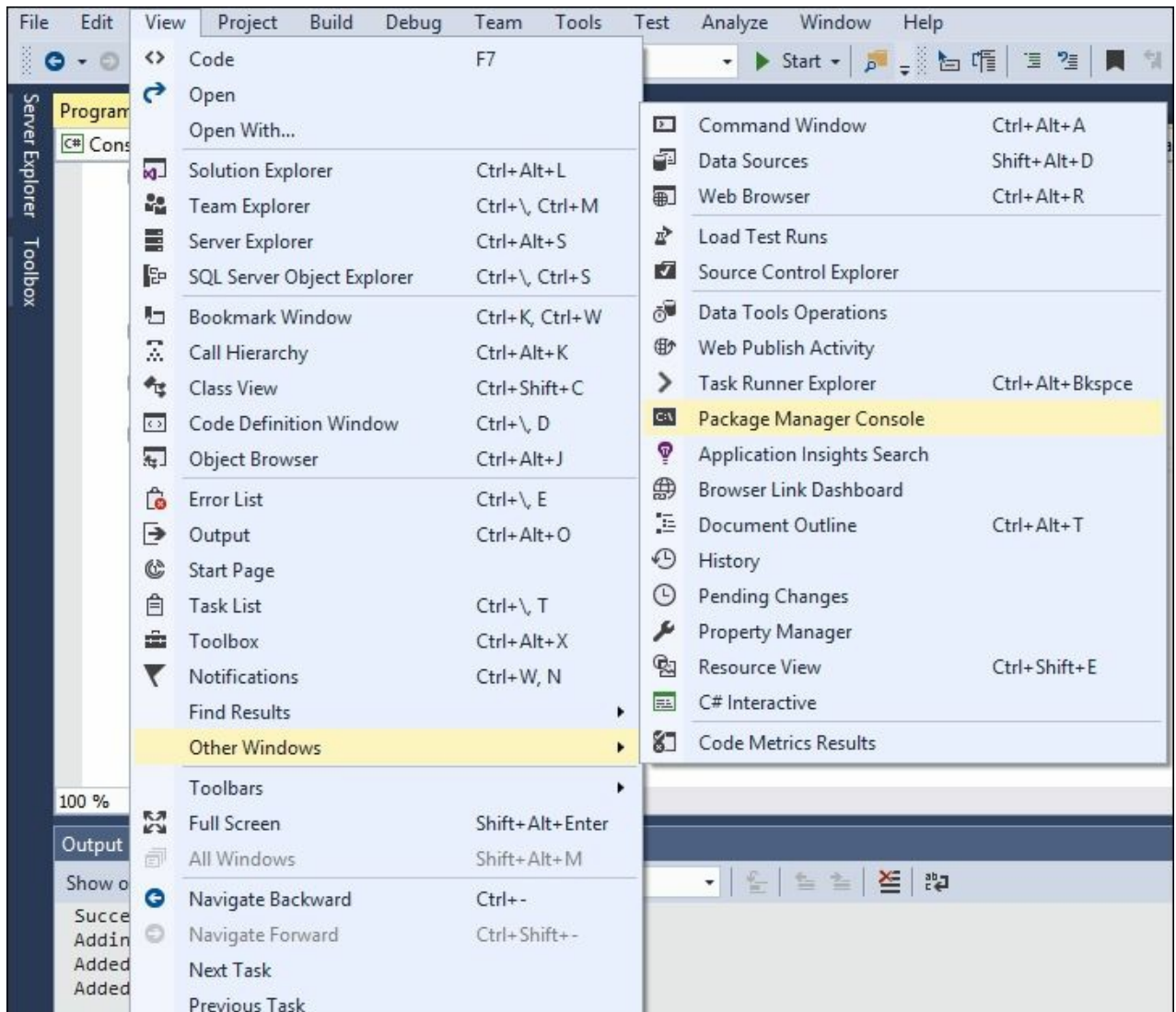
2. Type `Install-Package EntityFramework.MicrosoftSqlServer - Pre` in the **Package Manager Console** window as shown in the following screenshot:
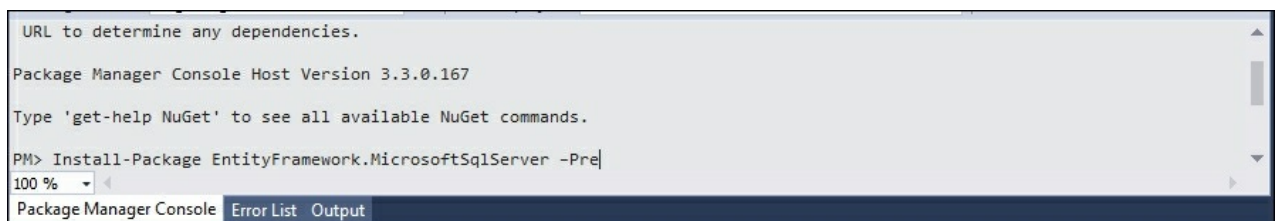
3. Once the installation is complete, a message, **Successfully installed 'EntityFramework.MicrosoftSqlServer 7.0.0-rc1-final'**, will be shown:

Installing Entity Framework commands

We need to install the Entity Framework commands package in order to perform migration activities. Migration includes the creation of a database and its associated tables. Any changes in the schema will also be taken care of by migration:

As discussed earlier, we need to follow three steps in order to interact with the database when we are using the Entity Framework:

1. Create the `Model` classes.
2. Create the `DbContext` class for your business domain object. The instance of the `DbContext` class represents the session with the database.
3. Construct the business domain object and save it to the database using the instance of the `DBContext` class.

Let us discuss each of the preceding steps in details and try to save an object to the database.

Creating Model classes

The `Model` classes are simple POCO objects, which can be used with the Entity Framework.

Let us create a POCO class for our business domain object, the `Employee` class in our case. I have created a new file named `Employee.cs` in our console application with the following content. This `Employee` class contains a few properties of an employee and has no special properties or fields to make it work with the Entity Framework.

Let's take a look at the following code snippet:

```
public class Employee {
  public int EmployeeId { get; set; }
```

```
    public string Name { get; set; }
    public decimal Salary { get; set; }
    public string Designation { get; set; }
}
```

By convention, if the property name is `Id` or `ClassName+Id`, it will be considered as a primary key by Entity Framework while creating the database table.

Properties with string data types will be created as fields of the type `nvarchar(max)`. However, we can override this behavior by using annotations, which we will be discussed later.

Creating the DbContext class

The instance of the `DbContext` class represents the session to the database and this `DbContext` class does most of the heavy lifting of your data access for your application.

Create a new class by the named `EmployeeDbContext` with the following content:

```
using Microsoft.Data.Entity;
using System.Configuration;

namespace ConsoleEF7 {
  public class EmployeeDbContext : DbContext{
    public DbSet<Employee> Employees {get; set;}

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder) {string connectionString =
ConfigurationManager.ConnectionStrings
["SqlServerExpress"].ConnectionString;
      optionsBuilder.UseSqlServer(connectionString);
      base.OnConfiguring(optionsBuilder);
    }
  }
}
```

Configure it using `App.Config`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
    sku=".NETFramework,Version=v4.6.1" />
  </startup>
  <connectionStrings>
    <add name="SqlServerExpress" connectionString="Data Source=
MUGIL-PC\SQLEXPRESS;Initial Catalog=EF7Console;Integrated
Security=True"/>
```

```
    </connectionStrings>
</configuration>
```
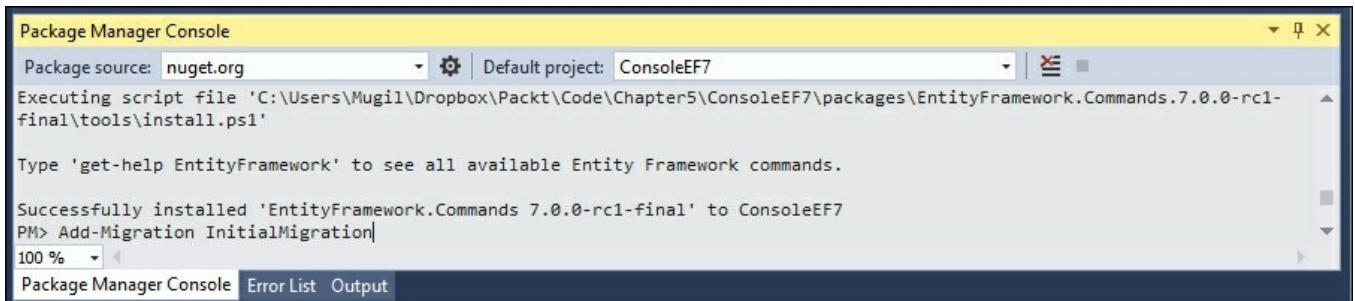
There are a few things to be noted in the preceding code snippet:

- Include the `Microsoft.Data.Entity` namespace as the `DbContext` class available in this namespace. Our connection string is available in the `App.Config` file. In order to read the contents of the `App.Config` file, we are including the `ConfigurationManager` class in `System.Configuration`.
- In order to use the `DbContext` API, a class has to be created which inherits from the `DbContext` class so that we can access methods of the `DbContext` API. We have created the `EmployeeDbContext` class which was inherited from `DbContext` class.
- `DbSet` is a class which allows operations of the Entity Framework to be performed for a given Entity type. We need to create the `DbSet` object for each of the Entity types that we use in our application. In this example, we are using only one `DbSet` object as we are working with the `Employee` class.
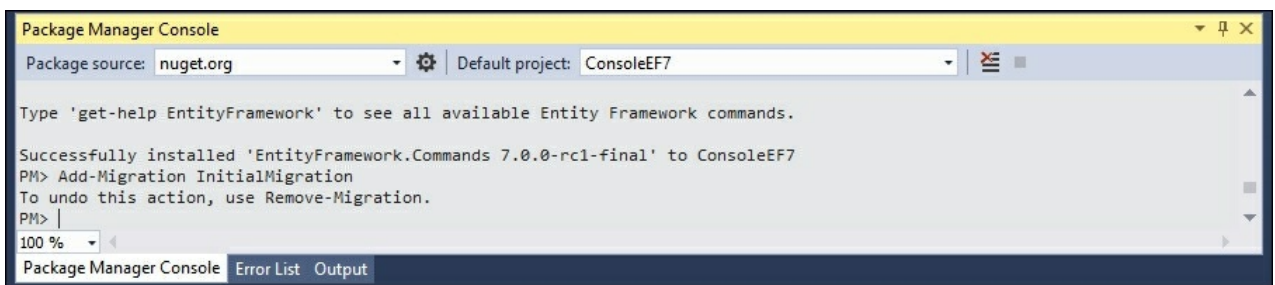
Create a migration

Migration is the process of recording all the changes of your database. `Add-Migration` is the Entity Framework command for adding migration:

1. Once you add the migration, you can revoke the changes by executing the `Remove-Migration` Entity Framework command.

   o This is what the migrations directory looks like:
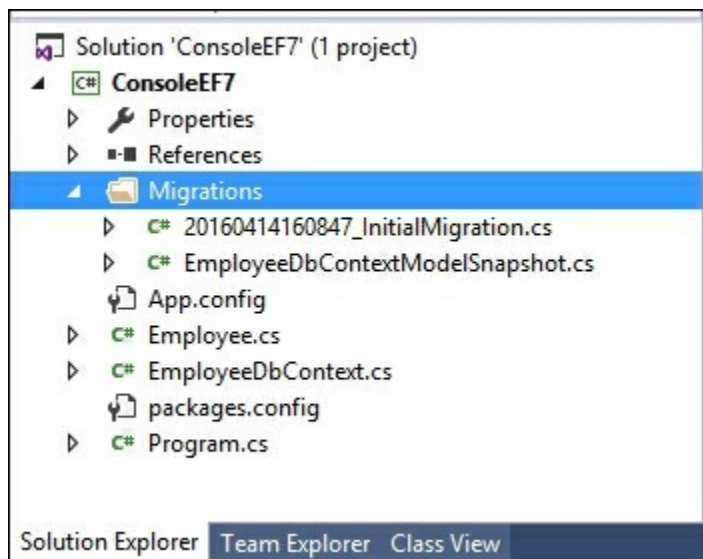
2. Update the database by issuing the Entity Framework command `Update-Database`, which updates the database tables as per the information available in the migration. As we have installed the `EntityFramework.Commands` package earlier, these commands will be available for the application:
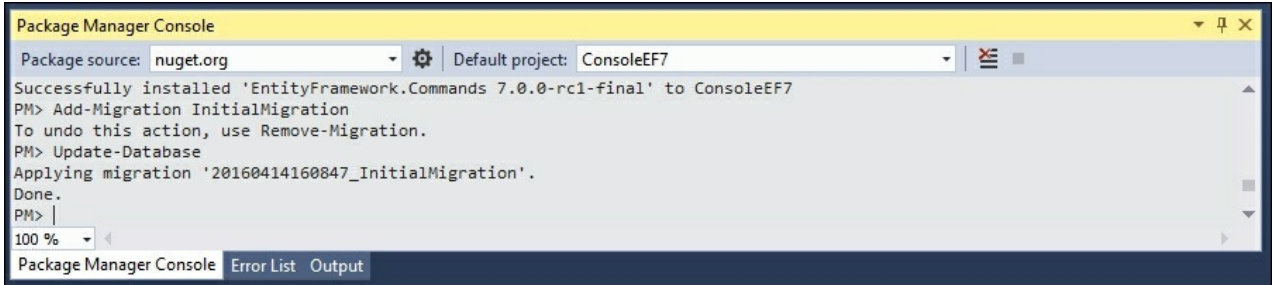
3. Once you update the database, you can see the changes in the database by connecting to SQL Server Management Studio:
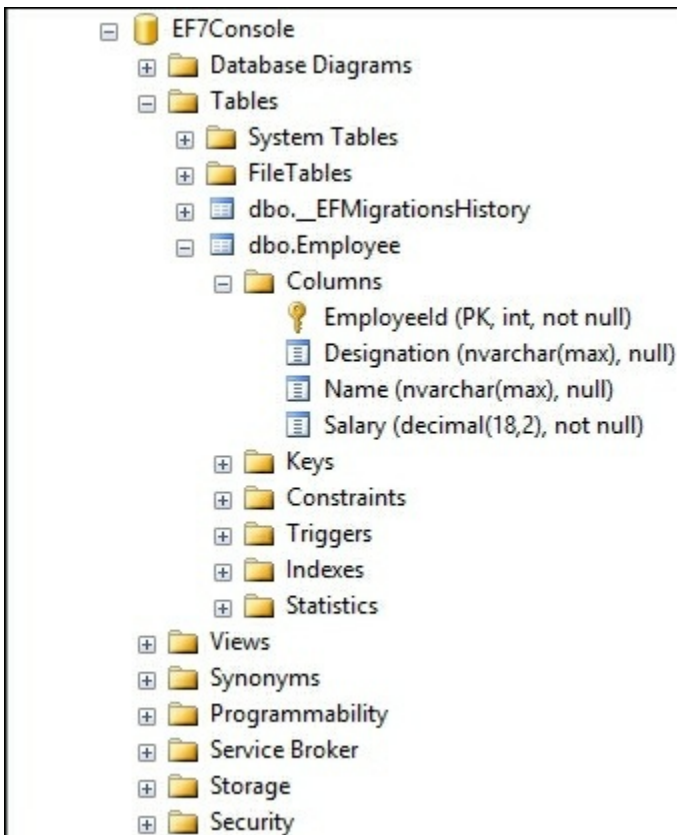
4. Perform the database operation to save the business domain object in the database. You can create the database manually or, if the database is not

available, it will create one for you.

The `Main` method is updated with the following code:

```
class Program {
  static void Main(string[] args) {
    AddEmployee();
  }

  static void AddEmployee() {
    using (var db = new EmployeeDbContext()) {
      Employee employee= new Employee {
        Designation = "Software Engineer",
        Name = "Scott",
        Salary = 5600
      };

      db.Employees.Add(employee);
      int recordsInserted = db.SaveChanges();
      Console.WriteLine("Number of records inserted:" +
recordsInserted);
      Console.ReadLine();
    }
  }
}
```

Firstly, we are constructing the business domain object. Then, we are adding the constructed `Employee` object to the employee's `DbSet` of the `DbContext` class. Finally, we are calling the `SaveChanges` method `DbContext` API, which will save all the pending changes to the database.

You might be wondering how it can save it to the database when we have not even given the connection string.

Let us discuss what happens behind the scenes when we run the program:

- When you make changes to any of the `DbSet` collection, the Entity Framework checks whether the database exists. If it does not exist, it creates a new one using the pattern `<Namespace of DbContextName>`. In our case, a database called by `EF6.EmployeeDbContext` would be created.
- Then, it creates database tables for the entities declared in `DbSet`. By convention, the Entity Framework uses the pluralized form of Entity for the table names. As we have declared `DbSet` for the `Employee` entity, the Entity Framework creates a pluralized form of `Employee` and creates the table named `Employees`.

The creation of the database and tables happens when the following code is
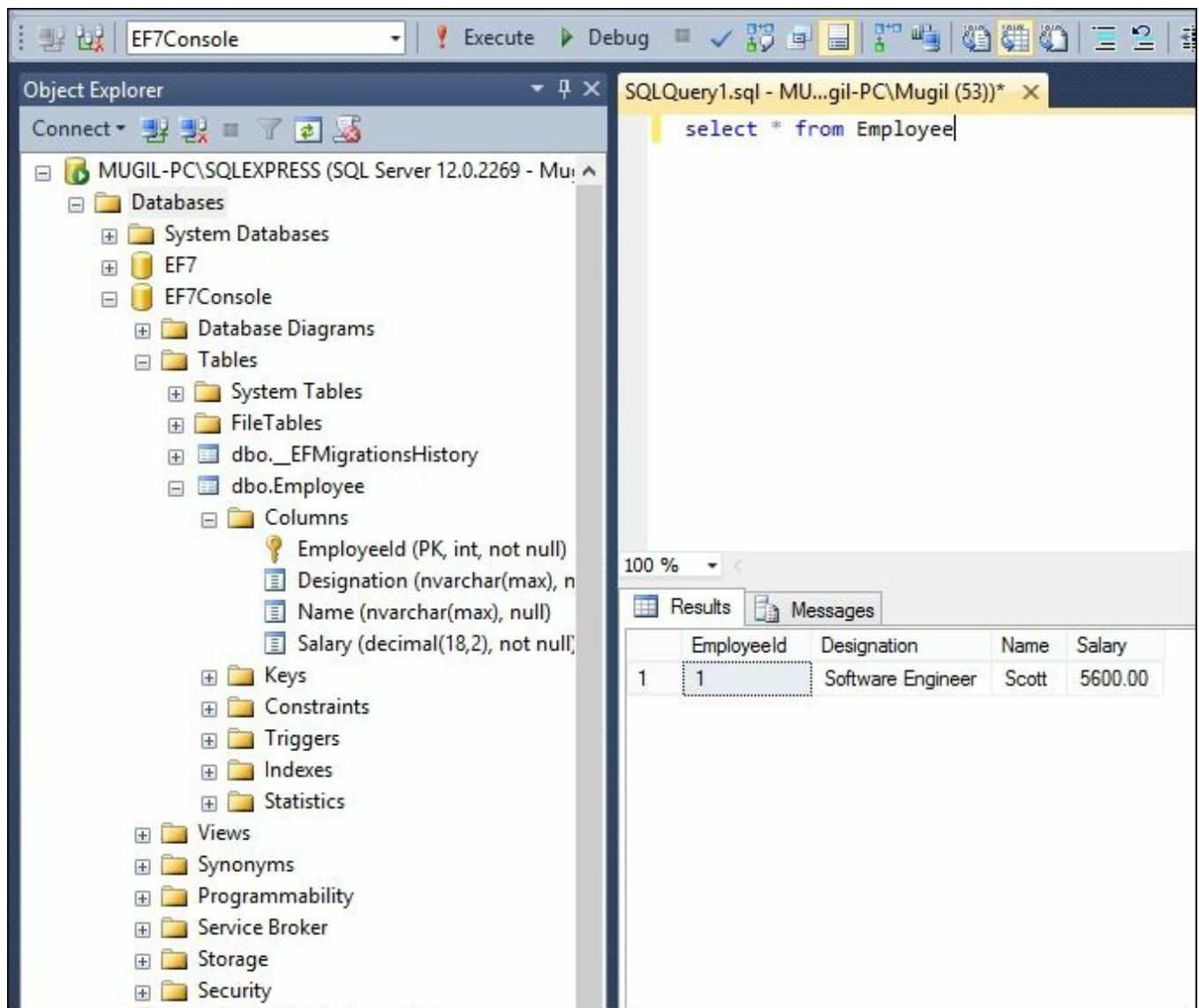
executed:

```
db.Employees.Add(employee);
```

When `SaveChanges` method is executed, the data in the `Employee` object will get saved to the database and returns the number of records affected. In the preceding case, it returns `1`.

When you run the application again, the first two steps mentioned previously will be skipped as the database and table will have already been created.

When you query the database, you can see the newly inserted record:

How the SaveChanges method works

When we are making changes, the Entity Framework tracks the state of each of the objects and executes the appropriate query when `SaveChanges` method is called.

For example, when we add an `Employee` object to the employees' collection (`DbSet`), this object is being tracked as `Entity` in the `Added` state. When `SaveChanges` is called, the Entity Framework creates an `insert` query for the same and executes it. The same is the case with updating and deleting the object. The Entity Framework sets the `Entity` state of the respective objects to `Modified` and `Deleted`. When `SaveChanges` is called, it creates and executes the `Update` and `Delete` queries.
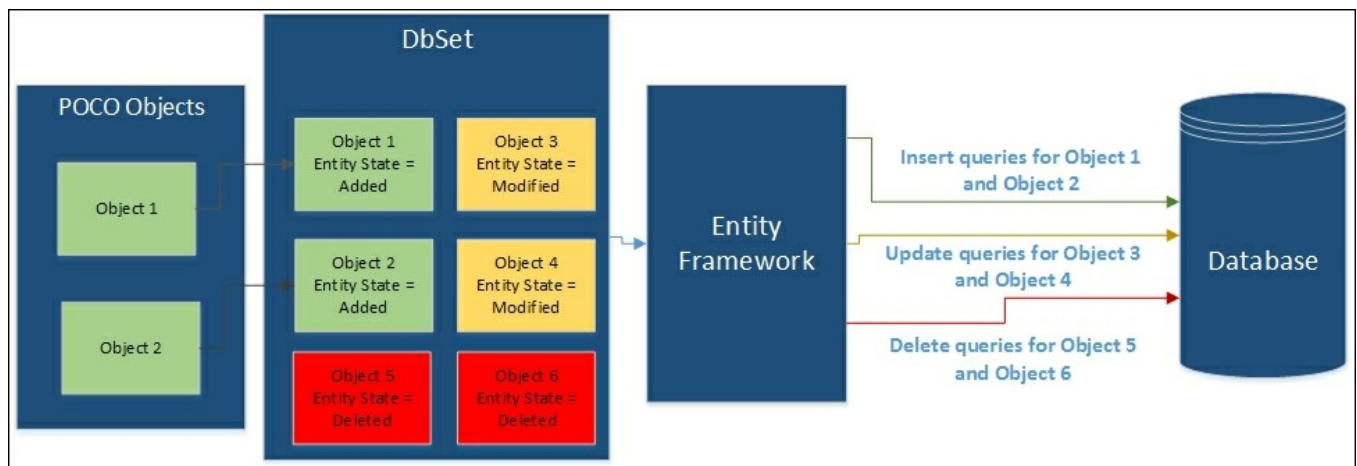
The preceding figure explains how the `SaveChanges` method works at a high-level for different types of change. We have a couple of POCO objects (**Object 1** and **Object 2**), which have been added to the employees' `DbSet` object. Let us assume **Object 3** and **Object 4** have been modified and objects **Object 5** and **Object 6** are in `Deleted` state. When you call `SaveChanges` method, it creates three sets of queries. The first set of queries is for the addition of objects, resulting in `insert` queries getting executed against the database. In the second set of queries, `Update` queries are created and executed for the objects whose state is modified. Finally, `Delete` queries are executed for the objects for all the `Deleted` state objects.

Updating the record

Let us try to update the salary of an inserted employee record using the Entity Framework:

```
static void UpdateSalary() {
  using (var db = new EmployeeDbContext()){
    Employee employee = db.Employees.Where(emp => emp.EmployeeId
== 1).FirstOrDefault();
    if(employee!=null){
      employee.Salary = 6500;
      int recordsUpdated = db.SaveChanges();
      Console.WriteLine("Records updated:" + recordsUpdated);
      Console.ReadLine();
    }
  }
}
```

In the preceding method, we find the employee with `EmployeeId = 1`. Then, we update the salary of the employee to `6500` and save the `employee` object to the database. Please note that, in the preceding method, we interact with the database a couple of times—once to find the correct employee record (`read` operation) and again to update the record (`update` operation).

```
static void Main(string[] args){
  UpdateSalary();
}
```

The `Main` method is updated to call the `UpdateSalary` method. When you query the database, you should see the record with the updated information:

Deleting the record

Deleting the record is a bit tricky as it involves setting the state directly. In the following method, firstly we get the object and setting the state of the object to `Deleted`. Then calling the `SaveChanges` method will generate the `delete` query for the object and execute it, which in turn will eventually delete the record in the database:

```
static void DeleteEmployee() {
  using (var db = new EmployeeDbContext()) {
    Employee employeeToBeDeleted = db.Employees.Where(emp =>
emp.EmployeeId == 1).FirstOrDefault();
    if (employeeToBeDeleted != null) {
      db.Entry(employeeToBeDeleted).State =
Microsoft.Data.Entity.EntityState.Deleted;
      int recordsDeleted = db.SaveChanges();
      Console.WriteLine("Number of records deleted:" +
recordsDeleted);
      Console.ReadLine();
    }
  }
}

static void Main(string[] args) {
  DeleteEmployee();
}
```

Using the Entity Framework in ASP.NET MVC applications

There is not much difference between using the Entity Framework in a console application and ASP.NET MVC application. Now, we are going to build a simple application with a single screen as shown in the following image. In this screen, we will have a form where the user will enter the information about the employee; once the user submits the form, the information will be saved to the database and reflected in the following screenshots:

We can create a simple Model for the employee. We need to build a ViewModel for this View, as we need to get the employee information from the user and we need to show a list of employees as well on the same screen.

Let us create an ASP.NET Core application, adding the employee and showing the list of employees. The following is the step-by-step instructions to create the application for the previously mentioned objective:

1. Create an ASP.NET Core project in Visual Studio by selecting an empty ASP.NET 5 application.
2. Install the ASP.NET Core `NuGet` package.
3. Install the Entity Framework 7 `NuGet` package and **ef** EntityFramework commands (for database migration) as explained earlier in this chapter.
4. Add `config.json` to declare the connection string of the database:

```json
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Data Source=MUGIL-
PC\\SQLEXPRESS;Initial Catalog=Validation;Integrated
Security=True"
    }
  }
}
```

5. Update `project.json` to include EntityFramework 7 and EntityFramework commands. The changes are highlighted in bold:

```json
{
  "version": "1.0.0-*",
  "compilationOptions":{
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.AspNet.IISPlatformHandler":        "1.0.0-rc1-
final",
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer":        "7.0.0-rc1-
final",
    "EntityFramework.Commands": "7.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel",
    "ef": "EntityFramework.Commands"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
  "publishExclude": [
```

```
      "**.user",
      "**.vspscc"
    ]
  }
```

6. Configure MVC in the `Startup` class (`Startup.cs`):
   - In the constructor, we are building the configuration by reading the `config.json` file
   - Add the MVC service and the Entity Framework service to the services in the `ConfigureServices` method
   - Configure the MVC routing in the `Configure` method:

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Validation.Models;
using Microsoft.Data.Entity;
using Microsoft.Extensions.PlatformAbstractions;

namespace Validation {
 public class Startup {
 public IConfigurationRoot Configuration { get; set; }

public Startup(IHostingEnvironment env,
IApplicationEnvironment appEnv) {
 var builder = new ConfigurationBuilder()
 .AddJsonFile("config.json")
 .AddEnvironmentVariables();
 Configuration = builder.Build();
 }

// This method gets called by the runtime. Use this method
to add services to the container.
// For more information on how to configure your
application, visit http
://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services)
{
services.AddEntityFramework()
 .AddSqlServer()
 .AddDbContext<EmployeeDbContext>(options => {
 options.UseSqlServer(Configuration.Get<string>
("Data:DefaultConnection:ConnectionString"));
 });
 services.AddMvc();
 }
// This method gets called by the runtime. Use this method
to configure the HTTP request pipeline.
```

```
public void Configure(IApplicationBuilder app) {
 app.UseIISPlatformHandler();
 app.UseMvc(routes => {
 routes.MapRoute(
 name: "default",
 template: "{controller=Employee}/ {action=Index}/{id?}");
 });
 }
// Entry point for the application.
 public static void Main(string[] args) =>
WebApplication.Run<Startup>(args);
 }
 }
```

7. Create `Models` and `DbContext` classes.
8. Create the `Models` folder and add the `Employee` model class and `EmployeeDbContext` class.
9. Create the `Employee` Model class (`Employee.cs` in the `Models` folder):

```
public class Employee {
  public int EmployeeId { get; set; }
  public string Name { get; set; }
  public string Designation { get; set; }
  public decimal Salary { get; set; }
}
```

10. Create `EmployeeDbContext` (`EmployeeDbContext.cs` in the `Models` folder):

```
using Microsoft.Data.Entity;
using Microsoft.Extensions.Configuration;

namespace Validation.Models {
  public class EmployeeDbContext : DbContext {

    public IConfigurationRoot Configuration { get; set; }

    public DbSet<Employee> Employees { get; set; }

    public EmployeeDbContext() {
      var builder = new ConfigurationBuilder()
      .AddJsonFile("config.json")
      .AddEnvironmentVariables();
      Configuration = builder.Build();
    }

    protected override void OnConfiguring
  (DbContextOptionsBuilder optionsBuilder) {
  optionsBuilder.UseSqlServer       (Configuration.Get<string>
  ("Data:DefaultConnection:ConnectionString"));
      base.OnConfiguring(optionsBuilder);
    }
```

```
        }
    }
```

11. Create `ViewModels`:
    - As we are going to show a list of employees and the form to add employees in the same screen, we are going to build a Model specific to this View. This model will contain information about the list of employees and the employee to be added.

12. Create the `ViewModels` folder and add the `EmployeeAddViewModel`:

```
using MVCEF7.Models;

namespace MVCEF7.ViewModels {
  public class EmployeeAddViewModel {
    public List<Employee> EmployeesList { get; set; }
    public Employee NewEmployee { get; set; }
  }
}
```

   - This `ViewModel` has a couple of properties. `EmployeesList` and `NewEmployee`. `EmployeesList` will contain the list of employees. This list would be fetched from the database. `NewEmployee` will hold the employee information entered by the user.

13. Create `Controllers` to handle the incoming requests:
    - Create a `Controllers` folder and add the `EmployeeController` class with a couple of action methods-one for `GET` and another for `POST`. The `Index` action method corresponding to the `GET` action method will be called when you access the URL (`http://localhost/Employee/Index`) or when you run the application. The `POST Index` action method will be called when you submit the form as following:

```
public IActionResult Index() {
  EmployeeAddViewModel employeeAddViewModel = new
EmployeeAddViewModel();
  using (var db = new EmployeeDbContext()) {
    employeeAddViewModel.EmployeesList =
db.Employees.ToList();
    employeeAddViewModel.NewEmployee = new Employee();

  }
  return View(employeeAddViewModel);
}
```

   - In the preceding `GET Index` action method, we are creating the `ViewModel` object and passing it to the View.
   - The following code uses `POST Index` action method:

```
[HttpPost]
public IActionResult Index(EmployeeAddViewModel
```

```
employeeAddViewModel) {

  using (var db = new EmployeeDbContext()) {
    db.Employees.Add(employeeAddViewModel.NewEmployee);
    db.SaveChanges();
    //Redirect to get Index GET method
    return RedirectToAction("Index");
  }

}
```

- We get the `NewEmployee` property in the ViewModel, which contains the information on the user. Save it to the database. Once we save the employee information to the database and we redirect the control to the `GET Index` action method, the `GET Index` action method will again show the form to enter the employee information and the list of employees in table format.

14. Add the `Views` folder:

1. Create `Views\_ViewStart.cshtml` with the following content:

```
@{
  Layout = "_Layout";
}
```

2. Create `Views\Shared\_Layout.cshtml` with the following content:

```
<!DOCTYPE html>

<html>
  <head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
  </head>
  <body>
    <div>
      @RenderBody()
    </div>
  </body>
</html>
```

3. Create `Views\Employee\Index.cshtml` with the following content:

```
@model MVCEF.ViewModels.EmployeeAddViewModel
@*
//For more information on enabling MVC for empty projects,
visit http://go.microsoft.com/fwlink/?LinkID=397860
*@
@{
}

<div>
```

```
    @using (Html.BeginForm("Index", "Employee",
FormMethod.Post)) {
    <table>
      <tr>
        <td>@Html.LabelFor(Model =>
Model.NewEmployee.Name)</td>
        <td>@Html.TextBoxFor(Model =>
Model.NewEmployee.Name)</td>
      </tr>
      <tr>
        <td>@Html.LabelFor(Model =>
Model.NewEmployee.Designation)</td>
        <td>@Html.TextBoxFor(Model =>
Model.NewEmployee.Designation)</td>
      </tr>
      <tr>
        <td>@Html.LabelFor(Model =>
Model.NewEmployee.Salary)</td>
        <td>@Html.TextBoxFor(Model =>
Model.NewEmployee.Salary)</td>
      </tr>
      <tr>
        <td colspan="2"><input type="submit"
value="Submit"/>
        </td>
      </tr>
    </table>

  }
</div>

<br/><br/> <br/>

<b> List of employees:</b> <br/>
<div>
  <table border="1">
    <tr>
      <th> ID </th>
      <th> Name </th>
      <th> Designation </th>
      <th> Salary </th>
    </tr>
    @foreach(var employee in Model.EmployeesList) {
      <tr>
        <td>@employee.EmployeeId</td>
        <td>@employee.Name</td>
        <td>@employee.Designation</td>
        <td>@employee.Salary</td>
      </tr>
    }
  </table>
</div>
```

In the preceding `Index` view, we create a form where we get the employee information from the user in the topmost `div`. In the next `div`, we show the list of employees in a tabular format.
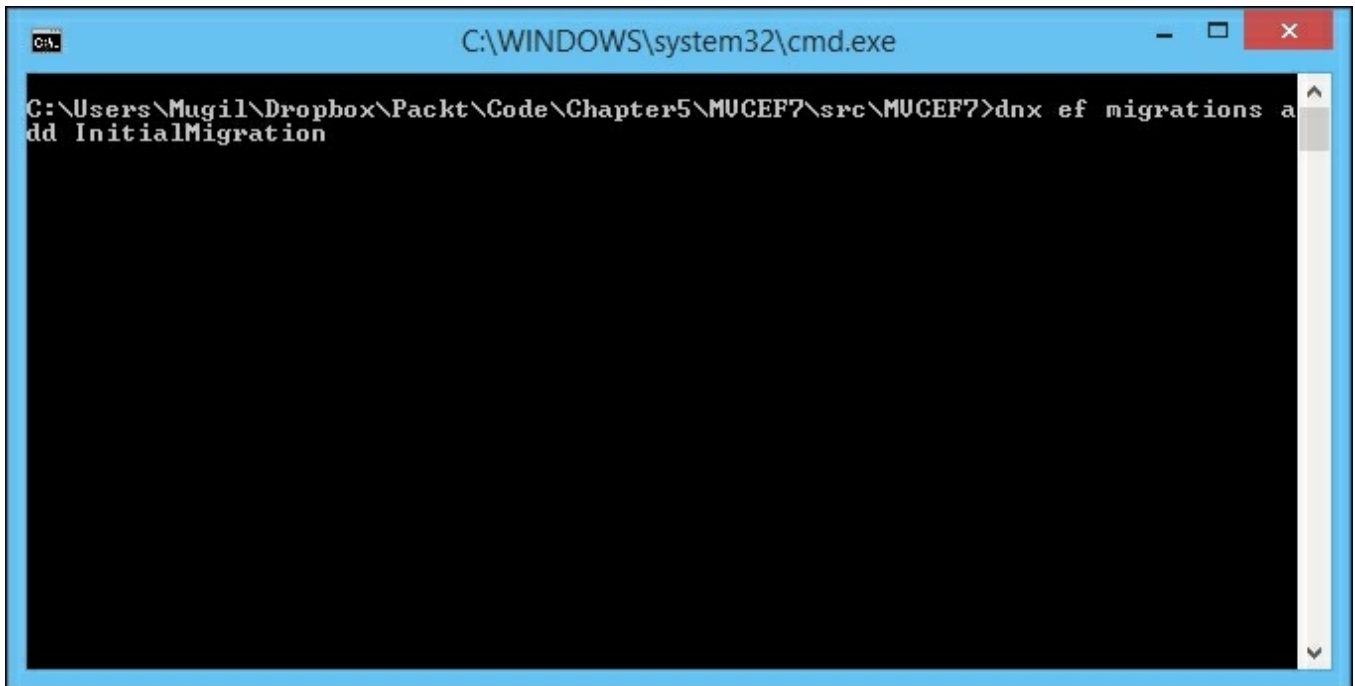
Once we create all the folders and the files, the project structure should look like the following:
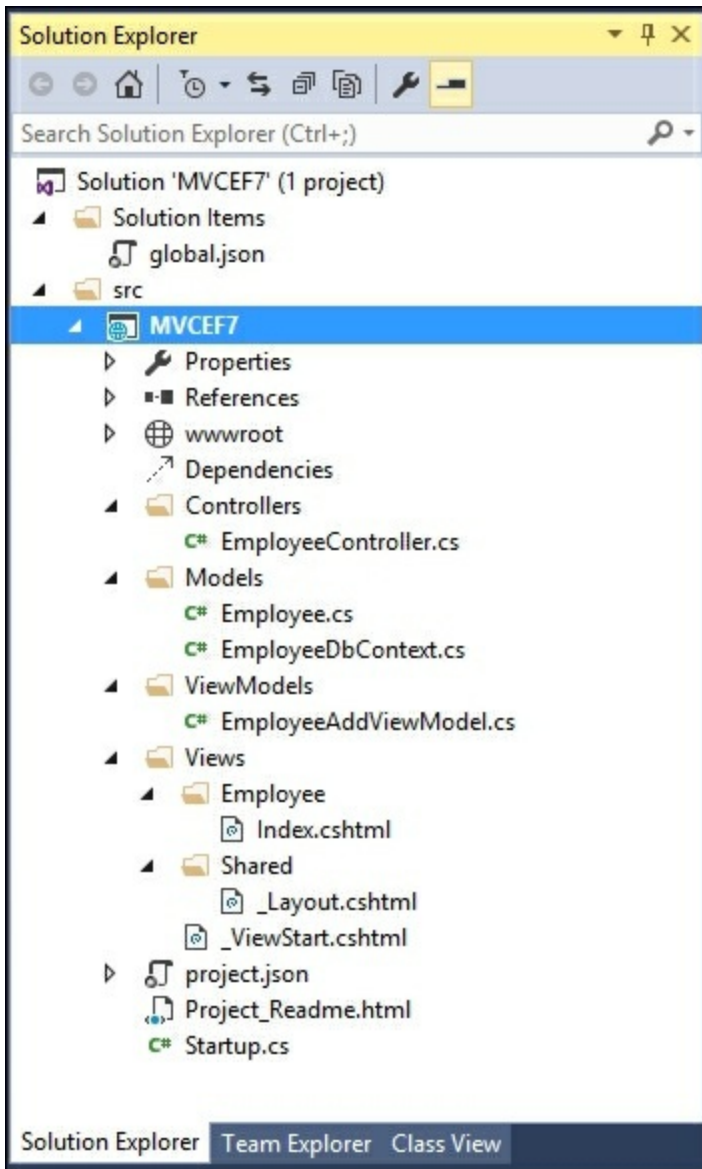
Database migration

We have created the business entity—the `Employee` class. Now, we can proceed with the migration. Migration is a two-step process: in the first step, we create the migration files. This can be done by executing the following command from the command prompt from the context of the project:

```
dnx ef migrations add InitialMigration
```


This command will create the migration files in your project, as shown in the following screenshot:
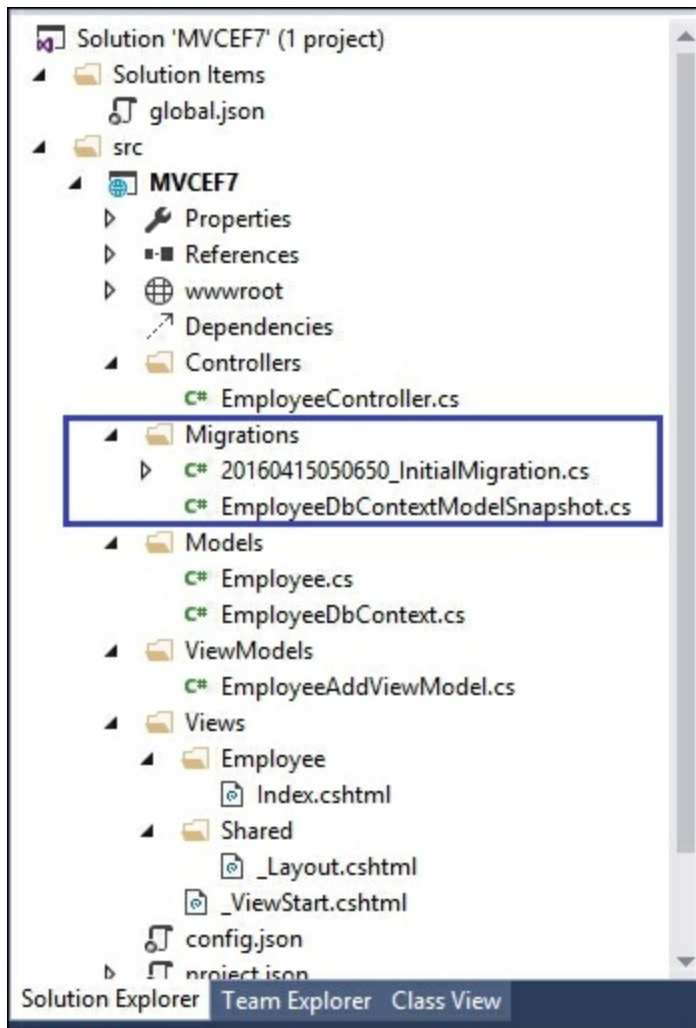
Then execute the following command to create the database:
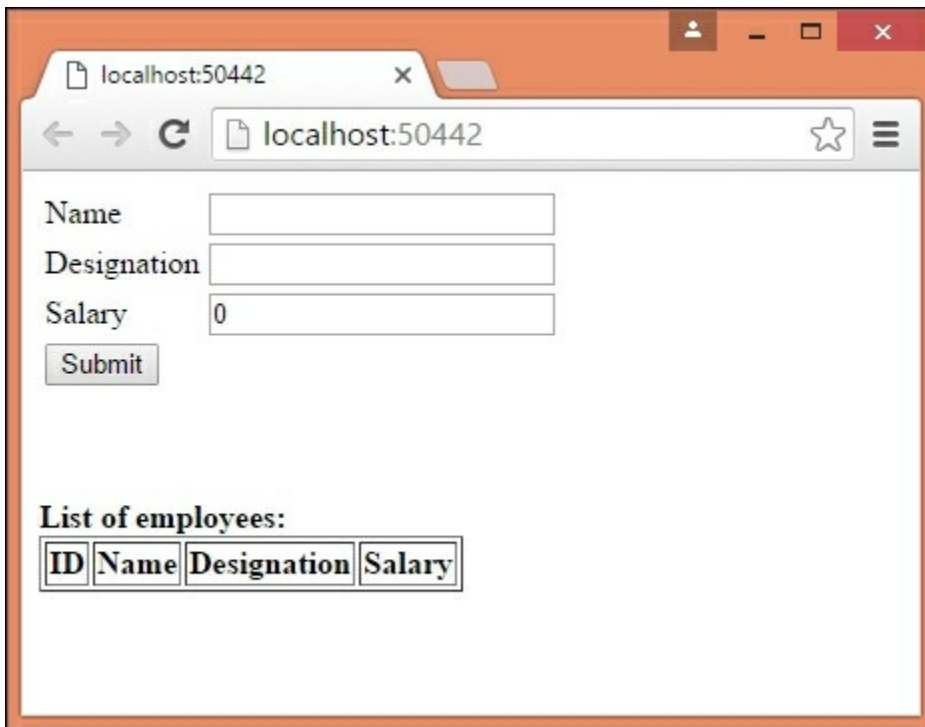
This command will read the migration files created in the previous step and create the database along with the associated tables:
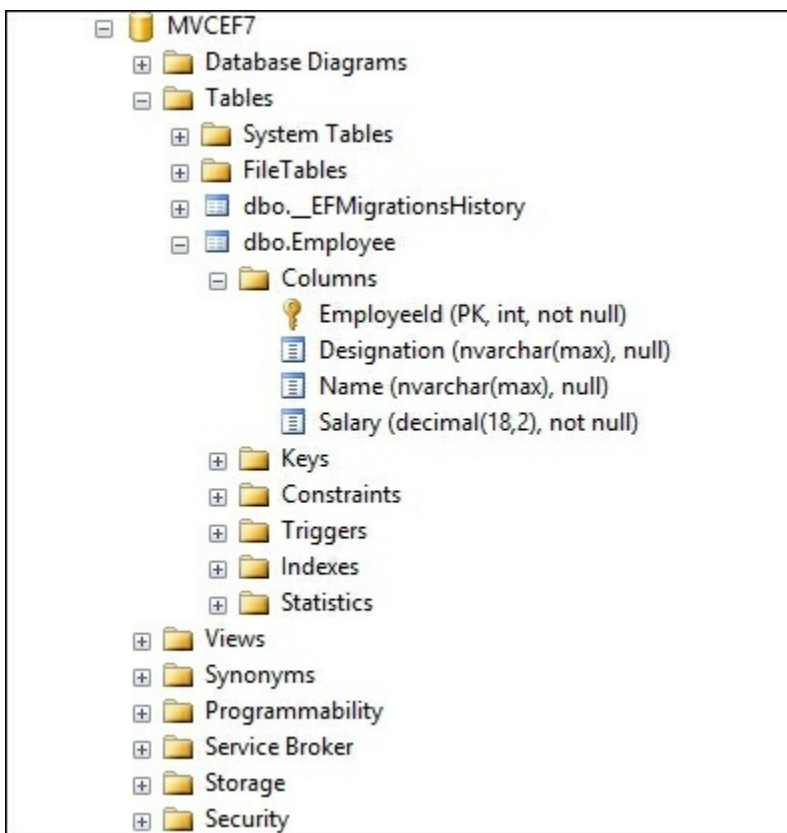
Run the application. You will get the following screen, where the user can enter the employee information in the form. As we are using the strongly typed model in our view, it takes the default values for all the properties. The `Name` and `Designation` are properties of type `string` and the default values are empty string for these fields, `Salary` is of type `decimal` and the default value for `decimal` is `0` hence `0` is shown in the form when it is loaded for the `Salary` field.

As there are no records, we are showing `0` records in the `List of employees` table:

When you enter the information in the form and submit it, the information gets saved in the database and all the database records in the `Employees` table will be presented as follows:

Summary

In this chapter, we learned what a Model is and how it fits in the ASP.NET MVC application. Then, we created a simple Model, built model data in a Controller, passed the Model to the View, and shown the data using the View. We have learned about the *Models specific to a View* and have discussed the flow of the data with respect to Models. We learned about the Entity Framework, an ORM framework from Microsoft, and how it simplifies database access from your .NET application. We have created simple console application where we have inserted, updated, and deleted the records. Finally, we have built an ASP.NET Core application that uses Model, ViewModel, and Entity Framework.

Chapter 6. Validation

We can never rely on the data entered by users. Sometimes they might be ignorant about the application and thus they may be entering incorrect data unknowingly. At other times, some malign users may want to corrupt the application by entering inappropriate data into it. In either case, we need to validate the input data before storing the data for further processing.

In this chapter, you'll be learning about the following topics:

- Different types of validation
- Server-side validation with an example
- Client-side validation with an example
- Unobtrusive JavaScript validation using jQuery unobtrusive libraries, where we don't have to write separate code for validation

In an ideal case, users will enter valid data in a proper format in your application. But, as you might realize, the real world is not so ideal. Users will enter incorrect data in your application. As a developer, it is the responsibility of us to validate the user input in our application. If the entered input is not valid, you need to inform the user, saying what has gone wrong, so that the user can correct the input data and submit the form again.

Validation can be done on the client-side or the server-side or at both ends. If the validation is done before sending the data to the server, it is called client-side validation. For example, if the user does not enter any data in a mandatory field, we can validate (by finding the data that is not entered) the form, at the client-side itself. There is no need to send the form data to the server. JavaScript is the most commonly used language being used for client-side validation.
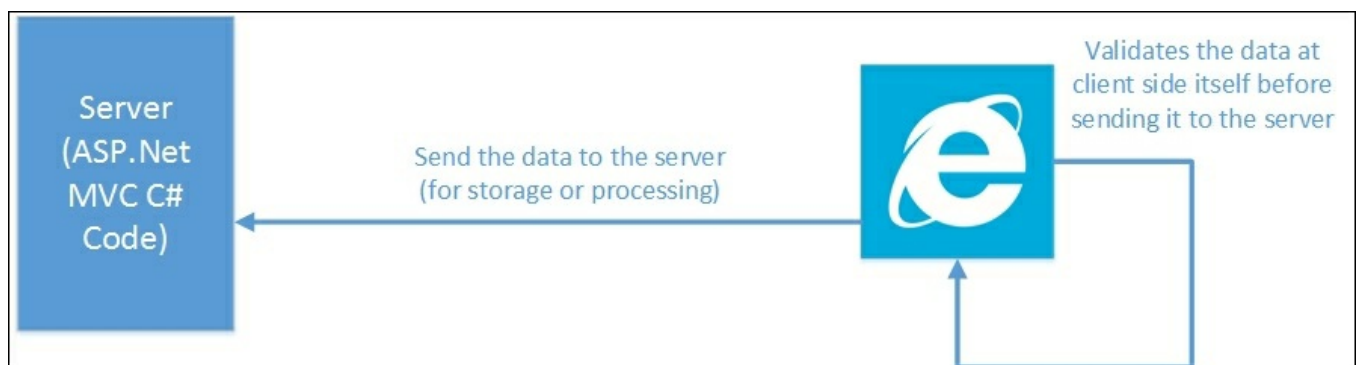
If the validation is done at the server-side (sending the form data to the server), it is

called server-side validation. For instance, you might want to validate data entered by the user against the data in the database. In this case, it is preferable to do server-side validation as we cannot have all the data in the database at the client-side.

Validation is done at server side. If validation succeeds, the data is processed. Else, the error message is shown to the user

Send the data to the server (for validation and further processing, if validation succeeds)

No validation is being done at client side

Server (ASP.Net MVC C# Code)

If validation fails at server, error msg is shown to the user so that user can correct the input data and send it again

Client-side and server-side validation

In the real world, it's not a case of either server-side or client-side validation. We can have both types of validation at the same time. In fact, it is recommended to validate the data at both ends to avoid unnecessary processing.
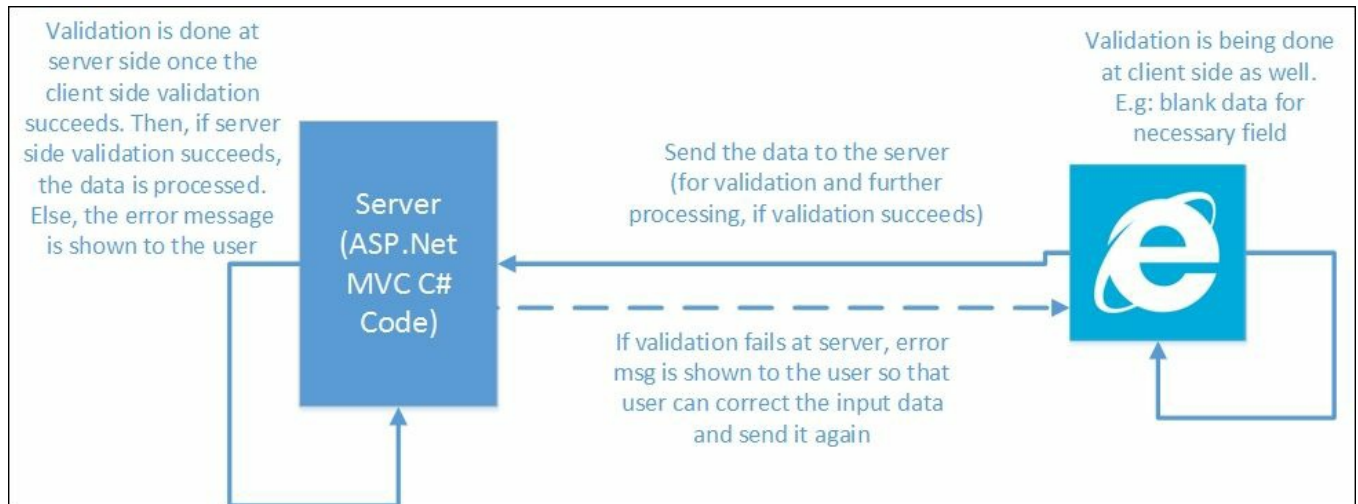
The preceding figure shows the validation is being performed at both the client-side and the server-side. If the data is not entered into the required field, we can catch that issue at the client-side itself. There is no need to send the data to the server to finally find out that there is no data entered. Once all the required data is entered, the data is sent back to the server to validate the entered data based on some business logic. If the validation fails, the form data is sent again to the browser with the error message so that the user can send the data again.

We have covered enough theory about the need for validation and the types of validations typically used in the application. Let us get our hands dirty by adding validation to the application that we built in the previous chapter.

The following screenshot is the form that we have built in the previous chapter. There is nothing fancy in this form—just three fields. When a user enters the data in the form, the data is stored in the database and the entire employee information is fetched back and shown in a tabular format.
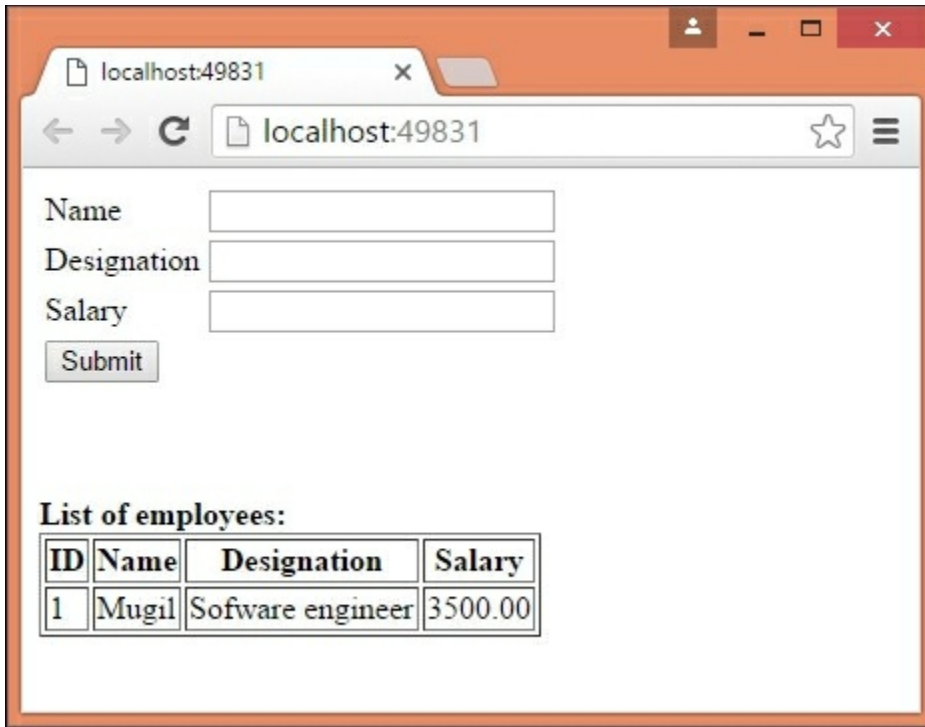
In the existing application that we have built, we do not show any message to the user even when the user does not enter any information in any of the fields and submits it. Instead, we silently store the default values for the fields (empty values for string types and 0.00 for decimal types) as shown in the following screenshot:
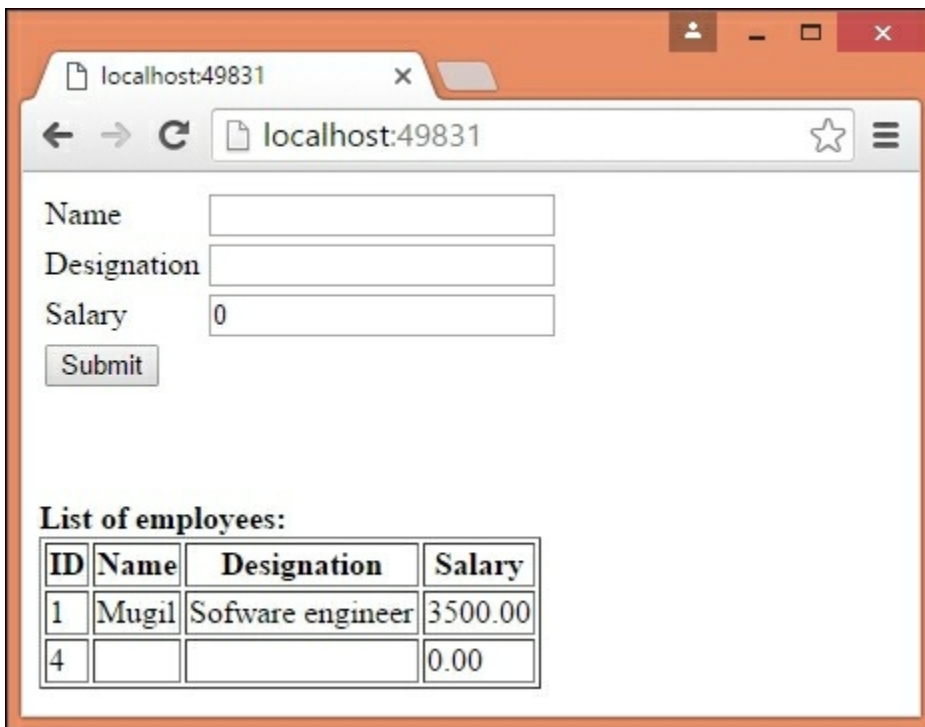
But this should not be the case. We should inform the user saying that the data entered is not valid and ask the user to correct the input data.

Server-side validation

Let us continue with the application that we built in the previous chapter. To do server-side validation, we need to do the following:

1. Add Data Annotation attributes to the `ViewModels` model class. The input data is validated against this metadata and the model state is updated automatically.
2. Update the `view` method to display the validation message for each of the fields. The `span` tag helper with the `asp-validation-for` attribute will be used to display the validation error message.
3. Update the controller action method to verify the model state. If the model state is valid, we insert the data into the database. Otherwise, the View model is updated and the `view` method is rendered again with the validation error message so that the user can update with valid input data and submit the form again.

Updating View models with the Data Annotation attribute

The Data Annotation attributes defines the validation rules for the properties of the `Model`/`ViewModel`. If the input data does not match with the attribute definition in the model, the validation will fail, which in turn makes the associated model state invalid.

There are several Data Annotation attributes available to validate the data. The following are the most commonly used Data Annotations attributes:

- **Required**: This attribute indicates the property is required.
- **Range**: This attribute defines the minimum and maximum constraints.
- **MinLength**: This defines the minimum length a property must have in order for the validation to succeed.
- **MaxLength**: As the name implies, this attribute defines the maximum length of the property. If the length of the property value exceeds the maximum length, the validation would fail.
- **RegularExpression**: We can use a regular expression for data validation if we use this attribute.

As Data Annotation attributes are available in the `System.ComponentModel.DataAnnotations` namespace, we need to include this namespace. The following is the updated View model code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Validation.Models;

namespace Validation.ViewModels
{
    public class EmployeeAddViewModel
    {
        public List<Employee> EmployeesList { get; set; }
        [Required(ErrorMessage ="Employee Name is required")]
        public string Name { get; set; }

        [Required(ErrorMessage ="Employee Designation is
required")]
        [MinLength(5, ErrorMessage = "Minimum length of designation
should be 5 characters")]
        public string Designation { get; set; }
```

```
        [Required]
        [Range(1000,9999.99)]
        public decimal Salary { get; set; }
    }
}
```

We have added Data Annotation attributes for all the three properties—`Name`, `Designation`, and `Salary`.

The `ErrorMessage` attribute displays a message to be displayed when the validation fails. If there is a failure of validation and if there is no `ErrorMessage` mentioned, the default error message will be displayed.

Updating the View model to display the validation error message

For each of the fields, we have added a `span` tag where the error message is displayed in a red color when the validation fails. When the validation succeeds, there will be no error message displayed. The attribute value of `asp-validation-for` represents the field name for which the validation error message has to be displayed. For example, we have used the `span` tag with the `asp-validation-for` attribute and with the value `Name`, which tells ASP.NET MVC to display the validation error message for the `Name` field:

```
<form asp-controller="Employee" asp-action="Index">
    <table>
        <tr>
            <td><label asp-for="Name"></label></td>
            <td><input asp-for="Name" /></td>
            <td><span asp-validation-for="Name"
style="color:red"></span></td>
        </tr>
        <tr>
            <td><label asp-for="Designation"></label> </td>
            <td><input asp-for="Designation" /></td>
            <td><span asp-validation-for="Designation"
style="color:red"></span> </td>
        </tr>
        <tr>
            <td><label asp-for="Salary"></label></td>
            <td><input asp-for="Salary"></td>
            <td> <span asp-validation-for="Salary"
style="color:red"></span> </td>
        </tr>
        <tr>
            <td colspan="2"><input type="submit"
id="submitbutton" value="Submit" /></td>
        </tr>
    </table>
</form>
```

Updating the controller action method to verify the model state

The model state is automatically updated based on the Data Annotation attribute specified on our View model and the input data. We are verifying whether the model state is valid in the following `Index` method, which is a `POST` action method. If the model state is valid (when the validation succeeds), we save the entered data to the database. If the validation fails, then the `ModelState` is set to `invalid` automatically. Then, we would populate the `ViewModel` with the entered data and render the `View` method again so that the user can correct the input data and re-submit the data:

```
[HttpPost]
    public IActionResult Index(EmployeeAddViewModel
employeeAddViewModel)
{
        if (ModelState.IsValid)
        {
            using (var db = new EmployeeDbContext())
            {
                Employee newEmployee = new Employee
                {
                    Name = employeeAddViewModel.Name,
                    Designation = employeeAddViewModel.Designation,
                    Salary = employeeAddViewModel.Salary
                };
                db.Employees.Add(newEmployee);
                db.SaveChanges();
                //Redirect to get Index GET method
                return RedirectToAction("Index");
            }
        }
        using (var db = new EmployeeDbContext())
        {
            employeeAddViewModel.EmployeesList =
db.Employees.ToList();
        }
        return View(employeeAddViewModel);
}
```

When you run the application after making aforementioned changes and submit the form without entering the values, error messages will be displayed beside the fields as shown in the following screenshot. Please note that, even in the case of a validation error, we display the employees' data in the following table, which is achieved by using the code block in the previous code snippet.
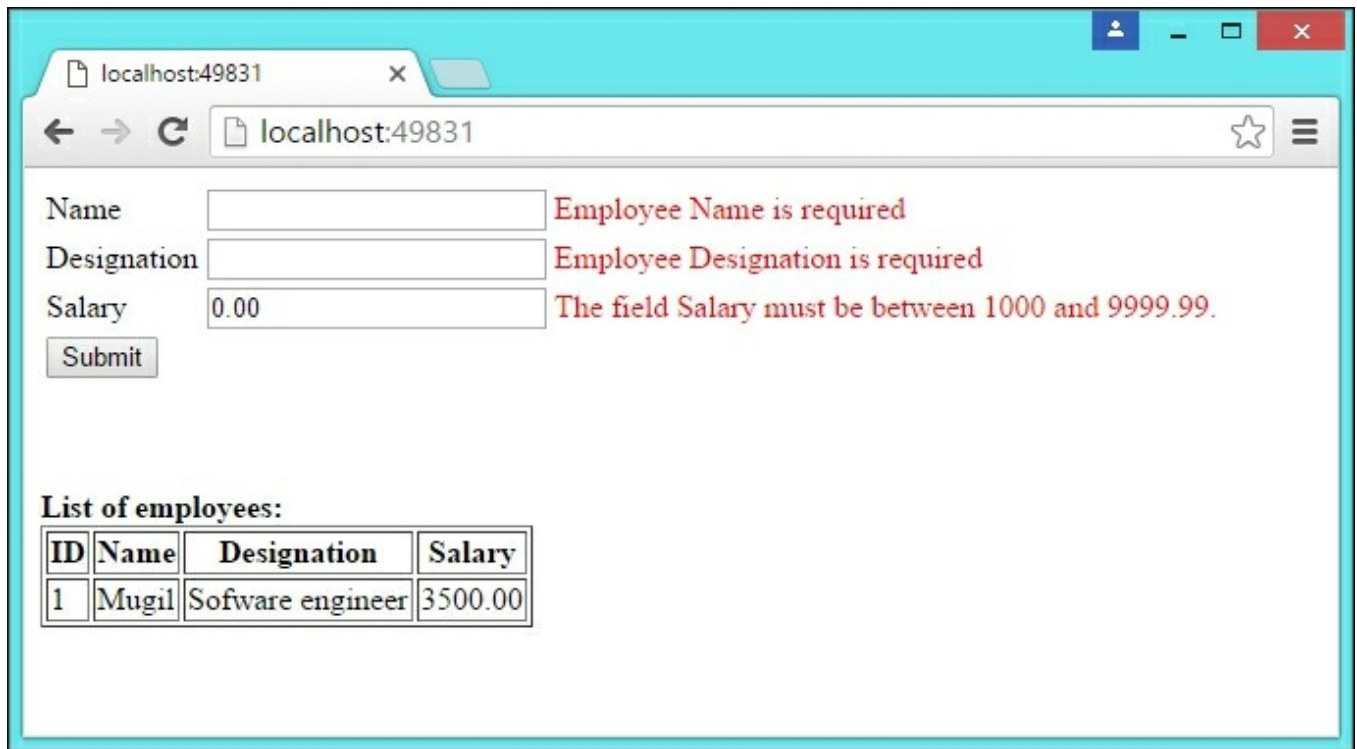
There are a few things to be noted in the previous validation and its error message:

- If the validation fails, error messages are displayed as expected.
- If there is more than one validation for a field, it will display one error message at a time. For example, we have a couple of validations for **Designation** field— the `Required` and `MinLength` attributes. If there is no data entered for the field, only the required field error message will be displayed. Only when the required field error is resolved (by entering some characters in the field), the second validation error message will be displayed.
- If no error message is available and if the validation fails, the default error message is displayed. We have not given any error message for the **Salary** field. So, when the validation fails for that field, ASP.NET MVC displays the default error message based on the field name and the type of validation failure.
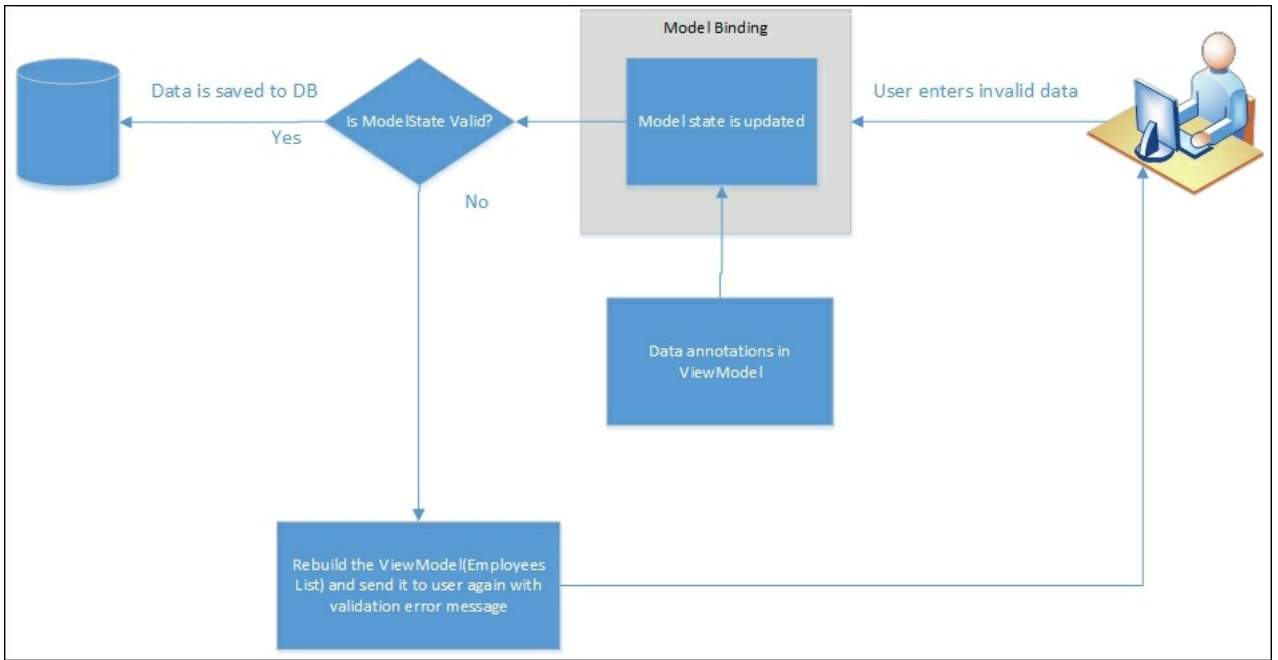
The preceding figure depicts the high-level sequence of events in server-side validation and is described as follows:

1. The user enters the invalid data.
2. Based on the Data Annotations attribute in the View model, the model state is updated automatically. This happens during the model binding process where the data in the `view` method is mapped to the data in the model or View model.
3. In the controller's action method, we are verifying the model state.
4. If the model state is valid, we are saving the entered data to the database.
5. If the model state is not valid, we are rending the View model again with the validation error message so that the user can correct the input data and submit the form again with the valid input data.

Client-side validation

There are scenarios where we don't need to go to the server to validate the input data. In the preceding example of the server-side validation, we do not need to go to the server to verify whether the user has entered the data for the **Name** field. We can validate at the client-side itself. This prevents round-trips to the server and reduces the server load.

We are going to use JavaScript to validate the data from the client-side. JavaScript is a high-level, interpreted language which is primarily used in client-side programming.

Note

These days, JavaScript is also being used at the server-side as part of Node.js.

We are going to make a couple of changes in our View model (`Index.cshtml` file) to validate the form at the client-side:

1. Changes in the form: add the `id` attribute to all the `span` tags so that we can access this HTML element to display the HTML error message. On submission of the form, call a JavaScript function to validate the input data.
2. Add the script HTML element and create a JavaScript function to validate the input data.

In the following code, we are calling the `validateForm` JavaScript function on submission of the form. If the `validateForm` function returns `true`, the data will be sent to the server. Otherwise, the data will not be sent. We have added the `id` attribute for all the `span` tags so that we can identify the `span` tags and display the validation error messages over there:

```
<form asp-controller="Employee" asp-action="Index" onsubmit="return
validateForm()">
        <table>
            <tr>
                <td><label asp-for="Name"></label></td>
                <td><input asp-for="Name" /></td>
                <td><span id="validationName" asp-validation-
for="Name" style="color:red"></span></td>
            </tr>
            <tr>
                <td><label asp-for="Designation"></label> </td>
                <td><input asp-for="Designation" /></td>
                <td><span id="validationDesignation" asp-
validation-for="Designation" style="color:red"></span> </td>
```

```
                </tr>
                <tr>
                    <td><label asp-for="Salary"></label></td>
                    <td><input asp-for="Salary"></td>
                    <td> <span id="validationSalary" asp-validation-
for="Salary" style="color:red"></span> </td>
                </tr>
                <tr>
                    <td colspan="2"><input type="submit"
id="submitbutton" value="Submit" /></td>
                </tr>
        </table>
</form>
```

We have added the JavaScript function to validate all three fields. We get the values of all the three fields and we store them in separate variables. Then we verify whether the value of each of the variables is null or empty. If the value is empty, we get the `span` element for the respective field and set the text context with the validation error message:

```
<script type="text/javascript">
        function validateForm() {
            var isValidForm = true;
            var nameValue = document.getElementById("Name").value;
            var designationValue =
document.getElementById("Designation").value;
            var salaryValue =
document.getElementById("Salary").value;

            //Validate the name field
            if (nameValue == null || nameValue == "") {

document.getElementById("validationName").textContent = "Employee
Name is required - from client side";
                isValidForm = false;
            }

            //validate the designation field
            if (designationValue == null || designationValue == "")
{

document.getElementById("validationDesignation").textContent =
"Employee Designation is required - from client side";
                isValidForm = false;
            }

            //validate the salary field - if it is empty
            if (salaryValue == null || salaryValue == "") {

document.getElementById("validationSalary").textContent = "Employee
Salary is required - from client side";
```

```
                    isValidForm = false;
            }else if (Number(salaryValue) == NaN ||
Number(salaryValue)<=0.0) {

document.getElementById("validationSalary").textContent = "Please
enter valid number for salary field - from client side";
                    isValidForm = false;
            }


            return isValidForm;

        }

</script>
```

When you run the application and submit the form without entering the data, you'll get the error message generated from the client-side itself without ever going to the server.

In real-world applications, we would not be hand coding the validation code at the JavaScript. Instead, most applications use unobtrusive validation, where we do not write JavaScript code for validating each of the fields. Simply adding the respective JavaScript libraries will do.

You might wonder how the fields get validated without ever writing the code. The magic lies in the data- attributes added to the input HTML elements based on the

Data Annotation attributes. This jQuery unobtrusive library gets a list of fields for which `data-` attributes are added and it gets validated.

Run the application and press *Ctrl + U* to see the source code. The source code will look something like the following:

```html
<div>

    <form method="post" role="form" action="/">
        <table>
            <tr>
                <td><label for="Name">Name</label></td>
                <td><input type="text" data-val="true" data-val-required="Employee Name is required" id="Name" name="Name" value="" /></td>
                <td><span id="validationName" style="color:red" class="field-validation-valid" data-valmsg-for="Name" data-valmsg-replace="true"></span></td>
            </tr>
            <tr>
                <td><label for="Designation">Designation</label> </td>
                <td><input type="text" data-val="true" data-val-minlength="Minimum length of designation should be 5 characters" data-val-minlength-min="5" data-val-
required="Employee Designation is required" id="Designation" name="Designation" value="" /></td>
                <td><span id="validationDesignation" style="color:red" class="field-validation-valid" data-valmsg-for="Designation" data-valmsg-replace="true"></span>
</td>
            </tr>
            <tr>
                <td><label for="Salary">Salary</label></td>
                <td><input type="text" data-val="true" data-val-number="The field Salary must be a number." data-val-range="The field Salary must be between 1000 and
9999.99." data-val-range-max="9999.99" data-val-range-min="1000" data-val-required="The Salary field is required." id="Salary" name="Salary" value="0.00"></td>
                <td> <span id="validationSalary" style="color:red" class="field-validation-valid" data-valmsg-for="Salary" data-valmsg-replace="true"></span> </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" id="submitbutton" value="Submit" /></td>
            </tr>

        </table>
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8BynGVnuIo9Jk-
ZC8mwuRmQIs8Rr2rfODjOinjqKbbEtTXIptb8p9BZSforM6kfRpp5KaUprON7Mxl5ypXKRHgBaf7QZFOJWkZcCKeedHk87ZnBYSA7toF3LCTkAZS5YlxfTJiLt3FD7HQ-62t9HuV0" /></form>

</div>
```

Different attributes will be added to different kinds of Data Annotation attributes. For the fields to be validated, the `data-val` attribute would be set to `true`. For the properties which are marked as required in the View model, the `data-val-required` attribute will have the value of the error message of the associated property.

Implementation

There will be a layout file (`_Layout.cshtml`) to define the layout structure of your web application. As JavaScript libraries are going to be used in all the pages, this is the right place to add common functionalities such as unobtrusive validation. Just add the JavaScript libraries (highlighted in bold) to the layout file (`_Layout.cshtml`) so that they will be available for all the `View` files:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>

    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
2.2.3.js"></script>
    <script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.
validate.min.js"></script>
    <script
src="https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unob
trusive.min.js"></script>
</body>
</html>
```

There is no change to the View model except for the removal of the JavaScript function we wrote earlier for validating the fields. The complete code for the view is as following:

```
@model Validation.ViewModels.EmployeeAddViewModel

<div>

    <form asp-controller="Employee" asp-action="Index"
method="post" role="form">
        <table>
            <tr>
                <td><label asp-for="Name"></label></td>
                <td><input asp-for="Name" /></td>
                <td><span id="validationName" asp-validation-
for="Name" style="color:red"></span></td>
            </tr>
            <tr>
```

```
                <td><label asp-for="Designation"></label> </td>
                <td><input asp-for="Designation" /></td>
                <td><span id="validationDesignation" asp-
validation-for="Designation" style="color:red"></span> </td>
            </tr>
            <tr>
                <td><label asp-for="Salary"></label></td>
                <td><input asp-for="Salary"></td>
                <td> <span id="validationSalary" asp-validation-
for="Salary" style="color:red"></span> </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit"
id="submitbutton" value="Submit" /></td>
            </tr>

        </table>
    </form>

</div>

<br /><br /> <br />

<b> List of employees:</b> <br />
<div>
    <table border="1">
        <tr>
            <th> ID </th>
            <th> Name </th>
            <th> Designation </th>
            <th> Salary </th>
        </tr>
        @foreach (var employee in Model.EmployeesList)
        {
            <tr>
                <td>@employee.EmployeeId</td>
                <td>@employee.Name</td>
                <td>@employee.Designation</td>
                <td>@employee.Salary</td>
            </tr>
        }
    </table>

</div>
```

The preceding diagram depicts the unobtrusive client validation process:

1. Data Annotations are added to `Model`/`ViewModels`.
2. The view takes `Model`/`ViewModels` and generates the HTML.
3. The generated HTML from the View model contains `data-*` attributes:
   - For the fields for which the `Required` attribute is set, the `data-val-required` attribute is created with the error message as its value.
   - For the fields with the `MinLength` Data Annotation attribute, the `data-val-minlength` attribute is set with the error message as its value.
   - For the range Data Annotation, the `data-val-range` attribute is set with the error message as its value. The `data-val-range-max` represents the maximum value in the range and the `data-val-range-min` attribute represents the minimum value in the range.
4. The jQuery unobtrusive validation library reads these elements with `data-*` attributes and does the client-side validation. This allows the developer to not write the separation validation code using JavaScript as everything is resolved by the configuration itself.

Summary

In this chapter, we have learned about the need for validation and the different kinds of validation available. We have even discussed how client-side and server-side validation work, along with the pros and cons of each type of validation. Later, we made code changes to validate the input data at the server-side. Then we used JavaScript to validate the input data in the client-side itself. Finally, we used the jQuery unobtrusive library to do the client-side validation without ever writing the JavaScript code to validate the input data at the client-side.

In the next chapter, we will discuss the routing principle and how to customize it. In an earlier chapter, we saw the basics of routing in an ASP.NET 5 application. Now we are going to explore the topic in depth.

Chapter 7. Routing

Routing is one of the important concepts in the ASP.NET MVC application as it takes care of incoming requests and maps them to the appropriate controller's actions.

In this chapter, we are going to learn about the following topics:

- Using the `MapRoute` method to configure routing
- Different types of routing with examples—convention and attribute-based
- Using HTTP verbs in attribute-based routing

We briefly discussed routing in [Chapter 3](#) , *Controllers*. In this chapter, we are going to discuss routing along with several options available to customize it in ASP.NET Core.

Convention-based routing

The routing engine is responsible for mapping the incoming requests to the appropriate action method of the controller.

In the `Configure` method of the `Startup` class, we have mapped the following route:

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

The `MapRoute` method has two parameters:

- `name`: This represents the name of the route as we could configure multiple routes for the same application.
- `template`: This signifies the actual configuration for the route. There are three parts to this configuration value. As we are supplying default parameters, if the values are not passed, it will take the default parameter values.
- `{controller=Employee}`: The first value acts as the name of the controller and we use the `Employee` controller as the default controller when the controller value is not available in the URL.
- `{action=Index}`: The `Index` action method will be acting as the default action method and the second parameter from the URL will be taken as the action method name.
- `{id?}`: By specifying "`?`" after the `id` parameter, we are saying that `id` is the optional parameter. If the value is passed as the third parameter, the `id` value will be taken. Otherwise, it would not be considered.

There is another method with the same functionality. The `app.UseMvcWithDefaultRoute()` method configures the route "`{controller=Employee}/{action=Index}/{id?}`". But we have used the earlier method to show that we can customize the route as we want.

Let us see a few examples and observe how our routing engine works. We are assuming the following routing for the preceding examples:

```
"{controller=Employee}/{action=Index}/{id?}"
```

Example 1

URL- `http://localhost:49831/`

In this URL, we have not passed a value for the `controller`, `action`, or `id`. Since we have not passed anything, it would take the default values for the controller and the action. So, the URL is converted into the following URL by the routing engine:

`http://localhost:49831/Employee/Index`

Example 2

URL-`http://localhost:49831/Employee/`

In this URL, we have passed the value for the `controller` (the first parameter), which is `Employee`, whereas we did not pass anything for `action` method (the second parameter) or `id` (the third parameter). So, the URL will be converted into the following URL by taking the default value for the `action` method:

`http://localhost:49831/Employee/Index`

Example 3

URL-`http://localhost:49831/Manager/List`

The routing engine will take the first parameter, `Manager`, as the `controller` method name and the second parameter, `List`, as the `action` method name.

Example 4

URL-`http://localhost:49831/Manager/Details/2`

We have passed all three parameters in this URL. So, the first parameter value, `Manager`, will be considered as the `controller` method name, the second parameter value will be considered as the `action` method name, and the third parameter value will be considered as the `id` method name.

When defining the map route, we have used the `MapRoute` method with a couple of parameters. The first parameter, `name`, represents the name of the route and the second parameter, `template`, represents the URL pattern to be matched along with the default values:

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

There are other overloaded variations of this `MapRoute` method. The following is another commonly overloaded `MapRoute` method, where the incoming URL pattern and the default values are passed for different parameters. The name of the route is `FirstRoute` and this route will be applied for all URLs starting with `Home`. The default values for the controller and the action are `Home` and `Index2` respectively:

```
routes.MapRoute(name:"FirstRoute",
              template:"Home",
              defaults:new {controller ="Home", action="Index2"});
```

You can define any number of routing maps for your ASP.NET MVC application. There is no restriction or limit on the routing maps. Let us add another routing map to our application. We have added another route map called `FirstRoute` to our application (highlighted in bold):

```
public void Configure(IApplicationBuilder app)
    {
        app.UseIISPlatformHandler();
        app.UseMvc(routes =>
        {
            routes.MapRoute(name:"FirstRoute",
            template:"Home", defaults:new {controller ="Home",
action="Index2"});

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

```
    }
```

And we have added another `controller` method by the name `HomeController` with a couple of simple `action` methods returning different strings:

```
public class HomeController : Controller
    {
        // GET: /<controller>/
        public IActionResult Index()
        {
            return Content("Index action method");
        }

        public IActionResult Index2()
        {
            return Content("Index2 action method");
        }
    }
```

When you try to access the application through the URL, `http://localhost:49831/Hello`, both routing maps, `FirstRoute` and the `default`, match with the URL pattern.

Which map routing, do you think, will get applied in this scenario?

The routing engine maps the incoming URL based on the following factors:

1. Matching pattern.
2. On the order defined in the routing engine.

The first factor is an obvious one. For a routing map to be picked up by the routing engine, the pattern of the incoming URL should get matched with the defined template in the routing map.

The second factor is subtle but important. If more than one routing map matches with the incoming URL, the routing engine will pick the first URL as defined in the configuration. For example, if the incoming URL matches with both the `FirstRoute` and `default` maps, the routing engine will pick the `FirstRoute` map as it was defined first in the configuration.
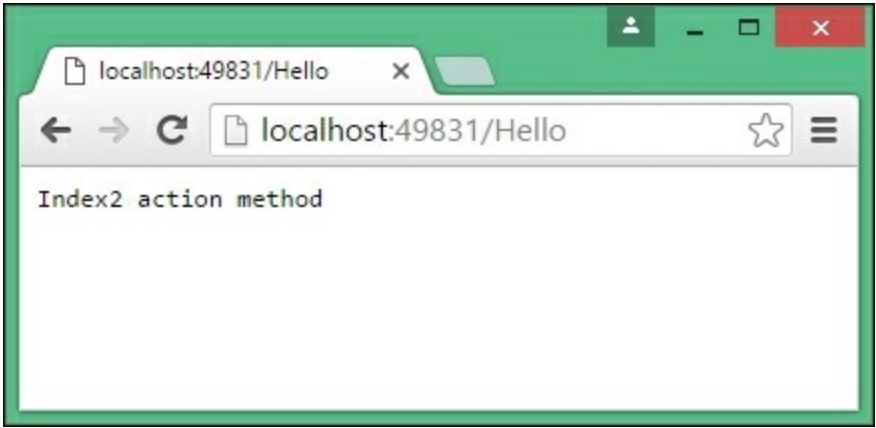
If the routing engine could not map the incoming URL to any of the mapping routes, we get an `HTTP 404 error`, meaning that no resource could be found. You can see the status (200 means *OK*, 404 means *No resource found*) by looking at the **Network** tab in the developer tools as shown in the following screenshot:
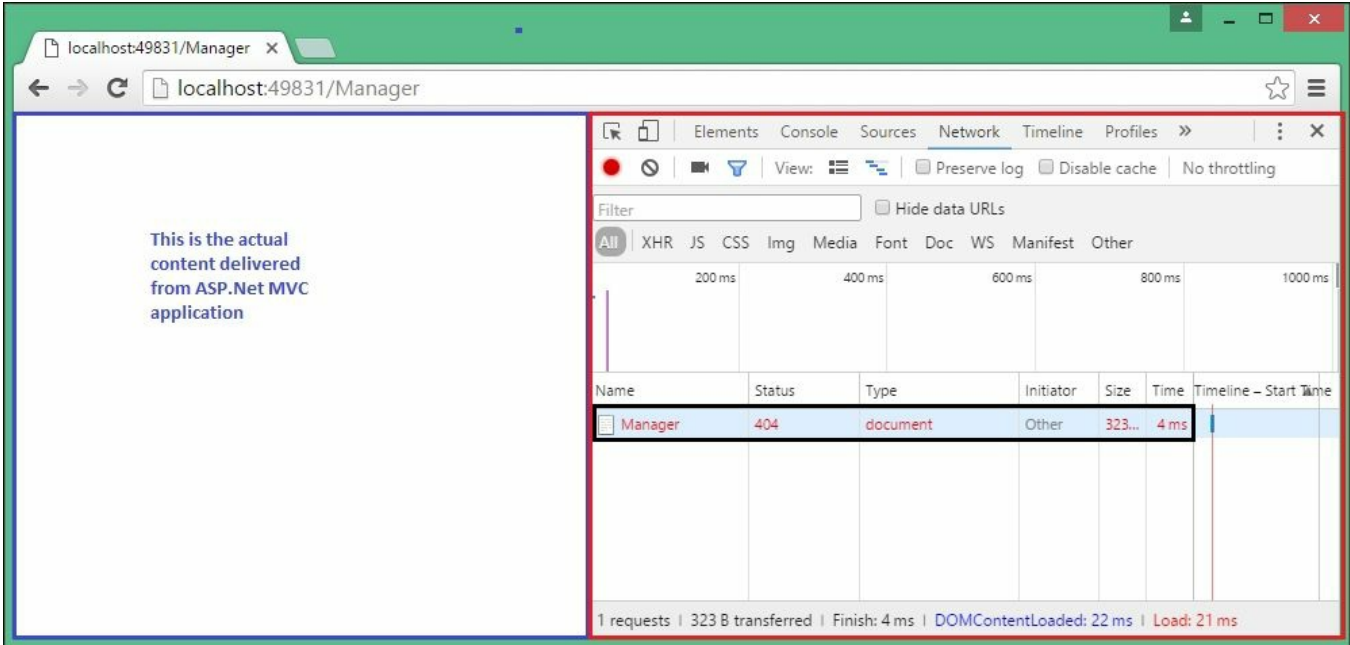
Attribute-based routing

Until now, we have used convention-based routing. In convention-based routing, we define the routing templates (which are just parameterized strings) in a centralized place these are applicable to all the available controllers. The problem with convention-based routing is that, if we want to define different URL patterns for different controllers, we need to define a custom URL pattern that is common to all the controllers. This makes things difficult.

There is another option for configuring the routing engine-attribute-based routing. In attribute-based routing, instead of configuring all the routing in a centralized location, the configuration will happen at the controller level.

Let us see an example of attribute-based routing.

First, let us remove the convention-based routing that we created earlier in the `Configure` method in the `startup.cs` class file:

```
public void Configure(IApplicationBuilder app)
    {
        app.UseIISPlatformHandler();
        app.UseMvc();
        //app.UseMvc(routes =>
        //{
        //    routes.MapRoute(name: "FirstRoute",
        //                    template: "Hello",
        //                    defaults: new { controller = "Home",
        //                    action = "Index2" });

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

Then, we can configure the routing at the controller itself. In the following code, we have added the routing configuration for the `home` controller that we created earlier:

```
namespace Validation.Controllers
{
    public class HomeController : Controller
    {
        // GET: /<controller>/
        [Route("Home")]
        public IActionResult Index()
        {
```

```
        return Content("Index action method");
    }
    [Route("Home/Index3")]
    public IActionResult Index2()
    {
        return Content("Index2 action method");
    }
}
}
```

We have used the `Route` attribute in the `action` methods of the controller. The value passed in the `Route` attribute will be acting as the URL pattern. For example, when we access the URL `http://localhost:49831/Home/`, the `Index` method of `HomeController` will be called. When we access the URL `http://localhost:49831/Home/Index3`, the `Index2` method of `HomeController` will be called. Please note that the URL pattern and `action` method name do not need to match. In the preceding example, we are calling the `Index2` action method but the URL pattern uses `Index3`, `http://localhost:49831/Home/Index3`.

When you use attribute-based routing and convention-based routing together, attribute-based routing will take precedence.

Route attribute at the controller level

You will notice that, with the URL pattern for the `action` methods, `Index` and `Index2`, we repeat the controller name, `Home`, in both URL patterns, `Home` and `Home/Index3`. Instead of repeating the `controller` method name (or any common part in the URL) at the `action` method level, we can define it at the `controller` level.

In the following code, the common part of the URL (`Home`) is defined at the `controller` level and the unique part is defined at the `action` method level. When the URL pattern is getting mapped to the `action` methods of the controller, both route parts (at the `controller` level and at the `action` method level) are merged and matched. So there will be no difference between the routes defined earlier and those that follow.

If you want two parameters in attribute-based routing, you can pass them within curly braces. In the following example, we did this for the `SayHello` action method.

For example, the URL pattern `http://localhost:49831/Home/Index3`, will still get mapped to `Index2` method of the `Homecontroller`:

```
namespace Validation.Controllers
{
    [Route("Home")]
    public class HomeController : Controller
    {
        // GET: /<controller>/
        [Route("")]
        public IActionResult Index()
        {
            return Content("Index action method");
        }

        [Route("Index3")]
        public IActionResult Index2()
        {
            return Content("Index2 action method");
        }

  [Route("SayHello/{id}")]
        public IActionResult SayHello(int id)
        {
            return Content("Say Hello action method"+id);
        }
    }
}
```

Passing routing values in HTTP action verbs in the Controller

Instead of passing the routing values as `Route` attributes, we can even pass the routing values in HTTP action verbs such as `HTTPGet` and `HTTPPost`.

In the following code, we have used the `HTTPGet` attribute to pass the route values. For the `Index` method, we did not pass any value and hence no route value will get appended to the route value defined at the `controller` method level. For the `Index2` method, we are passing the value `Index3` and `Index3` will get appended to the route value defined at the `controller` level. Please note that only URLs with `GET` methods will be mapped to the `action` methods. If you access the same URL pattern with the `POST` method, these routes will not get matched and hence these `action` methods will not get called.

```
namespace Validation.Controllers
{
    [Route("Home")]
    public class HomeController : Controller
    {
        // GET: /<controller>/
        [HttpGet()]
        public IActionResult Index()
        {
            return Content("Index action method");
        }

        [HttpGet("Index3")]
        public IActionResult Index2()
        {
            return Content("Index2 action method");
        }
    }
}
```

Route Constraints

Route Constraints enable you to constrain the type of values that you pass to the controller action. For example, if you want to restrict the value to be passed to the `int` type `int`, you can do so. The following is one such instance:

```
[HttpGet("details/{id:int?}")]
    public IActionResult Details(int id)
    {
      return View();
    }
```

ASP.NET 5 (ASP.NET Core) even supports default parameter values so that you can pass the default parameters:

```
[HttpGet("details/{id:int = 123}")]
    public IActionResult Details(int id)
    {
      return View();
    }
```

Summary

In this chapter, we have learned about routing and how it works. We learned about different kinds of routing available. We discussed convention-based routing and attribute-based routing with different examples. We also discussed route constraints and the default parameter values that could be passed.

In the next chapter, we are going to see how we can make the application look good.

Chapter 8. Beautifying ASP.NET MVC Applications with Bootstrap

You might have created an application with all the required functionalities. It may even work perfectly without any issues in all scenarios. But the success of your application depends on how well your users can access it. Your application should look good (if not great) and be user-friendly in order for it to be a success.

In this chapter, you are going to learn about the following topics:

- Role of HTML and CSS in the ASP.NET Core application
- Characteristics of front-end frameworks and different frameworks available
- Bootstrap and its grid system along its features
- CSS classes available in Bootstrap for form elements such as input and select elements
- CSS classes for different types of HTML elements such as table
- Using Bootstrap in your ASP.NET Core application

Before discussing how to make our application look good, let us take a step back and discuss the roles that HTML and CSS play in your application.

Knowing HTML and CSS

As mentioned earlier, all browsers can only understand HTML, CSS, and JavaScript. So, the applications that you build should produce output as HTML, CSS, and JavaScript. This holds true for web applications built using other technologies such as Java or Ruby on Rails. Having said that, we will only discuss HTML and CSS.

HTML (**Hyper Text Markup Language**) is used to structure the content in your web pages. For example, you can add content in a `title` tag so that it will be available in a browser's tab or window. Let us see an example.

Open any text editor (you can even use Notepad), type the following HTML content and the save file as `Bootstrap.html`. Please note the extension `.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title> Beautify your ASP.NET MVC applications using Bootstrap
</title>
</head>
<body>
    <h1> Bootstrap </h1>
    <p>
        Bootstrap is the most popular HTML, CSS, and JS framework
for developing responsive, mobile first projects on the web.
    </p>
</body>
</html>
```

The first line tells that the HTML content follows HTML 5 standards (the latest and the current version of HTML) and should be interpreted as such. The `html` tag tells the browser it is the start of an HTML document. Information in the `head` tag represents metadata and tells the browser to the web page rather than the web page itself. Examples include the page title, description about the page, and keywords for the search engine optimization. All the contents of the `body` tag will be shown in the browser's main window. In the preceding HTML code, we have made `Bootstrap` the heading and the rest of the content has been made as a paragraph.

Open the file in the browser and you should see something like the following screenshot:
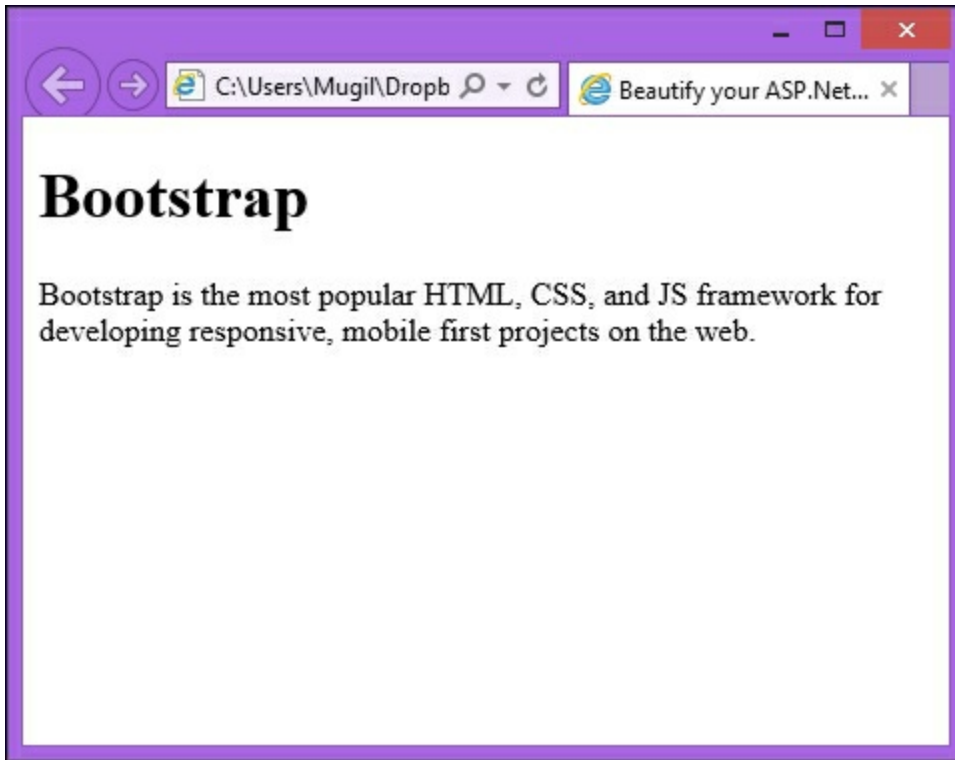
You will notice the content placed in the `title` tag shown as the browser's tab title, the heading content is made bold and bigger in size, and the paragraph starts on a new line.

CSS is all about styling. You can use CSS to customize how each of the elements in your web page looks. You can change the color of the button, the font of the heading text, the border size of a table, and so on. You can include CSS styles either inline or using a separate file. If you are using inline CSS, it should be within a `style` tag. If you are using external CSS, we can make use of a `link` tag and refer to the external CSS file.

CSS is nothing but a set of rules used for the presentation. Each rule consists of two parts—a selector for which a declaration has to be applied and a declaration containing the styling information. The styling information has a property and a value for the property.

Let us take the following simple CSS rule:

```
h1{
    color : #0094ff;
}
```

This CSS rule states that all the heading text should be in a blue color. `h1` is the selector, which tells the browser the following declaration has to be applied for all

`h1` tags. In the declaration, we are setting the blue color (`#0094ff` is blue in hexadecimal format).

The following is the updated HTML file where I've updated the CSS styles (highlighted in bold):

```html
<!DOCTYPE html>
<html>
<head>
    <title> Beautify your ASP.NET MVC applications using Bootstrap </title>
    <style type="text/css">

        body{
            font-family:Arial,Verdana,sans-serif;
        }

        h1{
            color : #0094ff;
        }

        p {
            color: #5f5e5e;
        }
    </style>

</head>
<body>
    <h1> Bootstrap </h1>
    <p>
        Bootstrap is the most popular HTML, CSS, and JS framework
for developing responsive, mobile first projects on the web.
    </p>
</body>
</html>
```

When you open the file in a browser after making the style changes, you will get the following output:

Having said that, you need to create CSS rules to make the elements in your web application look good. But creating different styles for each element is a time-consuming and tedious task. You can choose from any of the frontend frameworks available.

Any application should have the following characteristics:

- **Consistency**: The elements and the controls that are being used by your application should be familiar to the user. For example, if you use a drop-down list, the user should be able to select only one value from it.
- **Responsive**: The application that you build should look good across all devices of different sizes. The code that you write should adapt to the screen size of the device of your users.
- **Mobile-friendly**: This is related to the preceding point. These days, many applications being accessed from mobile devices rather than desktops or laptops. We have to make sure that the application that we build will look great on mobile devices.
- **Customizable**: If you are going to use any front-end application framework, it should be customizable according to your needs. For example, if you want to update the heading text color, you should be able to update or override the CSS file to make the necessary changes.
- **Easy to get started**: The learning curve for learning your front-end framework should be minimal as you should be spending time on delivering value to the

customer—building and delivering the solution. We are not getting paid to learn a new fancy front-end framework.

There are few front-end frameworks available such as Bootstrap, **Foundation**, and **PureCSS**. In this chapter, we are going to use the Bootstrap framework as it is the most widely used front-end framework.

Bootstrap

Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the web and you can access it at http://getbootstrap.com/ .
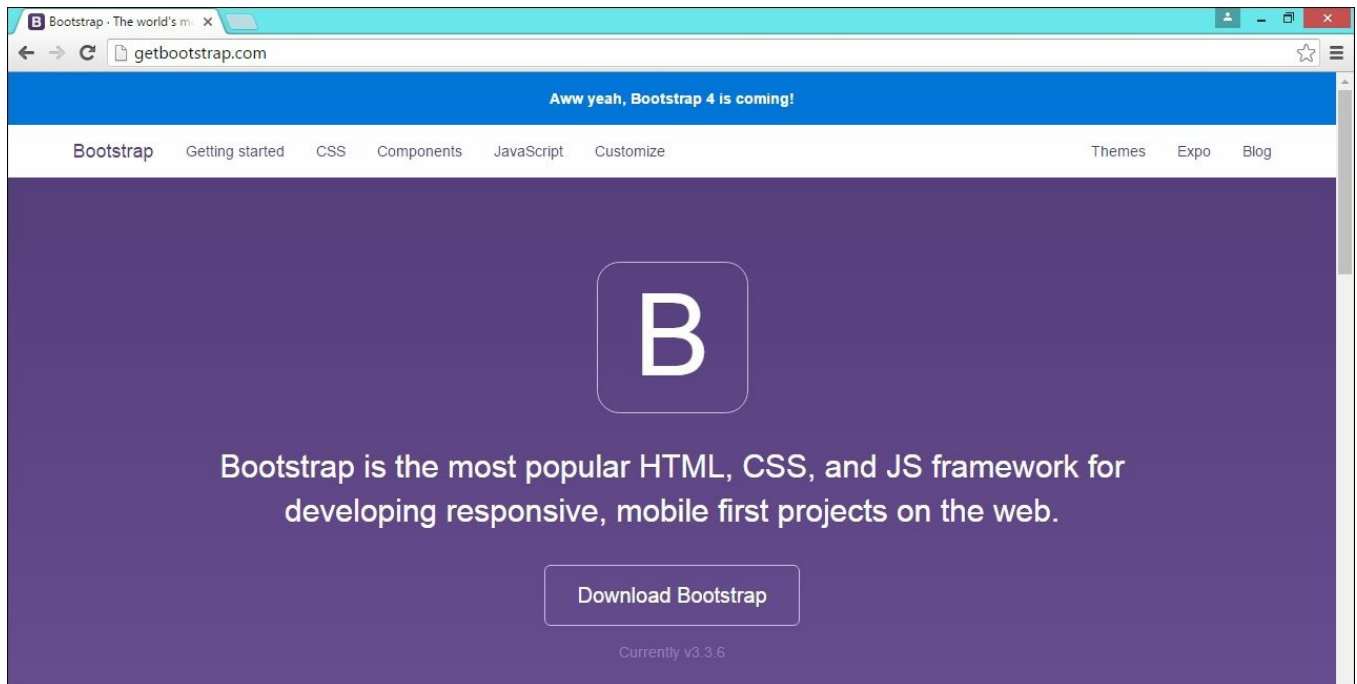
Bootstrap has many features that meet the various needs of a web application. It has different CSS classes for each of the HTML elements so that you can use it without touching any CSS code. However, if you wish to override it, you can do so.

Let us look at each of the features of Bootstrap in brief.

Bootstrap Grid system

The grid system in Bootstrap helps you to create a responsive layout for your application. This feature makes your application look great in all devices of different shapes including mobile devices.

Bootstrap provides a fluid grid system, which scales up to twelve columns as the device or viewport size increases. You can think of grid system as being like columns in an Excel sheet (like the one in the following screenshot):

	A	B	C	D	E	F	G	H	I	J	K	L
1	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8	Column 9	Column 10	Column 11	Column 12
2	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
3	.col-md-2		.col-md-2		.col-md-2		.col-md-2		.col-md-2		.col-md-2	
4	.col-md-3			.col-md-3			.col-md-3			.col-md-3		
5	.col-md-4				.col-md-4				.col-md-4			
6	.col-md-6						.col-md-6					
7	.col-md-8								.col-md-4			

We can combine multiple columns to create a wider column. In the second row in the preceding screenshot, we have used a single column (using class .col-md-1) twelve times. In the third row, we are using a CSS class (.col-md-2) to create six wider columns instead of twelve shorter columns. Likewise, we are creating a lesser number of wider columns.

Forms

All form controls receive global styling when you use Bootstrap in your application. The following is one such example (the screenshot is taken from the Bootstrap site):

The following is the code for the preceding form. Related HTML elements are grouped together in the form. For example, the labels for the Email and **Email address** input text boxes are grouped together. The same is the case for **Password** and **File input**.

```html
<form>
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" placeholder="Email">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1" placeholder="Password">
  </div>
  <div class="form-group">
    <label for="exampleInputFile">File input</label>
    <input type="file" id="exampleInputFile">
    <p class="help-block">Example block-level help text here.</p>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox"> Check me out
    </label>
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Inline forms

Inline forms are forms where all the form elements are on the same line (as shown in the following screenshot). We need to add the class `form-inline` to the `form` element.

| Name | Jane Doe | Email | jane.doe@example.com | Send invitation |

```
<form class="form-inline">
  <div class="form-group">
    <label for="exampleInputName2">Name</label>
    <input type="text" class="form-control" id="exampleInputName2" placeholder="Jane Doe">
  </div>
  <div class="form-group">
    <label for="exampleInputEmail2">Email</label>
    <input type="email" class="form-control" id="exampleInputEmail2" placeholder="jane.doe@example.com">
  </div>
  <button type="submit" class="btn btn-default">Send invitation</button>
</form>
```

Horizontal forms

In horizontal forms, we have each element group on a separate line; **Email** label, Email input on a line and **Password** label, Password input on a line (as shown in the following screenshot):

Email	Email
Password	Password
	☐ Remember me
	Sign in

To apply the horizontal form styling, we just need to add the class `form-horizontal` as in the following code. Just like other forms, we need to group the related HTML elements together by applying the CSS class `form-group`:

```html
<form class="form-horizontal">
  <div class="form-group">
    <label for="inputEmail3" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-10">
      <input type="email" class="form-control" id="inputEmail3" placeholder="Email">
    </div>
  </div>
  <div class="form-group">
    <label for="inputPassword3" class="col-sm-2 control-label">Password</label>
    <div class="col-sm-10">
      <input type="password" class="form-control" id="inputPassword3" placeholder="Password">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <div class="checkbox">
        <label>
          <input type="checkbox"> Remember me
        </label>
      </div>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-default">Sign in</button>
    </div>
  </div>
</form>
```

Table CSS classes

For basic styling, add the base CSS class `table` to the `table` HTML element as shown in the following screenshot:

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

```
<table class="table">
  ...
</table>
```

Striped tables

In a striped table, the background of alternate rows will be of the same color. In the following screenshot, the background color of the first row and third row are the same color. You can apply the `table-striped` class to apply the striped table behavior to the table HTML element.

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

```
<table class="table table-striped">
  ...
</table>
```

Hover tables

When you move your mouse over any of the rows in the table, the background color of the row is changed. This hover behavior can be achieved by applying the CSS class `table-hover` along with the class `table` to the HTML table element.

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

```
<table class="table table-hover">
  ...
</table>
```

Bordered tables

We can have a bordered table (as in the following screenshot), if we apply the CSS class `table-bordered` to the `table` element.

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

```
<table class="table table-bordered">
  ...
</table>
```

Contextual classes in table

There are times when you want to highlight rows based on the data value. For example, if you are showing inventory data in tabular format, you might want to highlight the rows with a red background color for the items whose count is less than the stipulated count. In this case, you can apply a `danger` class to table rows to highlight them in a red color. There are different types of contextual class available for highlighting in different colors. You can apply these classes to individual cells instead of the complete rows.

Contextual classes

Use contextual classes to color table rows or individual cells.

Class	Description
`.active`	Applies the hover color to a particular row or cell
`.success`	Indicates a successful or positive action
`.info`	Indicates a neutral informative change or action
`.warning`	Indicates a warning that might need attention
`.danger`	Indicates a dangerous or potentially negative action

EXAMPLE

#	Column heading	Column heading	Column heading
1	Column content	Column content	Column content
2	Column content	Column content	Column content
3	Column content	Column content	Column content
4	Column content	Column content	Column content
5	Column content	Column content	Column content
6	Column content	Column content	Column content
7	Column content	Column content	Column content
8	Column content	Column content	Column content

Buttons

There are different styling options available for making buttons appear in different colors. For all the buttons, the base button class `btn` has to be applied:

```
Default   Primary   Success   Info   Warning   Danger   Link

<!-- Standard button -->
<button type="button" class="btn btn-default">Default</button>

<!-- Provides extra visual weight and identifies the primary action in a set of buttons -->
<button type="button" class="btn btn-primary">Primary</button>

<!-- Indicates a successful or positive action -->
<button type="button" class="btn btn-success">Success</button>

<!-- Contextual button for informational alert messages -->
<button type="button" class="btn btn-info">Info</button>

<!-- Indicates caution should be taken with this action -->
<button type="button" class="btn btn-warning">Warning</button>

<!-- Indicates a dangerous or potentially negative action -->
<button type="button" class="btn btn-danger">Danger</button>

<!-- Deemphasize a button by making it look like a link while maintaining button behavior -->
<button type="button" class="btn btn-link">Link</button>
```

The `btn-primary` button class is used to highlight the button in blue whereas the `btn-success` button class highlights the button in green. In the preceding screenshot, different options for styling the button are shown.

Button sizes

You can change the size of the button according to your needs. The `btn-lg` class can be applied to a large button and the `btn-sm` class can be applied to buttons to make them appear small. The following are the different options available to control the size of the button.

Large button	Large button
Default button	Default button
Small button	Small button
Extra small button	Extra small button

```html
<p>
  <button type="button" class="btn btn-primary btn-lg">Large button</button>
  <button type="button" class="btn btn-default btn-lg">Large button</button>
</p>
<p>
  <button type="button" class="btn btn-primary">Default button</button>
  <button type="button" class="btn btn-default">Default button</button>
</p>
<p>
  <button type="button" class="btn btn-primary btn-sm">Small button</button>
  <button type="button" class="btn btn-default btn-sm">Small button</button>
</p>
<p>
  <button type="button" class="btn btn-primary btn-xs">Extra small button</button>
  <button type="button" class="btn btn-default btn-xs">Extra small button</button>
</p>
```

Contextual colors

Based on the context, you might want to change the color of the text. For example, you might want to make the text appear in green if the previous operation is successful. For an unsuccessful operation, you might want to show the error message in a red. In such scenarios, you might use this helper CSS classes to show them in different colors.

Fusce dapibus, tellus ac cursus commodo, tortor mauris nibh.

Nullam id dolor id nibh ultricies vehicula ut id elit.

Duis mollis, est non commodo luctus, nisi erat porttitor ligula.

Maecenas sed diam eget risus varius blandit sit amet non magna.

Etiam porta sem malesuada magna mollis euismod.

Donec ullamcorper nulla non metus auctor fringilla.

```
<p class="text-muted">...</p>
<p class="text-primary">...</p>
<p class="text-success">...</p>
<p class="text-info">...</p>
<p class="text-warning">...</p>
<p class="text-danger">...</p>
```

We have seen various features of Bootstrap. Now, let us use Bootstrap to make our application look good. Basically, we have two major components in our view—a form at the top to get the input from the user and a table at the bottom to display the results in a table.

Using Bootstrap in your ASP.NET MVC application

There are different ways to get Bootstrap for your application:

- Refer to the Bootstrap file available at the CDN (Content Delivery Network) in your application
- Download the source code
- Install with Bower
- Compile with Grunt

Of these options, the easiest option is the first one.

Open the layout file (`_Layout.cshtml`) in the application that we created earlier. Include the CSS files at the top (within the `head` tag) and the scripts at the bottom (at the end of the `body` tag):

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap
.min.css" integrity="sha384-
1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
    <!-- Optional theme -->
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap
-theme.min.css" integrity="sha384-
fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">

</head>
<body>
    <div>
        @RenderBody()
    </div>

    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
2.2.3.js"></script>
    <script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.
validate.min.js"></script>
```

```
    <script
src="https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unob
trusive.min.js"></script>
    <!-- Latest compiled and minified JavaScript -->
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.m
in.js" integrity="sha384-
0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>
</body>
</html>
```

Installing with Bower

Right click on the **Project** menu and select the **Manage Bower Packages** option from the **context** menu:

	Build	
	Rebuild	
	Clean	
	View	▶
	Analyze	▶
	Publish...	
	Add Application Insights Telemetry...	
	Power Commands	▶
	Scope to This	
	New Solution Explorer View	
	Show on Code Map	
	Add	▶
	Manage NuGet Packages...	
	Restore Packages	Ctrl+Shift+K, Ctrl+Shift+R
	Manage Bower Packages...	
	Manage User Secrets	
	Set as StartUp Project	
	Debug	▶
	Source Control	▶
	Cut	Ctrl+X
	Paste	Ctrl+V
	Remove	Del
	Rename	
	Unload Project	
	Open Folder in File Explorer	

Once you select **Manage Bower Packages**, you will be able to install or uninstall

the Bower package just as you installed or uninstalled the `NuGet` package.

HTML doctype

Bootstrap makes use of certain HTML elements and CSS properties that require the use of HTML 5 doctype. By default, the view that you create in ASP.NET Core will have HTML 5 doctype only. So, we do not need to do anything regarding this.

```
<!DOCTYPE html>

<html lang="en">
...
</html>
```

Let us make the following changes to our screen to make use of Bootstrap:

- Apply the CSS class `form-horizontal` to the form.
- For the label, input, and validation error spans use the CSS `col-sm-2`, `col-sm-4`, and `col-sm-3` classes respectively
- For labels, apply the CSS class `control-label`
- For input HTML elements, the `form-control` CSS class is applied
- For each of the form groups (containing the HTML elements such as label and input), apply the CSS class `form-group`
- For all validation error messages, apply the `text-danger` CSS class so that they will be shown in red
- Apply the `table`, `table-bordered` CSS class to style the table

The following is the complete updated view code; we have used Bootstrap CSS classes to make our application look great:

```
@model Validation.ViewModels.EmployeeAddViewModel

<div>
    <br/>
    <br/>
    <form asp-controller="Employee" asp-action="Index"
method="post" role="form" class="form-horizontal">

        <div class="form-group">
            <label asp-for="Name" class="col-sm-2 control-label">
</label>
            <div class="col-sm-4">
                <input asp-for="Name" class="form-control" />
            </div>
            <div class="col-sm-3 text-danger">
                <span id="validationName" asp-validation-for="Name"
></span>
            </div>
```

```html
        </div>

        <div class="form-group">
            <label asp-for="Designation" class="col-sm-2 control-label"></label>
            <div class="col-sm-4">
                <input asp-for="Designation" class="form-control" />
            </div>
            <div class="col-sm-3 text-danger">
                <span id="validationDesignation" asp-validation-for="Designation" ></span>
            </div>
        </div>


        <div class="form-group">
            <label asp-for="Salary" class="col-sm-2 control-label"></label>
            <div class="col-sm-4">
                <input asp-for="Salary" class="form-control" />
            </div>
            <div class="col-sm-3 text-danger">
                <span id="validationSalary" asp-validation-for="Salary" ></span>
            </div>
        </div>

        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-10">
                <button type="submit" class="btn btn-primary">Submit</button>
            </div>
        </div>


    </form>

</div>

<br /><br /> <br />

<h4> List of employees:</h4> <br />

    <table class="table table-bordered">
        <tr>
            <th> ID </th>
            <th> Name </th>
            <th> Designation </th>
            <th> Salary </th>
        </tr>
        @foreach (var employee in Model.EmployeesList)
```

```
                {
                    <tr>
                        <td>@employee.EmployeeId</td>
                        <td>@employee.Name</td>
                        <td>@employee.Designation</td>
                        <td>@employee.Salary</td>
                    </tr>
                }
            </table>
```

After making the preceding changes, when you run the application, your screen should look something like the following:

Summary

In this chapter, we have learned about the role of HTML and CSS in any web application including the ASP.NET Core application. We have analyzed the needs of a frontend framework and discussed the features of Bootstrap—the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the web. We have discussed CSS and components available in Bootstrap for different types of HTML elements with examples. Finally, we have discussed how to incorporate Bootstrap in our ASP.NET Core application.

Chapter 9. Deployment of ASP.NET Core Application

Once we have completed the development for our ASP.NET core application, we need to deploy the application so that it can be accessed by our users.

In any application, irrespective of whether it is the web, desktop, or mobile application, not all the functionalities have been achieved through code. In fact, you should not try to achieve everything through code.

In this chapter, you are going to learn about the following topics:

- Configuration in the ASP.NET Core application
- Signing up to the Microsoft Azure platform
- Deploying the ASP.NET Core application to the Azure Cloud platform

If you have built a web application using any of the previous versions of ASP.NET MVC, there will be a file by the name of `Web.config` (an XML file) where you can configure all the dependencies for your application. But in ASP.NET Core, there will be no `Web.config` file in your solution:
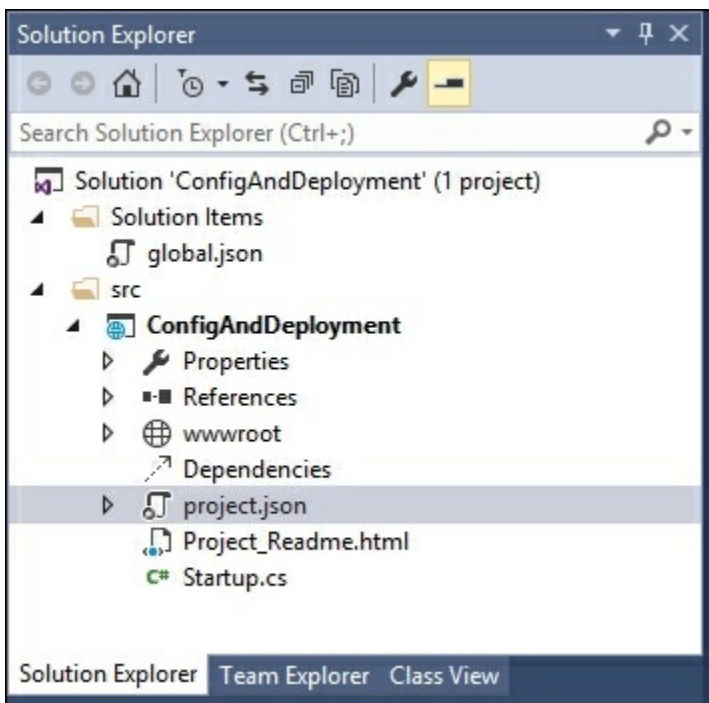
Instead, we have `project.json` (a JSON file), where we will configure the dependencies for your application. Before discussing the contents of `project.json`,

let us discuss a bit about JSON.

JSON is an acronym of **JavaScript Object Notation**. It is the open standard data exchange format. It will be in human-readable text and consist of attribute/value pairs. Consider the following JSON, and let's dissect it to see what it represents:

```
{
  "addressess": [
    {
      "DoorNo": 16,
      "Street": "King Street",
      "areaname": "Mascot"
    },
    {
      "DoorNo": 12,
      "Street": "High Street",
      "areaname": "North Sydney"
    }
  ]
}
```

Each piece of data is an attribute value pair, separated by a colon. For example, `"DoorNo": 16` tells that the value for `DoorNo` variable is 16 in the first record. Each attribute value pair (sometimes called a property) is separated by a comma. For example, consider the following three properties:

```
"DoorNo": 16,
"Street": "King Street",
"areaname": "Mascot"
```

Each record or object is contained within a pair of curly braces. For example, the following JSON data represents a record or an object:

```
{
    "DoorNo": 16,
    "Street": "King Street",
    "areaname": "Mascot"
}
```

Similar records can be grouped together and could be formed as an array (of objects). Square brackets are used to represent the array in JSON format as in the following example:

```
"addressess": [
  {
    "DoorNo": 16,
    "Street": "King Street",
    "areaname": "Mascot"
  },
```

```
{
    "DoorNo": 12,
    "Street": "High Street",
        "areaname": "North Sydney"
    }
]
```

If we have to represent the same data in XML format, you can do so as follows. Please note that for each piece of information, we should have a start tag and an end tag (ends with "/"):

```
<addresses>
  <address>
    <DoorNo>16</DoorNo>
    <Street>King Street</Street>
    <areaname>Mascot</areaname>
  </address>

  <address>
    <DoorNo>12</DoorNo>
    <Street>High Street</Street>
    <areaname>North Sydney</areaname>
  </address>
</addresses>
```

The project.json file

All of the project configuration should go into the `project.json` file for the
ASP.NET Core application. The following is the `project.json` file that was created
when using the predefined ASP.NET Core web application template:

```json
{
    "userSecretsId": "aspnet-AzureDeployment-66c8f80c-8186-4ac5-b4c4-9bb79b3f19ef",

    "dependencies": ...,

    "tools": ...,

    "frameworks": ...,

    "buildOptions": ...,

    "runtimeOptions": ...,

    "publishOptions": ...,

    "scripts": ...
}
```

There are different predefined nodes in this JSON file for different functionalities.
Let us take some important nodes in this `project.json` file and discuss them.

The dependencies node

The `dependencies` node lists all the dependencies for your ASP.NET Core application.

The following is a fragment of the `dependencies` node in the ASP.NET Core application. Each dependency is an attribute value pair where the attribute represents the dependency and the value represents the version of the dependency. If you need to provide more information for the dependency, you can have a nested JSON configuration as it is in `Microsoft.NETCore.App`:

```
"dependencies":{
  "Microsoft.NETCore.App":{
    "version": "1.0.0-rc2-3002702",
    "type": "platform"
    },
    "Microsoft.ApplicationInsights.AspNetCore": "1.0.0-rc2-final",
    "Microsoft.AspNetCore.Authentication.Cookies": "1.0.0-rc2-
final",
```

The frameworks node

In this node, we mention the frameworks that we depend on for the ASP.NET Core application. `dotnet5.6` represents the full blown .NET framework and `dnxcore50` represents the .NET Core framework containing the subset of functionalities of the complete .NET framework:

```
"frameworks":{
  "netcoreapp1.0":{
    "imports":[
      "dotnet5.6",
      "dnxcore50",
      "portable-net45+win8"
      ]
    }
  },
```

Microsoft Azure

Microsoft Azure is a cloud computing platform and infrastructure from Microsoft for building, deploying, and managing applications and services. It supports different programming languages and arrays of services.

You can deploy your application in any server with **Internet Information Service (IIS)** in your network. But this restricts your application to being accessed only from within your network, assuming your server could only be accessed from within your network (as in most network setups). In this section, we are going to deploy the ASP.NET Core application in Microsoft Azure so that your users across the globe can access your application.

Signing up to Microsoft Azure

In order for your application to be deployed to Azure, you need to have an account with Azure. You can create an Azure account for free and you'll have sufficient credits to deploy your application for free within the first 30 days (https://azure.microsoft.com/en-in/):
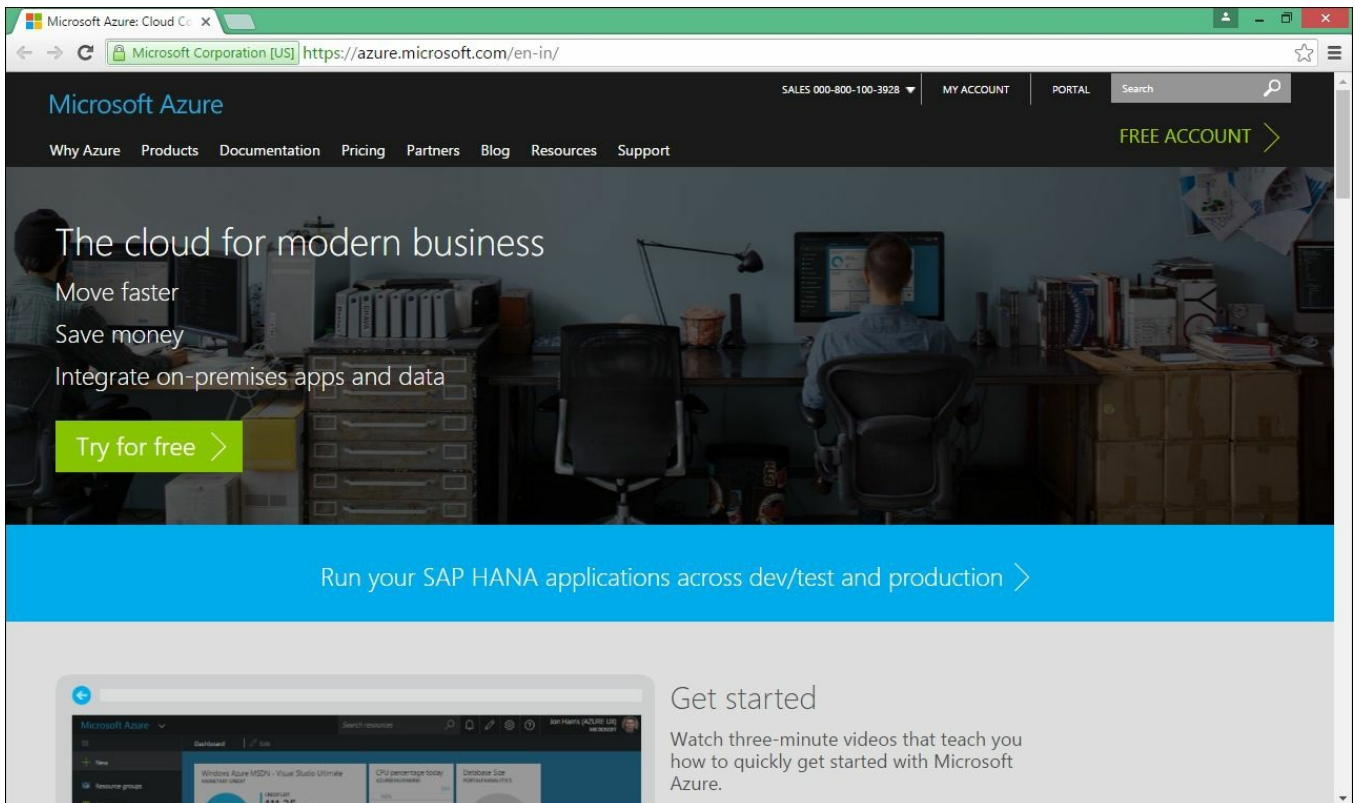
Click the **Try for free** button or **Free Account** link in the top right-hand corner and you'll be forwarded to the following page:

Click the **Start now** button and you'll be redirected to the following page. Enter your Microsoft account credentials and click the **Sign In** button. If you don't have a Microsoft account, you can create one by clicking on the **sign up now** link at the bottom of the page:
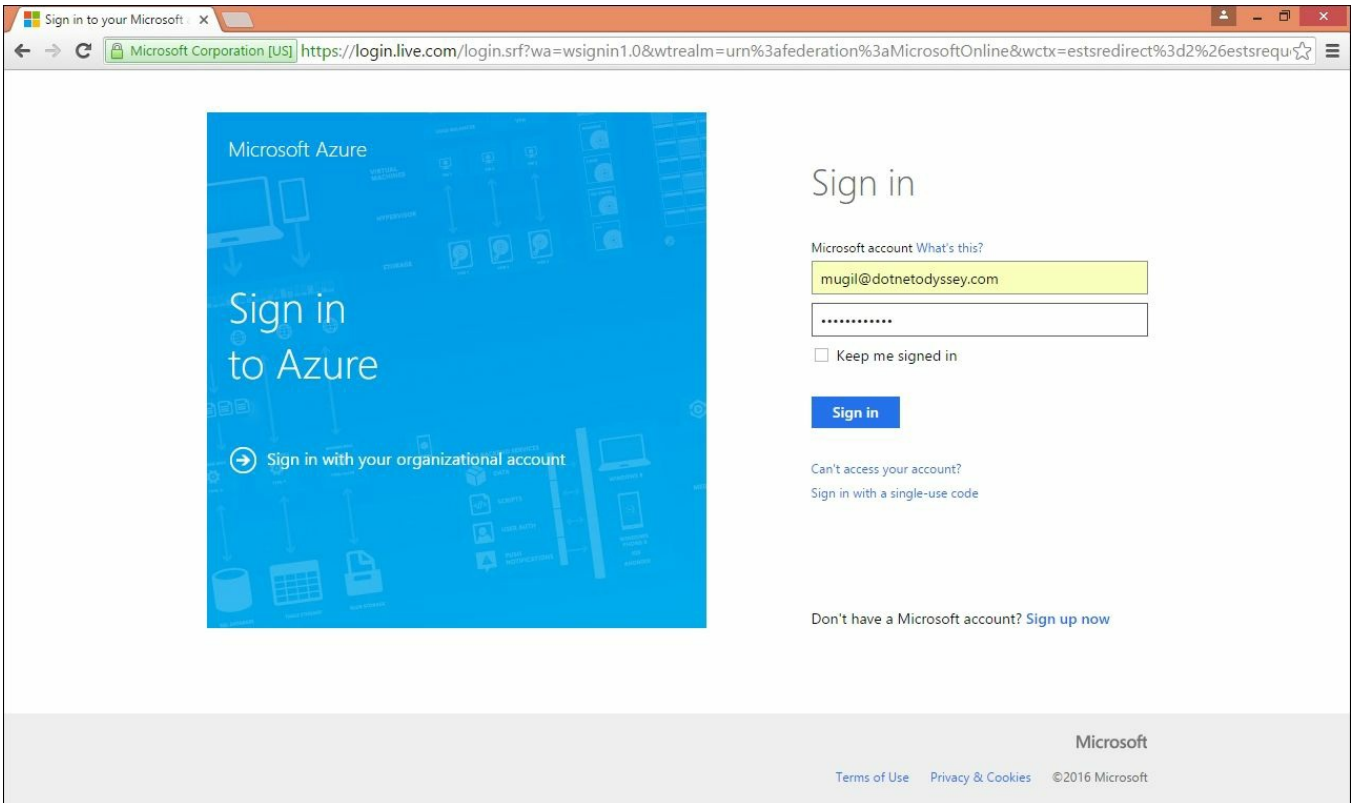
As I have a Microsoft account already, I have signed in with my credentials. Once you have signed-in, you will be asked for details about your country, first name, second name, and your work phone, as follows:
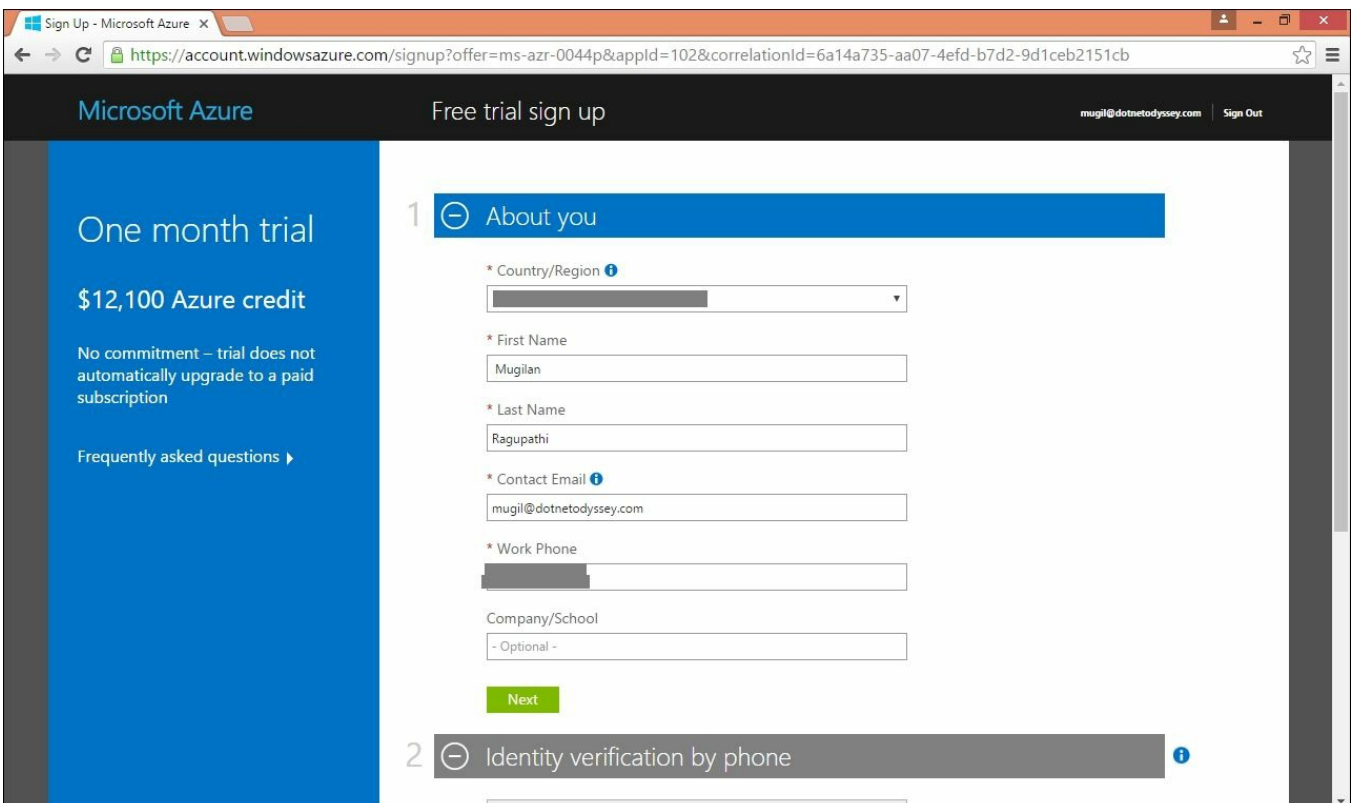
Once you have entered all the necessary details, you will be asked for your country code and phone number so that Azure can text you or call you to verify you are a real person and not a robot ☺. If you choose the option of **text me**, you will get a code to your mobile phone; you need to enter it in the last field and click **Verify Code**:
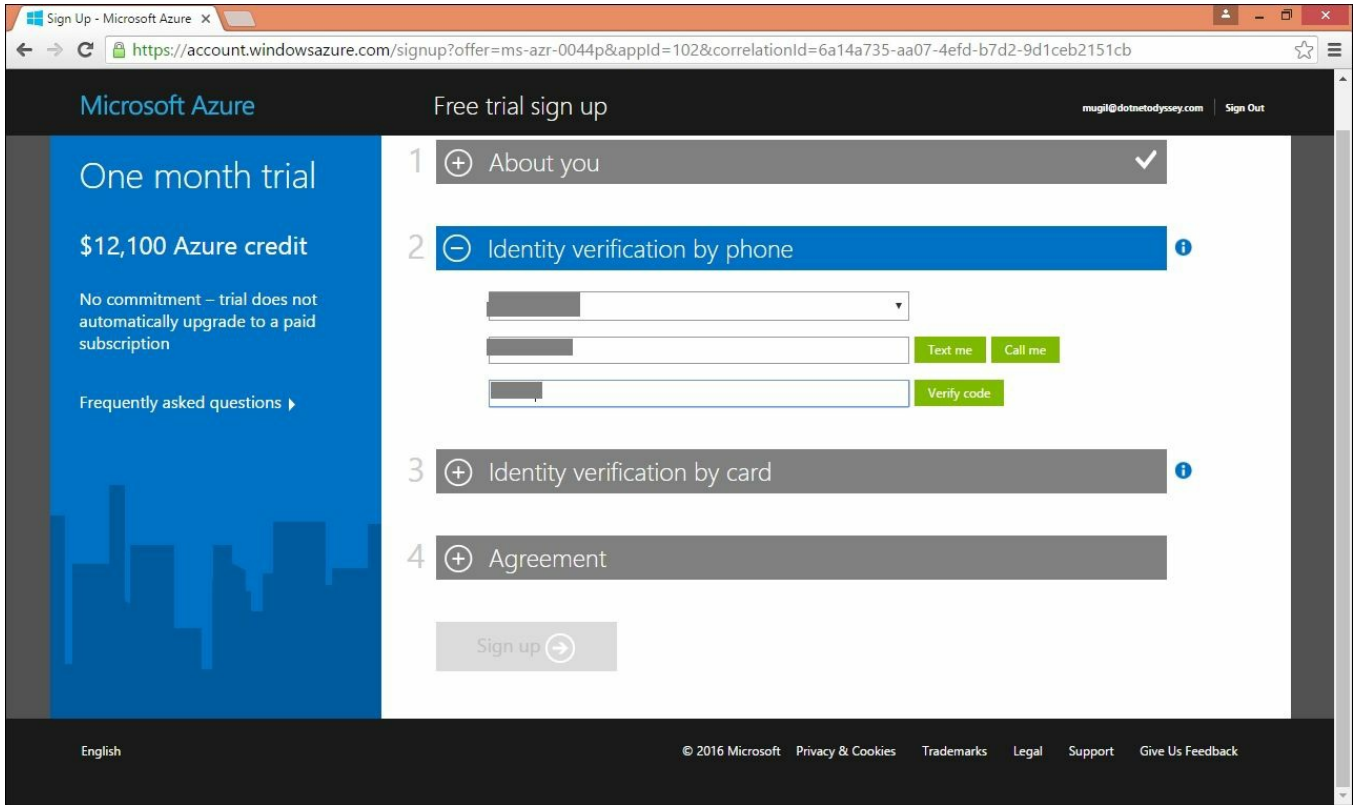
Once you have been verified by phone, you need to enter your credit card information in the following form. You'll be billed for approximately $1 and it will be refunded within five to six business days back to your account. This information is collected to identify the user's identity and the user will not be billed unless the user explicitly opted for the paid service:
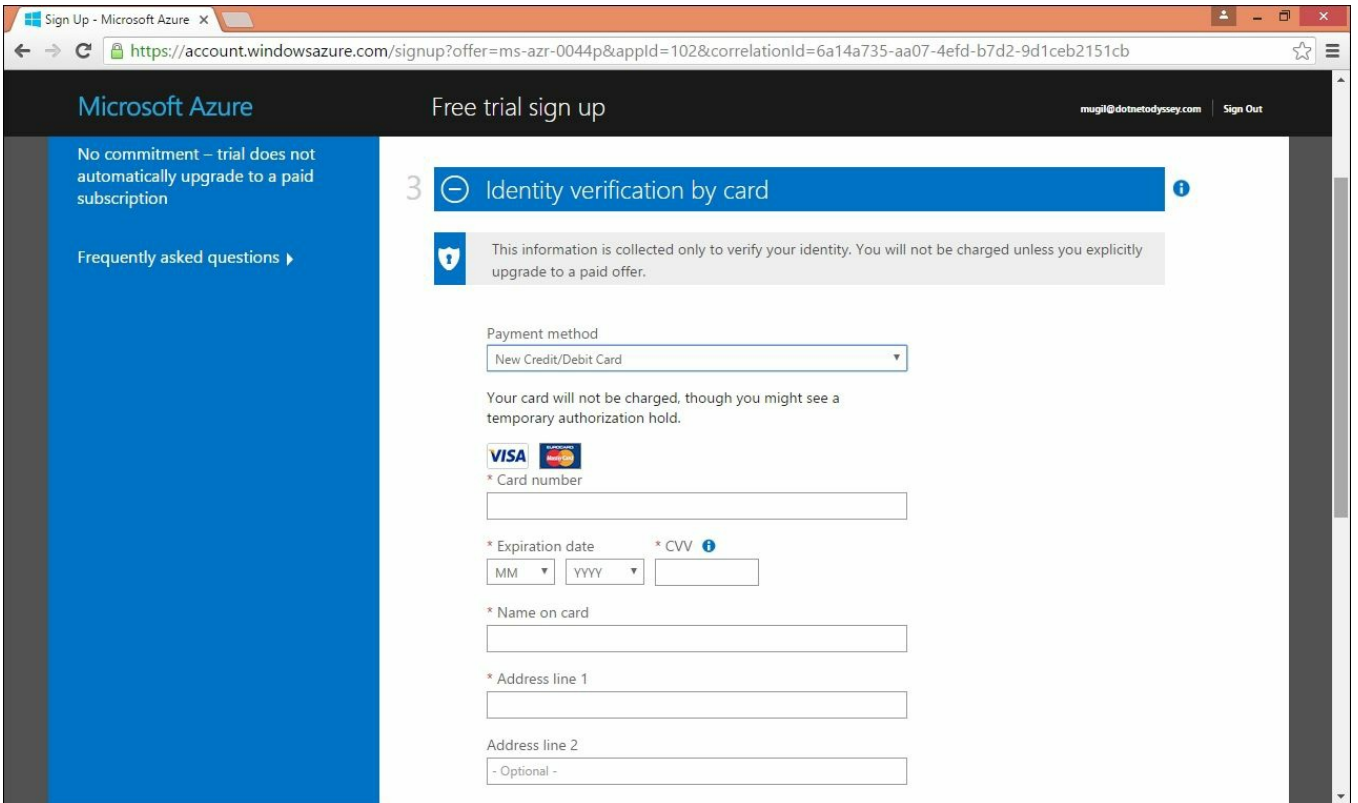
Once you enter your credit card information and click **Next**, you will have to agree to the subscription agreement as the final step in the sign-up process:
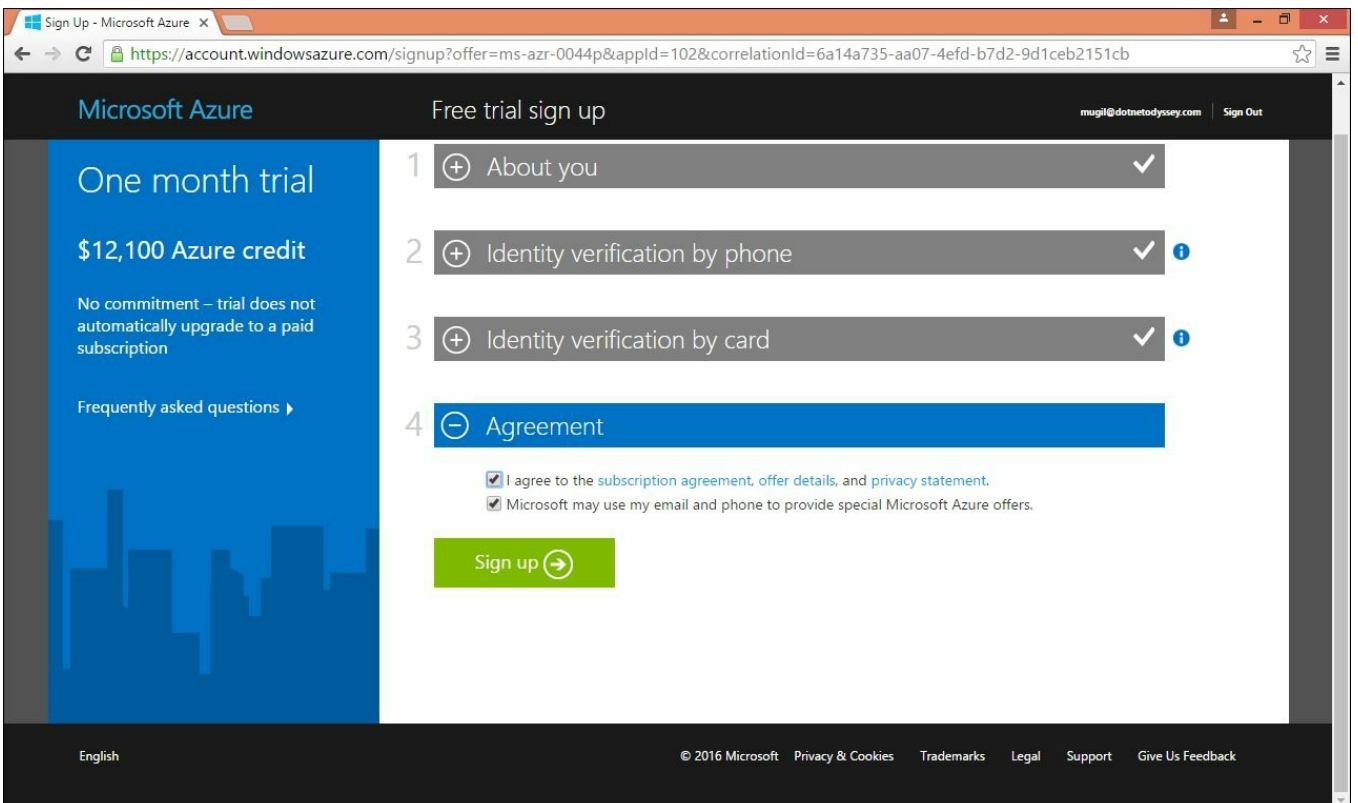
Once you click the **Sign up** button, it will take another five minutes to complete the process. You'll be shown the following screen until the process completes:
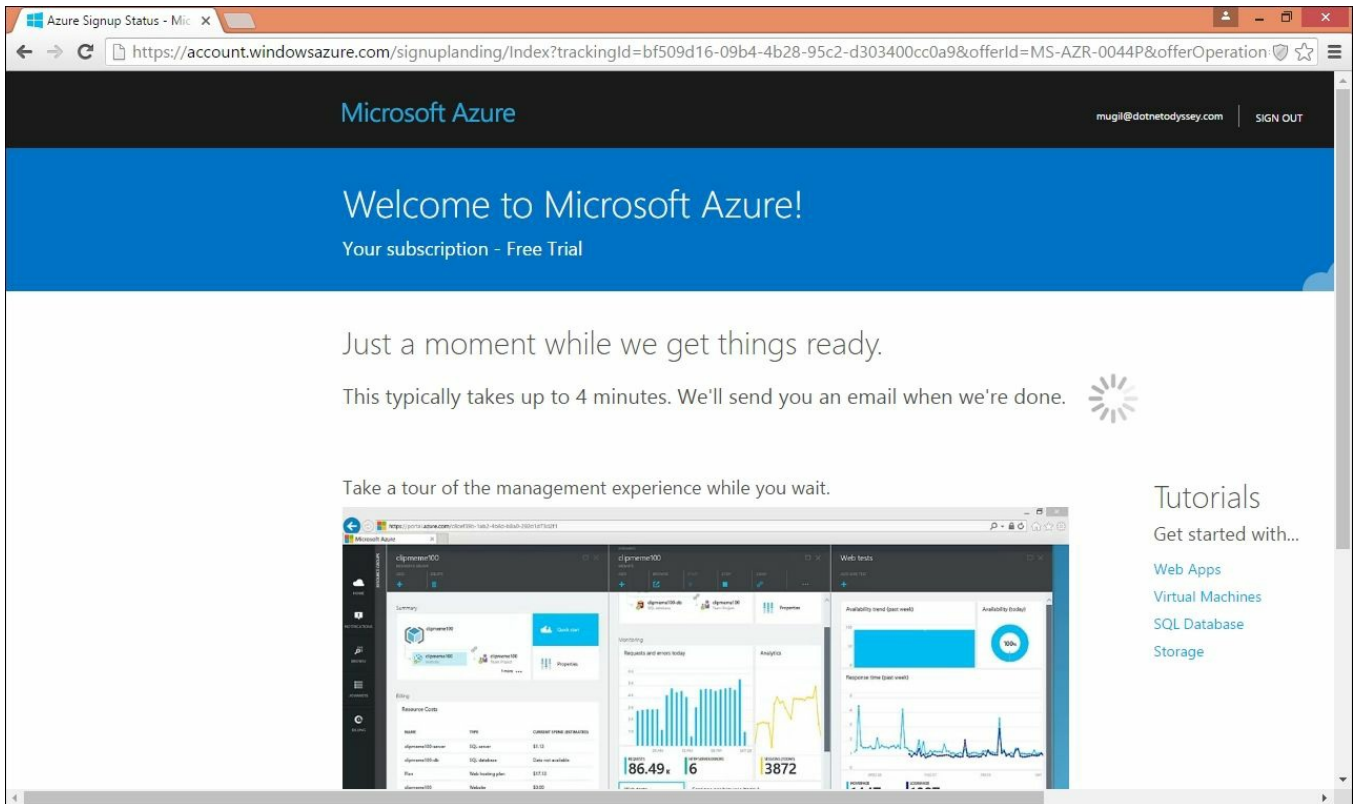
Once the sign-up process completes, you'll be shown the following screen. You'll also get a confirmation e-mail (to the e-mail ID that you gave in the first step) with the subscription details:
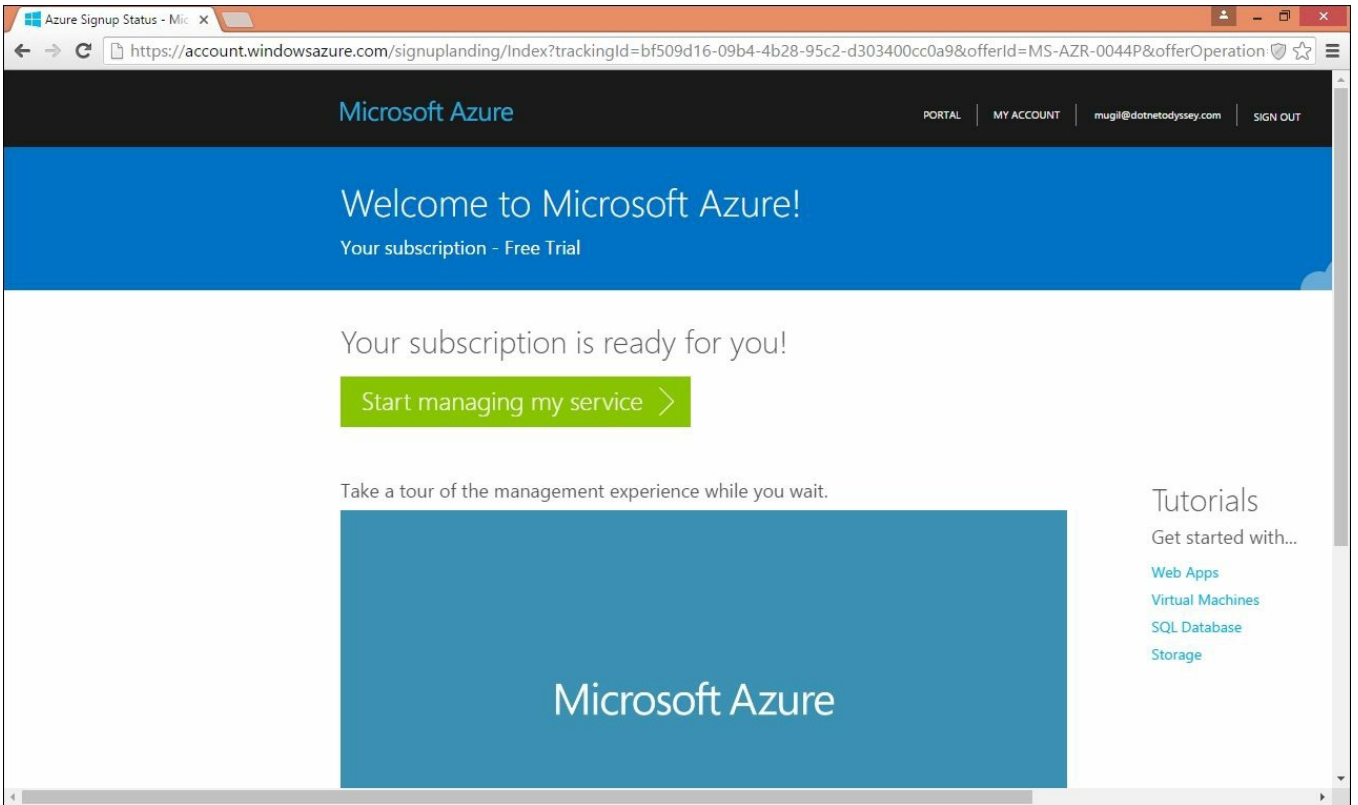
Prerequisites to Azure deployment

In order to publish the ASP.NET Core application to Azure from the Visual Studio 2015 Community Edition, you should have Visual Studio 2015 Update 2 installed (at least) and you should install/enable the SQL Server Data Tools.

Note

If you have the latest version of VS 2015, there is no need to install Update 2.

You can download the Visual Studio 2015 Update 2 from the URL at https://www.visualstudio.com/en-us/news/vs2015-update2-vs.aspx and install it.

To install the SQL Server Data Tools, go to **Control Panel | Programs and Features**. Right-click on the **Microsoft Visual Studio Community 2015** and select the **Change**, option, as shown in the following screenshot:
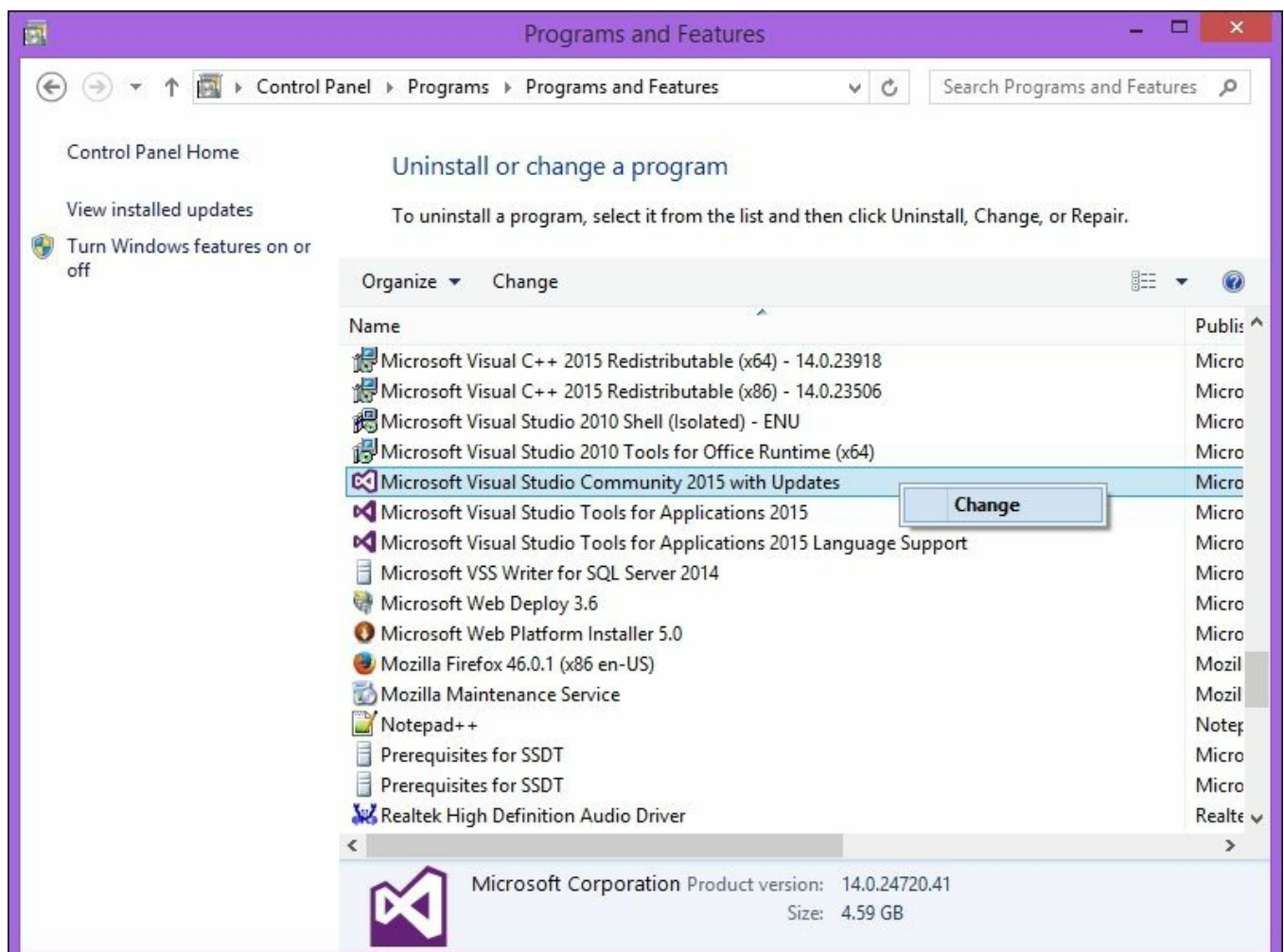
Once you click on **Change** option, you will get the following window—where you

have to choose the **Modify** button. Once you click the **Modify** button, you'll be given an option where you can modify the Visual Studio installation options. I have selected **Microsoft SQL Server Data Tools** , as depicted in the following screenshot:
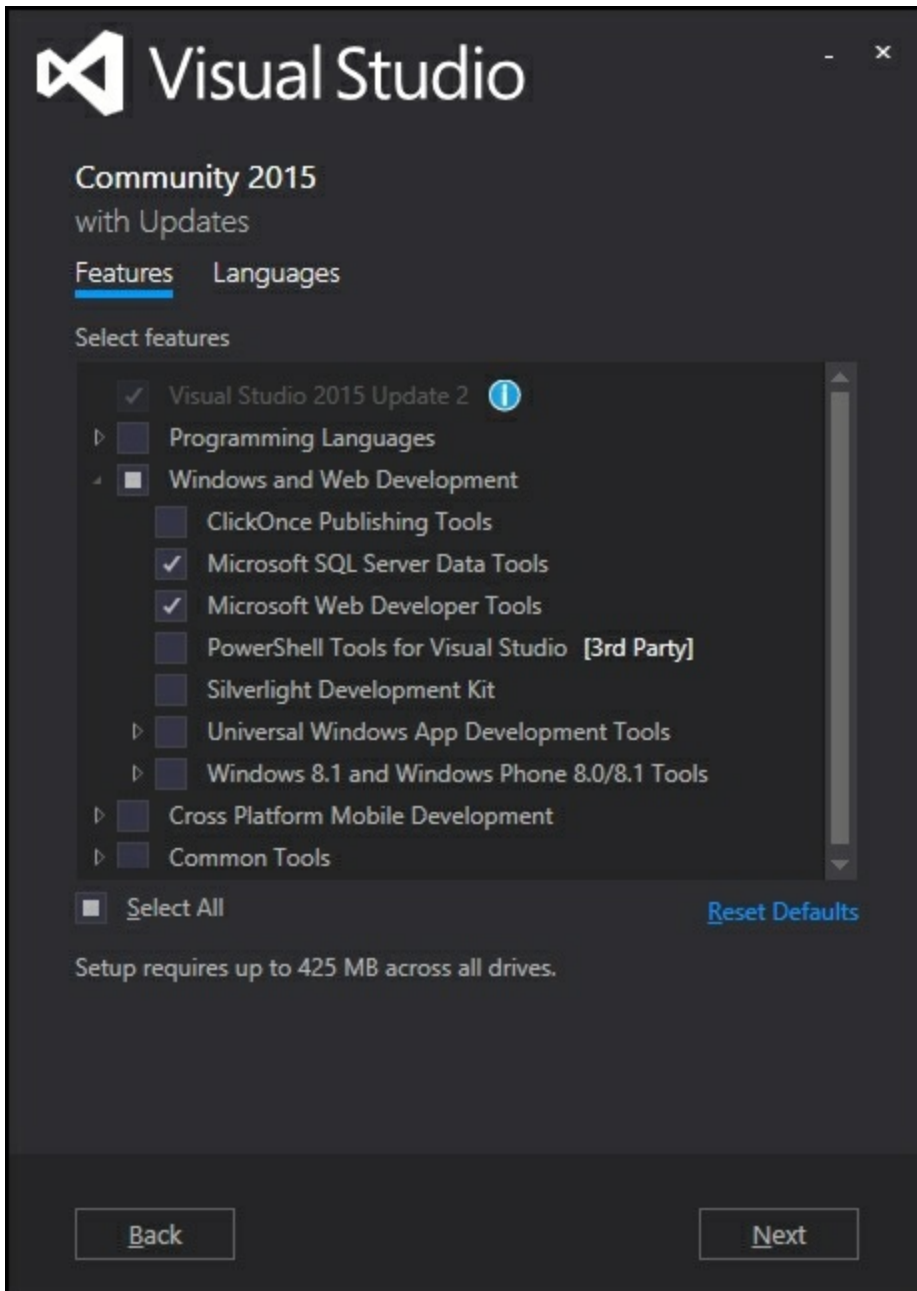
Once you click **Next**, Visual Studio will install the SQL Server Data Tools and once it is completed, you will get the following screen, which shows the setup completion status:

Deploying the ASP.NET Core application in Azure

Let's create an ASP.NET Core application that we can deploy in Microsoft Azure:
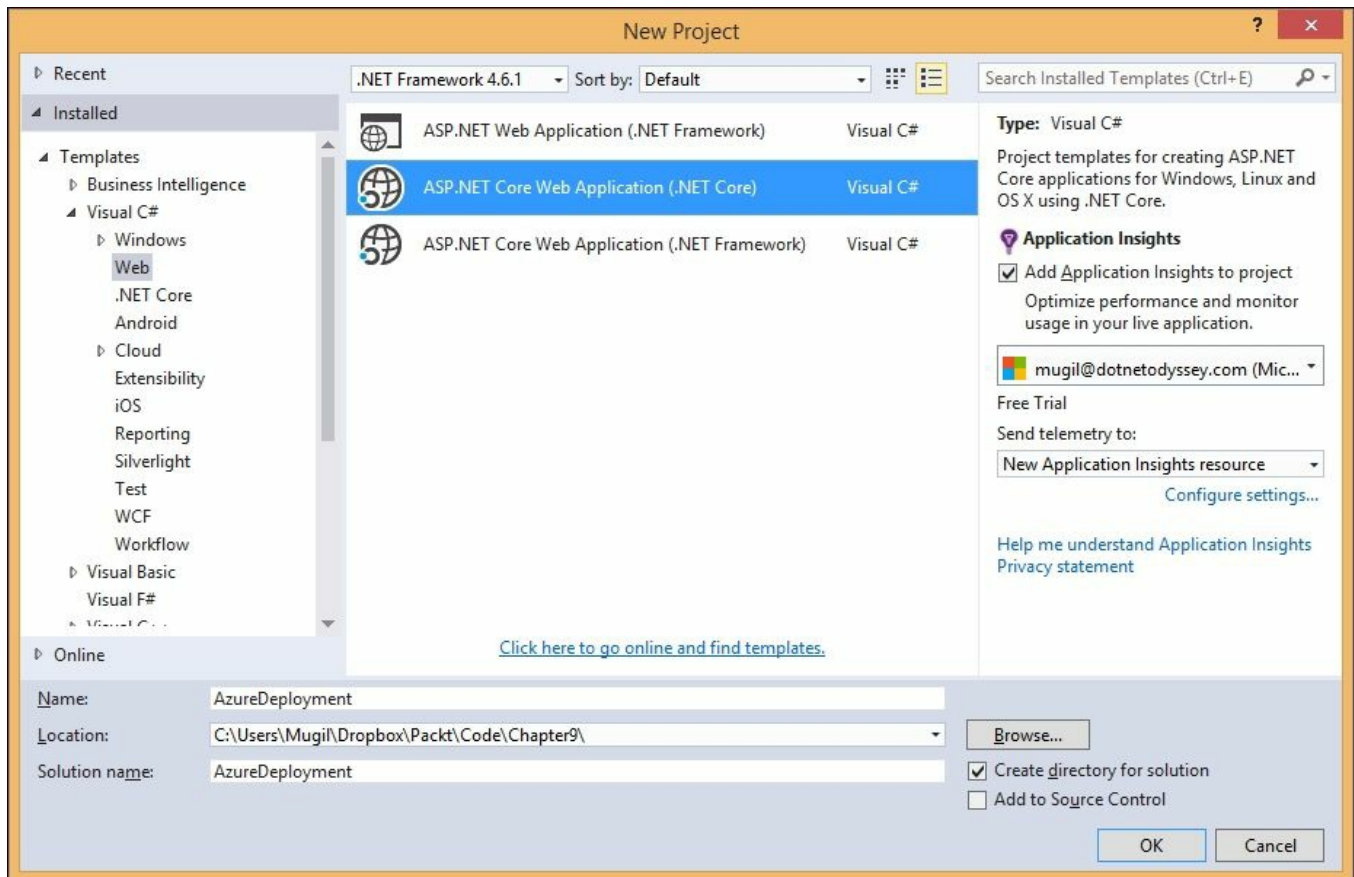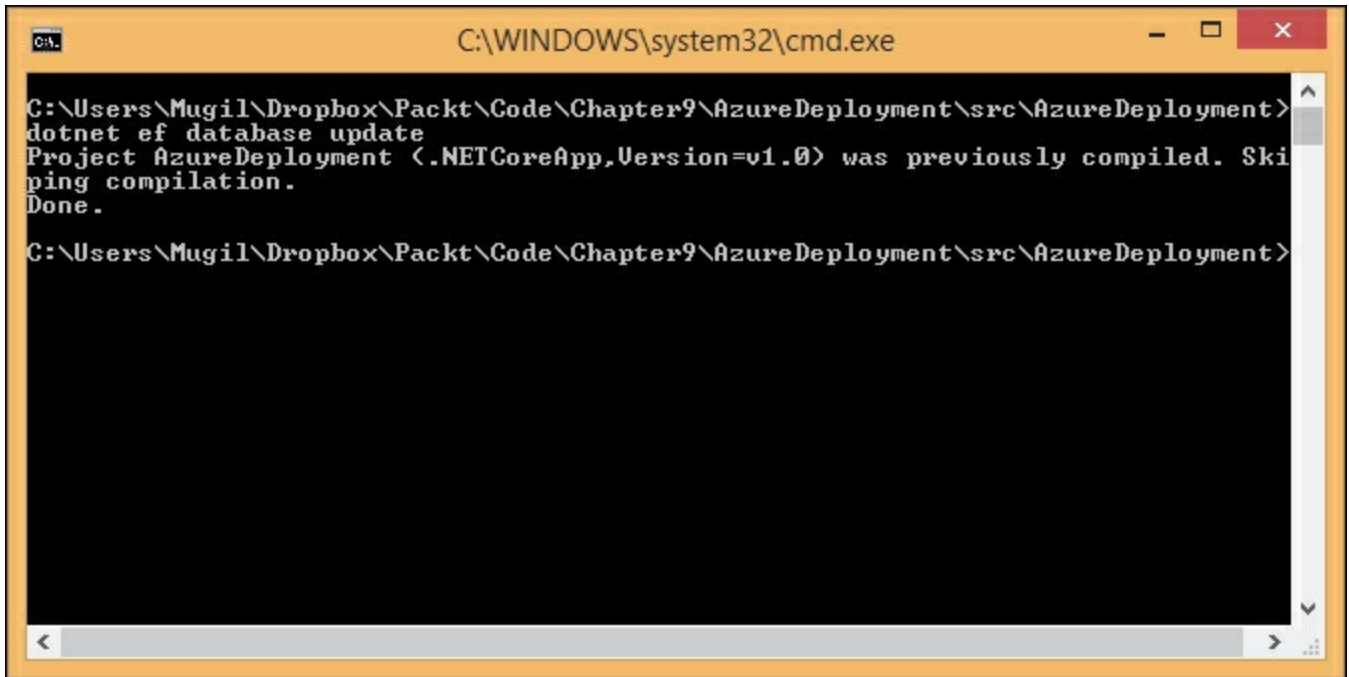
The ASP.NET Core application will be created once you click the **OK** button:
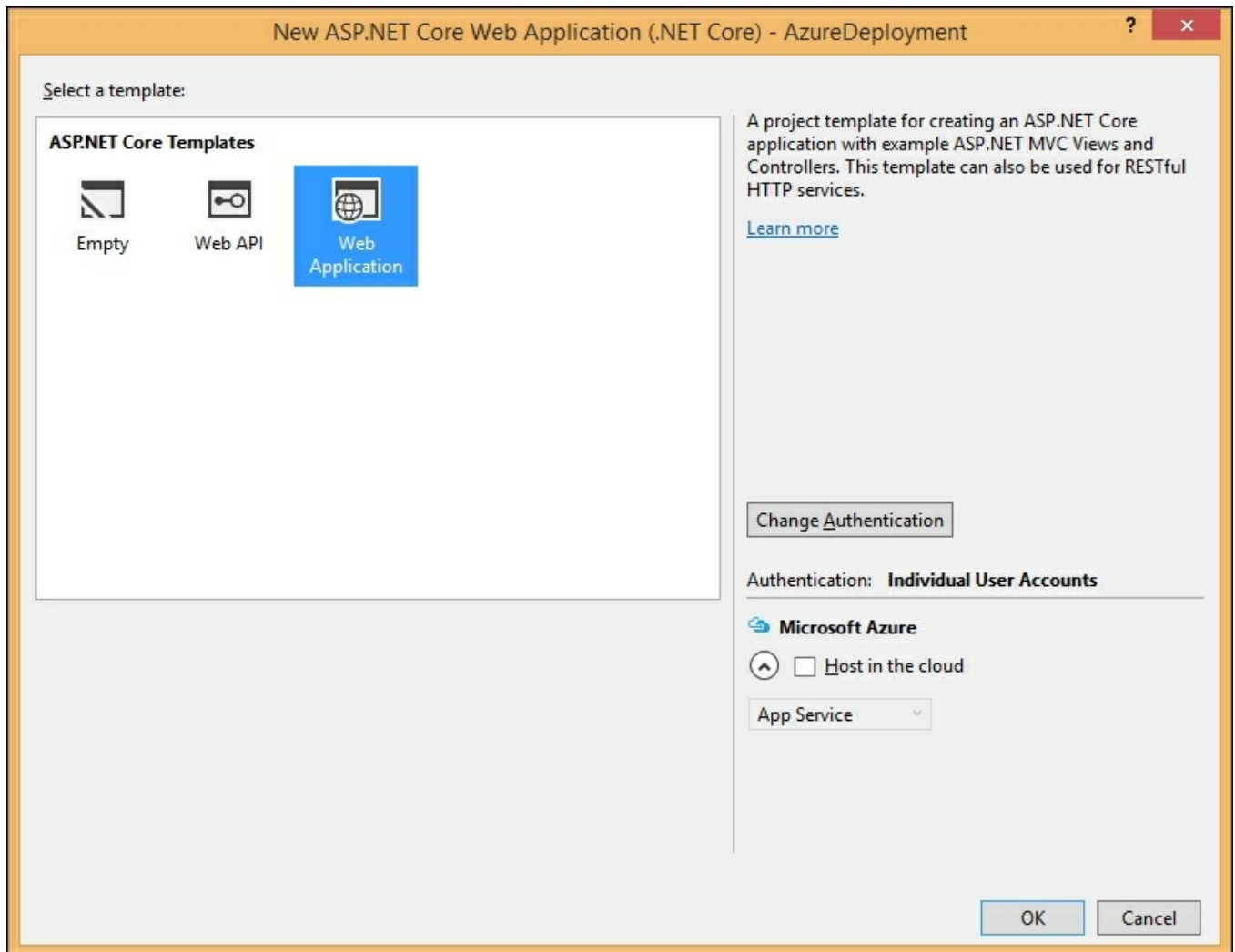
As the default ASP.NET Core Web Application template uses the Entity Framework, we need to execute the following command in order to create the database migration:

```
dotnet ef database update
```

Once you enter the command in **Command Prompt** (in the project's path), the migration file will be created. This migration file will contain all the changes to the database. This migration will be applied at the time of deployment at Azure so that Azure can create the necessary database scripts for Azure deployment:

Once the database migration is completed, right-click on the created Core application and select the **Publish** option, as shown in the following screenshot:
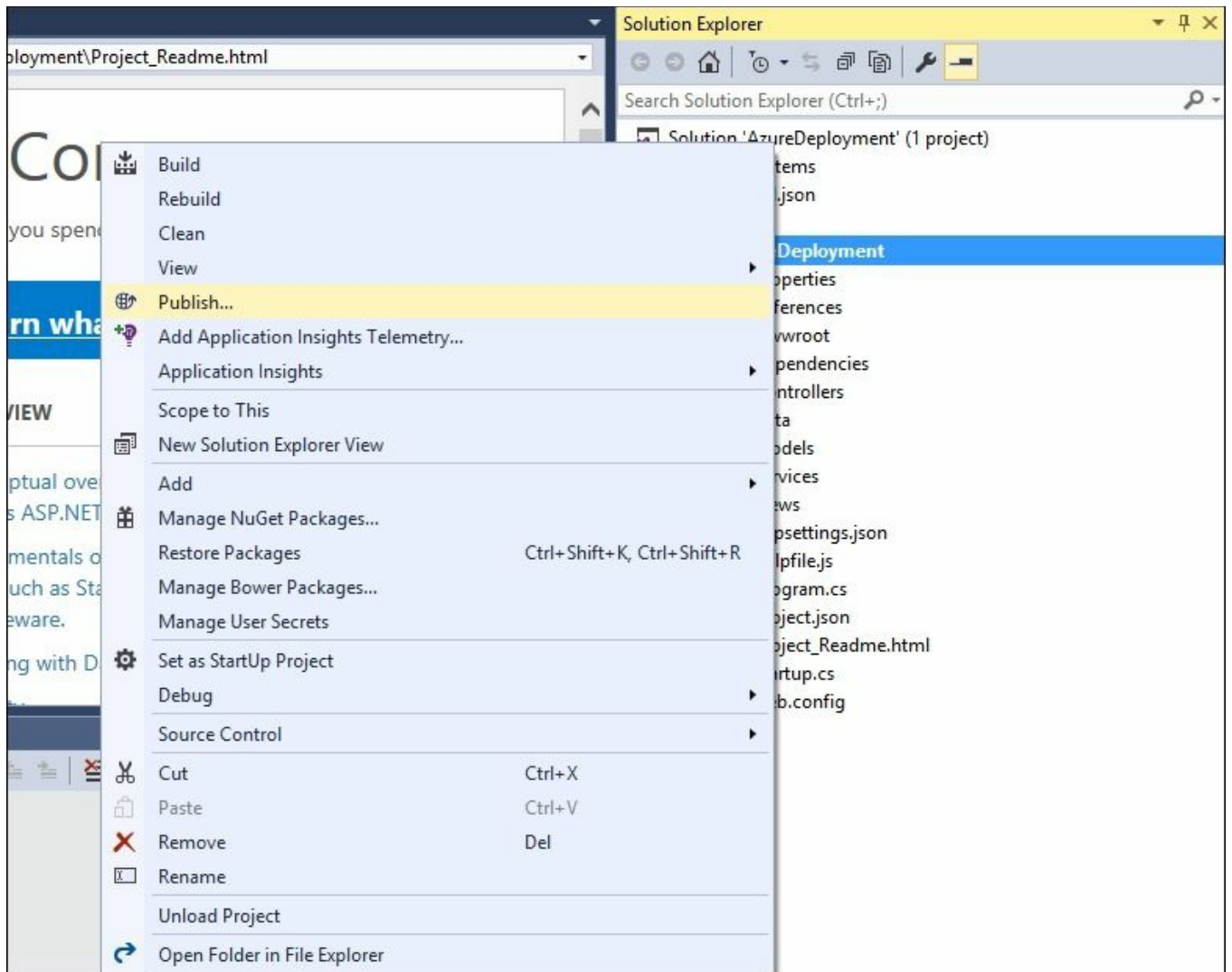
When you click the **Publish** option, you'll be shown the following screen, depicting various publishing options available for you:
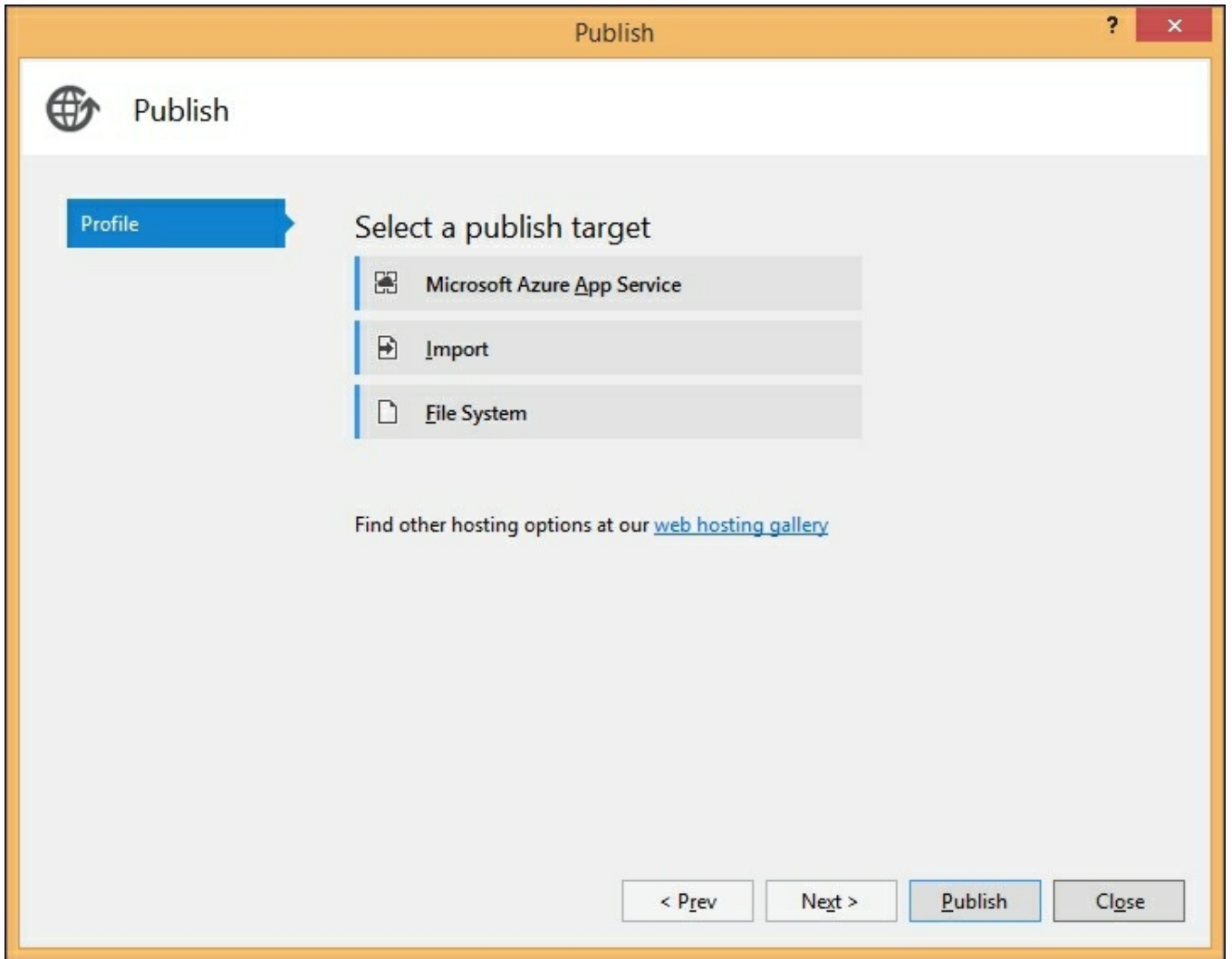
Please select the **Microsoft Azure App Service** option to publish the web application on the Microsoft Azure platform:
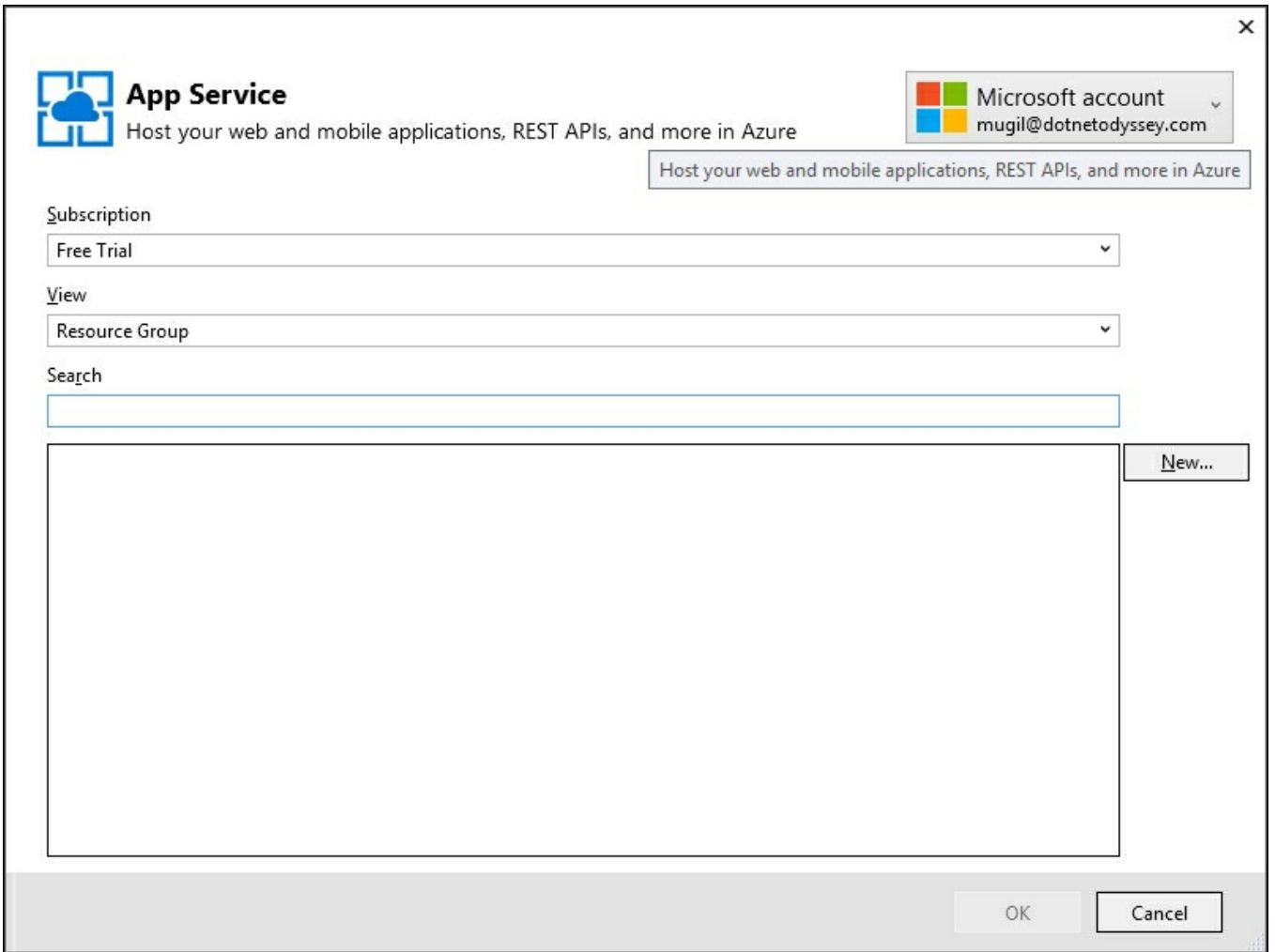
Click on the **New** button, and you will get the following screen:
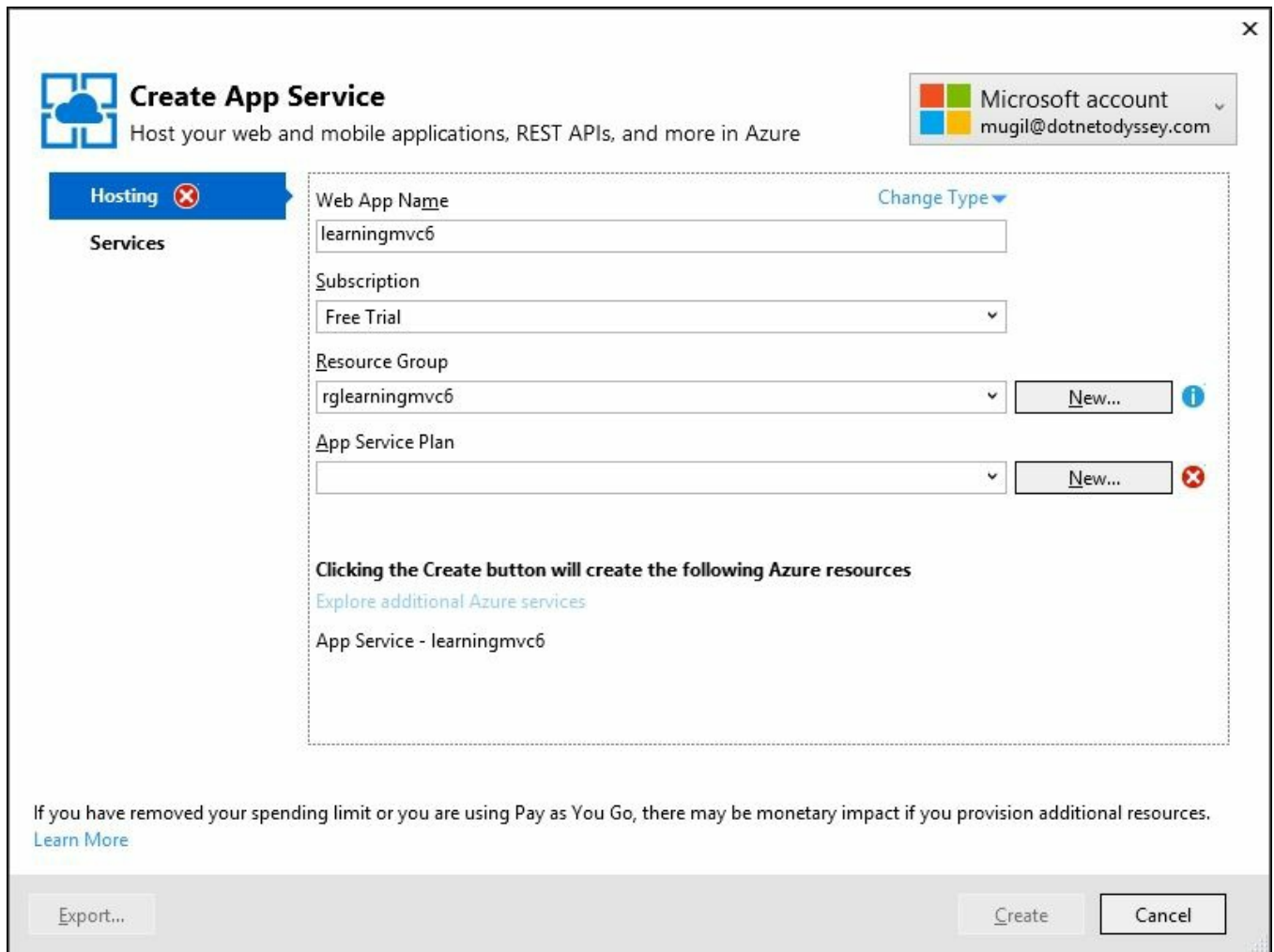
You can change the web app name to whatever name you would like to have. I have changed the web app name to **learningmvc6**.

Click on the **New** button beside the **Resource Group** and enter the name for the resource group. The resource group is just a tag where you can group all of your computing resources so that if you want to delete all the resources, you can just delete the resource group. For example, a resource group could comprise of a web server and a database server—you can think of it like a collection of resources.

Now, click on the **New** button beside the **App Service Plan**. You will get the following window where you can choose the location and size of your web application container. Your location could be anywhere from South Central US to Europe, from Japan to Canada. Your application container could be anything from free to a machine with 7 GB RAM. I have chosen the free option as our objective is to deploy the ASP.NET Core application in a cloud environment rather than to deploy an application which is to be accessed by millions of users. Of course, you can achieve the same with ASP.NET Core and Microsoft Azure:

Configure App Service Plan

An App Service plan is the container for your app. The App Service plan settings will determine the location, features,...

App Service Plan

learningmvc6Plan

Location

South Central US

Size

Free

OK   Cancel

Now, we can configure the SQL database which is available as an additional Azure service.

Click on the + button which is available at the top section which will lead us to the configuration of the SQL database.

If you have any existing SQL server in the Azure environment, you can use it. As I don't have any such server, I am going to create a SQL server by clicking the **New** button beside SQL Server:
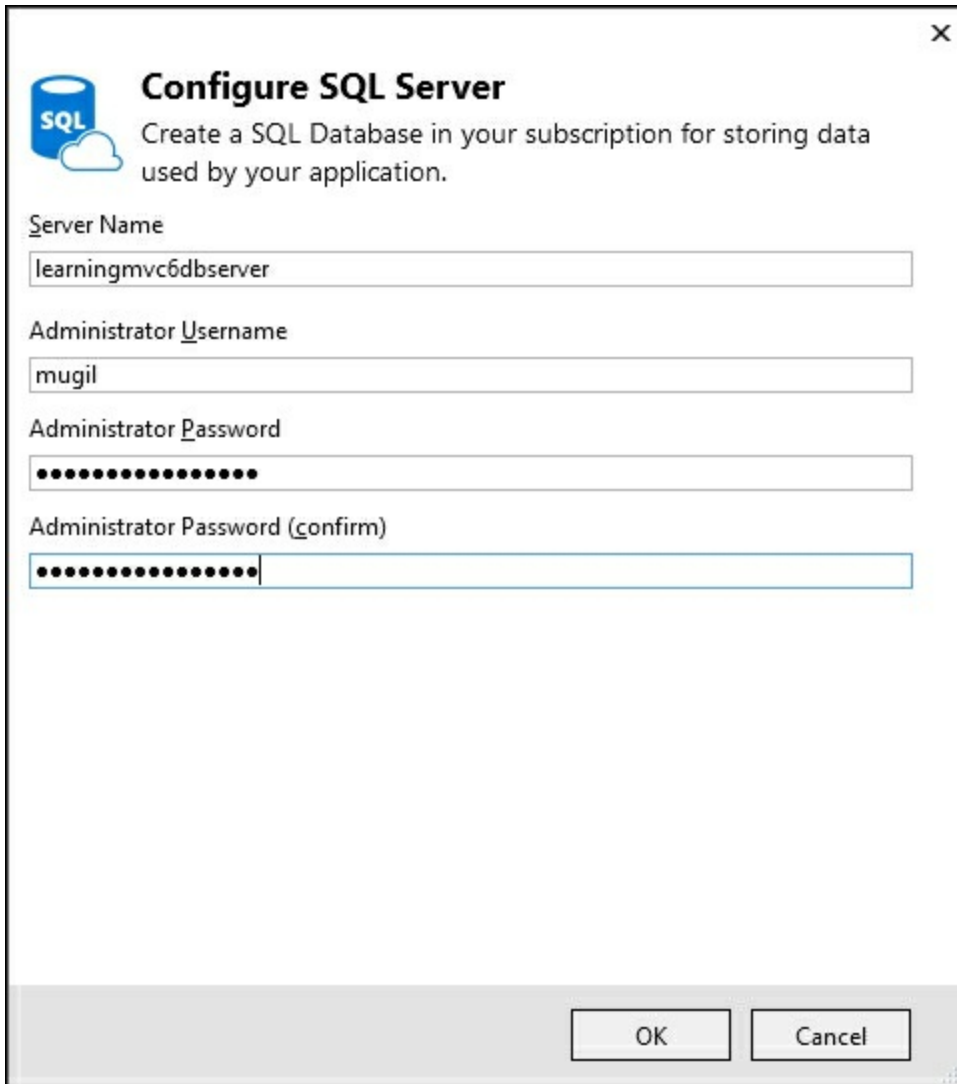
Please enter the administrator user name and password for the SQL Server and click **OK**. You will see the following screen:

Once you click **OK**, you will get the following screen:

Click **OK** on the preceding screen and we will see the **Create App Service** screen:

Click **Create** once we have configured all the required Azure services:
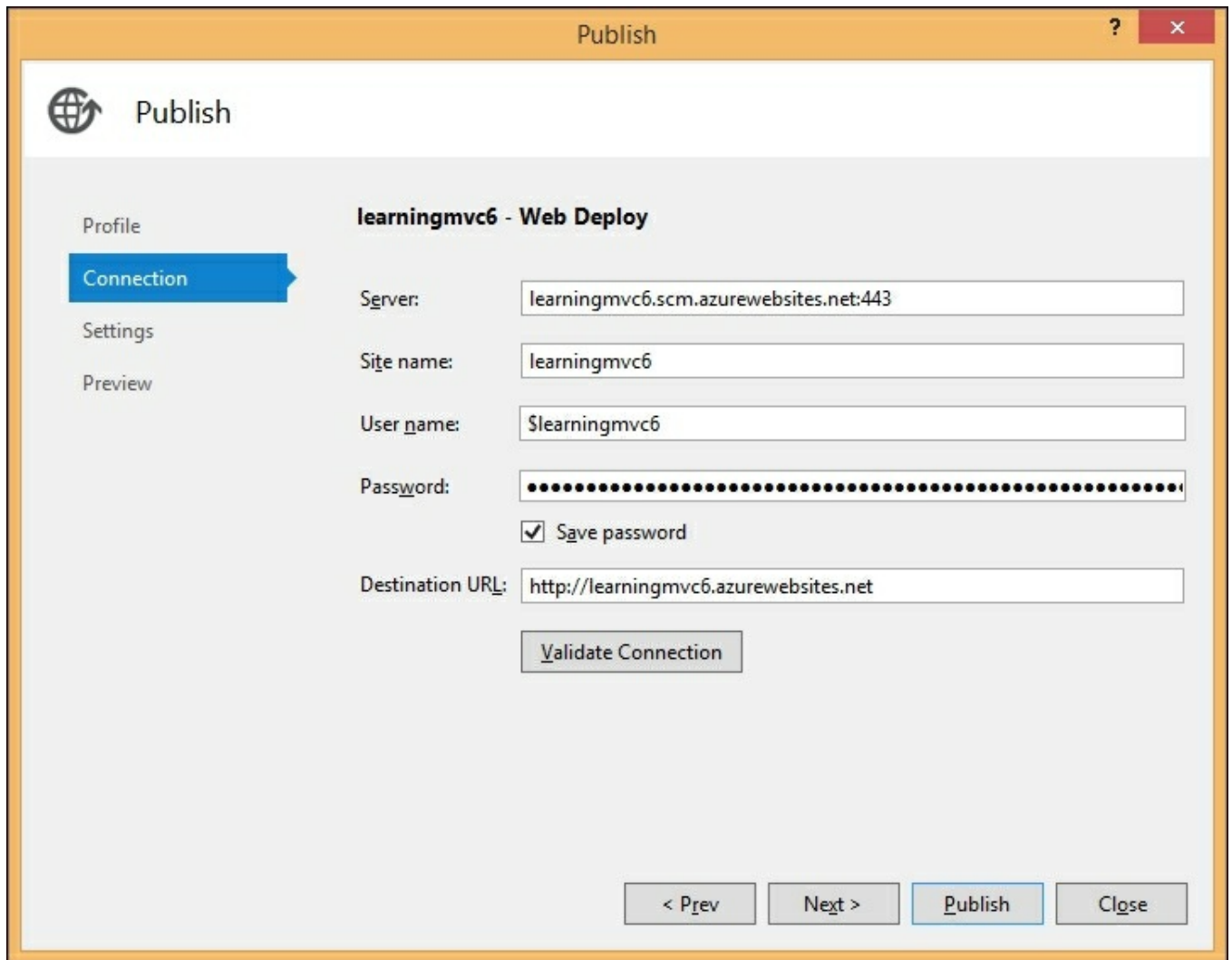
The preceding screen shows the deployment configuration options, such as the **Site name** and **Destination URL** for our application. Click **Next** on the preceding screen:
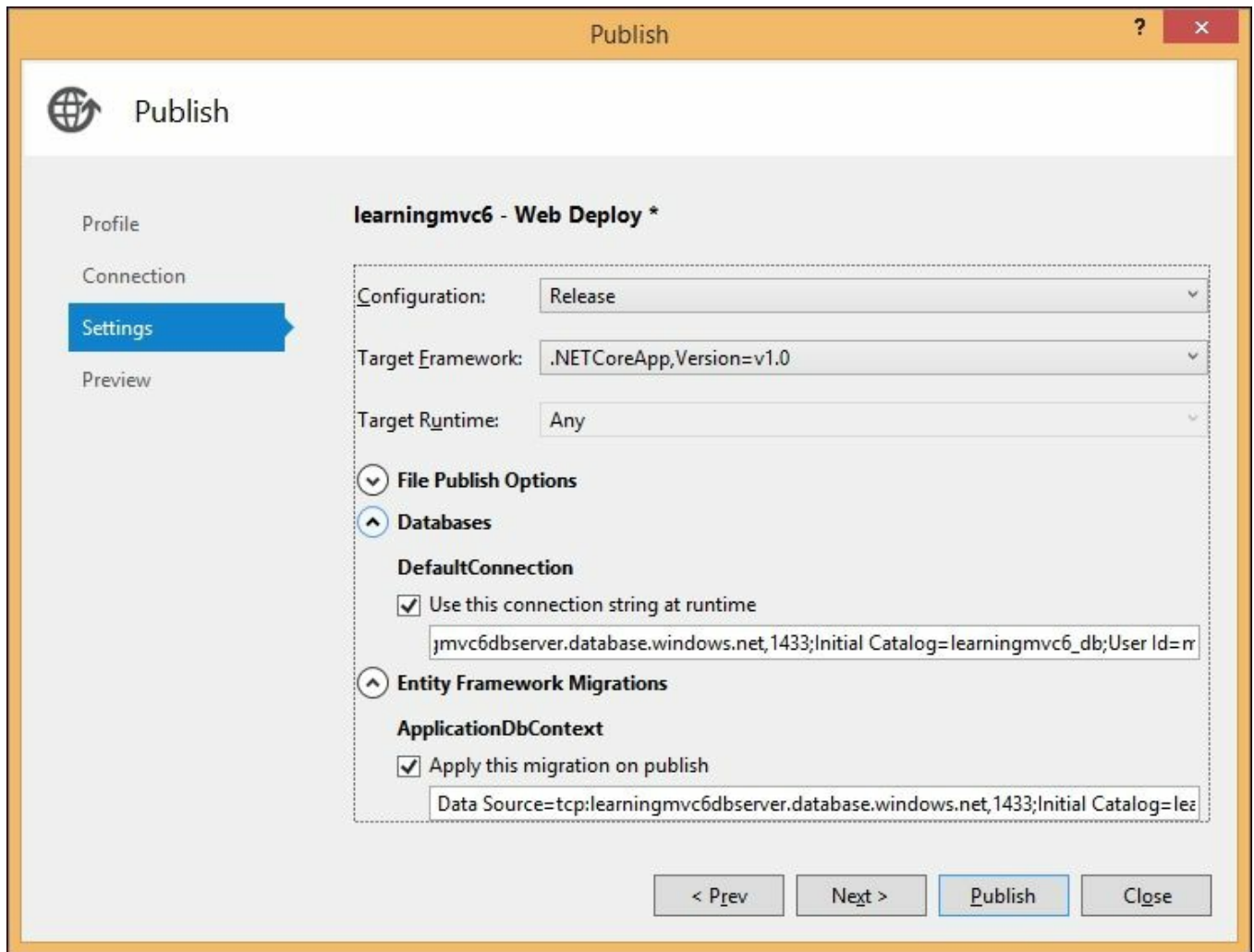
It is important to note that you need to expand the **Databases** option and **Entity Framework Migrations** options and select both the checkboxes. The first checkbox represents the connection string that should be used at runtime and the second checkbox represents the database migration that should be applied on publication.
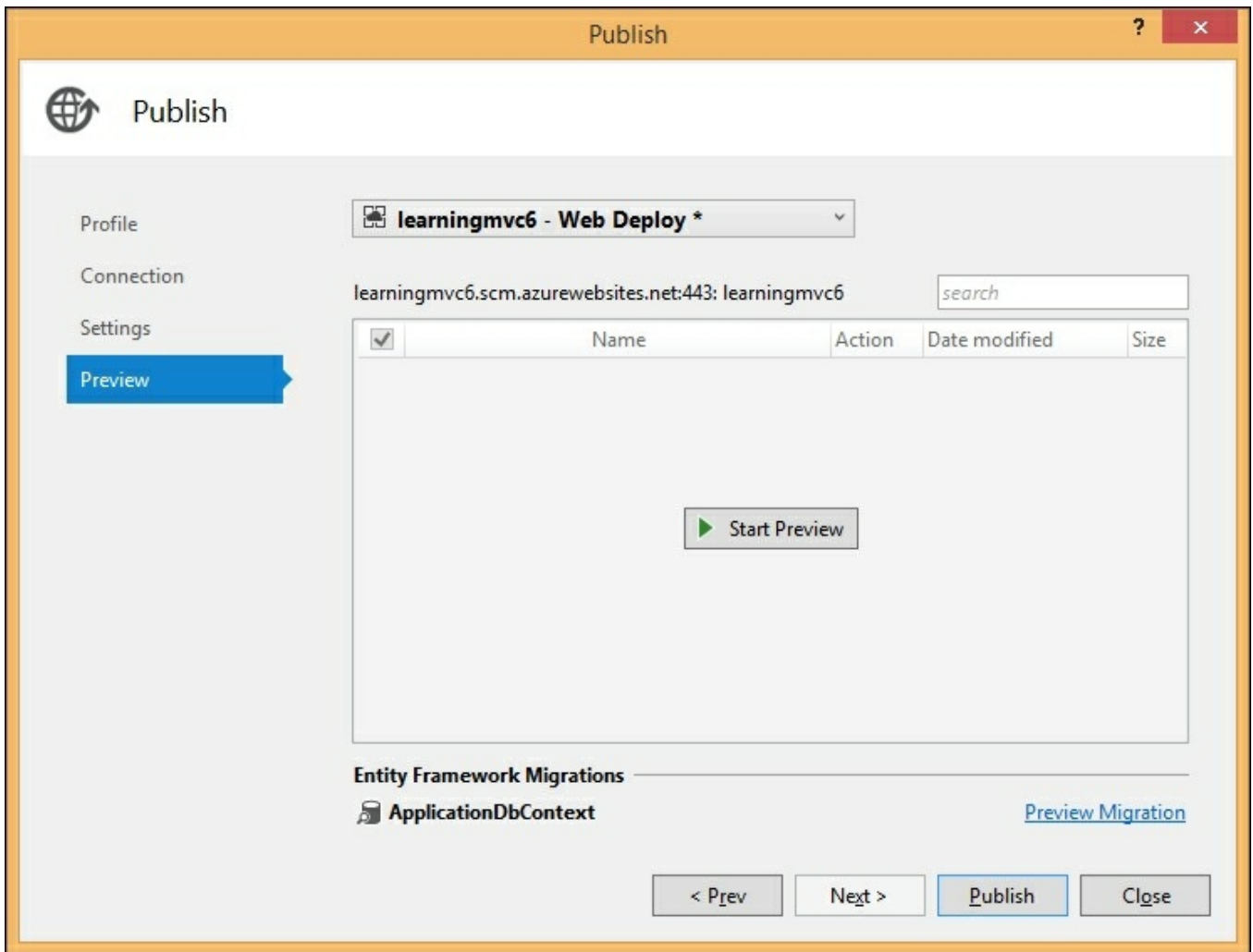
The preceding screen is the preview screen where you can see the files that would be deployed when you publish. This is an optional step—if you want to see the files, you can click the **Start Preview** button. Or else, you can click the **Publish** button to publish the web application in the Azure platform.

Once you click the **Publish** button, our ASP.NET Core application will be deployed in Azure and your application URL will be opened on successful publication. You will get the following screen:

Deploying the ASP.NET Core web application in the Linux environment

In this section of the chapter, we are going to learn how to create and deploy the ASP.NET Core web application in the Linux platform. I am going to deploy the application in the cloud using **Amazon Web Services** (**AWS**). Obviously, you don't need AWS to deploy the ASP.NET Core application on Linux. I am just using it so that I do not need to install Linux on my local machine. And another advantage of hosting with AWS (or any other public cloud service provider or any hosting provider) is that I can access the web application from anywhere as it will be publicly available.

We have the following prerequisites to create and deploy in the Linux environment:

- Linux machine
- Putty client (if you are using a remote Linux machine)

Creating a Linux machine

We are going to use AWS to create a Linux machine. The advantage of using AWS or any other cloud provider is that we can use their service only when we need it and we can wind down the machine when you are done with it. You only need to pay for the time when you are using it. For the first year, AWS has a free tier where you can host the machine (if it is eligible for the free tier) without having to pay anything. I have been using AWS for more than couple of years to try out many things in the cloud, hence I am not eligible for the free tier.

However, you can install Linux on your Windows PC by using any virtualization software. Ubuntu Linux has the option of booting up from the USB drive itself so that you don't need to disturb anything in your local PC.

Once you sign-up for an AWS account, you can go to the **EC2 dashboard**, where you can create **EC2 instances**:
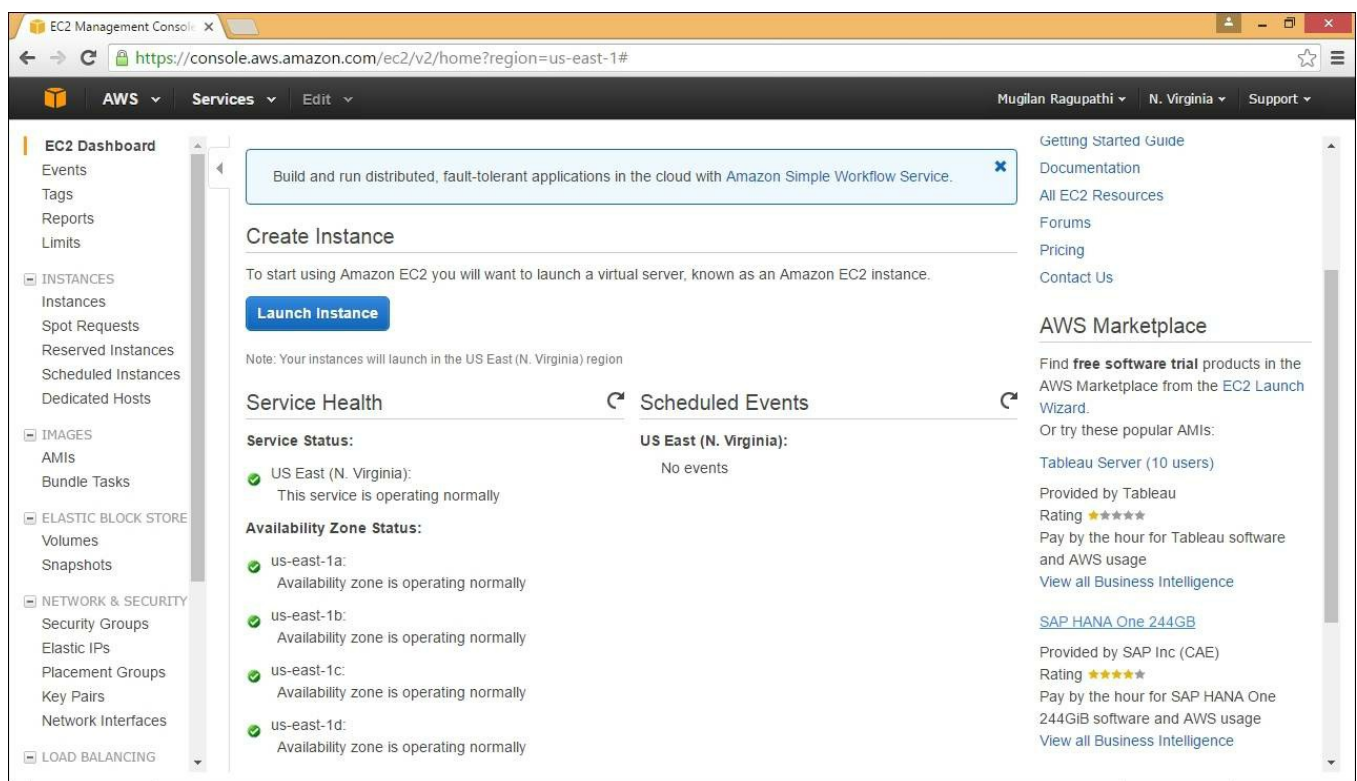
Click **Launch Instance** in the preceding screen. A wizard will be launched where it will help you in selecting and configuring the instances. In this step, we are selecting the Ubuntu Linux server because of its ease of use.
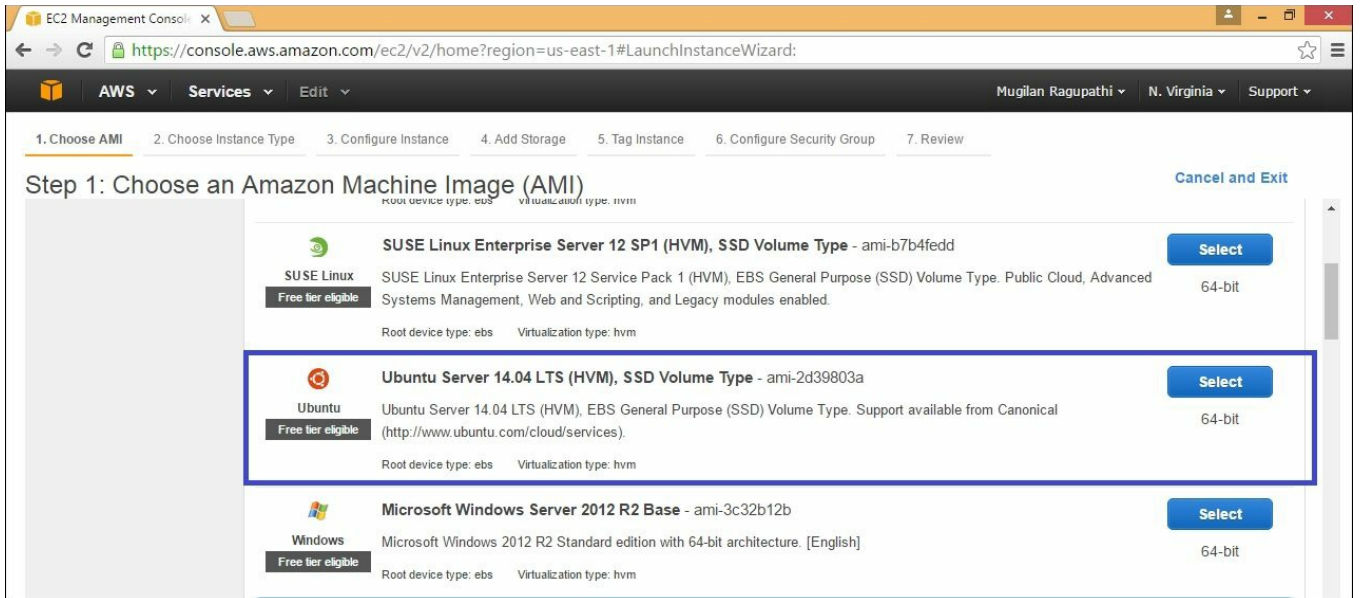
There are different kinds of instances available in AWS ranging from **nano** (with 0.5 GB RAM) to large machines (with 1952 GB RAM). We are going to choose **micro** instance as it is eligible for the free tier:
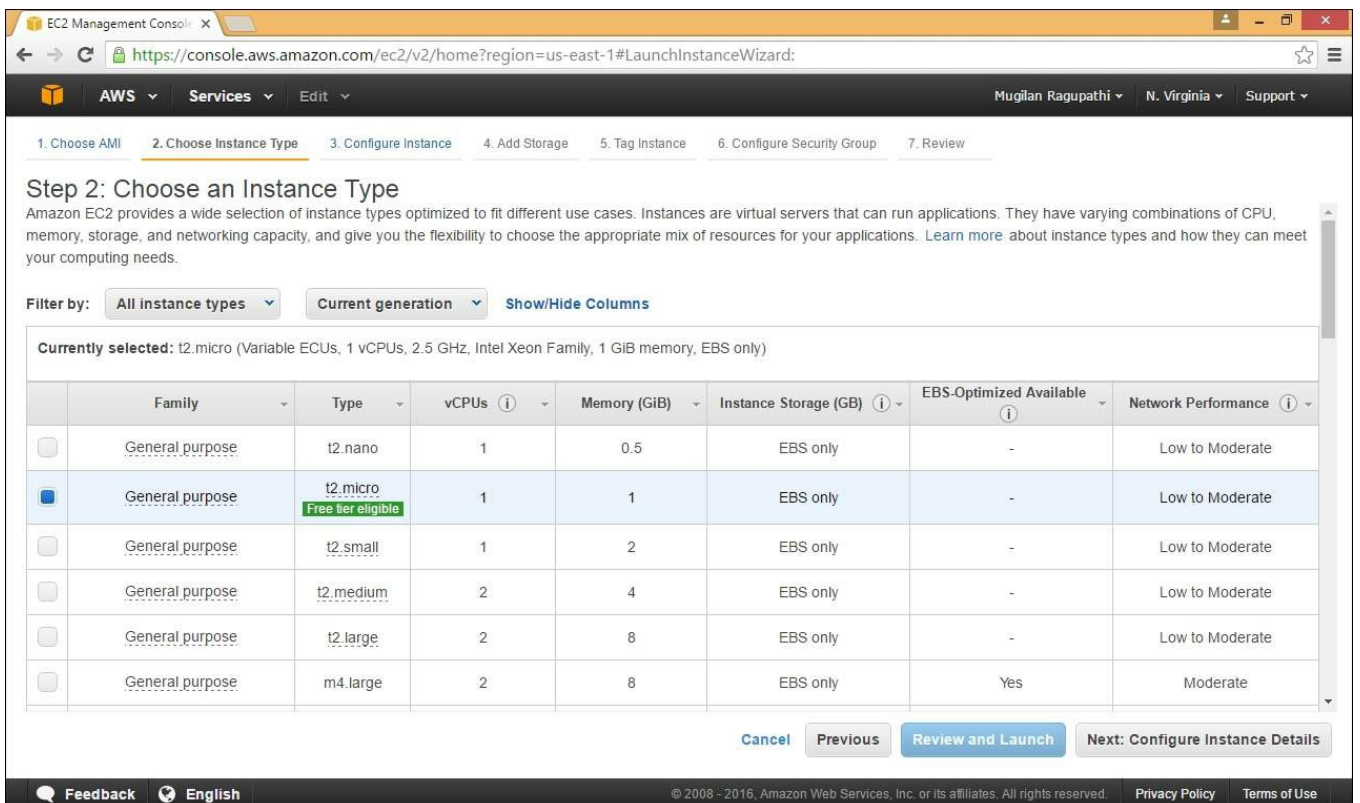
In previous step, we can configure the instances for the cloud. We can create an auto

scale group where the AWS cloud will spin up the instances automatically when the load is high. As our objective is to create and deploy the ASP.NET Core web application, we are going to leave the default values as they are and click **Next: Add Storage** to move to the next screen:
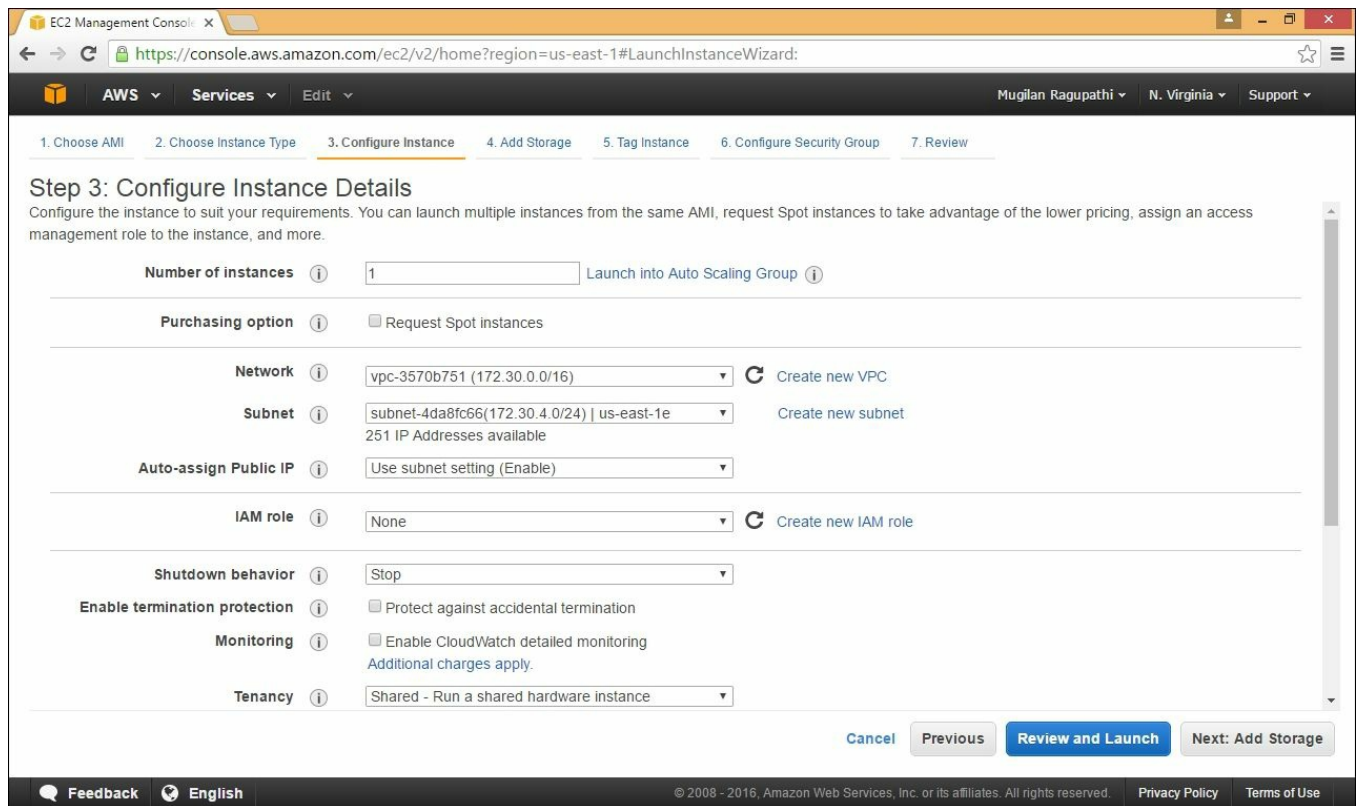
The **micro** instances do not come with any external storage. Hence, we need to add the storage in order to use it. We have three options for storage to choose from: **General Purpose SSD**, **Provisioned SSD**, and **Magnetic SSD**. Out of the three, the **General Purpose SSD** is the storage that would be usually used.

When your application is making high input-output operations, the throughput may come down. But in the Provisioned SSD you can maintain the required throughput from the storage device. Magnetic storage is just an old type of storage. We are going to use the General Purpose 8 GB **Solid State Drive (SSD)** as it serves our purpose well.
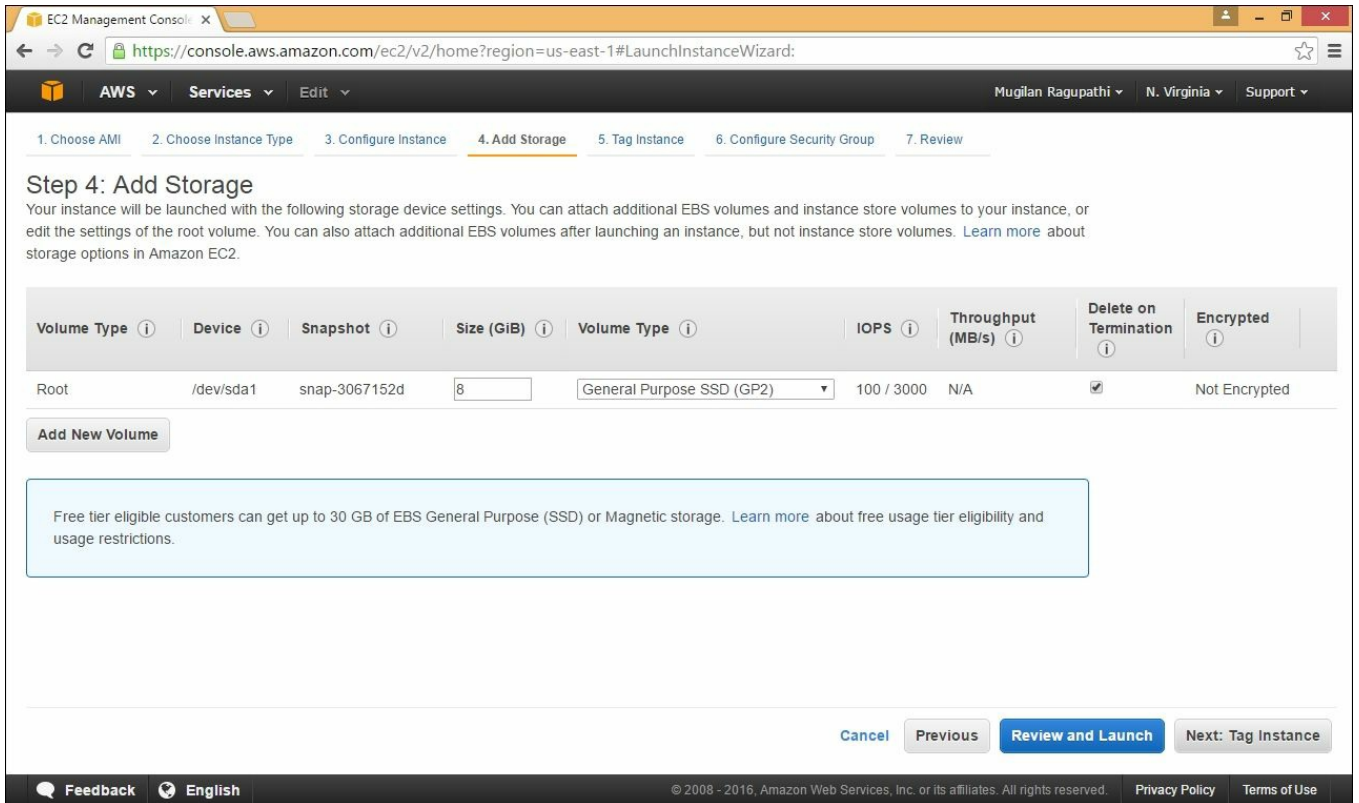
If you are using multiple instances, you can tag them so that you can control the instances by using the tag name. As we are going to launch only one instance, I am just going to leave it blank and move on to the next step:
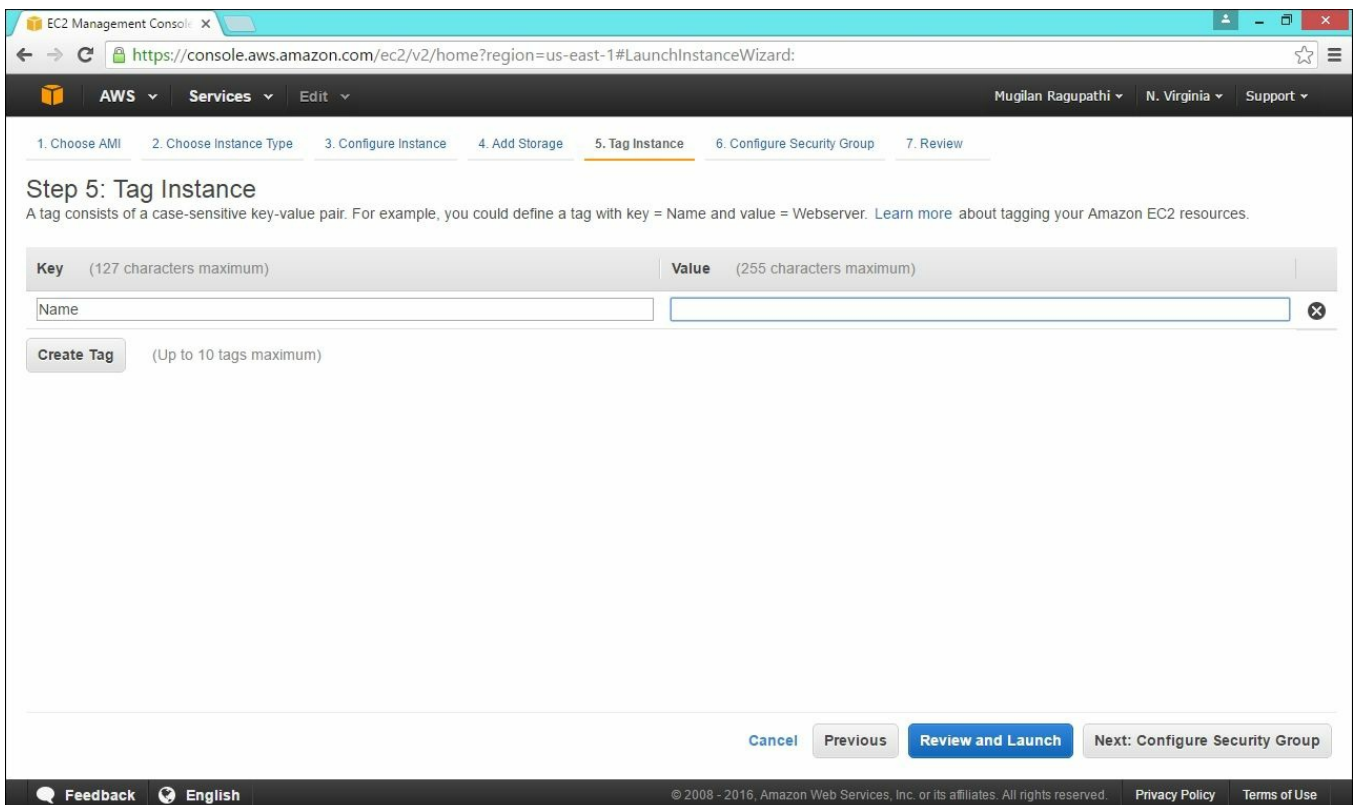
In this step, we can configure the security group for the instance—which ports should be opened for the incoming traffic. The general rule in any configuration is to only open up the ports what you need and nothing else. You also need to tell the IP (or its range) from where the machine could be accessed from. As it is a demo application, we are going to open ports `22`, for **Secure Shell** (**SSH**); for using PuTTY, and `80`, for accessing the Core web application.

Once you have configured the Security Groups, click **Review and Launch**.
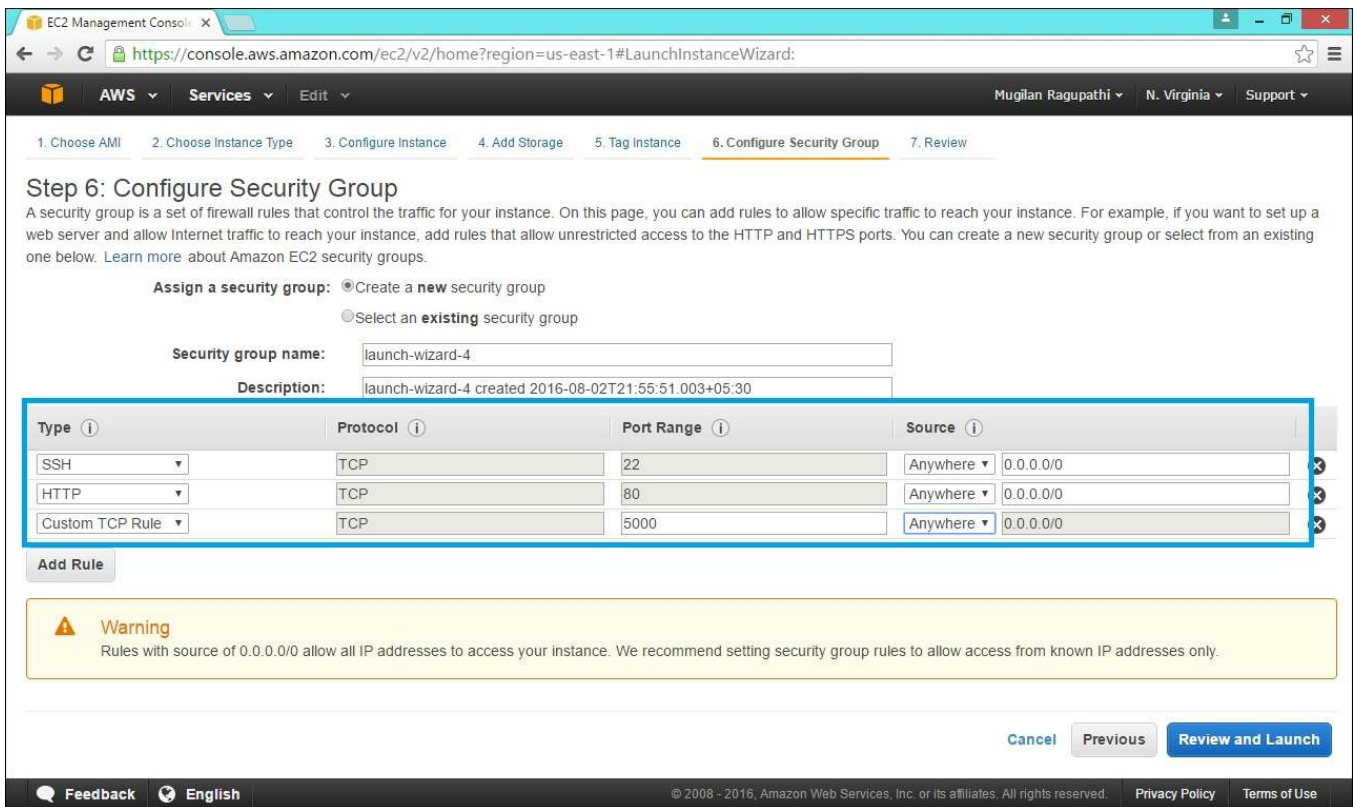
On the following screen, you can review the chosen options:
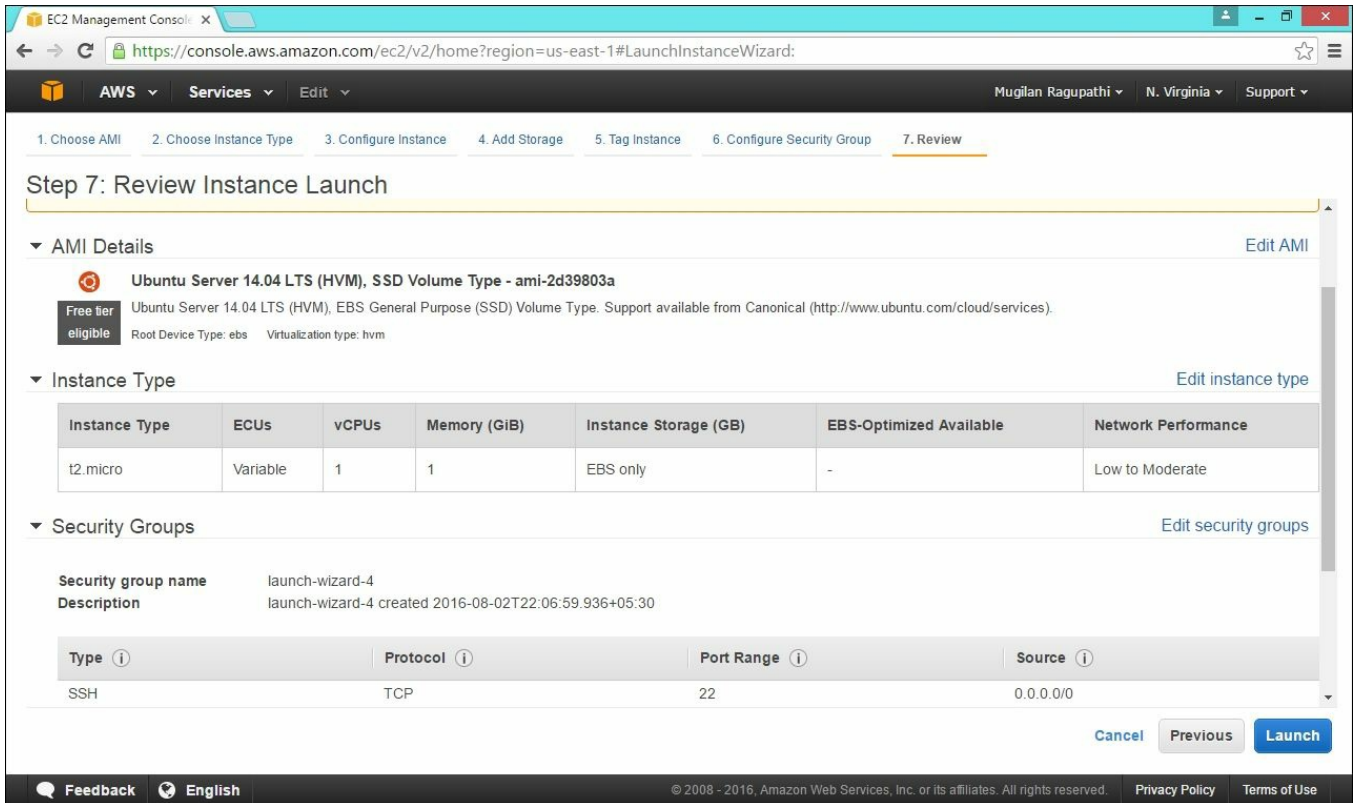
You can click **Launch** once you are fine with the selected options. Otherwise, you can go back to the previous step to reconfigure them with the correct values.

When you click **Launch**, it will ask you to choose a key pair which you will be using to log into any AWS server. If you do not have one, you can create one. As I have created one already, I am going to use the existing one, as shown in the following screenshot:
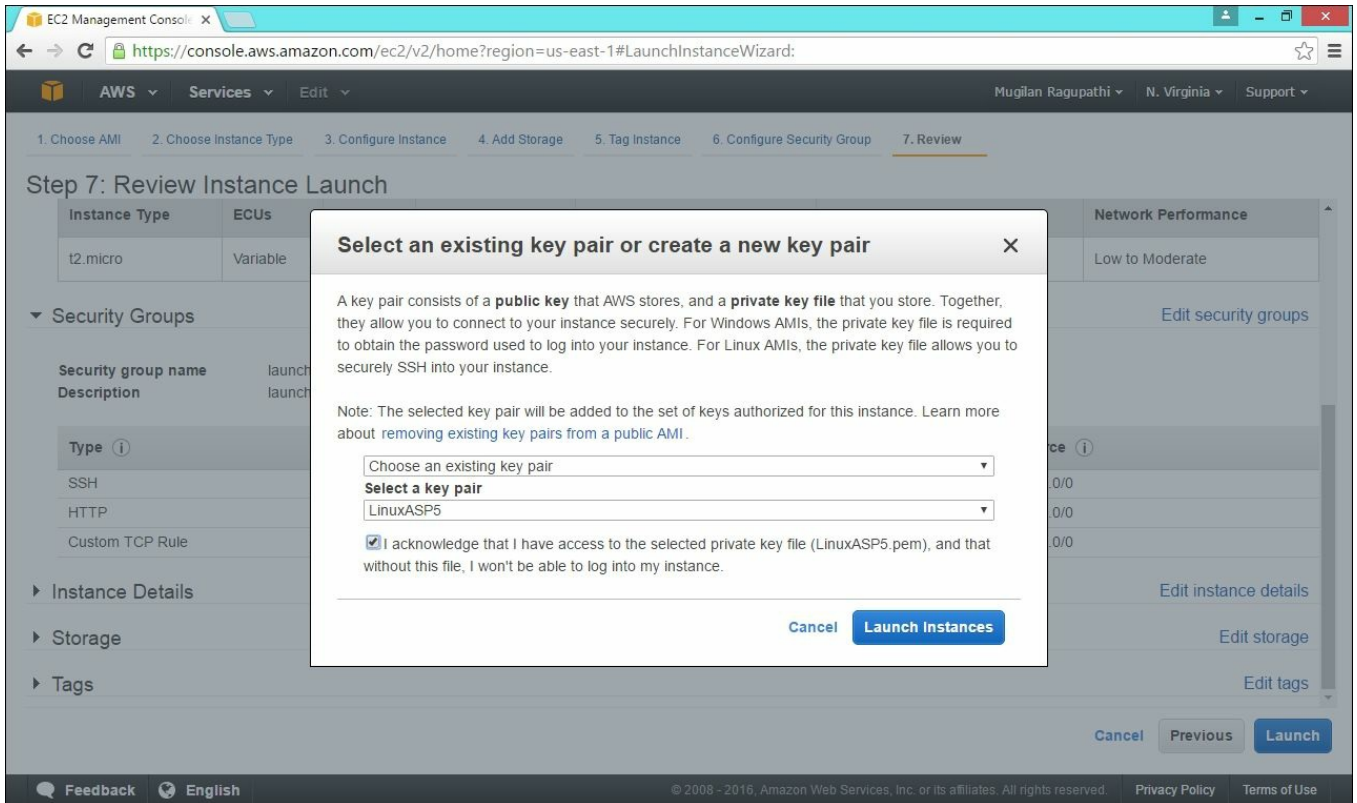
Select the key pair and click **Launch Instances**. AWS will spin up new instances for us and the status will be shown (as in the following screenshot). The instance ID will also be available (boxed in the screenshot):
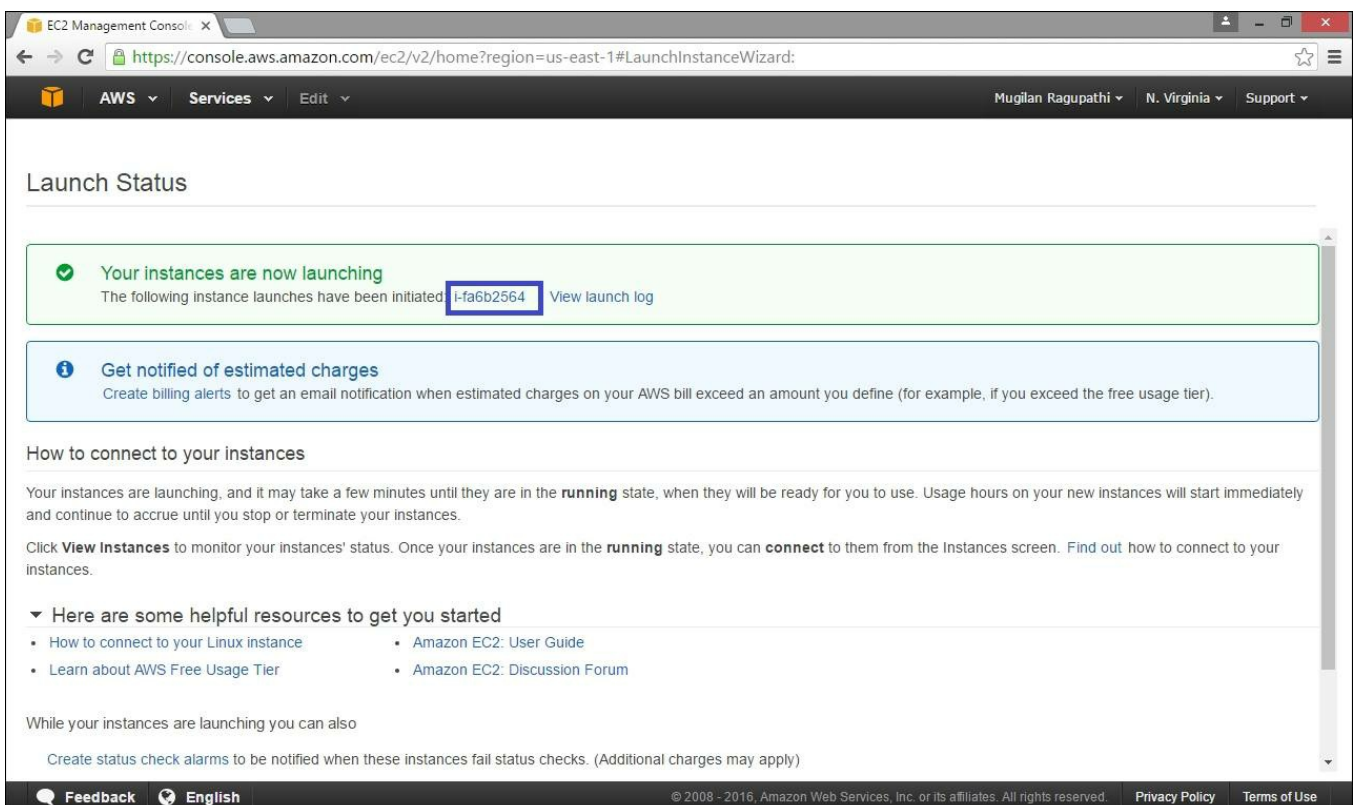
Clicking on the blue colored link will fetch you the status (as shown in the following screenshot). The **Public DNS** and **Public IP** are important values which you will be using to connect to that server. Hence, I've boxed them in the screenshot:
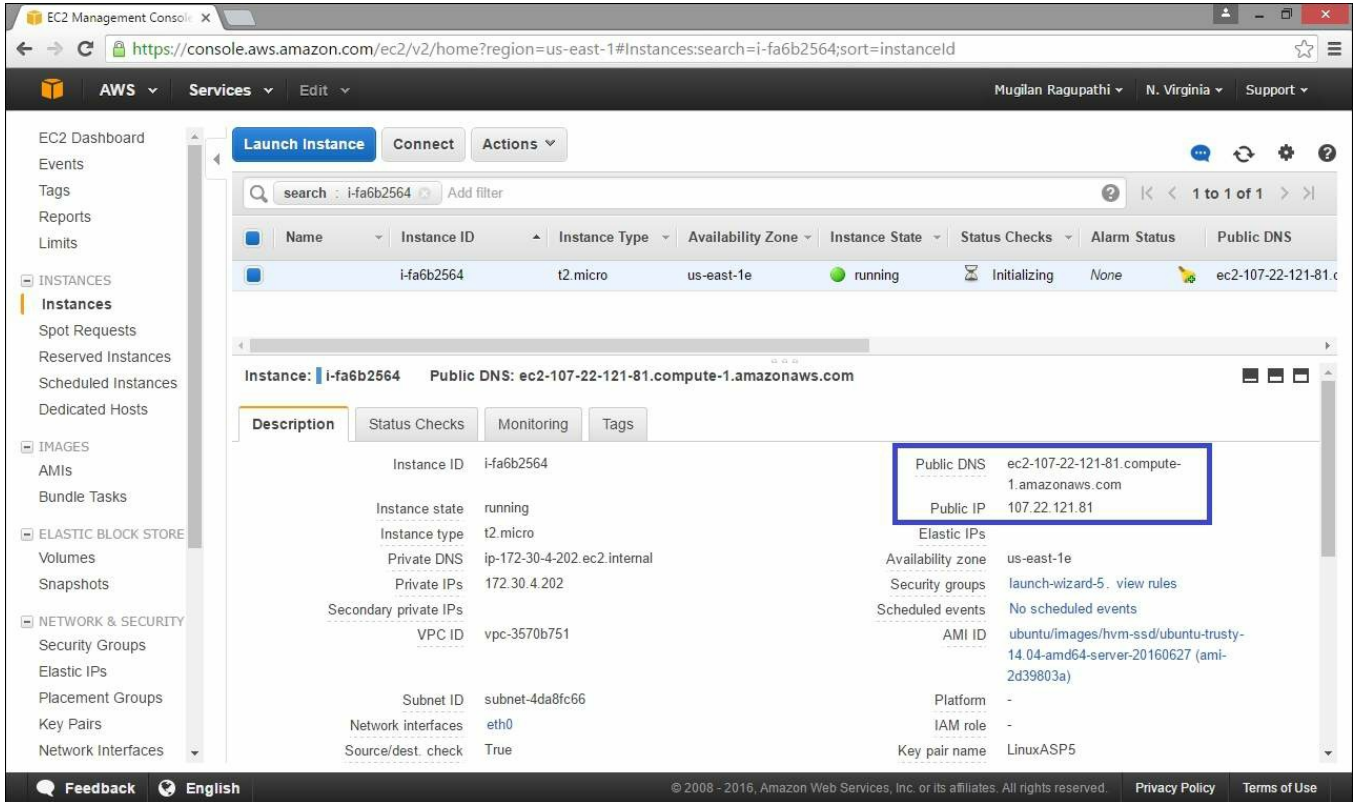
Installing the PuTTY client

Having created a new Linux server where we can create an ASP.NET 5 web application and host it, we need to install the PuTTY client, a small application that can send commands to the Linux server and receive the responses. As we are going to install the application in a Linux server, we need to have a way for connecting from your Windows PC to the Linux server. The PuTTY client application does exactly that.

You can download the PuTTY client by visiting http://www.chiark.greenend.org.uk/~sgtatham/putty/ .
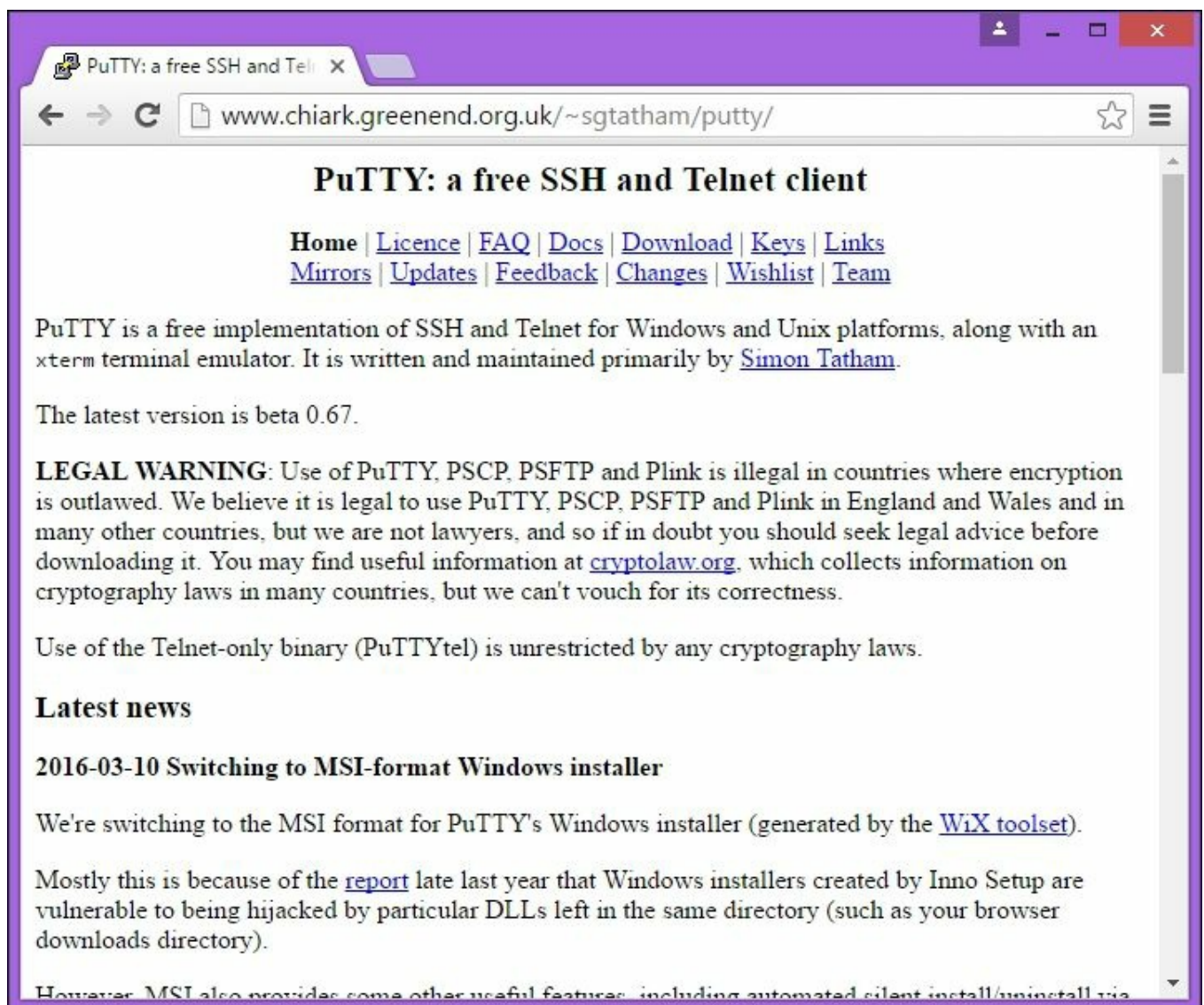
Click on the **Download** link and select the link (boxed in the screenshot) in the following screen:

It will download the MSI file. Once it's downloaded, launch the installer and you'll be shown the following welcome screen:

Click **Next** and you'll see following screen:
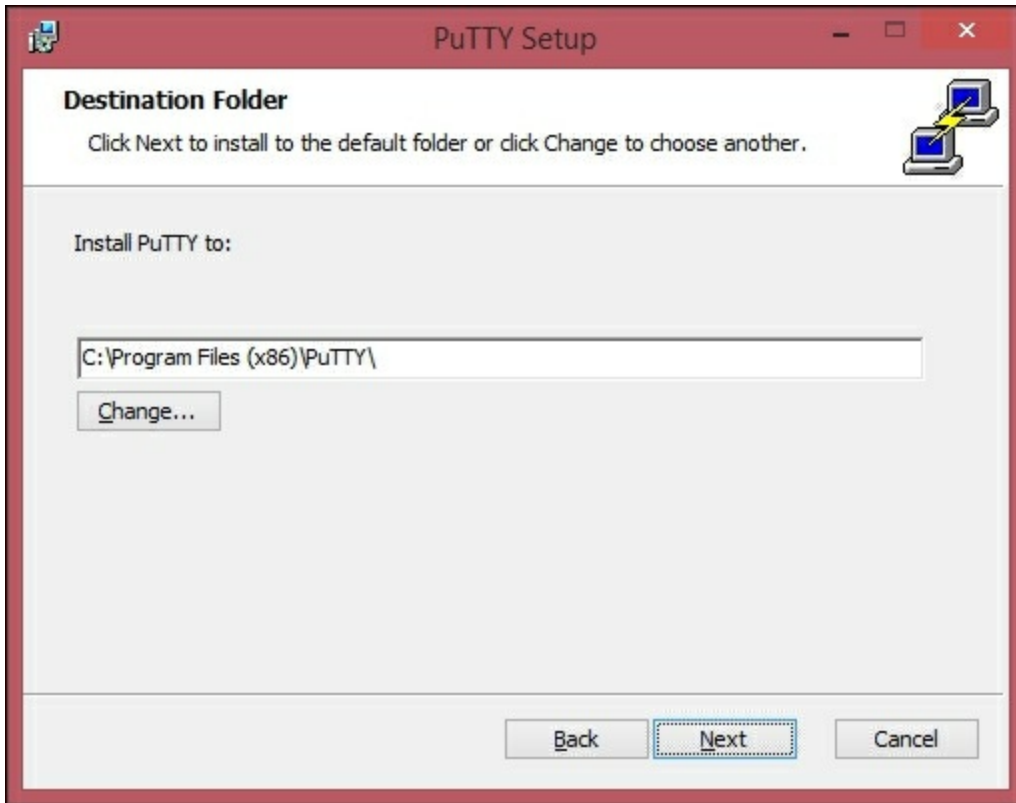
Choose the installation folder—you can leave it as it is and click **Next**:
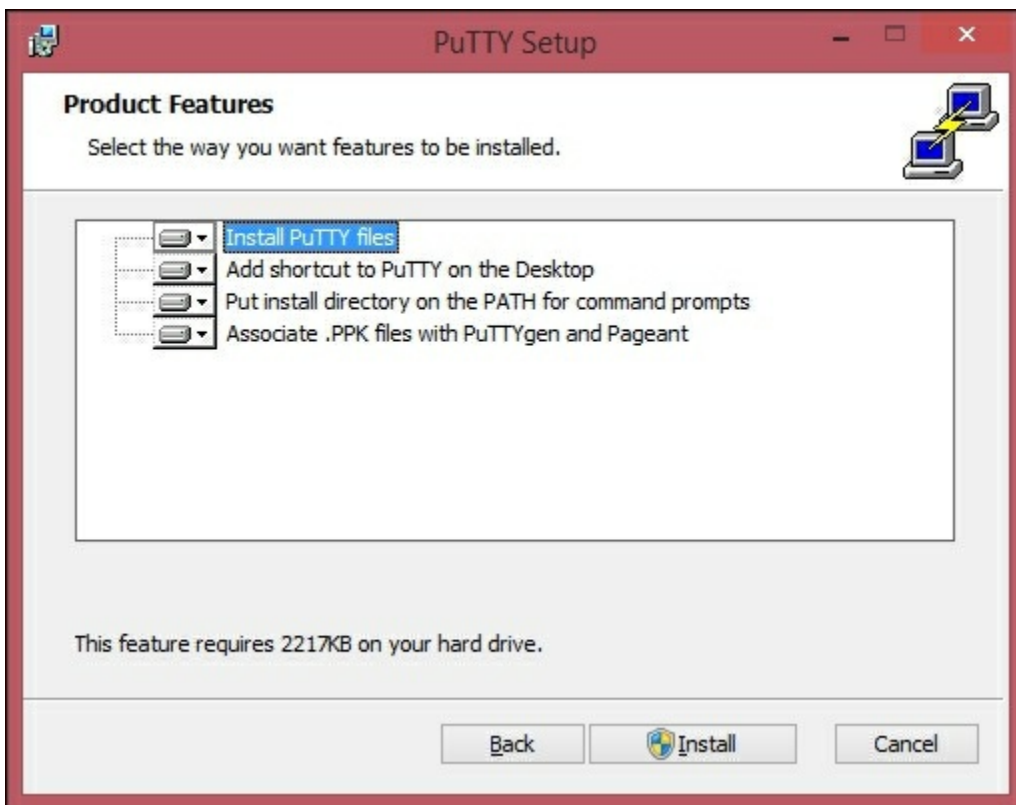
Select the product features which you want to install. You can leave the default selection and click **Install**. Once it is installed, you will be shown the following screen:

Click **Finish** and launch the PuTTY application. It will open the PuTTY configuration window, where we are going to enter the hostname and authentication details. The hostname is `<username>@<public DNS>`. In our case, it is **ubuntu@ec2-107-22-121-81.compute-1.amazonaws.com**. Ubuntu is the default user for the Ubuntu AMI that we have chosen. We can get the public DNS value in the status window as shown earlier:
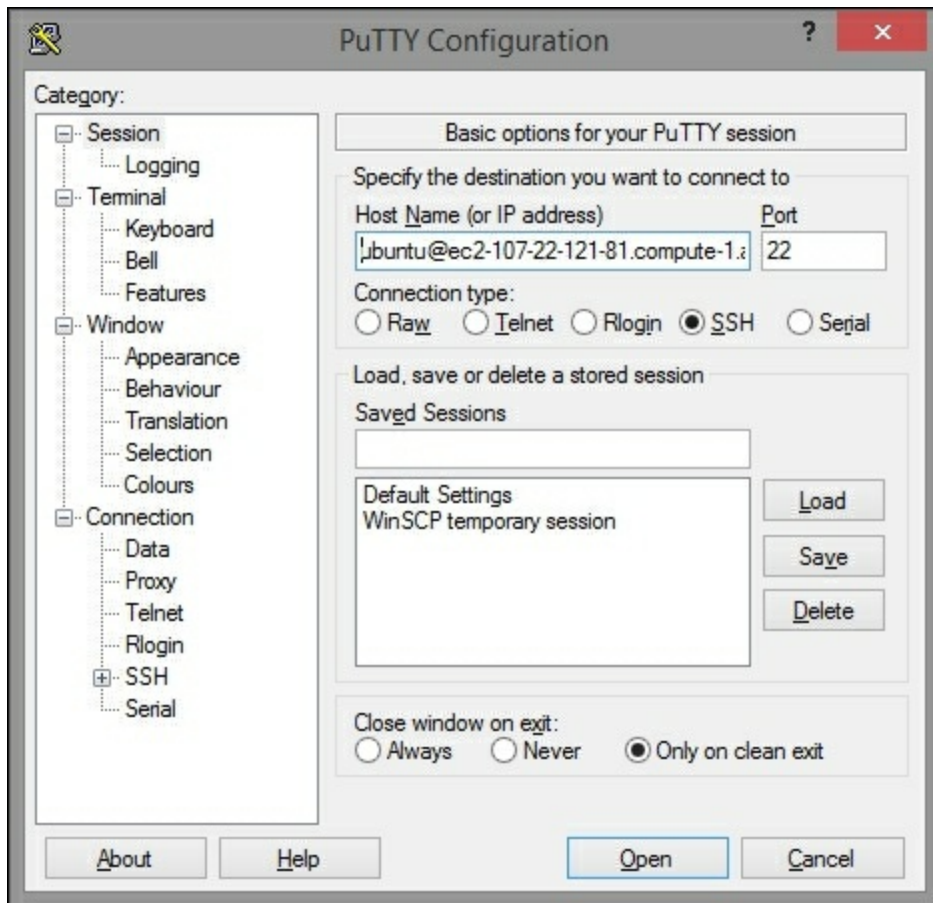
For authentication, select **Connection | SSH | Auth** in the left-hand pane and select the `PPK` file (private key file) that we created earlier:
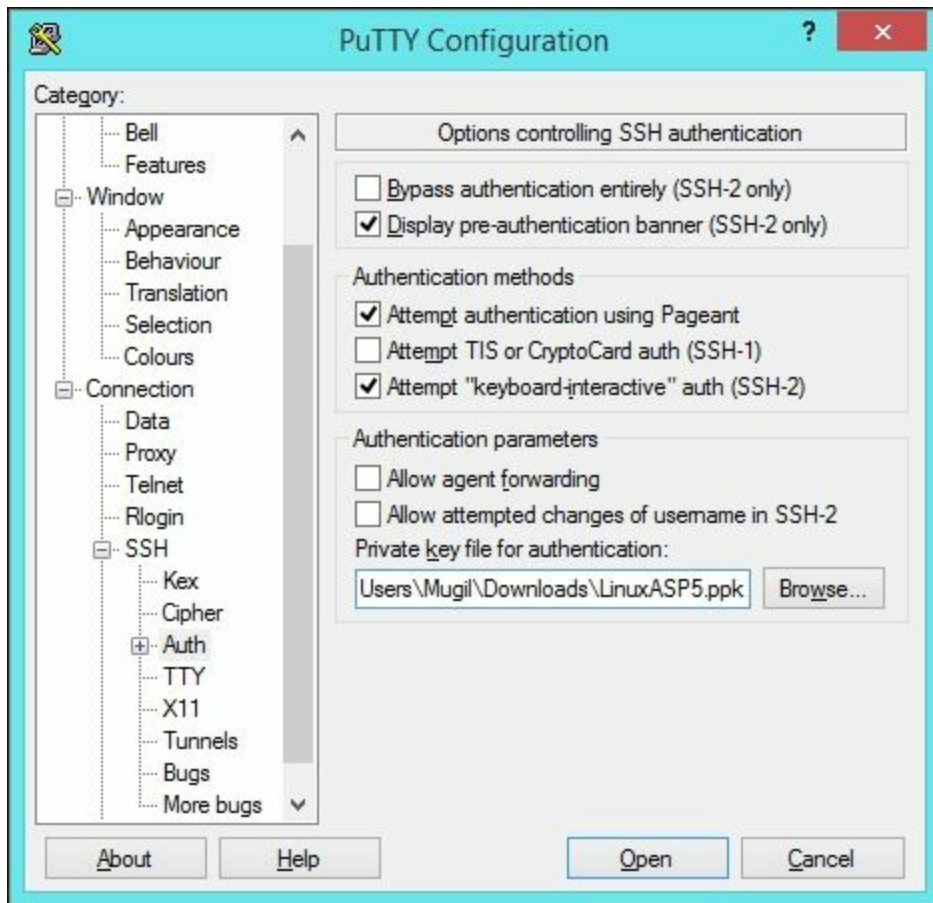
Click **Open**. You'll get a warning asking you whether you trust this host. Click **yes** and you'll be shown the Command Prompt of the Linux screen.

Next, we need to install .NET Core before creating the ASP.NET 5 application and eventually host them.

Installing of .NET Core in a Linux machine

In order to install .NET Core on Ubuntu, we need to first set up the apt and get feed that hosts the package that we need. Enter the following commands:

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-
mo.trafficmanager.net/repos/dotnet-release/ trusty main" >
/etc/apt/sources.list.d/dotnetdev.list'sudo apt-key adv --keyserver
apt-mo.trafficmanager.net --recv-keys 417A0893
```

You will get the following screen:

Then update it by issuing the following command, which will download the required packages and install them:

```
sudo apt-get update
```

You will see the following screen for this command:

Install the .NET Core with the following command:

```
sudo apt-get install dotnet-dev-1.0.0-preview2-003121
```

The following screen will be displayed:
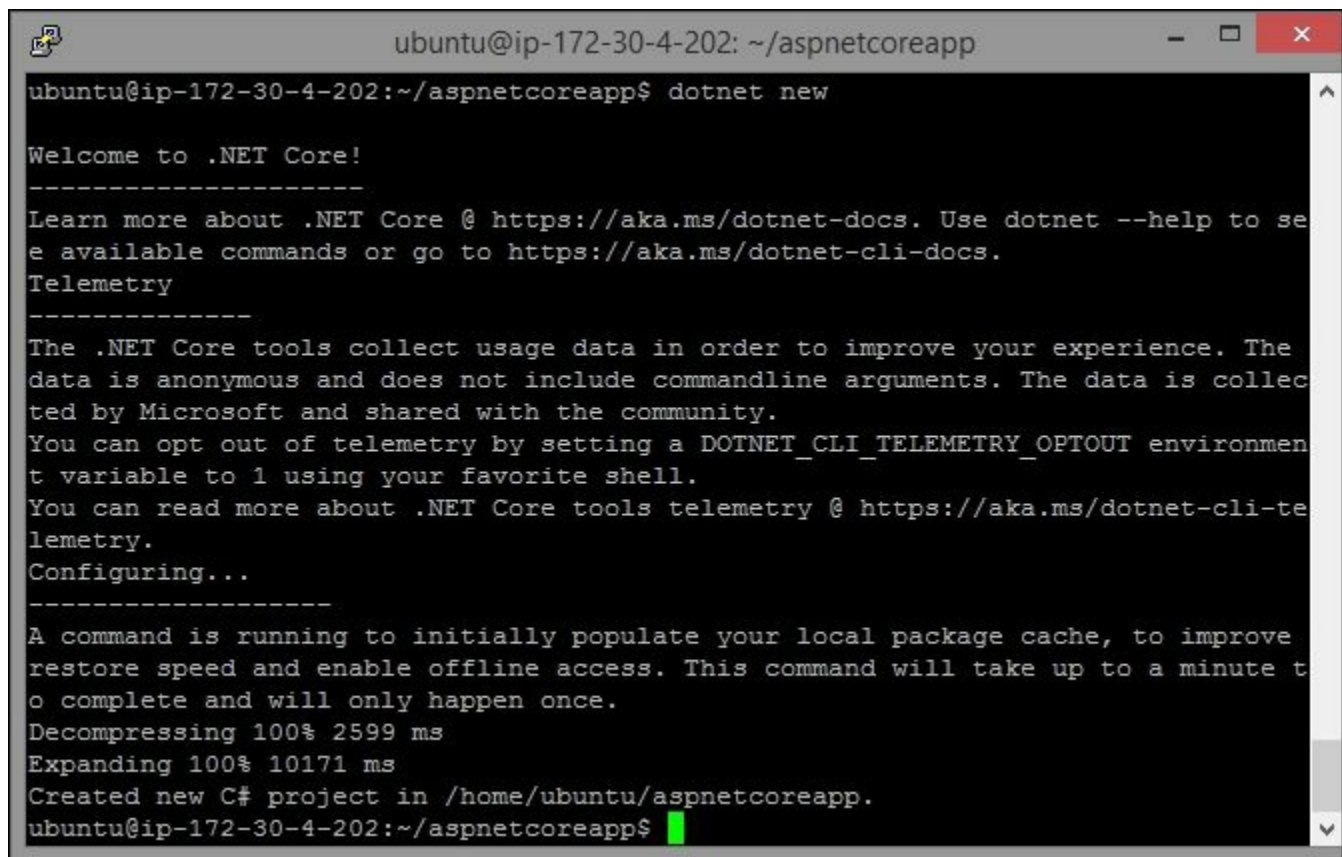
```
ubuntu@ip-172-30-4-202: ~                                    _ □ ×

.app-1.0.0.
Preparing to unpack .../dotnet-sharedframework-microsoft.netcore.app-1.0.0_1.0.0
-1_amd64.deb ...
Unpacking dotnet-sharedframework-microsoft.netcore.app-1.0.0 (1.0.0-1) ...
Selecting previously unselected package dotnet-dev-1.0.0-preview2-003121.
Preparing to unpack .../dotnet-dev-1.0.0-preview2-003121_1.0.0-preview2-003121-1
_amd64.deb ...
Unpacking dotnet-dev-1.0.0-preview2-003121 (1.0.0-preview2-003121-1) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up libllvm3.6:amd64 (1:3.6-2ubuntu1~trusty1) ...
Setting up liblldb-3.6 (1:3.6-2ubuntu1~trusty1) ...
Setting up libunwind8 (1.1-2.2ubuntu3) ...
Setting up liburcu1 (0.7.12-0ubuntu2) ...
Setting up liblttng-ust-ctl2:amd64 (2.4.0-4ubuntu1) ...
Setting up liblttng-ust0:amd64 (2.4.0-4ubuntu1) ...
Setting up dotnet-host (1.0.1-1) ...
Setting up dotnet-hostfxr-1.0.1 (1.0.1-1) ...
Setting up dotnet-sharedframework-microsoft.netcore.app-1.0.0 (1.0.0-1) ...
Setting up dotnet-dev-1.0.0-preview2-003121 (1.0.0-preview2-003121-1) ...
This software may collect information about you and your use of the software, an
d send that to Microsoft.
Please visit http://aka.ms/dotnet-cli-eula for more information.
Processing triggers for libc-bin (2.19-0ubuntu6.9) ...
ubuntu@ip-172-30-4-202:~$
```

Creating a new ASP.NET 5 project

Issue the following commands to create a new directory where we will create the ASP.NET 5 application. The first command (**mkdir** - make directory) is for creating a directory in Linux and the second command (**cd** - change directory) is for going inside the folder. And the last command is the command line to create a .NET Core application:

```
mkdir aspnetcoreapp
cd aspnetcoreapp
dotnet new
```

The following screen will be displayed:

This will create the .NET Core application, which has a couple of files —`Program.cs` and `project.json`. It's a bare minimum application that does not have even `Startup` file.

We need to add the `Kestrel HTTP Server` package as a dependency in `project.json`. You can edit the file by issuing the command **vi project.json** . By default, the vi editor will open the file in read-only mode. You need to press *Esc + I*

in order to make it to the edit mode. Add the line
"Microsoft.AspNetCore.Server.Kestrel": "1.0.0" as shown in the following
screenshot:

Press the Escape key and "*:*" and type `wq` to write into and quit the `vi` editor.

As we have added the dependency, we need to restore the packages by executing the
following command:

```
dotnet restore
```

Once you enter this command, all the packages will be restored as shown in the
following screenshot:

```
log  : Installing Microsoft.Extensions.FileProviders.Abstractions 1.0.0.
log  : Installing Microsoft.AspNetCore.Http.Features 1.0.0.
log  : Installing Microsoft.Extensions.Primitives 1.0.0.
log  : Installing Microsoft.Extensions.DependencyInjection.Abstractions 1.0.0.
log  : Installing Microsoft.Extensions.Configuration.Abstractions 1.0.0.
log  : Installing Microsoft.AspNetCore.Http 1.0.0.
log  : Installing Microsoft.AspNetCore.Http.Extensions 1.0.0.
log  : Installing Microsoft.Extensions.FileProviders.Physical 1.0.0.
log  : Installing Microsoft.AspNetCore.Hosting.Abstractions 1.0.0.
log  : Installing Microsoft.AspNetCore.Hosting.Server.Abstractions 1.0.0.
log  : Installing Microsoft.Extensions.Logging 1.0.0.
log  : Installing Microsoft.Extensions.Options 1.0.0.
log  : Installing Microsoft.Extensions.DependencyInjection 1.0.0.
log  : Installing Microsoft.Extensions.Configuration 1.0.0.
log  : Installing Microsoft.Extensions.Configuration.EnvironmentVariables 1.0.0.
log  : Installing Microsoft.Extensions.PlatformAbstractions 1.0.0.
log  : Installing Microsoft.AspNetCore.Hosting 1.0.0.
log  : Installing Microsoft.Extensions.Logging.Abstractions 1.0.0.
log  : Installing Microsoft.AspNetCore.Server.Kestrel 1.0.0.
log  : Writing lock file to disk. Path: /home/ubuntu/aspnetcoreapp/project.lock.
json
log  : /home/ubuntu/aspnetcoreapp/project.json
log  : Restore completed in 2868ms.
ubuntu@ip-172-30-4-202:~/aspnetcoreapp$
```
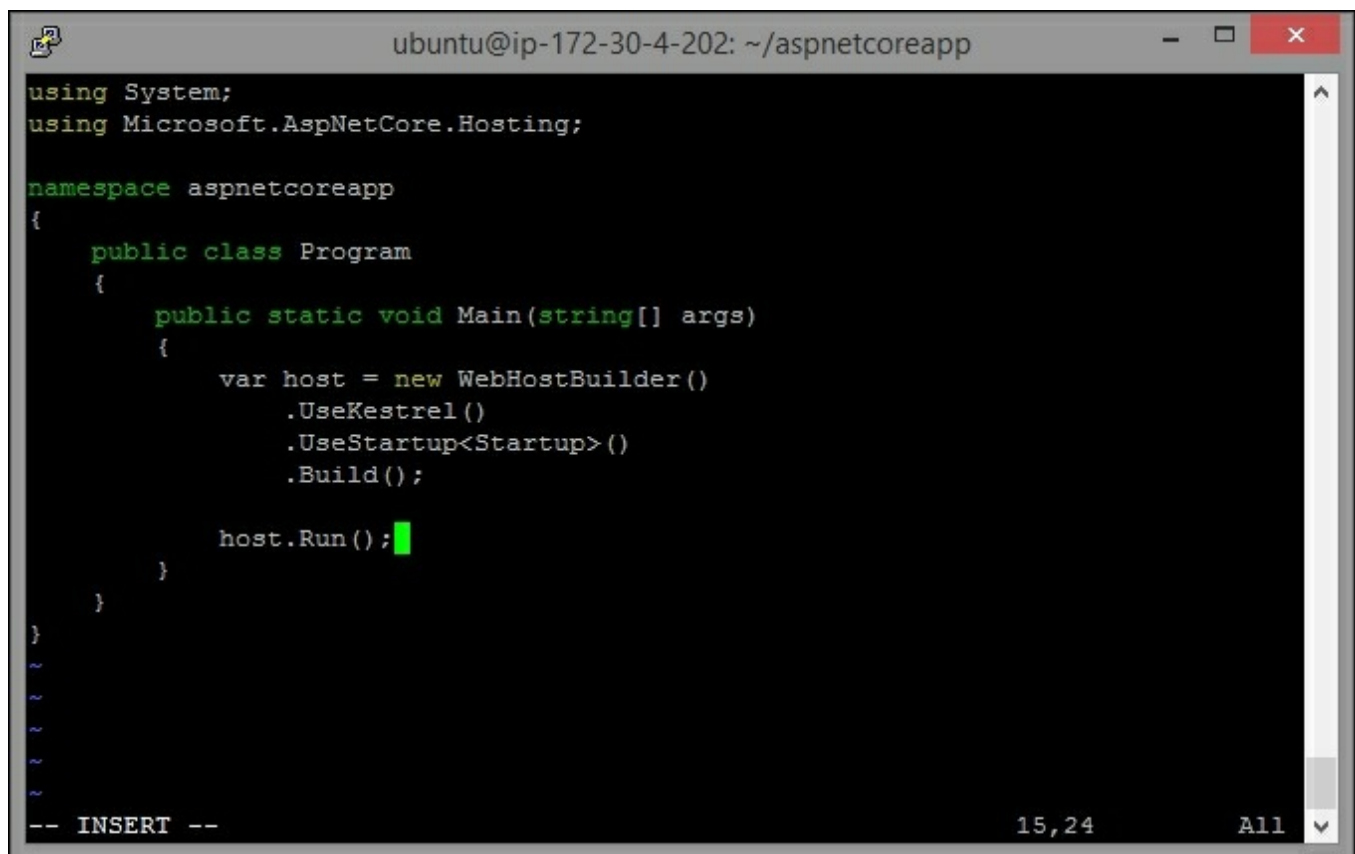
Create a new file, `Startup.cs`, with the following content. You can create a new file by issuing the command **vi Startup.cs** . As usual, we need to press *Esc + I* to make the file in write and read mode. Paste the following content (you can paste it by right-clicking on the mouse after copying it from here):

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
namespace aspnetcoreapp
{
  public class Startup
  {
    public void Configure(IApplicationBuilder app)
    {
      app.Run(context =>
        {
          return context.Response.WriteAsync("This is ASP.NET Core
application running in Linux!");
        });
    }
  }
}
```

Press *Esc + :* and type `wq`, to save the file. Update the `Program.cs` file with the following content:

```
namespace aspnetcoreapp
{
  public class Program
  {
    public static void Main(string[] args)
    {
      var host = new WebHostBuilder()
      .UseKestrel()
      .UseStartup<Startup>()
      .Build();
       host.Run();
    }
  }
}
```

You'll see the following screen:

We have created the ASP.NET Core web application. Now we need to install **Nginx**, a reverse proxy server, which enables you to offload work such as serving static content, caching, and compressing requests. You can configure Nginx to listen on a particular port (we'll discuss the details later in this chapter). You can install Nginx
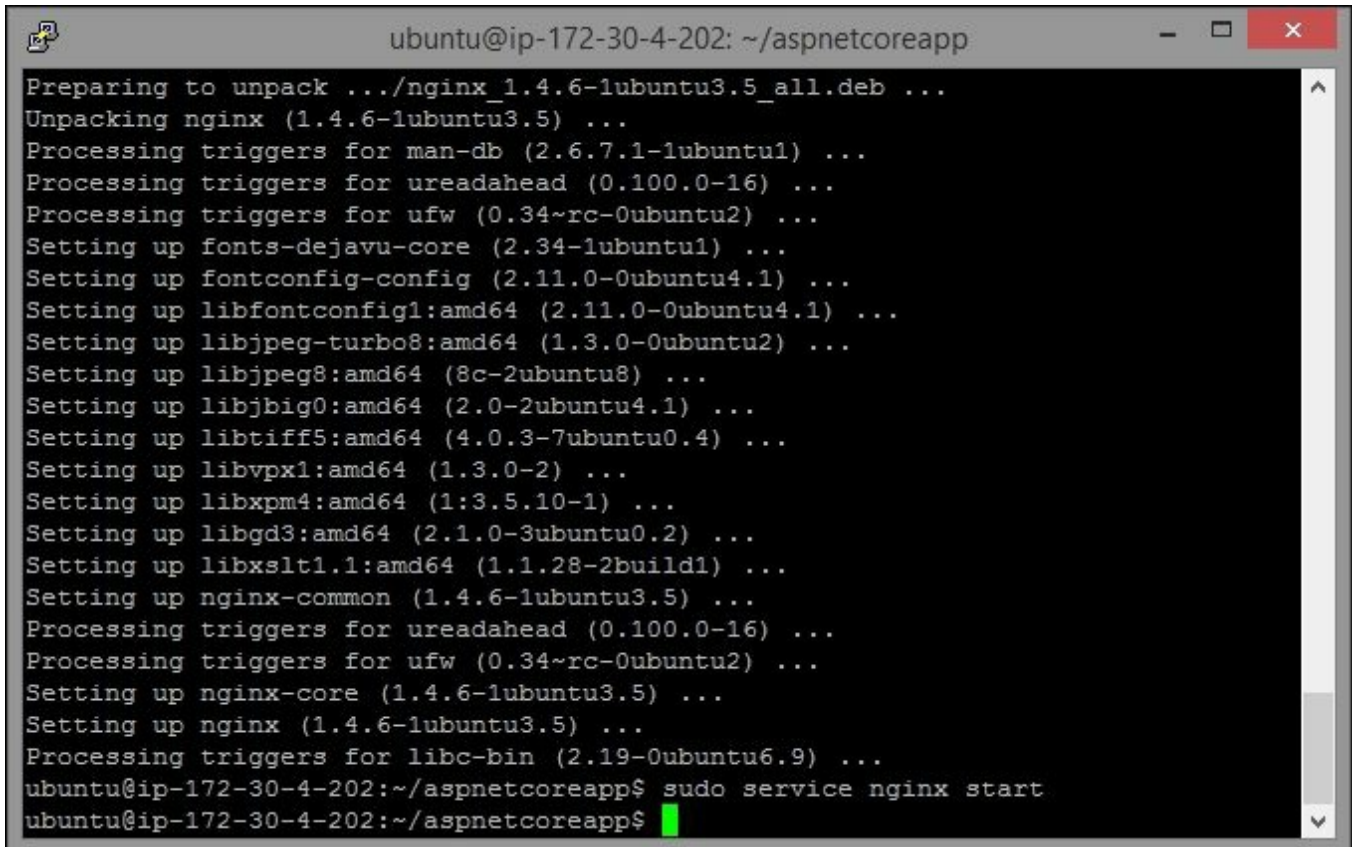
by issuing the following command:

```
sudo apt-get install nginx
```

Once it is installed, you can issue the following command to start the service:

```
sudo service nginx start
```

When you'll run the command, you'll see the following screen:
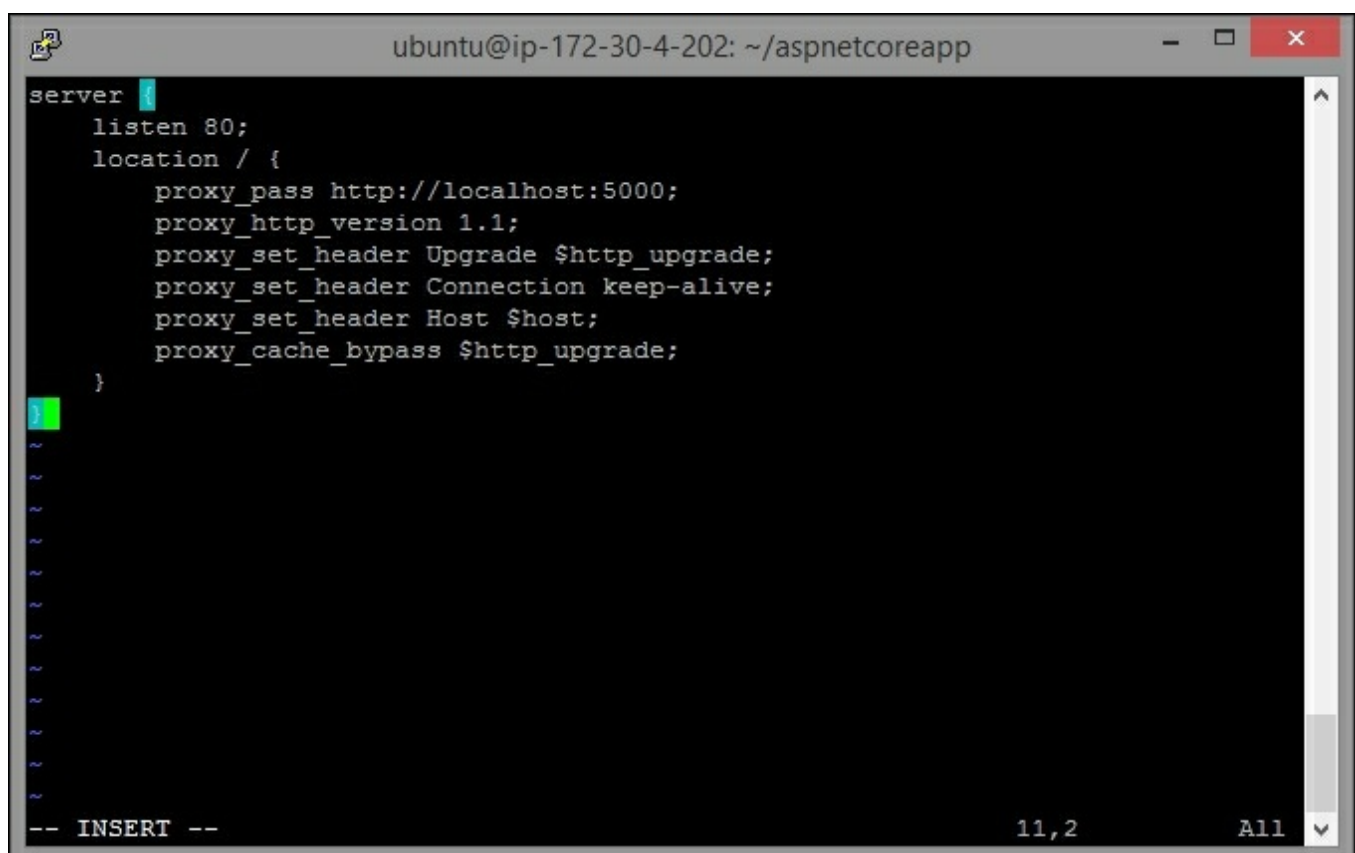
```
ubuntu@ip-172-30-4-202: ~/aspnetcoreapp

Preparing to unpack .../nginx_1.4.6-1ubuntu3.5_all.deb ...
Unpacking nginx (1.4.6-1ubuntu3.5) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Processing triggers for ureadahead (0.100.0-16) ...
Processing triggers for ufw (0.34~rc-0ubuntu2) ...
Setting up fonts-dejavu-core (2.34-1ubuntu1) ...
Setting up fontconfig-config (2.11.0-0ubuntu4.1) ...
Setting up libfontconfig1:amd64 (2.11.0-0ubuntu4.1) ...
Setting up libjpeg-turbo8:amd64 (1.3.0-0ubuntu2) ...
Setting up libjpeg8:amd64 (8c-2ubuntu8) ...
Setting up libjbig0:amd64 (2.0-2ubuntu4.1) ...
Setting up libtiff5:amd64 (4.0.3-7ubuntu0.4) ...
Setting up libvpx1:amd64 (1.3.0-2) ...
Setting up libxpm4:amd64 (1:3.5.10-1) ...
Setting up libgd3:amd64 (2.1.0-3ubuntu0.2) ...
Setting up libxslt1.1:amd64 (1.1.28-2build1) ...
Setting up nginx-common (1.4.6-1ubuntu3.5) ...
Processing triggers for ureadahead (0.100.0-16) ...
Processing triggers for ufw (0.34~rc-0ubuntu2) ...
Setting up nginx-core (1.4.6-1ubuntu3.5) ...
Setting up nginx (1.4.6-1ubuntu3.5) ...
Processing triggers for libc-bin (2.19-0ubuntu6.9) ...
ubuntu@ip-172-30-4-202:~/aspnetcoreapp$ sudo service nginx start
ubuntu@ip-172-30-4-202:~/aspnetcoreapp$
```

Configuring the Nginx server

Configure the Nginx server by modifying the file (`/etc/nginx/sites-available/default`) to have the following content—so that Nginx will forward the request to ASP.NET. In order to modify this file, you need to have sufficient rights—try switching to a super user. The **Sudo su** is the command for switching it to a super user. See the following code:

```
server {
  listen 80;
  location / {
    proxy_pass http://localhost:5000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection keep-alive;
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
  }
}
```

The code looks like the following:

Run the application by executing the following command:

```
dotnet run
```

You will see the following screen:

Now access the application from your browser using the public DNS (AWS created the public DNS when the instance was launched):
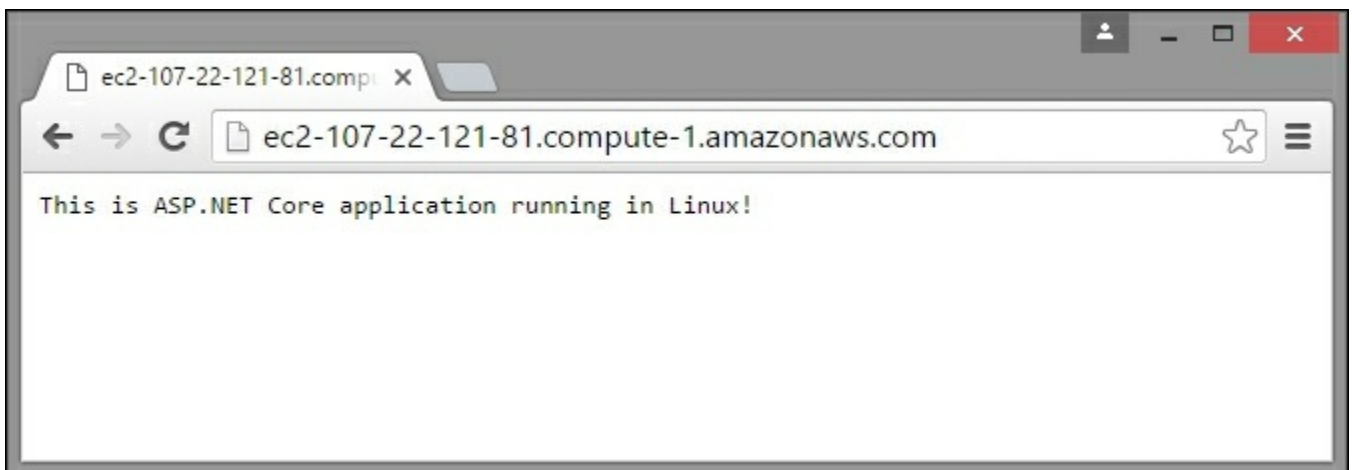
Voila! We have created the ASP.NET Core web application and launched it in the Linux box. We have even used the cloud through **Amazon Web Services** (**AWS**).

Summary

In this chapter, you have learned about the different components available in the `project.json` file, where all the configuration of your ASP.NET Core goes. We have discussed how to sign up to the Microsoft Azure cloud platform and deploy the ASP.NET Core application in the Azure platform. We have also learned how to create and deploy the ASP.NET Core web application in Linux using Amazon Web Services in the cloud.

Chapter 10. Building HTTP-based Web Services Using ASP.NET Web API

So far, we have learned how to create web applications using ASP.NET Core. But there are times when simply creating a web application is not enough. Let's assume you are using ASP.NET Core to create a web application that provides weather information for all the cities across the world. People access your web application to find out weather information, and they are satisfied with the service. But this weather information may be needed by many other websites or web applications, such as tourism websites, news websites, and many other mobile applications.

Instead of writing the code all over again for their websites, you can create and publish the web services and the websites can consume the required web services whenever they need to.

In this chapter, you are going to learn about the following topics:

- What an HTTP-based service is and how it is useful
- What Fiddler is
- How to compose an HTTP request using Fiddler and fire the same in order to get an HTTP response
- How to design and implement the HTTP service using Web API

Microsoft provides ASP.NET Web API for programmers to build HTTP-based services. But HTTP is not just used to serve the webpages. You can use HTTP as a platform. This brings several advantages:

- As web services built using ASP.NET Web API use HTTP for communication, these web services can be consumed from all kinds of applications from console applications to web applications, and from WCF services to mobile applications
- Whenever there is any change in the logic/code of the web services, the clients (the websites that consume the services) do not need to change anything. They can consume the web services just as they were consuming them earlier

HTTP basics

HTTP is a powerful platform for building services. You can use the existing HTTP verbs to build services. For example, you can use the existing HTTP verb GET to get the list of products or POST to update information about the product. Let's take a quick look at how HTTP works with respect to building the services.

There is no difference in the underlying mechanism between serving the HTML pages in ASP.NET MVC and serving the data in the context of HTTP services. Both follow a request-response pattern and the same routing mechanism.

An HTTP request can be sent from any client (desktop, laptop, tablet, mobile, and so on) to the server and the server will respond back with an HTTP response. An HTTP response can be sent to the client in any format such as JSON or XML. This is shown in the following figure:
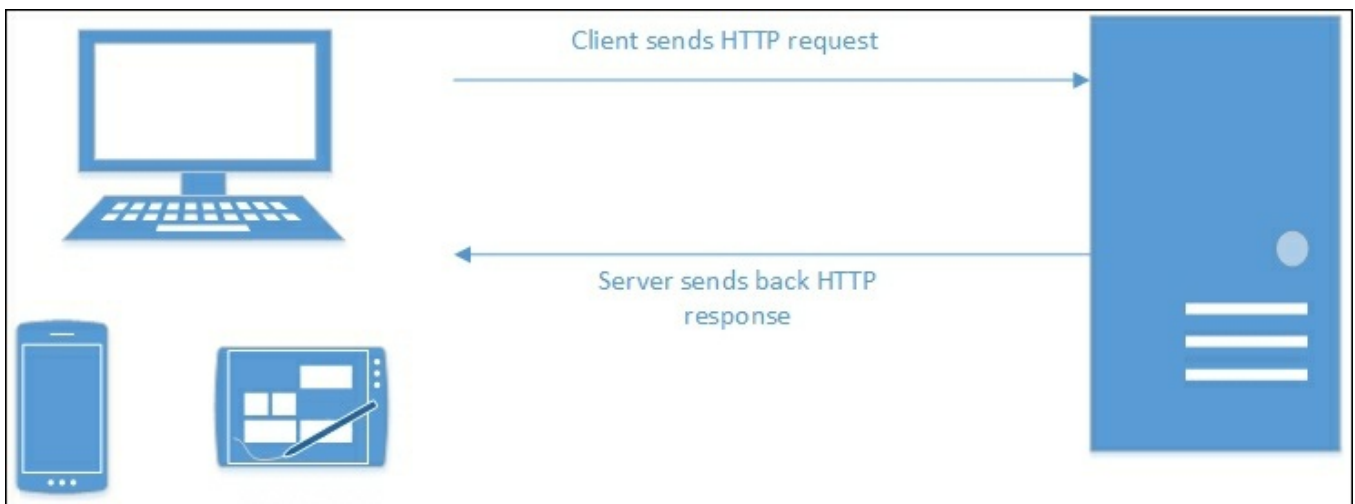
In the preceding diagram, a request is sent from the desktop computer (it could equally be sent from a mobile or tablet; it makes no difference) and the server sends back the HTTP response for the request. As HTTP is supported in most of the devices, it is ubiquitous.

HTTP verbs

HTTP verbs describe how the request has to be sent. These are the methods defined in HTTP that dictate how the HTTP requests are sent from the client to the server

GET method

When we use an HTTP GET request, the information is passed through the URL itself:

```
GET api/employees/{id}
```

This GET request gets the employee information based on the passed ID. The advantage of using the GET request is that it is lightweight, and all the required information will be passed in the URL or header itself, as shown in the following diagram:
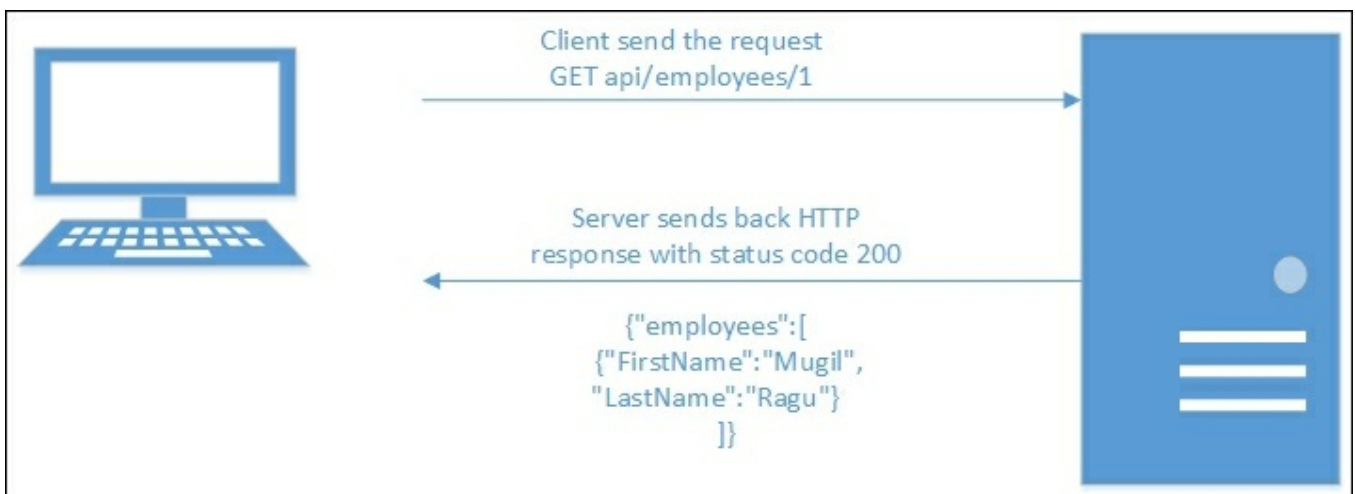
PUT method

The `PUT` method is used to create a resource or to update it. `PUT` is an idempotent operation, meaning that the expected behavior would not change even if it is executed multiple times:
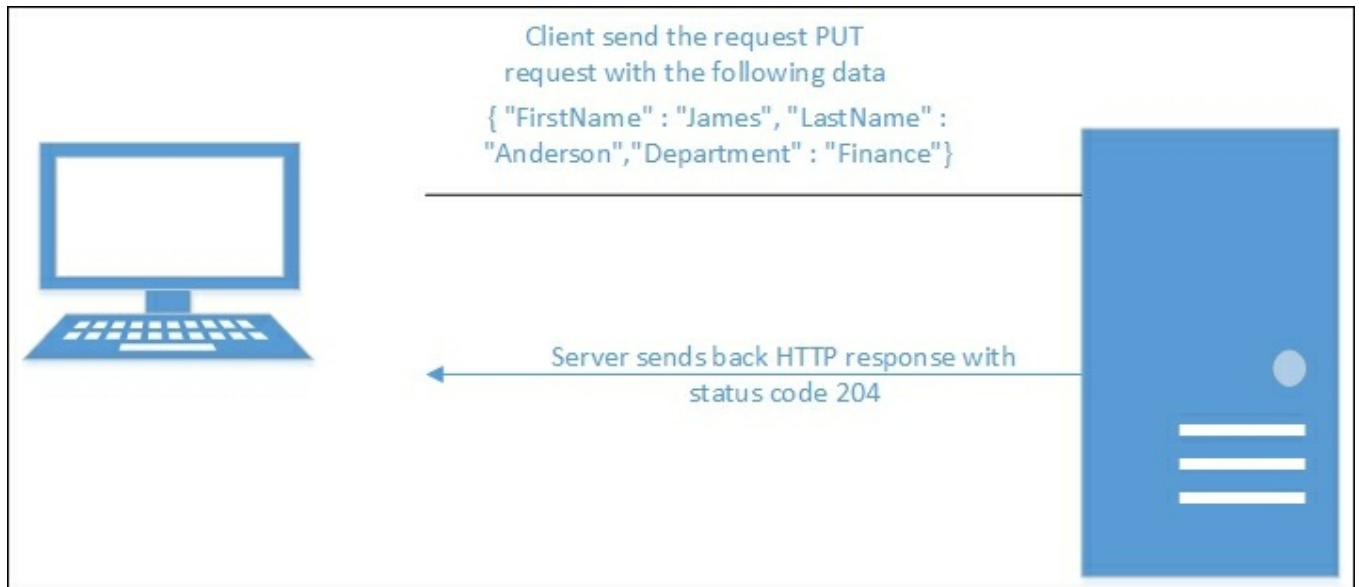
POST method

You can use POST to create or update the resource. Usually, POST is used to create the resource rather than update it. As per HTTP standards, whenever you create a new resource, you should return a **201 HTTP** status code:

Client send the request POST
request with the following data

{ "FirstName" : "James", "LastName" :
"Anderson","Department" : "IT"}

Server sends back HTTP response with
status code 201 – new resource created

{ "FirstName" : "James", "LastName" :
"Anderson","Department" :
"IT","Id":1234}
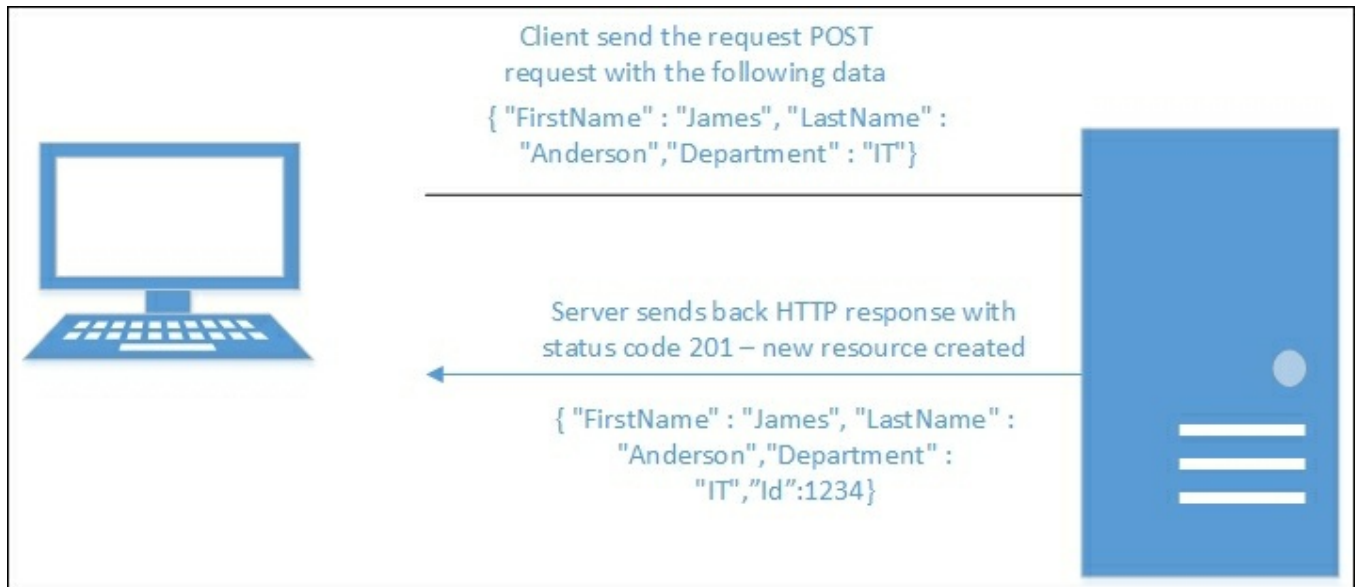
DELETE method

The DELETE method is used to delete the resource. Usually, when you delete a resource, you would be passing the ID as a parameter, and you would not be passing anything in the body of the request:
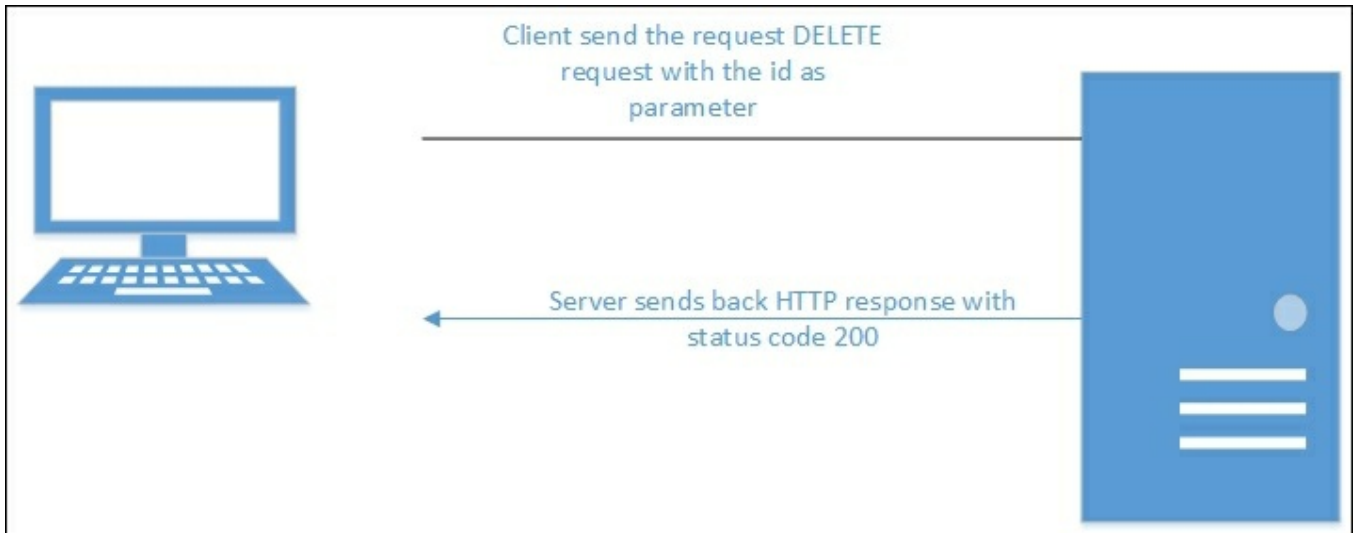
Usually, HTTP services would be consumed by other applications and services. Applications that consume services are referred to as clients. One of the options to test HTTP services is to build the clients. But this would be time-consuming, and we may throw away the client code once we test the HTTP services.

Another option, which is widely used, is to use applications that enable us to fire HTTP requests and monitor the responses. There are many applications available, Fiddler being one such widely used application.

Fiddler tool

Fiddler is the proxy server application used to monitor the HTTP and HTTPS traffic. You can monitor the requests that are being sent to the server from the client, the responses that are sent to the client, and the responses that are being received from the server. It is like seeing the traffic in the pipe between the server and the client. You can even compose a request, fire it, and analyze the response received without ever needing to write the client for the services.

You can download Fiddler at http://www.telerik.com/fiddler. You'll see the following window:
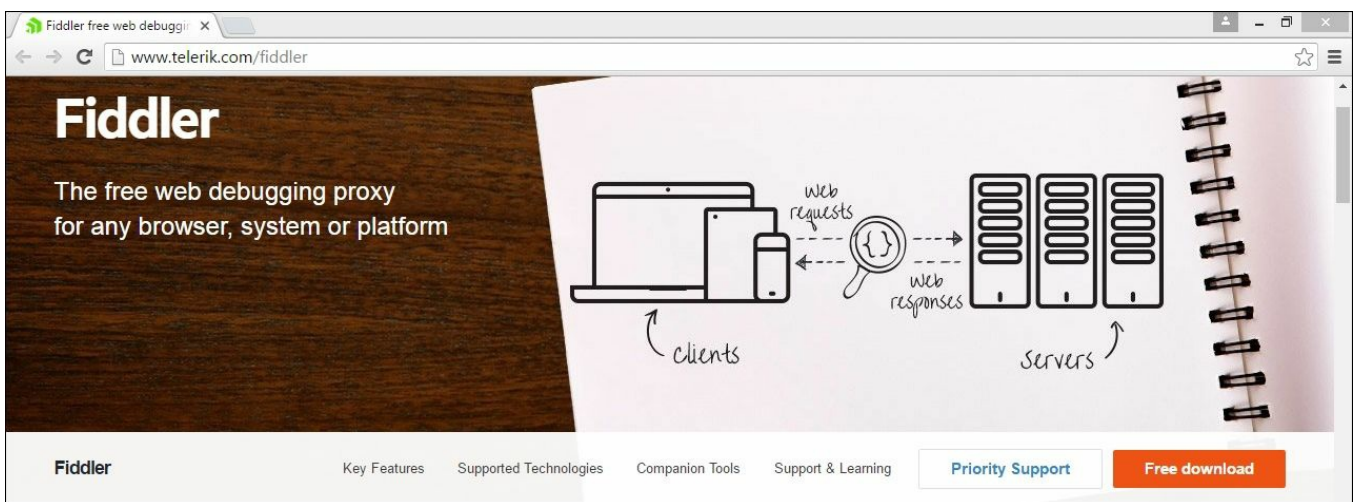
Enough theory. Let us create a simple web service using ASP.NET Web API.

Fire up Visual Studio 2015:
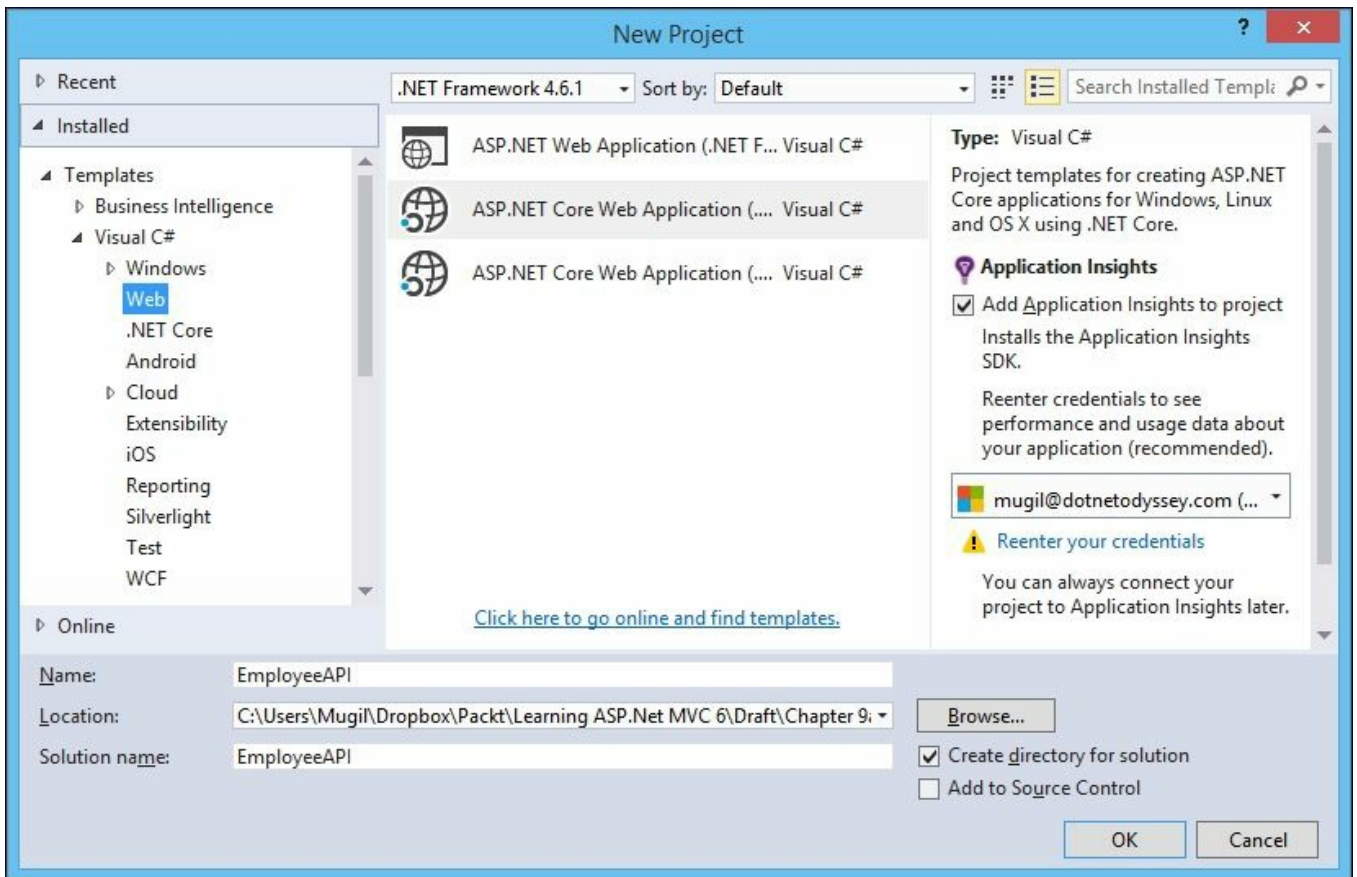
When you click **OK**, a Web API solution will be created. Just as the ASP.NET Core application controller inherits from the Controller class.
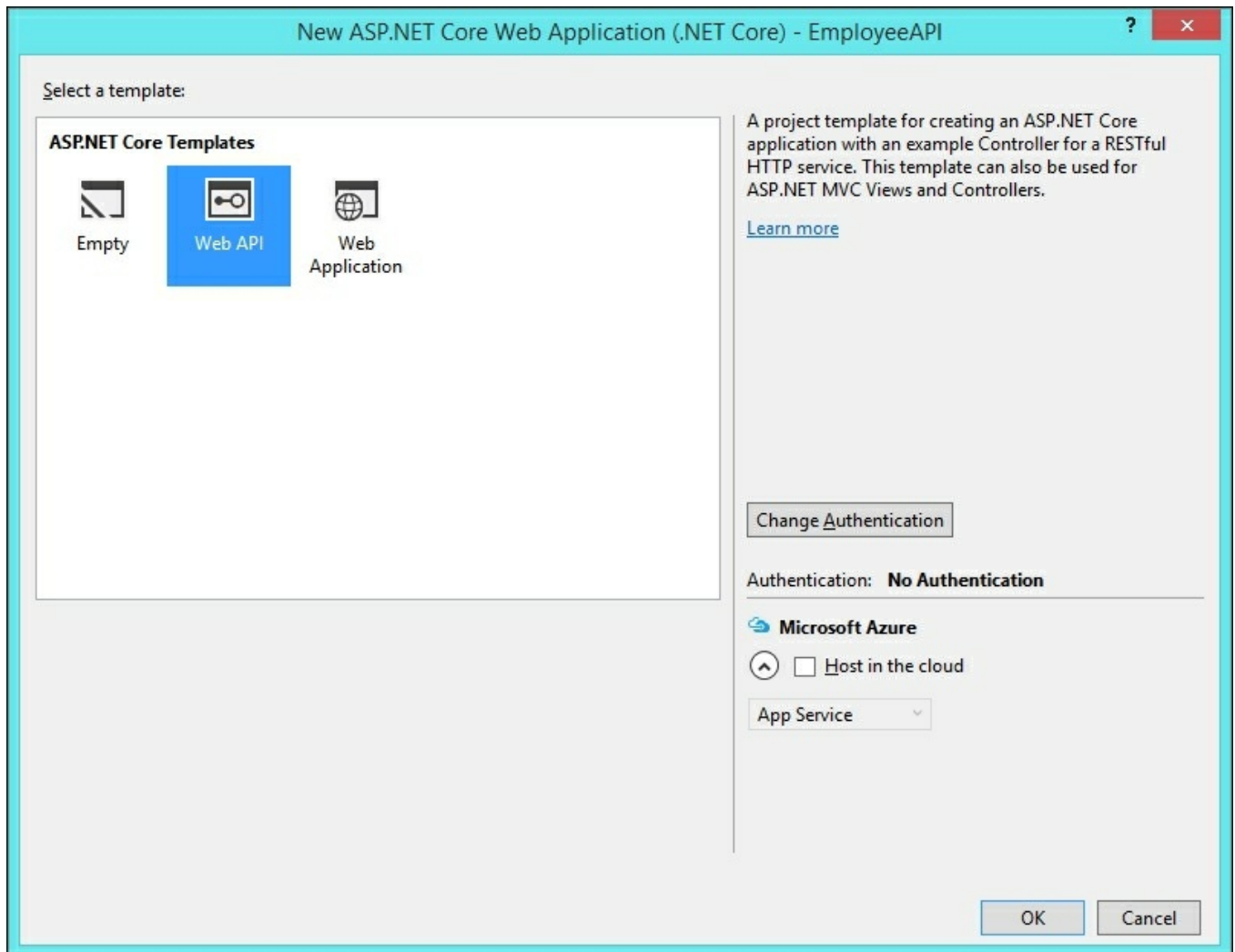
The Web API class will also inherit from the same Controller class. This is the difference between ASP.NET Core and earlier versions of ASP.NET MVC. In earlier versions, all Web API controller classes inherited from the `ApiController` class. In ASP.NET 5, it has been unified, and the same base Controller class is being used for both building web applications and services.

The following is the `ValuesController` class that will be created by default when you choose the **Web API** template option when creating the project:
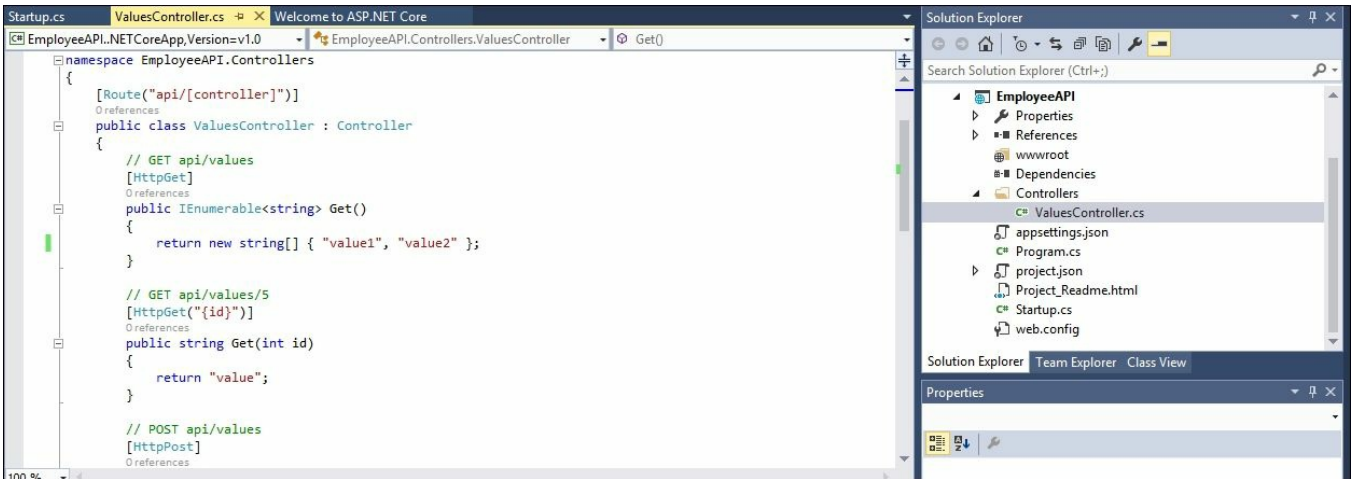
Before we create our own custom Controller, let's analyze the default API Controller. In the `ValuesController` file, several API methods are already defined.

There are two overloaded `GET` methods—one with a parameter and another without a parameter. The `GET` method without a parameter returns all the resources of the type. In this case, we are returning just a couple of strings. In the real world, we would be returning the metadata of the resources. For example, if we fire the `GET` request on the movies API Controller, it would return information about all the movies. The `GET` method with an `id` parameter returns the resource whose ID matches with the passed ID. For example, if you pass a movie ID, it would return the information about that movie. The body of the other methods, such as `PUT`, `POST`, and `DELETE`, are empty in this Controller, and we will talk about these methods later.

When you run the application, you will get the following output:
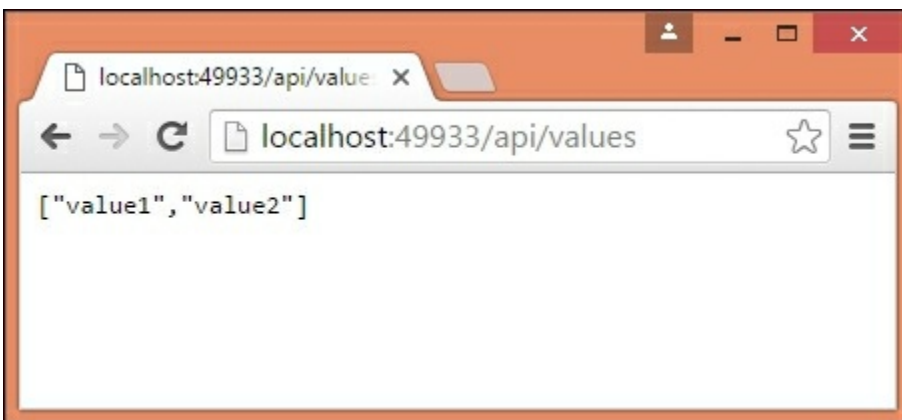
By default, it fires a request to `api/values`, and the values are displayed in the

browser.

Let's learn how to fire an HTTP request from the Fiddler application. Open the Fiddler application. In the bottom left-hand corner, select the **Web Browsers** option in the red box. Choosing this option will enable us to view the traffic coming from the **Web Browsers**:
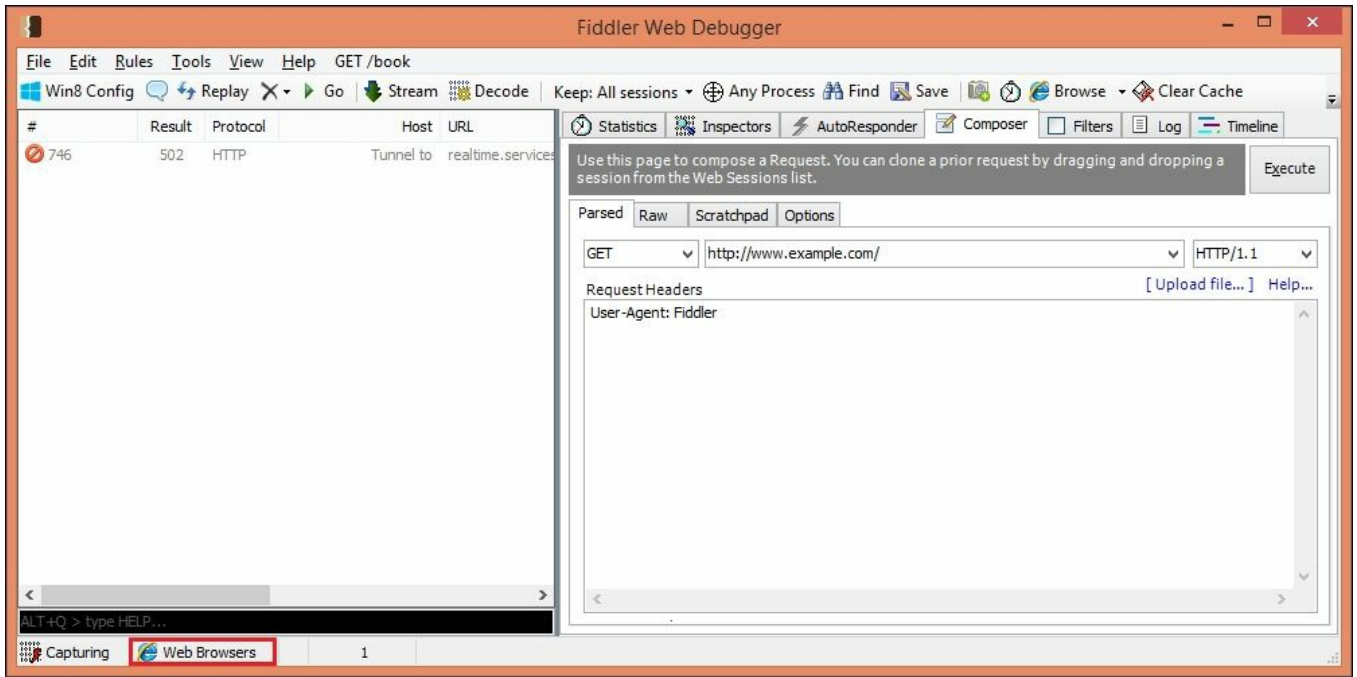
Select the **Composer** tab, enter the URL `http://localhost:49933/api/values`, as shown in the following screenshot, and click the **Execute** button in the top right-hand corner:
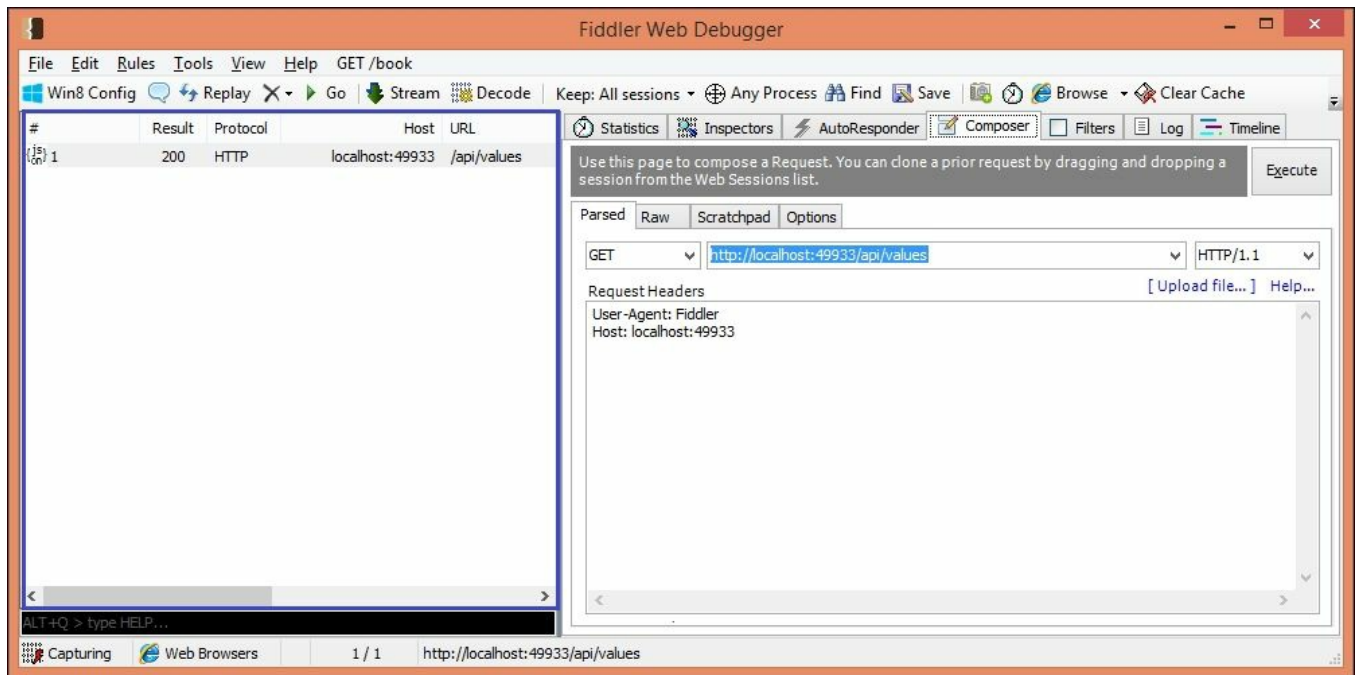
Once you click the **Execute** button, an HTTP session will be created, visible on the left-hand side pane (highlighted in the blue box). Click on the session and select the **Inspectors** tab on the top right-hand side pane. Select the JSON tab in the bottom right-hand side pane (highlighted by the purple-bordered box in the following screenshot).

You can see the JSON data returned from the HTTP request—**value1** and **value2** in the following screenshot:
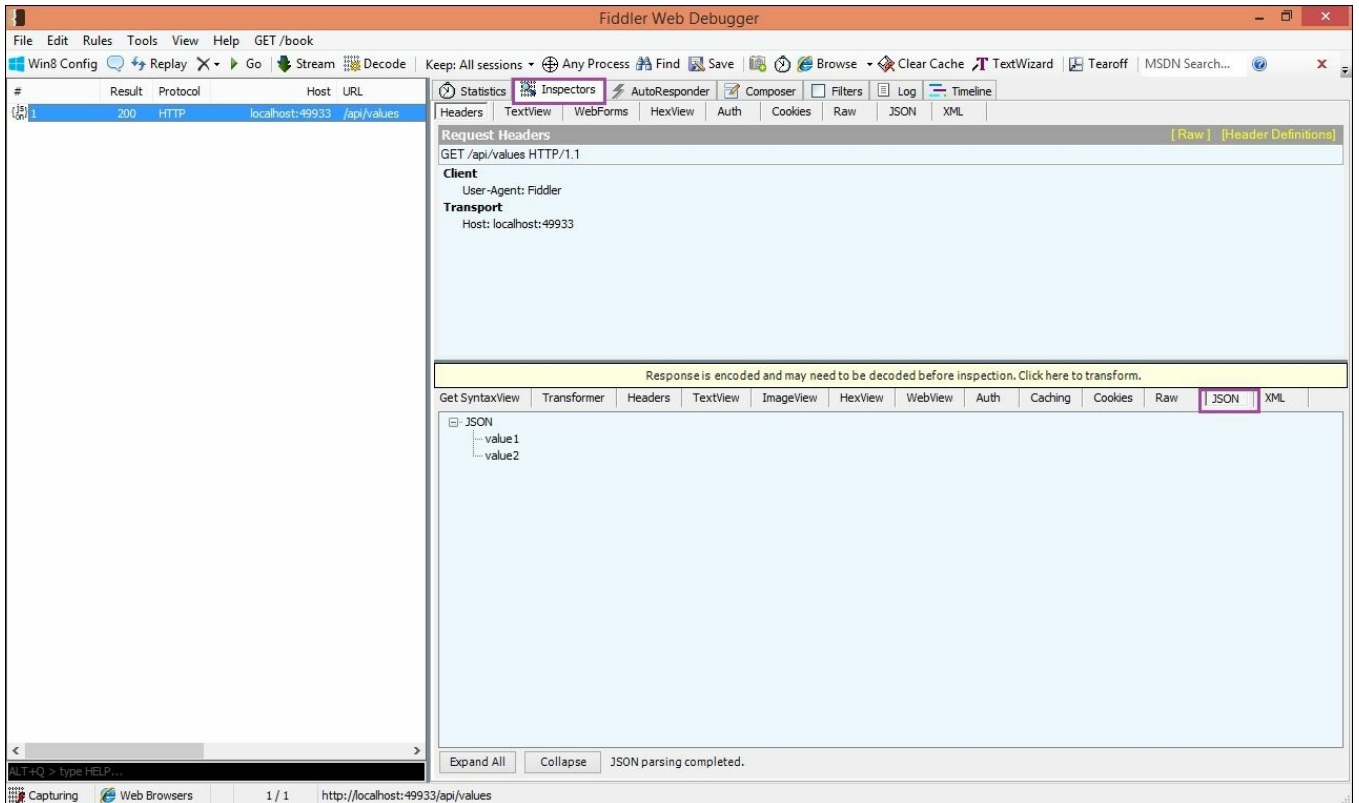
Now it's our turn to write a custom API.

In this custom API, we are going to provide API methods to create an employee object, list all the employee objects, and delete an employee object.

First, let us create a model for the employee. We need to create a folder to hold these models. Right-click on the project, select **Add** | **New folder**, and name the folder as Models:
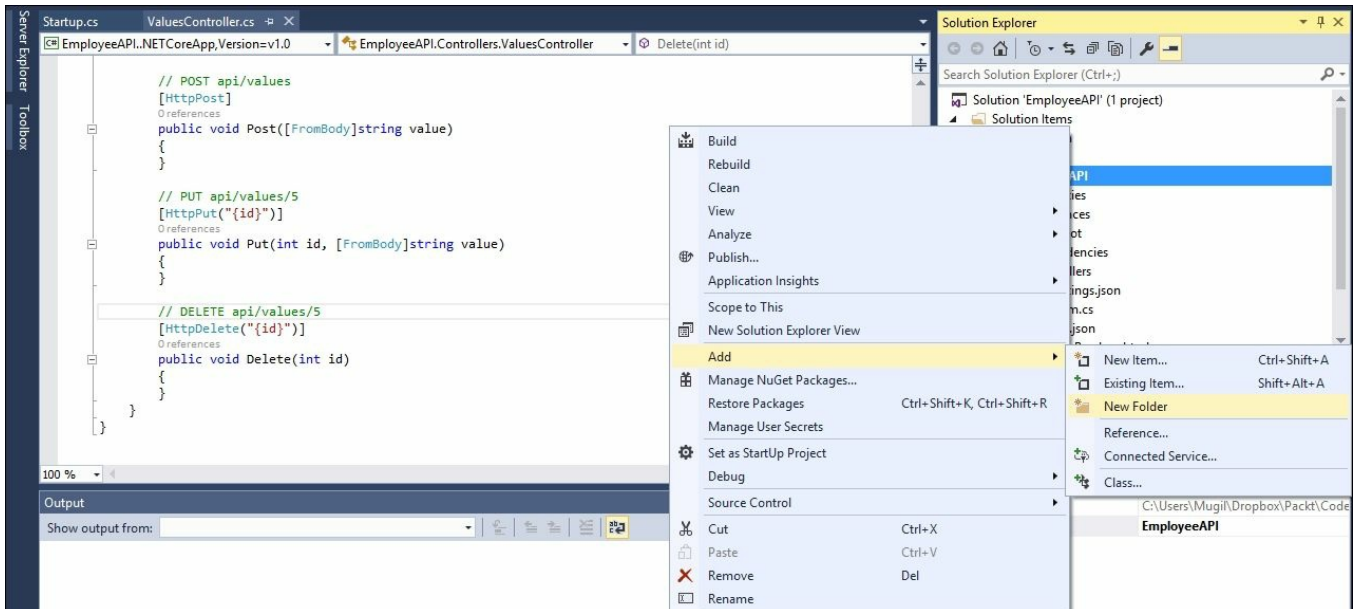
Right-click on the `Models` folder and select **Add** | **New Item…** to create an employee model class. This employee model class is just a POCO class. See the following code:

```
public class Employee
{
  public int Id {get; set;}
  public string FirstName {get; set;}
  public string LastName {get; set;}
  public string Department {get; set;}
}
```

Then, we define the repository interface to handle the model:

```
public interface IEmployeeRepository
{
  void AddEmployee(Employee e);
  IEnumerable<Employee> GetAllEmployees();
  Employee GetEmployee(int id);
  Employee RemoveEmployee(int id);
  void UpdateEmployee(Employee employee);
}
```

Then we implement the interface for this model:

```
public class EmployeeRepository : IEmployeeRepository
{
  private static List<Employee> employees = new List<Employee>();

  public EmployeeRepository()
  {
```

```csharp
    Employee employee1 = new Employee
    {
      FirstName = "Mugil",
      LastName = "Ragu",
      Department = "Finance",
      Id = 1
    };

    Employee employee2 = new Employee
    {
      FirstName = "John",
      LastName = "Skeet",
      Department = "IT",
      Id = 2
    };

    employees.Add(employee1);
    employees.Add(employee2);
  }

  public IEnumerable<Employee> GetAllEmployees()
  {
    return employees;
  }

  public void AddEmployee(Employee e)
  {
    e.Id = GetNextRandomId();
    employees.Add(e);
  }

  public Employee GetEmployee(int id)
  {
    return employees.Where(emp => emp.Id == id).FirstOrDefault();
  }

  public Employee RemoveEmployee(int id)
  {
    Employee employee = employees.Where(emp => emp.Id ==
id).FirstOrDefault();
    if (employee !=null )
    {
      employees.Remove(employee);
    }
    return employee;
  }

  public void UpdateEmployee(Employee emp)
  {
    Employee employee = employees.Where(e => e.Id ==
emp.Id).FirstOrDefault();
```

```
    if(employee != null)
    {
    employee.Department = emp.Department;
    employee.FirstName = emp.FirstName;
    employee.LastName = emp.LastName;
    }
  }

  private int GetNextRandomId()
  {
    int id = -1;
    bool isIdExists;
    Random random = new Random();
    do
    {
      id = random.Next();
      isIdExists = employees.Any(emp => emp.Id == id);
    } while (isIdExists);
    return id;
  }
}
```

There are few things to be noted in the implementation class:

- We have decided not to use the database as our objective is to create an HTTP service using Web API, and not to write the data access code.
- We are using an in-memory list to hold the data. All the operations will be performed on this list. As a matter of fact, the data could be in any form, ranging from relational databases to a simple in-memory list.
- In the constructor method, we are adding an object to the list. This list will be acting as the database for our HTTP service.
- The `GetAllEmployees` API method will return all the employees as the `IEnumerable` interface.
- The `AddEmployee` method will add the employee (passed as a parameter) to the list.
- The `GetEmployee` method will return the employee whose ID matches that of the parameter.
- The `RemoveEmployee` method will remove the employee from the list.
- The `UpdateEmployee` method will update the employee information.
- The `GetNextRandomId` method will return the next available random integer. This integer value is being used to generate the employee ID.

Dependency Injection

In most real-world projects, we do not instantiate any objects using the `new` instance in any of the Controllers, the reason being that we don't want to have tight coupling between the dependent components (between the Controller and the repository). Instead, we pass an interface to the Controller, and the Dependency Injection container (such as **Unity**) will create an object for us when it is needed for the Controller. This design pattern is commonly referred to as **Inversion of Control**.

Let's say that a class by the name of *ClassA* uses another class, *ClassB*. In this case, it is enough for *ClassA* to know about the behavior, methods, and properties of *ClassB*, and it doesn't need the internal implementation details of *ClassB*. So, we can abstract *ClassB* and make an interface out of the class, and then have that interface as the parameter instead of the concrete class. The advantage of this approach is that we can pass any class at runtime as long as it implements a commonly agreed contract (interface).

In ASP.NET 5 (including ASP.NET Core and Web API), we have inbuilt support for Dependency Injection. In the `ConfigureServices` method, we have added the line (highlighted in bold) that performs the Dependency Injection. We instruct the inbuilt Dependency Injection container to create the `EmployeeRepository` class wherever we are referring to the `IEmployeeRepository` interface and we also instruct it to be a singleton; meaning that the same object (which is to be created by the Dependency Injection container) is to be shared for the entire lifecycle of the application:

```
public void ConfigureServices(IServiceCollection services)
{
  // Add framework services.
  services.AddApplicationInsightsTelemetry(Configuration);
  services.AddMvc();
  services.AddSingleton<IEmployeeRepository, EmployeeRepository>();

}
```

In the preceding code, we have used the Singleton pattern for the Dependency Injection, which creates services only the first time they are requested. There are other types of lifetime services such as **Transient** and **Scoped**. Transient lifetime services are created each time they are requested and Scoped lifetime services are created once per request. The following are code snippets created when you use such lifetimes:

```
services.AddTransient
  <IEmployeeRepository, EmployeeRepository>();
```

```
services.AddScoped <
IEmployeeRepository, EmployeeRepository>();
```

Now it's time to get into the meat of the action creating the API controller. Right-click on the **Controllers** folder and select **Add|New Item**. Then select **Web API Controller Class** from the list, as shown in the following screenshot. Name your Controller, and click the **Add** button:
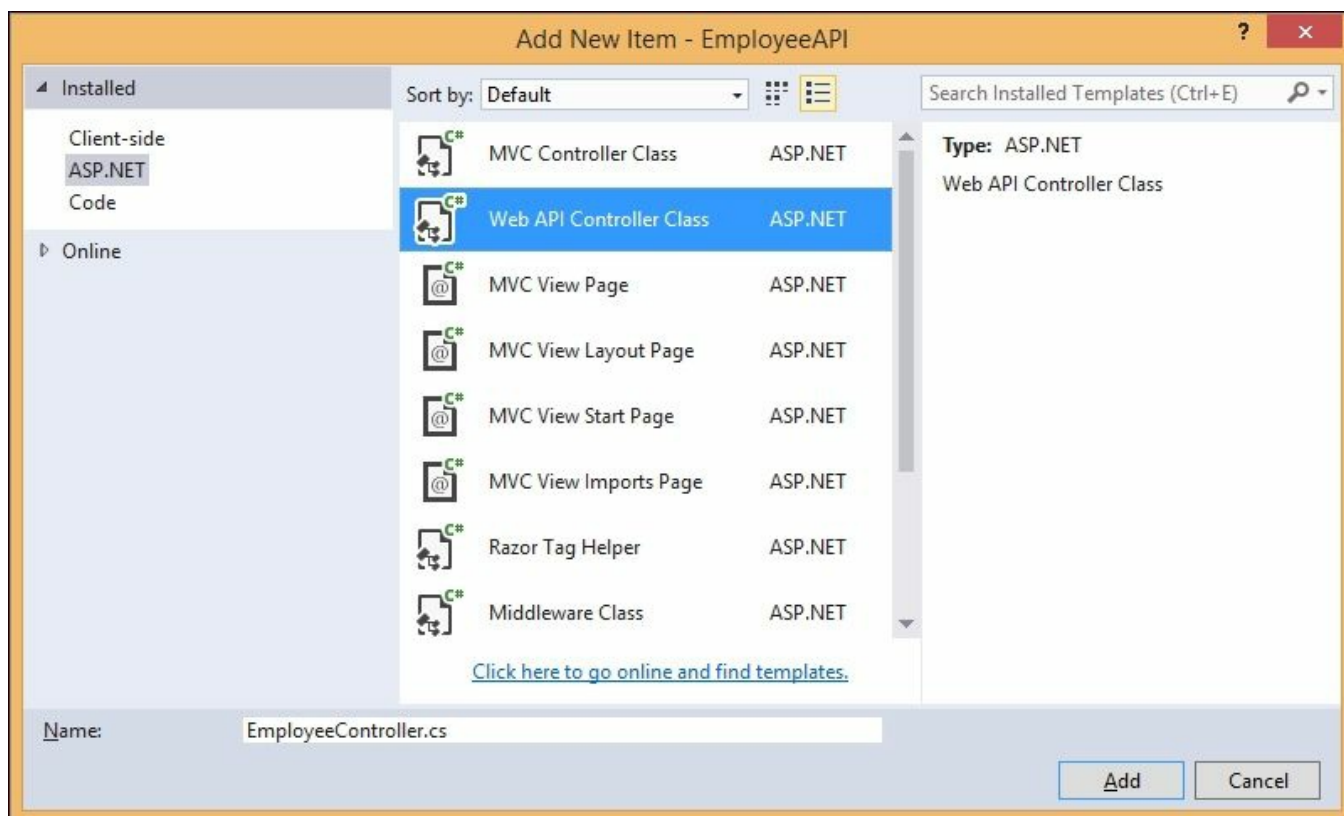
Remove the generated code in the Controller and add the following constructor:

```
public EmployeeController(IEmployeeRepository employeesRepo)
{
  employeeRepository = employeesRepo;
}
private IEmployeeRepository employeeRepository {get; set;}
```

In the preceding constructor, we are injecting the dependency. At the time of calling this constructor, the `EmployeeRepository` object will be created.

Let us implement a couple of `GET` methods—the first one will return all the employees' details and the second `GET` method will return the employee based on the passed employee ID:

```
public IEnumerable<Employee> GetAll()
{
  return employeeRepository.GetAllEmployees();
}

[HttpGet("{id}",Name ="GetEmployee")]
public IActionResult GetById(int id)
{
  var employee = employeeRepository.GetEmployee(id);
  if(employee == null)
  {
  return NotFound();
  }
  return new ObjectResult(employee);
}
```

Let us call these HTTP methods from Fiddler.

Run the solution, open the Fiddler application, and click on the **Composer** tab.

Select the HTTP method (we have chosen the GET method as we have a GET API method) and enter the URL http://localhost:49933/api/employee .

Please note that when I run my application, it runs on port 49933; the port number will be different in your case, so construct your URL accordingly.

Once you enter the URL and the method is selected, click the **Execute** button as shown in the following screenshot:
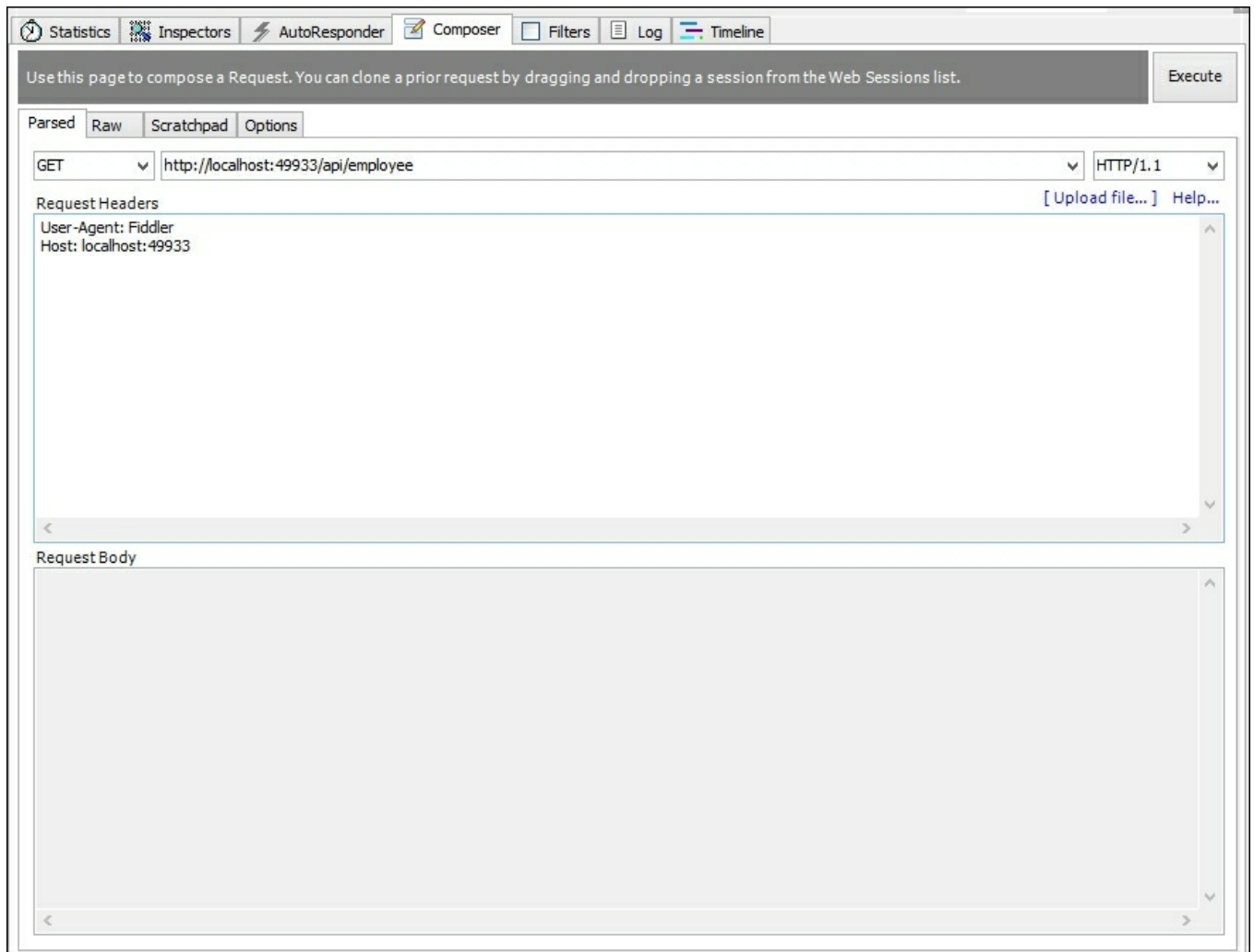
Once you click the **Execute** button, an HTTP session will be created, and the request will be fired.

Click on the session on the left-hand side pane (as shown in the following screenshot) and select the **Inspectors** tab in the right-hand side pane. You can view the result in the **JSON** tab in the bottom right-hand side pane:
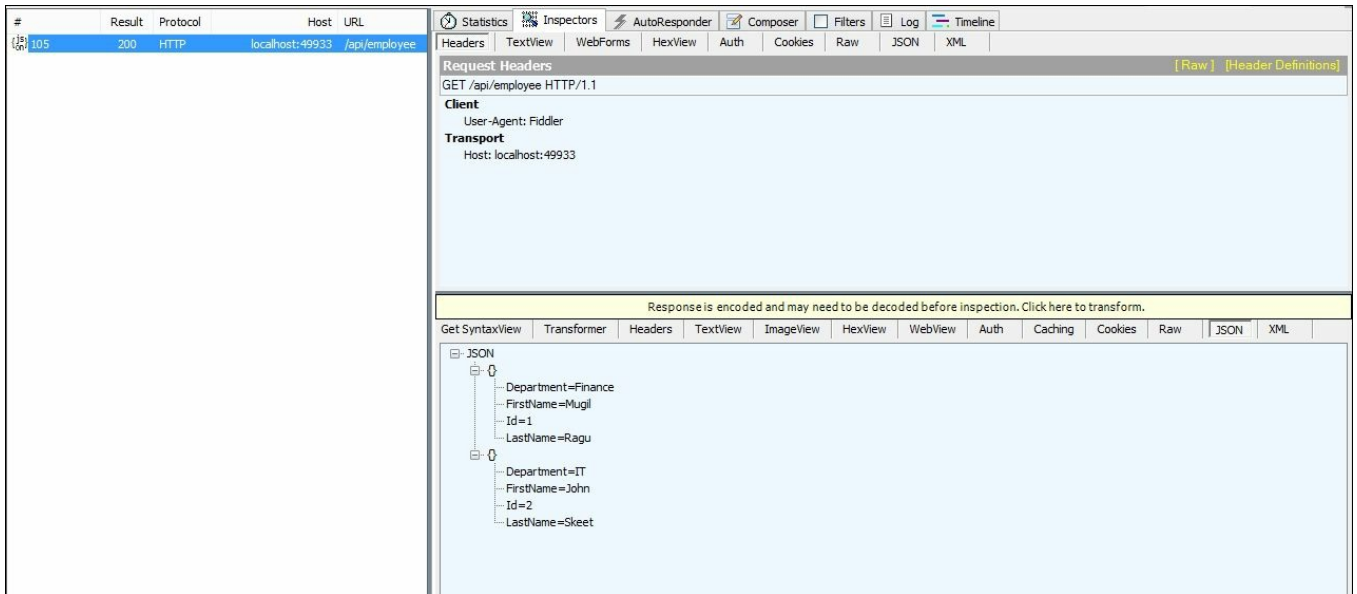
Let us fire another HTTP request to get a particular employee's information, say the employee whose ID is 2. We would construct the URL by appending the ID `http://localhost:49933/api/employee/2` as following:
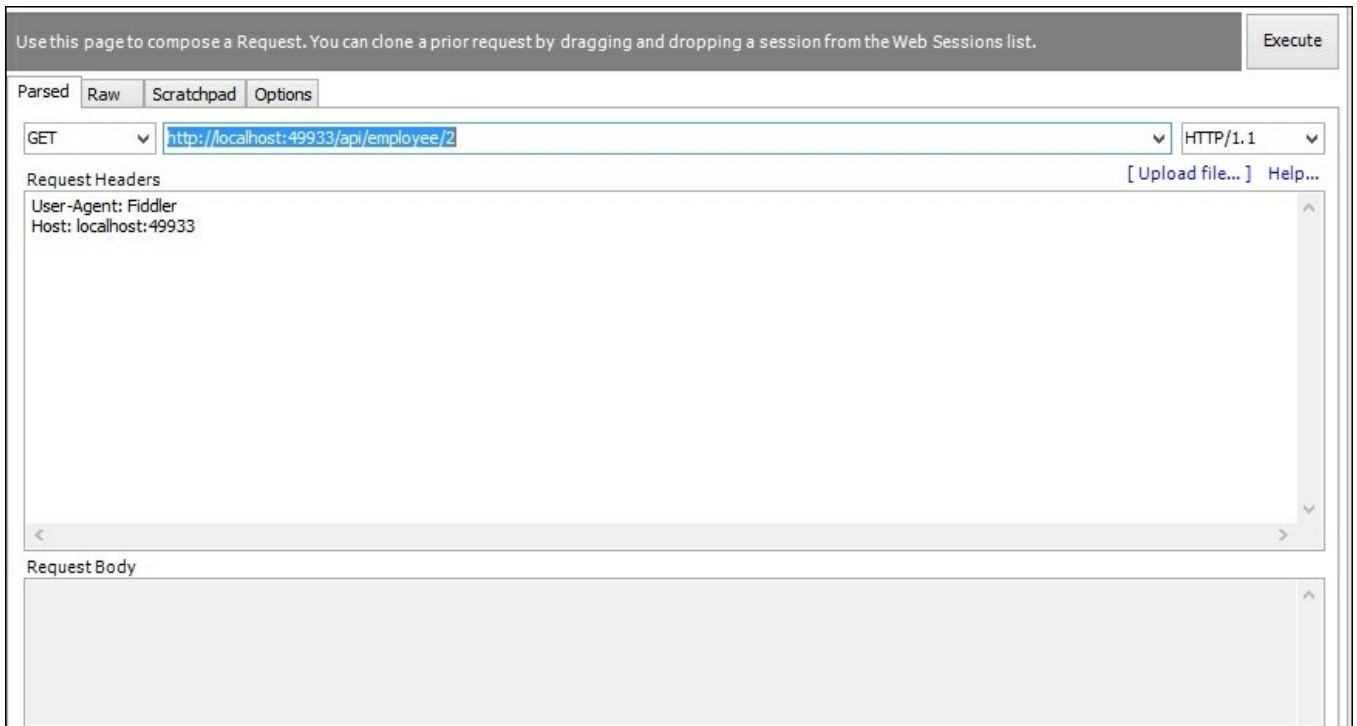
Select the recently created HTTP session and click on it:

You can see the result in JSON format in the right-hand side pane.

Now, we are going to add `Create`, `Update`, and `Delete` operations to our service. To start with, we are going to provide the Create functionality to add employees' to our service:

```
[HttpPost]
public IActionResult Add([FromBody] Employee emp)
{
  if (emp == null)
  {
    return BadRequest();
  }
  employeeRepository.AddEmployee(emp);
  return CreatedAtRoute("GetEmployee", new { id = emp.Id }, emp);
}
```

The following points should be considered when following the preceding `Add` method:

1. We are passing the `Employee` object as a parameter. We are instructing the `Add` method to take that object from the body of the request by specifying a `[FromBody]` attribute:
   - If no employee object is passed, we would be returning the bad request to the calling client
   - If it is not null, we would be calling the repository method to add the employee to our list (in the real world, we would be adding it to the database)
2. Once we have added the employee, we are returning the *201 status code* (as per the HTTP standards) when a new resource is created.

Open the Fiddler application and follow these steps to add the employee:

1. Select the HTTP method as `POST` and enter the URL `http://localhost:54504/api/employee/`.
2. You need to specify the content type as `application/json` in the request header. Please see the following screenshot, where we have added `Content-Type: application/json` to the request header.
3. As mentioned in the code, we have to pass the employee object in the form of JSON in the body of the request. In the following request, we have formed a JSON that contains the properties of the `Employee` object with the values in the brackets { "FirstName" : "James", "LastName" : "Anderson","Department" : "IT"}:
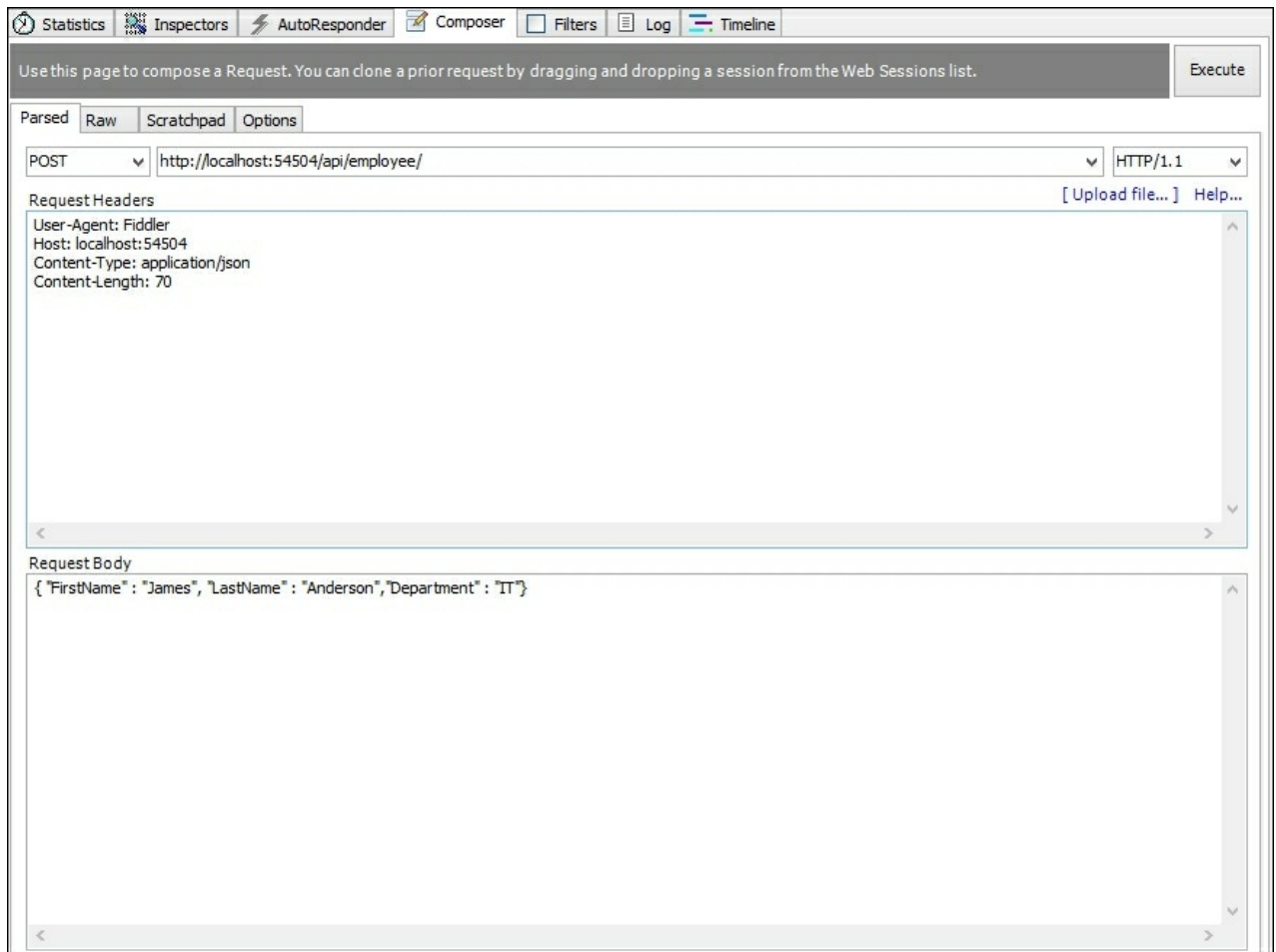
Once you have composed the request, you can click the **Execute** button to fire the request. This will return the *201 HTTP status code*, which is the standard HTTP response for creating a new resource:

As soon as we have created the resource in the server, we are redirecting the response to get the newly created resource. This occurs when we call the CreatedAtRoute method with the newly created employee ID passed as a parameter.

Click on the session on the left-hand side and select the **Inspector** tab in the right-hand side pane. Now you can see the response of the request. The response contains

the `Employee` object which was newly created in the server. We have to note that the ID of the `Employee` object is generated at the server, and is available in the following response. In this case, the ID generated for the employee is `1771082655`:
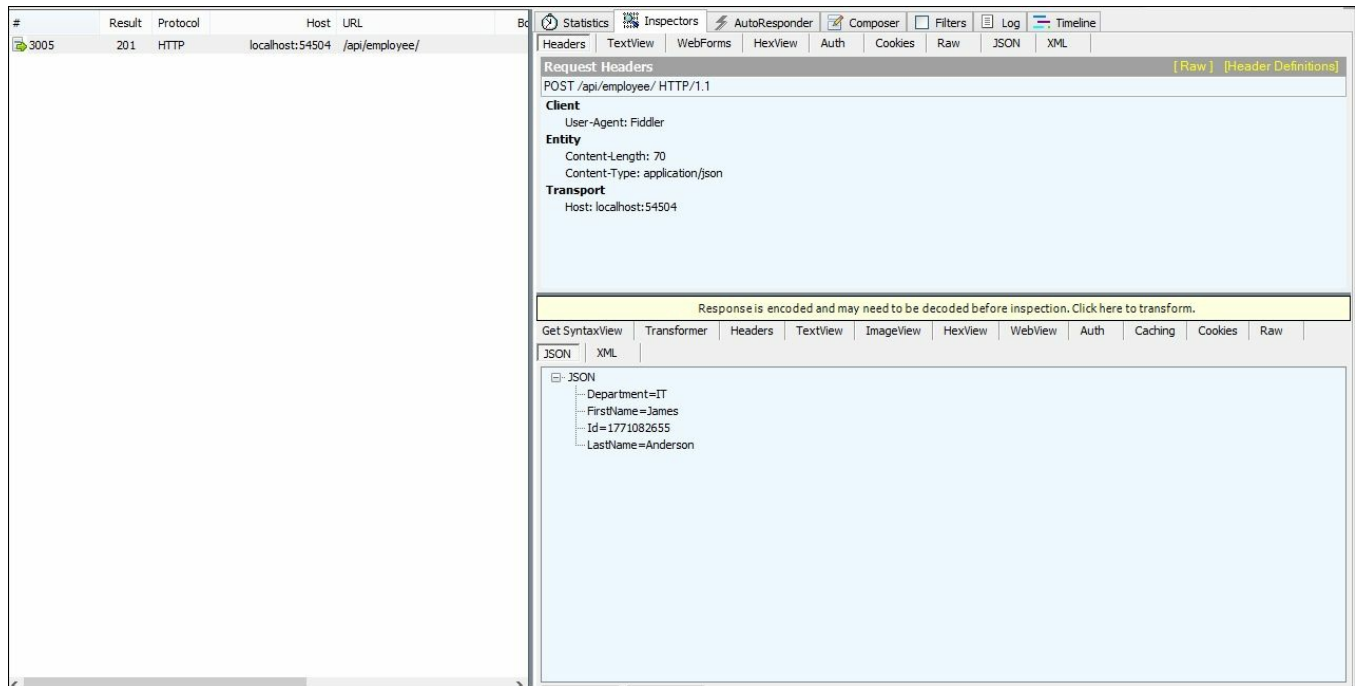
In the bottom right-hand side panel in the preceding Fiddler window, we can see the complete JSON response of the newly created resource.

Now we are going to add a Web API method to update the resource. The method for updating the resource is very similar to that used to create the resource, with only a few differences. When we created the resource, we used the `HTTP POST` method, whereas when we updated the resource, we used the `HTTP PUT` method.

If the passed employee ID could not be found in the repository, we return a *404 error* response, the HTTP standard error response for a resource that has not been found.

The following is the Web API controller method code for updating the resource:

```
[HttpPut]
public IActionResult Update([FromBody] Employee emp)
{
  if( emp == null)
  {
    return BadRequest();
  }
  Employee employee = employeeRepository.GetEmployee(emp.Id);
  if(employee == null)
  {
```

```
    return NotFound();
  }
  employeeRepository.UpdateEmployee(emp);
  return new NoContentResult();
}
```

The following is the repository layer code for updating the employee:

```
public void UpdateEmployee(Employee emp)
{
  Employee employee = employees.Where(e => e.Id ==
emp.Id).FirstOrDefault();
  if (employee != null)
  {
    employee.Department = emp.Department;
    employee.FirstName = emp.FirstName;
    employee.LastName = emp.LastName;
  }
}
```

Open the Fiddler application, and compose a request of HTTP PUT. As we are going to pass the Employee object in the body of the request, we need to mention the content type as application/json. In the body of the request, we need to supply the Employee object in JSON format, as shown in the following screenshot:
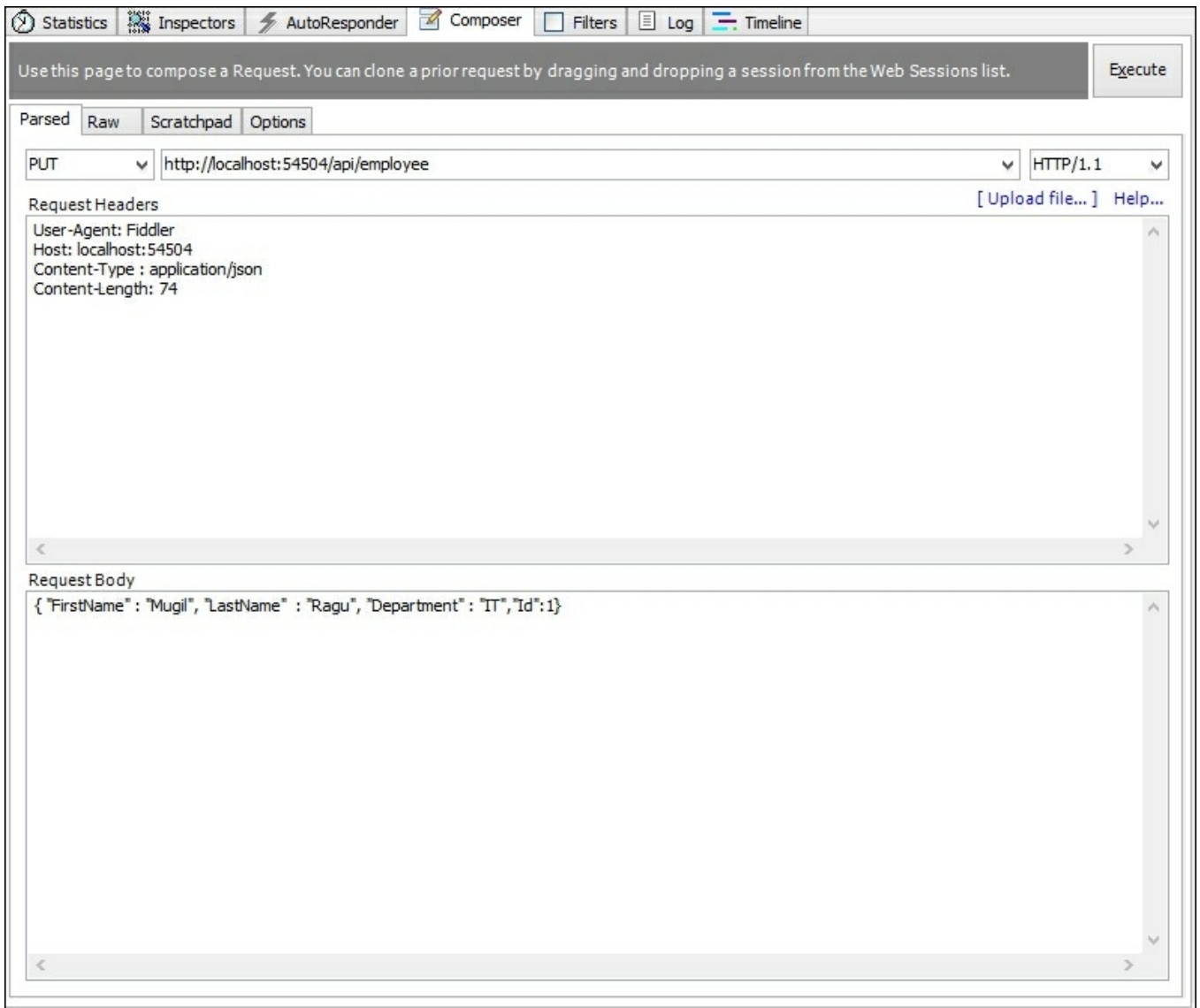
When you click the **Execute** button, the `HTTP PUT` request will be fired and our Web API method will get called. Once it succeeds, the *HTTP 204* response will be returned:

Delete method

The `HTTP DELETE` method should be used when deleting a resource. There is no need to pass anything in the body of the request.

The Web API method for deleting a resource

The `Delete` Web API method has a `void` return type, which will return an *HTTP 200* response:

```
[HttpDelete("{id}")]
public void Delete(int id)
{
  employeeRepository.RemoveEmployee(id);
}
```

Web Repository layer code for deleting the employee data

In the following repository layer method, we are removing the employee (whose ID matches with that of the parameter passed) from the internal list of employees. But in the real world, we would be interacting with the database to delete that particular employee. Consider the following code:

```
public Employee RemoveEmployee(int id)
{
  Employee employee = employees.Where(emp => emp.Id ==
id).FirstOrDefault();
  if(employee != null)
  {
    employees.Remove(employee);
  }
  return employee;
}
```

Open the Fiddler application, select the `DELETE` HTTP method, pass the URL with the parameter, and click on the **Execute** button. Please note that we are not passing the content type in the request header as we are not passing any employee object in the body of the request:
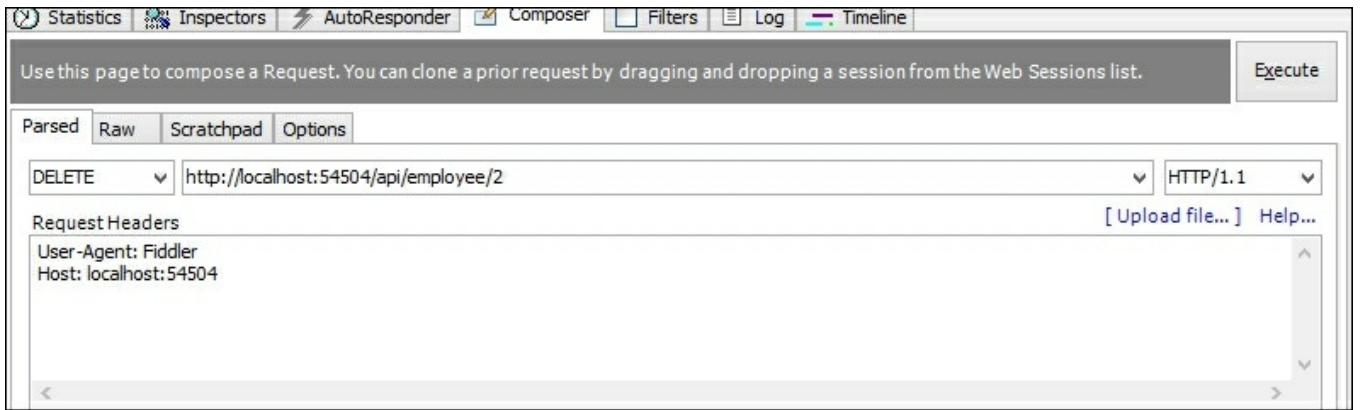
As we are returning void, the Web API `DELETE` method returns an *HTTP 200* status, as you can see in the left-hand side pane of the Fiddler application:

Summary

In this chapter, you learned about the HTTP service and its purpose. We discussed how to design and implement the HTTP service using Web API. We used the Fiddler tool to construct the HTTP request and get the response back. We also learned how to write the Web API method to perform CRUD operations end to end, from writing the Web API methods to firing the requests and getting the responses back.

Chapter 11. Improving Performance of an ASP.NET Core Application

When you think about frequently accessed applications (the ones that we use daily), such as Google, YouTube, and Facebook, it is the performance of these applications that distinguishes them from similar applications. Think for a moment. If Google took more than 10 seconds to provide search results, most people would switch over to Bing or some other search engine. So, performance is one of the primary factors in an application's success.

In this chapter, we are going to learn about the following things:

- The approach to analyzing the performance issues of an application
- How to make use of browser developer tools to analyze the performance of an application
- Performance improvements in the UI layer
- Performance improvements in the web/application layer
- Performance improvements in the database layer

Normally, when people talk about the performance of an application, they think about the application's speed. Though speed contributes significantly to the performance of the application, we also need to consider maintainability, scalability, and reusability of the application.

A well-maintained code will be clear and have less technical debt, which in turn will increase the productivity of the developer. When we write code based on service-oriented architecture or micro services, our code will be more usable by others. This would also make our code scalable.

Normally, people think about the performance of the application when they have almost completed the development of the application and pilot users are complaining about the speed of the application. The right time to discuss performance is before the development of the application; we need to work with the product owners, business analysts, and actual users in order to arrive at a standard of an acceptable level of performance for the application. Then we design and code with this expected level of performance as our goal.

This also depends on the domain of the application. For example, a mission-critical healthcare application would demand great performance (they might expect responses in less than a second), whereas the performance of a back-office application may not demand so much. So, it is critical to understand the domain in

which we are working.

If you have been asked to tune the performance of an existing application, it is also important to understand the existing architecture of the application. With ASP.NET Core, you can build a simple CRUD application to a mission-critical application serving millions of users across the world. A large application might have many other components, such as a load balancer, separate caching servers, **Content Delivery Networks (CDN)**, an array of slave DB servers, and so on. So, when you analyze the performance of the application, first you need to study architecture, analyze each of the individual components involved, measure the performance of each of the components, and try to optimize them when the application does not suit your acceptable performance. The main thing is not to jump into performance improvement techniques without studying and analyzing the architecture of the application. If you are creating a new application, you can think about performance right from the start of the application's creation.

We will examine a typical web application setup, shown in the following screenshot. We will then analyze it and consider how to improve it:
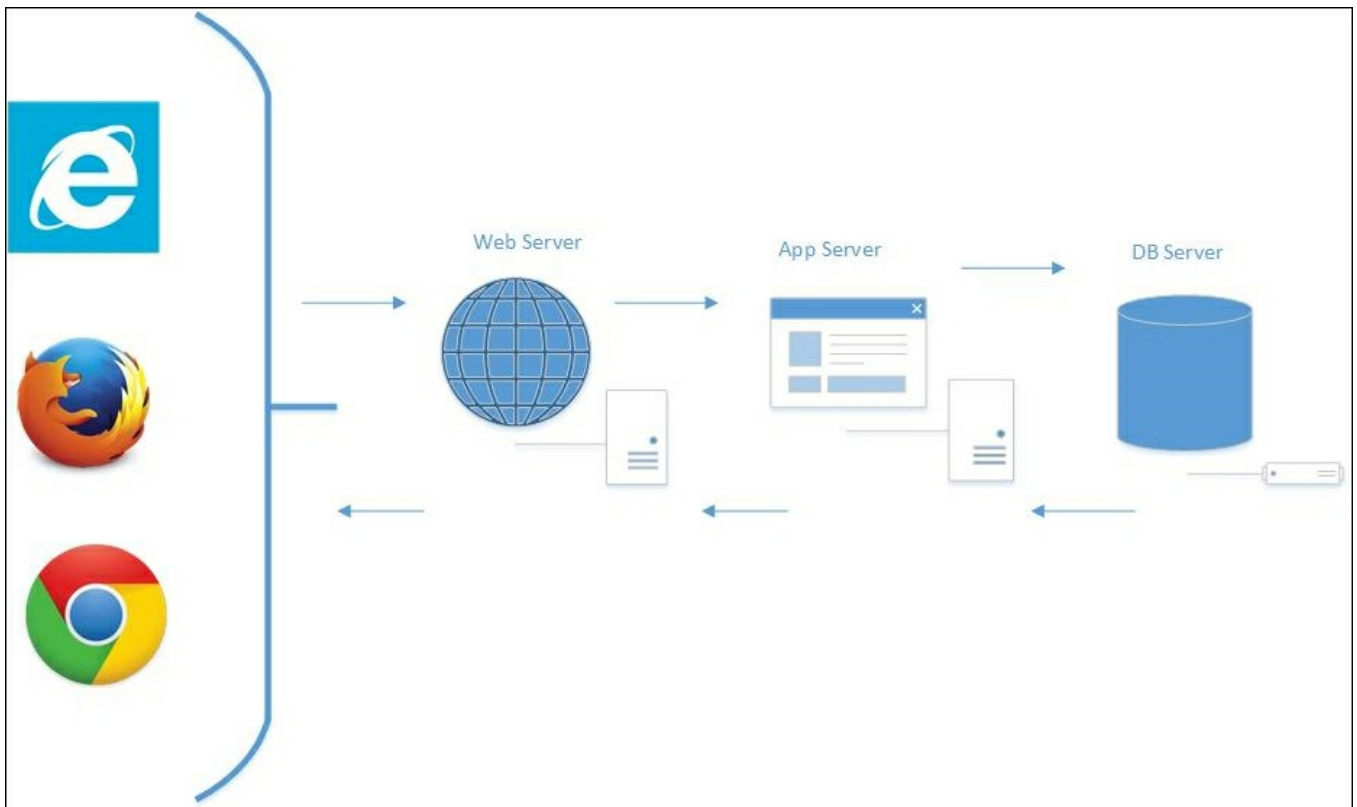
The following steps show the process of using a web application:

1. The user accesses an ASP.NET Core web application from a browser, such as

Internet Explorer, Firefox, or Chrome. When the user types the URL into the browser and presses the *Enter* key, the browser creates a session and fires the HTTP request. This is not specific to an ASP.NET Core application. This behavior is the same for all web applications, irrespective of the technology on which they are built.

2. The request reaches the web server. If it is a simple request, the web server itself will serve that request. Serving a static HTML file is a typical example of this. If the request is a bit complex, for example, returning some data based on the business logic, the request will be forwarded to the application server.

3. The application server will query the database to get the data. Then it might do some business processing on the received data before returning the data to the web server. Sometimes, the web server might act as an application server for a smaller web application.

4. Then, the web server will return the response, typically in HTML, to the requesting client.

Thus, we can categorize these components into three layers—the UI layer, the web/application layer, and the DB layer. With respect to improving the overall performance of the ASP.NET Core application, we need to have a thorough look at how we can improve the performance of each of the layers.

Before implementing any performance improvement techniques, we need to first analyze the performance in each of the layers in the application. Only then can we suggest ways improve the overall performance of the application.

The UI layer

The UI layer represents all the events (and associated stuff) happening between the browser and the server. There are many events, including, but not limited to, the following:

- Firing the HTTP request
- Getting the response
- Downloading the resources
- Rendering them in the browser
- Any JavaScript code execution

Reducing the number of HTTP requests

A typical web page might not have only HTML content. It may have references to CSS files, JS files, and images, or other sources. So, when you try to access a web page, the client will fire HTTP requests for each of these references and download those references from the server to the client.

Browser developer tools come in handy when you want to analyze the HTTP requests being fired from the client. Most of the browsers have developer tools that you can make use of.

When you press *F12* in Internet Explorer, the **Developer Tools** window will open at the bottom of the Internet Explorer window, as shown in the following screenshot:
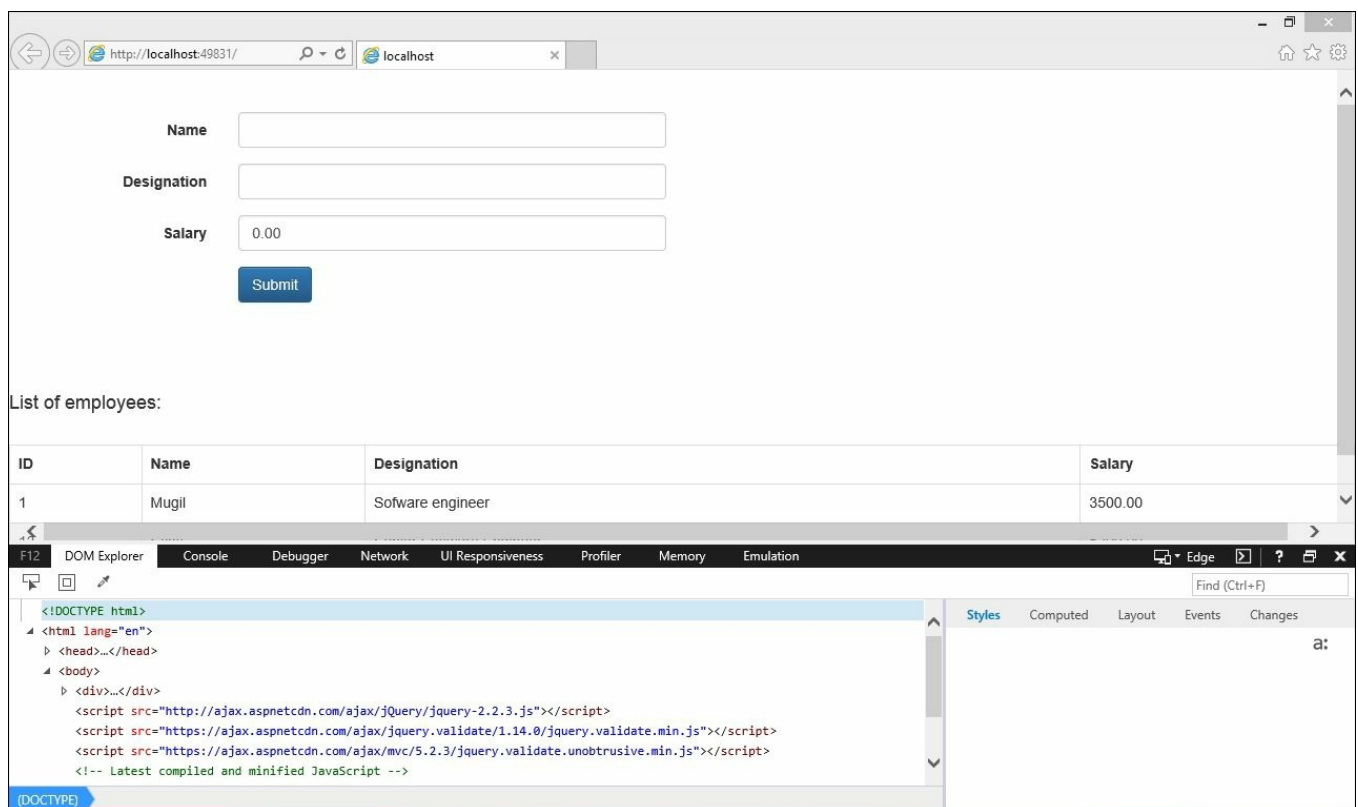
Click on the **Network** tab. Before entering the URL in the browser, click the *Start* button (the green play button), or click the green play button and refresh the page:
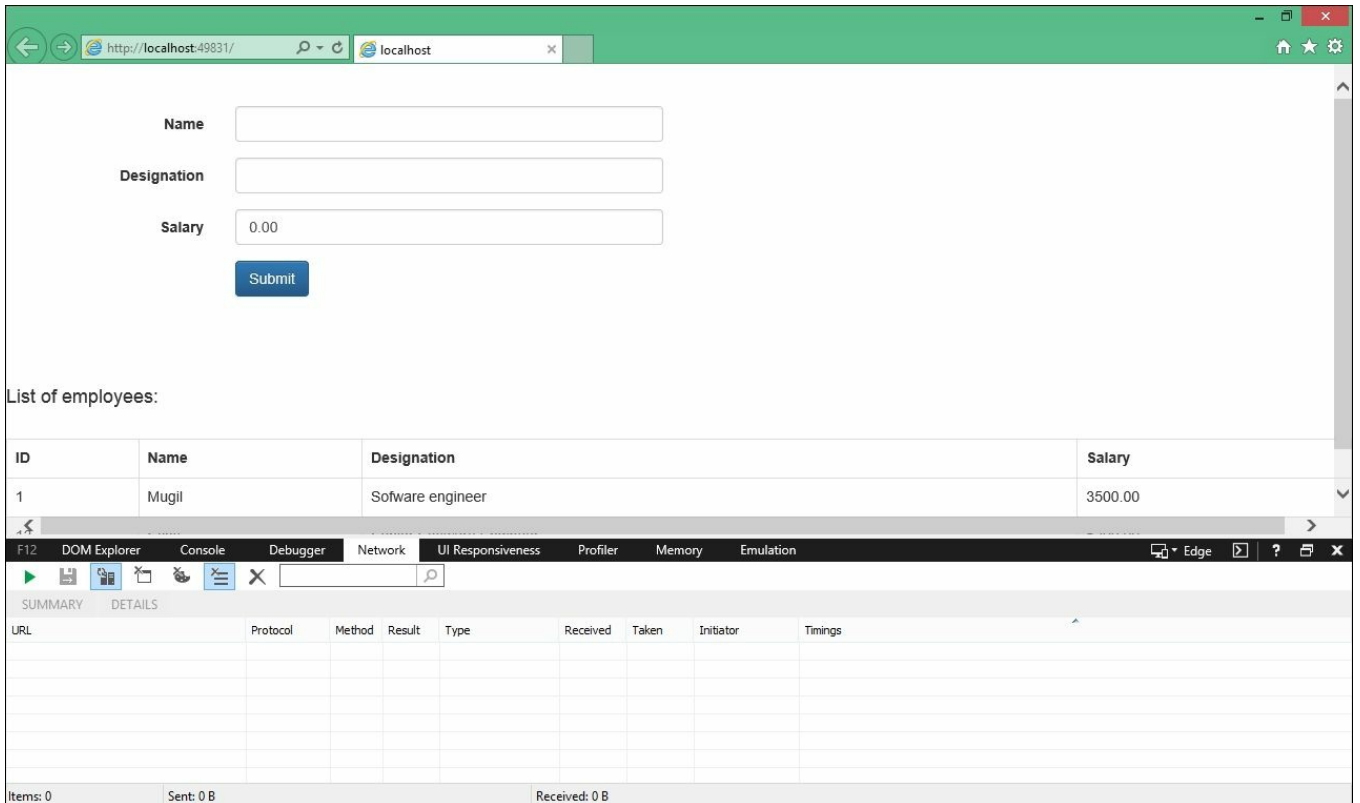
Once you press the **Network** tab's start button, Internet Explorer's **Network** tab will listen to each of the requests that are fired from the current tab. Each request will contain information, such as the URL, protocol, method, result (the HTTP status code), and other information.

I ran the application again with (**Tracking Network Requests** option ON) and I could see the requests being tracked, as shown in the following screenshot:

There are many useful pieces of data available in the Network tab. To begin with, the URL column shows the resource that is being accessed. The Protocol column, as the name implies, shows the protocol being used for accessing the resource.

To begin with, the URL column shows the resource that is being accessed. The Protocol column, as the name implies, shows the protocol being used for accessing the resource. The Method column shows the type of request, and in the Result column, we can see the HTTP status code of the request (HTTP 200 response means a successful GET request).

The Type column shows the type of resource that is being accessed, and the Taken column shows how much time it has taken to receive the file from the server. The Received column shows the size of the file that was downloaded as part of the request.

Using GZip compression

When you are serving the content, you can compress the content using GZip so that a smaller amount of data will be sent across the wire. You need to add the appropriate HTTP headers so that the browser can understand the mode of content being delivered. In IIS, this option is enabled for static resources by default. You can verify this by accessing the `applicationHost.config` file at the path `C:\Windows\System32\inetsrv\config`:

```
<httpCompression directory="%SystemDrive%\inetpub\temp\IIS
Temporary Compressed Files">
  <scheme name="gzip" dll="%Windir%\system32\inetsrv\gzip.dll" />
  <staticTypes>
    <add mimeType="text/*" enabled="true" />
    <add mimeType="message/*" enabled="true" />
    <add mimeType="application/x-javascript" enabled="true" />
    <add mimeType="application/atom+xml" enabled="true" />
    <add mimeType="application/xaml+xml" enabled="true" />
    <add mimeType="*/*" enabled="false" />
  </staticTypes>
</httpCompression>
```

If it is not available in your `applicationHost.config` file, you have to make the necessary changes.

Using the Content Delivery Network (CDN)

A Content Delivery Network is a system of distributed servers situated across the globe to serve the content based on the geographical location from where the content is accessed. Amazon's **CloudFront** is one example of a CDN. Amazon has edge locations (locations where servers are located) all over the world so that content can be served to users from the nearest location.

In the following line, we are accessing the jQuery from the CDN provided by the official jQuery website:

```
<script src="https://code.jquery.com/jquery-3.1.1.min.js" >
</script>
```

Using JavaScript wherever possible

If you can use JavaScript to achieve a functionality, then do it. For example, before validating the data of the form on the server, always try to do client-side validation first. This approach has a couple of advantages—the site will be very fast, as everything is done at the client-side itself, and the server would handle a larger number of requests, as some of the requests are handled on the client-side.
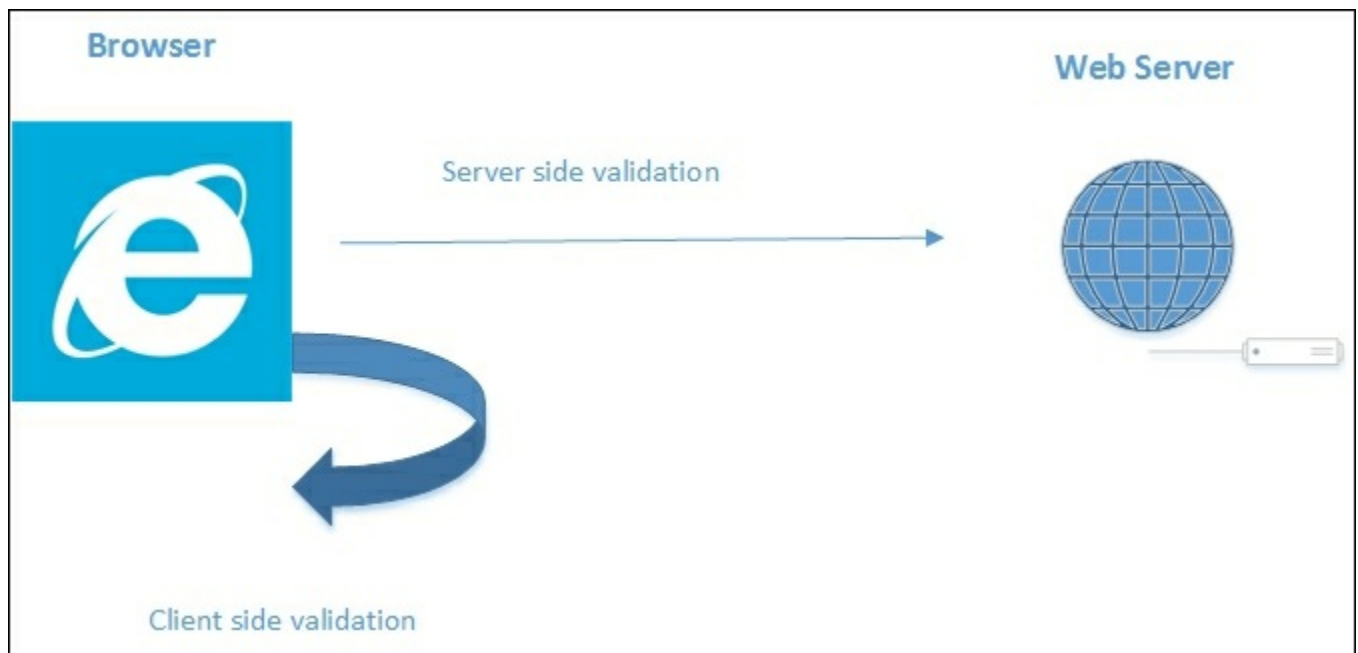
Using CSS stylesheets

As the browser renders the web page progressively (the browser will display whatever content it has, as soon as it receives it), it is better to place the stylesheets at the top rather than at the end of the web page. If we place the stylesheets at the bottom, it prohibits the progressive rendering as the browser has to redraw the content with the styles.

Most of the browsers will block parallel downloads when it comes to downloading the JavaScript files, so it is better to place the script at the bottom. This means that your content is shown to the user while the browser downloads the scripts. The following is the sample layout file created in an ASP.NET Core application where CSS files are referenced at the top and JavaScript files are referenced at the bottom:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>

    <!-- Latest compiled and minified CSS -->

    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap
.min.css" integrity="sha384-
1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">

    <!-- Optional theme -->

    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap
-theme.min.css" integrity="sha384-
fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">

  </head>
  <body>
    <div>
      @RenderBody()
    </div>
    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
2.2.3.js"></script>

    <script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.
validate.min.js"></script>
```

```html
    <script
src="https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unob
trusive.min.js"></script>

    <!-- Latest compiled and minified JavaScript -->
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.m
in.js" integrity="sha384-
0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>

  </body>
</html>
```

Minification of JavaScript and CSS files and their combination

The time taken to download the related resources of a web page is directly proportional to the size of the files that are downloaded. If we reduce the size of the file without changing the actual content, it will greatly increase the performance. Minification is the process of changing the content of the file in order to reduce the size of the file. Removing the extraneous white spaces and changing the variable names to shorter names are both common techniques used in the minification process.

Popular JavaScript libraries such as jQuery and frontend frameworks provide minified files by default. You can use them as they are. In the following screenshot, I have downloaded the compressed version of jQuery. You can minify the custom JavaScript and CSS files that you have written for your application:
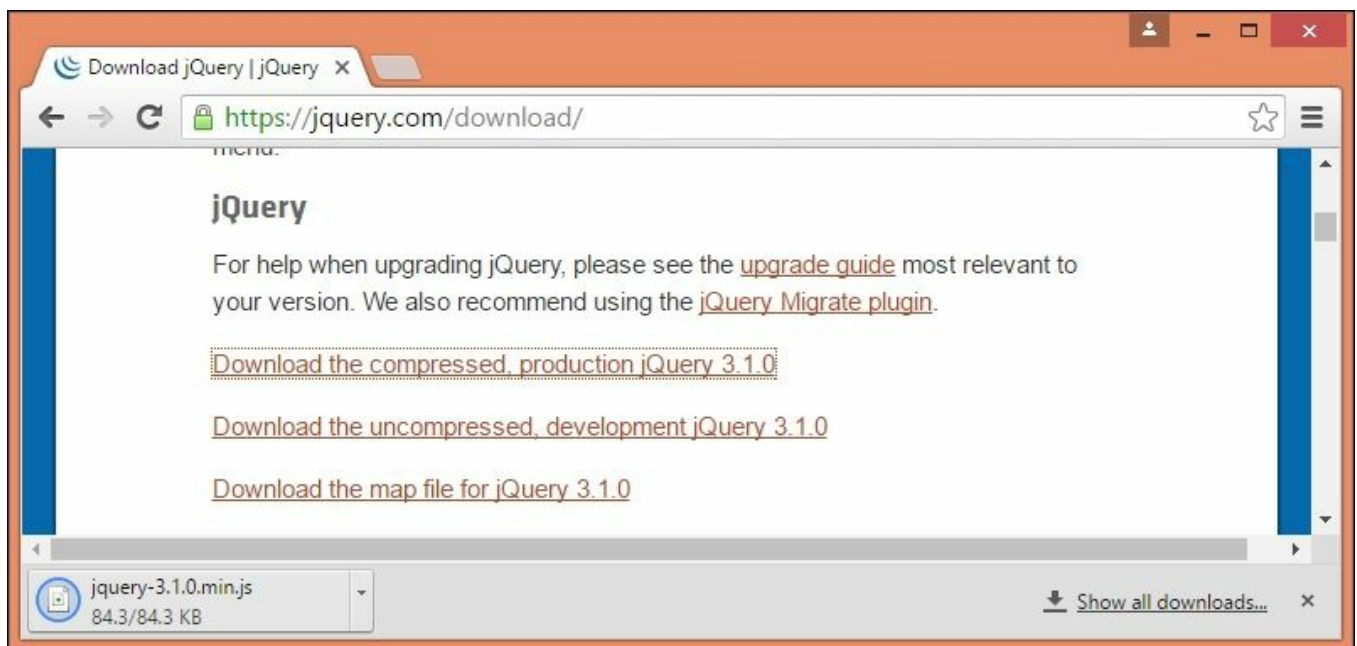
Bundling is the process where you can combine two or more files into one. Bundling and minification, when used together, will reduce the size of the payload, thereby increasing the performance of the application.

You can install the **Bundler & Minifier** Visual Studio extension from the following URL:

https://visualstudiogallery.msdn.microsoft.com/9ec27da7-e24b-4d56-8064-

[fd7e88ac1c40](fd7e88ac1c40)

Once you have installed this Visual Studio extension, you can select the files that you want to bundle and minify by selecting the files and selecting the **Bundler & Minifier** option from the **Context** menu, brought up by right-clicking. It is shown in the following screenshot:
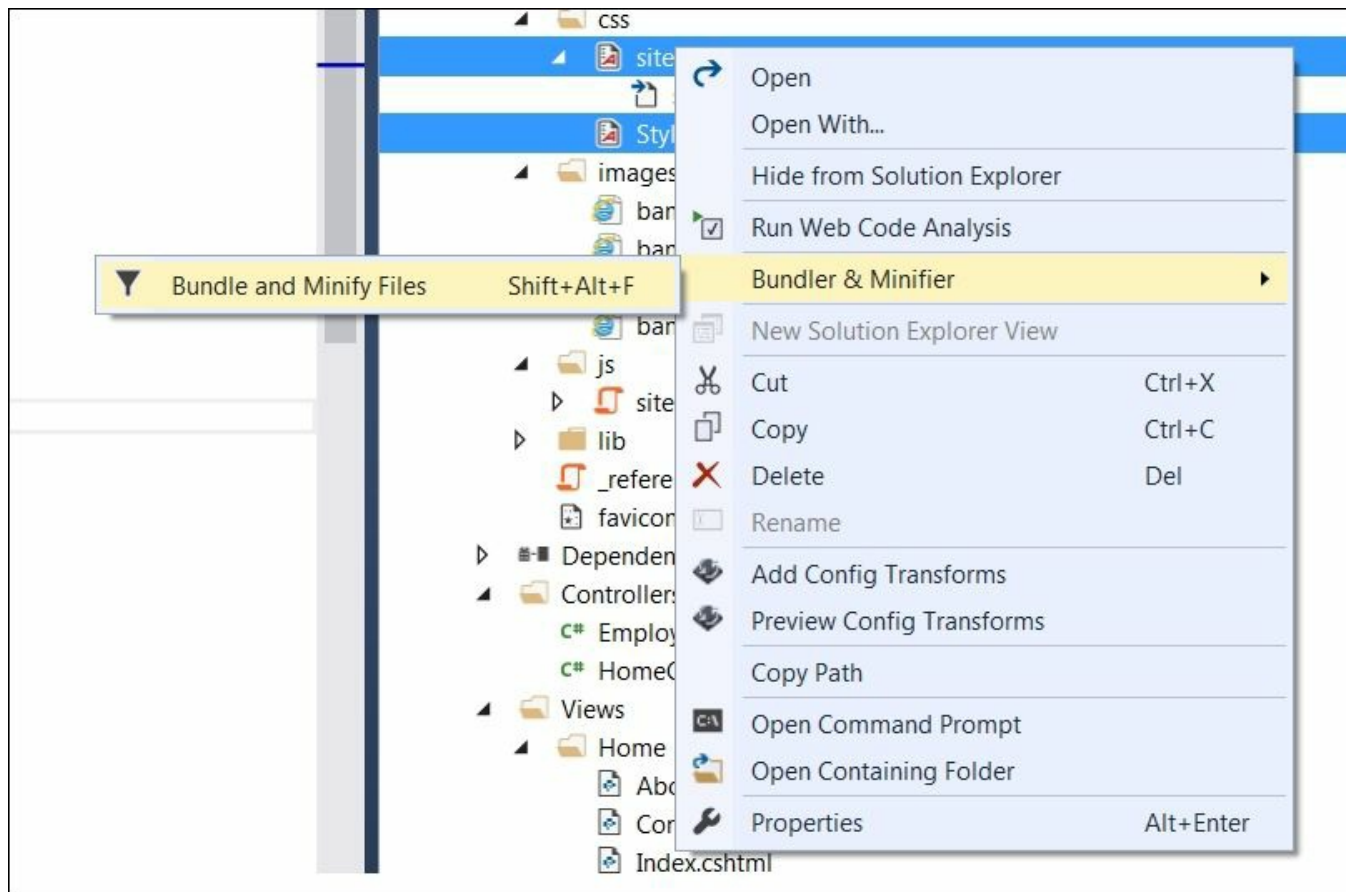
Once you select the **Bundle and Minify Files** option, it will ask you to save the bundled file as shown in the following screenshot:
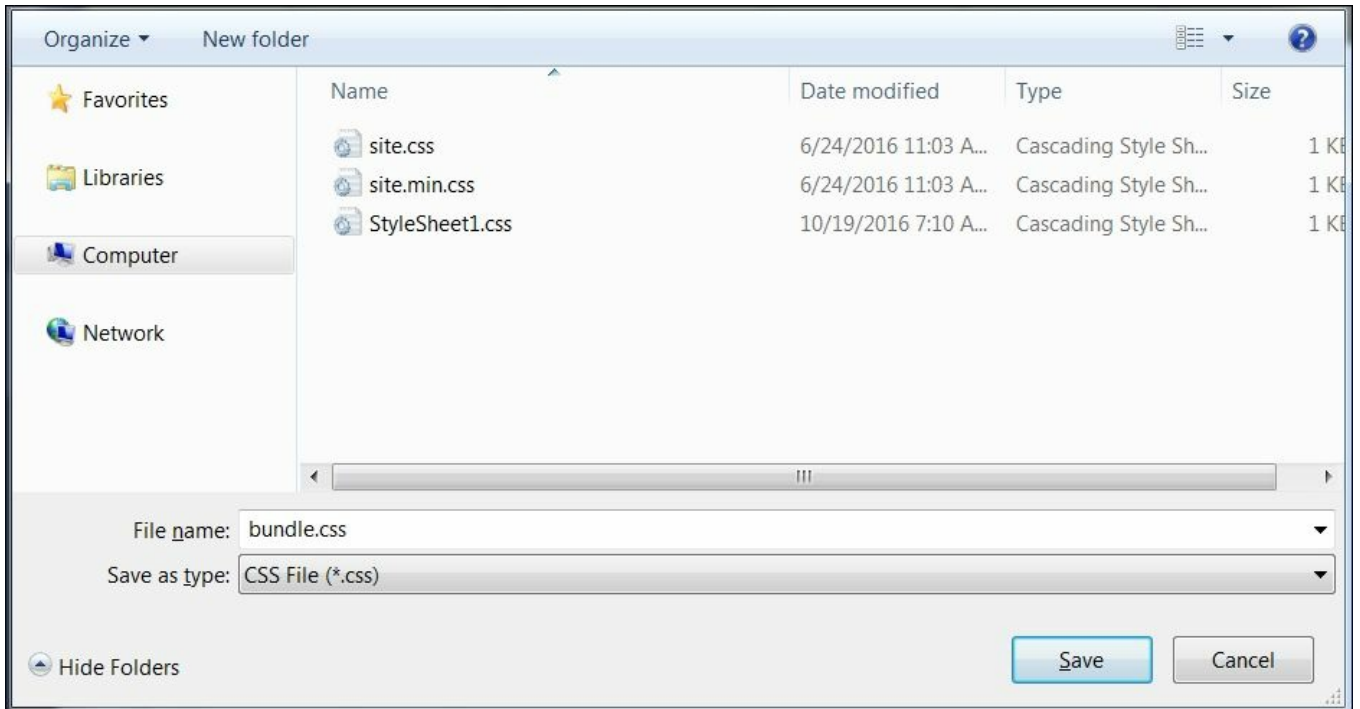
You can name the file of your wish and save the file. Once you save the file, another file would have been created in your solution—in our case, it is the `bundleconfig.json` file:
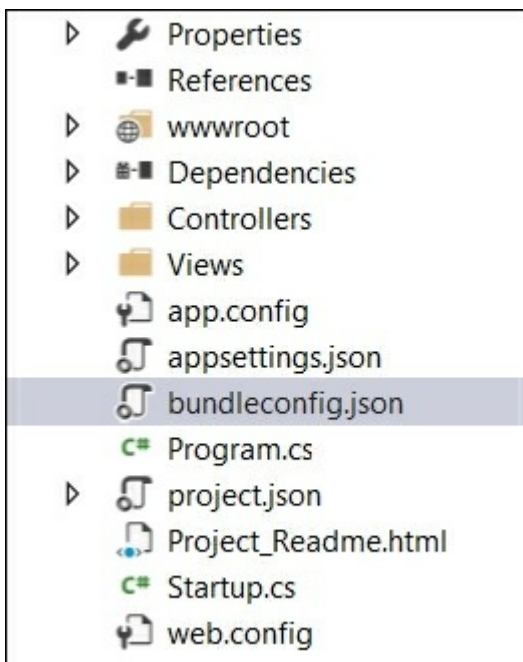
This file will have the information on the input files and the bundled output file. The following is one such example:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
    "wwwroot/css/site.css"
    ]
  },

{
  "outputFileName": "wwwroot/js/site.min.js",
  "inputFiles": [
  "wwwroot/js/site.js"
  ],
  "minify": {
  "enabled": true,
  "renameLocals": true
  }
},

{
  "outputFileName": "wwwroot/css/bundle.css",
  "inputFiles": [
  "wwwroot/css/site.css",
  "wwwroot/css/StyleSheet1.css"
    ]
  }
]
```

You can use this bundled file in your application, resulting in increased performance.

The caching process

Caching is the process of copying the data and having it in memory instead of getting the data again through an external resource, such as a network, file, or database. The data used in caching is ephemeral and can be removed at any time. As we are directly accessing the data, caching can greatly improve the performance of the application.

Caching can be done in any of the layers—client-side at the browser, at the proxy server (or at some middleware), or at the web/application server. For database layer caching, we might not need to do any custom coding. Based on the type of database server being used, you might need to make some configuration changes. However, most of the databases these days are powerful enough to cache the data as and when it is needed.

Client-side caching

We can cache at the client-side if we add the appropriate HTTP response headers. For example, if we want to cache all the static assets, such as CSS, images, and JavaScript files, we can add the **max-age** response header in the **Cache-Control** header:
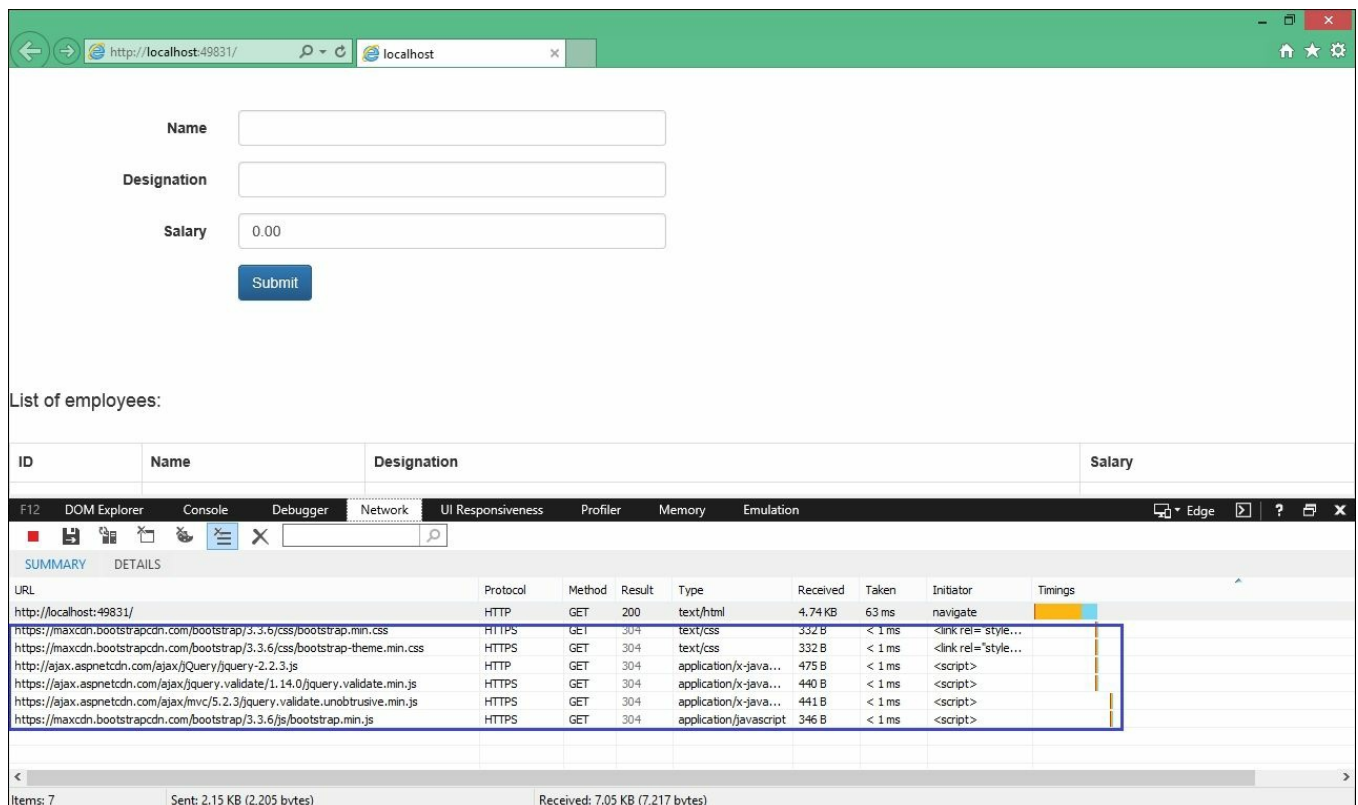
In the preceding screenshot of the **Developer Tool** window's **Network** tab, when the requests are fired again, we get *HTTP 304 response* (Not modified) as the response. This means the same files are not transferred back twice across the wire, as they are available in the browser itself.

Implementing browser caching for static files is pretty easy, and it involves just a couple of steps—adding dependencies and configuring the application.

Add the following `NuGet` package to the list of dependencies in the `project.json` file:

```
"Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
```

Add the following namespaces to the `Startup.cs` file and configure the application to use those static files:

```csharp
using Microsoft.AspNet.StaticFiles;
using Microsoft.Net.Http.Headers;

public void Configure(IApplicationBuilder app)
{
  app.UseIISPlatformHandler();
  app.UseMvc();
  app.UseMvc(routes =>
  {
    routes.MapRoute(name:"default", template:"
{controller=Employee}/{action=Index}/{id?}");});

    app.UseStaticFiles(new StaticFileOptions
()

    {
      OnPrepareResponse = (context) =>
    {

      var headers = context.Context.Response.GetTypedHeaders();

      headers.CacheControl = new CacheControlHeaderValue()

      {

        MaxAge = TimeSpan.FromSeconds(60),

      };

      }

  });
}
```

Response caching

In response caching, cache-related HTTP headers are added to HTTP responses when MVC actions are returned. The Cache-Control header is the primary HTTP header that gets added to the response.
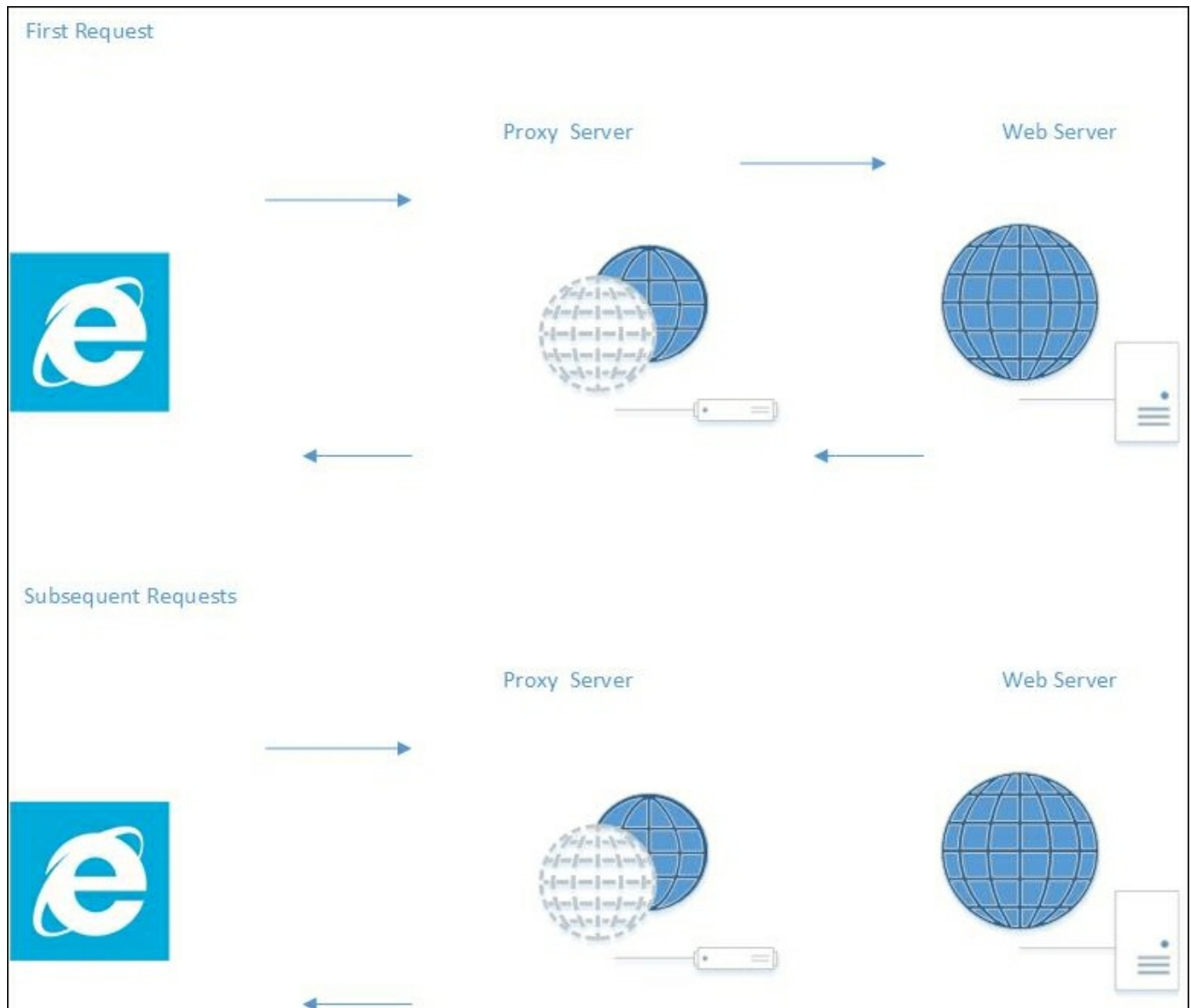
The preceding diagram shows response caching in action. In the first request, we are calling a Controller's action method; the request comes from the client and passes through the proxy server, actually hitting the web server. As we have added a response cache, any subsequent requests will not be forwarded to the web server, and the responses will be returned from the proxy server itself. This will reduce the number of requests to the web server, which in turn will reduce the load on the web server.

Caching the response of the Controller's action method is pretty easy. Just add the `ResponseCache` attribute with a duration parameter. In the following action method, we have added the response cache with a duration of 60 seconds, so that, for the next 60 seconds, if any requests come again, the responses will be returned from the proxy server itself instead of going to the web server:

```
[ResponseCache(Duration = 60)]
public IActionResult Index()
{
  EmployeeAddViewModel employeeAddViewModel = new
EmployeeAddViewModel();
  using (var db = new EmployeeDbContext())
  {
    employeeAddViewModel.EmployeesList = db.Employees.ToList();
  }
  return View(employeeAddViewModel);
}
```

The web/application layer

The web/application layer is composed of whatever happens between receiving the request from the client and sending back the response (or querying the DB layer to get the required data). Most of the web/application layer will be in a server-side language, such as C#, so when you try to optimize the web/application layer, you need to incorporate the best practices of ASP.NET MVC and C#.

No business logic in Views

A View is what is rendered to the browser, and it can contain presentation logic. Presentation logic represents where and how the data is to be displayed. ViewModels (actually, models specific to the View) are models that hold the data for a particular view.

Neither Views nor ViewModels should contain any business logic as this violates the separation of concerns principle.

Look at the following Razor View code. We are just looping through the list in the model and presenting the data in tabular format—nothing else:

```
<h4> List of employees:</h4> <br />
  <table class="table table-bordered">
  <tr>
    <th> ID </th>
    <th> Name </th>
    <th> Designation </th>
    <th> Salary </th>
  </tr>
  @foreach (var employee in Model.EmployeesList)
  {
  <tr>
    <td>@employee.EmployeeId</td>
    <td>@employee.Name</td>
    <td>@employee.Designation</td>
    <td>@employee.Salary</td>
  </tr>
  }
  </table>
```

In some code, there might be a repository layer in ViewModel, which should never be the case. Please be extra cautious about what is there in the View/ViewModel code.

Using asynchronous logging

Try to use asynchronous logging, wherever possible, to improve the performance. Most logging frameworks, such as **Log4Net**, provide an option for logging asynchronously. With respect to the ASP.NET Core, you can implement the logging through a Dependency Injection.

The following is a typical example of the implementation of a logging framework in an MVC Controller:

```
public class EmployeeController : Controller
{
  private readonly IEmployeeRepository _employeeRepo;
  private readonly ILogger<EmployeeController> _logger;
  public EmployeeController(IEmployeeRepository employeeRepository,
  ILogger<EmployeeController> logger)
  {
    _employeeRepo = employeeRepository;
    _logger = logger;
  }
  [HttpGet]
  public IEnumerable<Employee> GetAll()
  {
    _logger.LogInformation(LoggingEvents.LIST_ITEMS, "Listing all
employees");
    return _employeeRepo.GetAll();
  }
}
```

The DB layer

Though the DB layer is not directly related to ASP.NET Core applications, it is the developer's responsibility to take complete ownership of the application's performance, and that includes taking care of the database's performance as well. We will now look at a few of the areas in the DB layer that we need to consider when improving the performance of an ASP.NET Core application.

Understanding the queries generated by the ORM

In most applications these days, we use **Object-Relational Mapping** (**ORM**), such as Entity Framework or **NHibernate**. As you might know, the primary objective of the ORM is to enable you to write the data access layer using domain-based classes and objects instead of writing queries directly. However, it does not mean that you never need to understand the basics of the SQL queries generated, or the optimization of these queries. Sometimes, the generated query from Entity Framework may not be optimized, so a better practice would be to run the profiler, analyze the generated queries, and tune them as per your needs. You can use the interceptors in Entity Framework to log the SQL queries.

Using classic ADO.NET if you really want to

ASP.NET Core is just a web development framework, and it is not tied to any data access framework or technology. If the ORM that you use in your application does not support the performance that you expect it to, you can use the classic ADO.NET and manually write the queries/stored procedures.

Return only the required data

Always return only the data that you need nothing more, nothing less. This approach reduces the data that we send across the wire (from the database server to the web/application server).

For example, we would not use the following:

```
Select * from employees
```

Instead, we would use this:

```
Select FirstName,LastName from employees
```

The latter query would get only the required fields from the table, and, thus, only the required data is passed across to the calling client.

Fine tuning the indices

Beginners tend to add indices whenever they face a problem with the database. Adding an index to every column in the table is bad practice, and will reduce performance. The right approach is to take the list of queries that are most frequently executed. Once you have this list, try to fine tune them—remove unnecessary joins, avoid correlated subqueries, and so on. Only when you have tried and exhausted all query tuning options at your end should you start adding the indices. The important thing to note here is that you should add indices only on the required number of columns.

Using the correct column type and size for your database columns

When you want to use int as a datatype for a column, use an integer. Don't use double. This will save a lot of space if you have lots of rows in your table.

Avoiding correlated subqueries

Correlated subqueries use values from their parent query, which in turn makes it run row by row. This would significantly affect the query performance.

The following is one such example of a correlated subquery:

```
SELECT e.Name,
e.City,
(SELECT DepartmentName FROM EmployeeDepartment WHERE ID =
e.DepartmentId)
AS DepartmentName
FROM Employee e
```

Generic performance improvement tips

Here are a couple of pointers to improve the overall application performance in an ASP.NET Core Web Application.

Avoiding the Response.Redirect method

When we want to do client-side redirection, developers can call the `Response.Redirect` method with the URL passed as a parameter. But there is a small problem with this approach. If we use `Response.Redirect`, the browser will send the request to the server again, which needs another round trip to the server. So, if possible, it is better to avoid the `Response.Redirect` method and instead use `RedirectToAction` method if possible.

Using string builder

If your application involves a lot of string manipulation, it is preferable to use string builder instead of the usual string concatenation. String concatenation results in creating a new string object for each of the operations, whereas string builder works on the single object itself. We can achieve significantly better performance when we use string builder in large string manipulation operations.

Summary

In this chapter, we have learned how to analyze the performance of web applications and which layers to target when improving the performance. Then we discussed how to improve the performance in each of the layers—the UI layer, the web/application layer, and the DB layer.

Chapter 12. ASP.NET Core Identity

Security is essential to all types of applications, including web applications. Would you use Facebook if anyone could update your status by impersonating you? If that were possible, then no one would come back to Facebook. From this example, we can see that security is not so much a feature as it is a necessity for all applications.

In this chapter, we are going to learn about the following topics:

- Authentication and authorization
- ASP.NET Identity
- How to implement security in an ASP.NET Core application using ASP.NET Identity with Entity Framework

When we talk about the security of an application, we primarily want to prevent any unauthorized access, meaning that only the people who have access to the information should be able to access it—nothing more, nothing less.

Before proceeding further, I would like to clarify some of the core concepts regarding security.

Authentication

Authentication is the process of validating whether the user has access to the system. In any application, users will be authenticated first. This can be achieved by asking the user to enter their user ID and password.

Authorization

Authorization is the process where we verify whether the user has access to the requested resource. They might have legitimate access to the system, but they might not have access to the requested resource as they do not have the required access. For example, only the admin user can access the configuration page of the application, whereas normal users should not be allowed to use this page.

ASP.NET Identity provides several features for securing the application.

Let us consider the following simple scenario where the user tries to access the **Secure Page**, a page to which only authorized people should have access. As the user is not logged in, they will be redirected to the **Login Page** so that we can authenticate and authorize the user. Upon successful authentication, the user is redirected to the **Secure Page**. If for any reason, we can not authenticate and authorize the user, we can redirect them to the **"Access denied" Page**:
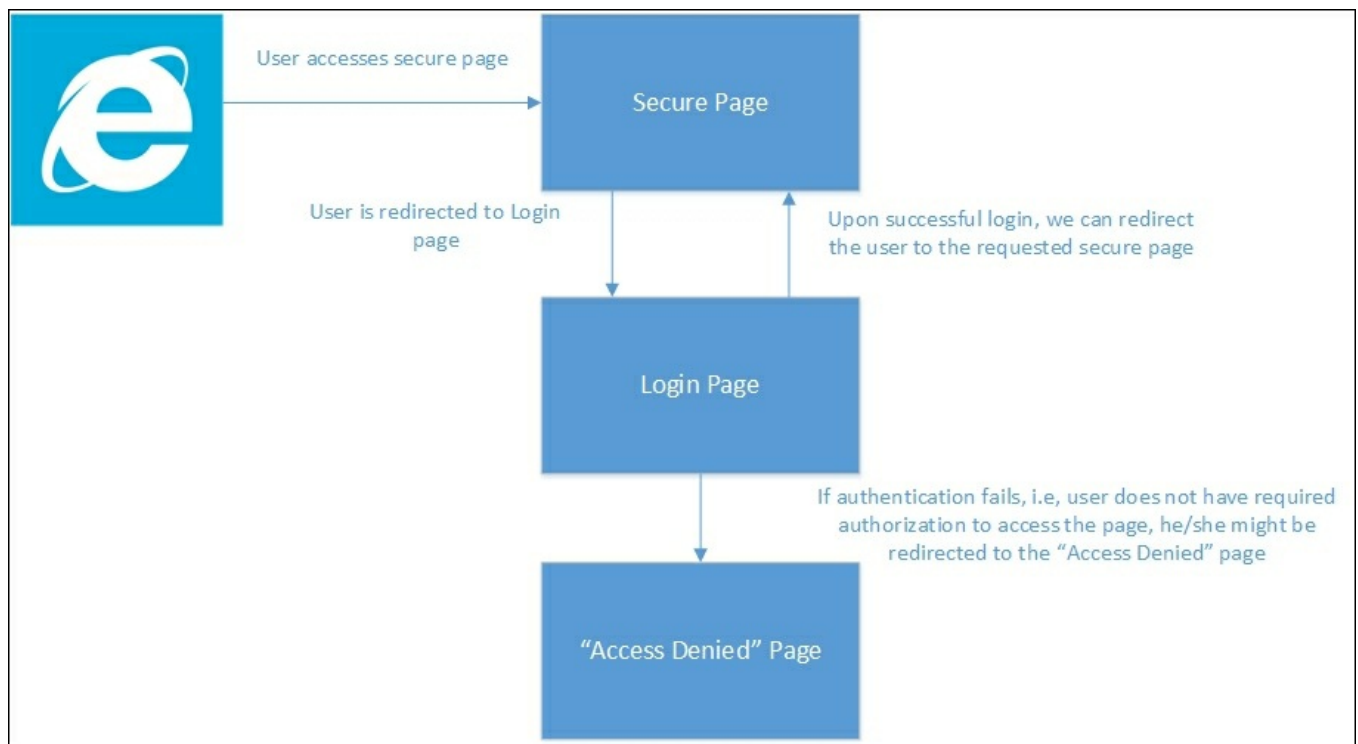
ASP.NET Core Identity is a membership system that enables you to secure the application easily, and which has features such as adding login functionality to your application. The following are the steps that we need to follow in order to use **ASP.NET Identity** (with Entity Framework) for our application:

1. Add the relevant dependencies to the `project.json` file.

2. Create an `appsettings.json` file and store the database connection string.
3. Create an `ApplicationUser` class and `ApplicationDbContext` class.
4. Configure the application to use ASP.NET Identity.
5. Create ViewModels for registration and login.
6. Create the necessary controller and associated action methods and Views.

Adding the relevant dependencies to the project.json file

If you want to use ASP.NET Identity with Entity Framework in your application, you need to add the following dependencies:

```
"EntityFramework.Commands": "7.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "Microsoft.AspNet.Authentication.Cookies": "1.0.0-rc1-final",
```

Create an `appsettings.json` file and store the database connection string.

Create a file with the name `appsettings.json` at the root level of the project, as shown in the following screenshot:
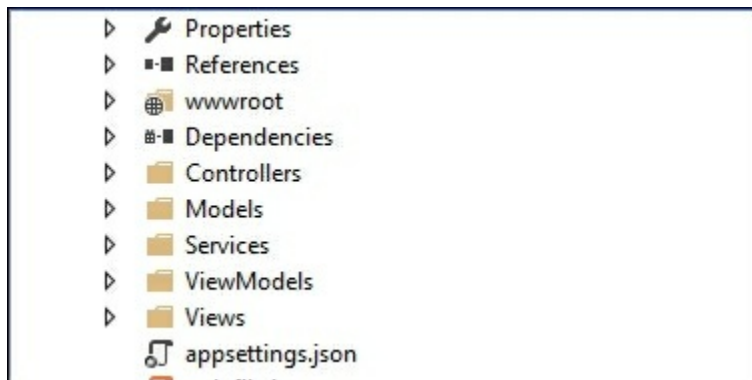
Store the following connection string in `appsettings.json`. This connection string will be used by ASP.NET Identity to store the data in relevant tables:

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Server=
(localdb)\\mssqllocaldb;Database=aspnet_security;Trusted_Connection
=True;MultipleActiveResultSets=true"
    }
  }
}
```

Adding ApplicationUser and ApplicationDbContext classes

Create a **Models** folder and a couple of files—**ApplicationDbContext.cs** and **ApplicationUser.cs**—as shown in the following screenshot:
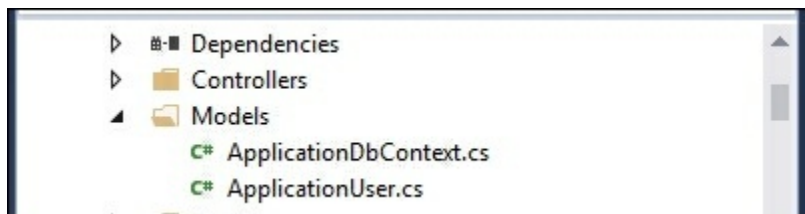
The `ApplicationUser` class inherits from the `IdentityUser` class (available at the `AspNet.Identity.EntityFramework6` namespace) as follows:

```
public class ApplicationUser : IdentityUser
{
..
}
```

You can add properties to the user as per the needs of your application. I have not added any properties as I would like to keep things simple to show the features of ASP.NET Identity.

The `ApplicationDbContext` class inherits from the `IdentityDbContext` class of `ApplicationUser`. In the constructor method, we pass the `connectionstring`, which is eventually passed to the base class.

Even the `OnModelCreating` method is overridden. If you want to change any table names (to be generated by Identity), you can do so as follows:

```
public class ApplicationDbContext :
IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(string nameOrConnectionString)
: base(nameOrConnectionString) { }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
        }
    }
```

Once we create the `Models` file, we need to configure the application and services. You can configure these in `Configure` and `ConfigureServices`, which are found in the `Startup` class.

Configuring the application to use Identity

In order to use Identity, we just need to add the following line in the `Configure` method of the `Startup` class:

```
app.UseIdentity();
```

The complete `Configure` method is shown in the following code, along with the call of the `UseIdentity` method, which is `app.UseIdentity()`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env,  ILoggerFactory loggerFactory)
        {

loggerFactory.AddConsole(Configuration.GetSection("Logging"));
            loggerFactory.AddDebug();

            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
                app.UseDatabaseErrorPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");



            app.UseIISPlatformHandler(options =>
options.AuthenticationDescriptions.Clear());

            app.UseStaticFiles();

            app.UseIdentity();

            // To configure external authentication please see
http://go.microsoft.com/fwlink/?LinkID=532715

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

In the `ConfigureServices` method, we will make the following changes:

- We will add the `ApplicationDbContext` class with the connection string taken from the `appsettings.json` file
- We will add Identity with `UserStore` and `RoleStore`
- Finally, we will ask ASP.NET Core to return `AuthMessageSender` whenever we ask for the `IEmailSender` and `ISMSSender` classes

```
public void ConfigureServices(IServiceCollection services
{
// Add framework services.

            services.AddScoped<ApplicationDbContext>(f => {
                return new
ApplicationDbContext(Configuration["Data:DefaultConnection:Conn
ectionString"]);
            });

            services.AddIdentity<ApplicationUser, IdentityRole>
()
                .AddUserStore<UserStore<ApplicationUser,
ApplicationDbContext>>()
                .AddRoleStore<RoleStore<ApplicationDbContext>>
()
                .AddDefaultTokenProviders();

            services.AddMvc();

            // Add application services.
            services.AddTransient<IEmailSender,
AuthMessageSender>();
            services.AddTransient<ISmsSender,
AuthMessageSender>();
        }
```

Creating ViewModels

Next, we will be creating several ViewModels that we will be using in our Views model.

To start with, we will create a `RegisterViewModel` class that contains three properties—`Email`, `Password`, and `ConfirmPassword`. We decorate the properties with appropriate attributes so that we can use client-side validation using an unobtrusive jQuery validation. We are making all the fields required as follows:

```
public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least
{2} characters long.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password", ErrorMessage = "The password and
confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }
```

Now, we can create the `LoginViewModel` model, which the user can use to log in to your application. There is an additional property, `RememberMe`, which, when checked, will enable you to log in without having to enter the password again:

```
public class LoginViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Display(Name = "Remember me?")]
        public bool RememberMe { get; set; }
    }
```

Creating Controllers and associated action methods

Now we need to create an `AccountController` class, where we will define the action methods for authentication and authorization:

```
public class AccountController : Controller
    {
        private readonly UserManager<ApplicationUser> _userManager;
        private readonly SignInManager<ApplicationUser>
_signInManager;
        private readonly IEmailSender _emailSender;
        private readonly ISmsSender _smsSender;
        private readonly ILogger _logger;

        public AccountController(
            UserManager<ApplicationUser> userManager,
            SignInManager<ApplicationUser> signInManager,
            IEmailSender emailSender,
            ISmsSender smsSender,
            ILoggerFactory loggerFactory)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _emailSender = emailSender;
            _smsSender = smsSender;
            _logger = loggerFactory.CreateLogger<AccountController>
();
        }
    }
```

In the preceding code, we are using services provided by different components. `UserManager` and `SignInManager` are provided by ASP.NET Identity. The `IEmailSender` and `ISmsSender` are custom classes that we have written which will be used for sending e-mails and SMS messages. We will look more at e-mail and SMS later in this chapter. Logging is provided by the Microsoft Logging extension. The following is a simple login `HTTPGET` method. It simply stores the URL from where the `Login` method is accessed and returns the login page:

```
[HttpGet]
        [AllowAnonymous]
        public IActionResult Login(string returnUrl = null)
        {
            ViewData["ReturnUrl"] = returnUrl;
            return View();
        }
```

Creating Views

Now, we will create respective View page for the login. In this View page, we are just showing the following details:

```
@using System.Collections.Generic
@using Microsoft.AspNet.Http
@using Microsoft.AspNet.Http.Authentication
@using AspNet.Identity.EntityFramework6

@model LoginViewModel
@inject SignInManager<ApplicationUser> SignInManager

@{
    ViewData["Title"] = "Log in";
}

<h2>@ViewData["Title"].</h2>
<div class="row">
    <div class="col-md-8">
        <section>
            <form asp-controller="Account" asp-action="Login" asp-
route-returnurl="@ViewData["ReturnUrl"]" method="post" class="form-
horizontal" role="form">
                <h4>Use a local account to log in.</h4>
                <hr />
                <div asp-validation-summary="ValidationSummary.All"
class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Email" class="col-md-2 control-
label"></label>
                    <div class="col-md-10">
                        <input asp-for="Email" class="form-control"
/>
                        <span asp-validation-for="Email"
class="text-danger"></span>
                    </div>
                </div>
                <div class="form-group">
                    <label asp-for="Password" class="col-md-2
control-label"></label>
                    <div class="col-md-10">
                        <input asp-for="Password" class="form-
control" />
                        <span asp-validation-for="Password"
class="text-danger"></span>
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-md-offset-2 col-md-10">
```

```html
                        <div class="checkbox">
                            <input asp-for="RememberMe" />
                            <label asp-for="RememberMe"></label>
                        </div>
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-md-offset-2 col-md-10">
                        <button type="submit" class="btn btn-
default">Log in</button>
                    </div>
                </div>
                <p>
                    <a asp-action="Register">Register as a new
user?</a>
                </p>
                <p>
                    <a asp-action="ForgotPassword">Forgot your
password?</a>
                </p>
            </form>
        </section>
    </div>

</div>

@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
}
```
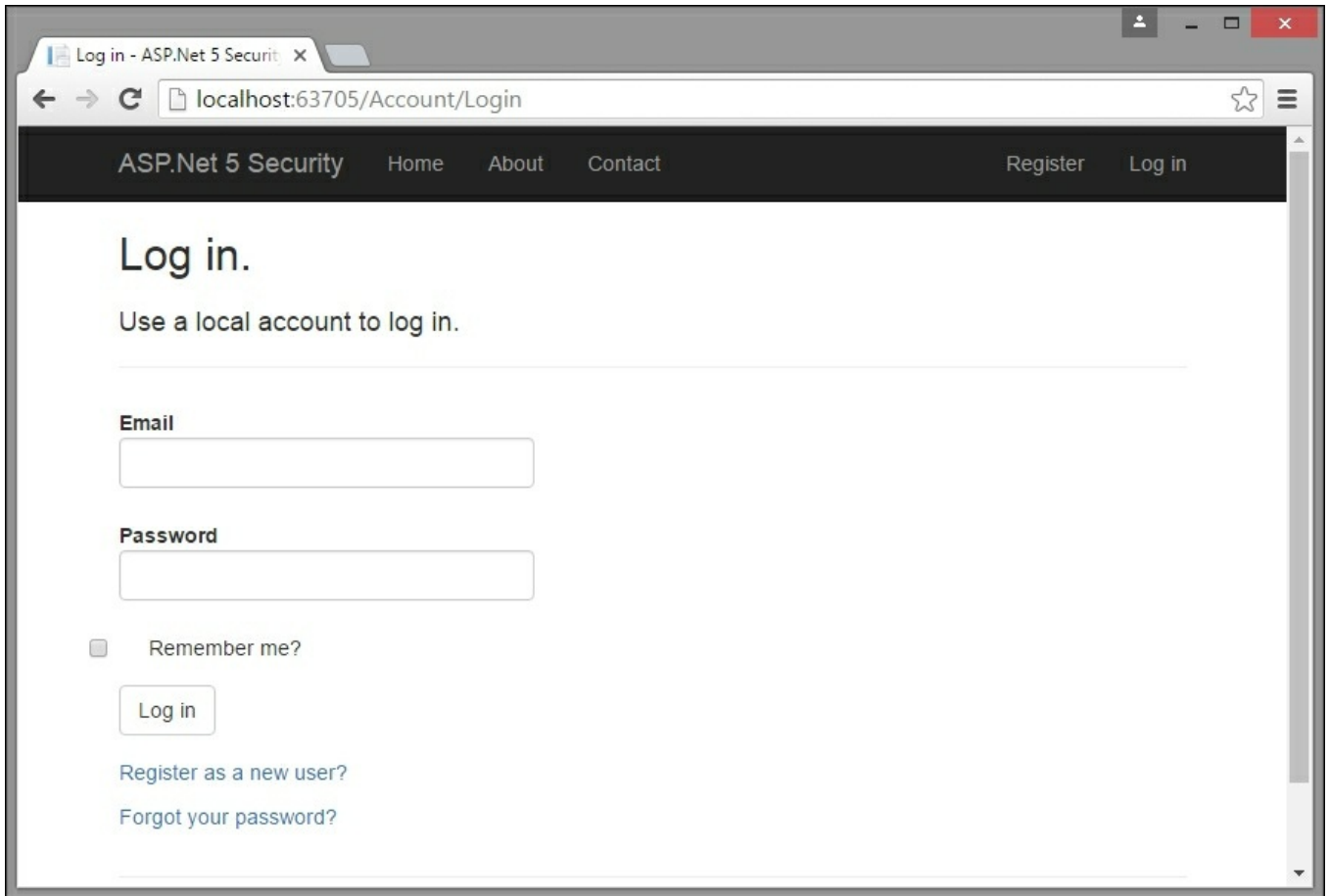
When the user logs into the application for the first time, they might not have any login credentials, so our application should provide a feature that they can use to create a login for themselves. We will create a simple `Register` action method that will just return a View with which the user can register themselves:

```
[HttpGet]
[AllowAnonymous]
public IActionResult Register()
{
    return View();
}
```

We will also create the corresponding View that contains input controls for e-mail, password, password confirmation, and a `Register` button:

```
@model RegisterViewModel
@{
    ViewData["Title"] = "Register";
}

<h2>@ViewData["Title"].</h2>
```

```
<form asp-controller="Account" asp-action="Register" method="post"
class="form-horizontal" role="form">
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All"
class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label">
</label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger">
</span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label">
</label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-
danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-
label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control"
/>
            <span asp-validation-for="ConfirmPassword" class="text-
danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-
default">Register</button>
        </div>
    </div>
</form>

@section Scripts {
    @{ await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
}
```

The following is the corresponding POST action method for registration. Here, the program checks whether the model is valid, and, if it is valid, it will create an ApplicationUser object using the model data and call the Identity API (the CreateAsync method). If it can create the user variable, the user will log in using

that user ID and be redirected to the `Home` page:

```
[HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Register(RegisterViewModel
model)
        {
            if (ModelState.IsValid)
            {
                var user = new ApplicationUser { UserName =
model.Email,  Email = model.Email };
                var result = await _userManager.CreateAsync(user,
model.Password);
                if (result.Succeeded)
                {
                    await _signInManager.SignInAsync(user,
isPersistent:  false);
                    return
RedirectToAction(nameof(HomeController.Index),  "Home");
                }
                AddErrors(result);
            }

            return View(model);
        }
```

The log-out functionality is pretty simple. It just needs to call the `SignoutAsync`
method of Identity API and be redirected to the `Index` page:

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> LogOff()
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation(4, "User logged out.");
            return RedirectToAction(nameof(HomeController.Index),
"Home");
        }
```

Coming back to the log-in functionality, the following is the respective action method.
We are calling the `PasswordSignInAsync` method of Identity API. Upon a successful
login, we redirect the URL from where the log-in functionality is accessed:

```
[HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginViewModel
model, string returnUrl = null)
        {
            ViewData["ReturnUrl"] = returnUrl;
```

```csharp
            if (ModelState.IsValid)
            {
                var result = await
_signInManager.PasswordSignInAsync(model.Email, model.Password,
model.RememberMe, lockoutOnFailure: false);
                if (result.Succeeded)
                {
                    return RedirectToLocal(returnUrl);
                }

            }
            // If there is any error, display the form again
            return View(model);
        }
```

E-mail and SMS services

If you want to add e-mail and SMS services to your application's authentication capabilities, you can do so by creating the interfaces and classes shown here:

```
public interface IEmailSender
{
        Task SendEmailAsync(string email, string subject, string
message)
    }
   public interface ISmsSender
     {
        Task SendSmsAsync(string number, string message);
     }
public class AuthMessageSender : IEmailSender, ISmsSender
     {
        public Task SendEmailAsync(string email, string subject,
string message)
        {
            // We can plug in our email service here to send an
email.
            return Task.FromResult(0);
        }
        public Task SendSmsAsync(string number, string message)
        {
            // We can plug in our SMS service here to send a text
message.
            return Task.FromResult(0);
        }
     }
```

Securing an action method in a Controller

For the sake of explanation, let us assume that the **About** page is a secure page and only authenticated users should be able to access it.

We just have to decorate the `About` action method in the `Home` controller with an`[Authorize]` attribute:

```
[Authorize]
        public IActionResult About()
        {
            ViewData["Message"] = "This is my about page";
            return View();
        }
```

Making the preceding change will redirect the user to the log-in page when the user tries to access the log-in page without logging in to the application:
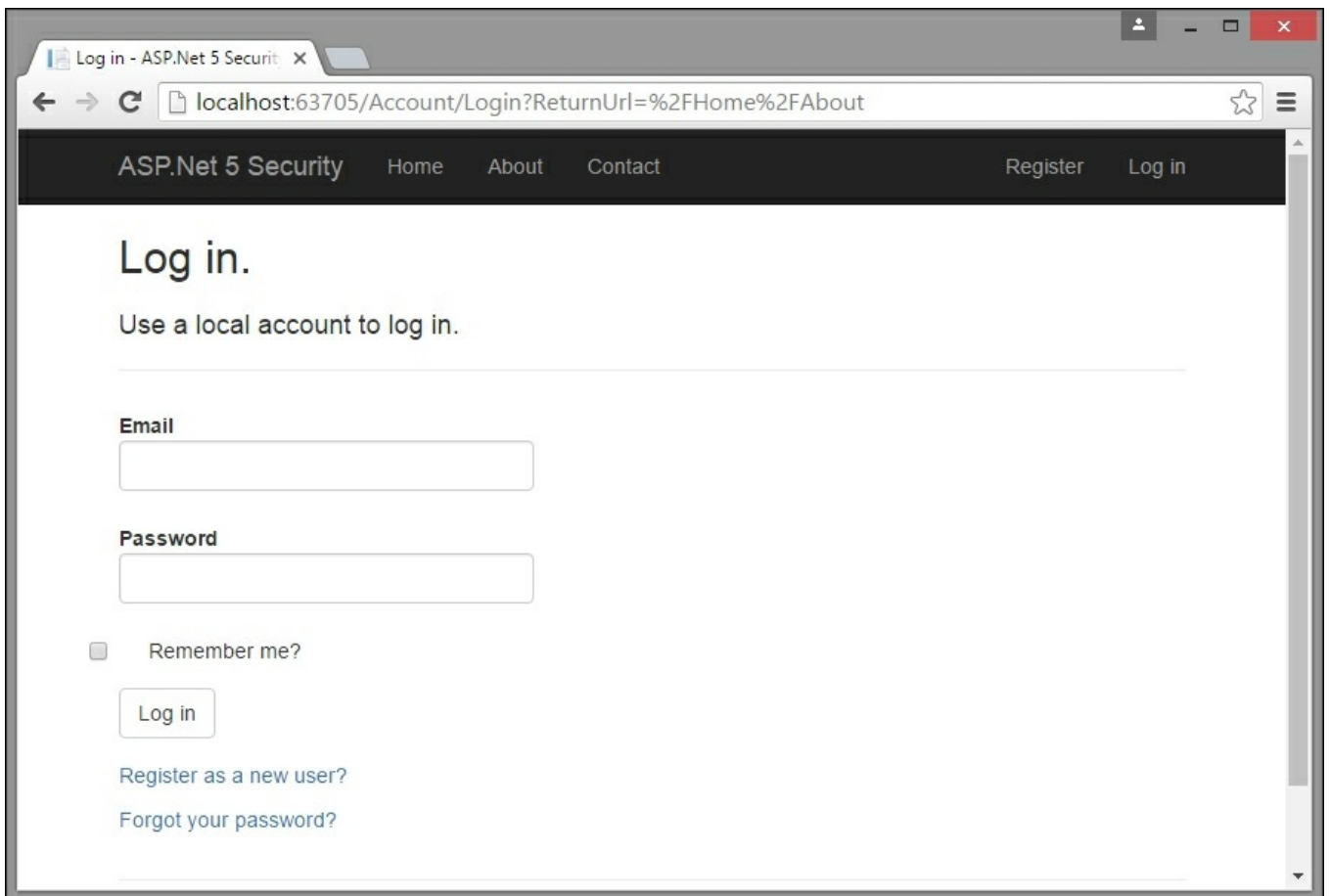
In the following screenshot, you will notice an additional query parameter, `ReturnURL`, in the URL. This `ReturnURL` parameter will redirect the application to that specific page (the value passed in the `ReturnURL` parameter—**Home**/**About** in our case).

Once you log in, you'll be redirected to the page that you requested earlier:
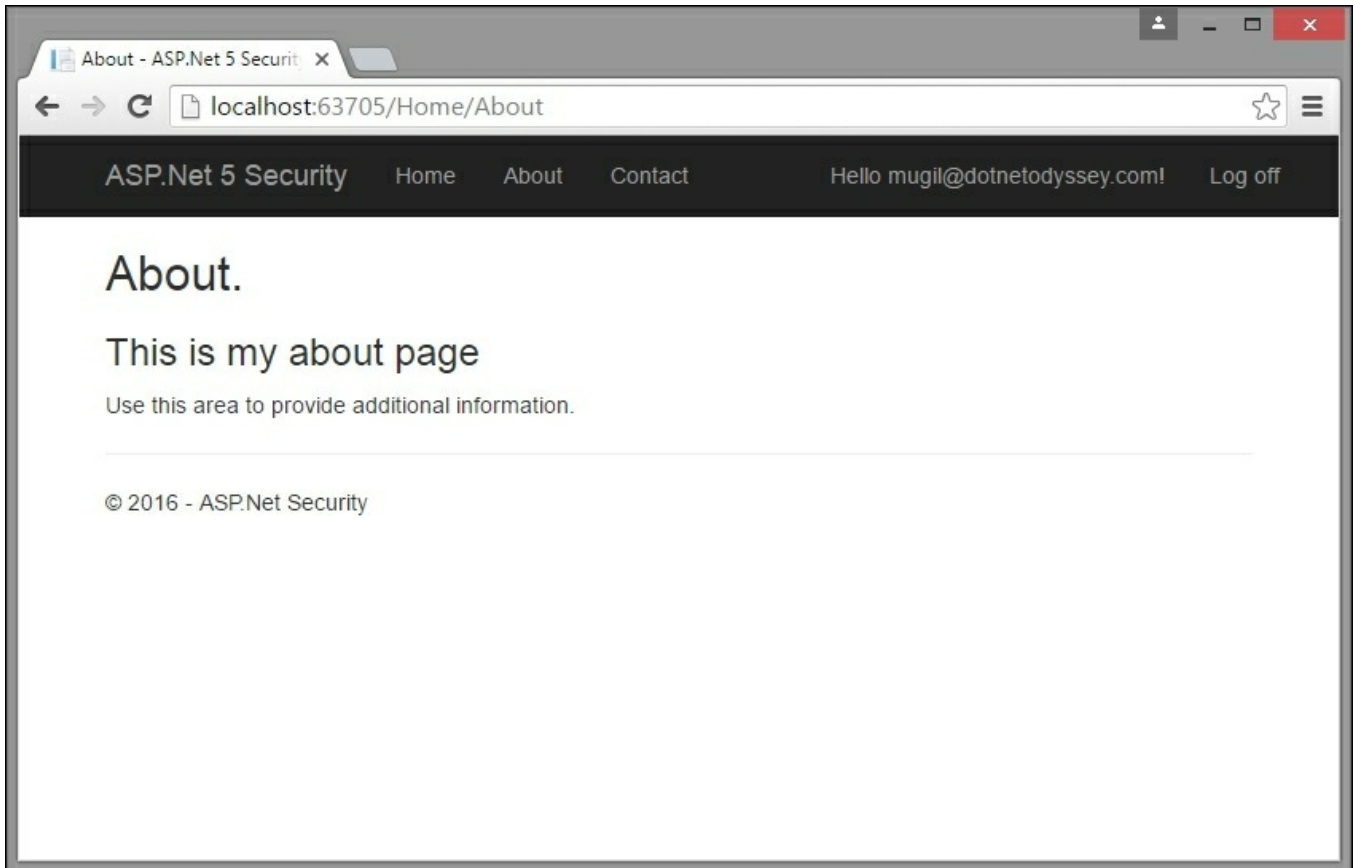
When you register a new user, the details of the user will be stored in the relevant tables created by ASP.NET Identity.

Open the SQL Server Object Explorer window by selecting the option **View** | **SQL Server Object Explorer**, as shown in the following screenshot:
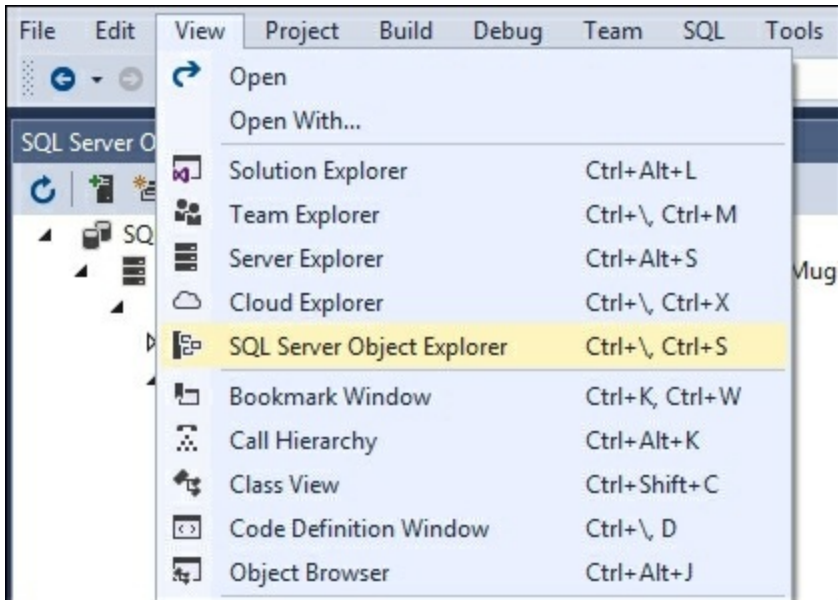
Once you select the **SQL Server Object Explorer** option, you will see a window similar to the following screenshot. ASP.NET Identity creates a database for us by using Entity Framework and the connection string that we provided earlier in the `appsettings.json` package.

ASP.NET Identity creates several tables to maintain identity-related information and the database migration history of Entity Framework. As we are using ASP.NET Identity at the basic level, none of the identity-related tables will get populated, apart from **dbo.AspNetUsers**.:
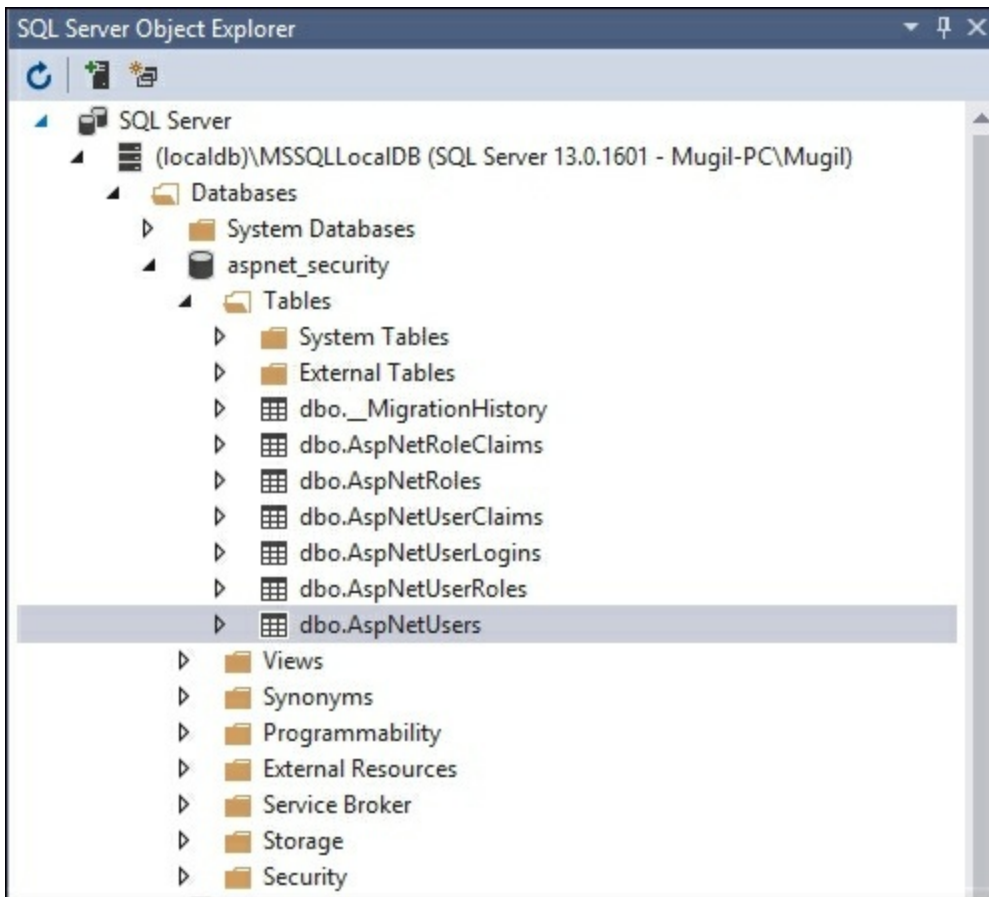
You can right-click on the **dbo.AspNetUsers** table and select **View Data** to see the data:

As only one user has been registered in our application, only one row has been created. Please note that the hashed password (marked by ASP.NET Identity for us)

and no blank passwords will get stored in the table.

Summary

In this chapter, we learned about authentication and authorization. We also learned how to implement ASP.NET Identity in an ASP.NET Core application by following a step-by-step process. We also discussed the tables involved in ASP.NET Identity and learned how to see the data created by ASP.NET Identity.

Part 2. Module 2

ASP.NET Core and Angular 2

Create powerful applications for the modern web

Chapter 1. Getting Ready

ASP.NET Core MVC is a web framework, built on top of the core .NET framework, specifically made for building efficient web applications and APIs that will be able to be reached by a massive range of clients including web browsers, mobile devices, smart TVs, web-based home automation tools, and more.

Angular 2 is the second major installment of AngularJS, a world-renowned development framework born with the idea of giving the coder the toolbox needed to build reactive, cross-platform web-based apps, which are optimized for desktop and mobile: it features a structure-rich templating approach which is based upon a natural, easy-to-write, and readable syntax.

These two frameworks were put together with the same assumption in mind: the HTTP protocol is not limited to serving web pages, it can be also used as a viable platform to build web-based APIs to effectively send and receive data. This theory has slowly made its way through the first 20 years of the World Wide Web and is now an undeniable, widely acknowledged statement, and also a fundamental pillar of almost every modern web development approach.

As for the reasons behind this perspective switch, there are plenty of good reasons for it, the most important of them being related to the intrinsic characteristics of the HTTP protocol. It is simple to use, flexible enough to match most development needs in the always-changing environment which the World Wide Web happens to be, not to mention how universal it has become nowadays, almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

Two players one goal

From the perspective of a fully-functional web-based application, we could say that the Web API interface provided with the ASP.NET Core framework is a programmatic set of server-side handlers used by the server to expose a number of hooks and/or endpoints to a defined request-response message system, typically expressed in structured markup languages such as JSON or XML. This "exposition" is provided using the HTTP protocol thanks to a publicly available web server (typically IIS). Similarly, Angular can be described as a modern, feature-rich client-side library that gives the browser the ability to bind input and/or output parts of an HTML web page to a flexible, reusable, and easily testable JavaScript model.

These assumptions allow us to answer a simple, yet inevitable question: can we put together the server-side strengths of ASP.NET Core's Web API capabilities with the frontend capabilities of the Angular library in order to build a modern, feature-rich, and production-ready web application?

The answer, in short, is yes. In the following chapters, we'll see how we can do that by analyzing all the fundamental aspects of a well-written, properly designed web-based product and how ASP.NET Core and/or Angular can be used to handle each one of them.

What's new in Angular 2?

The new major version of Angular is a complete rewrite of the previous one, entirely based upon TypeScript and ECMAScript 6 specifications. The choice of not making it backward compatible with the previous installment clearly demonstrates the intention of the authors to adopt a completely new approach, any developer who already knows AngularJS will undoubtedly face a huge number of breaking changes, not only in the code syntax but also in the way of thinking and designing your client app. Angular 2 is highly modular, entirely component-based, features a new and improved dependency injection model and has the main goal of being able to easily integrate with other server-side and client-side frameworks.

However, the most important reason why we're picking Angular 2 over other excellent JS libraries such as `ReactJS` and `EmberJS` is the fact that it arrives with a huge stack of features out of the box, making it way simpler to use than the aforementioned competitors. If we combine that with the consistency given by TypeScript language we could very well say that, despite being the youngster, Angular 2 has embraced the framework approach more convincingly than the others. That's a great reason to invest in it, hoping it will keep up with these compelling promises.

The ASP.NET Core revolution

Summarizing what has happened in the ASP.NET world within the last year is not an easy task, in short, we could say that we're undoubtedly facing the most important series of changes in the .NET Framework since the year it came to life. ASP.NET Core is a complete re-implementation of ASP.NET, which unites all the previous web application technologies such as MVC, Web API and Web Pages into a single programming module, formerly known as MVC6. The new framework introduces a fully featured cross-platform component, also known as .NET Core, shipped with a brand new open source **.NET Compiler Platform** (currently known as **Roslyn**), a cross-platform runtime (known as **CoreCLR**), and an improved x64 Just-In-Time compiler (**RyuJIT**).

Note

You might be wondering what happened to ASP.NET 5 and Web API 2, as these used to be quite popular names until mid-2016.

ASP.NET 5 was the original name of ASP.NET Core, before the developers chose to rename it to emphasize the fact that it is a complete rewrite. The reasons for that, together with the Microsoft vision about the new product, are further explained in the following blog post from Scott Hanselman that anticipated the changes on January 16, 2016:

http://www.hanselman.com/blog/ASPNET5IsDeadIntroducingASPNETCore10AndN

For those who don't know, Scott Hanselman has been the outreach and community manager for .NET/ASP.NET/IIS/Azure and Visual Studio since 2007.

Additional information regarding the perspective switch is also available in the following article by the senior Microsoft developer (and NuGet Program Manager) Jeffrey T. Fritz: https://blogs.msdn.microsoft.com/webdev/2016/02/01/an-update-on-asp-net-core-and-net-core/

As for Web API 2, it was a dedicated framework for building HTTP services returning pure JSON or XML data instead of web pages. Initially born as an alternative to the MVC platform, it has been merged with the latter into the new, general-purpose web application framework known as MVC6, which is now shipped as a separate module of ASP.NET Core.

The field of choice – single-page application

In order to demonstrate how ASP.NET Core and Angular 2 can work together to their full extent, we couldn't think of anything better than building a single-page application project. The reason for that is quite obvious: although neither ASP.NET Core nor Angular came into being with the specific purpose of building a SPA, there is no better approach for showcasing some of the best features they have to offer nowadays: we're talking about HTML5 pushState API, webhooks, data transport-based requests, dynamic web components, UI data bindings, and a stateless, AJAX-driven architecture capable of flawlessly encompassing all of these.

Common features of a competitive SPA

If you have never worked on building single-page applications, you need to know what you'll be facing. If you already have, feel free to skip to the following paragraph, unless you don't mind taking another look at the key features provided by any competitive SPA:

- **No server-side round-trips**: A competitive SPA is able to redraw any part of the client UI without requiring a full server-side round-trip to retrieve a full HTML page. This is mostly achieved by implementing a separation of concerns design principle, meaning that the data will be separated from the presentation of data by using a model layer that will handle the former, and a view layer that reads from the latter.
- **Efficient routing**: A competitive SPA is able to keep track of the user current state and location during his whole navigation experience using organized, JavaScript-based routers. This is usually accomplished in one of two ways: the **Hashbang** technique and HTML5 **History API** usage. We'll talk about both of them in [Chapter 2](), *ASP.NET Controllers and Server-side Routes*.
- **Performance and flexibility**: A competitive SPA usually transfers all of its UI to the client thanks to its JavaScript SDK of choice (Angular, JQuery, Bootstrap, and so on). This is often good for network performance, as increasing client-side rendering and offline processing reduces the UI impact over the network. But the real deal brought about by this approach is the flexibility granted to the UI, as the developer will be able to completely rewrite the application's frontend with little to no impact on the server, aside from a few static resource files.

The list could easily grow, as these are only some of the major advantages of a properly designed, competitive SPA. These aspects play a major role nowadays, as many business websites and services are switching from their traditional multi-page application (also known as **MPA**) mindset to fully committed or hybrid single-page application based approaches. The latter have become increasingly popular since 2015 and are commonly called **Native Web Applications** (**NWAs**) because they tend to implement a number of small-scale, single-page modules bound together upon a multi-page skeleton rather than building a single, monolithic SPA.

Furthermore, there are also a lot of enterprise level single-page applications and NWAs flawlessly serving thousands of users everyday, examples include WhatsApp Web, Teleport Web, and Flickr, plus a wide amount of Google web services including Gmail, Contacts, Spreadsheet, Maps, and more. Most of these services are not there, thanks to the heat of the moment; they were intentionally built that way years ago, and they are meant to stay.

Product owner expectations

One the most interesting, yet underrated, concepts brought out by many modern agile software development frameworks such as SCRUM is the importance given to the meanings and definitions of roles. Among these, there's nothing as important as the product owner, also known as the customer in **Extreme Programming** methodology, or customer representative elsewhere. In case you don't know, he's the guy that brings the expectations you'll struggle to satisfy to the development table. He will tell you what's most important to deliver and when; he will prioritize your work based on its manifest business value rather than its underlying architectural value; he'll be entitled by the management to take decisions and make tough calls, sometimes these are great, sometimes not so, that will often have a significant impact on your development schedule. In short, he's the one in charge of the project, that's why, in order to deliver a web application matching his expectancies, you'll need to understand his vision and feel it as if it were yours.

This is always true, even if your project's product owner is your dad, wife, or best friend; even if he happens to be you.

Now that we've made that clear, let's take a look to some of the most common product owner's expectations for a typical web-based single-page application project, we ought to see if the choice of using ASP.NET Core and Angular is good enough to fulfill each one of them:

- **Early release(s)**: No matter whether you're selling a bunch of salad or web-based services, the customer will always want to see what he's buying. If you're using SCRUM, you'll have to release a potentially-shippable product at the end of each sprint; in a Waterfall-based approach you'll have Milestones, and so on. One thing is for sure, the best thing you can do in order to efficiently organize your development is to adopt an iterative and/or modular-oriented approach. ASP.NET Core and Angular, along with the strong separation of concerns granted by their underlying **MVC** or **MVVM** based patterns, will gracefully push you into the mindset needed to do just that.
- **GUI over backend**: You'll often be asked to work on the GUI and frontend functionalities because they will be the only real viewable and measurable thing for the customer. This basically means that you'll have to mock up the data model and start to work on the frontend as soon as possible, delaying everything that goes on under the hood, even if that means leaving it empty, we could say that the hood is what you need the most. Please notice that this kind of approach is not necessarily bad, by all means, we're not tying up the donkey where the (product) owner wants. On the other hand, the choice of using ASP.NET Core together with Angular will grant you the chance of easily decoupling the

presentation layer and the data layer implementing the first and mocking the latter, which is a great thing to do. You'll be able to see where you're going before wasting valuable time or being forced to make potentially wrong decisions. ASP.NET Core's Web API interface will provide the proper tools to do that by allowing you to create a sample web application skeleton in a matter of seconds using **Visual Studio** Controller templates and in-memory data contexts powered by the **Entity Framework 6**, which you will be able to access using entity models and code-first. As soon as you do that, you'll be able to switch to the GUI design using the Angular 2 presentation layer toolbox as often as you want until you reach the desired results. Once you are satisfied, you'll just need to properly implement the web API controller interfaces and hook up the actual data.

- **Fast completion**: None of them will work unless you also manage to get everything done in a reasonable time span. This is one of the key reasons for choosing to adopt a server-side framework and a client-side framework working together with ease. ASP.NET Core and Angular 2 are the tools of choice not only because they're both built on solid, consistent ground, but also because they're meant to do precisely that: get the job done on their respective side and provide a usable interface to the other partner.

- **Adaptability**: As stated by the Agile manifesto, being able to respond to change requests is more important than following a plan. This is especially true in software development, where we could even claim that anything that cannot handle change is a failed project. That's another great reason to embrace the separation of concerns enforced by our two frameworks of choice, as this grants the developer the ability to manage, and even welcome, to some extent, most of the layout or structural changes that will be expected during the development phase.

That's about it. Notice that we didn't cover everything here, as it would be impossible without undertaking an actual assignment. We just tried to give an extensive answer to the following general questions. If we were to build SPA, would ASP.NET Core and Angular 2 be an appropriate choice? The answer is undoubtedly yes, especially when used together. Does it mean that we're done already? Not a chance, as we have no intention of taking this assumption for granted. Conversely, it's time for us to demonstrate it by ceasing to speak in general terms and starting to put things in motion.

A sample single-page application project

What we need now is to conceive a suitable test-case scenario similar to the ones we will eventually have to deal with: a fully featured, production-ready single-page application project complete with all the core aspects you would expect from a potentially shippable product.

In order to do this, the first thing we need to do is to become our own customer for a minute and come up with an idea, a vision to share with our own other self. We'll then be able to put our developer shoes back on and split our abstract plan into a list of items we'll need to implement: these will be the core requirements of our own project. Finally, we'll set up our workstation by getting the required packages, adding the resource files, and configuring both the ASP.NET Core and Angular 2 frameworks into the Visual Studio 2015 IDE.

The vision

If we're going to demonstrate the key features of ASP.NET Core and Angular we can't really take into consideration the number of presentation-oriented websites such as demos, product galleries, corporate or marketing showcases, photo/video/media reels, and so on, as we need something that can show the asynchronous and parallel request processing capabilities of both frameworks to their full extent. In order to fulfill these expectations, we would instead need something similar to a blog engine, a content-management system, and/or a groupware platform such as a community forum or a wiki.

The latter seems to be the most appropriate, as it will mean going through a number of implementation challenges including account management, login/session handling, search-based navigation, full-text search queries, titles and contents indexing, multi-language support, markup syntax rendering, dynamic media handling and a not-so-trivial data model on top of an ever-growing, potentially huge amount of data.

The application we are going to build won't be just a shallow demonstration; we won't throw some working code here and there and expect the reader to connect the dots. Our objective is to create a solid, realistic application using our frameworks of choice while following the current development best practices. Each chapter will be dedicated to a single core aspect and if you feel like you already know your way there, feel free to skip to the next one. Conversely, if you're willing to follow us through the whole loop, you'll have a great journey through the most useful aspects of ASP.NET Core and Angular 2 and how they can work together to deliver the most common and useful web development tasks, from the most trivial ones to the more complex beasts. It's an investment that will pay dividends, as it will leave you with a maintainable, extensible, and well-structured project, plus the knowledge needed to build your own.

To avoid making things too boring, we'll pick an enjoyable, enticing theme from the entertainment industry: a collaborative, wiki-enabled database of open-source video games, where registered users will be able to add/modify entries.

We'll call it **OpenGameList**, also known as `opengamelist.com`. Luckily enough, the domain was still available at the time of writing this book. If you go there now, you'll be able to see what we're going to build from scratch (don't do that if you don't like spoilers, as it could ruin some of your fun).

Core requirements

Our application will follow a rather classic approach: our users will be able to browse the latest entries and navigate through the wiki using internal wiki links or via simple or complex search queries. If they're authenticated, they'll also be allowed to switch to edit mode to modify an item or add a new one, otherwise, they'll be prompted to enter their credentials via a classic login or registration form. On top of all that, we'll also add an administration area where we can do our management stuff such as deleting entries, editing/disabling/deleting users, running maintenance tasks, and so on.

Let's break down these requirements to a list of development topics:

- **Routing**: The app will be able to properly respond to client requests, that is to say, routing them accordingly to what they're up to.
- **Data model**: We'll definitely adopt a database engine to store our entries and the proper tools to access it in a modern, fashionable way. In order do so, we need to define our data architecture by setting up Data Repositories and Domain Entities that will be handled by the server and hooked to Angular 2 through ASP.NET Core's Web API controller interfaces.
- **Web API controllers**: From an MVC-based architectural perspective, one of the main differences between multi-page and single-page applications is that the former's controllers are designed to return views, while the latter ones, also known as API controllers, return serialized data. These are what we will need to implement to put Angular 2 components in charge of the presentation layer.
- **Angular components**: Switching to client-side, we will need to define a set of components to handle UI elements and state changes. As you probably already know, components are the most fundamental elements in Angular 2, replacing Angular 1's controllers and scopes. We'll get to know more about them soon enough.
- **Authentication**: Soon enough we'll have the need to empower our application with a membership context, that way we'll be able to restrict CRUD operations to authenticated users only, keep track of each user actions, prepare the administration layer, and so on.

These will be our main development challenges: if we don't know how to properly handle them, we won't be able to succeed.

The following chapters will address each one of them: we'll also cover other important aspects such as SEO, security, and deployment, as they will be very important when you are ready to publish your application.

Preparing the workspace

The first thing we have to do is to set up our workstation: it won't be difficult because we only need a small set of essential tools. These include Visual Studio 2015, the web platform installer, .NET Framework 4.5 or higher, a web server such as IIS or IIS Express, and a decent source code control system such as Git, Mercurial, or Team Foundation, which we will take for granted as you most likely already have them in place. Just make sure you're all set with these before going further.

Note

IMPORTANT

If you haven't already, be sure to download and install **Visual Studio 2015 Update 3** and the **.NET Core for Visual Studio Tools Preview 2**. These are the latest updates at the time of writing, but will be updated in the near future.

These updates will address some important issues for web development tools, as well as adding improved support for **TypeScript**, **NPM**, and other components that we'll be using later on:

- **Visual Studio 2015 Update 3**: http://go.microsoft.com/fwlink/?LinkId=691129
- **.NET Core for Visual Studio Tools Preview 2.0.1**: https://go.microsoft.com/fwlink/?LinkID=824849

Setting up the project

The first thing we need to do is to create a new **ASP.NET Core web application project**:

1. Fire up Visual Studio 2015 and, from the **File** menu, expand **New** and select **Project** to open a new project modal window.

2. From the **Templates** tree, expand the **Visual C#** node and select the **Web** subfolder: the right section of the modal window will be populated by a number of available project templates. Among these, there are two choices for creating an ASP.NET Core web application project: the first one, optimized for cross-platform deployment, entirely relies upon the new .NET Core Framework; the other one, ideal for a Windows environment, is based upon the latest .NET Framework version (4.6.1 at the time of writing).
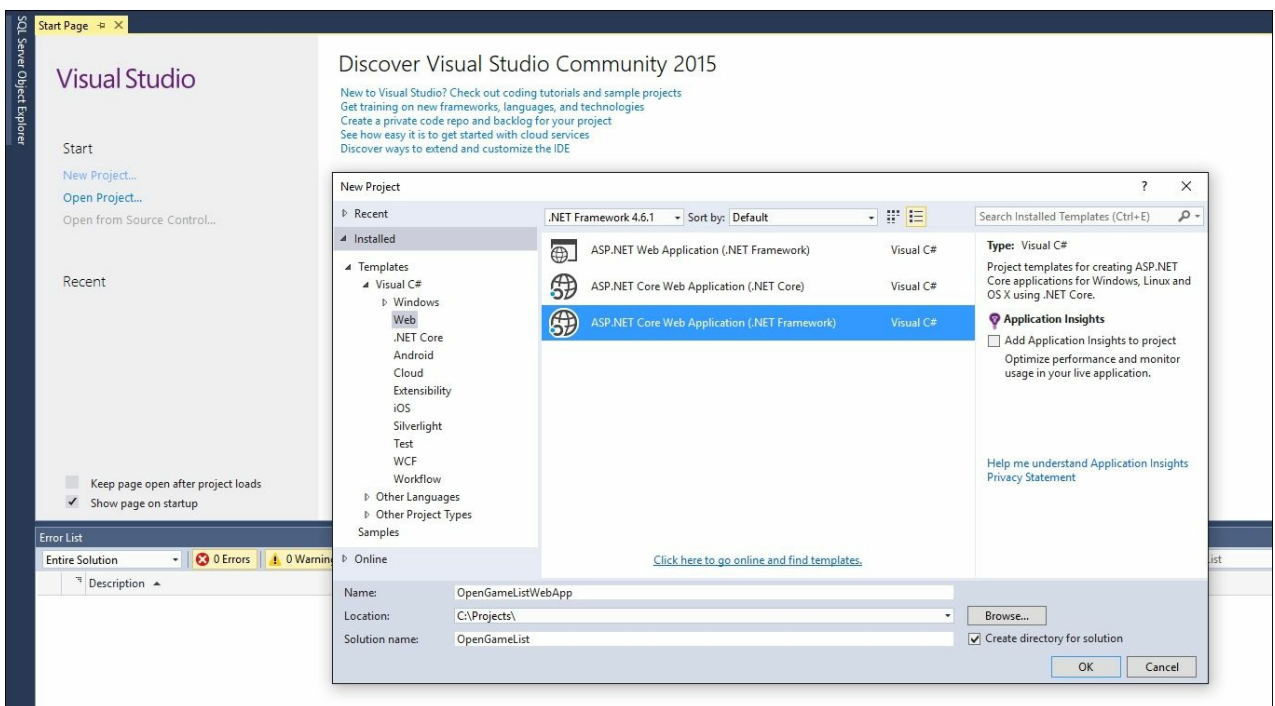
3. The good thing here is that, thanks to the ASP.NET Core versatility, we are free to choose the approach we like the most, as both frameworks are mature enough to support almost everything we will use within this book. The only relevant downside of the .NET Core choice is the lack of compatibility with some NuGet packages that haven't been ported there yet: that's why, at least for our example, we'll be choosing to stick with the full-scale .NET Framework. In order to do that, select the **ASP.NET Core Web Application (.NET Framework)** template and fill in the relevant **Name**, **Location**, and **Solution name** fields. We'll name

the solution **OpenGameList**, while the project will be called
OpenGameListWebApp,as shown in the previous screenshot. Once done, click
OK to continue.

4. In the next modal window, we can further customize our template by choosing
the default contents to include in our project (Empty, Web API, or Web
Application) and the authentication mechanism, should we want to use one.
Choose **Web API** and **No authentication**, then click the **OK** button to create the
project.
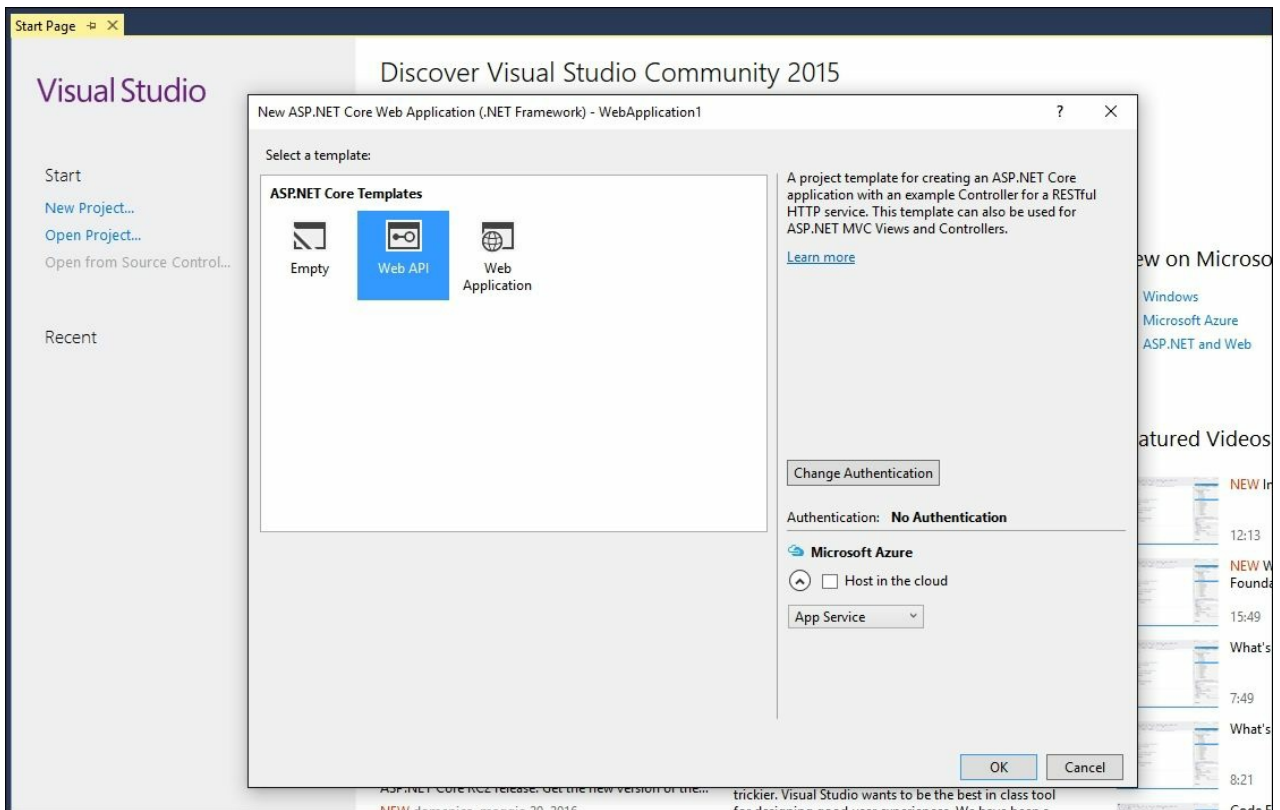
5. If you're used to the Visual Studio Web Application Project templates from
previous ASP.NET versions you may be tempted to choose **Empty** instead, thus
avoiding the insane amount of sample classes, folders, and components,
including a number of potentially outdated versions of various client-side
frameworks such as **Bootstrap**, **KnockoutJS**, **JQuery**, and more. Luckily, these
new ASP.NET Core project templates are extremely lightweight - the one we
did choose comes with nothing more than a `/Controller/` folder, a
`ValuesController.cs` sample, and some useful references in the `Startup.cs`
file.

Package and resource managers

Now that our project has been created, it's time to add the resources we're going to use. As we already mentioned, the layout of a standard ASP.NET Core solution is quite different from what it used to be. The main differences are:

- The solution's projects are now created in a `/src/` folder by default. This is just a convention, though, as they can be placed anywhere.
- There is a brand-new `wwwroot` folder, which will contain the compiled, ready-to-publish contents of our application, while everything else will be the project source code.

Other things worth noting are a `Controller` folder, containing a sample `ValueController.cs` class, a `Startup.cs` file containing the application class and a couple of other files we'll address in a while.
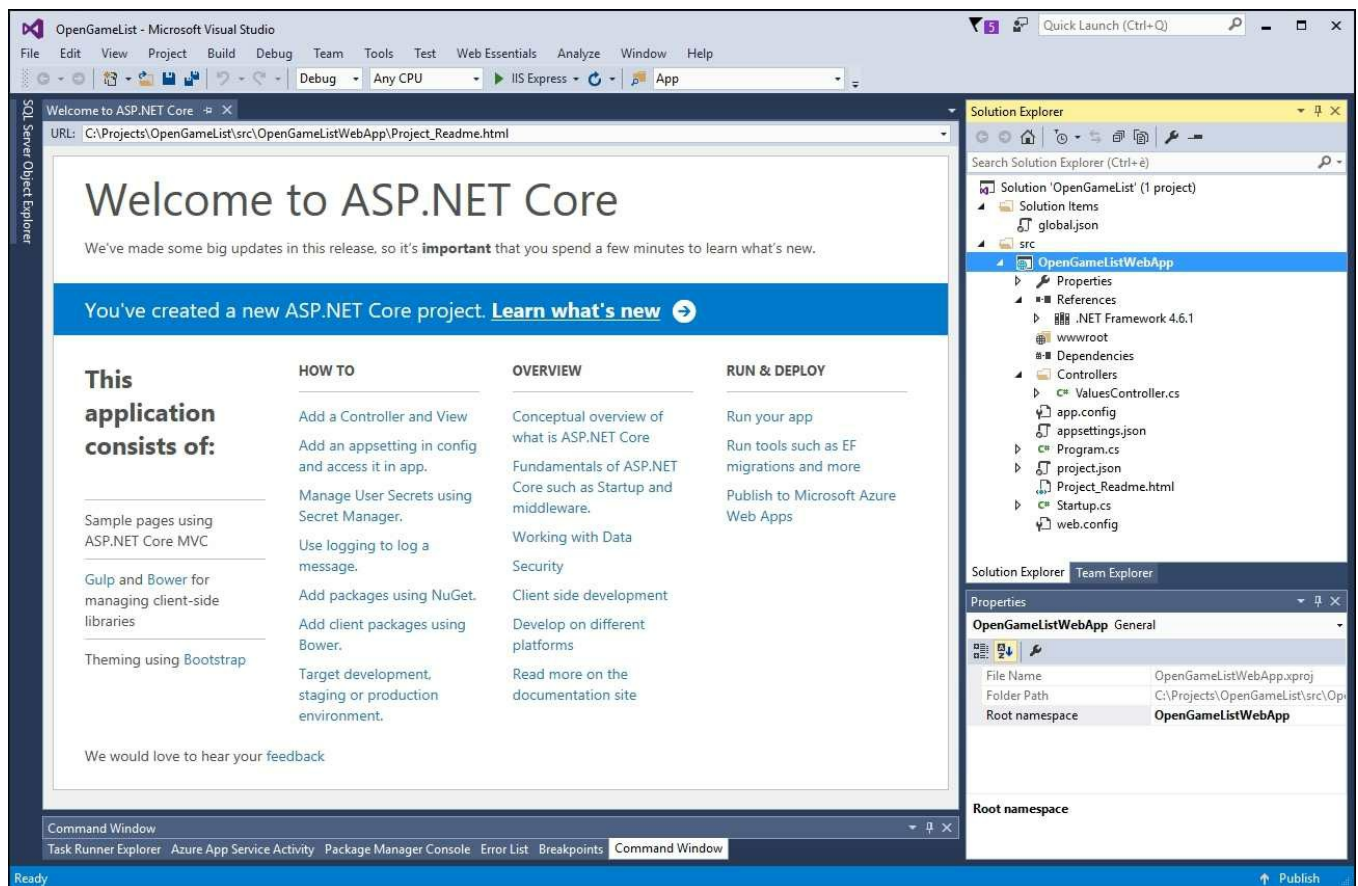
The first thing we need to do is to add a folder called `Scripts` to the root of our project. We'll use this folder to place all of our JavaScript files, and then we'll combine and minify these using a **JavaScript Task Runner** such as Gulp, this way we'll make sure that the resulting file will be added to the `wwwroot` folder

automatically after each build.

Installing the packages

Now it's time to make a good use of the three package managers natively supported by ASP.NET, namely **NuGet**, **NPM**, and **Bower**. These tools will allow you to gather all the resources you need to build your application: they will download all the resources and their dependencies automatically, so you needn't do it manually, thus saving a lot of time. In case you're wondering why we need three of them, it can be useful to briefly recap their scope:

- **NuGet**: This will take care of all the .NET native and third-party packages such as Entity Framework, ASP.NET MVC, and so on. The full package list is stored in the `project.json` file so they can be retrieved and/or checked for updates at any time.
- **NPM**: This was the default package manager for the JavaScript runtime environment known as **Node.js**. Over the last few years, though, it has also been used to host a number of projects, libraries, and frameworks of any kind, including **Angular 2**. The package list is stored in the `package.json` file.
- **Bower**: Another package management system for client-side programming, created by Twitter and maintained on GitHub, specifically designed for frontend development such as **jQuery**, **Bootstrap**, and **AngularJS**. It depends on Node.js and NPM and works under `git`. Its configuration file is called `bower.json`. Notice that, since the Angular 2 team is pushing their code using NPM rather than Bower, we won't be using it in our project.

NuGet and ASP.NET

ASP.NET Core gives us at least four different ways to add NuGet packages to our project:

- Using the Visual Studio powered GUI, accessible by right-clicking the project and choosing **Manage NuGet Packages**.
- Using the Package Manager Console, with the well-renowned `Install-Package` command followed by the package name and build version.
- Using the on-screen helper tools provided by Intellisense, limited to the native .NET modules/libraries.
- Directly adding the package reference to the project's NPM configuration file, also known as `project.json`.

The first three methods, although being absolutely viable, are basically shortcuts for populating the fourth one; the latter has the advantage of being the less opaque one, so we'll just use it.

Project.json

Open the `project.json` file, find the `dependencies` section and add the following packages to the list (new lines are highlighted):

```
"dependencies": {
  "Microsoft.AspNetCore.Mvc": "1.0.0",
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.Extensions.Configuration.EnvironmentVariables":
"1.0.0",
  "Microsoft.Extensions.Configuration.FileExtensions": "1.0.0",
  "Microsoft.Extensions.Configuration.Json": "1.0.0",
  "Microsoft.Extensions.Logging": "1.0.0",
  "Microsoft.Extensions.Logging.Console": "1.0.0",
  "Microsoft.Extensions.Logging.Debug": "1.0.0",
  "Microsoft.Extensions.Options.ConfigurationExtensions":
"1.0.0",
  "Microsoft.AspNetCore.Diagnostics": "1.0.0",

  "Microsoft.AspNetCore.Routing": "1.0.0",

  "Microsoft.AspNetCore.Authentication.JwtBearer": "1.0.0",

  "Microsoft.AspNetCore.StaticFiles": "1.0.0",

  "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0"
}
```

We have added a space to visually separate the default dependencies required by all ASP.NET Core projects from our newly added ones.

Tip

The listed builds are the latest at the time of writing, but they won't last forever: as soon as ASP.NET Core passes its initial release (1.0.0 at the time of writing), these numbers will gradually increase over time, whenever a new version comes out. To check the latest version of each package, just place the cursor between the quotes and delete the version number, a dynamic drop-down list will be shown containing all the latest versions for that given module.

While we're here, it can be useful to check which version of the .NET Framework we are targeting by looking at the `frameworks` key. Since we choose the .NET Framework template, we should find something like this:

```
"frameworks": {
  "net461": { }
},
```

This will most likely change in the future, so be sure to target a version compatible

with the packages you want to use. For the purpose of this book, the **.NET Framework 4.6.1** will do the job.

As soon as we save the `project.json` file, Visual Studio will start to retrieve the missing packages from the web. Wait for it to finish, then proceed with opening the `Startup.cs` file, which is also in the project root.

Startup.cs

If you're a seasoned .NET developer you might already be familiar with the `Startup.cs` file, introduced in OWIN-based applications to replace most of the tasks previously handled by the good old `Global.asax` file. In ASP.NET Core, the `Startup.cs` file serves the same purpose as the `OWIN` startup class, being nothing less than the application main entry point, it is the place where we can add services, choose which application modules and middleware functions to load, handle dependency injection tasks, and configure the pipeline.

However, the similarities end here, the class has been completely rewritten to be as pluggable and lightweight as possible, meaning that it will include and load only what's strictly necessary to fulfill our application's tasks.

To better understand this, let's take a look at the following lines taken from the `Startup.cs` source code shipped with the ASP.NET Core Web API project template we chose:

```
// This method gets called by the runtime. Use this method to
configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    app.UseMvc();
}
```

Notice how empty our application's HTTP request pipeline is, it won't ever serve static files, since there is nothing telling it to do so. To better understand it, let's perform a quick test before proceeding.

Testing the HTTP request pipeline

In order to check that the ASP.NET pipeline is properly working, click on the **Start** Debugging option in the **Debug** menu, or just hit the *F5* hotkey. If everything is OK, your default web browser will open pointing to the following URL: `http://localhost:14600/api/values`.

The page content will show the following:
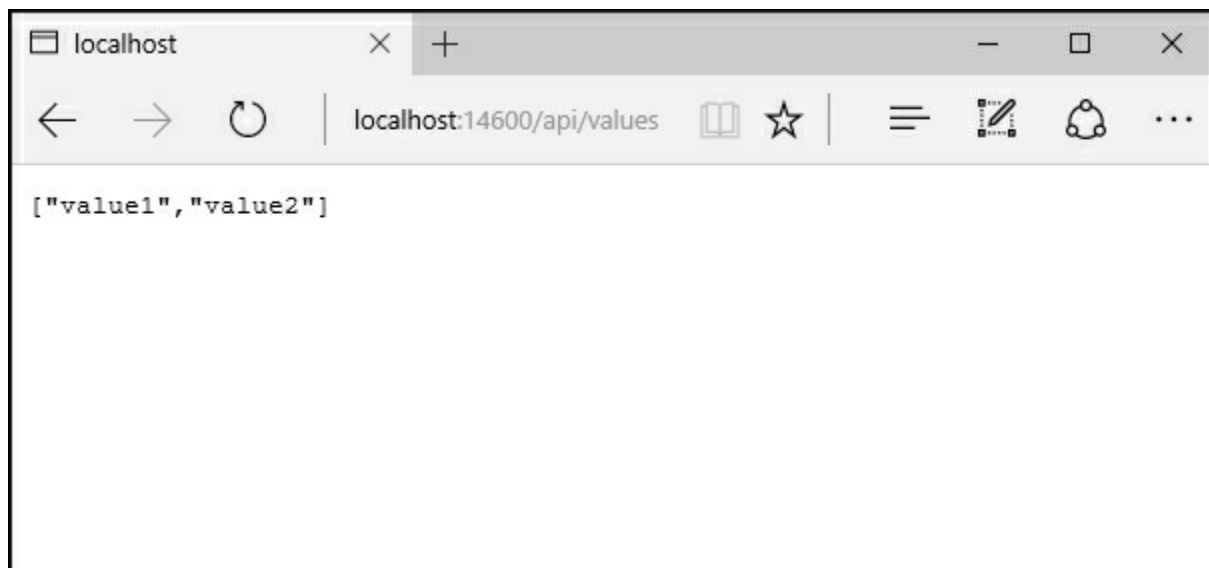
If we're seeing the preceding screenshot, it means that the request pipeline is working fine, the MVC module has been added within the `Startup.cs` file and it's working as expected. That's because there's a sample `ValueController.cs` class in the `/Controllers/` folder, conveniently added by the Web API project template we chose a while ago, that behaves exactly like this.

Now let's try to request the static `Project_Readme.html` file, also added by our chosen template in the project root. In order to reach it, we need to move it inside the `/wwwroot/` folder. Once done, it should be reachable by pointing at the following URL: `http://localhost:14600/Project_Readme.html`.

However, if we try to do that, and then issue that request using the same browser we used before, we would get the following response:
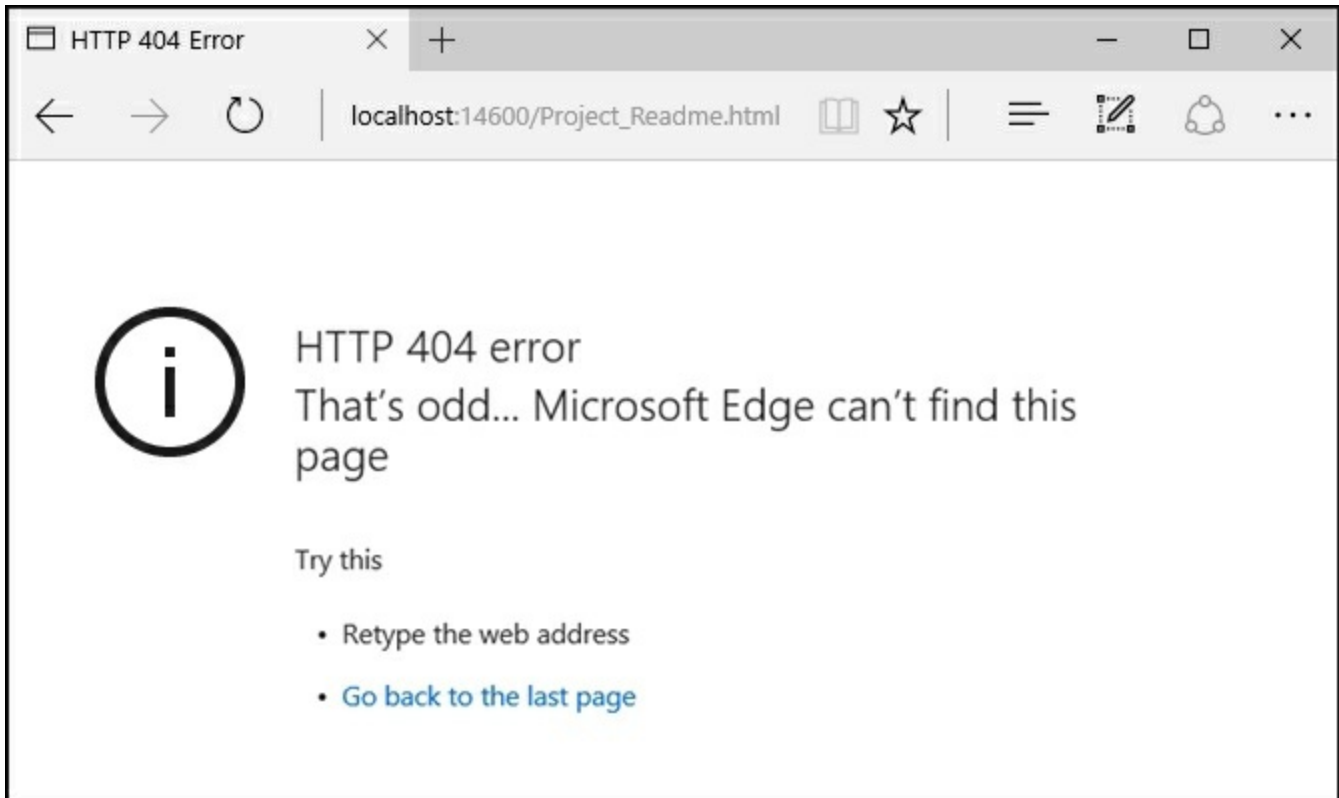
This HTTP 404 error clearly demonstrates what we've just said, the HTTP request pipeline won't serve static files, simply because we didn't tell it to. However, we can easily fix that behavior by adding them to the pipeline within the `Startup.cs` file (new lines highlighted):

```
// This method gets called by the runtime. Use this method to
configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
the following URL:

    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.

    app.UseStaticFiles();

    // Add MVC to the pipeline
```

```
        app.UseMvc();
}
```

While we are here, we also added the following:

- A rewrite rule to enable support for the most common default files (such as `index.htm`, `index.html`, and so on), which will be automatically served without the user having to fully qualify the URI.
- A series of comments to better clarify the meaning of each module, including a reference link to the official ASP.NET Core documentation.

If we run our application again, we should now be welcomed with the following:
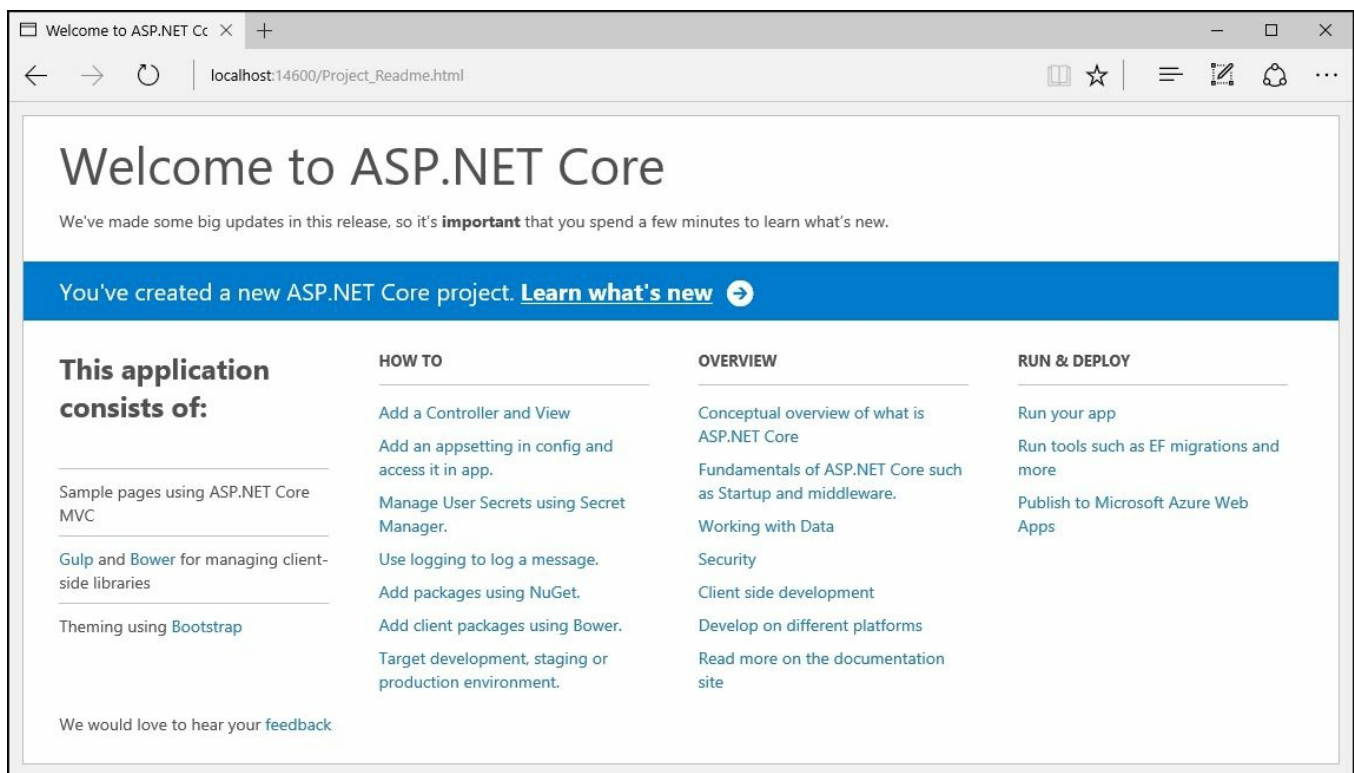
That's it. We have enabled static file support, so that we'll be able to serve not only HTML but also CSS, JS files, and so on. Delete the `Project_Readme.html` and get ready to install an important third-party NuGet package that we'll be using a lot in the following chapters.

Newtonsoft.Json

If you're working with ASP.NET and you've never heard about **Newtonsoft.Json** (formerly **Json.NET**), you've almost certainly missed something that could've eased your job, big time. We're talking about one of the finest libraries - and most useful tools, ever developed for .NET, at least for the writer. It's a very efficient (and thus

very popular), high-performance JSON serializer, deserializer, and all-around framework for .NET, which also happens to be completely open source.

We won't dig into it anymore here, as we'll be using it soon enough. For now, let's just install it by right-clicking on our solution's node in the **Solution Explorer**, then choosing **Manage NuGet Packages for this Solution...** to open up the following panel:
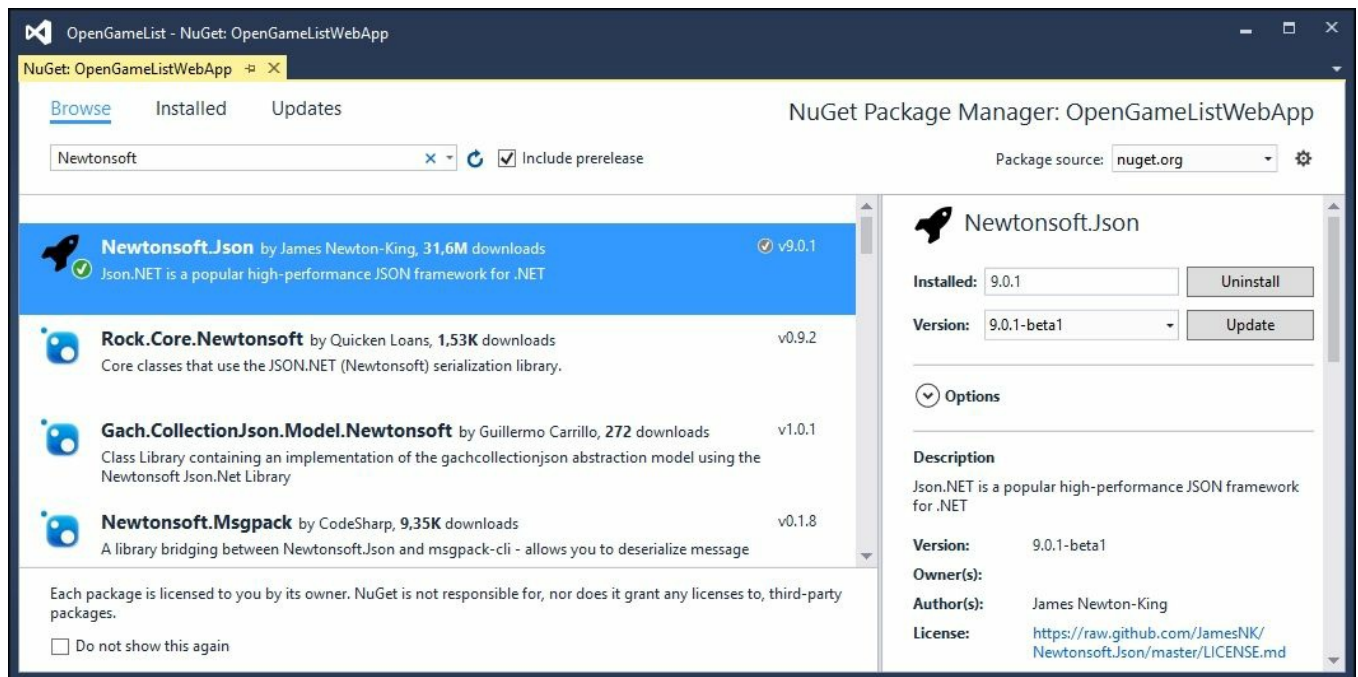
Search for `Newtonsoft.Json` to make it appear if it isn't there already, then left-click to select it. Ensure to select the latest stable version (9.0.1 at the time of writing), click on **Install**, and then wait for the installer to complete its job.

That's it for now: if we open the `project.json` file now, we can see that the `Newtonsoft.Json` package reference has been added at the end of the `dependencies` section, together with the other project-specific dependencies we've manually added before. We'll be installing other packages using the GUI in the following chapters, as soon as we need them, now that we know how easy it is to do that.

JavaScript, TypeScript, or Dart?

Now it's time to choose the client programming language to adopt. Given the fact we're planning to use Angular 2, our choices are basically the following three: good old JavaScript, its Microsoft superset known as **TypeScript**, or the Google growing beast known as **Dart**.

In this project, we're going to use TypeScript for a number of good reasons, the most important of them are as follows:

- TypeScript has a number of features that JavaScript doesn't, such as static typing, classes, and interfaces. Using it in Visual Studio also gives us the chance to benefit from the built-in IntelliSense, which, together with its distinctive features, will allow us to spot most programming errors as we type the code, potentially saving a great amount of time.
- For a large client-side project, TypeScript will allow us to produce a more robust code, which will also be fully deployable anywhere a plain JavaScript file would run. As a matter of fact, since TypeScript is a superset of JavaScript it can be used alongside any JavaScript code without problems.
- Dart is a wonderful newcomer, and it will probably surpass its ECMA script-rivals soon. Currently though, it is still quite immature in terms of available third-party libraries, documentation, development support, and overall community knowledge.

We're not the only ones praising TypeScript: it's something acknowledged by the Angular team itself, considering the fact that the Angular 2 source code has been written using TypeScript, as proudly announced by Microsoft in the following MDSN blog post: https://blogs.msdn.microsoft.com/typescript/2015/03/05/angular-2-built-on-typescript/.

Adding the tsconfig.json file

The first thing we hav to do to set up TypeScript is to add a `tsconfig.json` file to our root project.

In the **e to do to set up TypeScript is to add a tsconfig.json file to our root project. In the Solution Explorer**, right-click on the root project node and select **Add**, then **New Item**; switch to the **client-side** from the top left tree view, then select **TypeScript Configuration File,** and add the `tsconfig.json` file to the project root.
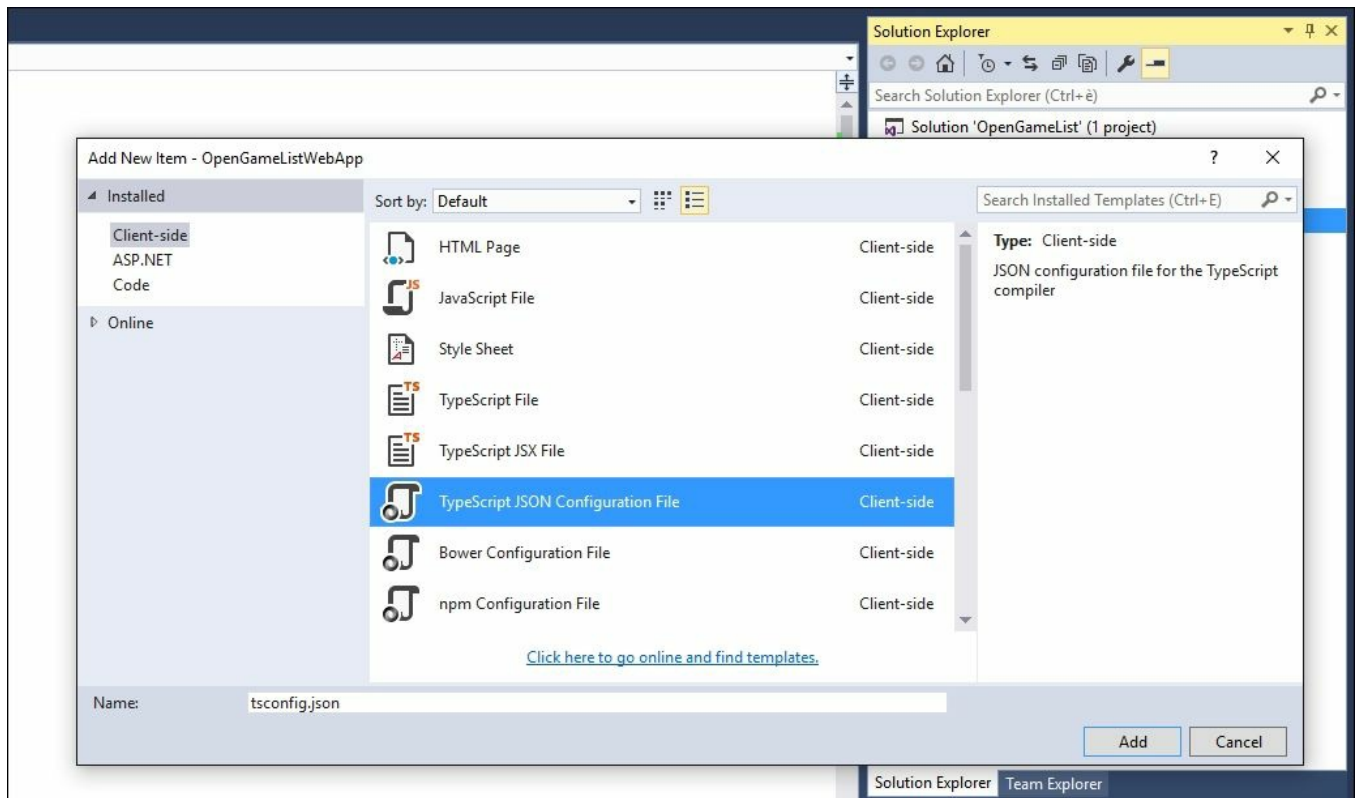
The `tsconfig.json` file will be generated with its default set of options, which are good for most scenarios but not ideal for what we're going to do. That's why we need to add/change some settings until it will look like the following:

```json
{
  "compileOnSave": false,
  "compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "module": "system",
    "moduleResolution": "node",
    "noImplicitAny": false,
    "noEmitOnError": false,
    "removeComments": false,
    "target": "es5"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

These instructions will influence how Intellisense and our external TypeScript compiler will work; two things that will help us, big time.

Wait a minute, did we just say external TypeScript compiler? What about the built-in, integrated TypeScript compile feature provided by the Visual Studio IDE?

Well, we just said that one of TypeScript's biggest advantages is that we can use it anywhere a plain JavaScript file would run, that's because any `.ts` file can be compiled into a `.js` file without any problem. The compile task is handled by the TypeScript compiler itself, also known as **tsc**, which can be run directly from the Visual Studio IDE with the help of the `tsconfig.json` we added in the preceding paragraph. That file, as we've just seen, contains a wide set of compiling options such as creating source maps, creating definition files, concatenating everything into a single output file, and so on. Looking great, isn't it?

Unfortunately, not so much. Although Visual Studio 2015 does a pretty decent job of acting as a bridge between our TypeScript sources and **tsc**, it still lacks an important key feature, it doesn't give the option to minify/uglify the resulting JS files, thus leaving them uncompressed after the compile task.

Such feature can be trivial during development, yet it happens to be quite important in production because we'll definitely want to cloak our source code before publishing it. That's why the best thing we can do is to stop the Visual Studio IDE from compiling TypeScript files and implement a custom alternative that does support minify/uglify.

Note

In case you're wondering about what the terms minify and/or uglify actually mean, we strongly suggest that you read the following Wikipedia page: https://en.wikipedia.org/wiki/Minification_(programming).

Luckily, we're going to use something we would need anyway, a dedicated, streamlined, and modern task runner that goes by the name of **Gulp**. In order to install it, though, we need to set up the appropriate package manager.

Introducing NPM

NPM is the tool we will use to add some important packages to our project, the most relevant ones being **Gulp** and **Angular 2**. To install it, do the following:

1. Go to the **Solution Explorer.**
2. Right-click on the root project node.
3. Select **Add**, then **New Item.**
4. Switch to **Client-side** from the top left tree view, then select **NPM Configuration File**, and add the `package.json` file to the project root.
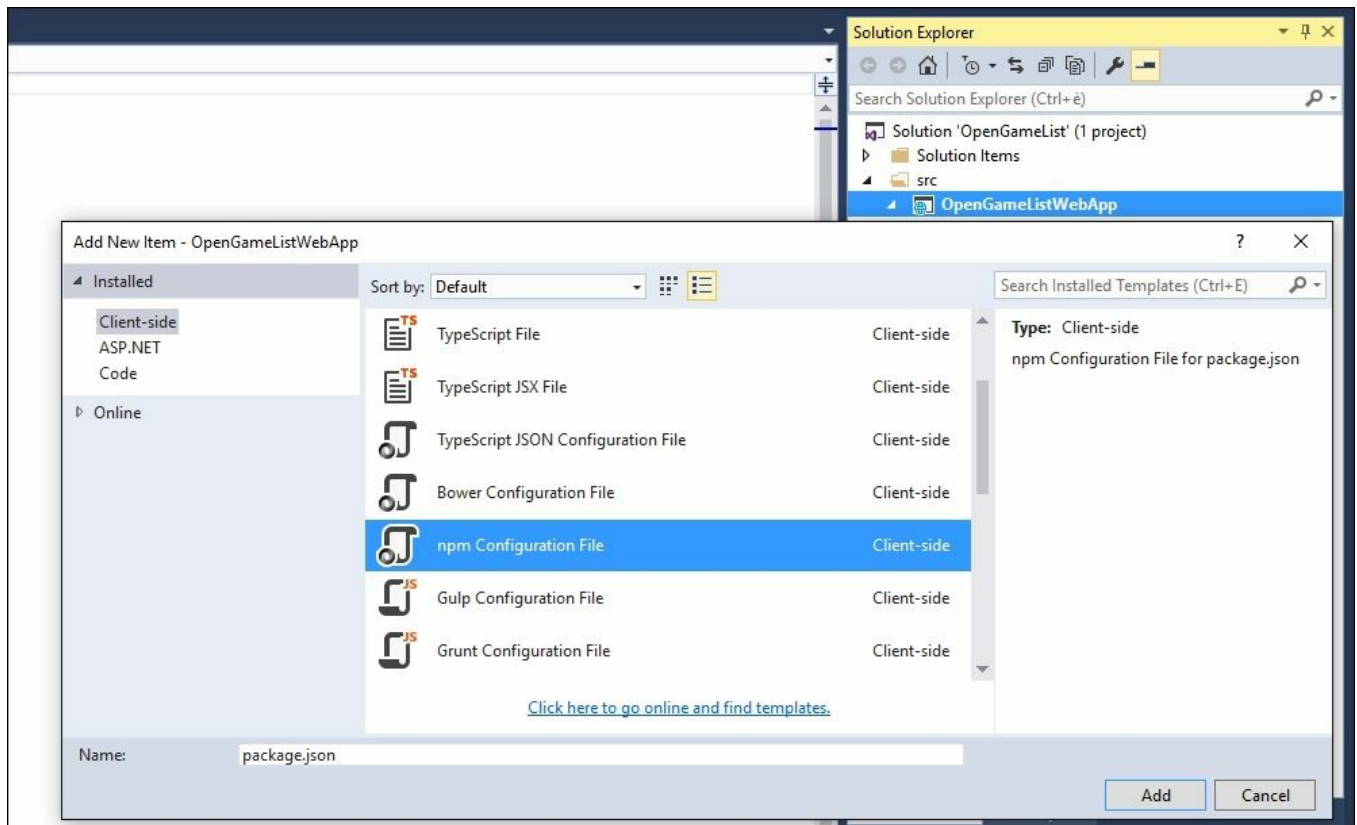
Change the contents of the automatically-generated `package.json` file to match the following:

```json
{
  "version": "1.0.0",
  "name": "opengamelistwebapp",
  "private": true,
  "dependencies": {
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "gulp-clean": "^0.3.2",
    "gulp-concat": "^2.6.0",
    "gulp-sourcemaps": "^1.6.0",
    "gulp-typescript": "^2.13.6",
    "gulp-uglify": "^2.0.0",
    "typescript": "^1.8.10"
  }
}
```

Please notice that, at the time of writing, Gulp 3.9.1 is the latest version. You can check the most recent build by checking the Visual Studio **Intellisense**, which is able to perform impressive real-time update checks upon each package's latest build just like it does within the `project.json` file:
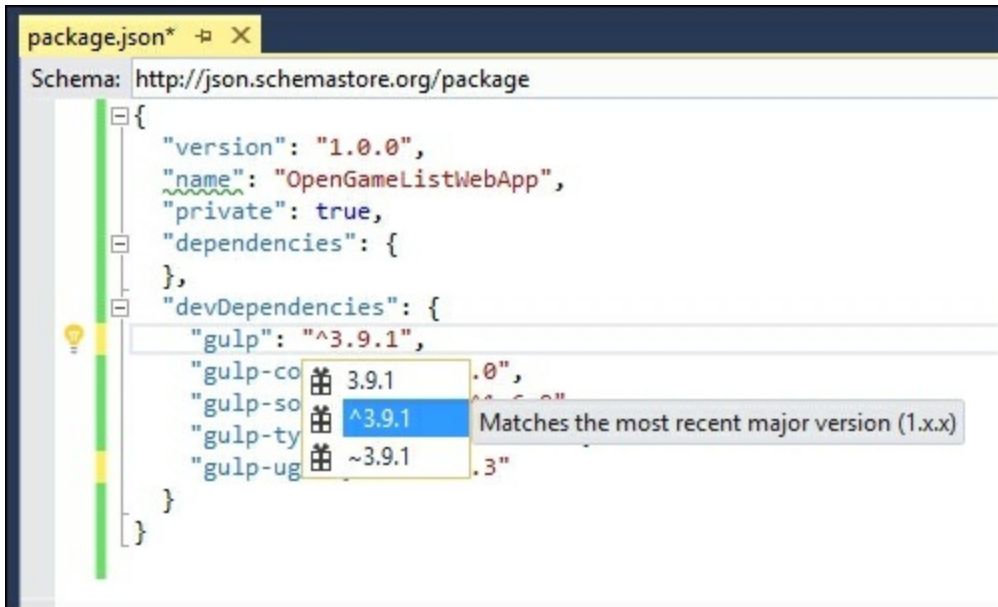
As an alternative, you can always visit the Gulp project NPM page at
https://www.npmjs.com/package/gulp and change the build version numbers
accordingly.

Notice that you aren't forced to input precise build numbers, as you can also use the
standard **npmjs** syntax to specify auto-update rules bound to custom version ranges
using the supported prefixes, such as the following:

- **The Tilde (~)**: ~1.1.4 will match all 1.1.x versions, excluding 1.2.0, 1.0.x and
  so on.
- **The Caret (^)**: ^1.1.4 will match everything above 1.1.4, excluding 2.0.0 and
  above.

This is another scenario where Intellisense will come in handy, as it will also suggest
how to do it.

Note

For an extensive list of available `npmjs` commands and prefixes, you can also check
out the official npmjs documentation at https://docs.npmjs.com/files/package.json.

Working with Gulp

As you most likely already know, Gulp is a powerful task runner toolkit that we will
use to automate some time-consuming tasks in our development workflow. If you
never worked with a task runner, think of it as a batch file or, even better, a
PowerShell script powered with some useful web development tools such as file

concatenation, js/css shrinking, and more.

You will notice that, together with Gulp, we're also installing some Gulp-related plugins:

- `gulp-clean`: This deletes the contents of the destination folders
- `gulp-concat`: This merges multiple files into one
- `gulp-sourcemaps`: This generates the Source Maps
- `gulp-typescript`: This compiles TypeScript files into JS
- `gulp-uglify`: This minifies JavaScript files

Configuring Gulp is as simple as adding the **Gulp Configuration** file to your project. You can do that just like you did with the **NPM Configuration** file, right-click on the root project node in the **Solution Explorer** and select **Add**, then **New Item**. From the **client-side** tree panel, locate and select the **Gulp Configuration** file and add a `gulpfile.js` file to the project root.

The Gulp configuration file, generally known as **Gulpfile**, is basically a list of tasks and commands that Gulp will execute, written using a mostly readable JavaScript syntax. Working with it can be tricky at first, especially if you want to use it to handle complex tasks. Digging too much into it would take us out of the scope of this book, so we'll just see how we can tell it to do a rather simple set of jobs, merge all the JavaScript files in the `/Scripts/` folder, minus those in a directory for third-party scripts/libraries, into a single, minified `all.min.js` file which will be generated into the `wwwroot` folder. This can be obtained with the following code:

```
var gulp = require('gulp'),
    gp_clean = require('gulp-clean'),
    gp_concat = require('gulp-concat'),
    gp_sourcemaps = require('gulp-sourcemaps'),
    gp_typescript = require('gulp-typescript'),
    gp_uglify = require('gulp-uglify');

/// Define paths
var srcPaths = {
    app: ['Scripts/app/main.ts', 'Scripts/app/**/*.ts'],
    js: ['Scripts/js/**/*.js']
};

var destPaths = {
    app: 'wwwroot/app/',
    js: 'wwwroot/js/'
};
// Compile, minify and create sourcemaps all TypeScript files and
place them to wwwroot/app, together with their js.map files.
gulp.task('app', function () {
    return gulp.src(srcPaths.app)
```

```
                .pipe(gp_sourcemaps.init())

.pipe(gp_typescript(require('./tsconfig.json').compilerOptions))
            .pipe(gp_uglify({ mangle: false }))
            .pipe(gp_sourcemaps.write('/'))
            .pipe(gulp.dest(destPaths.app));
});


// Delete wwwroot/app contents
gulp.task('app_clean', function () {
    return gulp.src(destPaths.app + "*", { read: false })
        .pipe(gp_clean({ force: true }));
});


// Copy all JS files from external libraries to wwwroot/js
gulp.task('js', function () {
    return gulp.src(srcPaths.js)
        // .pipe(gp_uglify({ mangle: false })) // disable uglify
        // .pipe(gp_concat('all-js.min.js')) // disable concat
        .pipe(gulp.dest(destPaths.js));
});


// Delete wwwroot/js contents
gulp.task('js_clean', function () {
    return gulp.src(destPaths.js + "*", { read: false })
        .pipe(gp_clean({ force: true }));
});


// Watch specified files and define what to do upon file changes
gulp.task('watch', function () {
    gulp.watch([srcPaths.app, srcPaths.js], ['app', 'js']);
});
// Global cleanup task
gulp.task('cleanup', ['app_clean', 'js_clean']);


// Define the default task so it will launch all other tasks
gulp.task('default', ['app', 'js', 'watch']);
```

Here's a brief explanation of what these commands actually do:

- In lines 1-6, we set up Gulp and the required plugins. Notice that every one of them will be initialized into a variable, as that's the way Gulp works.
- In lines 8-17,we define the file paths we will use. Doing that here will allow us to write them only once, thus making the Gulp file easier to maintain. Notice that we're splitting the source and the destination paths into two distinct objects, this will mentally help us to keep the development environment separated from the production one.
- In lines 19-27, we have the app task, which is the most important and complex one. This task makes use of three different plugins: sourcemaps, typescript, and uglify. Notice how each line is bound to a specific plugin action, which takes

care of a single job, following the *"It should do one thing"* paradigm Gulp is all about. Line 21 defines the source files, line 22 initializes the sourcemaps plugin, line 23 instructs the plugin to fetch the TypeScript compiler options from the `tsconfig.json` file, and so on.

- In lines 29-33, we have the `app_clean` task, which will erase every file and subfolder within the `wwwroot/app` target path.
- In lines 35-39, we define the `js` task, which is meant to publish external JavaScript libraries manually added to the project. This is a rather simple task, as it performs a mere copy of the `.js` files from a source folder into its production counterpart. Notice how we could easily choose to also minify and/or concatenate the source files into a single one by uncommenting a few lines of code. However, since this task is meant to handle third-party JavaScript libraries, it would be wise not to manipulate their code.
- In lines 43-47, there's the `js_clean` task that will erase everything within the `wwwroot/js` target path.
- In lines 49-52, we have the `watch` task. This is basically a file watcher/monitoring task that will execute the `app` and `js` tasks if one or more of their source files will change.
- In lines 54-55, we created a `cleanup` task that will erase the content of all destination paths by launching the `app_clean` and `js_clean` tasks. It can be useful to reset these folders to their initial state.
- Last but not least, in lines 57-58, we define the `default` task that will execute all the other tasks, including `watch`, within a single call. This is the one we will launch.

Note

It's worth noting that the **uglify** plugin requires the `mangle` option to be set to `false`: this is a workaround for a bug in the Angular 2 RC5 that will hopefully be fixed in future releases. For more info regarding this topic, check out the following URL from the official Angular 2 GitHub repository: https://github.com/angular/angular/issues/10618.

To execute the Gulp file, right-click on it from the **Solution Explorer** and choose **Task Runner Explorer**, or open it manually by selecting it from **File**, **Other Windows** in the Visual Studio 2015 main menu bar. Once opened, click the refresh button, then right-click the `default` task and select **Run** to activate it.
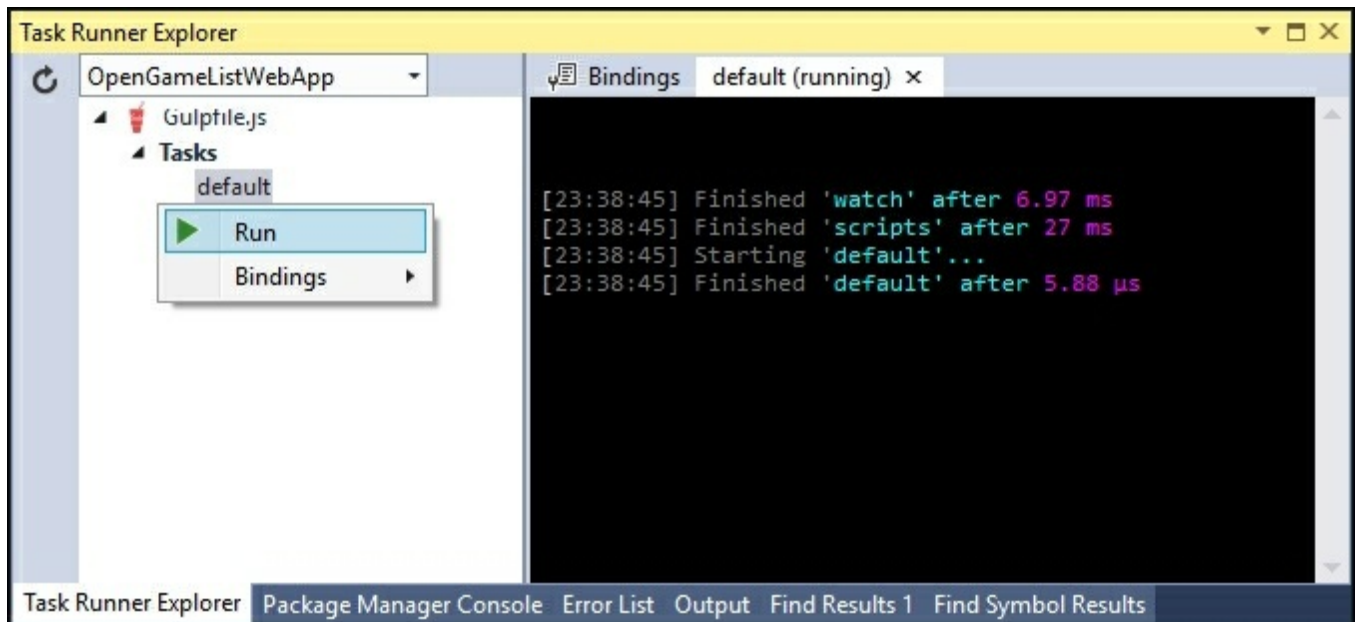
Once you do that, the `watch` task we just created will silently run in background, keeping tracks of our changes and act accordingly. Notice the **default (running)** word within the tab label, which will remind us that there is still an ongoing task: closing that console window will immediately shut it down.

That's everything we'll ever need from Gulp for this project; there's no need for us to go further. If you want to learn more about Gulp and its configuration file syntax, you will find a lot of resources and documentation, together with a lot of useful samples, in the official website at the following address: http://gulpjs.com/.

Dependent tasks

Since we're going to do a lot of modifications to the files contained in the `/Scripts/app/` folder within the following chapters, it would be great if the `app_clean` task could run automatically before the `app` task: such behavior would ensure that we'll always get rid of outdated and orphaned files without having to manually perform the `cleanup`. Luckily enough, Gulp allows us to easily do that by passing an optional array of dependent tasks that will be launched before the main one.

Let's go back to the line where we defined the `app` task and add the following (updated code are highlighted):

```
gulp.task('app', ['app_clean'], function () {
```

That's it. From now on the `app` task will launch the `app_clean` dependent task and wait for its completion before executing itself, meaning that the `/wwwroot/app/`

folder contents will be erased before the arrival of the new file, this is precisely what we wanted.

Note

It's worth noting that the synchronous behavior of the `app_clean` task is guaranteed by its `return` value, whenever a dependent task is returning itself, the main task will wait for its completion before running.

Using Grunt instead of Gulp

Before Gulp was widely welcomed by the web application development community, **Grunt** used to be the king of the hill. Does that mean the former killed the latter? Well, most certainly not. Grunt is still an excellent tool and can be a great alternative to Gulp, especially if you already know how to use it. Unless you do, though, we suggest starting with Gulp because we think that it has a fair edge on code flexibility, and also a more streamlined approach. However, if you don't feel like using it, sticking to Grunt and its renowned plugins is just as good, you won't ever get fired for such a choice.

Using Grunt instead of Gulp is as easy as doing the following:

1. In **NPM's** `package.json` file, replace the Gulp references - together with the relevant plugins, with the Grunt packages.
2. In the **Solution Explorer**, add a `gruntfile.js` file instead of a `gulpfile.js`.
3. In the `gruntfile.js`, rewrite the same tasks defined previously using the Grunt syntax.

This is a good example of the Grunt-powered `package.json`:

```
{
  "version": "1.0.0",
  "name": "opengamelistwebapp",
  "private": true,
  "dependencies": {
  },
  "devDependencies": {
    "grunt": "^0.4.5",
    "grunt-contrib-clean": "^1.0.0",
    "grunt-contrib-copy": "^1.0.0",
    "grunt-contrib-uglify": "^1.0.0",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-ts": "^5.3.2",
  }
}
```

And this is how the `gruntfile.js` would look after porting all the Gulp tasks defined previously to Grunt syntax:

```javascript
module.exports = function (grunt) {
    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-ts');

    grunt.initConfig({
        clean: [
            'Scripts/app/*',
            'Scripts/js/*'
        ],
        ts: {
            base: {
                src: [
                    'Scripts/app/main.ts',
                    'Scripts/app/**/*.ts'
                    ],
                outDir: 'wwwroot/ app',
                tsconfig: './tsconfig.json'
            }
        },

        uglify: {
            my_target: {
                files: [{
                    expand: true,
                    cwd: 'wwwroot/ app',
                    src: ['**/*.js'],
                    dest: 'wwwroot/ app'
                }]
            },
            options: {
                sourceMap: true
            }
        },

        // Copy all JS files from external libraries and required
NPM packages to wwwroot/js
        copy: {
            main: {
                files: [{
                    expand: true,
                    flatten: true,
                    src: [
                        'Scripts/js/**/*.js'
                    ],
                    dest: 'wwwroot/js/',
                    filter: 'isFile'
```

```
                }]
            }
        },

        // Watch specified files and define what to do upon file
changes
        watch: {
            scripts: {
                files: [
                    'Scripts/**/*.ts',
                    'Scripts/**/*.js',
                    ],
                tasks: ['clean', 'ts
', 'uglify', 'copy']
            }
        }
    });

    // Global cleanup task
    grunt.registerTask('cleanup', ['clean']);

    // Define the default task so it will launch all other tasks
    grunt.registerTask('default', ['clean', 'ts', 'uglify', 'copy',
'watch']);
};
```

As we can see, the syntax is very different but the underlying logic and features are quite similar, we still have a bunch of dedicated tasks to handle TypeScript files, minify actions, copy operations, and monitor content change, and then the default one to wrap everything up. Notice how, just like we did with Gulp, the TypeScript compiler options are fetched from the `tsconfig.json` file so we don't have to write them a second time.

Note

To learn more about Grunt and its configuration file syntax you can visit the official website at the following address: http://gruntjs.com/.

That's basically all we need to know about task runners. Too bad we don't yet have any TS and/or JS files to properly test what we just did! Don't worry, though, we'll get there soon enough. Before that, let's finish our **NPM** journey by bringing the last (but not least) of our players into the loop.

Adding Angular 2

There are basically two ways to get Angular 2, both with their pros and cons, using NPM, which implies fetching the latest build and hosting the code within our project, or by a dedicated set of links to a suitable CDN.

Using NPM

To install Angular 2 using NPM we need to open the `package.json` file and add a bunch of packages under the `dependencies` node. This is how the file should look like at the end (Angular lines are highlighted):

```json
{
  "version": "1.0.0",
  "name": "OpenGameListWebApp",
  "private": true,
  "dependencies": {
    "@angular/common": "2.0.0-rc.5",

    "@angular/compiler": "2.0.0-rc.5",

    "@angular/core": "2.0.0-rc.5",

    "@angular/http": "2.0.0-rc.5",

    "@angular/platform-browser": "2.0.0-rc.5",

    "@angular/platform-browser-dynamic": "2.0.0-rc.5",

    "@angular/upgrade": "2.0.0-rc.5",

    "core-js": "^2.4.1",

    "reflect-metadata": "^0.1.3",

    "rxjs": "5.0.0-beta.6",

    "systemjs": "^0.19.37",

    "zone.js": "^0.6.12"
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "gulp-clean": "^0.3.2",
    "gulp-concat": "^2.6.0",
    "gulp-sourcemaps": "^1.6.0",
    "gulp-typescript": "^2.13.6",
    "gulp-uglify": "^2.0.0",
    "typescript": "^1.8.10"
  }
}
```

The packages starting with the `@` symbol are part of the Angular 2 bundle, which, at the time of writing, reached its **Release Candidate 5 (RC5)** development stage. The other ones are a set of ES6 shims (`core-js`), polyfills (`reflect-metadata`), loading libraries (`systemjs`), dependencies (`rxjs`), and helper tools (`zone.js`).

All of these packages are required for a number of good reasons:

- They ensure backward-compatibility between the new **ECMAScript v6 (ES6)** language features - used by most Angular 2 packages, and **ECMAScript v5 (ES5)**, which will be our compilation target.
- They make our code compatible with the most common/used web browsers, including older ones.
- They adopt a modern, module-based loading API to handle the required JS dependencies in an efficient way.
- They use Angular 2 features to their full extent.

Note

All this information can be quite confusing, especially for old-school JavaScript developers, but don't worry, we're going to come back to these topics later on.

As usual, all of these packages will be downloaded in the background by the IDE as soon as you **Save** the file. Eventually, you will have a local instance of Angular 2 available under the following folder:

```
<project_root>/node_modules/@angular/
```

It's worth noting that we have also added a new `script` node that will execute a post-install command against the `typings` package we just added.

Now we need to move these files to the `wwwroot` folder. We can achieve this by adding a dedicated task to our Gulp configuration file as follows (new lines are highlighted):

```
/*
This file in the main entry point for defining Gulp tasks and using
Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?
LinkId=518007
*/
var gulp = require('gulp'),
    gp_clean = require('gulp-clean),
    gp_concat = require('gulp-concat'),
    gp_sourcemaps = require('gulp-sourcemaps'),
    gp_typescript = require('gulp-typescript'),
    gp_uglify = require('gulp-uglify');

/// Define paths
var srcPaths = {
    app: ['Scripts/app/main.ts', 'Scripts/app/**/*.ts'],
    js: [
        'Scripts/js/**/*.js',
        'node_modules/core-js/client/shim.min.js',
```

```
        'node_modules/zone.js/dist/zone.js',

        'node_modules/reflect-metadata/Reflect.js',

        'node_modules/systemjs/dist/system.src.js',

        'node_modules/typescript/lib/typescript.js'

    ],

    js_angular: [

        'node_modules/@angular/**'

    ],

    js_rxjs: [

        'node_modules/rxjs/**'

    ]
};

var destPaths = {
    app: 'wwwroot/app/',
    js: 'wwwroot/js/',
    js_angular: 'wwwroot/js/@angular/',

    js_rxjs: 'wwwroot/js/rxjs/'
};

// Compile, minify and create sourcemaps all TypeScript files and
place them to wwwroot/app, together with their js.map files.
gulp.task('app', ['app_clean'], function () {
    return gulp.src(srcPaths.app)
        .pipe(gp_sourcemaps.init())

.pipe(gp_typescript(require('./tsconfig.json').compilerOptions))
        .pipe(gp_uglify({ mangle: false }))
         .pipe(gp_sourcemaps.write('/'))
        .pipe(gulp.dest(destPaths.app));
});

// Delete wwwroot/app contents
gulp.task('app_clean', function () {
    return gulp.src(destPaths.app + "*", { read: false })
        .pipe(gp_clean({ force: true }));
});

// Copy all JS files from external libraries to wwwroot/js
gulp.task('js', function () {
```

```
        gulp.src(srcPaths.js_angular)

            .pipe(gulp.dest(destPaths.js_angular));

        gulp.src(srcPaths.js_rxjs)

            .pipe(gulp.dest(destPaths.js_rxjs));
    return gulp.src(srcPaths.js)
        // .pipe(gp_uglify({ mangle: false })) // disable uglify
        // .pipe(gp_concat('all-js.min.js')) // disable concat
        .pipe(gulp.dest(destPaths.js));
});

// Delete wwwroot/js contents
gulp.task('js_clean', function () {
    return gulp.src(destPaths.js + "*", { read: false })
    .pipe(gp_clean({ force: true }));
});

// Watch specified files and define what to do upon file changes
gulp.task('watch', function () {
    gulp.watch([srcPaths.app, srcPaths.js], ['app', 'js']);
});

// Global cleanup task
gulp.task('cleanup', ['app_clean', 'js_clean']);

// Define the default task so it will launch all other tasks
gulp.task('default', ['app', 'js', 'watch']);
```

As you can see, it hasn't changed much, we just added a bunch of JS files that we need to copy from the `/node_modules/` folder into the `wwwroot/js/` folder, the same one we were already using to host third-party libraries. There's nothing odd there, NPM package files are external libraries, after all. For this very reason, it is also preferable to restrain ourselves from uglifying and/or concatenating them. It's also worth noting that we defined a separate, dedicated folder for **Angular 2** and **Rxjs** packages, since both of them are spanned into multiple files.

Adding Typings

Before moving forward, there is another thing we need to take care of. Since we plan to transpile our TypeScript code into ECMAScript5, we have added the `core-js` NPM module into our `package.json` file. In case you've never heard of it, let's just say that it happens to be a standard JavaScript library providing a great set of polyfills for ES6, which is precisely what we need.

Note

If you would like to know more about the `core-js`, here's the URL to the project's

official GitHub repository: https://github.com/zloirock/core-js.

The only problem is that it doesn't come with a proper TypeScript definition file, meaning that both of our TypeScript compilers, either Visual Studio's or Gulp's, won't be aware of its existence, thus throwing a fair amount of TS2304 (type not found) exceptions upon each build attempt.

The best thing we can do in order to fix that is to add the proper type definitions to our project. To do that, open the `package.json` file again and add the following (new lines are highlighted):

```
{
  "version": "1.0.0",
  "name": "opengamelistwebapp",
  "private": true,
  "dependencies": {
    "@angular/common": "2.0.0-rc.5",
    "@angular/compiler": "2.0.0-rc.5",
    "@angular/core": "2.0.0-rc.5",
    "@angular/http": "2.0.0-rc.5",
    "@angular/platform-browser": "2.0.0-rc.5",
    "@angular/platform-browser-dynamic": "2.0.0-rc.5",
    "@angular/upgrade": "2.0.0-rc.5",

    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.0-beta.6",
    "systemjs": "^0.19.37",
    "typings": "^1.3.2",
    "zone.js": "^0.6.12"
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "gulp-concat": "^2.6.0",
    "gulp-sourcemaps": "^1.6.0",
    "gulp-typescript": "^2.13.6",
    "gulp-uglify": "^2.0.0",
    "typescript": "^1.8.10"
  },
  "scripts": {

    "postinstall": "typings install dt~core-js --global"

  }
}
```

We can see that there are two new things here:

- A new NPM package called `typings`, which is a TypeScript type definition

manager. In other words, a tool we can use to retrieve type definitions from the web.

- A whole new `script` key containing a small command that will be executed during the post-install phase. This basically means that the script will trigger every time we change something within the `package.json` file, right after all the NPM modules are retrieved and installed (or removed).

As soon as we **Save** our `package.json` file, a new `typings` folder will be added to our **OpenGameListWebApp** project's root, containing the type definition file we need.

Tip

If typings didn't install successfully during the **Save**, try to use the **Restore Package** option by right-clicking on the project's **Dependencies** node. Another way is to use the command line to install the typings explicitly. To do this, navigate to the root folder of your app and press **CTRL+SHIFT**, then select **Open command window here**. In the command line, type the following command: **> npm run typings install** That should do the trick.

Using a CDN

At the time of writing, the only valid CDN hosting Angular 2 updated builds is npmcdn.com. If we want to use it, we can avoid updating our `gulpfile.js` file and wait until we are working on the `index.html` and `systemjs.config.js` files. Keep reading, as we'll get there shortly.

Upgrading the typescriptServices.js file

At the time of writing, adding Angular 2 via NPM would produce the following TypeScript compilation error:

Invalid module name in augmentation, module '../../Observable' cannot be found.

This is due to a known bug in the TypeScript version currently shipped with Visual Studio 2015. The most effective way to fix that is to replace this local VS2015 file:
```
C:\Program Files (x86)\Microsoft Visual Studio
14.0\Common7\IDE\CommonExtensions
\Microsoft\TypeScript\typescriptServices.js.
```

With the following remote file:

```
https://raw.githubusercontent.com/Microsoft/TypeScript/Fix8518/lib/t
```

Doing that will also fix a couple more errors we will most likely get in Chapter 3, *Angular 2 Components and Client-Side Routing. N*eedless to say, it's highly advisable to make a backup of the original file before replacing it.

Additional information regarding the bug is available through the following URLs: https://github.com/Microsoft/TypeScript/issues/8518 and https://github.com/Microsoft/TypeScript/pull/7507.

Setting up the client code

It's time to lay down the first bricks of our app, a working skeleton, used as a starting point for what we're going to do in the next chapters. It will consist of:

- An Angular 2 component file: `Scripts/app/app.component.ts`
- An Angular 2 module file: `Scripts/app/app.module.ts`
- An Angular 2 bootstrap file: `Scripts/app/main.ts`
- A module loader configuration file: `wwwroot/systemjs.config.js`
- An HTML file to wrap everything up: `wwwroot/index.html`

The component file

The component is the most basic and fundamental concept in Angular 2. Think of it like a class that controls a specific piece of a web page where we can either display some data to each user and/or respond to their feedbacks. We can say that our Angular 2 app will be almost entirely built upon multiple **Components** serving specific purposes-most of them will be reusable, others will be only used once. They can also either be as small as a few lines or they can result in a ridiculously long piece of code.

Luckily, our first component is quite simple. In the **Solution Explorer**, right-click on the `/Scripts/app/` folder and add a new `app.component.ts` file:
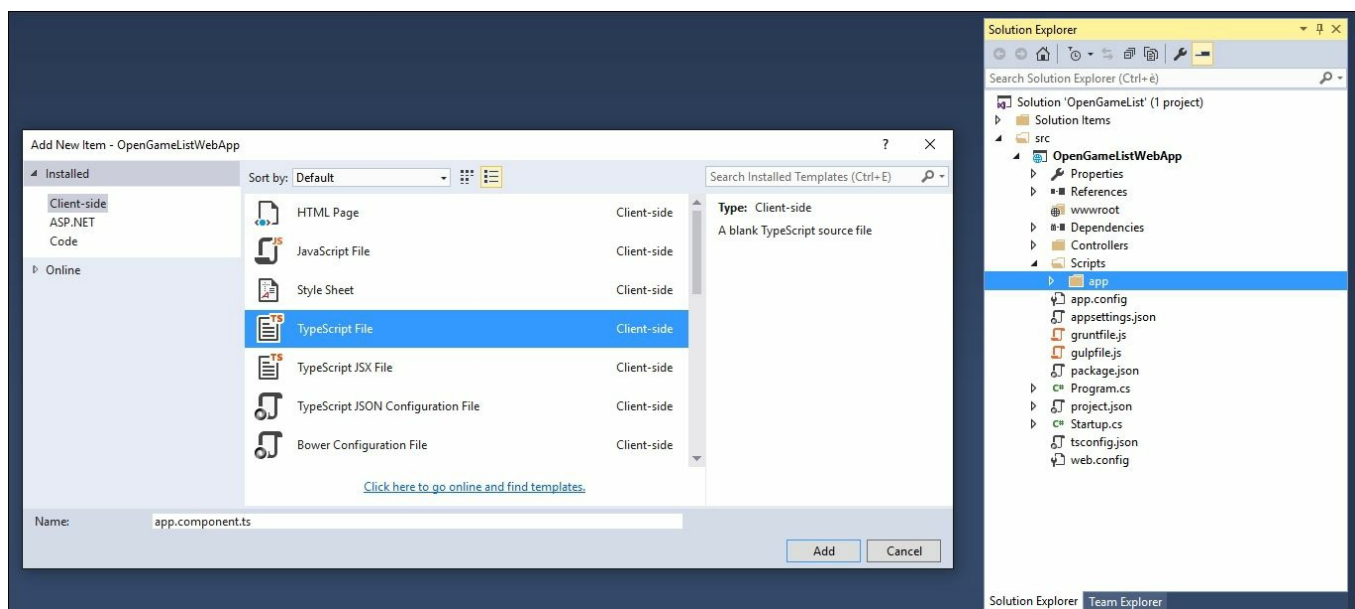
Once created, feed it with the following code:

```
import {Component} from "@angular/core";
```

```
@Component({
    selector: "opengamelist",
    template: `<h1>OpenGameList</h1><div>Work in progress...</div>`
})

export class AppComponent { }
```

That's it. Let's see what we just did here in detail:

- In line 1, we're importing the Component function from the Angular 2 main library, which is `@angular/core`. The Component function is what we need to define a Component's metadata for a class, which basically means being able to make Components, this is a required step for what we'll do next.
- In line 3-6, we're creating our first Component by applying the Component function to the class. In TypeScript, we do that by prefixing it with the `@` symbol and invoking it just above the component class. In other words, the `@Component` instruction tells Angular that this class is an Angular component. Notice that the `selector` and `template` fields are passed as a configuration object, as we will analyze them soon enough.
- In line 8, we're defining the `AppComponent` class. Notice the `export` keyword, which will allow us to `import` it from other components. The act of exporting makes our `app.component.js` file a module.

A word on components and modules

Angular 2 is a modular framework: this means that Angular 2 apps are also modular, as they consist of many files dedicated to a specific purpose. Typically, most application files export a single `Component`class, which is a class bundled with component metadata. Our `app.component.js` file, for example, exports the `AppComponent` class decorated with its component metadata, thus being a Module file exporting a Component class. We could also say that the Component is the content, while the Module is the file itself. We'll see how to import Components in the next chapter.

The module file

Angular Modules, also known as `NgModules`, have been introduced in **Angular 2 RC5**, which is the latest version at the time of writing, and are a great and powerful way to organize and bootstrap any Angular 2 application. They help developers to consolidate their own set of components, directives, and pipes into reusable blocks.

Note

If you've already played with previous versions of Angular 2, you will find the

following URL useful to understand the transition by checking out the following URL:

https://angular.io/docs/ts/latest/cookbook/rc4-to-rc5.html.

Every Angular 2 application since RC5 must have at least one module, which is conventionally called the root module and given the `AppModule` class name.

From the **Solution Explorer**, right-click on the `/Scripts/app/` folder, add a new `app.module.ts` file, and fill it with the following code:

```
///<reference path="../../typings/index.d.ts"/>
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {HttpModule} from "@angular/http";
import "rxjs/Rx";

import {AppComponent} from "./app.component";

@NgModule({
    // directives, components, and pipes
    declarations: [
        AppComponent
    ],
    // modules
    imports: [
        BrowserModule,
        HttpModule
    ],
    // providers
    providers: [
    ],
    bootstrap: [
        AppComponent
    ]
})
export class AppModule { }
```

Again, let's take a look at what we just wrote:

- In line 1, we added a reference to the type definitions we fetched during the previous section to ensure our TypeScript compiler(s) could find it. Notice that, if we're using a CDN or a pre-compiled version of Angular 2, we could (and should) definitely remove this line.
- In lines 2-4, we import the basic Angular 2 modules that we will need right from the start.
- In line 5, we import the `rxjs` library definition file(s), which will be useful to compile the Angular 2 libraries.

- In line 7, right after the first empty line, we import the application root component that we just wrote.
- In line 9, we declare our root `NgModule`. As we can see it consists in an array of named arrays, each one containing a set of Angular 2 objects that serves a common purpose: directives, components, pipes, modules, and providers. The last one of them contains the component(s) we want to bootstrap, which in our case is the `AppComponent` one.

Working with the root module only is a very viable approach until the Angular 2 app grows to a certain size. When it becomes bigger, it will be more practical to refactor it into a number of feature modules, each one of them grouping together a set of related tasks.

The bootstrap file

Now that we have our root module, we need to bootstrap it. From the **Solution Explorer**, right-click on the `/Scripts/app/` folder and add a new `main.ts` file, then fill it with the following code:

```
import {platformBrowserDynamic} from "@angular/platform-browser-dynamic";
import {AppModule} from "./app.module";

platformBrowserDynamic().bootstrapModule(AppModule);
```

Now we're just missing an entry point to load with the browser. Let's add it right now.

The module loader configuration file

In this application, we're going to use the **SystemJS** module loader library to load our application and all the required Angular 2 modules and dependencies. In order to do that, we have to add a `systemjs.config.js` file to the `wwwroot` folder and define a number of configuration rules within it, as follows:

```
(function (global) {
    // map tells the System loader where to look for things
    var map = {
        'app': 'app', // our application files
        '@angular': 'js/@angular', // angular2 packages
        'rxjs': 'js/rxjs' // Rxjs package
    };

    // packages tells the System loader which filename and/or
extensions to look for by default (when none are specified)
    var packages = {
        'app': { main: 'main.js', defaultExtension: 'js' },
```

```
        'rxjs': { defaultExtension: 'js' }
    };

    // configure @angular packages
    var ngPackageNames = [
        'common',
        'compiler',
        'core',
        'http',
        'platform-browser',
        'platform-browser-dynamic',
        'upgrade',
    ];

    function packIndex(pkgName) {
        packages['@angular/' + pkgName] = { main: 'index.js',
defaultExtension: 'js' };
    }

    function packUmd(pkgName) {
        packages['@angular/' + pkgName] = { main: '/bundles/' +
pkgName + '.umd.js', defaultExtension: 'js' };
    };

    var setPackageConfig = System.packageWithIndex ? packIndex :
packUmd;
    ngPackageNames.forEach(setPackageConfig);
    var config = {
        map: map,
        packages: packages
    }
    System.config(config);
})(this);
```

The code is pretty much self-documented with inline comments, yet it could be useful to highlight the most relevant tasks:

- The map variable will host the three dynamic packages we're using SystemJS for, all of them relative to /wwwroot/: app for our application; js/@angular for Angular 2 and js/rxjs for Rxjs.
- The packages variable will set the default filename and/or extension values for each package. These will be used whenever we define an import statement without specifying them.
- The rest of the file is dedicated to dynamically loading the Angular 2 built-in packages.

If we want to use a CDN instead of relying upon the local JS folder, we only need to perform a minor update within the previous code, in the map section, as follows:

```
    var map = {
        'app': 'app', // our application files
        '@angular': 'js/@angular', // angular2 packages
        'rxjs': 'https://npmcdn.com/rxjs@5.0.0-beta.6' // Rxjs
package (CDN)
    };
```

And also within the two module loader functions:

```
    function packIndex(pkgName) {
        packages['https://npmcdn.com/' + pkgName] = { main:
'index.js', defaultExtension: 'js' };
    }

    function packUmd(pkgName) {
        packages['https://npmcdn.com/' + pkgName] = { main:
'/bundles/' + pkgName + '.umd.js', defaultExtension: 'js' };
    };
```

Note

For further info regarding SystemJS and its Configuration API, including advanced options, we strongly suggest reading the official documentation on the project's GitHub page:

https://github.com/systemjs/systemjs and https://github.com/systemjs/systemjs/blob/master/docs/config-api.md.

Why use a dynamic module loader?

Before going further, it might be useful to explain why we worked so hard with a module loader instead of adding all the relevant JS files into the `index.html` file right from the start.

To keep it simple, we did it because it's the only way to efficiently handle any modern JavaScript modular system such as `Angular 2`, `Rxjs` and also all applications based upon them, including the one we're working on right now.

What's a modular system exactly? It's nothing more than a package, library, or application split into a series of smaller files which depend on each other using reference statements such as `import`, `require`, and so on. ASP.NET, Java, Python, and most compilation-based languages have it. That's not the case with script-based languages such as PHP and JavaScript: they that are doomed to pre-load everything in the memory before being able to determine whether they'll be using it or not. All of these changes, with the introduction of **ECMAScript 6 (ES6)**, bring a fully-featured module and dependency management solution for JavaScript. **SystemJS** basically acts as an ES6-polyfill for browsers that don't support it already, allowing us to get

that module system working in modern browsers. Since both `Angular 2` and `Rxjs` leverage that dynamic-loading approach, implementing it within our project will result in a huge performance gain.

Note

Keep in mind that **SystemJS** is not the only choice we have to load Angular 2 packages: there are other good choices out there, for example the popular module bundler known as **webpack**. Should we want to use that instead, here's a great guide for doing that:

https://angular.io/docs/ts/latest/guide/webpack.html.

The index.html file

The HTML file serves two main purposes: being an entry point for the browser so it can load the client-script files and execute the application, and laying out the DOM structure used by Angular 2 to display it. In the **Solution Explorer**, right-click on the `wwwroot` folder and add a new `index.html` file, then fill it with the following code:

```html
<!DOCTYPE html>
<html>
<head>
    <base href="/">
    <title>OpenGameList</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Step 1. Load libraries -->
    <!-- Polyfill(s) for older browsers -->
    <script src="js/shim.min.js"></script>
    <script src="js/zone.js"></script>
    <script src="js/Reflect.js"></script>
    <script src="js/system.src.js"></script>

    <!-- Step 2. Configure SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err);
});
    </script>
</head>

<!-- Step 3. Display the application -->
<body>
    <!-- Application PlaceHolder -->
    <opengamelist>Loading...</opengamelist>
</body>
</html>
```

After adding the `index.html` file, we need to set it as the main entry point when executing our app in the Debug mode. In the **Solution Explorer**, right-click on the project node and select **Properties**, then switch to the **Debug** tab and change the **Launch URL** parameter accordingly.
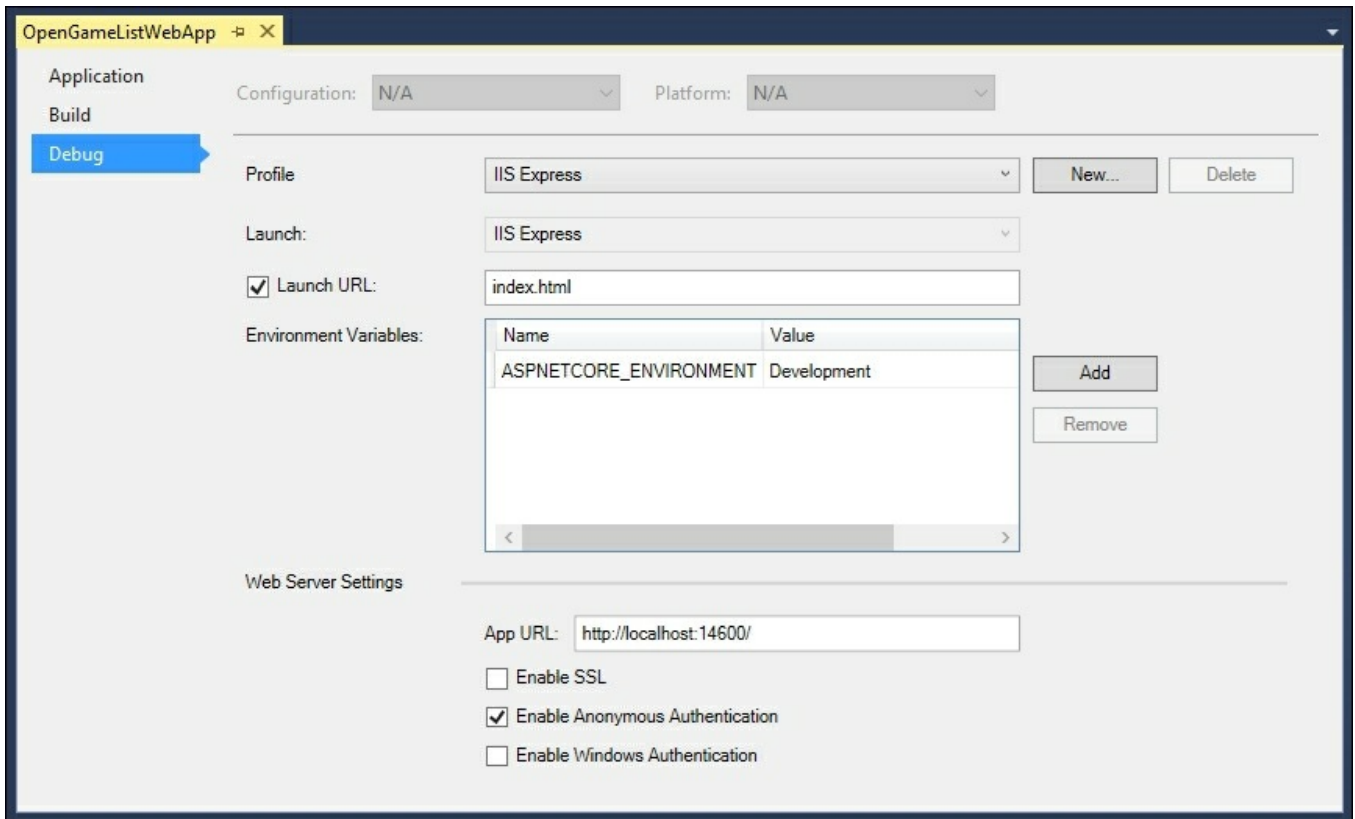
Note

While we're here, we might also take the chance to set the HTTP port that will be used by the local web server during development by changing the numeric part of the **App URL** textbox value. As we can see in the preceding screenshot, we're going to use **14600** throughout the whole book.

If we want to use a CDN instead of local JS files, we can replace the `<script>` elements right below the Step 1 comment with the following:

```
    <script src="https://cdnjs.cloudflare.com/ajax/libs/core-
js/2.4.1/shim.min.js"></script>
    <script src="https://npmcdn.com/zone.js@0.6.12"></script>
    <script src="https://npmcdn.com/reflect-metadata@0.1.3">
</script>
    <script src="https://cdnjs.cloudflare.com/ajax/
libs/systemjs/0.19.37/system.js"></script>
```

These are the latest versions at the time of writing. Be sure to replace them with the most recent Angular 2-compatible versions.

First run

Now that we've set everything up, it's time to take a look at where we're doing. The first thing we should do is to turn our task runner on:

1. Expand the root node, right-click to the **default** task, and hit **Run**.
2. From the **Solution Explorer**, right-click on your `gulpfile.js` file and select **Task Runner Explorer**.
3. If you did everything correctly, the Task Runner will silently execute its tasks, keeping the one named `watch` active in background.
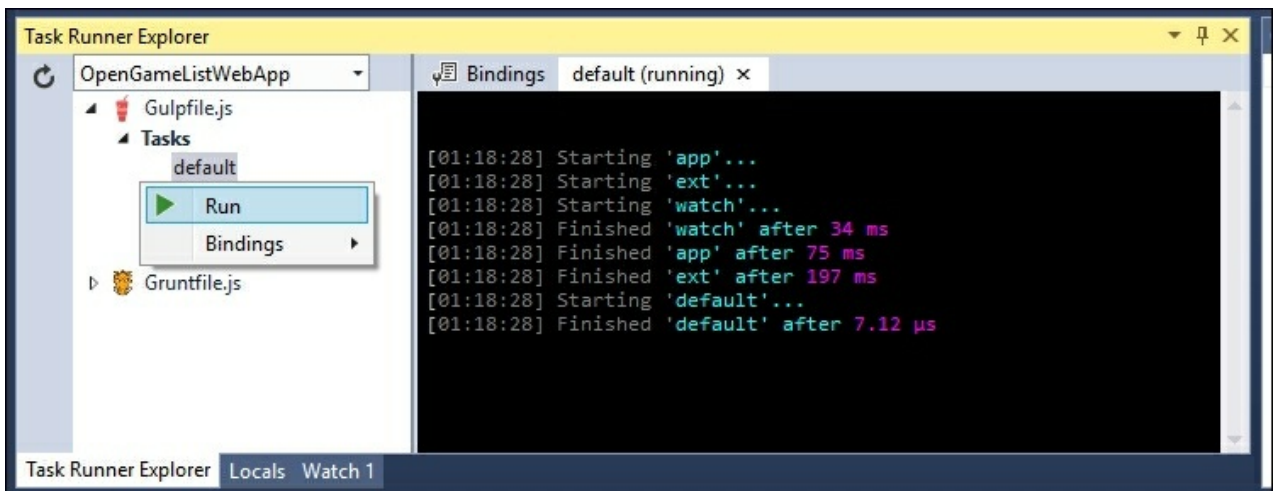
4. Right after that we can hit *F5* (or left-click the **Start Debugging** button) and test our still rather minimalistic, yet fully functional, home screen.
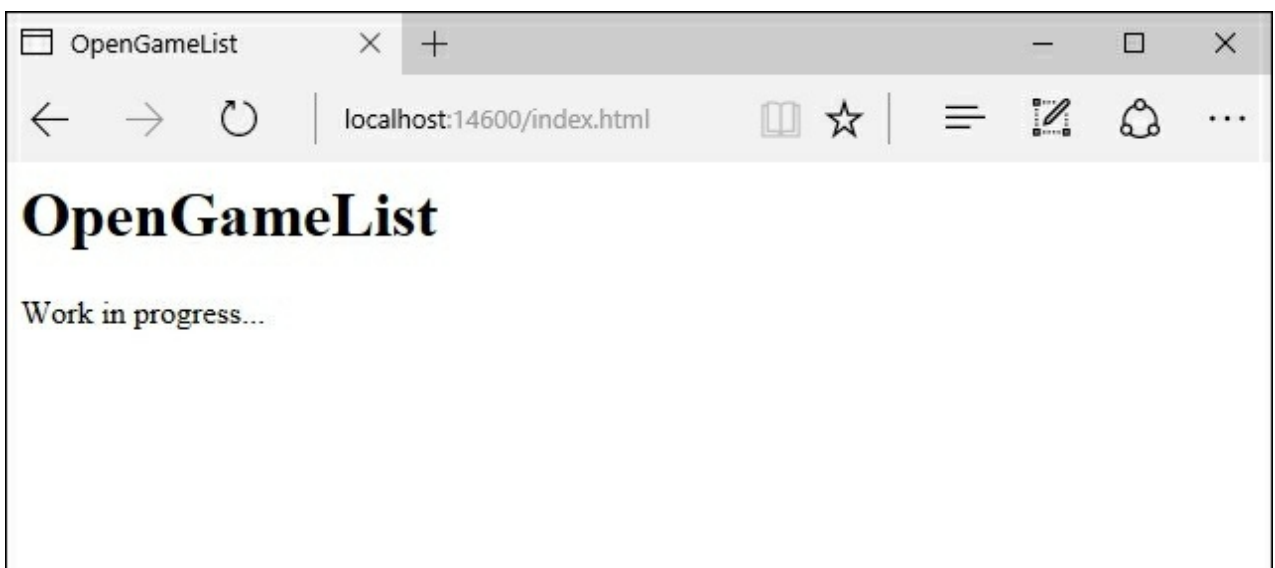
If things are working as they should, we will see something very close to the preceding image. That's pretty good. Before going further, let's check if our task runner and its plugins are also working as they should.

Back in Visual Studio, open the `/Scripts/app/app.component.ts` file and change the `<div>` content with the highlighted text as follows:

```
import {Component} from "@angular/core";

@Component({
    selector: "opengamelist",
    template: `<h1>OpenGameList</h1><div>...the best is yet to
come!</div>`
})

export class AppComponent { }
```

After you're done, hit *CTRL + S* or **Save**, then move back to the browser and issue a page refresh by hitting *F5* to see if the task runner did its job. If it did, you should see something like the following:
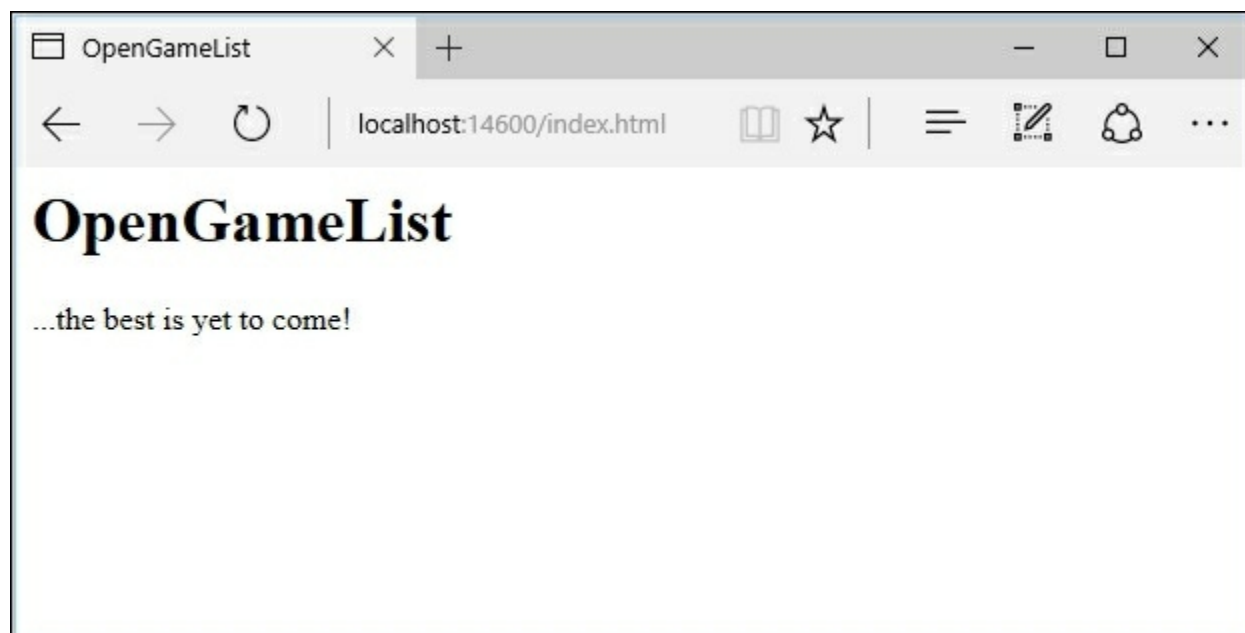
So far so good, we have just set up a working skeleton of what's about to come.

Caching issues

If you're not seeing this, and the page is still showing the Work in progress..., phrase, chances are that you have caching issues. This is quite a common issue, since our client code relies upon static files (such as `index.html`) which are served by default with a set of `cache-control` HTTP headers to ensure a proper client-side cache. This is usually great for production, but it can be quite annoying while our app is in the development stage. If we want to fix it, we need to change the default caching behavior for static files.

If we were developing an ASP.NET 4 web application, we could do that by adding some lines to our main application's `web.config` file such as the following:

```
<caching enabled="false" />
<staticContent>
  <clientCache cacheControlMode="DisableCache" />
</staticContent>
<httpProtocol>
  <customHeaders>
    <add name="Cache-Control" value="no-cache, no-store" />
    <add name="Pragma" value="no-cache" />
    <add name="Expires" value="-1" />
  </customHeaders>
</httpProtocol>
```

And that should be it.

However, that's not the case. The new ASP.NET Core's configuration system has been re-architected from scratch and is now quite different from the previous versions. The most important consequence of this is that XML configuration files such as `web.config` , together with the whole `System.Configuration` namespace, are not part of the new pattern and shouldn't be used anymore.

The new ASP.NET Core configuration pattern

The new configuration model is based upon key/value settings that can be retrieved from a wide variety of sources, including, and mostly being, Json files. Once retrieved, they can be accessed within our code in a strongly-typed fashion. We can take a look at the new pattern by watching a couple of lines contained within the `Startup` class constructor, which is contained in the `Startup.cs` file (relevant lines are highlighted):

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true)

        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

And also the `appsettings.json` file they refer to:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Now that we understand the basics, let's see how we can solve that caching issue by taking advantage of the new configuration model.

The first thing to do is to understand how we can modify the default HTTP headers for static files. As a matter of fact, we can do that by adding a custom set of options to the `app.UseDefaultFiles()` method we added to the `Startup` class earlier. In order to do that, open the `Startup.cs` and change that part of code in the following way (new/modified lines are highlighted):

```
// Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
app.UseDefaultFiles();
```

```
// Serve static files (html, css, js, images & more). See also the
following URL:
// https://docs.asp.net/en/latest/fundamentals/static-files.html
for further reference.
app.UseStaticFiles(new StaticFileOptions()
{

    OnPrepareResponse = (context) =>

    {

        // Disable caching for all static files.

        context.Context.Response.Headers["Cache-Control"] = "no-
cache, no-store";

        context.Context.Response.Headers["Pragma"] = "no-cache";

        context.Context.Response.Headers["Expires"] = "-1";

    }
});
```

That wasn't hard at all. However, we're not done yet, now that we've learned how to change the default behavior, we just need to change these static values with some convenient references pointing to the `appsettings.json` file.

To do that, we can add the following key/value section to the `appsettings.json` file in the following way (new lines are highlighted):

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },

  "StaticFiles": {

    "Headers": {

      "Cache-Control": "no-cache, no-store",

      "Pragma": "no-cache",

      "Expires": "-1"
```

```
        }

    }
}
```

And then change the preceding `Startup.cs` code accordingly (modified lines are highlighted):

```
// Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
app.UseDefaultFiles();

// Serve static files (html, css, js, images & more). See also the
following URL:
// https://docs.asp.net/en/latest/fundamentals/static-files.html
for further reference.
app.UseStaticFiles(new StaticFileOptions()
{

    OnPrepareResponse = (context) =>

    {

        // Disable caching for all static files.

        context.Context.Response.Headers["Cache-Control"] =
        Configuration["StaticFiles:Headers:Cache-Control"];

        context.Context.Response.Headers["Pragma"] =
        Configuration["StaticFiles:Headers:Pragma"];

        context.Context.Response.Headers["Expires"] =
        Configuration["StaticFiles:Headers:Expires"];

    }
});
```

That's about it. Learning how to use this pattern is strongly advisable, as it's a great and effective way to properly configure our application's settings.

A faster alternative using scaffolding tools

In the latest few paragraphs, we basically built our very own server-side and client-side environment by configuring everything manually. Truth be told, what we just did is neither the quickest nor the most effective way to start a project involving a client-side framework such as Angular 2, as a matter of fact, we could achieve the same results in a fraction of the time we just spent by using the ASP.NET Core project scaffolding tools such as **Yeoman**, which are available through NPM, together with a proper generator template such as `aspnetcore-spa`.

Not only will these tools generate a fully-functional ASP.NET Core and Angular 2 boilerplate, much like the one we just made, they'll also take care of a number of issues beforehand that we would otherwise have to manually set up later. These include server-side pre-rendering, efficient cache management, optimized production builds, and much more. On top of that, they can be easily enhanced with other helper packages such as **WebPack** and **JavaScriptServices**, a great set of tools and middlewares for building SPA projects with ASP.NET Core and Angular 2.

The reasons to say no (not yet)

All of this sounds so exciting that we may well ask ourselves why in the world didn't we go for that?

The reason is simple, what we just made, together with a lot of things we'll assemble during the course of the following chapters, is also part of our learning process. In other words, this is why we're reading this book instead of just copying some working samples from a bunch of web pages, or GitHub projects, and putting them together. This doesn't mean we don't have to use **Yeoman**, as a matter of fact, we will be using it, together with a lot of other great tools that will speed up our development and help us to write better code. Eventually, just not on our first run, that's for sure. We don't want our journey to be spoiled, do we?

A quick scaffolding sample

However, it's also true that taking a look at these tools, and eventually being able to properly use them, can easily be seen as part of our learning process. If you want to do that, you need to have Node.js installed, unless you prefer to use what's bundled with Visual Studio 2015.

Open a command prompt with administrative rights and navigate through theNode.js installation folder. If you're using the one shipped with VS2015 it should be something like the following: `C:\Program Files (x86)\Microsoft Visual Studio 14.0\Web\External`.

If you went for a new install, it should be the following (for x64 architectures): `C:\Program Files\Nodejs`.

Once there, type the following command and execute it to update NPM to the latest version:

```
npm install npm -g
```

Then type and execute the following to install **Yeoman** together with the `aspnetcore-spa` generator:

```
npm install -g yo generator-aspnetcore-spa
```

Finally, type and execute the following commands to create your application's entry point:

```
cd C:\Projects\Your-SPA-folder\
yo aspnetcore-spa
```

That's about it. For further reference regarding how to use **Yeoman**, it's strongly advisable to take a look at the official documentation at the following URL: http://yeoman.io/learning/.

If you're bold enough to also install WebPack and JavaScriptServices, you should definitely read the ASP.NET Core JavaScript Services official documentation, written by *Steve Sanderson*, who is the author of the JavaScriptServices project and the **KnockoutJs** library, as well as being a Microsoft employee: https://github.com/aspnet/JavaScriptServices#javascriptservices.

Note

Before moving forward, it's very important to understand the fact that we will not

cover these packages throughout the book. We will take it for granted that we don't have these features available, so we will often spend some time designing and building our own implementation. That's why we don't suggest that you do that during the first reading.

References

- *Native Web Apps*, Henrik Joreteg, 2015.
- *Manifesto for Agile Software Development*, Kent Beck, Mike Beedle and many others, 2001

Suggested topics

SCRUM, Extreme Programming, MVC and MVVM architectural patterns, ASP.NET Core, .NET Core, Roslyn, CoreCLR, RyuJIT, task runner, Gulp, Grunt, NuGet, NPM, ECMAScript 6, Bower, SystemJS, Rxjs, Cache-Control, HTTP Headers.

Summary

So far, so good, we have just set up a working skeleton of what's about to come. Before moving further, let's quickly recap what we just did in this first chapter.

We briefly described our platforms of choice, ASP.NET Core and Angular 2, and acknowledged their combined potential in the process of building a modern web application. Then we chose a NWA with a single-page application approach as the ideal field of choice for testing what our frameworks are able to do (and how to do it).

In an attempt to reproduce a realistic production-case scenario, we also went through the most common SPA features: first from a technical point of view, then by putting us in the shoes of a typical product owner and trying to enumerate his expectations. We also made a quick list of everything we need to put together a potentially shippable product featuring all the expected goodies.

Eventually, we spent an appropriate amount of time setting up our development environment. This included installing package managers, choosing a suitable client-side framework, introducing task runners and configuring both ASP.NET Core and Angular 2.

Finally, we performed a quick test to see that all the bricks we'd lain were in place and ready to hold their ground against what's coming next, setting up a request-response cycle, building our very first controller, defining efficient routing strategies, and more.

Chapter 2. ASP.NET Controllers and Server-Side Routes

Now that we have our skeleton up and running, it's time to explore the client-server interaction capabilities of our frameworks. To put it in other words, we need to understand how Angular 2 will be able to fetch data from ASP.NET Core using its brand new Core Web API structure. We won't be worrying about how will ASP.NET Core retrieve these data, be it from session objects, data stores, DBMS, or any possible data source. All of that will come later on. For now, we'll just put together some sample static data in order to understand how to pass them back and forth by using a well-structured, highly configurable, and viable interface.

Data flow

As you might already know, a Native Web App following the single-page application approach will roughly handle the client-server communication in the following way:
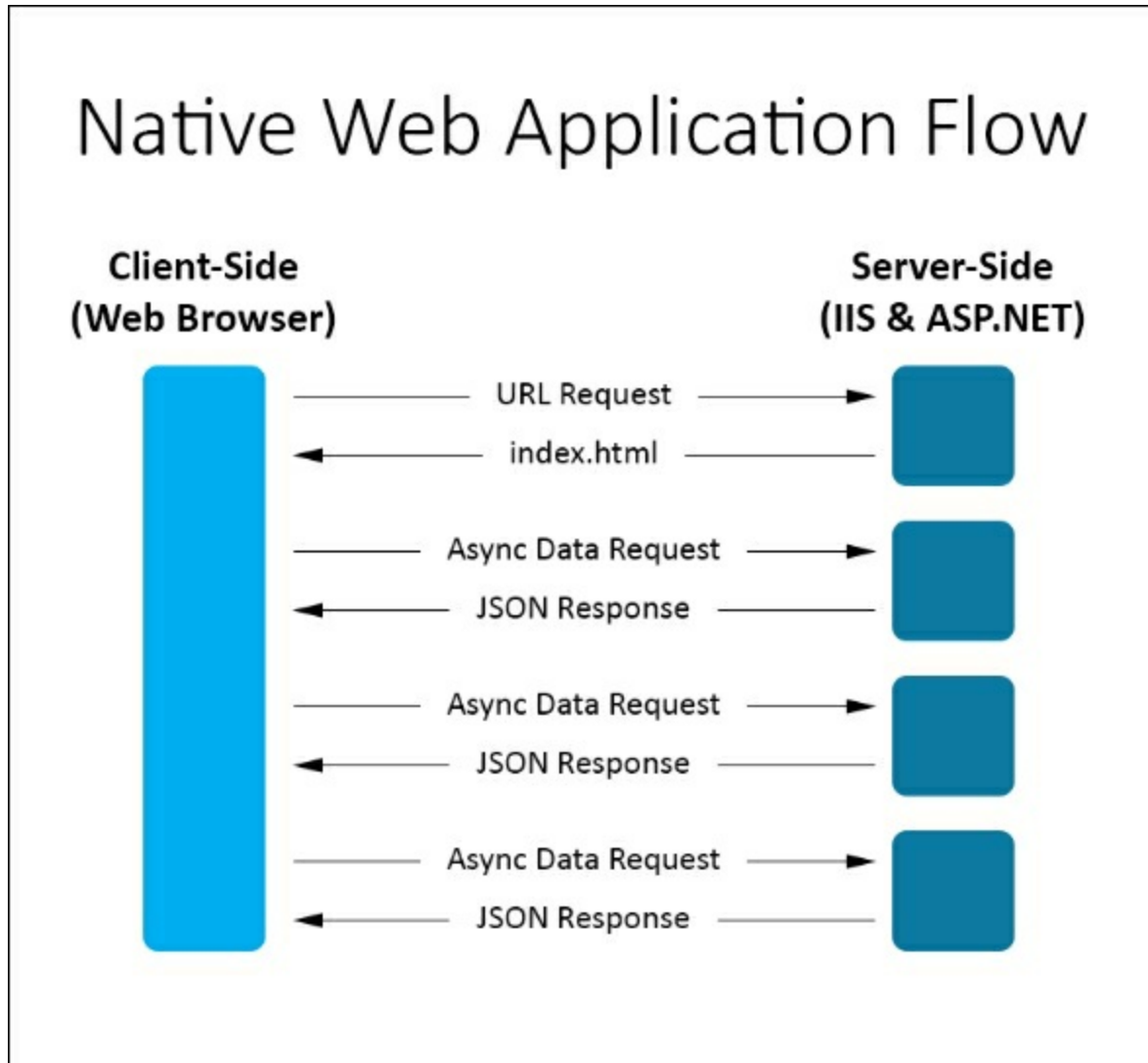
In case you're wondering about what these **Async Data Requests** actually are, the answer is simple: everything, as long as it needs to retrieve data from the server, which is something that most of the common user interactions will normally do, including (yet not limited to): *pressing a button* to show more data or to edit/delete something, *following a link* to another app view, *submitting a form*, and so on.

In other words, unless the task is so trivial, or it involves a minimal amount of data, that the client can entirely handle it, meaning that it already has everything that they need. Examples of such tasks are as follows: show/hide element toggles, in-page navigation elements (such as internal anchors), and any temporary job requiring a

confirmation or *save* button to be pressed before actually being processed.

The preceding diagram shows, in a nutshell, what we're going to do: define and implement a pattern to serve these JSON-based, server-side responses that our application will need to handle the upcoming requests. Since we've chosen a strong, data-driven application pattern such as a Wiki, we'll surely need to put together a bunch of common CRUD-based requests revolving around a defined object which will represent our entries. For the sake of simplicity, we'll call it "item" from now on.

These requests will address some common CMS-inspired tasks such as displaying a list of items, viewing/editing the selected item's details, handling filters and text-based search queries, and also deleting an item.

Before going further, let's have a more detailed look at what happens between any of these **Data Requests** issued by the client and **JSON Responses** sent out by the server, that is, what's usually called the request/response flow:

Request/Response Flow
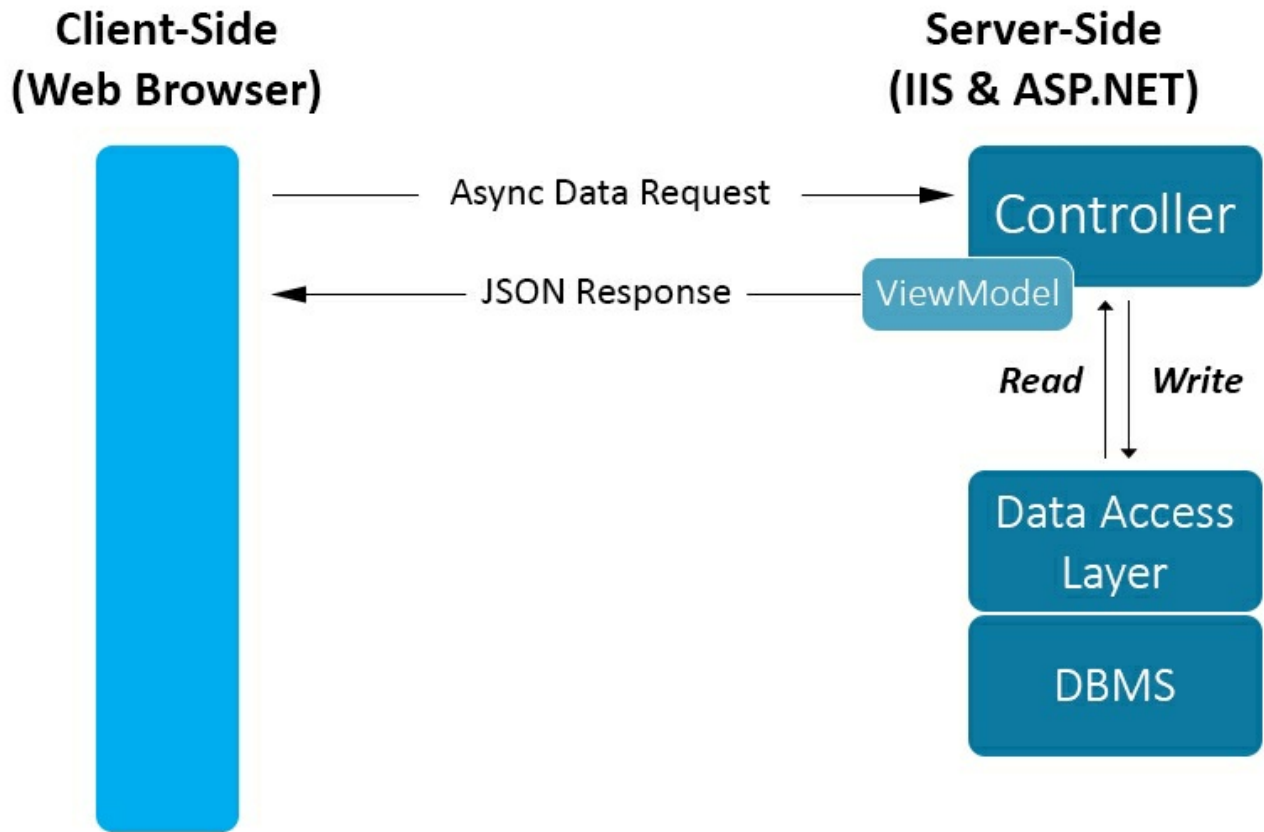
As we can see, in order to respond to any client-issued **Data Request** we need to build a *server-side* Core Web API **Controller** featuring the following capabilities:

- **Read** and/or **Write** data using the **Data Access Layer**
- Organize this data in a suitable JSON-serializable **ViewModel**
- Serialize the **ViewModel** and send it to the client as a **Response**

Based on these points, we could easily conclude that the **ViewModel** is the key item here. That's not always correct: it could or couldn't be the case, depending on the project we're building. To better clarify that, before going further, it could be useful to spend a couple of words on the **ViewModel** object itself.

The role of the ViewModel

We all know that a **ViewModel** is a container-type class, which represents only the data we want to display on our web page. In any standard MVC-based ASP.NET application, the **ViewModel** is instantiated by the **Controller** in response to a GET request using the data fetched from the model. Once built, the **ViewModel** is passed to the **View**, where it's used to populate the page contents/input fields.

The main reason for building a **ViewModel** instead of directly passing the model entities is that it only represents the data that we want to use and nothing else. All the unnecessary properties that are in the model domain object will be left out, keeping the data transfer as lightweight as possible. Another advantage is the additional security it gives since we can protect any field from being serialized and passed through the HTTP channel.

In a standard Web API context, where the data is passed using RESTful conventions via serialized formats such as JSON or XML, the **ViewModel** could be easily replaced by a JSON-serializable dynamic object created on the fly, such as this:

```
var response = new {
    Id = "1",
    Title = "The title",
    Description = "The description"
};
```

This approach is often viable for small or sample projects, where creating one (or many) **ViewModel** classes could be a waste of time. However, that's not our case. Conversely, our project will greatly benefit from having a well-defined, strongly typed **ViewModel** structure, even if they will all be eventually converted into JSON strings.

Our first Controller

Now that we have a clear vision of the request/response flow and its main actors, we can start building something up. Let's start with the *Welcome View*, which is the first page that any user will see upon connecting to our Native Web App. This is something that in a standard web application would be called *Home Page*, but since we are following a single-page application approach, that name isn't appropriate. After all, we're not going to have more than one page.

In most wikis, the *Welcome View/Home Page* contains a brief text explaining the context/topic of the project and then one or more lists of items ordered and/or filtered in various ways, such as:

- The last inserted ones (most recent first)
- The most relevant/visited ones (most viewed first)
- Some random items (in random order)

Let's try to do something like that. This will be our master plan for a suitable *Welcome View*:
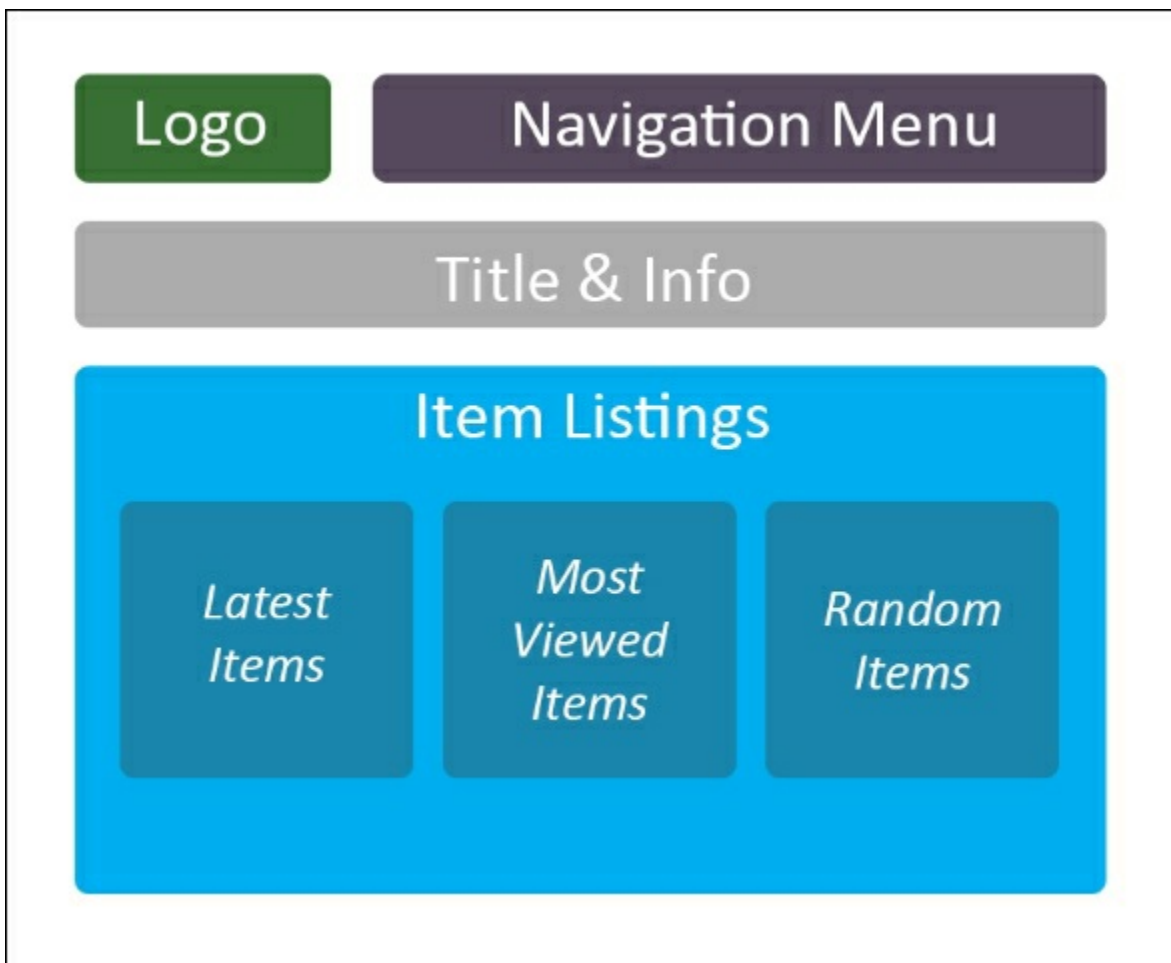
In order to do that, we're going to need the following set of API calls:

- `api/items/GetLatest` (to fetch the last inserted items)
- `api/items/GetMostViewed` (to fetch the last inserted items)
- `api/items/GetRandom` (to fetch the last inserted items)

As we can see, all of them will be returning a list of items ordered by a well-defined logic. That's why, before working on them, we should provide ourselves with a suitable **ViewModel**.

The ItemViewModel

One of the biggest advantages in building a Native Web App using ASP.NET and Angular 2 is that we can start writing our code without worrying too much about data sources because they will come later and only after we're sure about what we really need. This is not a requirement either. You are also free to start with your data source for a number of the following good reasons:

- You already have a clear idea of what you'll need
- You already have your entity set(s) and/or a defined/populated data structure to work with
- You're used to starting with the data and subsequently moving to the GUI

All the preceding reasons are perfectly fine. You won't ever get fired for doing that. Yet, the chance to start with the frontend might help you a lot if you're still unsure about what your application will look like, either in terms of GUI and/or data. In building this Native Web App, we'll take advantage of that. Hence, why we'll start defining our `ItemViewModel` instead of creating its data source and entity class?

From **Solution Explorer**, right-click to the project root node and **Add** a **New Folder** named `ViewModels`. Once created, right-click on it and **Add** a **New Item**. From the server-side elements, pick a standard **Class**, name it `ItemViewModel.cs`, and hit the **Add** button. Then, type in the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Threading.Tasks;
using Newtonsoft.Json;

namespace OpenGameListWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class ItemViewModel
    {
        #region Constructor
        public ItemViewModel()
        {

        }
        #endregion Constructor

        #region Properties
        public int Id { get; set; }
        public string Title { get; set; }
```

```csharp
        public string Description { get; set; }
        public string Text { get; set; }
        public string Notes { get; set; }
        [DefaultValue(0)]
        public int Type { get; set; }
        [DefaultValue(0)]
        public int Flags { get; set; }
        public string UserId { get; set; }
        [JsonIgnore]
        public int ViewCount { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime LastModifiedDate { get; set; }
        #endregion Properties
    }
}
```

As we can see, we're defining a rather complex class. This isn't something we could easily handle using a dynamic object created on the fly; hence why we're using a `ViewModel` instead?

Remember the Newtonsoft's `Json.NET`package we installed in <u>Chapter 1</u>, *Getting Ready*, using NuGet? We'll start using it in this class by including its namespace in line 6 and decorating our newly created `Item` class with a `JsonObject` attribute in line 10. That attribute can be used to set a list of behaviors of the `JsonSerializer`/`JsonDeserializer` methods, overriding the default ones. Notice that we used `MemberSerialization.OptOut`, meaning that any field will be serialized into JSON unless being decorated by an explicit `JsonIgnoreAttribute` or `NonSerializedAttribute`. We're making this choice because we're going to need most of our ViewModel's properties serialized, as we'll be seeing soon enough.

The ItemsController

Now that we have our `ItemViewModel` class, let's use it to return some server-side data. From your project's root node, open the `/Controllers/` folder, right-click on it, select **Add | New Item**, then create a **Web API Controller Class**, name it `ItemsController.cs`, and click on the **Add** button to create it.

The controller will be created with a bunch of sample methods, they are identical to those present in the default `ValuesController.cs`, which we already tested in [Chapter 1](), *Getting Ready*, and hence, we don't need to keep them. Delete the entire file content and replace it with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using OpenGameListWebApp.ViewModels;
using Newtonsoft.Json;

namespace OpenGameListWebApp.Controllers
{
    [Route("api/[controller]")]
    public class ItemsController : Controller
    {
        // GET api/items/GetLatest/5
        [HttpGet("GetLatest/{num}")]
        public JsonResult GetLatest(int num)
        {
            var arr = new List<ItemViewModel>();
            for (int i = 1; i <= num; i++) arr.Add(new
ItemViewModel() {
                Id = i,
                Title = String.Format("Item {0} Title", i),
                Description = String.Format("Item {0} Description",
i)
            });
            var settings = new JsonSerializerSettings() {
                Formatting = Formatting.Indented
            };
            return new JsonResult(arr, settings);
        }
    }
}
```

This controller will be in charge of all item-related operations within our app.

As we can see, we started defining a `GetLatest` method accepting a single integer

parameter value. The method accepts any GET request using the custom routing rules configured via the `HttpGetAttribute`, this approach is called **attribute routing**, and we'll be digging more into it later in this chapter. For now, let's stick to the code inside the method itself.
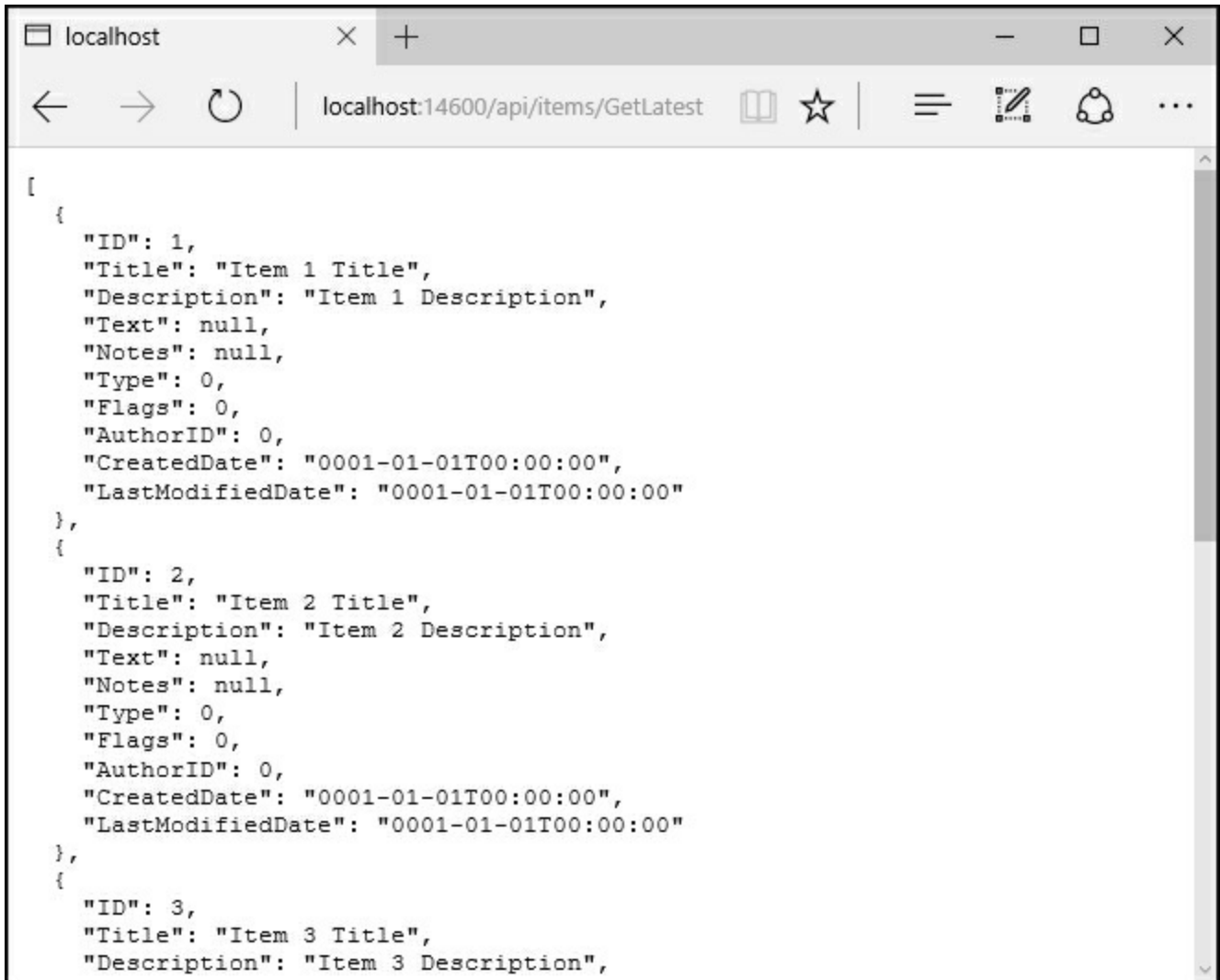
The behavior is really simple since we don't have a data source we're basically mocking a bunch of `ItemViewModel` objects. Notice that, although it's just a fake response, we're doing it in a structured and credible way, respecting the number of items issued by the request and also providing different content for each one of them.

It's also worth noticing that we're using a `JsonResult` return type, which is the best thing we can do as long as we're working with `ViewModel` classes featuring the `JsonObject` attribute provided by the `Json.NET` framework. That's definitely better than returning plain `string` or `IEnumerable<string>` types, as it will automatically take care of serializing the outcome and setting the appropriate response headers (`Content-Type`, `charset`, and so on).

Let's try our controller by running our app in the debug mode: select **Debug | Start Debugging** from the main menu or press *F5* . The default browser should open, pointing to the `index.html` page because we set it as the `Launch URL` in our project's debug properties. To test our brand new API Controller, we need to manually replace the URL with the following:

```
/api/items/GetLatest/5
```

If we have done everything correctly, it will show something like the following:

```
[
  {
    "ID": 1,
    "Title": "Item 1 Title",
    "Description": "Item 1 Description",
    "Text": null,
    "Notes": null,
    "Type": 0,
    "Flags": 0,
    "AuthorID": 0,
    "CreatedDate": "0001-01-01T00:00:00",
    "LastModifiedDate": "0001-01-01T00:00:00"
  },
  {
    "ID": 2,
    "Title": "Item 2 Title",
    "Description": "Item 2 Description",
    "Text": null,
    "Notes": null,
    "Type": 0,
    "Flags": 0,
    "AuthorID": 0,
    "CreatedDate": "0001-01-01T00:00:00",
    "LastModifiedDate": "0001-01-01T00:00:00"
  },
  {
    "ID": 3,
    "Title": "Item 3 Title",
    "Description": "Item 3 Description",
```

Our first controller is up and running. As you can see, the `ViewCount` property is not present in the JSON-serialized output: that's by design since it has been flagged with the `JsonIgnore` attribute, meaning that we're explicitly opting it out.

Now that we've seen that it works, we can come back to the routing aspect of what we just did: since it is a major topic, it's well worth some of our time.

Understanding routes

In [Chapter 1](), *Getting Ready*, we acknowledged the fact that the ASP.NET Core pipeline has been completely rewritten in order to merge the MVC and WebAPI modules into a single, lightweight framework to handle both worlds. Although this certainly is a good thing, it comes with the usual downside that we need to learn a lot of new stuff. Handling routes is a perfect example of this, as the new approach defines some major breaking changes from the past.

Defining routing

The first thing we should do is give out a proper definition of what routing actually is.

To cut it simple, we could say that URL routing is the server-side feature that allows a web developer to handle HTTP requests pointing to URIs not mapping to physical files. Such techniques could be used for a number of different reasons, including the following:

- Giving dynamic pages semantic, meaningful, and human-readable names in order to advantage readability and/or search-engine optimization (SEO)
- Renaming or moving one or more physical files within your project's folder tree without being forced to change their URLs
- Setup aliases and redirects

Routing through the ages

In earlier times, when ASP.NET was just Web Forms, URL routing was strictly bound to physical files. To implement viable URL convention patterns the developers were forced to install/configure a dedicated URL rewriting tool using either an external ISAPI filter such as Helicontech's SAPI Rewrite or start with IIS7-the IIS URL Rewrite Module.

When ASP.NET MVC was released, the routing pattern was completely rewritten: the developers could set up their own convention-based routes in a dedicated file (`RouteConfig.cs` and `Global.asax`, depending on the template) using the `Routes.MapRoute` method. If you've played along with MVC 1-5 or WebAPI 1 and/or 2, snippets like this should be quite familiar to you:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
);
```

This method of defining routes, strictly based upon pattern matching techniques used to relate any given URL requests to specific controller actions, went by the name of convention-based routing.

ASP.NET MVC5 brought something new, as it was the first version supporting the so-called attribute-based routing. This approach was designed as an effort to give developers a more versatile approach. If you used it at least once you'll probably

agree that it was a great addition to the framework, as it allowed the developers to define routes within the controller file. Even those who chose to keep the convention-based approach could find it useful for one-time overrides such as the following, without having to sort it out using some regular expressions:

```
[RoutePrefix("v2Products")]
public class ProductsController : Controller
{
    [Route("v2Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

In ASP.NET Core MVC (MVC 6), with the routing pipeline being completely rewritten, attribute-based routing is quickly becoming a de facto standard, replacing the convention, based approach in most boilerplates and code samples. Setting routing conventions using the `Routes.MapRoute()` method is still possible, yet it ceased to be the preferred way of work. You won't be finding anything like that in the new `Startup.cs` file, which contains a very small amount of code and (apparently) nothing about routes.

Handling routes in ASP.NET Core

We could say that the reason behind the `Routes.MapRoute` method disappearance in the application's main configuration file is due to the fact that there's no need to set up default routes anymore. Routing is handled by the two brand new `services.AddMvc()` and `services.UseMvc()` methods called within the `Startup.cs` file, which respectively register MVC using the dependency injection framework built into ASP.NET Core and add a set of default routes to our app.

We can take a look at what happens under the hood looking at the current implementation of the `services.UseMvc()` method in the framework code (relevant lines are highlighted):

```
public static IApplicationBuilder UseMvc(
    [NotNull] this IApplicationBuilder app,
    [NotNull] Action<IRouteBuilder> configureRoutes)
{
    // Verify if AddMvc was done before calling UseMvc
    // We use the MvcMarkerService to make sure if all the services
were added.

MvcServicesHelper.ThrowIfMvcNotRegistered(app.ApplicationServices);
```

```
    var routes = new RouteBuilder
    {
        DefaultHandler = new MvcRouteHandler(),
        ServiceProvider = app.ApplicationServices
    };

    configureRoutes(routes);

    // Adding the attribute route comes after running the user-code
because
    // we want to respect any changes to the DefaultHandler.
    routes.Routes.Insert(0,
    AttributeRouting.CreateAttributeMegaRoute(

        routes.DefaultHandler,

        app.ApplicationServices));

    return app.UseRouter(routes.Build());
}
```

The good thing about this is the fact that the framework now handles all the hard work, iterating through all the controller's actions and setting up their default routes, thereby saving us some work. It is worth noticing that the default ruleset follows the standard RESTful conventions, meaning that it will be restricted to the following action names: `Get`, `Post`, `Put`, and `Delete`. We could say here that ASP.NET Core is enforcing a strict WebAPI-oriented approach, which is to be expected, since it incorporates the whole ASP.NET Core framework.

Following the RESTful convention is generally a great thing to do, especially if we aim to create a set of pragmatic, RESTful-based public APIs to be used by other developers. Conversely, if we're developing our own app and want to keep our API accessible only to our eyes, going for custom routing standards is just as viable. As a matter of fact, it could even be a better choice to shield our controllers against most trivial forms of request flood and/or DDoS-based attacks. Luckily enough, both the convention-based routing and the attribute-based routing are still alive and well, allowing you to set up your own standards.

Convention-based routing

If we feel like using the most classic routing approach, we can easily resurrect our beloved `MapRoute()` method by enhancing the `app.UseMvc()` call within the `Startup.cs` file in the following way:

```
app.UseMvc(routes =>
{
    // Route Sample A
```

```
    routes.MapRoute(
        name: "RouteSampleA",
        template: "MyOwnGet",
        defaults: new {
            controller = "Items",
            action = "Get"
        }
    );
    // Route Sample B
    routes.MapRoute(
        name: "RouteSampleB",
        template: "MyOwnPost",
        defaults: new {
            controller = "Items",
            action = "Post"
        }
    );
});
```

Attribute-based routing

The previously shown `ItemsController.cs` makes good use of the attribute-based routing approach, featuring it either at the controller level:

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

And also featuring it at the action method level:

```
[HttpGet("GetLatest")]
public JsonResult GetLatest()
```

Three choices to route them all

Long story short, ASP.NET Core is giving us three different choices for handling routes: enforcing the standard RESTful conventions, reverting back to the good old convention-based routing, or decorating the controller files with the attribute-based routing.

It's also worth noticing that attribute-based routes, if/when defined, would override any matching convention-based pattern. Both of them if/when defined, would override the default RESTful conventions created by the built-in `UseMvc()` method.

In this chapter, we're going to use all of these approaches in order to learn when, where, and how to properly make use of either of them.

Adding more routes

Let's get back to our `ItemsController`. Now that we're aware of the routing patterns we can use, we can use that knowledge to implement the API calls we're still missing.

Open the `ItemsController.cs` file and add the following code (new lines are highlighted):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using OpenGameListWebApp.ViewModels;
using Newtonsoft.Json;

namespace OpenGameListWebApp.Controllers
{
    [Route("api/[controller]")]
    public class ItemsController : Controller
    {
        #region Attribute-based Routing
        /// <summary>
        /// GET: api/items/GetLatest/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing the last inserted items.</returns>
        [HttpGet("GetLatest/{n}")]
        public IActionResult GetLatest(int n)
        {
            var items = GetSampleItems().OrderByDescending(i =>
            i.CreatedDate).Take(n);

            return new JsonResult(items, DefaultJsonSettings);

        }

        /// <summary>

        /// GET: api/items/GetMostViewed/{n}

        /// ROUTING TYPE: attribute-based

        /// </summary>

        /// <returns>An array of {n} Json-serialized objects
representing
        the items with most user views.</returns>
```

```csharp
        [HttpGet("GetMostViewed/{n}")]

        public IActionResult GetMostViewed(int n)

        {

            var items = GetSampleItems().OrderByDescending(i =>
            i.ViewCount).Take(n);

            return new JsonResult(items, DefaultJsonSettings);

        }

        /// <summary>

        /// GET: api/items/GetRandom/{n}

        /// ROUTING TYPE: attribute-based

        /// </summary>

        /// <returns>An array of {n} Json-serialized objects
representing
        some randomly-picked items.</returns>

        [HttpGet("GetRandom/{n}")]

        public IActionResult GetRandom(int n)

        {

            var items = GetSampleItems().OrderBy(i =>
Guid.NewGuid()).Take(n);

            return new JsonResult(items, DefaultJsonSettings);

        }

        #endregion

        #region Private Members

        /// <summary>

        /// Generate a sample array of source Items to emulate a
database
        (for testing purposes only).

        /// </summary>

        /// <param name="num">The number of items to generate:
```

```
            default is 999</param>

        /// <returns>a defined number of mock items (for testing
purpose only)
        </returns>

        private List<ItemViewModel> GetSampleItems(int num = 999)

        {

            List<ItemViewModel> lst = new List<ItemViewModel>();

            DateTime date = new DateTime(2015, 12, 31).AddDays(-
num);

            for (int id = 1; id <= num; id++)

            {

                lst.Add(new ItemViewModel()

                {

                    Id = id,

                    Title = String.Format("Item {0} Title", id),

                    Description = String.Format("This is a sample
description

                    for item {0}: Lorem ipsum dolor sit amet.",
id),

                    CreatedDate = date.AddDays(id),

                    LastModifiedDate = date.AddDays(id),

                    ViewCount = num - id

                });

            }

            return lst;

        }

        /// <summary>

        /// Returns a suitable JsonSerializerSettings object that
can
        be used to generate the JsonResult return value for this
Controller's
```

```
        methods.

        /// </summary>

        private JsonSerializerSettings DefaultJsonSettings

        {

            get

            {

                return new JsonSerializerSettings()

                {

                    Formatting = Formatting.Indented

                };

            }

        }

        #endregion
    }
}
```

We did a lot of things there, that's for sure. Let's see what's new:

- We added the `GetMostViewed(n)` and `GetRandom(n)` methods, which are built upon the same mocking logic used for `GetLatest(n)`. Either one requires a single parameter of integer type to specify the (maximum) number of items to retrieve.
- We added some new private members:
  - The `GetLatestItems()` method to generate some sample `Item` objects when we need them. This method is an improved version of the dummy item generator loop we had inside the previous `GetLatest()` method implementation, as it acts more like a dummy data provider. We'll discuss this in more detail later on.
  - The `DefaultJsonSettings` property, so we won't have to manually instantiate a `JsonSerializerSetting` object every time.

- We also decorated each class member with a dedicated `<summary>` documentation tag explaining what it does and its return value. These tags will be used by IntelliSense to show real-time information about the type within the Visual Studio GUI. They will also come in handy when we want to generate an autogenerated XML documentation for our project by using industry-standard

documentation tools such as Sandcastle.

- Finally, we added some `#region`/`#endregion` pre-processor directives to separate our code into blocks. We'll do this a lot from now on, as this will greatly increase our source code readability and usability, allowing us to expand or collapse different sections/parts when we don't need them, thus focusing more on what we're working on.

Tip

For more info regarding documentation tags, take a look at the following MSDN official documentation page: https://msdn.microsoft.com/library/2d6dt3kf.aspx.

If you want to know more about C# pre-processor directives, this is the one to check out instead: https://msdn.microsoft.com/library/9a1ybwek.aspx.

The dummy data provider

Our new `GetLatestItems()` method deserves a couple more words. As we can easily see, it emulates the role of a data provider, returning a list of items in a credible fashion. Notice that we built it in a way that it will always return identical items, as long as the `num` parameter value remains the same:

- The generated items `Id` will follow a linear sequence, from `1` to `num`.
- Any generated item will have incremental `CreatedDate` and `LastModifiedDate` values based upon their `Id`: the higher the `Id` is, the most recent the two dates will be, up to 31 December, 2015. This follows the assumption that most recent items will have a higher `Id`, as it normally is for DBMS records featuring numeric, auto-incremental keys.
- Any generated item will have a decreasing `ViewCount` value based upon their `Id`: the higher the `Id` is, the less it will be. This follows the assumption that newer items will generally get fewer views than older ones.

While it obviously lacks any insert/update/delete features, this dummy data provider is viable enough to serve our purposes until we'll replace it with an actual, persistence-based data source.

Note

Technically speaking, we could do something better than we did by using one of the many Mocking Frameworks available through NuGet: Moq, NMock3, NSubstitute, or Rhino, just to name a few.

These frameworks are the ideal choice when using a test-driven development (TDD) approach, which is not the case in this book. In this specific scenario, our dummy

data provider is still a viable way to get what we need while keeping the focus on our main topic: ASP.NET Core and Angular 2 interaction.

Dealing with single items

Our updated `ItemsController` class has everything we need to build our *Welcome View*: as soon as we learn how to call it from our client code, we'll be able to fetch the required data to populate all the item lists we expect to have within our GUI.

However, we don't have anything yet to handle any kind of user interaction. When our users click an item, they will surely expect to read something more about it; ideally, they will want to see something similar to a detail page.

To put it in other words: since our *Welcome View* features a list of clickable items, sooner or later we'll have to give our users the chance to select one of them, ideally with a left mouse click and display the selected item's details: something like a master/detail navigation pattern of any sort.

We're not dealing with the client-side code yet, so we don't know how we'll present such a scenario to the user. However, we already know what we'll eventually need: an API call to retrieve a single `Item` by passing its unique `Id`. To do that we need to provide our controller with the appropriate API method, which could be something like this:

```
/api/items/Get/{id}
```

This means that we need to add something more to our `ItemsController` source code. While we're at it, let's take the chance to make some more improvements:

- A sample method based upon the standard RESTful conventions, just to demonstrate how we can handle that
- A parameterless overload for each method, to make their usage easier
- A couple more private properties to define:
    - The default number of items to retrieve using the parameterless overload
    - The maximum number of items to retrieve within a single API call

The resulting code can be seen as follows (new lines are highlighted):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using OpenGameListWebApp.ViewModels;
using Newtonsoft.Json;

namespace OpenGameListWebApp.Controllers
{
```

```csharp
[Route("api/[controller]")]
public class ItemsController : Controller
{
    #region RESTful Conventions

    /// <summary>

    /// GET: api/items

    /// </summary>

    /// <returns>Nothing: this method will raise a HttpNotFound
HTTP
     exception, since we're not supporting this API call.
</returns>

    [HttpGet()]

    public IActionResult Get()

    {

        return NotFound(new { Error = "not found" });

    }

    /// <summary>

    /// GET: api/items/{id}

    /// ROUTING TYPE: attribute-based

    /// </summary>

    /// <returns>A Json-serialized object representing a single
item.
    </returns>

    [HttpGet("{id}")]

    public IActionResult Get(int id)

    {

        return new JsonResult(GetSampleItems()

            .Where(i => i.Id == id)

            .FirstOrDefault(),

            DefaultJsonSettings);
```

```csharp
        }

        #endregion

        #region Attribute-based Routing

        /// <summary>

        /// GET: api/items/GetLatest

        /// ROUTING TYPE: attribute-based

        /// </summary>

        /// <returns>An array of a default number of Json-
serialized
        objects representing the last inserted items.</returns>

        [HttpGet("GetLatest")]

        public IActionResult GetLatest()

        {

            return GetLatest(DefaultNumberOfItems);

        }

        /// <summary>
        /// GET: api/items/GetLatest/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing the last inserted items.</returns>
        [HttpGet("GetLatest/{n}")]
        public IActionResult GetLatest(int n)
        {
            if (n > MaxNumberOfItems) n = MaxNumberOfItems;
            var items = GetSampleItems().OrderByDescending(i =>
i.CreatedDate).Take(n);
            return new JsonResult(items, DefaultJsonSettings);
        }

        /// <summary>

        /// GET: api/items/GetMostViewed

        /// ROUTING TYPE: attribute-based

        /// </summary>

        /// <returns>An array of a default number of Json-
```

```
serialized
        objects representing the items with most user views.
</returns>

        [HttpGet("GetMostViewed")]

        public IActionResult GetMostViewed()

        {

                return GetMostViewed(DefaultNumberOfItems);

        }

        /// <summary>
        /// GET: api/items/GetMostViewed/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing the items with most user views.</returns>
        [HttpGet("GetMostViewed/{n}")]
        public IActionResult GetMostViewed(int n)
        {
                if (n > MaxNumberOfItems) n = MaxNumberOfItems;
                var items = GetSampleItems().OrderByDescending(i =>
i.ViewCount).Take(n);
                return new JsonResult(items, DefaultJsonSettings);
        }

        /// <summary>

        /// GET: api/items/GetRandom

        /// ROUTING TYPE: attribute-based

        /// </summary>

        /// <returns>An array of a default number of
        Json-serialized objects representing some randomly-picked
items.
         </returns>

        [HttpGet("GetRandom")]

        public IActionResult GetRandom()

        {

                return GetRandom(DefaultNumberOfItems);

        }
```

```csharp
        /// <summary>
        /// GET: api/items/GetRandom/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing some randomly-picked items.</returns>
        [HttpGet("GetRandom/{n}")]
        public IActionResult GetRandom(int n)
        {
            if (n > MaxNumberOfItems) n = MaxNumberOfItems;
            var items = GetSampleItems().OrderBy(i =>
Guid.NewGuid()).Take(n);
            return new JsonResult(items, DefaultJsonSettings);
        }
        #endregion

        #region Private Members
        /// <summary>
        /// Generate a sample array of source Items to emulate a
database (for testing purposes only).
        /// </summary>
        /// <param name="num">The number of items to generate:
default is 999</param>
        /// <returns>a defined number of mock items (for testing
purpose only)</returns>
        private List<ItemViewModel> GetSampleItems(int num = 999)
        {
            List<ItemViewModel> lst = new List<ItemViewModel>();
            DateTime date = new DateTime(2015, 12, 31).AddDays(-
num);
            for (int id = 1; id <= num; id++)
            {
                date = date.AddDays(1);
                lst.Add(new ItemViewModel()
                {
                    Id = id,
                    Title = String.Format("Item {0} Title", id),
                    Description = String.Format("This is a sample
description for item {0}: Lorem ipsum dolor sit amet.", id),
                    CreatedDate = date,
                    LastModifiedDate = date,
                    ViewCount = num - id
                });
            }
            return lst;
        }

        /// <summary>
        /// Returns a suitable JsonSerializerSettings object that
can be used to generate the JsonResult return value for this
Controller's methods.
        /// </summary>
```

```csharp
        private JsonSerializerSettings DefaultJsonSettings
        {
            get
            {
                return new JsonSerializerSettings()
                {
                    Formatting = Formatting.Indented
                };
            }
        }

        /// <summary>
        /// Returns the default number of items to retrieve when using the
        ///  parameterless overloads of the API methods retrieving item lists.
        /// </summary>
        private int DefaultNumberOfItems
        {
            get
            {
                return 5;
            }
        }

        /// <summary>
        /// Returns the maximum number of items to retrieve when using the
        /// API methods retrieving item lists.
        /// </summary>
        private int MaxNumberOfItems
        {
            get
            {
                return 100;
```

```
            }

        }
        #endregion
    }
}
```

We've already explained the new goodies out there, yet it can be useful to focus on a couple of them:
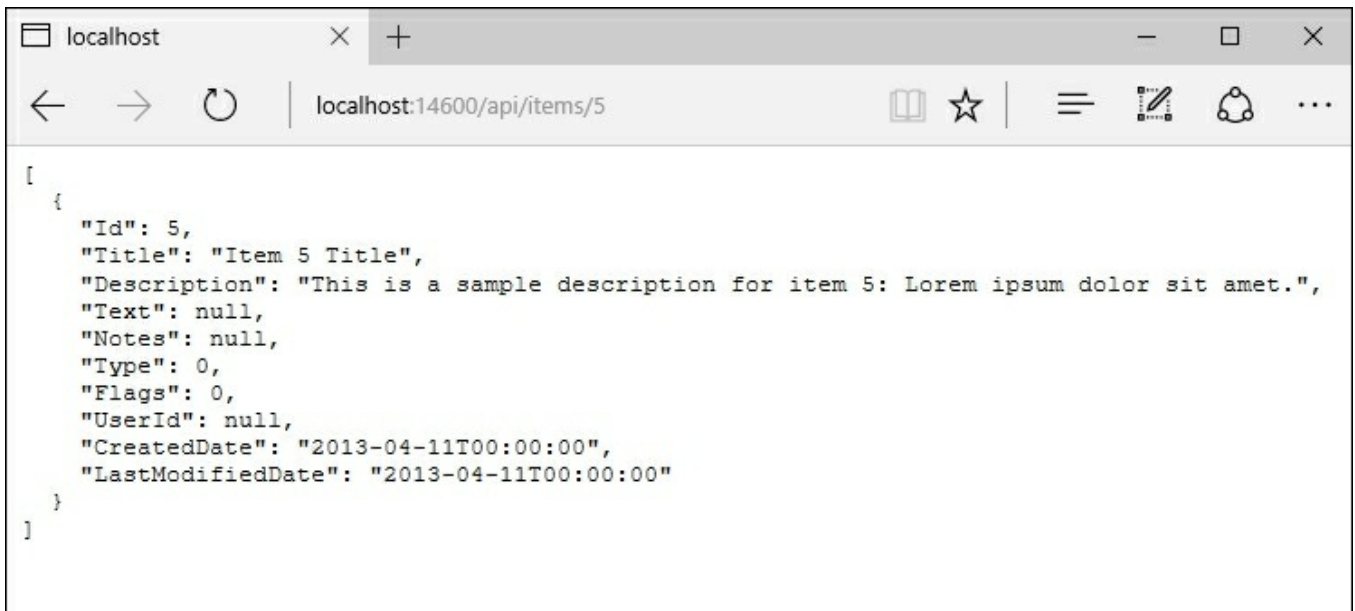
- Our brand new `Get()` method won't do anything more than return a `404 HTTP Error` with a custom error string. We added it to grant an explicit response to that RESTful default call and also to demonstrate how we can choose between handling them or not.
- The `DefaultNumberOfItems` and `MaxNumberOfItems` have been added to store their values in a centralized way, so we can avoid repeating them multiple times. We made them as properties, but they could also be defined as constants or private variables as well since they are meant to be for internal use only; we won't reference them outside this class or anywhere else.

Let's test our new methods: select **Debug** | **Start Debugging** from the main menu (or hit **F5**) and type the following URLs in the browser's address bar:

```
/api/items
```
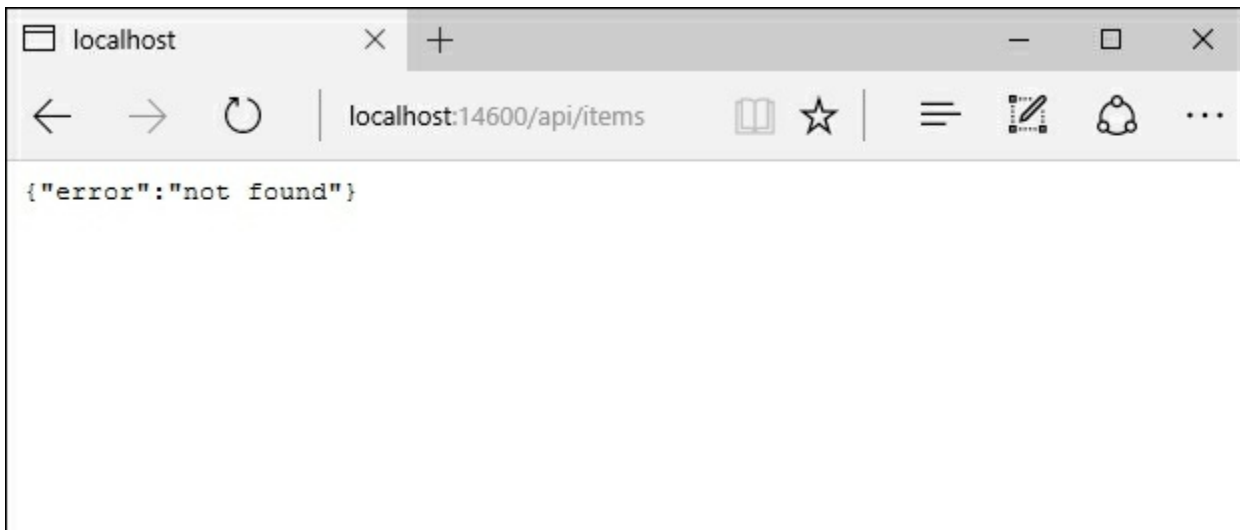


```
/api/items/5
```

As we've already said, the **404 Page Not Found** error in response to the first HTTP request is perfectly fine; we did it on purpose to demonstrate how we can handle these kinds of errors. This is also good practice when dealing with ASP.NET Core API interfaces; since we chose to not accept any `/api/items` get call without parameters, we want the client to be aware of that.

So far, so good; we've got ourselves a number of server-side APIs to retrieve JSON arrays filled by a client-defined (or default) number of latest items, and an additional one to retrieve a single item from its unique ID. All of these calls will be very handy in the following chapter, where we'll start developing client-side components using Angular 2.

Suggested topics

HTTP request, HTTP response, convention-based routing, attribute-based routing, RESTful conventions, mock objects, test-driven development, XML documentation tags, and C# preprocessor directives.

Summary

We spent some time putting the standard application data flow under our lens: a two-way communication pattern between the server and their clients, built upon the HTTP protocol. We acknowledged the fact that we'll be mostly dealing with JSON-serializable object such as **Items**, so we chose to equip ourselves with an `ItemViewModel` server-side class, together with an `ItemsController` that will actively use it to expose the data to the client.

We started building our MVC6-based Web API interface by implementing a number of methods required to create the client-side UI we chose for our *Welcome View*, consisting of three item listings to show to our users: last inserted ones, most viewed ones, and some random picks. We routed the requests to them by using a custom set of attribute-based routing rules, which seemed to be the best choice for our specific scenario.

While we were there, we also took the chance to add a dedicated method to retrieve a single `Item` from its unique `Id`, assuming that we were going to need it for sure.

In the next chapter, we will see how we can consume the ASP.NET Core Web API using Angular 2 in order to build an interactive user interface for our application.

Chapter 3. Angular 2 Components and Client-Side Routing

Our server-side Web API is working fine; however, the `ItemsController` is still missing parts that we're going to need eventually, such as the POST-based methods to insert, update, and delete items. There's no sense in adding them now since we're still mocking up our objects. We'll implement these soon enough when we switch to a real data model.

For the time being, let's see how we can make good use of what we just did by implementing a basic master-detail navigation pattern using Angular 2, which is our application client framework of choice.

Let's summarize what we're going to do in this chapter:

- Greet our users with a **Welcome View** with three different lists of items: latest, most viewed, and random. We'll populate these by fetching the relevant data using the `ItemsController` API methods built in Chapter 2, *ASP.NET Controllers and Server-Side Routes*.
- Let our users navigate to an **Item Detail View** upon clicking on any item.
- Allow our users to go back from the **Item Detail View** to the **Welcome View** upon clicking on a back link.

This isn't anything different from a standard, straightforward master-detail pattern. When we're done, users will be able to perform a basic navigation loop like the following:
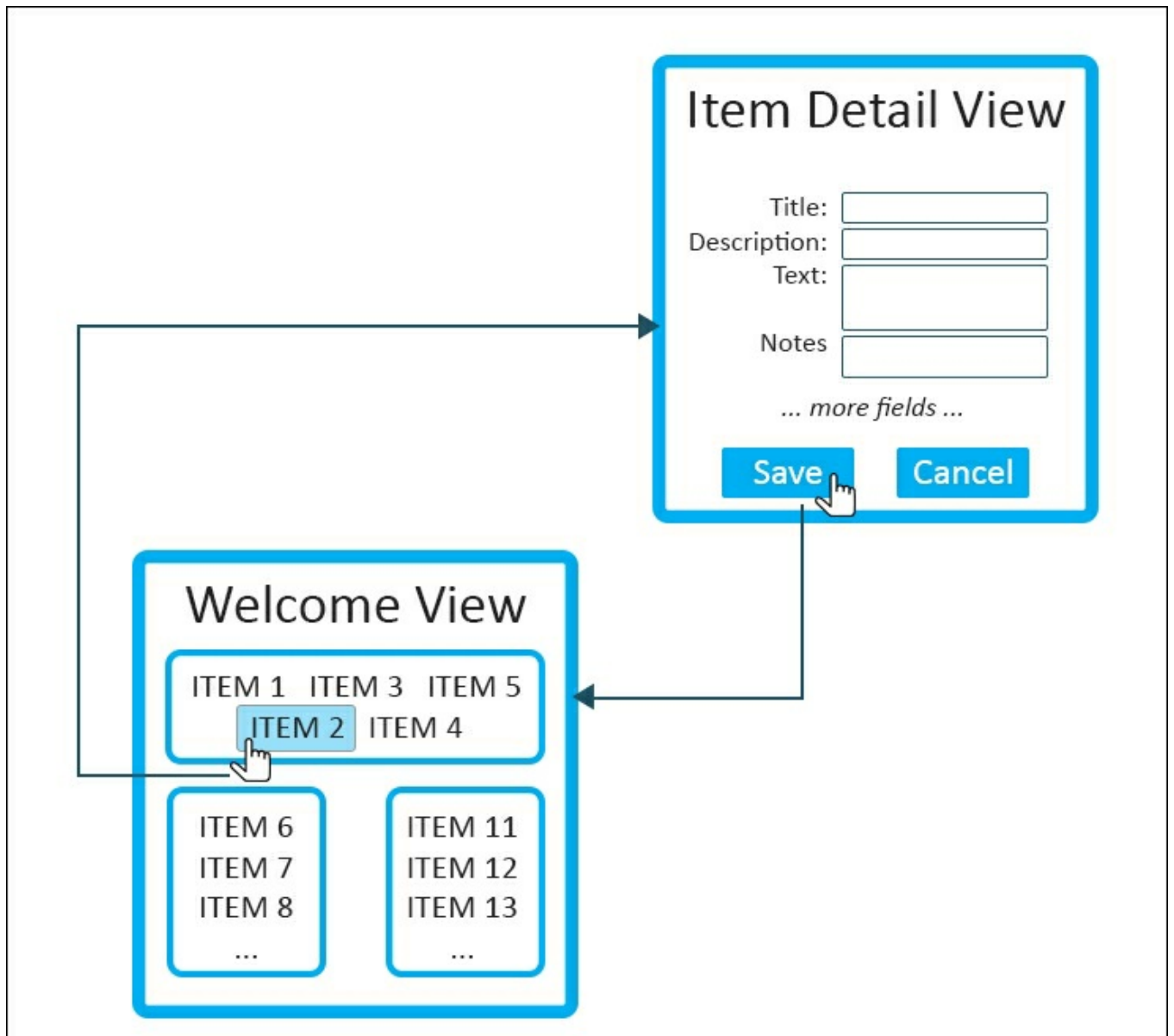
Notice that we will also give the user the chance to modify some item fields, reflecting these changes in the underlying client-side data model. Doing this in Angular 2 is a rather seamless task due to the framework's built-in two-way data binding features.

Persisting these changes to the application's data source is a whole different story though, as it requires us to implement a couple more features:

1. A dedicated server-side API that will receive the updated data from the client model and sync it to the server model.
2. A client-side update command pattern, such as a button, a focus event, or something like that, that will trigger the aforementioned update API call.

This is something we'll do in the next chapter by implementing a persistent data source and replacing our dummy data provider sample with a working one. We won't be able to properly handle any persisting update command until then.

Master-detail binding

Our main focus now is to implement a master-detail navigation pattern. We'll do that in two steps:

1. Putting together a temporary, component-based master-detail relationship based upon the Angular 2 data-binding feature.
2. Replacing the preceding with a view-based navigation pattern with full client-side routing support.

Truth be told, step *1* isn't mandatory. We could just implement step *2* and get the task over with. However, by choosing to do that, we would skip some really important concepts regarding Angular 2 and its interactions with the Web API structure we just built. This stuff will come in very handy later on; therefore, it's highly recommended that we restrain ourselves from rushing things.

The Item class

The first thing we need to do is to add the `Item` class to our Angular-enabled client. Wait a minute... should we really do that? Can't we just use the raw JSON data sent by the controller's `Get` and `GetLatest` methods defined earlier, consuming them as anonymous JavaScript objects?

Theoretically speaking, we could try as much as we could output raw JSON from the controller instead of creating our `ItemViewModel` class. In a well-written app, though, we should always restrain ourselves from the temptation to handle raw JSON data and/or to use anonymous objects for a number of good reasons:

- We're using TypeScript because we want to work with type definitions: anonymous objects and undefined properties are the exact opposite and lead into the JavaScript way of doing things, which is something we wanted to avoid in the first place.
- Anonymous objects are not easy to validate. We don't want our data items to be error prone, being forced to deal with missing properties or anything like that.
- Anonymous objects are hardly reusable and won't benefit from many Angular 2 handy features, such as the object mapping, which will require our objects to be actual instances of an interface and/or a type.

The first two points are very important, especially if we're aiming for a production-ready application. However, point 3 is also crucial as long as we want to use Angular 2 to its full extent. If that's still the case, using an undefined array of properties like raw JSON data basically would be out of the question. We're going to use objects, i.e., actual instances of classes.

That's why we need a client-side, TypeScript `Item` class to properly map our JSON-serialized `ItemViewModel` class.

From the **Solution Explorer**, add a new **TypeScript File** in the `/Scripts/app/` folder, call it `item.ts`, and fill it with the following code:

```
export class Item {
    constructor(
        public Id: number,
        public Title: string,
        public Description: string
    ) { }
}
```

Notice that we're not adding all the properties present in the `ItemViewModel` class. As a general rule of thumb, we'll be keeping these classes as lightweight as possible,

defining only what we know we're going to use for sure. We can always add more properties later, as soon as we need them.

The ItemService class

Now that we can properly handle JSON-serialized `Item` objects, we need to set up a service to fetch the required data from the Web API. We'll do that by issuing a request to the `ItemsController` we built in the first part of this chapter.

We will do that using HTTP, which is the primary protocol for browser/server communication. Notice that thanks to the features introduced with HTML5, this is not our only option anymore. We could use the WebSocket protocol as well. We won't cover it in this chapter, though, as the HTTP request pattern is still viable enough to do what we need to do: fetch a JSON array of `Item` objects.

To be more specific, we're going to use the Angular HTTP client to communicate via `XMLHttpRequest` (**XHR**), which is a rather complex HTTP-based API that provides client functionality for transferring data between a client and a server. We won't have to dig too much into it, though, as Angular greatly simplifies the application programming with an easy-to-use interface featuring a small number of high-level methods.

From the **Solution Explorer**, add a new **TypeScript File** in the `/Scripts/app/` folder, call it `item.service.ts`, and fill it with the following code:

```typescript
import {Injectable} from "@angular/core";
import {Http, Response} from "@angular/http";
import {Observable} from "rxjs/Observable";
import {Item} from "./item";

@Injectable()
export class ItemService {
    constructor(private http: Http) { }

    private baseUrl = "api/items/";  // web api URL

    // calls the [GET] /api/items/GetLatest/{n} Web API method to
retrieve the latest items.
    getLatest(num?: number) {
        var url = this.baseUrl + "GetLatest/";
        if (num != null) { url += num; }
        return this.http.get(url)
            .map(response => response.json())
            .catch(this.handleError);
    }

    // calls the [GET] /api/items/GetMostViewed/{n} Web API method
to retrieve the most viewed items.
    getMostViewed(num?: number) {
        var url = this.baseUrl + "GetMostViewed/";
```

```
            if (num != null) { url += num; }
            return this.http.get(url)
                .map(response => response.json())
                .catch(this.handleError);
    }

    // calls the [GET] /api/items/GetRandom/{n} Web API method to
retrieve n random items.
    getRandom(num?: number) {
        var url = this.baseUrl + "GetRandom/";
        if (num != null) { url += num; }
        return this.http.get(url)
            .map(response => response.json())
            .catch(this.handleError);
    }

    // calls the [GET] /api/items/{id} Web API method to retrieve
the item with the given id.
    get(id: number) {
        if (id == null) { throw new Error("id is required."); }
        var url = this.baseUrl + id;
        return this.http.get(url)
            .map(res => <Item>res.json())
            .catch(this.handleError);
    }

    private handleError(error: Response) {
        // output errors to the console.
        console.error(error);
        return Observable.throw(error.json().error || "Server
error");
    }
}
```

You will notice some major similarities between the preceding code and the
`ItemsController.cs` source code. It's rather obvious since this is the class that our
client will use to fetch the data from that Web API controller itself.

Let's see what these lines of code will actually do:

- In lines 1-4, we're importing the modules we need to perform our tasks. Notice
  that, in line 4, we're importing the `Item` class we created in the previous
  chapter, as we're going to use it here.
- In line 6, we make use of the `Injectable` decorator, declaring that ours is an
  `Injectable` class. Doing this will attach a set of metadata to our class that will
  be consumed by the DI system upon instantiation. Basically, what we're doing
  here is telling the DI injector that the constructor parameter(s) should be
  instantiated using their declared type(s). The TypeScript code allows a very
  fluent syntax to achieve this result at constructor level, as can be seen in line 8.

- In line 10, we define a variable containing our Web API-based URL, so we won't have to write it multiple times. Avoiding repetition is always a good thing and is one of the main rules to follow in order to write decent code.
- In the subsequent lines, we create our `getLatest()`, `getMostViewed()`, `getRandom()`, and `Get()` client-side methods that will call their Web API counterpart. Notice that the method names differ in the naming conventions, as we're respecting Javascript's `camelCase` here, while the `ItemsController` follows the `PascalCase` (also known as `UpperCamelCase`) .NET standard. We should also note how the retrieved `ItemViewModel` array of data is mapped to an equal number of `Item` objects by making use of the Angular native mapping module.
- Finally, there's a rudimental implementation of an exception handling method that will be triggered by the preceding two data-fetching methods, in the case of an error. For now, we'll just output the errors to the browser console.

Now that we've got an `Item` and an `ItemService`, we should have everything we need to display our listing of items except one thing: the Angular-based component that will dynamically build the list in plain HTML code. Let's close this gap.

Tip

If you get the odd TypeScript compilation error after implementing this class, you might need to replace the `typescriptServices.js` file, as mentioned in Chapter 1, *Getting Ready*.

The ItemListComponent class

What we're going to do here is to create a dedicated component to handle the "display a list of items task." We could also do that in our already-present `AppComponent` class, but our app is growing fast. Sooner or later we will have to split it into multiple, reusable assets. That being the case, there's no reason to do it right from the start. To be more specific, we're aiming to have the following file structure:

- `app.component.ts`: Our **Welcome View**, with a number of references to sub-components enclosed by a lightweight HTML skeleton
- `item-list.component.ts`: A flexible, reusable item listing component that can be easily included by any view, one or multiple times

We'll stick to this pattern from now on, create a separate `xyz.component.ts` file for any component we'll require. Let's start with building the first one.

Again, from the **Solution Explorer**, add a new **TypeScript File** in the `/Scripts/app/` folder, call it `item-list.component.ts`, and fill it with the following code:

```
import {Component, OnInit} from "@angular/core";
import {Item} from "./item";
import {ItemService} from "./item.service";

@Component({
    selector: "item-list",
    template: `
        <h2>Latest Items:</h2>
            <ul class="items">
            <li *ngFor="let item of items"
                [class.selected]="item === selectedItem"
                (click)="onSelect(item)">
                <span>{{item.Title}}</span>
            </li>
        </ul>
    `,
    styles: [`
        ul.items li {
            cursor: pointer;
        }
        ul.items li.selected {
            background-color: #cccccc;
        }
    `]
})
```

```
export class ItemListComponent implements OnInit {
    selectedItem: Item;
    items: Item[];
    errorMessage: string;

    constructor(private itemService: ItemService) { }

    ngOnInit() {
        this.getLatest();
    }

    getLatest() {
        this.itemService.getLatest()
            .subscribe(
            latestItems => this.items = latestItems,
            error => this.errorMessage = <any>error
            );
    }

    onSelect(item: Item) {
        this.selectedItem = item;
        console.log("item with Id " + this.selectedItem.Id + " has
been selected.");
    }
}
```

That's quite an amount of non-trivial source code. Let's see what we just did in detail:

- In line 1, we imported the Angular classes that we require. Since we're creating a component, we require the `Component` base class and need to implement the `OnInit` interface because our component will need to execute something upon its initialization.
- In lines 2-3, we import our previously created `Item` and `ItemService` classes. No surprises here, since we're obviously using them here.
- In lines 5-25, we set up the component UI aspect and settings. In particular, we'll perform the following tasks:
  - **Line 6**: We're defining the selector, which is the custom HTML element that will be replaced by the component itself.
  - **Lines 7-16**: We're creating the HTML structure of what will be rendered on screen. Note that we used a bit of Angular 2 Template Syntax there in order to get the job done. Specifically, we used a master template, a `ngFor` directive, a property binding, and an event binding. Notice how each command relies on some local properties and/or methods: `item`, `items`, `selectedItem`, and `onSelect()`. We will explain them soon enough, as they are declared further down.
  - **Lines 17-24**: We're adding some minimalistic CSS styles in order to have

some visual feedback about what we're going to do, which is to change the background color of the currently selected item. We'll do something way better later on.

- In lines 27-51, we can find the core `ItemListComponent` class declaration, together with all its properties, constructor and methods:
    - **Lines 28-30**: Here, we define our class `Properties`, the same that we used in our template (see lines 7-16).
    - **Line 32**: The class constructor. Notice how we make use of the Angular 2 native dependency injection features. We're not instantiating the `ItemService` object, we let the framework do that instead. This means that our `ItemsComponent` class won't be creating an `ItemService`, it will just *consume* it right on the spot. While we're there we also made good use of the flexible constructor syntax provided by TypeScript, allowing us to declare the parameter and the property in a single shot.
    - **Lines 34-36**: The `ngOnInit` implementation, which is pretty much straightforward. The `getLatest()` internal method will be executed upon initialization and that's it.
    - **Lines 38-50**: these are the two methods that will be used, respectively, to retrieve the items (`getLatest`), and to handle the only user interaction we need our class to be aware of - the left-button mouse click (`onSelect`).

Note

If you feel curious about the aforementioned Angular 2 Template Syntax, don't worry! We're going to talk way more about that in a while. Nonetheless, if you just can't wait to get a hold on that, you can take a look at the official documentation available on the Angular 2 website:

https://angular.io/docs/ts/latest/guide/template-syntax.html.

A quick implementation test

We just added three major things to our client-side app: a class (`Item`), a JSON-consuming service (`ItemService`), and a UI component specifically designed to use them both (`ItemListComponent`). Before going further, it may be a good call to see the results of our work up to this point to ensure that everything is working as expected.

The first thing we need to do is to add these new features to our application's root module. Open the `/Scripts/app/app.module.ts` file and add the following (new lines are highlighted):

```
///<reference path="../../typings/index.d.ts"/>
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {HttpModule} from "@angular/http";
import "rxjs/Rx";

import {AppComponent} from "./app.component";
import {ItemListComponent} from "./item-list.component";

import {ItemService} from "./item.service";

@NgModule({
    // directives, components, and pipes
    declarations: [
        AppComponent,
        ItemListComponent
    ],
    // modules
    imports: [
        BrowserModule,
        HttpModule
    ],
    // providers
    providers: [
        ItemService
    ],
    bootstrap: [
        AppComponent
    ]
})
export class AppModule { }
```

Let's now get back to our `/Scripts/app/app.component.ts` file, which is supposed to draw our **Welcome View**, and thus is the starting point of our whole app. We left it out near the end of [Chapter 1](#), *Getting Ready*, when we used a minimalistic template, featuring a "... the best is yet to come!" catchphrase as its main content. Do you remember?

It's time to deliver something better. Replace it with the following code:

```
import {Component} from "@angular/core";

@Component({
    selector: "opengamelist",
    template: `
        <h1>{{title}}</h1>

        <item-list></item-list>
```

```
    `
})

export class AppComponent {
    title = "OpenGameList";
}
```

Let's say a few words about what we're doing here:

- In the template, we replaced our previous `hello-worldish` catchphrase with the `<item-list>` custom element, which is the defined selector of the `ItemListComponent` template. This basically means that an `ItemListComponent` will be instantiated there.
- While we were there, we added a title variable within the `AppComponent` class and used it in the template as well.

As soon as we perform these changes, we can launch our application in the debug mode by selecting **Debug** | **Start Debugging** from the main menu or by hitting *F5* and see if everything is working correctly. If that's the case, we should be able to see something like this:
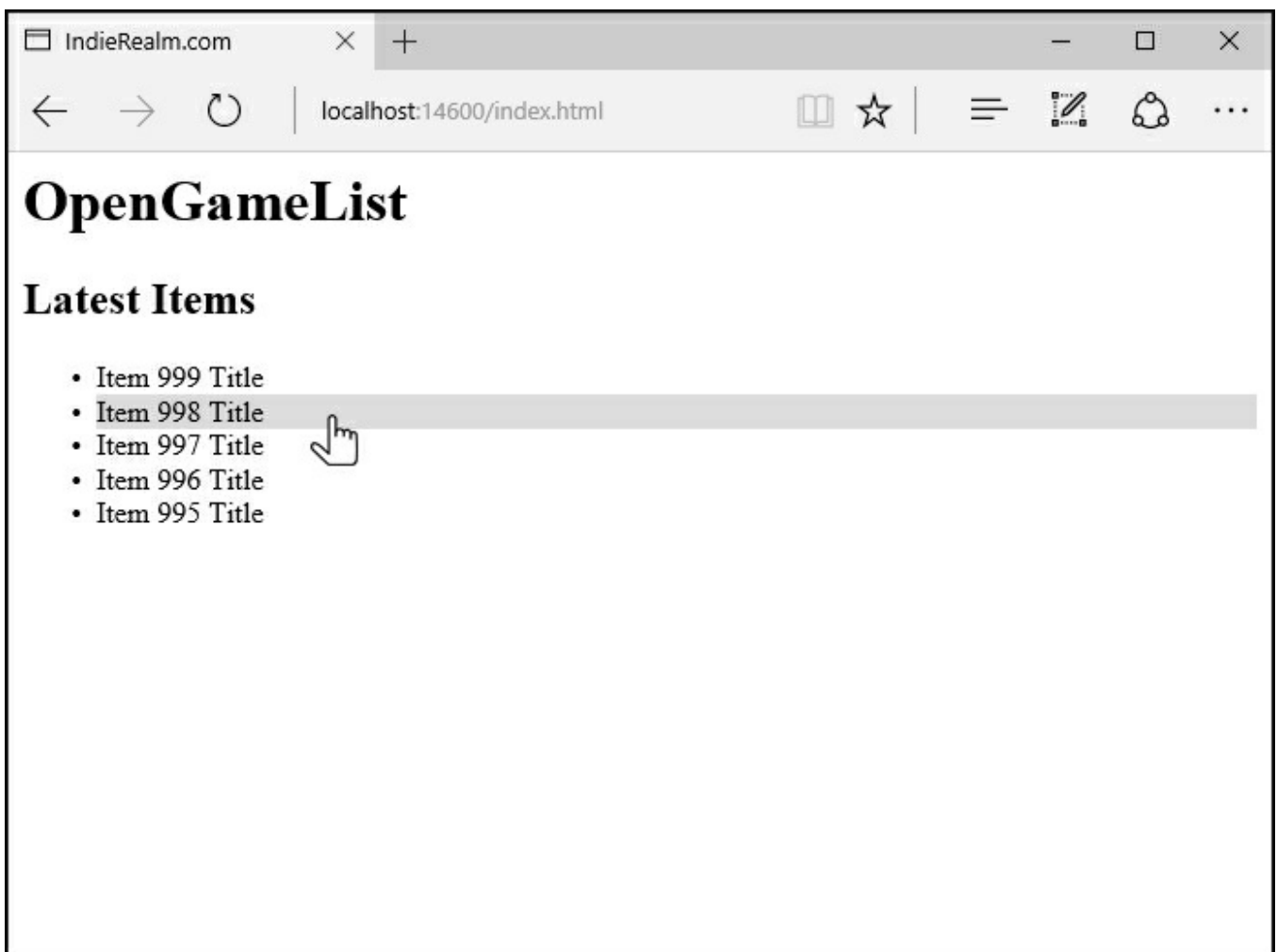
You should also be able to select any item with a left mouse click. The currently selected item will change its background color accordingly.

Tip

Remember to activate the **Gulp** (or **Grunt**) default task, otherwise, your code changes won't be copied to the `/wwwroot/` folder. If you'd forgotten to do that, you can do it now. Right-click on the `gulpfile.js` (or `gruntfile.js`) file and select the **Task Runner Explorer**. Once there, right-click on the **default** task, and activate it by clicking on **Run**.

This is more than enough to tell us that everything is going well. Now that we're sure about it, we can go back to work and add the component that will handle the item's detail view.

The ItemDetailComponent class

Once again, from the **Solution Explorer**, add a new **TypeScript File** in the
`/Scripts/app/` folder, call it `item-detail.component.ts`, and fill it with the
following code:

```typescript
import {Component, Input} from "@angular/core";
import {Item} from "./item";

@Component({
    selector: "item-detail",
    template: `
        <div *ngIf="item" class="item-details">
           <h2>{{item.Title}} - Detail View</h2>
           <ul>
               <li>
                   <label>Title:</label>
                   <input [(ngModel)]="item.Title"
placeholder="Insert the title..."/>
               </li>
               <li>
                   <label>Description:</label>
                   <textarea [(ngModel)]="item.Description"
placeholder="Insert a suitable description..."></textarea>
               </li>
           </ul>
        </div>
     `,
    styles: [`
        .item-details {
            margin: 5px;
            padding: 5px 10px;
            border: 1px solid black;
            background-color: #dddddd;
            width: 300px;
        }
        .item-details * {
            vertical-align: middle;
        }
        .item-details ul li {
            padding: 5px 0;
        }
     `]
})

export class ItemDetailComponent {
    @Input("item") item: Item;
}
```

Here we go. Let's take a quick look at what we're doing here:

- In lines 1-2, we're importing what we need. Notice that we're importing the `Input` module from Angular 2 Core, which we need to create a data-bound input property (see *line 39*). Needless to say, we also need to reference our `Item` class, as this component will display our item's details.
- In line 5, we define the `selector`, the custom HTML element that will be replaced by the component itself.
- In lines 6-22, we're creating the template UI, a simple HTML snippet to present our item details on screen. It's worth noting that the input elements of our simple form feature a strange `ngModel` attribute between parentheses and brackets. We'll dig into that in a while, but for now, let's just say that it's the directive that will enable the two-way data binding feature of Angular 2.
- In lines 23-37, we're adding another minimalistic set of CSS rules. This will still lead to a rather ugly outcome, but there's nothing to worry about. We're not in the styling phase yet, that will come later on. For now, let's just put something up in order to distinguish this component's HTML from the rest.
- Finally, in lines 40-42, we're declaring our `ItemDetailComponent` class. Note that it only has a single `item` property. We have also defined it as using a `@Input()` decorator function (see line 1), that will add the required metadata to make this property available for property binding. We need to do this because we expect this property to be populated from a binding expression within a parent component. We're going to do that in the next paragraph.

Note

The `@Input()` decorator function deserves a couple more words. Note that it accepts an optional alias parameter, which we explicitly added (`"item"`) to demonstrate how it works under the hood. When we add our `ItemDetailComponent` in a parent component, we will also need to pass the item object to show. We'll do that using a custom attribute in the following way:

```
<item-detail *ngIf="selectedItem" [item]="selectedItem"></item-
detail>
```

The alias parameter will be used to determine the attribute hosting the property binding. Its value will be dynamically bound to our target input property.

That being the case, you might be wondering why we called it optional. That's because it can be skipped as long as the attribute name matches the target input property name. In our case, they share the same name, so we could safely remove it.

More information regarding the input property is available through the following Angular 2 official documentation page:
`https://angular.io/docs/ts/latest/guide/attribute-`

```
directives.html#!#why-input.
```

Adding the @angular/forms library

Although the `ItemDetailComponent` component doesn't yet contain a proper `<form>` element, it has some input elements that can greatly benefit from the Angular 2 two-way data binding feature. To implement it, though, we need to add the `@angular/forms` library to our NPM package list.

To do that, open the `package.json` file and add the following highlighted line to the `dependencies` section, right below the other `@angular` packages:

```
"@angular/common": "2.0.0-rc.5",
"@angular/compiler": "2.0.0-rc.5",
"@angular/core": "2.0.0-rc.5",
"@angular/http": "2.0.0-rc.5",
"@angular/platform-browser": "2.0.0-rc.5",
"@angular/platform-browser-dynamic": "2.0.0-rc.5",
"@angular/upgrade": "2.0.0-rc.5",
"@angular/forms": "0.3.0",
```

Subsequently, open the `/wwwroot/systemjs.config.js` file and append the following highlighted line to the `ngPackageNames` array:

```
// configure @angular packages
var ngPackageNames = [
  'common',
  'compiler',
  'core',
  'http',
  'platform-browser',
  'platform-browser-dynamic',
  'upgrade',
  'forms'
];
```

This will allow us to import the `FormsModule` within our application's root module in the following paragraph, thereby enabling the two-way data binding feature.

Updating the root module

Every time we add an Angular 2 component we also need to import and declare it to our root module (or a nested module, which we don't have). The `ItemDetailComponent` won't be an exception, so we have to add it just like we did with the `ItemListComponent` a short while ago. On top of that, since the same rule also applies to modules and providers, we need to do that for the `FormsModule` as well.

Open the `/Scripts/app/app.module.ts` file and update its contents accordingly (new lines are highlighted):

```
///<reference path="../../typings/index.d.ts"/>
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {HttpModule} from "@angular/http";
import {FormsModule} from "@angular/forms";
import "rxjs/Rx";

import {AppComponent} from "./app.component";
import {ItemListComponent} from "./item-list.component";
import {ItemDetailComponent} from "./item-detail.component";
import {ItemService} from "./item.service";

@NgModule({
    // directives, components, and pipes
    declarations: [
        AppComponent,
        ItemListComponent,
        ItemDetailComponent
    ],
    // modules
    imports: [
        BrowserModule,
        HttpModule,
        FormsModule
    ],
    // providers
    providers: [
        ItemService
    ],
    bootstrap: [
        AppComponent
    ]

})
export class AppModule { }
```

Connecting the dots

Now that we have a component to display our item's details, we need to connect it to the item listing one we created previously. Doing that is as easy as adding a couple of lines to our `ItemListComponent` class.

From the **Solution Explorer**, open the `/Scripts/app/item-list.component.ts` file and change it as we have in the following code (added lines are highlighted):

```
import {Component, OnInit} from '@angular/core';
import {Item} from "./item";
import {ItemService} from "./item.service";
```

```
@Component({
    selector: "item-list",
    template: `
        <h2>Latest Items:</h2>
        <ul class="items">
            <li *ngFor="let item of items"
                [class.selected]="item === selectedItem"
                (click)="onSelect(item)">
                <span>{{item.Title}}</span>
            </li>
        </ul>
        <item-detail *ngIf="selectedItem" [item]="selectedItem">
</item-detail>
    `,
    styles: [`
        ul.items li {
            cursor: pointer;
        }
        ul.items li.selected {
            background-color: #dddddd;
        }
    `]
})

export class ItemListComponent implements OnInit {
    selectedItem: Item;
    items: Item[];
    errorMessage: string;

    constructor(private itemService: ItemService) { }

    ngOnInit() {
        this.getLatest();
    }

    getLatest() {
        this.itemService.getLatest()
            .subscribe(
            latestItems => this.items = latestItems,
            error => this.errorMessage = <any>error
            );
    }

    onSelect(item: Item) {
        this.selectedItem = item;
        console.log("item with Id " + this.selectedItem.Id + " has
been selected.");
    }
}
```

All we did here was to add an `<item-detail>` element with an `item` property,

which is the target of a property binding. This is the reason why it is in square brackets, to the left of the equal sign. We already knew about it from the previous paragraph, when we were referring to the `@Input()` decorator attribute and its optional alias parameter.

Note

It's worth noting that we don't need to import the `ItemDetailComponent` to use the `<item-detail>` element, as the compiler will automatically fetch that from the `AppModule`. That's precisely what our root module is for. Before Angular 2 RC5, we had to manually handle the required import throughout all the component files, resulting in major (and also error-prone) code repetition.

Testing it up

It's time to perform a full test of what we've built up to this point.

Click on **Debug** | **Start Debugging** (or hit **F5**) to display the same view from `Image03.02` again, then browse through the items by clicking the left mouse button. You should be presented with something like the following:
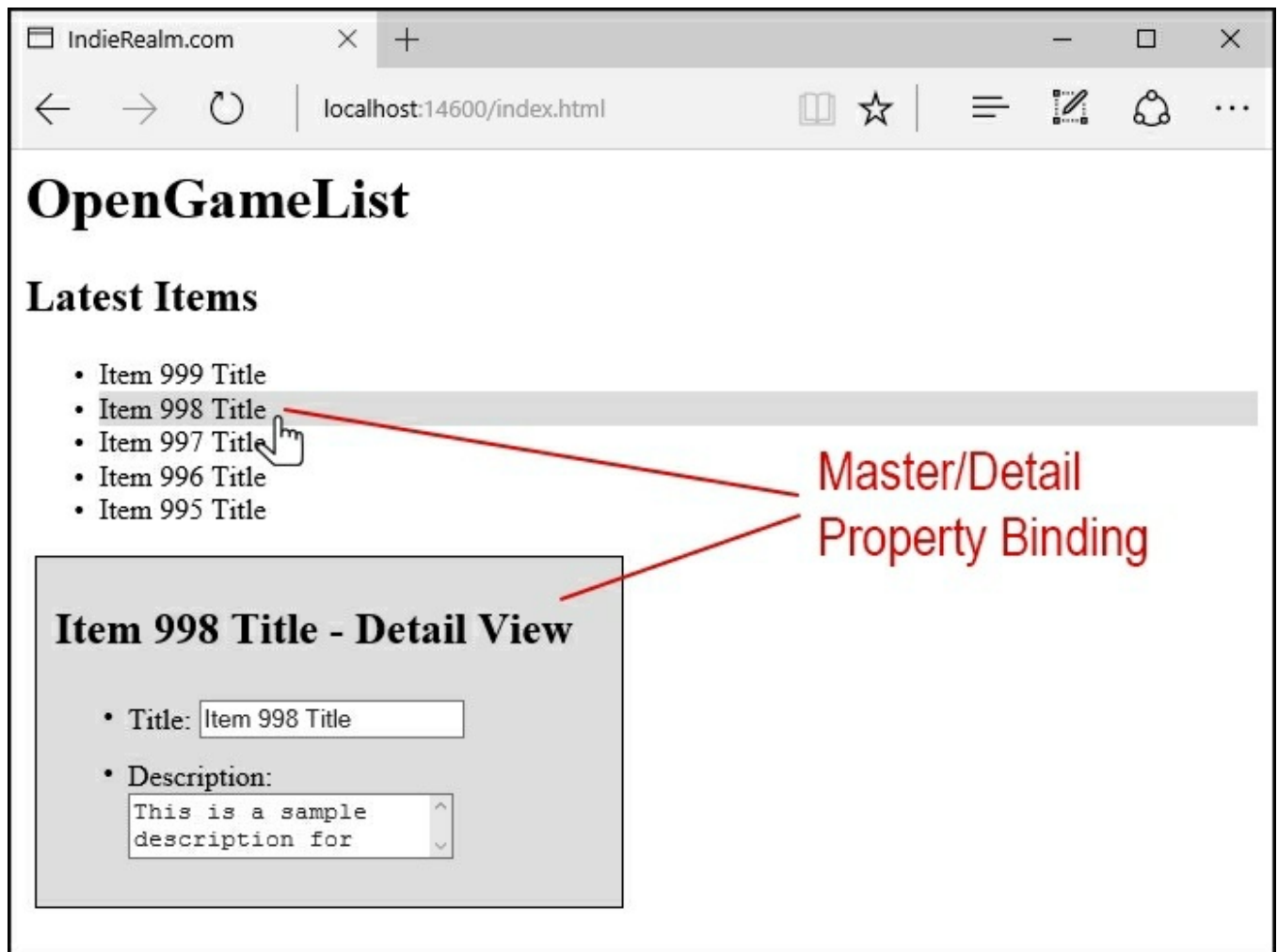
If everything is working as it should, as we change the `selectedItem` the **Detail View** panel should be updated as well, showing that item's details.

Adding additional lists

Our welcome page isn't done yet. Two out of three item lists are yet to be done, at least on the client side. We're talking about the most viewed items and the randomly picked ones, so let's add them to the loop. We basically have two ways to do this:

1. Adding two more Angular components that are very similar to the `ItemListComponent` one.
2. Extending our `ItemListComponent` and making it configurable, thereby making it able to handle all the three item listings.

Adding two more components would be rather easy. We could clone the `item-list.component.ts` file a couple of times, change the inner method of the two new files to make it fetch the relevant data from the other Web API method (respectively, `GetMostViewed()` and `GetRandom()`), define a different selector, add it to the `AppComponent` template code and we would be done.

Yet, it would also be a horrible approach. Let's restrain us from cloning any part of our code, unless there's really no other way to get the things done. We're using Angular 2 because we want to build versatile and reusable components, and we're going to stick to this path as much as we can. We also don't want to spawn unnecessary components, as it would be a pain to keep them in sync each and every time we have to apply a definition update, a member rename, an interface change, or any other source code modification that will affect either of them.

For the previous reasons, we'll definitely choose the second option. Not to mention the fact that it will be just as easy, once we know how to do it properly.

Tip

Regarding that choice, it's easy to see that we've already started off on the right foot. We called it `ItemListComponent`, instead of `LatestItemsComponent` because we never really wanted to have it showing the last inserted items only. We could say that we already knew that we would be choosing the second option right from the start. It's hardly a surprise, though, since we're fully committed to building reusable components.

Multiple component instances

The first thing we need to do is to configure the `AppComponent` HTML template to make it render a couple more `<item-list>` component tags. While doing that, we also need to find a way to uniquely identify them in order to issue a different behavior for each instance: latest items, most viewed items, and random items.

Open the `app.component.ts` file and update our previous code in the following way (added/modified lines are highlighted):

```
import {Component} from '@angular/core';

@Component({
    selector: "opengamelist",
    template: `
        <h1>{{title}}</h1>

        <item-list class="latest"></item-list>

        <item-list class="most-viewed"></item-list>

        <item-list class="random"></item-list>

    `,

    styles: [`

        item-list {

            min-width: 332px;

            border: 1px solid #aaaaaa;

            display: inline-block;

            margin: 0 10px;

            padding: 10px;

        }

        item-list.latest {

            background-color: #f9f9f9;

        }
```

```
        item-list.most-viewed {

            background-color: #f0f0f0;

        }

        item-list.random {

            background-color: #e9e9e9;

        }

    `]
})
export class AppComponent {
    title = 'OpenGameList';
}
```

Let's take a closer look at what we did here:

- We added two more `<item-list>` elements.
- We defined a standard `class` attribute with a different value for each instance. This is what we will use to uniquely identify each one of them. Notice that we could've used the `id` attribute, or any other standard or custom attribute. Using `class` seems to be a rather elegant choice, as it can also be used to apply different styles.
- We took the chance to implement some minimalistic CSS styles to arrange the three elements horizontally and add some space between them. Since they have different `class` attribute values now, we also gave a unique background color to each element.

We're halfway through. Now we need to change the `ItemListComponent` class accordingly, so it can follow a different behavior depending on the type of attribute value.

Here's how we can do that (added/modified lines are highlighted):

```
import {Component, Input, OnInit} from "@angular/core";
import {Item} from './item';
import {ItemService} from './item.service';

@Component({
    selector: "item-list",
    template: `

        <h2>{{title}}</h2>
```

```
        <ul class="items">
            <li *ngFor="let item of items"
                [class.selected]="item === selectedItem"
                (click)="onSelect(item)">
                <span>{{item.Title}}</span>
            </li>
        </ul>
        <item-detail *ngIf="selectedItem" [item]="selectedItem">
</item-detail>
    `,
    styles: [`
        ul.items li {
            cursor: pointer;
        }
        ul.items li.selected {
            background-color: #dddddd;
        }
    `]
})

export class ItemListComponent implements OnInit {
    @Input() class: string;

    title: string;
    selectedItem: Item;
    items: Item[];
    errorMessage: string;

    constructor(private itemService: ItemService) { }

    ngOnInit() {
        console.log("ItemListComponent instantiated with the
following type: "+this.class);

        var s = null;

        switch (this.class) {

            case "latest":

            default:

                this.title = "Latest Items";

                s = this.itemService.getLatest();

                break;

            case "most-viewed":

                this.title = "Most Viewed Items";
```

```
                s = this.itemService.getMostViewed();

                break;

            case "random":

                this.title = "Random Items";

                s = this.itemService.getRandom();

                break;

        }

        s.subscribe(

            items => this.items = items,

            error => this.errorMessage = <any>error

        );
    }

    onSelect(item: Item) {
        this.selectedItem = item;
        console.log("item with Id " + this.selectedItem.Id + " has
been selected.");
    }
}
```

We have a fair amount of changes here. Let's see what we did:

- In line 1, we added a reference to an old friend, the `Input` module from Angular 2 Core. We need it here so our class will be able to issue a data binding between the `class` input property (see line 39) and the attribute with the same name we defined in `AppComponent` within the previous code sample.
- In line 8, we replaced our static title string with a local `title` variable, thereby making it dynamic. From now on, this component's title will have to change accordingly with the `class` attribute value of the current instance.
- In line 29, we defined the `class` input property we mentioned previously. We'll be using it to determine how to initialize the current instance (see lines 37-59).
- In the subsequent lines, we implemented a brand new whole initialization logic based upon the preceding defined `class` input property. To be more specific, we switch through the three supported types and set the `title` and the `service` method accordingly.

Tip

You might have noticed that the `getLatest()` method is gone. We removed it because we transferred all the `service` method calls within the `ngOnInit()` method, so there's no need to keep it. That's a relief, as its name would also be rather obsolete now anyway.

It's time to see the results of what we just did. As usual, doing that is just as easy as hitting **F5** and waiting for our default browser to load:
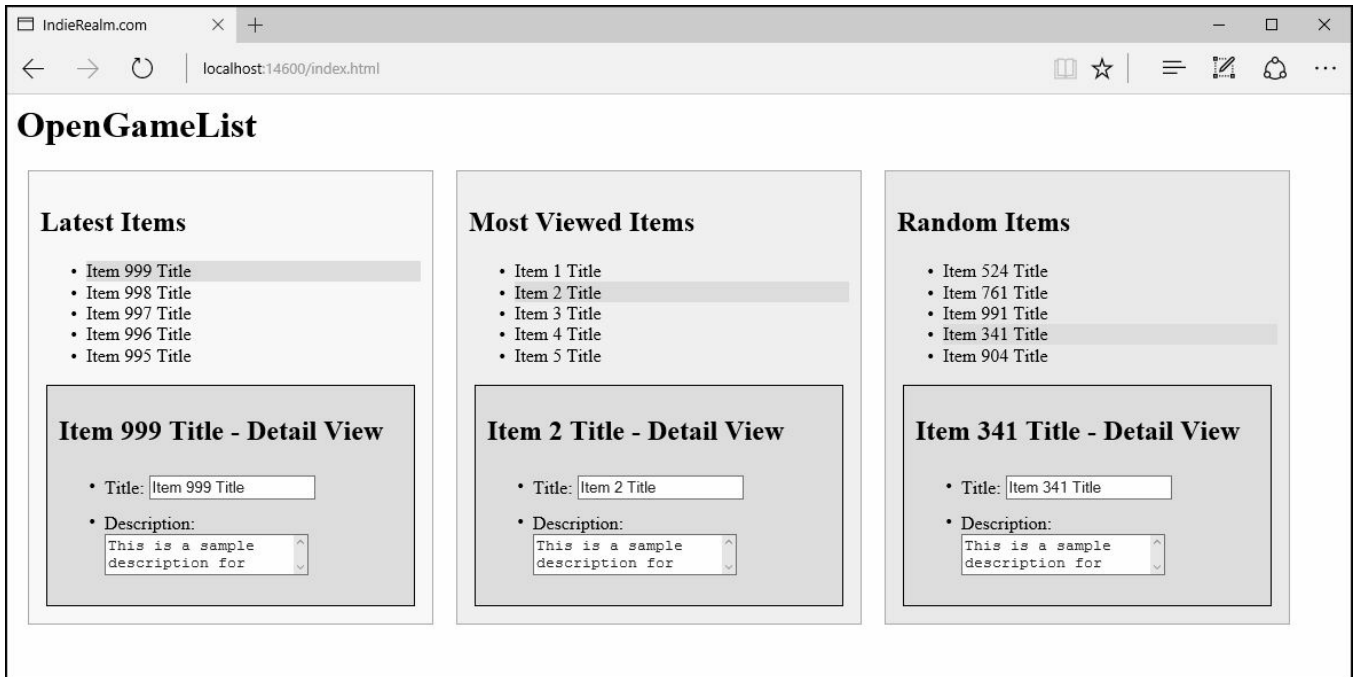
Our application is growing fast. We've got the Web API together with a small, yet versatile set of working Angular 2 components able to fetch and display our sample data.

We're still missing a fair amount of features, though, such as the following:

- Two-way binding
- Client-side routing

It's time to add both of them.

Two-way data binding

We already mentioned it a number of times, as it is one of the most convenient and widely known features of Angular 2, as well as many other reactive frameworks out there. Nonetheless, before going further, let's make sure we know what we're talking about.

Two-way data binding, also known as two-way binding, means that whenever the data model changes, the UI changes accordingly and vice versa. To be more specific:

- Whenever the model is updated, the changes are immediately reflected to the views implementing it.
- Whenever the view is updated, the changes are immediately reflected in the underlying model.

From a practical development perspective, two-way data binding will help us a lot because we won't have to manually synchronize the UI components with the data model.

The good news is, since we're using Angular 2, we're already set, and thanks to the `ngModel` we implemented earlier, our application is already equipped with a fully functional two-way data binding between the `Item` model and the two component classes implementing it: `ItemListComponents` and `ItemDetailComponent`.

We can easily check this out: hit *F5* again to run the application in the **Debug** mode, then select an item and change its `Title` property using the input textbox provided by the `ItemDetailComponent`. You will notice that any change will be immediately reflected to the `ItemListComponent` accordingly:
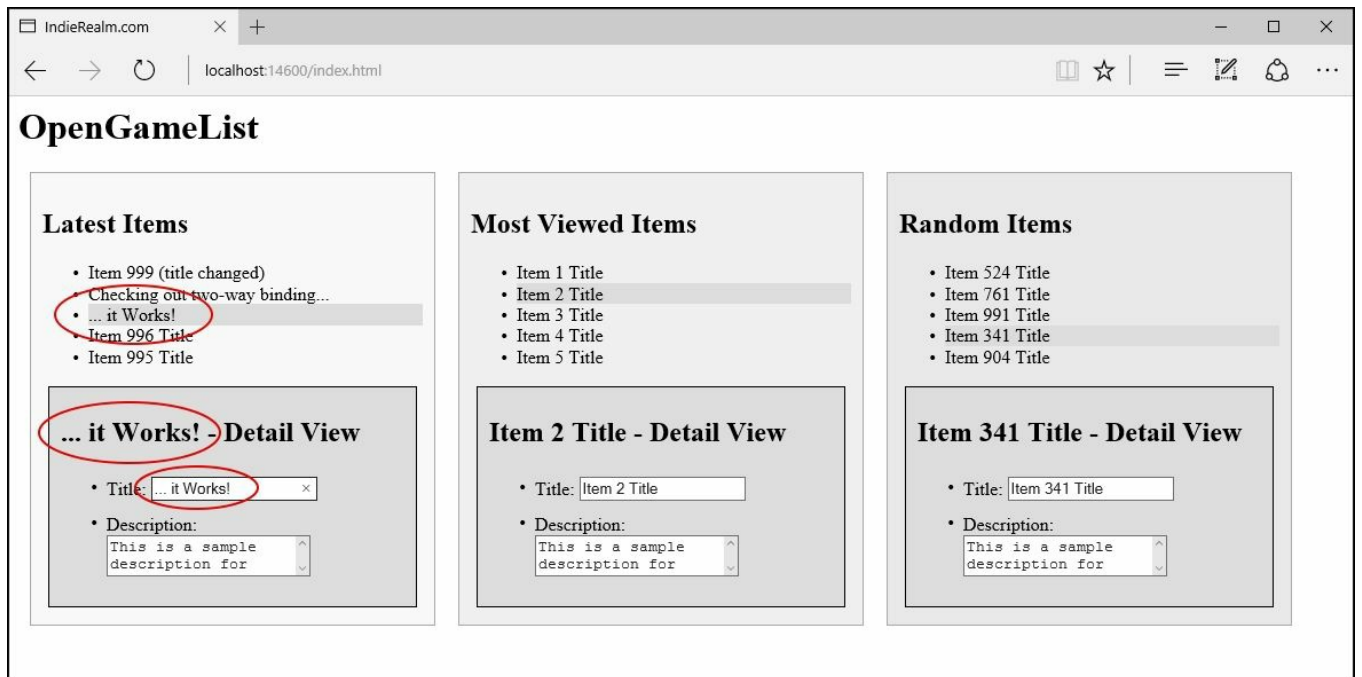
As we've already said, all of these things happen on the client side only. To persist them through the server model, we would need to implement a fully featured data source, which is something we're going to do in the next chapter.

Disabling two-way data binding

Wherever we don't want to have a two-way binding relationship, we can easily turn it off removing the parentheses around the `ngModel` directive, leaving only the brackets:

```
<input [ngModel]="item.Title" placeholder="Insert the title..." />
```

Note

The parentheses within brackets that enable two-way binding `[()]` are widely known as banana brackets. This funny name has its roots in the **Adventure in Angular** podcast episode 078, featuring Brad Green, Misko Hevery, and Igor Minar, in which they referred to that syntax by calling it a *box of bananas*". Other than being an impressive visualization, the banana-box concept greatly helps to avoid common mistakes such as placing the brackets inside the parentheses.

Client-side routing

Our master-detail relationship is indeed working, yet it has some major flaws. The current in-page navigation approach, for example, is completely different from the original plan. We wanted our users to switch back and forth between the **Welcome View** and a dedicated **Item Detail View**, but instead we're opening a detail panel under the first one. No, not just one. Three different panels, one for each item list. That doesn't make any sense! We need to fix that as soon as possible.

While doing that, we also have another severe issue to solve. You might have noticed that regardless of what we do within our app, the URL in the browser's address bar is always the same. This means that we won't be able to share, say, a URL that will directly lead to a specific item detail view, we'll be forced to share the starting URL because it is the only supported one.

Wait a minute! Isn't it our native web application's most expected behavior? This is what the single-page approach is all about after all, isn't it?

The answer is no. The single-page application approach has nothing to do with keeping an immutable URL in the browser's address bar. URLs are not pages. As the name suggests, they are unique identifiers for accessing resources.

standard web applications are usually built upon a rather small number of pages that answer to multiple URLs and serve specific contents based upon query string parameter values and/or URL rewriting techniques. single-page applications make no exceptions, as they can adapt their inner state accordingly to the request URL and also track the user navigation by updating the browser's address bar accordingly.

This technique is called client-side routing, which has the same meaning as navigation. In order to add this much-needed feature, we're going to equip our application with a router, which is the mechanism that will handle the navigation from view to view.

Adding the @angular/router library

Luckily, we won't have to implement the router from scratch as there is already an excellent routing service, known as Angular 2 Component Router that seamlessly does all the hard work. We just need to install it, since it's not part of the Angular 2 core modules. We can still get it through NPM and then import it in our application just like we did with the `@angular/forms`.

Open the `package.json` file and append the following line to the `dependencies` section, right below the `@angular/forms` package:

```
"@angular/router": "3.0.0-rc.1",
```

Right after that, open the `/Scripts/app/app.module.ts` file and add the following highlighted lines:

```
///<reference path="../../typings/index.d.ts"/>
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {HttpModule} from "@angular/http";
import {FormsModule} from "@angular/forms";
import {RouterModule}  from "@angular/router";
import "rxjs/Rx";

import {AppComponent} from "./app.component";
import {ItemListComponent} from "./item-list.component";
import {ItemDetailComponent} from "./item-detail.component";
import {ItemService} from "./item.service";

@NgModule({
    // directives, components, and pipes
    declarations: [
        AppComponent,
        ItemListComponent,
        ItemDetailComponent
    ],
    // modules
    imports: [
        BrowserModule,
        HttpModule,
        FormsModule,
        RouterModule
    ],
    // providers
    providers: [
        ItemService
    ],
    bootstrap: [
        AppComponent
```

```
    ]
})
export class AppModule { }
```

Last but not least, open the `/wwwroot/systemjs.config.js` file and update the `ngPackageNames` array accordingly:

```
var ngPackageNames = [
  'common',
  'compiler',
  'core',
  'http',
  'platform-browser',
  'platform-browser-dynamic',
  'upgrade',
  'forms',
  'router'
];
```

Now that we made the component router available, we need to properly implement and configure it within our Angular 2 application to provide it with a decent navigation pattern. While doing this we'll also take the chance to refactor our app to make it more similar to our original plan.

PathLocationStrategy versus HashLocationStrategy

Before doing that, we need to understand how the router can help us to achieve what we want, which is changing the browser's location and history whenever the user navigates within the app, without triggering a new page request to the server.

We can configure the router to follow one of two different patterns:

`PathLocationStrategy` or `HashLocationStrategy`. The former is based upon the HTML5 `history.pushState` technique, which is by far the preferable one, however, such a technique won't work well on older browsers because they will automatically send a page request every time the current `location.href` value changes, ruining the whole SPA approach, i.e., unless the change is limited to the part of the URL after the hash character (`#`).

The `HashLocationStrategy` pattern is mostly a workaround that exploits such behavior, as it will instruct the router to compose all the navigation URLs prepending them with a hash character (`#`) in the following way:

```
http://localhost:14600/app/#/item-detail/2
```

The Angular 2 routing library uses `PathLocationStrategy` by default, hence our

app will do the same. Should we prefer to take the other route, we can switch to `HashLocationStrategy` with an override during the bootstrap phase and we will see how to do that in a short while.

Refactoring our app

The other step is a bit more complex, as it involves not only changing some of our application files in order to make the router service available but also changing the component file structure to make it routing-friendly. We should start with the latter, since it will necessarily involve some minor refactoring.

Let's try to get a visual picture of what we need to do before proceeding. It's time to get rid of this working, yet rather inconsistent, cascading structure:
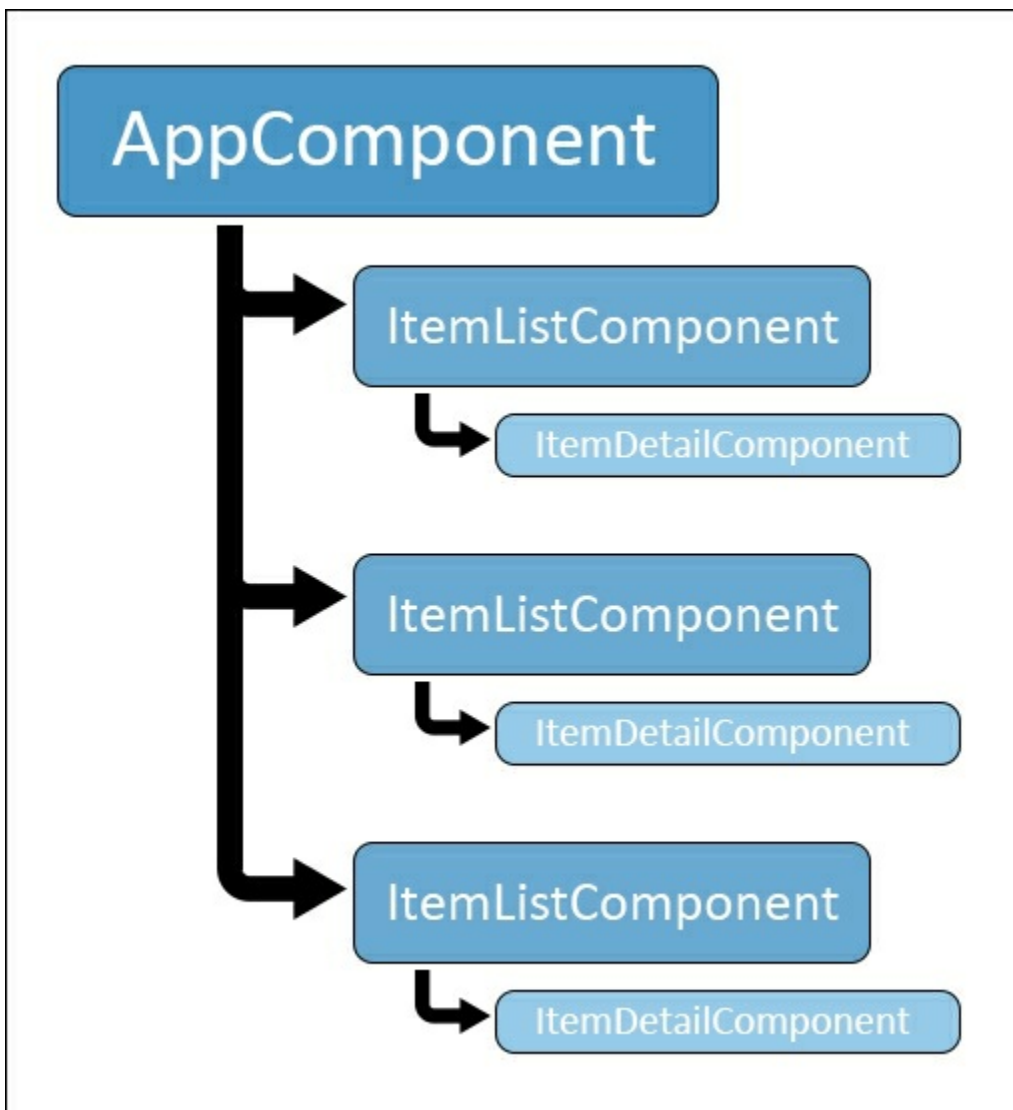
And switch to this truly navigable one:
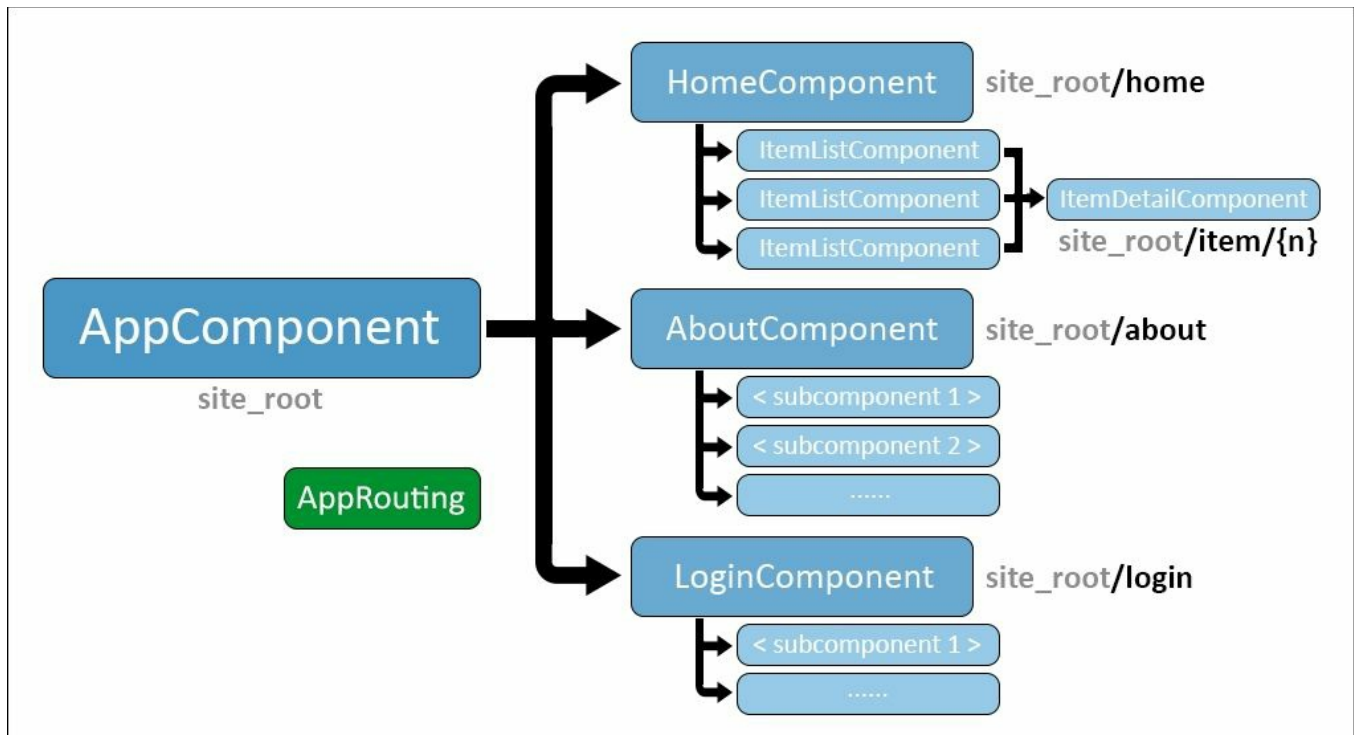
In order to achieve such a result, we need to take care of the following stuff:

- **Add an AppRouting scheme**: This is nothing more than a configuration object containing an array of routing rules called routes. Each one of them connects a URL path to a component, thus telling the client how to navigate.
- **Add a new HomeComponent**: This will behave just like the `AppComponent` does, so the latter can become a container of many different components, and configure it into the aforementioned `AppRouting` scheme.
- **Add more sample components**: This tests the routing behavior with a number of different requests and configures them into the `AppRouting` scheme as well.

Let's do it.

Adding the AppRouting scheme

From the **Solution Explorer**, right-click on the `/Scripts/app/` folder and add a new `app.routing.ts` TypeScript file. Once done, fill it with the following code:

```
import {ModuleWithProviders} from "@angular/core";
import {Routes, RouterModule} from "@angular/router";

import {HomeComponent} from "./home.component";
import {AboutComponent} from "./about.component";
import {LoginComponent} from "./login.component";
import {PageNotFoundComponent} from "./page-not-found.component";
```

```
const appRoutes: Routes = [
    {
        path: "",
        component: HomeComponent
    },
    {
        path: "home",
        redirectTo: ""
    },
    {
        path: "about",
        component: AboutComponent
    },
    {
        path: "login",
        component: LoginComponent
    },
    {
        path: '**',
        component: PageNotFoundComponent
    }
];

export const AppRoutingProviders: any[] = [
];

export const AppRouting: ModuleWithProviders =
RouterModule.forRoot(appRoutes);
```

The content is quite self-explanatory. As we can see, it's merely a list of routing rules
(we'll call them routes from now on) connecting a given path with a corresponding
component of our choice. Just by looking at them we can easily infer the underlying
logic:

1. All requests to `/` (the site root) will be routed to the `HomeComponent`.
2. All request to `/home` will be redirected to `/` (the site root).
3. All request to `/about` will be routed to the `AboutComponent`.
4. All request to `/login` will be routed to the `LoginComponent`.
5. All other requests will be routed to the `PageNotFoundComponent`.

As we can see there are two routes that behave quite differently from the other ones:

- Route #2 features a `redirect` instead of a `rewrite`. This means that it will tell
  the browser to go somewhere else instead of directly handling the request.
- Route #5 is basically a `catch-all` rule that will take care of any unmanned
  scenario, that is all the requests not directly handled by other routes.

Adding the HomeComponent

The next thing we need to do is to create a brand new `HomeComponent` that will basically behave just like the `AppComponent` actually does. The simplest way to achieve such a result is to copy our current `app.component.ts` file contents into a new `home.component.ts` file. Needless to say, since we're creating a new component, we also need to change the `selector`, `class`, and the `title` variable accordingly.

Here's how the new `home.component.ts` file should appear after our copy and replace work (relevant changes are highlighted):

```
import {Component} from "@angular/core";

@Component({
    selector: "home",
    template: `
        <h2>{{title}}</h2>
        <item-list class="latest"></item-list>
        <item-list class="most-viewed"></item-list>
        <item-list class="random"></item-list>
    `,
    styles: [`
        item-list {
            min-width: 332px;
            border: 1px solid #aaaaaa;
            display: inline-block;
            margin: 0 10px;
            padding: 10px;
        }
        item-list.latest {
            background-color: #f9f9f9;
        }
        item-list.most-viewed {
            background-color: #f0f0f0;
        }
        item-list.random {
            background-color: #e9e9e9;
        }
    `]
})

export class HomeComponent {

    title = "Welcome View";

}
```

As soon as we do that we can update the `AppComponent` file to act like a container of the currently active Component, as defined by the `AppRouting` scheme.

Open the `app.component.ts` file and entirely replace its now-replicated content with the following (relevant lines highlighted):

```
import {Component} from "@angular/core";

@Component({
    selector: "opengamelist",
    template: `
        <h1>{{title}}</h1>
            <div class="menu">

                <a class="home" [routerLink]="['']">Home</a>

                | <a class="about" [routerLink]="
['about']">About</a>

                | <a class="login" [routerLink]="
['login']">Login</a>

            </div>

        <router-outlet></router-outlet>
    `
})
export class AppComponent {
    title = "OpenGameList";
}
```

What we did here is pretty simple: we created a simple minimalistic HTML menu that we can use to test our routing engine. The `[routerLink]` attribute directive will tell the component router to populate the `<router-outlet>` element below with the component corresponding to the given path.

That's more or less how the component router works to its simplest extent. While being only a fraction of its many available features, this is already a good portion of what we need to make our application work like we want it to.

Adding new components

It's definitely time to create the missing components we defined earlier within the `app.routing.ts` file, otherwise we would be unable to run our app. These are the following components:

- `AboutComponent`
- `LoginComponent`
- `PageNotFoundComponent`

They won't be much more than a placeholder, at least for now. We will properly implement each one of them as soon as we need to.

AboutComponent

From the **Solution Explorer**, add a new `about.component.ts` file in the `Scripts/app` folder and fill it with the following code:

```
import {Component} from "@angular/core";

@Component({
    selector: "about",
    template: `
        <h2>{{title}}</h2>
        <div>
            OpenGameList: a production-ready, fully-featured SPA
sample powered by ASP.NET Core Web API and Angular 2.
        </div>
    `
})

export class AboutComponent {
    title = "About";
}
```

To be honest, we ought to say that our app is neither production-ready nor fully featured yet, but that's what we're aiming for, so a little encouragement won't hurt. It won't be a lie forever, after all!

LoginComponent

Going back to the **Solution Explorer**, add another file in that same folder, name it `login.component.ts`, and fill it with following code:

```
import {Component} from "@angular/core";

@Component({
    selector: "login",
    template: `
        <h2>{{title}}</h2>
        <div>
            TODO: Not implemented yet.
        </div>
    `
})

export class LoginComponent {
    title = "Login";
}
```

As we have already said, this is just a placeholder. There's no way we can implement the login now, as we're still missing a real, persistent data source. Rest assured, though we could arrange another dummy data provider, there's no need to do that since we'll start implementing the real deal in the following chapter.

PageNotFoundComponent

Last but not least, add a new `/Scripts/app/login.component.ts` file with the following code:

```
import {Component} from "@angular/core";

@Component({
    selector: "page-not-found",
    template: `
        <h2>{{title}}</h2>
        <div>
            Oops.. This page does not exist (yet!).
        </div>
    `
})

export class PageNotFoundComponent {
    title = "Page not Found";
}
```

Doing this will put our Angular 2 application back on track, meaning that we should be able to compile it again. However, if we try to run it, we'll hit some run-time errors due to the fact that the `AppModule` is still completely unaware of what we did. This is an expected behavior. Now that we're done adding new stuff, we need to bring our root module up to speed.

Updating the root module

Open the `/Scripts/app/app.module.ts` file and add the new content accordingly (new lines highlighted):

```
///<reference path="../../typings/index.d.ts"/>
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {HttpModule} from "@angular/http";
import {FormsModule} from "@angular/forms";
import {RouterModule}  from "@angular/router";
import "rxjs/Rx";


import {AboutComponent} from "./about.component";


import {AppComponent} from "./app.component";
```

```
import {HomeComponent} from "./home.component";
import {ItemDetailComponent} from "./item-detail.component";
import {ItemListComponent} from "./item-list.component";
import {LoginComponent} from "./login.component";
import {PageNotFoundComponent} from "./page-not-found.component";
import {AppRouting} from "./app.routing";
import {ItemService} from "./item.service";

@NgModule({
    // directives, components, and pipes
    declarations: [
        AboutComponent,
        AppComponent,
        HomeComponent,
        ItemListComponent,
        ItemDetailComponent,
        LoginComponent,

        PageNotFoundComponent
    ],
    // modules
    imports: [
        BrowserModule,
        HttpModule,
        FormsModule,
        RouterModule,
        AppRouting
    ],
    // providers
    providers: [
        ItemService
    ],
    bootstrap: [
        AppComponent
    ]
})
export class AppModule { }
```

As we can see, to ensure that our routing rules are applied we have to add the
`AppRouting` constant defined within the `app.routing.ts` file to the AppModule's
`import` section.

Doing all that will put our Angular 2 application back on track, meaning that we
should be able to compile it again.

Note

While we were there, we took the chance to separate the import references by type
and sort them into alphabetical order. This will help us to manage them, as their

number will further increase throughout the book.

Revising the master-detail strategy

We're not done yet. Our current implementation features a fully functional master-slave binding relationship between the `ItemListComponent` and the corresponding `ItemDetailComponent`. While it has been very useful to demonstrate how two-way binding works, it's time to replace it with a more reasonable, route-based navigation mechanism.

In order to do that, we need to do the following:

1. Add another route to the `AppComponent` class for the `ItemDetailComponent`.
2. Change the `ItemListComponent` behavior so it will issue a routing action towards the `ItemDetailComponent` instead of showing a built-in instance of it.
3. Modify the `ItemDetailComponent` initialization pattern to make it fetch the `item` data from the `Id` contained within the routing rule instead of relying upon an item object instantiated by the parent `ItemListComponent`.

Adding the ItemDetail route

Open the `/Scripts/app/app.routing.ts` file and add the following route to the `appRoutes` array, right below the one pointing to the `LoginComponent`:

```
{
    path: "item/:id",
    component: ItemDetailComponent
},
```

Consequently, add the relevant reference to the `import` statements on top:

```
import {ItemDetailComponent} from "./item-detail.component";
```

Changing the ItemListComponent

Right after that, open the `item-list.component.ts` file and update its code as follows (changed lines are highlighted):

```
import {Component, Input, OnInit} from "@angular/core";
import {Router} from "@angular/router";
import {Item} from "./item";
import {ItemService} from "./item.service";

@Component({
    selector: "item-list",
    template: `
        <h2>{{title}}</h2>
        <ul class="items">
```

```
            <li *ngFor="let item of items"
                [class.selected]="item === selectedItem"
                (click)="onSelect(item)">
                <span>{{item.Title}}</span>
            </li>
        </ul>
    `,
    styles: [`
        ul.items li {
            cursor: pointer;
        }
        ul.items li.selected {
            background-color: #dddddd;
        }
    `]
})

export class ItemListComponent implements OnInit {

    selectedItem: Item;
    @Input() class: string;
    title: string;
    items: Item[];
    errorMessage: string;

    constructor(private itemService: ItemService, private router:
Router) { }

    ngOnInit() {
        console.log("ItemListComponent instantiated with the
following class: " + this.class);
        var s = null;
        switch (this.class) {
            case "latest":
            default:
                this.title = "Latest Items";
                s = this.itemService.getLatest();
                break;
            case "most-viewed":
                this.title = "Most Viewed Items";
                s = this.itemService.getMostViewed();
                break;
            case "random":
                this.title = "Random Items";
                s = this.itemService.getRandom();
                break;
        }
        s.subscribe(
            items => this.items = items,
            error => this.errorMessage = <any>error
        );
    }
```

```
    onSelect(item: Item) {
        this.selectedItem = item;
        console.log("Item " + this.selectedItem.Id + " has been
clicked: loading ItemDetailComponent...");

        this.router.navigate(["item", this.selectedItem.Id]);
    }
}
```

We have a small number of important changes here:

- Within the class constructor, we defined a new DI for an object of the type `Router` to consume within our class. In order to use that, we also added the required `import` reference at the beginning of the file.
- Near the end of the file, we changed the implementation of the `OnSelect` event delegate to tell the router to trigger the `ItemDetail` route instead of loading a local `ItemDetailComponent` as we were doing before.
- Consequently, we removed the `<item-detail>` tag from the template HTML code accordingly.

Updating the ItemDetailComponent

The last thing we need to change is the `ItemDetailComponent` behavior, as it won't be able to get its required `item` data from a parent `ItemListComponent` anymore. We need to make it be able to fetch the data by itself using the only parameter it will receive from its routing entry, the unique `Id` of the `item` to display.

In order to do that, we need to add a couple of references at the beginning of the file:

```
import {Router, ActivatedRoute} from "@angular/router";
```

This way we'll be able to get the `Id` from the request URL and issue a call to the `itemService.get(id)` method to retrieve the item accordingly. That's why we also need to add a reference to the `ItemService` itself:

```
import {ItemService} from "./item.service";
```

What else do we need to do? Since we're removing the parent property binding, we can safely remove the reference to the `Input` interface module as well as the `@Input` decorator from our local `item` variable. We're not using them here anymore.

Yet we will need to retrieve the item data, which is something that's normally done during the initialization phase. That's why we need to add a reference to the `OnInit` interface we used before:

```
import {Component, OnInit} from "@angular/core";
```

We will also need a local variable to host the retrieved item, an improved constructor to instantiate an `ActivatedRoute` object instance through DI and an `OnInit` method to consume it when the class is initialized:

```
export class ItemDetailComponent {
    item: Item;

    constructor(private itemService: ItemService,
        private activatedRoute: ActivatedRoute) {
    }

    ngOnInit() {
        var id = +this.activatedRoute.params['id'];
        if (id) {
            this.itemService.get(id).subscribe(item => this.item =
item);
            console.log(this.item);
        }
    }
}
```

Here's the full, updated `ItemDetailComponent` code with the relevant new/updated lines highlighted:

```
import {Component, OnInit} from "@angular/core";
import {Router, ActivatedRoute} from "@angular/router";
import {Item} from "./item";
import {ItemService} from "./item.service";

@Component({
    selector: "item-detail",
    template: `
        <div *ngIf="item" class="item-details">
          <h2>{{item.Title}} - Detail View</h2>
          <ul>
              <li>
                  <label>Title:</label>
                  <input [(ngModel)]="item.Title"
placeholder="Insert the title..."/>
              </li>
              <li>
                  <label>Description:</label>
                  <textarea [(ngModel)]="item.Description"
placeholder="Insert a suitable description..."></textarea>
              </li>
          </ul>
        </div>
    `,
    styles: [`
        .item-details {
            margin: 5px;
```

```
            padding: 5px 10px;
            border: 1px solid black;
            background-color: #dddddd;
            width: 300px;
        }
        .item-details * {
            vertical-align: middle;
        }
        .item-details ul li {
            padding: 5px 0;
        }
    `]
})

export class ItemDetailComponent {
    item: Item;

    constructor(private itemService: ItemService,

        private router: Router,

        private activatedRoute: ActivatedRoute) {

    }

    ngOnInit() {

        var id = +this.activatedRoute.snapshot.params["id"];

        if (id) {

            this.itemService.get(id).subscribe(

                item => this.item = item

            );

        }

        else {

            console.log("Invalid id: routing back to home...");

            this.router.navigate([""]);

        }

    }
}
```

The use of the injected `ActivatedRoute` object instance here is a bit cryptic, so it

deserves a brief explanation.

As we can easily see, it is an object containing information about route parameters, query parameters and URL fragments for the currently active route. In order to access the id query parameter, we need to look into the params property, which happens to be an Observable object. This basically means that we would normally need to subscribe to it in the following way:

```
this.activatedRoute.params.subscribe(
    params => {
        let id = +params['id'];
        // do something with id
    });
```

This would indeed work, however, we were able to retrieve the id parameter using less code and avoiding the Observable entirely thanks to the snapshot property, which returns a flattened representation of the currently active route. As a general rule of thumb, we can and should use the snapshot whenever we don't need to actively monitor the Observable changes.

Note

Observables are one of the most interesting features introduced by Angular 2. We'll talk more about them in Chapter 5, *Persisting Changes*.

This ItemDetailComponent update was the last thing we needed to do to finalize our refactoring task. The revamp of our app's client-side navigation is now complete. Before going further, let's perform a full routing and navigation test to see if everything is looking good up to this point.

Full routing test

It's time to hit **F5** and see if our refactoring worked out well. If we did everything correctly, we should be greeted with something like this:
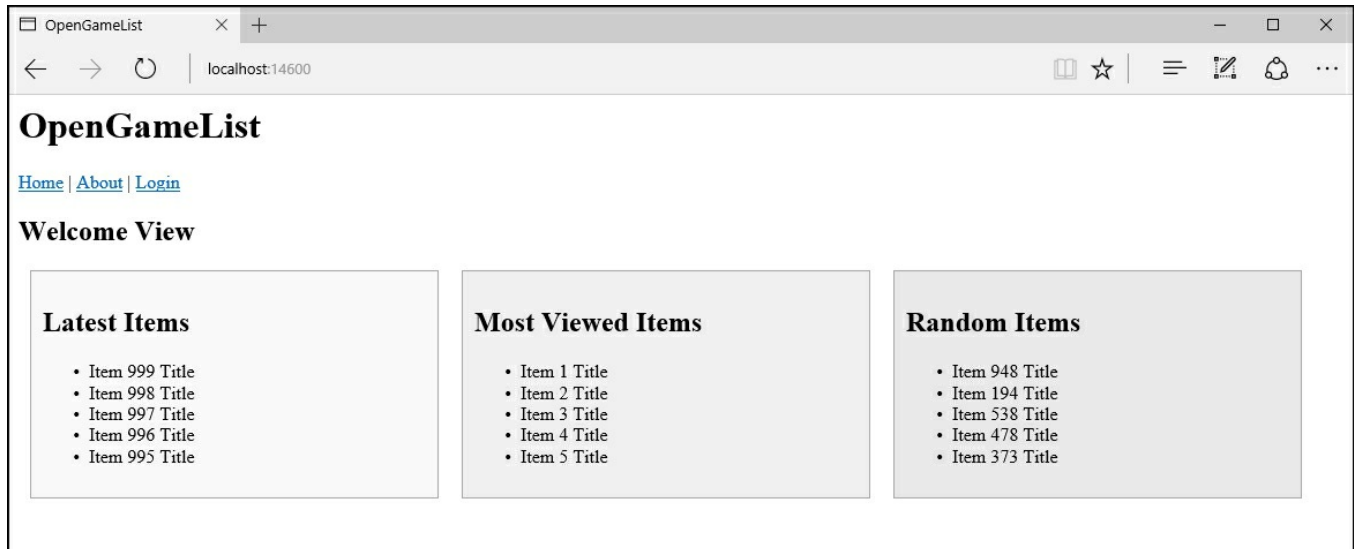
That's definitely the **Welcome View** we wanted. Let's see if the improved master-detail navigation pattern is working as well by left-clicking on one of the items. The view should change, displaying the item detail data.
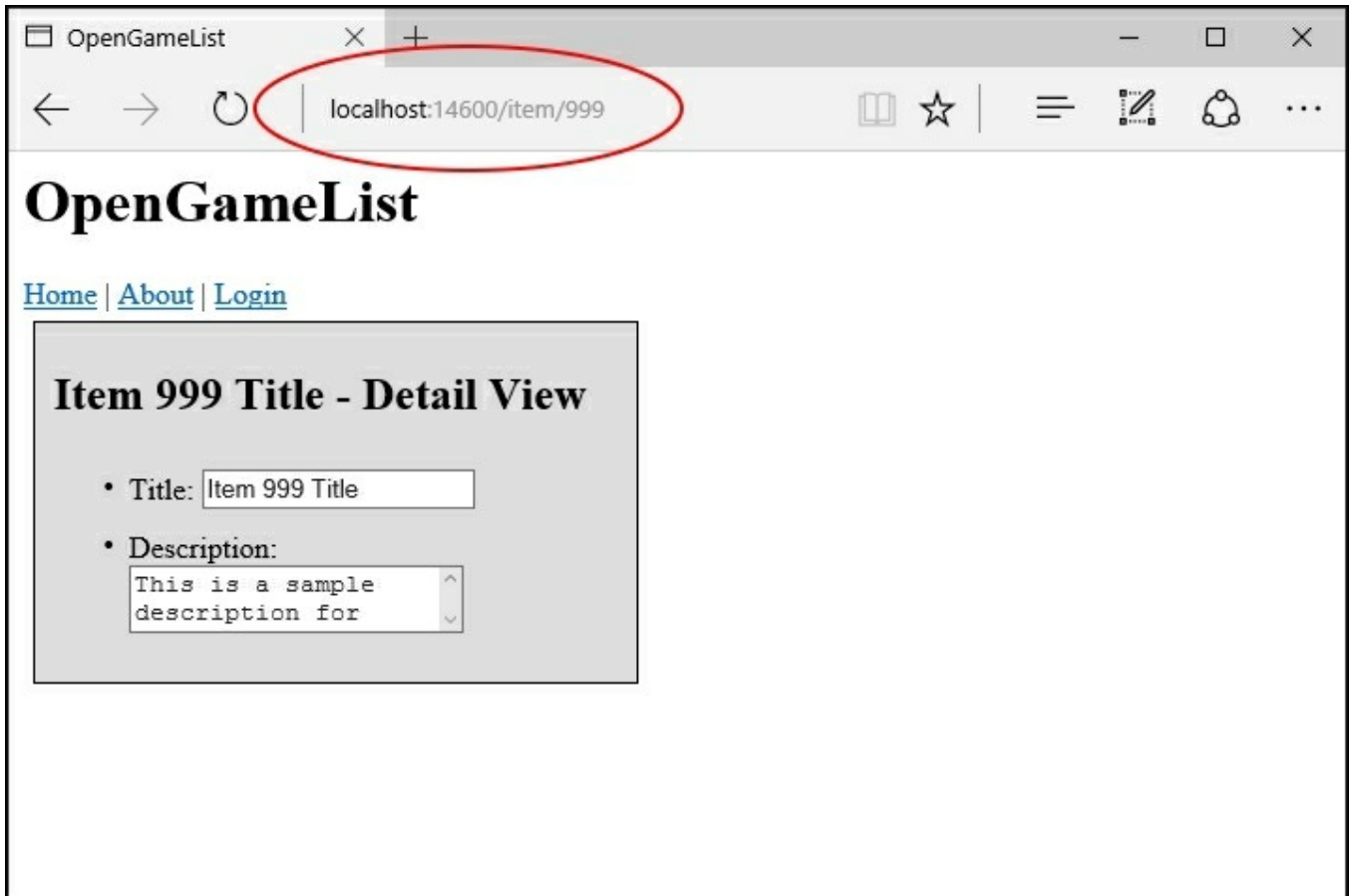
Notice how the URL in the address bar properly switches from `localhost` to `localhost/item/{n}`, reflecting the user navigation up to this point.

Since the master-detail route is working well, let's test our new navigation menu. Clicking on the **About** link element should update our page in the following way:
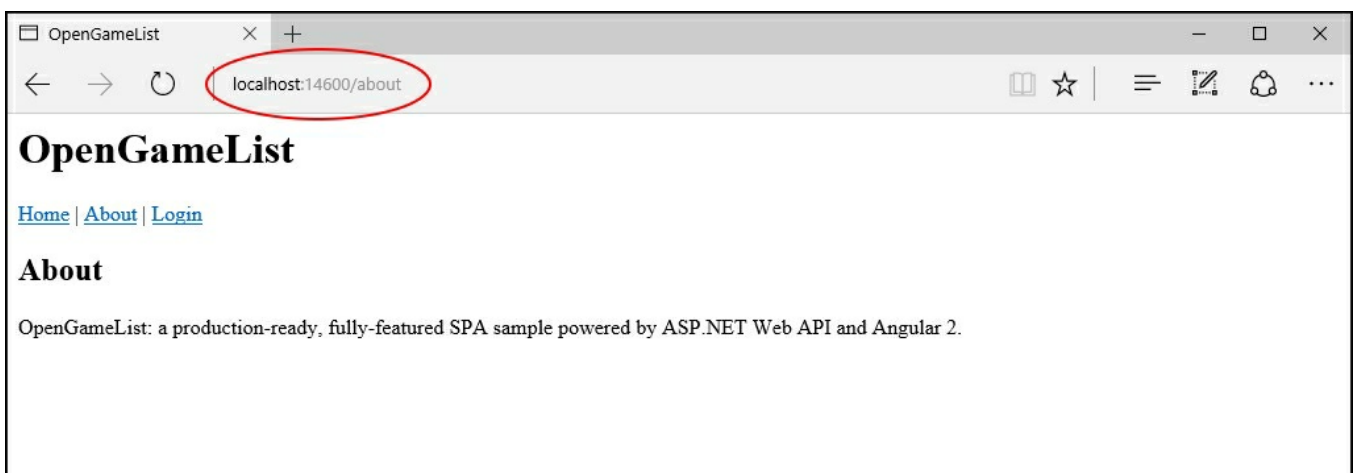
If this all works, there's no reason why the **Login** link shouldn't be working as well:
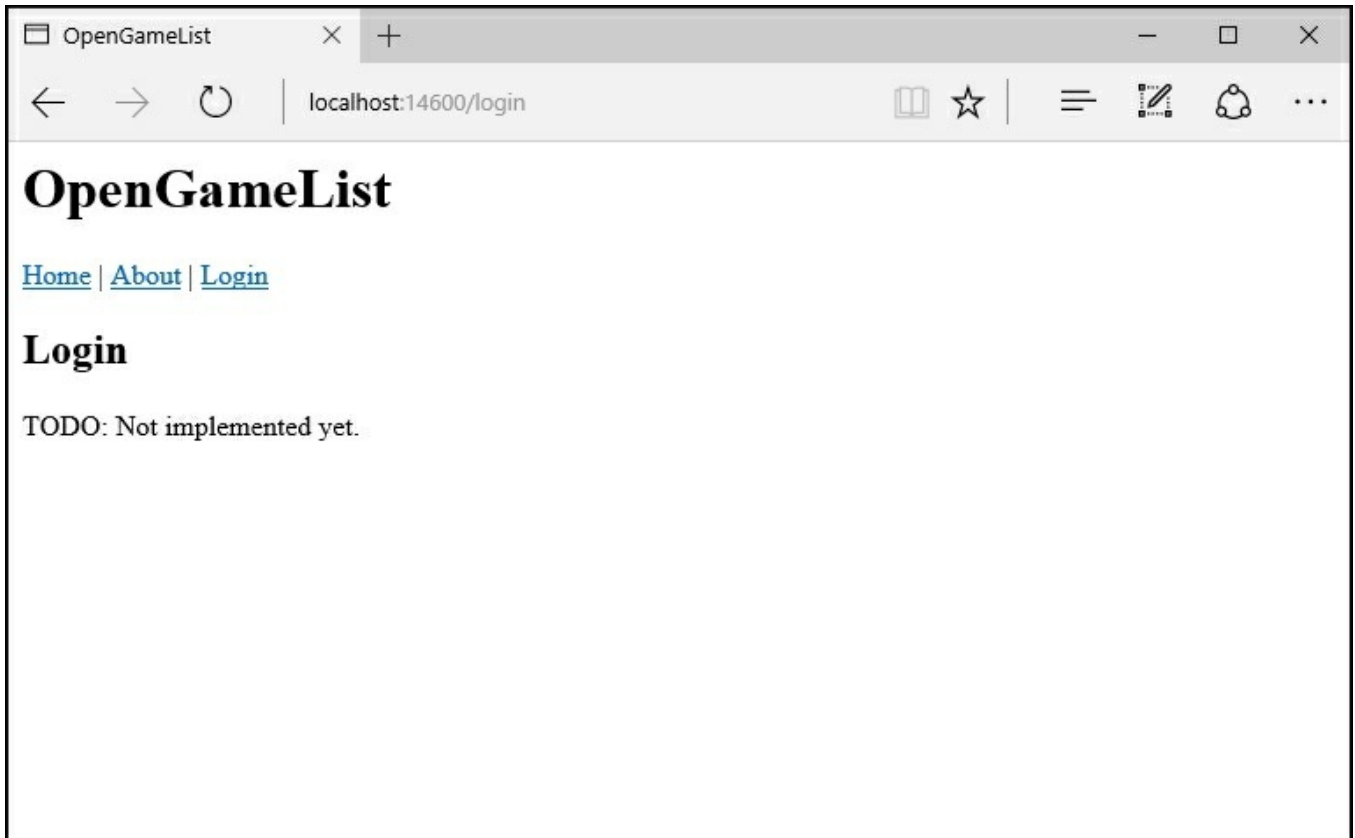
Finally, we can go back to the Home view and bring back the initial layout:
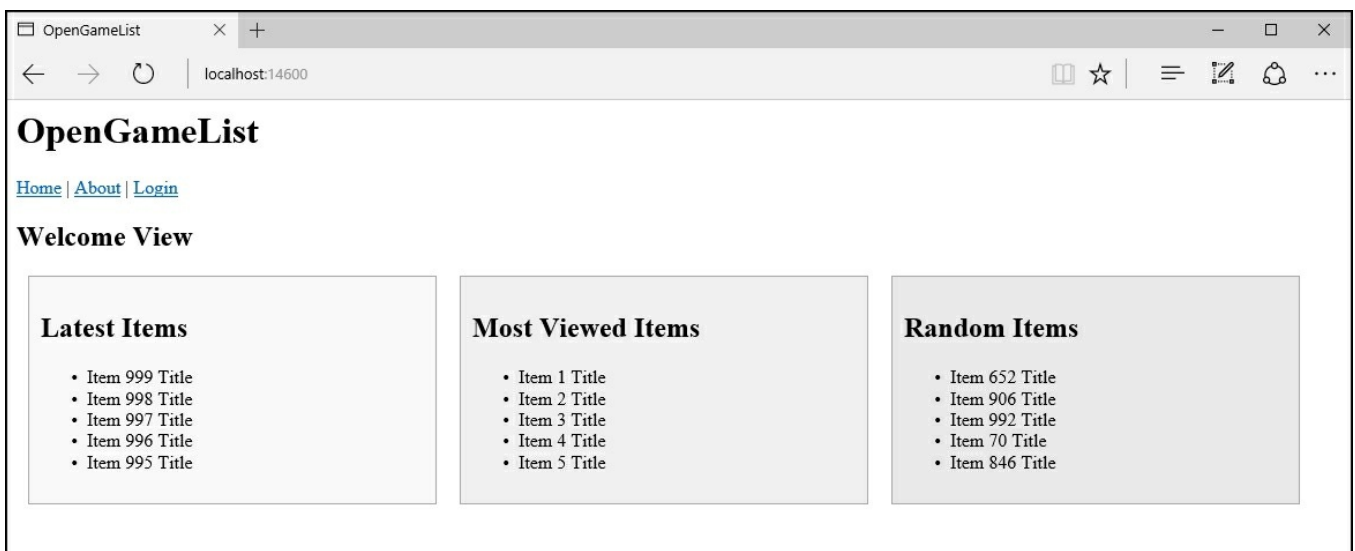
If we look closely at this welcome page we'll be able to see a small difference with the former one. The **Random Items** listing is showing different items. This is

working as indented since we want our app to always refresh that content by issuing a new API call each time. Neither Angular 2 nor the Web API will serve cached content unless we instruct them to do so.

Tip

Conversely, the web server and/or the browser could definitely do that due to their default behavior for static files, that's why we explicitly disabled file caching in Chapter 1, *Getting Ready*.

Handling rewrites in ASP.NET Core

In for a penny, in for a pound, there's still an issue we need to address before going further. We address that within our final test as it's something that can be better demonstrated here.

Showing the issue

We might notice that, if we issue a browser refresh by hitting *F5* on any page, we'll end up with an **HTTP 404 error** (page not found). This is what happens when we try this from the **About** view:
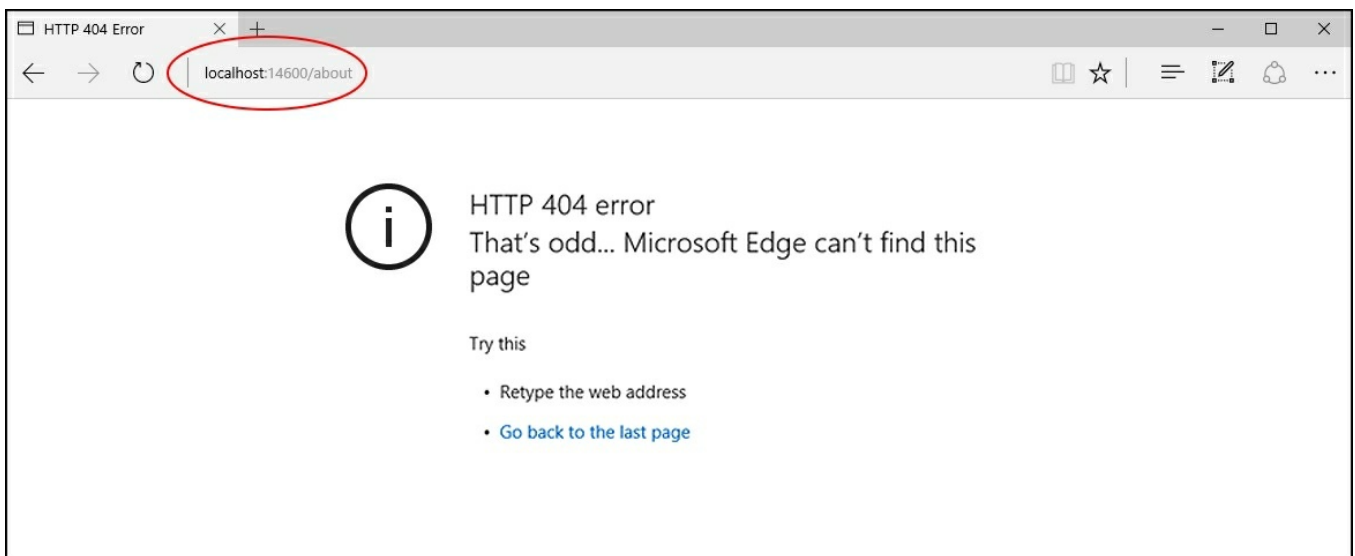
The same problem will occur whenever we try to refresh any of our app's URL including the **Item Detail View**, the **Login View**, and even the **Welcome View**. This is unacceptable for a production-ready app because our users won't be able to properly bookmark or share anything from our app, the sole exception being the `localhost/index.html` default starting URL.

Understanding the causes

The reasons for this are very simple. We have never told our web application how to properly handle these URLs; hence, it follows the default behavior: it looks for a matching `HttpModule` and checks the filesystem for a file or folder having the same name, then it gives up by returning an **HTTP 404 error** (page not found).

You might wonder why the issue is not affecting the **Welcome View**. If we navigate back there, then manually remove the `index.html` part and hit *F5* right after that, it

will still work like a charm.

The reason for that is simple. Remember that `app.UseDefaultFiles()` line of code we added in the `Setup.cs` file back in Chapter 1, *Getting Ready*? That's it! Our `index.html` is clearly a default file, meaning that the web application will look for it even if it's not part of the requesting URL. Unfortunately, this behavior won't be able to solve the issue on any internal page. In order to fix that, we need to find a way to implement a custom set of URL rewriting rules.

Implementing the fix

In for a penny, in for a pound. We can tell our web server to rewrite all routing URLs, including the root one, to the `index.html` file by adding the following lines to the `<system.webServer>` section of our app's root `web.config` file:

```
<rewrite>
  <rules>
    <rule name="Angular 2 pushState routing" stopProcessing="true">
      <match url=".*" />
      <conditions logicalGrouping="MatchAll">
        <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />
        <add input="{REQUEST_FILENAME}" pattern=".*\.[\d\w]+$"
negate="true" />
        <add input="{REQUEST_URI}" pattern="^/(api)" negate="true"
/>
      </conditions>
      <action type="Rewrite" url="/index.html" />
    </rule>
  </rules>
</rewrite>
```

By implementing these rules we're basically asking our web server (IIS or IIS Express) to readdress any incoming request to the `/index.html` file, with the sole exception of those pointing to:

- Any existing file, to preserve references to actual `.js`, `.css`, `.pdf`, image files, and more.
- Any existing folder, to preserve references to actual, potentially browsable and/or Angular-unrelated subfolders.
- Any URL starting with `/api`, to preserve any call to our Web API Controllers.

It's worth mentioning that this implementation will only work under IIS if the URL Rewrite Module is properly installed. Conversely, IIS Express won't have this issue since that module is bundled as a built-in feature with all the latest versions.

Note

We won't be digging into the URL Rewrite Module or URL Rewriting anytime soon. If you want to retrieve additional info regarding how it works and/or have a better grip of its `rule`/`match`/`conditions` syntax, we suggest reading the following official MSDN documentation page:

http://www.iis.net/learn/extensions/url-rewrite-module/creating-rewrite-rules-for-the-url-rewrite-module.

Here's how the `web.config` file will look after these changes (new lines are highlighted):

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!--
    Configure your application settings in appsettings.json. Learn
more at http://go.microsoft.com/fwlink/?LinkId=786380
  -->
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModule" resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%"
arguments="%LAUNCHER_ARGS%" stdoutLogEnabled="true"
stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false" />
    <rewrite>

      <rules>

        <rule name="Angular 2 pushState routing"
stopProcessing="true">

          <match url=".*" />

          <conditions logicalGrouping="MatchAll">

            <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />

            <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />

            <add input="{REQUEST_FILENAME}" pattern=".*\.[\d\w]+$"
negate="true" />

            <add input="{REQUEST_URI}" pattern="^/(api)"
negate="true" />
```

```
        </conditions>

        <action type="Rewrite" url="/index.html" />

      </rule>

    </rules>

  </rewrite>
  </system.webServer>
</configuration>
```

Validating the outcome

Time to check out if the preceding implementation solves our 404 problems or not.
Run the app again in debug mode (by hitting **F5** from Visual Studio IDE), wait for the
browser to load the **Welcome View**, then refresh it by hitting **F5** . This time, instead
of an **HTTP 404 error** (page not found), you should be able to see an actual page
reload with the same contents as before (except for **Random Items**, as they will be
reshuffled as always). Move to the **About View** and do the same, then follow-up with
the **Login View** and the **Item Detail View** as well.

Suggested topics

XHR, two-way data binding, master-detail navigation patterns, `PathLocationStrategy, HashLocationStrategy, location.pushState`, URL Rewrite Module, URL Rewriting, `JavaScriptServices`.

Summary

A lot of things have happened here. We turned our attention to the client-side components of our app, switching the focus from the Web API to Angular 2; we chose to implement a **Welcome View** featuring multiple listings of clickable items, giving our users the chance to navigate to their detail page through a classic master-detail relationship.

To achieve such a result we created a bunch of Angular components: the `Item` class for the model, the `ItemService` class to retrieve the required data from the Web API and the `ItemListComponent` class to show them; then we connected them all using our already-existing `AppComponent`, updating it accordingly; we also added multiple lists to the **Welcome View**, improving our `ItemListComponent` and turning it into a versatile, reusable class.

Everything we did was indeed working, but it didn't match our expectations in terms of seamless navigation between views; we chose to address this issue with the help of the Angular 2 routing library; implementing the library triggered a major refactor of our components, which helped us to greatly improve our previous master-detail approach.

As soon as we had built our improved, navigation-based pattern we performed a final test to see if everything was working as expected. It turned out that our web server couldn't handle our new client-side routes, so we added a set of URL Rewrite rules to the `web.config` file to fix the issue.

Within the previous chapter we implemented a minimalistic, dummy-based, yet functional Web API; in this chapter, we built an unpolished, ugly-looking yet working Angular 2 client-side app. In the following chapters, we'll address these flaws by adding a data model, an improved set of controller methods and also some styling to improve the UI appearance.

Chapter 4. The Data Model

Our app is growing fine, yet it's starting to show its limits. There's no way we can add, update or delete our items, or properly implement our **Login View**, since it would require handling some sort of user authentication in terms of credential storage and session persistence, to say the least. Truth be told, we can't even say we're actually showing something close to our original plan. Our items don't resemble open-source game entries at all, they're more like a generic container put together in a rather random fashion by a sample, method acting as a Dummy Data Provider.

It's time to get rid of that provisional demo and start working on the real thing. We won't use Angular for that, as what we need to implement has little or nothing to do with the *client-side* portion of our app. Nonetheless, we're fully aware of the fact that most *entities* of the **Data Model** we're about to build will have their correspondence in an Angular model class, just like we did with the C# **ItemViewModel** and the TypeScript **Item** classes, as long as we don't forget we're doing this to feed Angular, we'll be good.

Getting ready

We need to do a lot of things here, so it's better to avoid wasting our time by introducing the whole data model concept, as well as the various meanings of these two words. The experienced reader, as well as the seasoned developer, will be most likely aware of all the relevant stuff. We'll just say that, when we're talking about data model, we don't mean anything more or anything less than a lightweight, definitely-typed set of entity classes representing persistent, code-driven data structures that we can use as resources within our Web API code.

We used the term *persistent* here for a reason: we want our data structure to be stored in a database. That's rather obvious for any application based upon data. `OpenGameList`, won't be an exception since we want it to act as a directory of open source games, which is more than just requiring a database, our app is basically a database by itself.

Installing the EntityFramework Core

We're going to create ours with the help of the **EntityFramework Core** (**EF Core**), the well-known open-source **object-relational mapper** (**ORM**) for ADO.NET developed by Microsoft. The reasons for such a choice are many:

- Seamless integration with the Visual Studio IDE
- A conceptual model based upon entity classes (**Entity Data Model** (**EDM**)) that will enable us to work with data using domain-specific objects without the need to write data-access code, which is precisely what we're looking for
- Easy to deploy, use and maintain in development and production phases
- Compatible with all the major open-source and commercial SQL-based engines, including MSSQL, MySQL, PostgreSQL, Oracle, and more, thanks to the official and/or third-party EF-compatible connectors available via NuGet

Note

It's worth mentioning that the EF Core was previously known as EntityFramework 7 until its latest RC release. The name change follows the ASP.NET 5/ASP.NET Core perspective switch we've already talked about, as it also emphasizes the EF Core major rewrite/redesign.

You might be wondering why we're choosing to adopt a SQL-based approach instead of going for a NoSQL alternative. There are many good NoSQL products such as MongoDB, RavenDB, and CouchDB that happen to have a C# connector library. What about using one of them instead?

The answer is rather simple: they are not supported yet by the EF Core 1.0.0, which, at the time of writing, happens to be the latest stable release. If we look at the EF Core team backlog we can see that non-relational database providers, such as Azure Table Storage, Redis, and others, are indeed mentioned for upcoming support, but it's unlikely that we'll be able to see any of them implemented within the EF Core's future releases as well.

Note

If you want to know more about the upcoming release, and/or if you feel bold enough to use it anyway maybe with a NoSQL DB as well, we suggest reading more about the EF Core project status by visiting the following links:

- **Project roadmap**:

  https://github.com/aspnet/EntityFramework/wiki/Roadmap
- **Source code on GitHub**:

https://github.com/aspnet/EntityFramework
- **Official documentation**:

  https://docs.efproject.net/en/latest/

In order to install the EF Core, we need to add the relevant packages to the `dependencies` section of our `project.json` file. We can easily do that using the visual GUI like we did for the `Newtonsoft.json` package in the following way:

1. Right-click on the `OpenGameListWebApp` project.
2. Select **Manage NuGet Packages**.
3. Ensure that the **Package source** drop-down list is set to **All** and also that the **Include prerelease** switch is checked.
4. Go to the **Browse** tab and search for the packages containing the `EntityFrameworkCore` keyword.
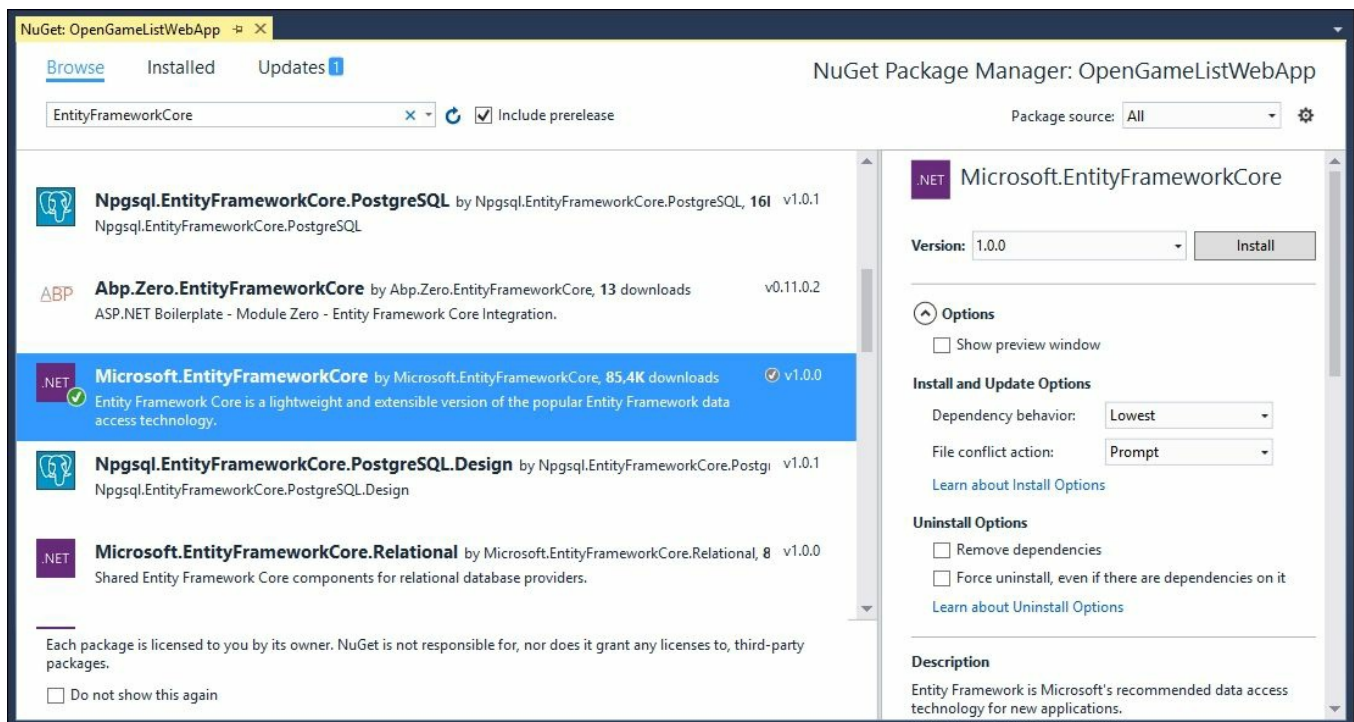
Next, install the following packages:

- `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`
- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.EntityFrameworkCore`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Design`

Alternatively, we could add the relevant packages manually within the `project.json` file as follows (new lines highlighted):

```
  "dependencies": {
    "Microsoft.AspNetCore.Mvc": "1.0.0",
    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
    "Microsoft.Extensions.Configuration.EnvironmentVariables":
"1.0.0",
    "Microsoft.Extensions.Configuration.FileExtensions": "1.0.0",
    "Microsoft.Extensions.Configuration.Json": "1.0.0",
    "Microsoft.Extensions.Logging": "1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.Extensions.Logging.Debug": "1.0.0",
    "Microsoft.Extensions.Options.ConfigurationExtensions":
"1.0.0",

    "Microsoft.AspNetCore.Diagnostics": "1.0.0",
    "Microsoft.AspNetCore.Routing": "1.0.0",
    "Microsoft.AspNetCore.Authentication.JwtBearer": "1.0.0",
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0",
    "Newtonsoft.Json": "9.0.1",

    "Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore":
"1.0.0",

    "Microsoft.AspNetCore.Identity.EntityFrameworkCore": "1.0.0",

    "Microsoft.EntityFrameworkCore": "1.0.0",

    "Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",

    "Microsoft.EntityFrameworkCore.Design": "1.0.0-preview2-final",
  }
```

Note

At the time of writing, the latest released version of EF Core was 1.0.0, with the sole exception of `Microsoft.EntityFrameworkCore.Design`, which is still in `1.0.0-preview2`. We're going to use these builds in our project.

As usual, the advantage of doing it manually is that we can keep the packages more organized by separating them into commented blocks.

Among the installed namespaces we can easily see the presence of the `EntityFrameworkCore.SqlServer`, a highly versatile connector providing an interface with the whole MSSQL server database family: SQL Server 2008-2014, as well as the Express and Compact editions for personal and development usage.

We're free to choose between using one of them and picking another DBMS engine such as MySQL, PostgreSQL, or any other EF-compatible product. Should we take this decision now? It entirely depends on the data modeling approach we want to adopt. For the time being and for the sake of simplicity, we'll be sticking to it.

Data modeling approaches

Now that we have EF installed, we have to choose between one of the three available approaches to model the data structure: model-first, database-first and code-first. Each one of them comes with its fair amount of advantages and disadvantages, as the experienced readers and seasoned .NET developers will most certainly know. While we won't dig too much into these, it could be useful to briefly summarize each one of them before making the choice.

The model-first approach

If you're not familiar with the Visual Studio IDE design tools such as the XML-based **DataSet Schema** (**XSD**) and the **Entity Designer Model XML** (**EDMX**) visual interface, the model-first approach can be rather confusing. The key to understanding it is to acknowledge the fact that the word model here is meant to define a visual diagram built with the design tools. This diagram will then be used by the framework to autogenerate the database SQL script and the data model source code files.

To summarize, we can say that choosing the model-first option means working on a visual EDMX diagram and letting EF create/update the rest accordingly.
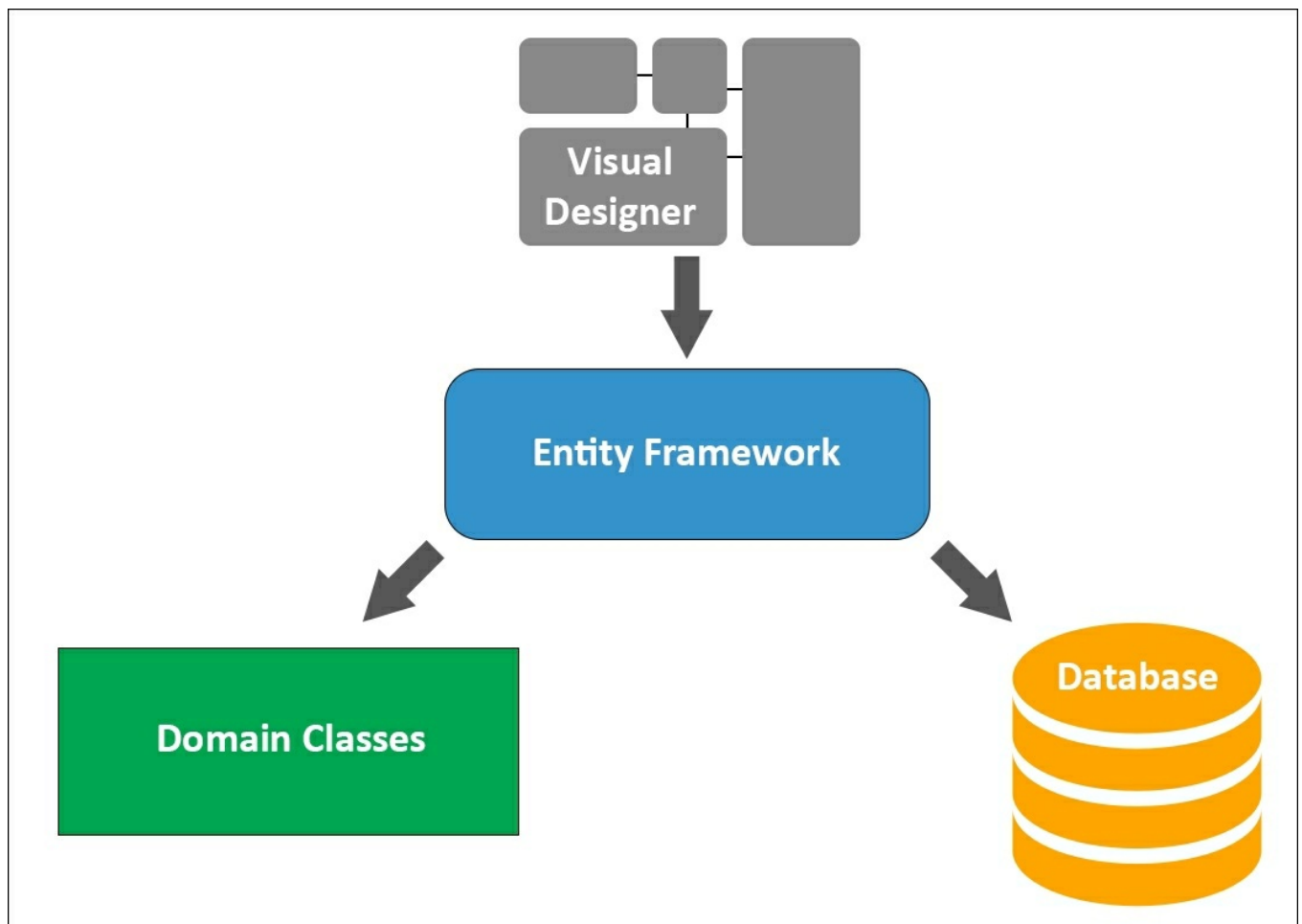
Pros

- We will be able to create the database scheme and the class diagram as a whole using a visual design tool, which could be great when the data structure is quite big.
- Whenever the database changes, the model can be updated accordingly without data loss.

Cons

- The diagram-driven, autogenerated SQL scripts could lead to data loss in case of updates. An easy workaround for that would be generating the scripts on disk and manually modifying them, which would require decent SQL knowledge.
- Dealing with the diagram can be tricky, especially if we want to have precise control over our model classes. We wouldn't always be able to get what we want, as the actual source code would be auto-generated by a tool.

The database-first approach

Given the disadvantages of the model-first approach, we might think that the database-first approach might be the way to go. This could be true if we either have a database already or don't mind building it beforehand. That being the case, the database-first approach is similar to the model-first one, except it goes the other way around. Instead of designing EDMX manually and generating the SQL script to create the database, we build the latter and then generate the former using the ED Designer tool.

We can summarize it by saying that choosing the Database-first approach means building the database and letting EF create/update the rest accordingly.
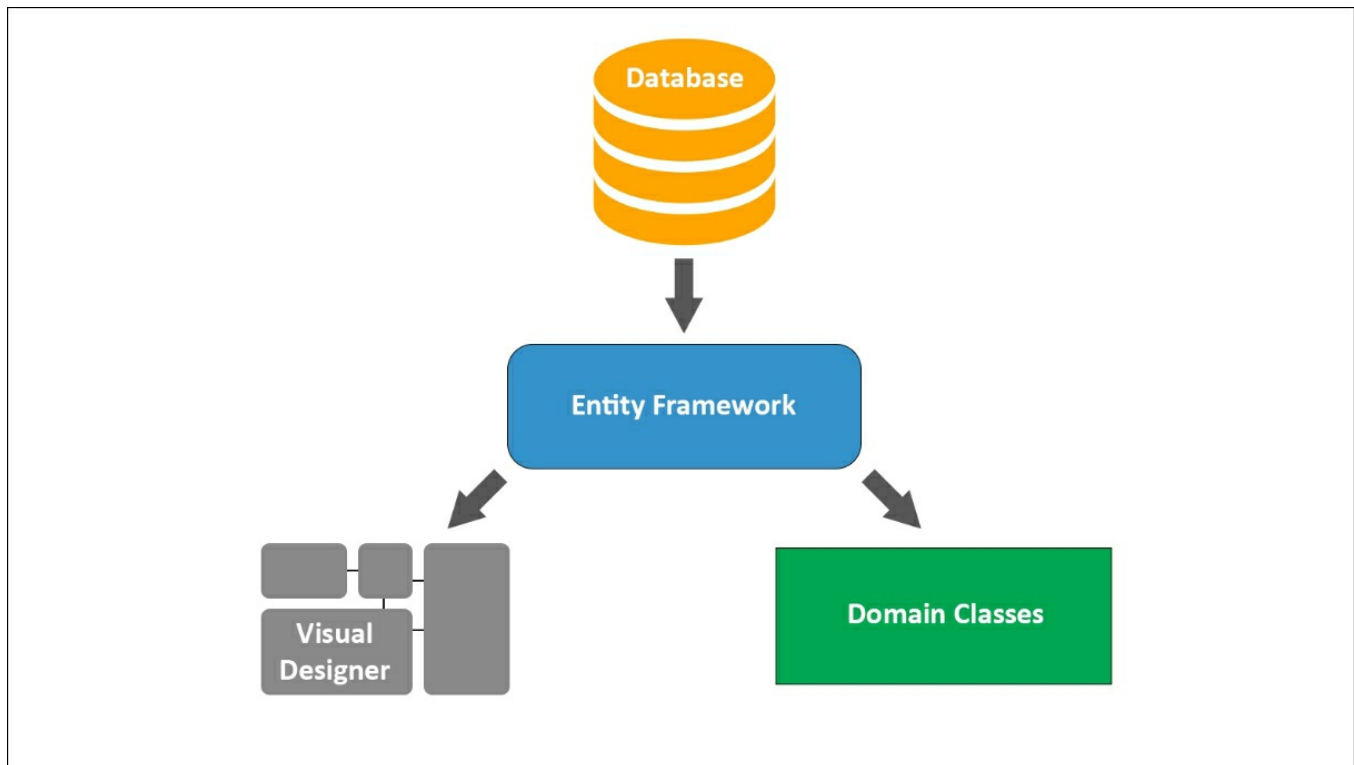
The database-first approach

Pros

- If we have an already-existing database in place, this will most likely be the way as it will spare us the need to re-create it.
- The risk of data-loss will be kept to a minimum because any change or update will be always performed on the database.

Cons

- Manually updating the database can be tricky if we're dealing with clusters, multiple instances or a number of development/testing/production environments, as we will have to manually keep them in sync instead of relying upon code-driven updates/migrations or autogenerated SQL scripts.
- We would have even less control over the autogenerated model classes (and their source code) than if we were using the model-first approach. It would require an extensive knowledge of EF conventions and standards, otherwise, we'll often struggle to get what we want.

The code-first approach

Last but not least comes the EF flagship approach since EF4, which enables an elegant, highly efficient data model development workflow. The appeal of this approach can easily be found in its premise: the code-first approach allows the

developer to define model objects using only standard classes, without the need for any design tool, XML mapping files or cumbersome piles of autogenerated code.

To summarize, we could say that choosing the code-first approach means writing the **Data Model** entity classes we'll be using within our project and letting EF generate the database accordingly.
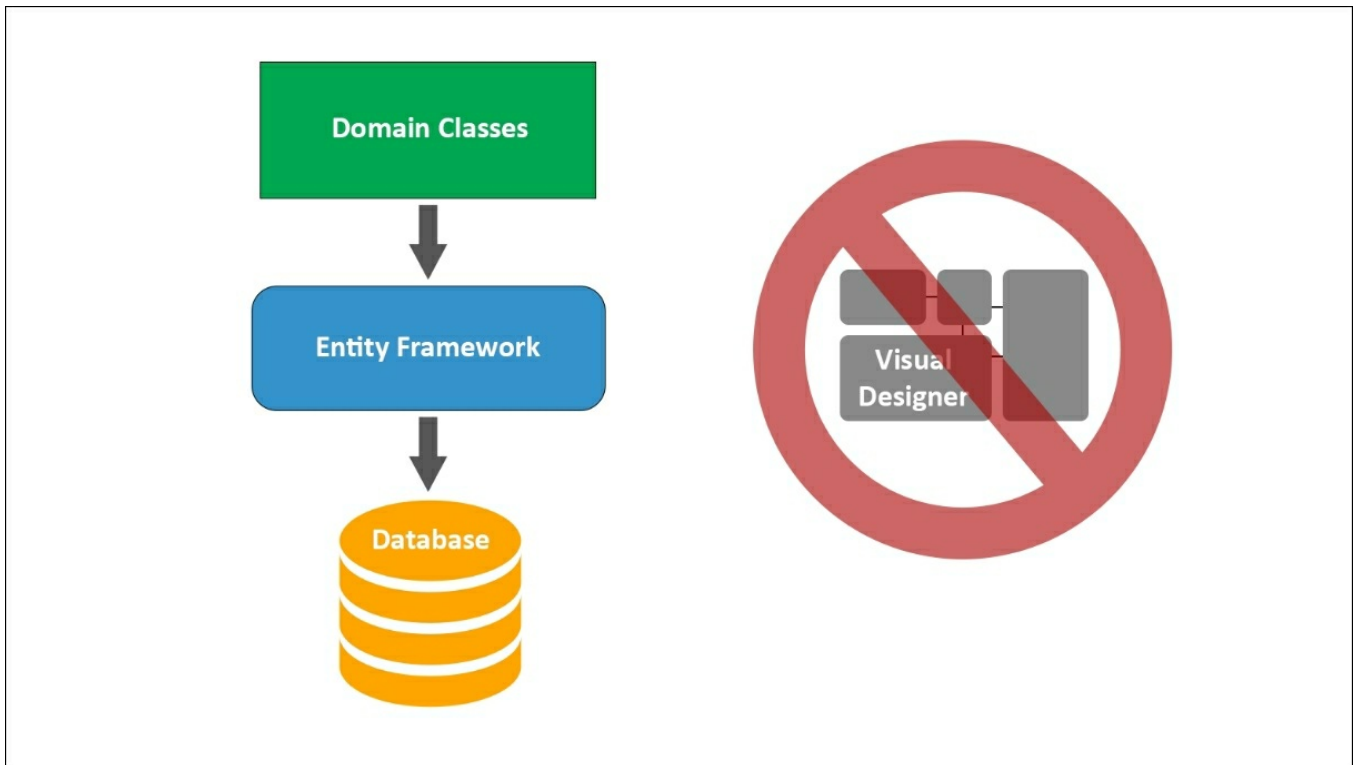
The code-first approach

Pros

- No need for diagrams and visual tools whatsoever, which could be great for small-to-medium size projects as it would save us a lot of time.
- A fluent code API that allows the developer to follow a convention over configuration approach, and to handle the most common scenarios, while also giving him the chance to switch to custom, attribute-based implementation overrides whenever he needs to customize the database mapping.

Cons

- A good knowledge of C# and updated EF conventions is required.
- Maintaining the database could be tricky sometimes, as well as handling updates without suffering data loss. The migrations support, added in 4.3 to overcome the issue which has been continuously updated since then, greatly mitigates the

problem, although it has also affected the learning curve in a negative way.

Making a choice

As we can easily see by judging the advantage and disadvantage listings, there is no such thing as an overall better or best approach. Conversely, we could say that each project scenario will likely have a most suitable approach.

Regarding our project, considering the fact we don't have a database yet and we're aiming for a flexible, mutable small-scale data structure, adopting the code-first approach would probably be a good choice. That's what we're going to do, starting from the following paragraph.

Creating entities

We're definitely going to use one of the big advantages of the Code-First approach and start writing our entity classes immediately, without worrying too much about the database engine we're going to use.

Note

Truth be told, we already know something about what we're eventually going to use. We won't be adopting a NoSQL solution, as they are not supported by EF yet; we also don't want to commit ourselves into purchasing expensive license plans, so Oracle and the commercial editions of SQL Server are most likely out of the picture as well.

This leaves us with relatively few choices: SQL Server Compact Edition, SQL Server Express, MySQL, or other less-known solutions such as PostgreSql. That being said, adopting the code-first approach will give us the chance to postpone the call until our data model is ready.

Items

Select the `OpenGameListWebApp` project from the **Solution Explorer**, then do the following tasks:

1. Create a `/Data/` folder: this will be where all our EF-related classes will reside.
2. Create a `/Data/Items/` folder.
3. Right-click on that folder and select **Add...** | **New Item**.
4. From the **Visual C#** items | **Code** | **Class**.
5. Name the new class `Item.cs` and create it.
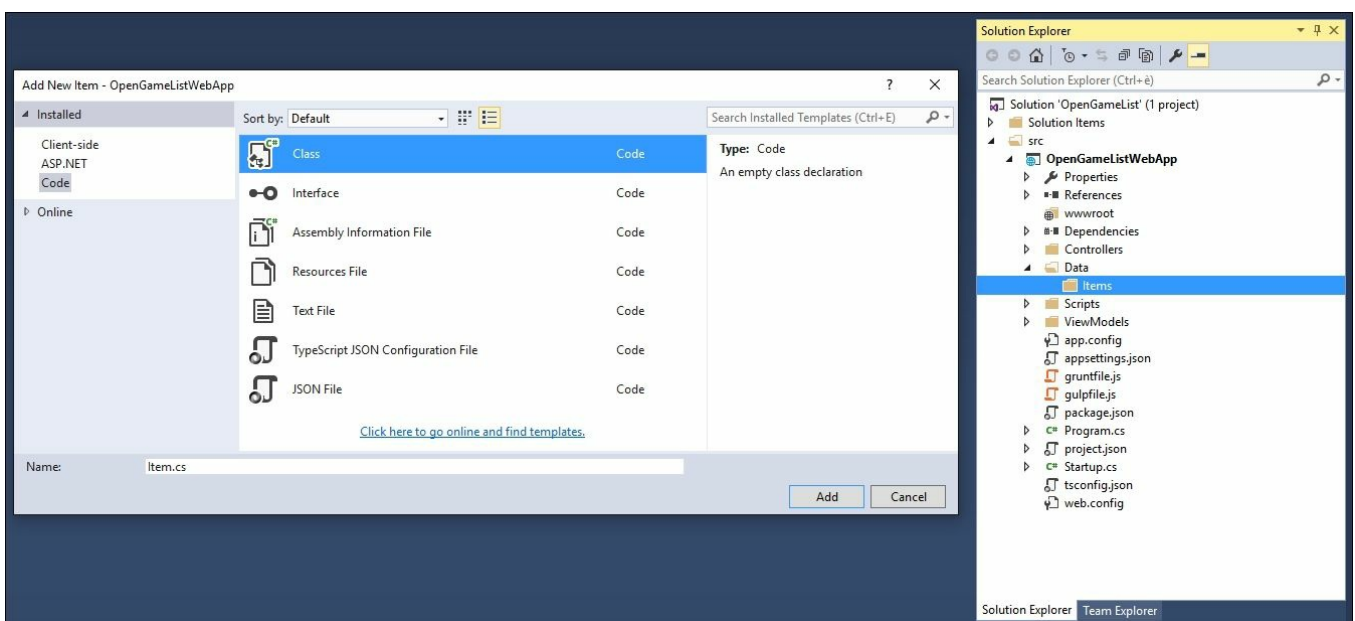
As for the source code, we could start with something like the following:

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

namespace OpenGameListWebApp.Data.Items
{
    public class Item
    {
        #region Constructor
        public Item()
        {

        }
```

```
        #endregion Constructor

        #region Properties
        [Key]
        [Required]
        public int Id { get; set; }
        [Required]
        public string Title { get; set; }
        public string Description { get; set; }
        public string Text { get; set; }
        public string Notes { get; set; }
        [Required]
        public int Type { get; set; }
        [Required]
        public int Flags { get; set; }
        [Required]
        public string UserId { get; set; }
        [Required]
        public int ViewCount { get; set; }
        [Required]
        public DateTime CreatedDate { get; set; }
        [Required]
        public DateTime LastModifiedDate { get; set; }
        #endregion Properties
    }
}
```

Note the presence of the `UserId` foreign key. We'll get to them soon.

Note

It's also worth noting that we used a lot of `Data Annotations` attributes, those being the most convenient way to override the default the code-first conventions.

If you want to know more about `Data Annotations` in the EF Core, we strongly suggest reading the official documentation at the following URL:

https://docs.efproject.net/en/latest/modeling/index.html.

As we can easily see, this entity class is very similar to the `ItemViewModel` class we created in Chapter 2, *ASP.NET Controllers and Server-Side Routes*. That's perfectly fine because that class was originally meant to resemble the public properties of the Web API underlying model, which is precisely what we're defining now.

The following diagram can help to better understand this:
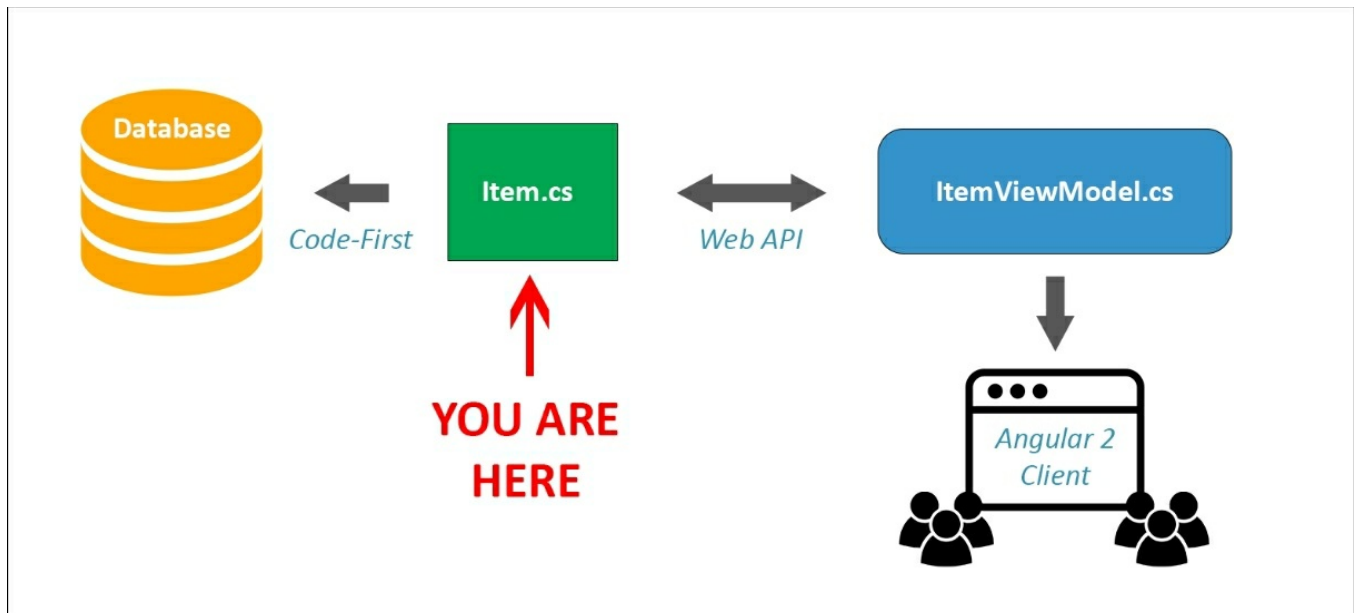
As we can see, we're creating the entity that will be used by EF to generate the database (using the code-first approach) and also translate (using property mapping) into the `ItemViewModel` we'll be using to serve our content to our Angular client.

As we might guess, the `Item` entity alone will hardly be enough. In order to complete our initial requirements we need to define a couple more entity classes:

- `Comments`, where we can store the comments related to each `Item` (if any).
- `Users`, which will serve as the main reference for all items and comments and also handle the authentication/login phase.

Comments

Let's start with the first one by doing the following:

1. Create a `/Data/Comments/` folder.
2. Right-click to that folder and select **Add...** | **New Item**.
3. From the **Visual C#** items | **Code** | **Class**.
4. Name the new class `Comment.cs` and create it.

As for the code itself, here's a good start:

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

namespace OpenGameListWebApp.Data.Comments
{
    public class Comment
    {
        #region Constructor
        public Comment()
        {

        }
        #endregion Constructor

        #region Properties
        [Key]
        [Required]
        public int Id { get; set; }
        [Required]
        public int ItemId { get; set; }
        [Required]
        public string Text { get; set; }
        [Required]
        public int Type { get; set; }
        [Required]
        public int Flags { get; set; }
        [Required]
        public string UserId { get; set; }
        public int? ParentId { get; set; }
        [Required]
        public DateTime CreatedDate { get; set; }
        [Required]
        public DateTime LastModifiedDate { get; set; }
        #endregion Properties
    }
}
```

That's it. Notice that we have three foreign keys here:

- `ItemId`, pointing to the specific item a comment is about. This is a required field, since there won't be any comment without a related item.
- `UserId`, pointing to the comment's author, which is something we'll arguably set when we'll define the `User` entity later on. This is also a required field, as we're not planning to support any comment without an author.
- `ParentId`, pointing to the master comment this entry is replying to, or null if the comment is not a reply. This is a classic implementation of the standard recursive foreign key pattern (also known as the root-leaf or parent-child), which is one of the most common ways to handle threading.

To better understand how `ParentId` will work, consider the following image:
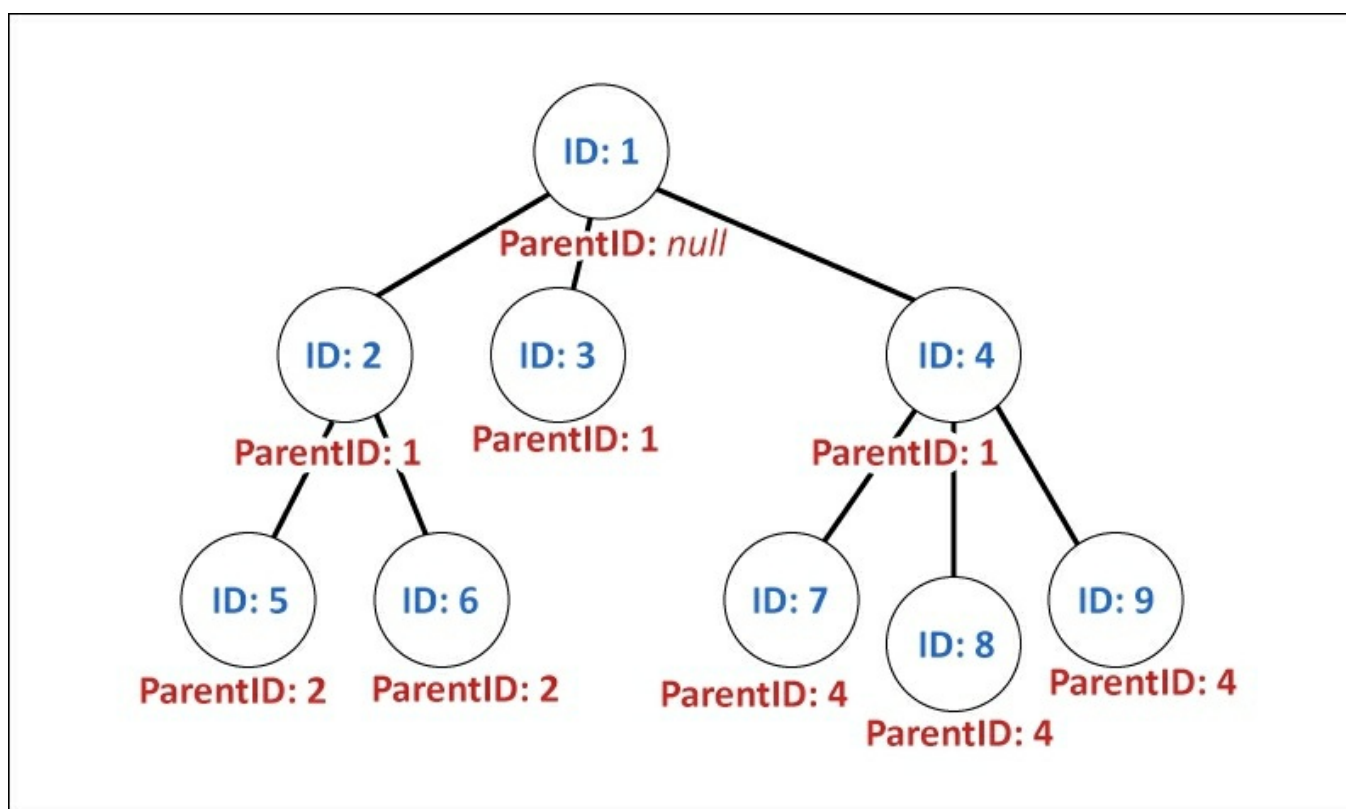
Users

It's time to add the `ApplicationUser` entity. Create a new `/Data/Users/` folder, add an `ApplicationUser.cs` class and fill it up with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OpenGameListWebApp.Data.Users
{
    public class ApplicationUser
    {
        #region Constructor
        public ApplicationUser()
        {

        }
        #endregion Constructor

        #region Properties
        [Key]
        [Required]
        public string Id { get; set; }
        [Required]
        [MaxLength(128)]
        public string UserName { get; set; }
        [Required]
        public string Email { get; set; }
        public string DisplayName { get; set; }
        public string Notes { get; set; }
        [Required]
        public int Type { get; set; }
        [Required]
        public int Flags { get; set; }
        [Required]
        public DateTime CreatedDate { get; set; }
        [Required]
        public DateTime LastModifiedDate { get; set; }
        #endregion Properties
    }
}
```

Here we go. Note that there are no foreign keys here, since the one-to-many relations with items and comments will be handled on the other side.

Note

We could ask ourselves why we used the `ApplicationUser` class name instead of `User`. The answer is pretty simple: `ApplicationUser` is the conventional name given to the class of the custom implementation of the `IdentityUser` base class used by the ASP.NET Identity module. We're using it in compliance with that convention, as we plan to implement this module later on.

Defining relationships

Now that we have built our main entity skeleton we need to create some relationships between them. We want to be able to do stuff like retrieving an `Item`, browsing to their related `Comments`, then getting the `ApplicationUser` for each one of them, and also the other way around. We can easily do that by implementing a set of related entity properties: EF will load them on demand using its default lazy-load retrieval feature.

The first thing we'll do is to add a new region to our `Item` class, containing two new properties:

```
#region Related Properties
/// <summary>
/// Current Item's Author: this property will be loaded on first
use using EF's Lazy-Loading feature.
/// </summary>
[ForeignKey("UserId")]
public virtual ApplicationUser Author { get; set; }

/// <summary>
/// Current Item's Comment list: this property will be loaded on
first use using EF's Lazy-Loading feature.
/// </summary>
public virtual List<Comment> Comments { get; set; }
#endregion Related Properties
```

To do this, we also have to add a reference to the following namespaces at the beginning of the file:

```
using OpenGameListWebApp.Data.Users;
using OpenGameListWebApp.Data.Comments;
```

Anyone who has some experience of EF won't miss the `ForeignKey` data annotation. This is one of the many code-first configuration overrides we'll need to use to have our data model properly built. There's nothing complex here, we're just telling EF that this property should be loaded using the `UserId` field. Needless to say, this will also create a one-to-many binding relationship (also known as a **constraint**), so long as our database supports it.

Tip

In order to use the `ForeignKey` attribute (and all the other EF data annotation), remember to add the `System.ComponentModel.DataAnnotations.Schema` within the `using` section of the `Item` class.

Let's do the same with the `Comments` class:

```
#region Related Properties
/// <summary>
/// Current Comment's Item: this property will be loaded on first
use using EF's Lazy-Loading feature.
/// </summary>
[ForeignKey("ItemId")]
public virtual Item Item { get; set; }

/// <summary>
/// Current Comment's Author: this property will be loaded on first
use using EF's Lazy-Loading feature.
/// </summary>
[ForeignKey("UserId")]
public virtual ApplicationUser Author { get; set; }

/// <summary>
/// Parent comment, or NULL if this is a root comment: this
property will be loaded on first use using EF's Lazy-Loading
feature.
/// </summary>
[ForeignKey("ParentId")]
public virtual Comment Parent { get; set; }

/// <summary>
/// Children comments (if present).
/// </summary>
public virtual List<Comment> Children { get; set; }
#endregion Related Properties
```

Again, remember to also add a reference to the following namespaces:

```
using OpenGameListWebApp.Data.Items;
using OpenGameListWebApp.Data.Users;
```

The `Related Properties` region is slightly more crowded here, as we have three foreign keys, but the beef is the same. For each `Comment`, we want to retrieve the related item, the user who wrote it, and also the parent comment it's replying to (if any): therefore, we define four properties and decorate them with the `ForeignKey` data annotation attribute; we also add a fourth property to keep track of the child comments (if present).

Last but not least, let's move on to the `ApplicationUser` entity class and add the following code to it:

```
#region Related Properties
/// <summary>
/// A list of items wrote by this user: this property will be
loaded on first use using EF's Lazy-Loading feature.
```

```
/// </summary>
public virtual List<Item> Items { get; set; }

/// <summary>
/// A list of comments wrote by this user: this property will be
loaded on first use using EF's Lazy-Loading feature.
/// </summary>
public virtual List<Comment> Comments { get; set; }
#endregion Related Properties
```

And the required namespace references as well:

```
using OpenGameListWebApp.Data.Items;
using OpenGameListWebApp.Data.Comments;
```

There's nothing fancy here, just a couple of lists we can use to track down all the contents posted by each user.

You will easily notice that each foreign key spawned the following:

- A single object entity property with the same type of the entity we're referring to in the class containing the foreign key
- A type-defined listing property in the related class

This pattern won't change as long as we're defining one-to-many relationships only, an object to the left, leading to a list of related objects to the right.

Are we done with our entities? Yes.

Are we ready to deploy our code-first database? Hardly. Before doing that, we need to do two things:

1. Set up an appropriate database context.
2. Enable the code-first migrations support to our project.

Let's do that right away.

Setting up DbContext

To interact with data as objects/entity classes, EF uses the
`Microsoft.EntityFrameworkCore.DbContext` class also called `DbContext` or
simply `Context`. This class is in charge of all the entity objects during execution,
including populating them with data from the database, keeping track of changes, and
persisting them to the database during CRUD operations.

We can easily create our very own `DbContext` class for our project, which we will
call `ApplicationDbContext`, by performing the following tasks:

Right-click on the `/OpenGameList/Data` folder and add a new
`ApplicationDbContext.cs` class. Fill it up with the following code:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;
using OpenGameListWebApp.Data.Items;
using OpenGameListWebApp.Data.Users;
using OpenGameListWebApp.Data.Comments;

namespace OpenGameListWebApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        #region Constructor
        public ApplicationDbContext(DbContextOptions options) :
base(options)
        {
        }
        #endregion Constructor

        #region Methods
        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

modelBuilder.Entity<ApplicationUser>().ToTable("Users");
            modelBuilder.Entity<ApplicationUser>().HasMany(u =>
u.Items).WithOne(i => i.Author);
            modelBuilder.Entity<ApplicationUser>().HasMany(u =>
u.Comments).WithOne(c => c.Author).HasPrincipalKey(u => u.Id);

            modelBuilder.Entity<Item>().ToTable("Items");
            modelBuilder.Entity<Item>().Property(i =>
i.Id).ValueGeneratedOnAdd();
            modelBuilder.Entity<Item>().HasOne(i =>
```

```
i.Author).WithMany(u => u.Items);
            modelBuilder.Entity<Item>().HasMany(i =>
i.Comments).WithOne(c => c.Item);

            modelBuilder.Entity<Comment>().ToTable("Comments");
            modelBuilder.Entity<Comment>().HasOne(c =>
c.Author).WithMany(u => u.Comments).HasForeignKey(c =>
c.UserId).OnDelete(DeleteBehavior.Restrict);
            modelBuilder.Entity<Comment>().HasOne(c =>
c.Item).WithMany(i => i.Comments);
            modelBuilder.Entity<Comment>().HasOne(c =>
c.Parent).WithMany(c => c.Children);
            modelBuilder.Entity<Comment>().HasMany(c =>
c.Children).WithOne(c => c.Parent);
        }
        #endregion Methods

        #region Properties
        public DbSet<Item> Items { get; set; }
        public DbSet<Comment> Comments { get; set; }
        public DbSet<ApplicationUser> Users { get; set; }
        #endregion Properties
    }
}
```

There are a number of things worth noting here:

- In the second constructor method's implementation, we set the `DbInitializer`, which is the class that will handle the database initialization strategy. If you're used to EF6 you know why we need to do this, otherwise don't worry as we'll get there in the following paragraph.
- We overrode the `OnModelCreating` method to manually define our data model relationships for our `ApplicationUser`, `Item`, and `Comment` entity classes. Notice that we manually configured the table names for each entity using the `modelBuilder.Entity<TEntityType>().ToTable` method. We did that with the sole purpose of showing how easy it is to customize the code-first generated database.
- Finally, we added a `DbSet` property for each of our entities, so we can easily access them later on.

Database initialization strategies

Creating the database for the first time isn't the only thing we need to worry about, for example, how can we keep tracks of the changes that will definitely occur to our data model?

In EF's previous versions, we could choose between one of the database management patterns (known as database initializers or DbInitializers) offered by the code-first approach, that is picking the appropriate database initialization strategy for our specific needs: `CreateDatabaseIfNotExists`, `DropCreateDatabaseIfModelChanges`, `DropCreateDatabaseAlways`, and `MigrateDatabaseToLatestVersion`. Additionally, if we need to address specific requirements, we could also set up our own custom initializer by extending one of the preceding options and overriding their core methods.

The major flaw of DbInitializers was that they were not that immediate and streamlined for the average developer. They were viable, yet difficult to handle without an extensive knowledge of the whole EF logic.

In EF Core the pattern has been greatly simplified: there are no DbInitializers and automatic migrations have also been removed. The database initialization aspect is now entirely handled through PowerShell commands, with the sole exception of a small set of commands that can be placed directly on the `DbContext` implementation constructor to partially automatize the process:

- `Database.EnsureCreated()`
- `Database.EnsureDeleted()`
- `Database.Migrate()`

There's currently no way to create migrations programmatically, they must be added via PowerShell, as we're going to do shortly.

Choosing the database engine

Before doing that though, we need to choose which database engine we would like to use. We're going to take this as an opportunity to demonstrate the versatility of the code-first approach.

From the main menu, select **View | SQL Server Object Explorer** and look through the list of available development-ready databases:
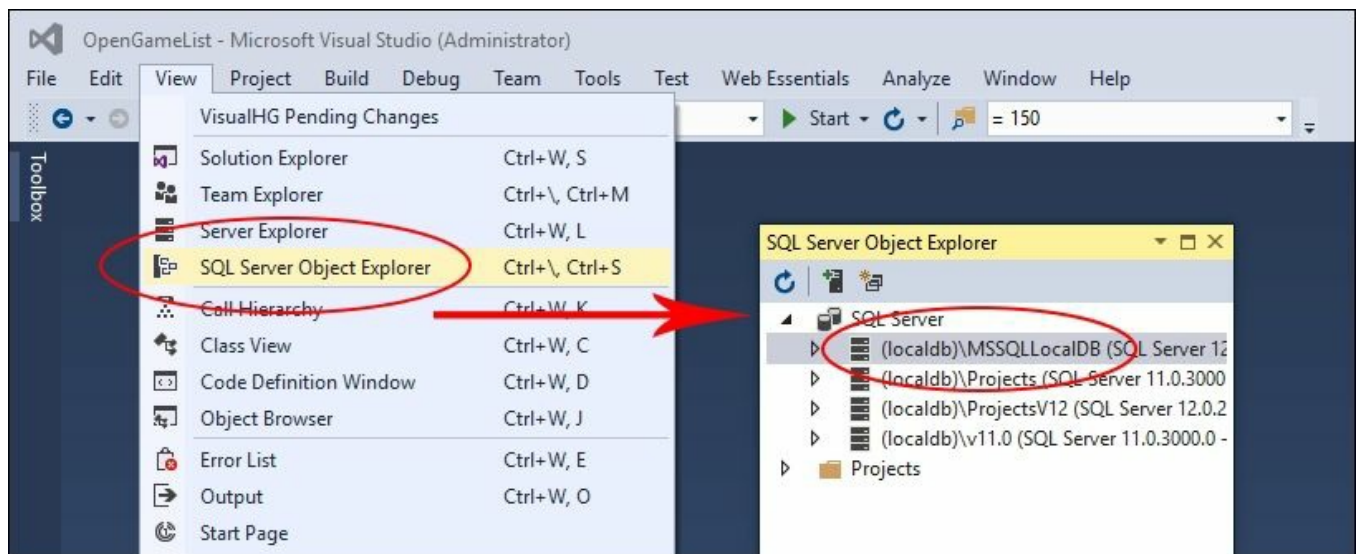
Open the SQL server node to show the `localdb` instances installed on your system. If you have one or more instances of SQL Server/SQL Express installed, you will also find a reference for each one of them.

Tip

If you have no entries there (no SQL server node), you are most likely missing the **SQL Server Data Tools** components from your Visual Studio 2015 installation. In order to fix that you need to close Visual Studio, go to **Control Panel | Programs and Features**, then select **Microsoft Visual Studio 2015** and choose **Change**: you'll be able to modify your existing installation by adding the **SQL Server Data Tools** components. Once you're done, restart Visual Studio: the default `localdb` instance should be ready and available.

For now, our pick will be the default `localdb` instance that comes with Visual Studio 2015, which goes by the name of `(localdb)\MSSQLLocalDB`. We need to keep track of that name, as we'll be using it in the `appsettings.json` file in a short while.

Note

The default `localdb` instance we just chose might be viable enough for development, but it won't work in production. Don't worry, though, we will choose a whole different database engine when we get to deployment phase. As we said before, we're doing that on purpose in order to demonstrate the versatility of the code-first approach.

Updating appsettings.json

From the **Solution Explorer**, expand the `OpenGameListWebApp` project root node, open the `appsettings.json` file, and add the following (new lines highlighted):

```json
{
  "Data": {

    "DefaultConnection": {

      "ConnectionString": "Data Source=
(localdb)\\MSSQLLocalDB;Initial Catalog=OpenGameList;Integrated
Security=True; MultipleActiveResultSets=True"

    }

  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "StaticFiles": {
    "Headers": {
      "Cache-Control": "no-cache, no-store",
      "Pragma": "no-cache",
      "Expires": "-1"
    }
  }
}
```

This is the connection string we'll be referencing to in the `Startup.cs` file later on.

Creating the database

Now that we have our own `DbContext` and a valid `Connection String`, we can easily add the initial migration and create our database.

Updating Startup.cs

The first thing we have to do is add the EF support and our `ApplicationDbContext` implementation to our application startup class. Open the `Startup.cs` file and update the `ConfigureServices` method in the following way (new lines are highlighted):

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    // Add EntityFramework's Identity support.

    services.AddEntityFramework();

    // Add ApplicationDbContext.

    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration["Data:DefaultConnection:Connecti
onString"])
        );
}
```

These new code lines will also require the following namespace references:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using OpenGameListWebApp.Data;
```

Configuring the EF tools

Before we can use the required *PowerShell* commands we need to properly configure the EF Core tools. To do that, open the `project.json` file, locate the tools section, and add the following (new lines are highlighted):

```
"tools": {
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-
preview2-final",
  "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
}
```

Right after that, it may be wise to issue a full rebuild to make sure there are no build errors in our updated code.

Adding the initial migration

Right after that, open a PowerShell command prompt and navigate through the project's root folder, which in our example is as follows:

`C:\Projects\OpenGameList\src\OpenGameListWebApp\`

Once there, type the following command to add the first migration:

`dotnet ef migrations add "Initial" -o "Data\Migrations"`

Tip

The optional `-o` parameter can be used to change the location where the migration code-generated files will be created. If we don't specify it, a root-level `/Migrations/` folder will be created and used as the default. Since we put all the EF Core classes into the `/Data/` folder, it's advisable to store migrations there as well.

Wait for the migration to be created, then type the following to execute it:

`dotnet ef database update`

Once done, open the **Server Object Explorer** and check that the `OpenGameList` database has been created, together with all their relevant tables:
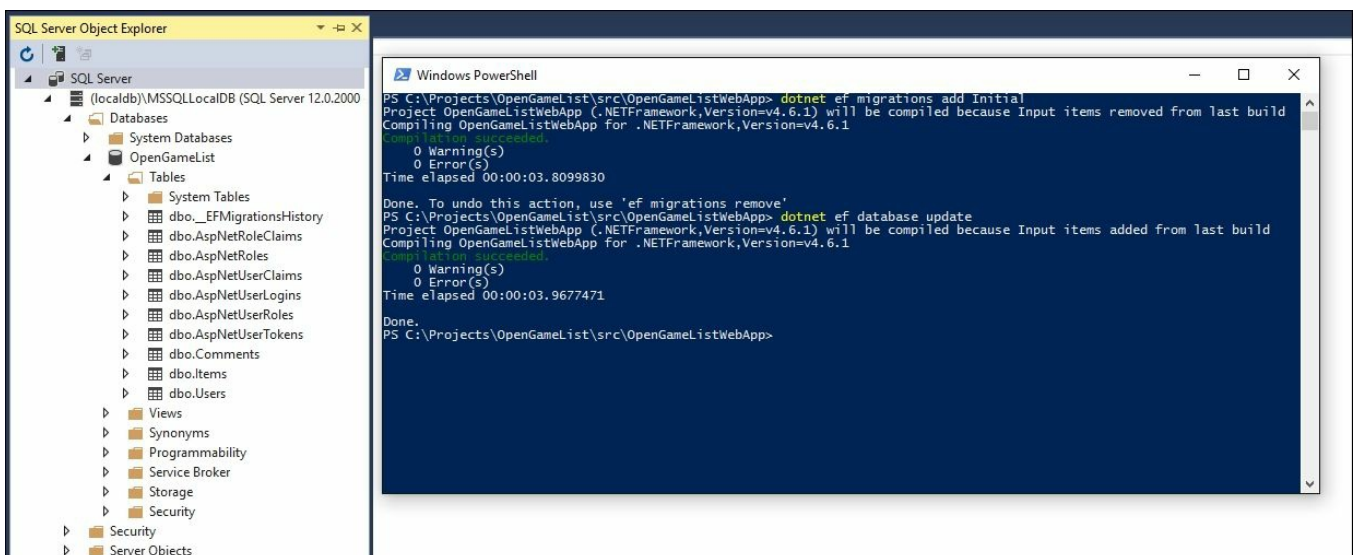
Note

Those who have used migrations before might be asking why we didn't use the Visual

Studio's Package Manager Console to execute these commands, just like it used to be done in the past. The reason is simple: Unfortunately doing this won't work because the commands need to be executed within the project root folder, which is not where the Package Manager Console commands get executed. It is also unknown if that behavior will change in the near future, even though it would definitely make sense.

If we go back to Visual Studio, we can see that there's a new `/Data/Migrations/` folder containing the EF Core code-generated files.

Understanding migrations

Before going further it may be useful to spend a short while explaining what code-first migrations actually are and the advantages we gain by using them.

Whenever we're developing an application and defining a data model, we can be sure that it will change a number of times for many good reasons: new requirements from the product owner, optimization processes, consolidation phases and so on. A bunch of properties will be added, deleted, or have their type changed. Chances are that sooner or later we'll be adding new entities as well, and/or changing their relation pattern according to our ever-changing needs.

Each time we do something like that, we'll also put our data model out of sync with its underlying, code-first generated database. This usually won't be a problem when we're debugging our app within a development environment because that scenario usually allows us to recreate the database from scratch whenever the project changes.

Upon deploying the application into production we'll be facing a whole different story. As long as we're handling real data, dropping and recreating our database won't be an option anymore, which is precisely what the code-first migrations feature is meant to address, giving the developer a chance to alter the database schema without having to drop/recreate the whole thing.

Note

We won't dig more into this topic: EF Core is a world on its own and talking too much about it would undoubtedly bring us far from the scope of this book.

If you want to go further, we can suggest starting with the official documentation hosted by the EF Core documentation website at the following URL:

https://ef.readthedocs.io/en/latest/.

Implementing a data seed strategy

We have created the database, yet it's still completely empty. In order to test it against our existing application, it could be useful to find an easy way to add some sample data programmatically.

In the most recent EF versions, up to and including EF6, it was possible to do that using the `DbMigrationsConfiguration.Seed()` method. Unfortunately though, a migrations configuration doesn't exist in EF Core: this seems to be more an implementation choice than a lack of features since the seeding tasks can now be handled directly within the `Startup.cs` file.

Note

Although this is definitely true, there is still a controversy going on between the EF Core developers community regarding that specific aspect. The absence of a high-level API and/or a consolidated pattern to run seeding after applying migrations is indeed something that should be addressed somehow, as executing such a delicate task during application running creates a number of issues, and it doesn't seem to be a viable solution in most scenarios.

Creating a DbSeeder class

Let's start with adding a `DbSeeder.cs` class in the `/Data/` folder. This class will be using the `ApplicationDbContext` to create sample data, so we'll have to find a way to have it available there without creating an additional instance. We can do that using the ASP.NET Core DI pattern in the following way:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using OpenGameListWebApp.Data;
using OpenGameListWebApp.Data.Comments;
using OpenGameListWebApp.Data.Items;
using OpenGameListWebApp.Data.Users;
using System;
using System.Linq;
using System.Threading.Tasks;

public class DbSeeder
{
    #region Private Members
    private ApplicationDbContext DbContext;
    #endregion Private Members

    #region Constructor
    public DbSeeder(ApplicationDbContext dbContext)
    {
        DbContext = dbContext;
    }
    #endregion Constructor

    #region Public Methods
    public async Task SeedAsync()
    {
        // Create the Db if it doesn't exist
        DbContext.Database.EnsureCreated();
        // Create default Users
        if (await DbContext.Users.CountAsync() == 0) CreateUsers();
        // Create default Items (if there are none) and Comments
        if (await DbContext.Items.CountAsync() == 0) CreateItems();
    }
    #endregion Public Methods

        #region Seed Methods
private void CreateUsers()
    {
        // local variables
        DateTime createdDate = new DateTime(2016, 03, 01, 12, 30, 00);
        DateTime lastModifiedDate = DateTime.Now;
```

```csharp
        // Create the "Admin" ApplicationUser account (if it
doesn't exist already)
        var user_Admin = new ApplicationUser() { Id =
Guid.NewGuid().ToString(), UserName = "Admin", Email =
"admin@opengamelist.com", CreatedDate = createdDate,
LastModifiedDate = lastModifiedDate };

        // Insert "Admin" into the Database
         DbContext.Users.Add(user_Admin);

#if DEBUG
        // Create some sample registered user accounts (if they
don't exist already)
        var user_Ryan = new ApplicationUser() { Id =
Guid.NewGuid().ToString(), UserName = "Ryan", Email =
"ryan@opengamelist.com", CreatedDate = createdDate,
LastModifiedDate = lastModifiedDate };
        var user_Solice = new ApplicationUser() { Id =
Guid.NewGuid().ToString(), UserName = "Solice", Email =
"solice@opengamelist.com", CreatedDate = createdDate,
LastModifiedDate = lastModifiedDate };
        var user_Vodan = new ApplicationUser() { Id =
Guid.NewGuid().ToString(), UserName = "Vodan", Email =
"vodan@opengamelist.com", CreatedDate = createdDate,
LastModifiedDate = lastModifiedDate };

        // Insert sample registered users into the Database
        DbContext.Users.AddRange(user_Ryan, user_Solice,
user_Vodan);
#endif
        DbContext.SaveChanges();
    }

    private void CreateItems()
    {
        // local variables
        DateTime createdDate = new DateTime(2016, 03, 01, 12, 30,
00);
        DateTime lastModifiedDate = DateTime.Now;

        var authorId = DbContext.Users.Where(u => u.UserName ==
"Admin").FirstOrDefault().Id;

#if DEBUG
        var num = 1000;  // create 1000 sample items
        for (int id = 1; id <= num; id++)
        {
            DbContext.Items.Add(GetSampleItem(id, authorId, num -
id, new DateTime(2015, 12, 31).AddDays(-num)));
        }
#endif
```

```csharp
        EntityEntry<Item> e1 = DbContext.Items.Add(new Item()
        {
            UserId = authorId,
            Title = "Magarena",
            Description = "Single-player fantasy card game similar
to Magic: The Gathering",
            Text = @"Loosely based on Magic: The Gathering, the
game lets you play against a computer opponent or another human
being.
                                    The game features a well-developed
AI, an intuitive and clear interface and an enticing level of
gameplay.",
            Notes = "This is a sample record created by the Code-
First Configuration class",
            ViewCount = 2343,
            CreatedDate = createdDate,
            LastModifiedDate = lastModifiedDate
        });

        EntityEntry<Item> e2 = DbContext.Items.Add(new Item()
        {
            UserId = authorId,
            Title = "Minetest",
            Description = "Open-Source alternative to Minecraft",
            Text = @"The Minetest gameplay is very similar to
Minecraft's: you are playing in a 3D open world, where you can
create and/or remove various types of blocks.
                            Minetest feature both single-player and
multi-player game modes.
                            It also has support for custom mods,
additional texture packs and other custom/personalization options.
                            Minetest has been released in 2015 under
GNU Lesser General Public License.",
            Notes = "This is a sample record created by the Code-
First Configuration class",
            ViewCount = 4180,
            CreatedDate = createdDate,
            LastModifiedDate = lastModifiedDate
        });

        EntityEntry<Item> e3 = DbContext.Items.Add(new Item()
        {
            UserId = authorId,
            Title = "Relic Hunters Zero",
            Description = "A free game about shooting evil space
ducks with tiny, cute guns.",
            Text = @"Relic Hunters Zero is fast, tactical and also
very smooth to play.
                            It also enables the users to look at the
source code, so they can can get creative and keep this game alive,
fun and free for years to come.
                            The game is also available on Steam.",
```

```csharp
                Notes = "This is a sample record created by the Code-
First Configuration class",
                ViewCount = 5203,
                CreatedDate = createdDate,
                LastModifiedDate = lastModifiedDate
        });

        EntityEntry<Item> e4 = DbContext.Items.Add(new Item()
        {
                UserId = authorId,
                Title = "SuperTux",
                Description = "A classic 2D jump and run, side-
scrolling game similar to the Super Mario series.",
                Text = @"The game is currently under Milestone 3. The
Milestone 2, which is currently out, features the following:
                                - a nearly completely rewritten game engine
based on OpenGL, OpenAL, SDL2, ...
                                - support for translations
                                - in-game manager for downloadable add-ons
and translations
                                - Bonus Island III, a for now unfinished
Forest Island and the development levels in Incubator Island
                                - a final boss in Icy Island
                                - new and improved soundtracks and sound
effects
                                ... and much more!
                                The game has been released under the GNU
GPL license.",
                Notes = "This is a sample record created by the Code-
First Configuration class",
                ViewCount = 9602,
                CreatedDate = createdDate,
                LastModifiedDate = lastModifiedDate
        });

        EntityEntry<Item> e5 = DbContext.Items.Add(new Item()
        {
                UserId = authorId,
                Title = "Scrabble3D",
                Description = "A 3D-based revamp to the classic
Scrabble game.",
                Text = @"Scrabble3D extends the gameplay of the classic
game Scrabble by adding a new whole third dimension.
                                Other than playing left to right or top to
bottom, you'll be able to place your tiles above or beyond other
tiles.
                                Since the game features more fields, it
also uses a larger letter set.
                                You can either play against the computer,
players from your LAN or from the Internet.
                                The game also features a set of game
servers where you can challenge players from all over the world and
```

```
get ranked into an official, ELO-based rating/ladder system.
                        ",
            Notes = "This is a sample record created by the Code-
First Configuration class",
            ViewCount = 6754,
            CreatedDate = createdDate,
            LastModifiedDate = lastModifiedDate
        });

        // Create default Comments (if there are none)
        if (DbContext.Comments.Count() == 0)
        {
            int numComments = 10;    // comments per item
            for (int i = 1; i <= numComments; i++)
DbContext.Comments.Add(GetSampleComment(i, e1.Entity.Id, authorId,
createdDate.AddDays(i)));
            for (int i = 1; i <= numComments; i++)
DbContext.Comments.Add(GetSampleComment(i, e2.Entity.Id, authorId,
createdDate.AddDays(i)));
            for (int i = 1; i <= numComments; i++)
DbContext.Comments.Add(GetSampleComment(i, e3.Entity.Id, authorId,
createdDate.AddDays(i)));
            for (int i = 1; i <= numComments; i++)
DbContext.Comments.Add(GetSampleComment(i, e4.Entity.Id, authorId,
createdDate.AddDays(i)));
            for (int i = 1; i <= numComments; i++)
DbContext.Comments.Add(GetSampleComment(i, e5.Entity.Id, authorId,
createdDate.AddDays(i)));
        }
        DbContext.SaveChanges();
    }
    #endregion Seed Methods

    #region Utility Methods
    /// <summary>
    /// Generate a sample item to populate the DB.
    /// </summary>
    /// <param name="userId">the author ID</param>
    /// <param name="id">the item ID</param>
    /// <param name="createdDate">the item CreatedDate</param>
    /// <returns></returns>
    private Item GetSampleItem(int id, string authorId, int
viewCount, DateTime createdDate)
    {
        return new Item()
        {
            UserId = authorId,
            Title = String.Format("Item {0} Title", id),
            Description = String.Format("This is a sample
description for item {0}: Lorem ipsum dolor sit amet.", id),
            Notes = "This is a sample record created by the Code-
First Configuration class",
```

```
                ViewCount = viewCount,
                CreatedDate = createdDate,
                LastModifiedDate = createdDate
            };
        }

        /// <summary>
        /// Generate a sample array of Comments (for testing purposes
only).
        /// </summary>
        /// <param name="n"></param>
        /// <param name="item"></param>
        /// <param name="authorID"></param>
        /// <returns></returns>
        private Comment GetSampleComment(int n, int itemId, string
authorId, DateTime createdDate)
        {
            return new Comment()
            {
                ItemId = itemId,
                UserId = authorId,
                ParentId = null,
                Text = String.Format("Sample comment #{0} for the item
#{1}", n, itemId),
                CreatedDate = createdDate,
                LastModifiedDate = createdDate
            };
        }
        #endregion Utility Methods
}
```

That's an impressive amount of code, yet there's nothing to worry about since it's full of repeating tasks. To properly understand what we're doing here let's split it into six parts, each one corresponding to a `#region` section we defined within the source code:

- The `Private Members`, where we define the `DbContext` object that we'll be using a lot in the methods in the following sections.
- The `Constructor` region, containing the aforementioned `DbContext` using DI.
- The `Seed` region, containing the only public method of this class: `SeedAsync`, which is in charge of the seeding task by making use of the other private methods defined in the following section.
- The `Create Entities` region, containing a couple of methods that will create the `Admin` user (plus a set of sample users) plus a grand total of 105 generic `Item` entities.
- The `Utility Methods` region, containing some internal functions used to create sample `Item` and `Comment` entities with bulk strategies.

Private members

Here we can find the reference for the `ApplicationDbContext` instance that we'll be using throughout all the classes.

Constructor

Within this region, we can find a classic class constructor implementing the standard DI pattern provided by the ASP.NET Core. An `ApplicationDbContext` will be requested of the DI container using the instantiation mode configured within the `Startup.cs` file and assigned to the `DbContext` private variable defined earlier for further reference.

Public methods

This region contains the `SeedAsync` method, the only one that can be called from outside to initiate or update the seeding task. It has been set to call the `CreateUsers` and/or `CreateItems` methods only if there are no users/items in the database already, to ensure that they won't get executed multiple times. There's not much else to say here, other than this being an `async` method to comply with the used EF Core `async` methods.

Seed methods

This region contains all the internal methods that actually insert one or more entities in our database: `CreateUsers` and `CreateItems`.

- `CreateUsers`: This method will add one or more `ApplicationUsers` to the database that will be used as authors for the `Items` and `Comments` that will be added soon after. It's worth mentioning that we used a C# pre-processor directive here (also known as conditional compilation directive), so we can use two different behaviors for our testing environment and for production. We don't want to create the sample users in our production environment, so we've put that part of code inside a conditional compilation block that gets executed only if the application is running in **Debug** mode.
- `CreateItems`: This method will add a number of `Items` to the Database that will populate the item listings of our web application. It will also conditionally create some sample `Comments` for each one of them by using the same compilation directives used within the `CreateUsers` method described previously. In our production environment, we'll have only five sample items and no comments.

Utility methods

We have two methods here: `GetSampleItem` and `GetSampleComment`. Both of them come with a self-explanatory name, as they will return a sample `Item` or `Comment`

with some minimal, parameter-dependent logic to show small differences between them.

Notice that these methods don't write anything into the database, they only return sample entities that will be persisted by the caller methods mentioned previously.

Note

Notice how we implemented our `Seed()` method to be as conservative as possible, as it will be executed each and every time our data model changes. We don't want any user, item, or comment to be added twice, not to mention repeating role assignments. In order to avoid this, we make sure that all entities are not already present in the database right before adding them.

Adding the DbSeeder to Startup.cs

Our next task will be adding the `DbSeeder` to our `Startup` class. Since we've used DI, we need to do that using DI as well, adding it to the list of available services handled by the DI container (new lines are highlighted):

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    // Add EntityFramework's Identity support.
    services.AddEntityFramework();

    // Add ApplicationDbContext.
    services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration["Data:DefaultConnection:Connecti
onString"])
        );

    // Add ApplicationDbContext's DbSeeder

    services.AddSingleton<DbSeeder>();
}
```

Notice how we used the `AddSingleton` operation mode, so that a single instance of it will be created for the whole application lifetime.

We can now use the `DbSeeder` in the `Startup.cs Configure` method, providing that we add the required parameter to have it injected there. Here's how it should be changed to reflect these changes (new/updated lines highlighted):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory, DbSeeder dbSeeder)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
the following URL:
    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.
    app.UseStaticFiles(new StaticFileOptions()
    {
```

```
        OnPrepareResponse = (context) =>
        {
            // Disable caching for all static files.
            context.Context.Response.Headers["Cache-Control"] =
Configuration["StaticFiles:Headers:Cache-Control"];
            context.Context.Response.Headers["Pragma"] =
Configuration["StaticFiles:Headers:Pragma"];
            context.Context.Response.Headers["Expires"] =
Configuration["StaticFiles:Headers:Expires"];
        }
    });

    // Add MVC to the pipeline
    app.UseMvc();

    // Seed the Database (if needed)

    try

    {

        dbSeeder.SeedAsync().Wait();

    }

    catch (AggregateException e)

    {

        throw new Exception(e.ToString());

    }
}
```

Note

We won't talk about DI any further, as it would bring us far from the scope of this book. For those who want to understand more about DI in ASP.NET we strongly suggest reading this awesome article by Steve Smith in the ASP.NET official documentation website:

http://docs.asp.net/en/latest/fundamentals/dependency-injection.html.

Handling exceptions

Notice how we wrapped the `dbSeeder.SeedAsync().Wait()` call in a `try-catch` block to handle a specific type of `Exception`: the `AggregateException`, which is commonly used to group together multiple exceptions that might simultaneously occur during the execution of parallel processes. Needless to say, these are extensively

used by EF Core.

The only real downside of `AggregateException` lies in the fact that we have to navigate through the `InnerExceptions` container property to figure out what really happened, which can be quite annoying. That's why it might be useful to catch and display them in a human-readable fashion. Luckily, the `AggregateException.ToString()` method implementation does exactly that, as it conveniently formats the inner exception messages for readability, with separators between them.
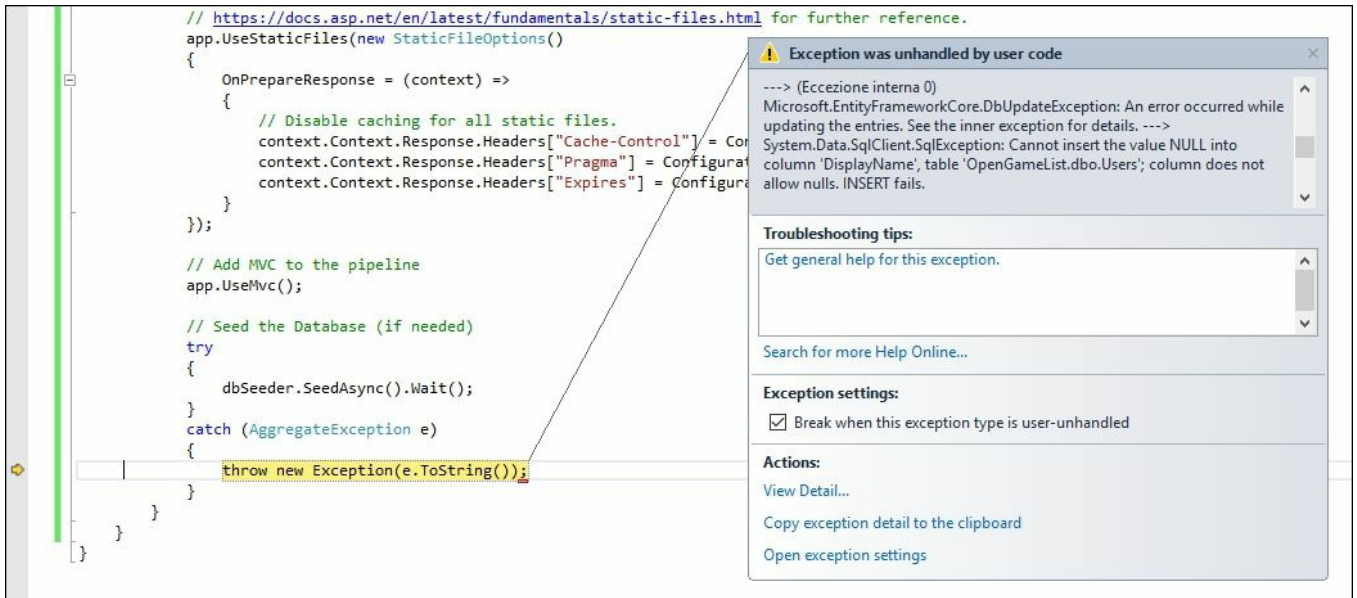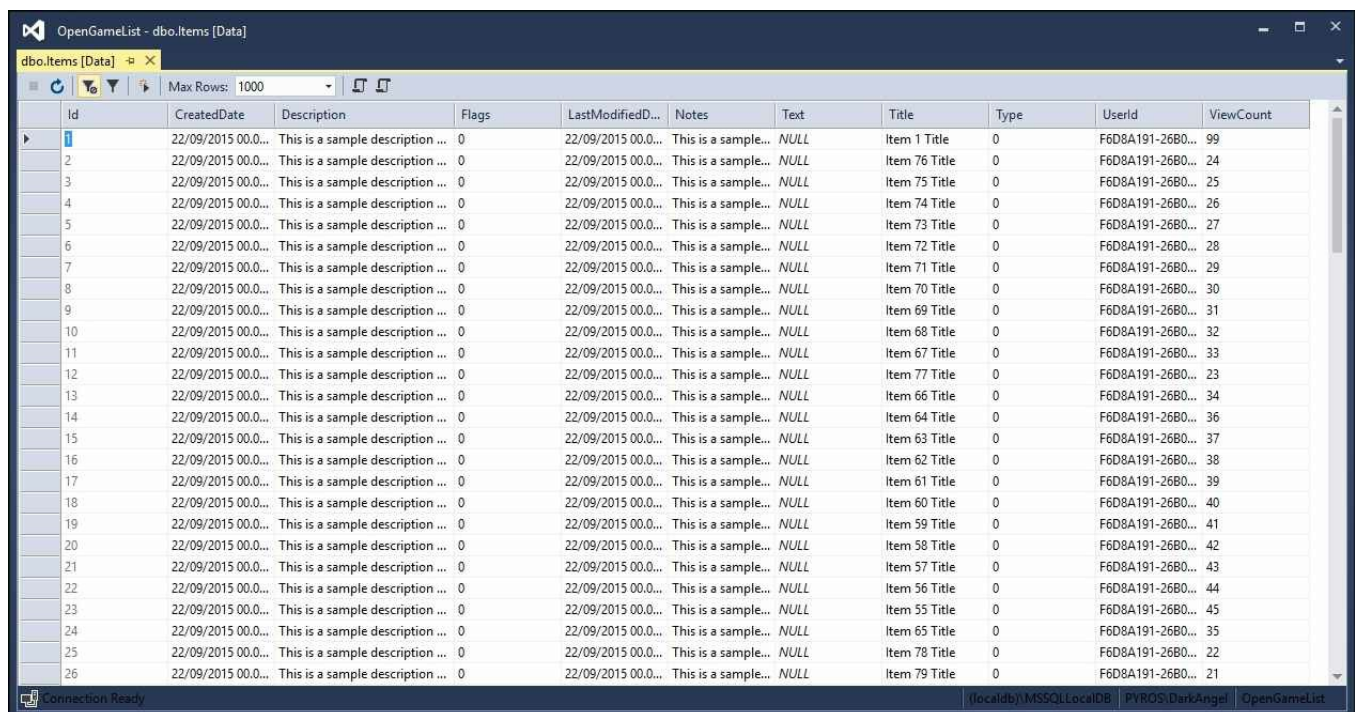
If we need further data about one or more `InnerExceptions`, we can still do that by placing a breakpoint inside the `catch` statement and analyzing them using the **Watch** panel.

Seeding the database

We're now ready to seed our database. Since we hooked the `DbSeeder.SeedAsync` to the `Startup` class, it'll be as easy as hitting **F5** and letting the application work its magic. If we have done everything correctly, our database should be populated in no time. In order to check that, we can:

1. Open the **Server Object Explorer** panel.
2. Expand the nodes up to our `OpenGameList` database.
3. Right-click on the `dbo.Items` table and select **View Data**.

Upon doing that, we should see something like the following:

Updating the ItemsController

Last but not least, we need to modify our `ItemsController` to use the `ApplicationDbContext` to retrieve data, getting rid of our dummy data provider once and for all.

In order to do that, the first thing we need to do is find an efficient way to map each **Item** entity to a corresponding `ItemViewModel` object, as our new data provider won't generate them anymore. We can achieve such a result in a number of ways, including the following:

- Adding a `Helper` method, such as `GetItemViewModel(Item item)`, handling the mapping manually with a few lines of code
- Adding a `Constructor` method, such as `ItemViewModel(Item item)`, doing pretty much the same thing as the aforementioned helper method
- Adding one of the many object-to-object auto-mapping tools freely available via NuGet and configuring it to handle the mapping automatically whenever we need it

We'll definitely go for the latter.

Installing TinyMapper

ASP.NET features a lot of object-to-object mapping tools, AutoMapper being the most used and acknowledged one. You're free to use the one you like the most and/or are most used to. For the purpose of this book we're going to use TinyMapper because it's lightweight, simple to use, and often performs better than its big brothers.

From the **Solution Explorer**, right-click on the **OpenGameListWebApp** project and select **Manage NuGet Packages**. Make sure the **Browse** tab is selected, then type `TinyMapper` into the search box and press **_Enter_** . Select the appropriate result and hit the **Install** button to add it.
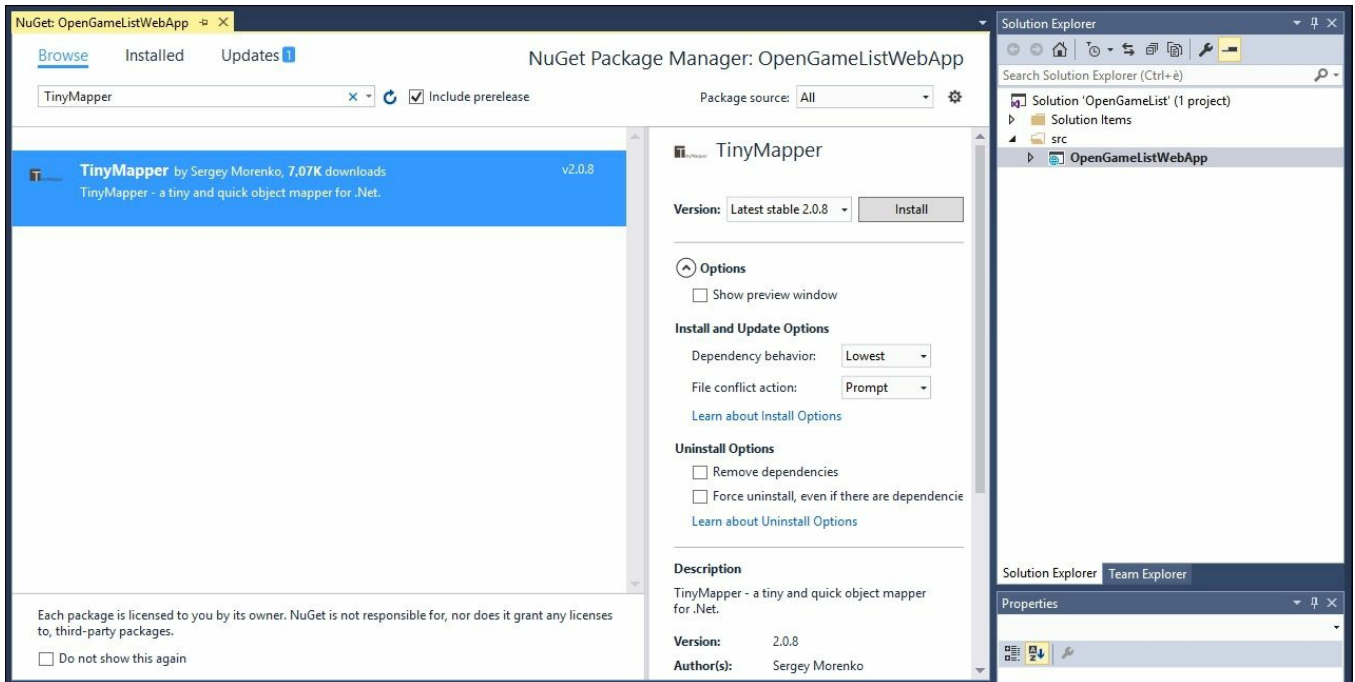
Alternatively, you can manually install it via the Package Manager Console by typing the following command:

```
Install-Package TinyMapper
```

Note

The only real downside to TinyMapper is the lack of compatibility with the .NET Core CLR, meaning that we won't be able to use it unless we target the .NET Framework runtime. That's precisely what we did back in Chapter 1, *Getting Ready*, meaning that we're good to go. However, should we ever want to target the .NET Core runtime in the future, we will definitely have to replace it.

Implementing the mapping

Like most mappers, TinyMapper does its job using two main methods:

1. `TinyMapper.Bind`, to define/configure a binding between a source type and a target type.
2. `TinyMapper.Map`, to perform an actual map from the source object to target object.

We're going to register all the bindings on application start, so we'll be able to use the `Map` method only within our Web API controllers.

Open the `Startup.cs` file and add the following lines to the `Configure` method

(new lines highlighted):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory, DbSeeder dbSeeder)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
the following URL:
    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = (context) =>
        {
            // Disable caching for all static files.
            context.Context.Response.Headers["Cache-Control"] =
Configuration["StaticFiles:Headers:Cache-Control"];
            context.Context.Response.Headers["Pragma"] =
Configuration["StaticFiles:Headers:Pragma"];
            context.Context.Response.Headers["Expires"] =
Configuration["StaticFiles:Headers:Expires"];
        }
    });

    // Add MVC to the pipeline
    app.UseMvc();

    // TinyMapper binding configuration
    TinyMapper.Bind<Item, ItemViewModel>();

    // Seed the Database (if needed)
    try
    {
        dbSeeder.SeedAsync().Wait();
    }
    catch (AggregateException e)
    {
        throw new Exception(e.ToString());
    }
}
```

On top of that, we also need to add the following references to the top of the file:

```
using Nelibur.ObjectMapper;
using OpenGameListWebApp.Data.Items;
using OpenGameListWebApp.ViewModels;
```

We're almost done. The last thing we need to do is open the `ItemsController.cs` file and do the following:

1. Add a `private` variable of the `ApplicationDbContext` type to host the `DbContext`.
2. Add a `Constructor` method to instantiate it via DI and set it to the private variable defined earlier.
3. Change the data-retrieval lines of code within the `Get`, `GetLatest`, `GetMostViewed`, and `GetRandom` methods to use the new `DbContext` instead of the old `GetSampleItems` method.
4. Update the last line of the aforementioned methods to return one or more `ItemViewModel` objects created via `TinyMapper`.
5. Remove the old `GetSampleItems` method entirely, as we don't need it anymore.

Here's the updated source code (new/modified lines are highlighted):

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Mvc;
using OpenGameListWebApp.ViewModels;
using Newtonsoft.Json;
using OpenGameListWebApp.Data;

using OpenGameListWebApp.Data.Items;

using Nelibur.ObjectMapper;

namespace OpenGameListWebApp.Controllers
{
    [Route("api/[controller]")]
    public class ItemsController : Controller
    {
        #region Private Fields

        private ApplicationDbContext DbContext;

        #endregion Private Fields

        #region Constructor

        public ItemsController(ApplicationDbContext context)

        {

            // Dependency Injetion
```

```csharp
            DbContext = context;

        }

        #endregion Constructor

        #region RESTful Conventions
        /// <summary>
        /// GET: api/items
        /// </summary>
        /// <returns>Nothing: this method will raise a NotFound
HTTP exception, since we're not supporting this API call.</returns>
        [HttpGet()]
        public IActionResult Get()
        {
            return NotFound(new { Error = "not found" });
        }
        #endregion

        #region Attribute-based Routing
        /// <summary>
        /// GET: api/items/{id}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>A Json-serialized object representing a single
item.</returns>
        [HttpGet("{id}")]
        public IActionResult Get(int id)
        {
            var item = DbContext.Items.Where(i => i.Id ==
id).FirstOrDefault();
            return new JsonResult(TinyMapper.Map<ItemViewModel>
(item), DefaultJsonSettings);
        }

        /// <summary>
        /// GET: api/items/GetLatest
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of a default number of Json-
serialized objects representing the last inserted items.</returns>
        [HttpGet("GetLatest")]
        public IActionResult GetLatest()
        {
            return GetLatest(DefaultNumberOfItems);
        }

        /// <summary>
        /// GET: api/items/GetLatest/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
```

representing the last inserted items.</returns>

```csharp
        [HttpGet("GetLatest/{n}")]
        public IActionResult GetLatest(int n)
        {
            if (n > MaxNumberOfItems) n = MaxNumberOfItems;
            var items = DbContext.Items.OrderByDescending(i =>
i.CreatedDate).Take(n).ToArray();

            return new JsonResult(ToItemViewModelList(items),
DefaultJsonSettings);
        }

        /// <summary>
        /// GET: api/items/GetMostViewed
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of a default number of Json-
serialized objects representing the items with most user views.
</returns>
        [HttpGet("GetMostViewed")]
        public IActionResult GetMostViewed()
        {
            return GetMostViewed(DefaultNumberOfItems);
        }

        /// <summary>
        /// GET: api/items/GetMostViewed/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing the items with most user views.</returns>
        [HttpGet("GetMostViewed/{n}")]
        public IActionResult GetMostViewed(int n)
        {
            if (n > MaxNumberOfItems) n = MaxNumberOfItems;
            var items = DbContext.Items.OrderByDescending(i =>
i.ViewCount).Take(n).ToArray();

            return new JsonResult(ToItemViewModelList(items),
DefaultJsonSettings);
        }

        /// <summary>
        /// GET: api/items/GetMostViewed
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of a default number of Json-
serialized objects representing some randomly-picked items.
</returns>
        [HttpGet("GetRandom")]
        public IActionResult GetRandom()
        {
```

```csharp
            return GetRandom(DefaultNumberOfItems);
        }

        /// <summary>
        /// GET: api/items/GetRandom/{n}
        /// ROUTING TYPE: attribute-based
        /// </summary>
        /// <returns>An array of {n} Json-serialized objects
representing some randomly-picked items.</returns>
        [HttpGet("GetRandom/{n}")]
        public IActionResult GetRandom(int n)
        {
            if (n > MaxNumberOfItems) n = MaxNumberOfItems;
            var items = DbContext.Items.OrderBy(i =>
Guid.NewGuid()).Take(n).ToArray();

            return new JsonResult(ToItemViewModelList(items),
DefaultJsonSettings);
        }
        #endregion

        #region Private Members
        /// <summary>

        /// Maps a collection of Item entities into a list of
ItemViewModel objects.

        /// </summary>

        /// <param name="items">An IEnumerable collection of item
entities</param>

        /// <returns>a mapped list of ItemViewModel
objects</returns>
        private List<ItemViewModel>
ToItemViewModelList(IEnumerable<Item> items)

        {

            var lst = new List<ItemViewModel>();

            foreach (var i in items)
lst.Add(TinyMapper.Map<ItemViewModel>(i));

            return lst;

        }

        /// <summary>
        /// Returns a suitable JsonSerializerSettings object that
can be used to generate the JsonResult return value for this
Controller's methods.
```

```csharp
        /// </summary>
        private JsonSerializerSettings DefaultJsonSettings
        {
            get
            {
                return new JsonSerializerSettings()
                {
                    Formatting = Formatting.Indented
                };
            }
        }

        /// <summary>
        /// Returns the default number of items to retrieve when
        /// using the parameterless overloads of the API methods retrieving
        /// item lists.
        /// </summary>
        private int DefaultNumberOfItems
        {
            get
            {
                return 5;
            }
        }

        /// <summary>
        /// Returns the maximum number of items to retrieve when
        /// using the API methods retrieving item lists.
        /// </summary>
        private int MaxNumberOfItems
        {
            get
            {
                return 100;
            }
        }
        #endregion
    }
}
```

That's it. Notice that we also have added a private `ToItemViewModelList()` utility method to centralize the *Model-to-ViewModel* mapping task, avoiding code repetition and making it easier to update.

Testing the data provider

Before going further, it's time to perform a final test to see if our `ApplicationDbContext` is working as expected.

To do that, just launch the application in **Debug** mode by hitting the *F5* key. If everything has been implemented properly you should be presented with a **Welcome View** similar to the following one:
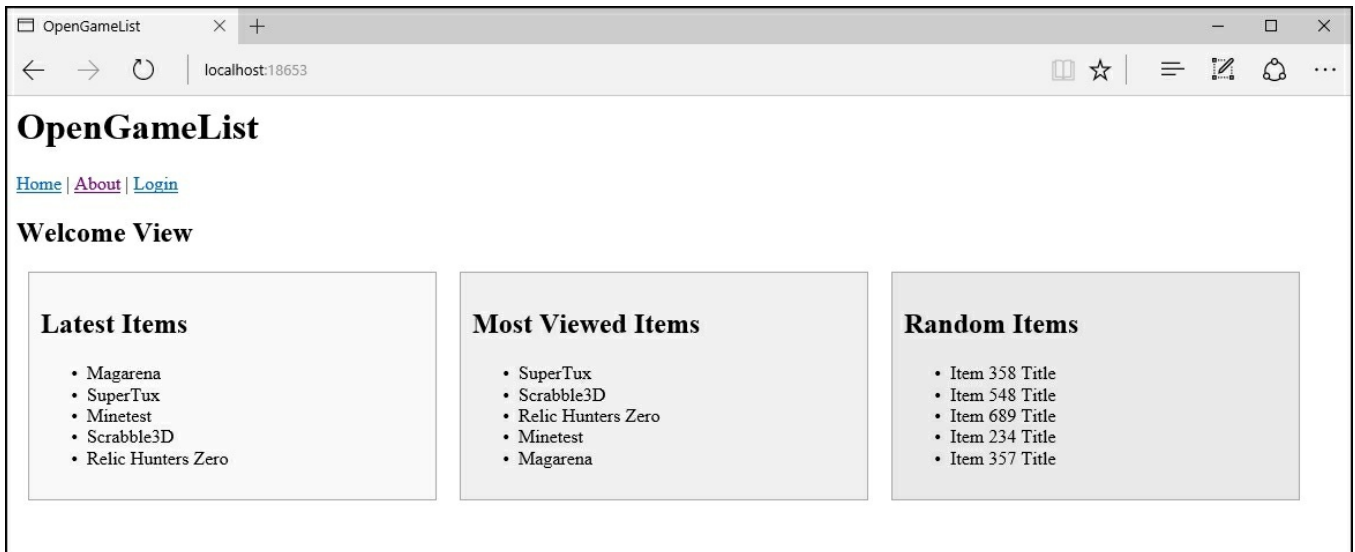
Although it doesn't seem too different from what we already had by the end of [Chapter 3](#), *Angular 2 Components and Client-Side Routing*, we know that a lot of stuff has changed under the hood. Our application is now equipped with a persistent database built on top of a real data model handled by an EF-powered, migrations-aware `DbContext` available through DI upon a per-request scope.

Suggested topics

Data model, data provider, ADO.NET, object relational mapper, EF, `Entity` class, data annotations, NOSQL, MongoDB, RavenDB, `DbContext`, `AggregateException`, CRUD operations, DI, ORM mapping.

Summary

We started this chapter enumerating a number of things we couldn't implement due to our dummy data provider limitations. In order to overcome these, we chose to replace it with a real data provider built upon a persistent database.

ED Core seemed an obvious choice to get what we wanted, so we added its relevant packages to our project. We briefly enumerated the available data modeling approaches and we resorted to using code-first due to its flexibility.

Right after that, we proceeded to create our entity classes: `Item`, `Comment`, and `ApplicationUser`, together with a set of relationships taking advantage of the renowned EF's convention over configuration approach. Then we built our `ApplicationDbContext` class accordingly.

After completing our data model we chose the database engine, quickly resorting to the Visual Studio 2015 default `LocalDb` instance. We added the connection string to the `appsettings.json` file and passed it to the `ApplicationDbContext` through the `Setup` class. Doing this allowed us to add our first migration using PowerShell and, once done, use code-first to generate our database accordingly.

We didn't want to leave our database empty, so we implemented a `DbSeeder` class to seed it with some sample data by making good use of the ASP.NET Core DI approach. Doing that took some reasonable, yet well spent, amount of time.

Finally, we switched back to the `ItemsController` class and updated it to make it use the new data provider, replacing the outdated one. In order to do this in the best possible way, we also installed `TinyMapper`, a fast and easy to use ORM mapping tool that we used inside `ItemsController` to perform model-to-view model conversions with ease.

After completing all these tasks, we ran our application in Debug mode to test that everything was still working as intended. The **Welcome View** is very similar to the one we had at the end of [Chapter 3](), *Angular 2 Components and Client-Side Routing*, yet a lot of things have changed on the inside. Our application is now ready to grow the way we want it to.

Chapter 5. Persisting Changes

In the previous chapter, we replaced our dummy data provider with a real data provider built upon EF Core using a code-first approach. Now that we have data persistence, we're ready to entrust our users with the ability to interact with our application; this means that we can implement some much needed features such as login view, access control, and server-side sessions, as well as specific views to create, edit, or delete our entities, whether they are `Items`, `Comments`, or `Users`, ensuring that all the changes will be reflected in the database.

In this chapter we will take care of the latter, leaving the authentication features out; we will address these later on, as they will have their own dedicated chapter.

Add, update, and delete items

The first thing we'll do is to implement the add, update, and delete methods for our Web API's `ItemsController`. We'll adhere to RESTful conventions and good practices, using the proper HTTP verb for each scenario: `POST` for create, `PUT` for update, and `DELETE` for delete.

Note

From now on, we won't explain how to select, open, add, rename, delete, or otherwise interact with your ASP.NET and/or Angular 2 files from **Solution Explorer**; we take for granted that you already know (or have figured out) how to navigate the GUI and what to do with the code samples.

Updating the Web API

Here are the new `ItemsController.cs`-relevant methods (new/updated lines are highlighted):

```
#region RESTful Conventions
/// <summary>
/// GET: api/items
/// </summary>
/// <returns>Nothing: this method will raise a NotFound
HTTP exception, since we're not supporting this API call.</returns>
[HttpGet()]
public IActionResult Get()
{
    return NotFound(new { Error = "not found" });
}

/// <summary>
/// GET: api/items/{id}
/// ROUTING TYPE: attribute-based
/// </summary>
/// <returns>A Json-serialized object representing a single
item.</returns>
[HttpGet("{id}")]
public IActionResult Get(int id)
{
    var item = DbContext.Items.Where(i => i.Id ==
id).FirstOrDefault();

    if (item != null) return new
JsonResult(TinyMapper.Map<ItemViewModel>(item),
DefaultJsonSettings);

    else return NotFound(new { Error = String.Format("Item
ID {0} has not been found", id) });
}

/// <summary>

/// POST: api/items

/// </summary>

/// <returns>Creates a new Item and return it accordingly.
</returns>
[HttpPost()]

public IActionResult Add([FromBody]ItemViewModel ivm)

{
```

```csharp
            if (ivm != null)

            {

                // create a new Item with the client-sent json data

                var item = TinyMapper.Map<Item>(ivm);

                // override any property that could be wise to set
from server-side only

                item.CreatedDate =

                item.LastModifiedDate = DateTime.Now;

                // TODO: replace the following with the current
user's id when authentication will be available.

                item.UserId = DbContext.Users.Where(u => u.UserName
== "Admin").FirstOrDefault().Id;

                // add the new item

                DbContext.Items.Add(item);

                // persist the changes into the Database.

                DbContext.SaveChanges();

                // return the newly-created Item to the client.

                return new JsonResult(TinyMapper.Map<ItemViewModel>
(item), DefaultJsonSettings);

            }

            // return a generic HTTP Status 500 (Not Found) if the
client payload is invalid.

            return new StatusCodeResult(500);

        }

        /// <summary>

        /// PUT: api/items/{id}

        /// </summary>

        /// <returns>Updates an existing Item and return it
accordingly.
```

```
    </returns>

    [HttpPut("{id}")]

    public IActionResult Update(int id, [FromBody]ItemViewModel
ivm)

    {

        if (ivm != null)

        {

            var item = DbContext.Items.Where(i => i.Id ==
    id).FirstOrDefault();

            if (item != null)

            {

                // handle the update (on per-property basis)

                item.UserId = ivm.UserId;

                item.Description = ivm.Description;

                item.Flags = ivm.Flags;

                item.Notes = ivm.Notes;

                item.Text = ivm.Text;

                item.Title = ivm.Title;

                item.Type = ivm.Type;

                // override any property that could be wise to
set from server-side only

                item.LastModifiedDate = DateTime.Now;

                // persist the changes into the Database.

                DbContext.SaveChanges();

                // return the updated Item to the client.

                return new
JsonResult(TinyMapper.Map<ItemViewModel>(item),
DefaultJsonSettings);

            }
```

```csharp
        }

        // return a HTTP Status 404 (Not Found) if we couldn't
find a suitable item.

        return NotFound(new { Error = String.Format("Item ID
{0} has not been found", id) });

    }

    /// <summary>

    /// DELETE: api/items/{id}

    /// </summary>

    /// <returns>Deletes an Item, returning a HTTP status 200
(ok) when done.</returns>

    [HttpDelete("{id}")]

    public IActionResult Delete(int id)

    {

        var item = DbContext.Items.Where(i => i.Id ==
id).FirstOrDefault();

        if (item != null)

        {

            // remove the item to delete from the DbContext.

            DbContext.Items.Remove(item);

            // persist the changes into the Database.

            DbContext.SaveChanges();

            // return an HTTP Status 200 (OK).

            return new OkResult();

        }

        // return a HTTP Status 404 (Not Found) if we couldn't
find a suitable item.

        return NotFound(new { Error = String.Format("Item ID
{0} has not been found", id) });
```

```
        }
#endregion
```

For the sake of simplicity, we haven't pasted the whole file; since we're following the RESTful conventions, we put all the new stuff into the region hosting these kinds of methods. Just remember to add a `using OpenGameListWebApp.Data.Items` line at the beginning of the file.

The code contains some comments that will help to focus on what we just did. Nonetheless, it would be useful to highlights some specific aspects of the new methods, such as the following:

- We didn't have to use the `Json.NET` library at all here, as the ASP.NET framework can automatically handle the conversion task between a JSON-type input and a `JsonSerializable` object; all we did was specify an object of type `ItemViewModel` as the main input parameter of the `Add` and `Update` methods and tell the framework to retrieve it from the request body.
- We used the `TinyMapper` mapping library at the end of the `Add` and `Update` methods to return a new `ItemViewModel` to the client built upon the created/modified `Item`. Notice how we didn't use it the other way around, that is, to populate the `Item` properties from the `ItemViewModel` instance built upon the JSON data sent by the client in the `Update` method, as we preferred to manually treat and check each of the properties separately.

Note

We should also spend a few moments talking about *what we didn't do* here: no error-handling strategies, no specific controls on user input, and no authentication, just to name a few. This isn't a robust, production-ready code yet, and we need to be fully aware of that. There's nothing wrong with it; we're still in development phase, after all, and we'll refine these aspects once we get a good grip on the Web API features we need to know.

Adapting the client

Now that our Web API supports the four basic CRUD functions, we can modify our Angular 2 client to make use of them.

Improving the ItemService

Let's start with updating the code of our `ItemService` class. Open the `/Scripts/app/item.service.ts` file and change its contents with the following code:

```
import {Injectable} from "@angular2core";
import {Http, Response, Headers, RequestOptions} from
"@angular/http";
import {Observable} from "rxjs/Observable";
import {Item} from "./item";

@Injectable()
export class ItemService {
    // private Data: { Latest: Item[], MostViewed: Item[], Random:
Item[] };
    private baseUrl = 'api/items/';   // URL to web api

    constructor(private http: Http) { }

    // calls the [GET] /api/items/GetLatest/{n} Web API method to
retrieve the latest items.
    getLatest(num?: number) {
        var url = this.baseUrl + "GetLatest/";
        if (num != null) url += num;
        return this.http.get(url)
            .map(response => response.json())
            .catch(this.handleError);
    }

    // calls the [GET] /api/items/GetMostViewed/{n} Web API method
to retrieve the most viewed items.
    getMostViewed(num?: number) {
        var url = this.baseUrl + "GetMostViewed/";
        if (num != null) url += num;
        return this.http.get(url)
            .map(response => response.json())
            .catch(this.handleError);
    }

    // calls the [GET] /api/items/GetRandom/{n} Web API method to
retrieve n random items.
    getRandom(num?: number) {
        var url = this.baseUrl + "GetRandom/";
        if (num != null) url += num;
```

```
        return this.http.get(url)
            .map(response => response.json())
            .catch(this.handleError);
    }

    // calls the [GET] /api/items/{id} Web API method to retrieve
the item with the given id.
    get(id: number) {
        if (id == null) throw new Error("id is required.");
        var url = this.baseUrl + id;
        return this.http.get(url)
            .map(res => <Item>res.json())
            .catch(this.handleError);
    }

    // calls the [POST] /api/items/ Web API method to add a new
item.


    add(item: Item) {

        var url = this.baseUrl;

        return this.http.post(url, JSON.stringify(item),
        this.getRequestOptions())

            .map(response => response.json())

            .catch(this.handleError);

    }

    // calls the [PUT] /api/items/{id} Web API method to update an
existing item.

    update(item: Item) {

        var url = this.baseUrl + item.Id;

        return this.http.put(url, JSON.stringify(item),
this.getRequestOptions())

            .map(response => response.json())

            .catch(this.handleError);

    }

    // calls the [DELETE] /api/items/{id} Web API method to delete
the item with the given id.

    delete(id: number) {
```

```
        var url = this.baseUrl + id;

        return this.http.delete(url)

            .catch(this.handleError);

    }

    // returns a viable RequestOptions object to handle Json
requests

    private getRequestOptions() {

        return new RequestOptions({

            headers: new Headers({

                "Content-Type": "application/json"

            })

        });

    }

    private handleError(error: Response) {
        // output errors to the console.
        console.error(error);
        return Observable.throw(error.json().error || 'Server
error');
    }
}
```

Let's see what we did here:

- In line 2, we added a reference for the `Response` and `RequestOptions` classes from the Angular 2 built-in `http` service.
- We then implemented three public methods (`add`, `update`, and `delete`) to handle the corresponding Web API calls we added earlier.
- Since each of our new methods requires a `RequestOptions` object, we eventually created a `getRequestOptions` method to avoid repeating the same lines of codes multiple times. Since `add` and `update` are both sending JSON data, having a centralized factory method returning a `RequestOptions` object is definitely a good choice.

Updating the GUI

Now that we have these `insert`, `update`, and `delete` methods, we need to make the

GUI aware of that by adding the relevant commands. The former should be put on our main menu, while the other two will be added to the item's detail view. The reason for that is simple: we want our users to be able to create a new item from any view, or `update`/`delete` only the specific item they clicked on.

Add new

Let's start with the `insert` command. Open the `\Scripts\app\app.component.ts` file and add the following element to our `@Component`'s `template` (new/updated lines are highlighted):

```
@Component({
    selector: "opengamelist",
    template: `
    <h1>{{title}}</h1>
      <div class="menu">
        <a class="home" [routerLink]="['']">Home</a>
        | <a class="about" [routerLink]="['about']">About</a>
        | <a class="login" [routerLink]="['login']">Login</a>
        | <a class="add" [routerLink]="['item', 0]">Add New</a>
    </div>
    <router-outlet></router-outlet>
    `
})
```

Notice that we're deliberately exploiting the `item` route we already have in place, passing an `id` with a value of 0. This means that any item with an ID of 0, as per our internal convention, should be treated by our `ItemDetail` component as a new, not-yet-existing item.

Needless to say, in order to do that, we need to update our `item-detail.component.ts` file accordingly (new/modified lines are highlighted):

```
import {Component, OnInit} from "@angular/core";
import {Router, ActivatedRoute} from "@angular/router";
import {Item} from "./item";
import {ItemService} from "./item.service";

@Component({
    selector: "item-detail",
    template: `
        <div *ngIf="item" class="item-details">
          <h2>{{item.Title}} - Detail View</h2>
          <ul>
              <li>
                  <label>Title:</label>
                  <input [(ngModel)]="item.Title"
placeholder="Insert the title..."/>
              </li>
```

```
            <li>
                <label>Description:</label>
                <textarea [(ngModel)]="item.Description"
placeholder="Insert a suitable description..."></textarea>
            </li>
        </ul>
        <div *ngIf="item.Id == 0" class="commands insert">

            <input type="button" value="Save"
(click)="onInsert(item)" />

            <input type="button" value="Cancel"
(click)="onBack()" />

        </div>
    </div>
    `,
    styles: [`
        .item-details {
            margin: 5px;
            padding: 5px 10px;
            border: 1px solid black;
            background-color: #dddddd;
            width: 300px;
        }
        .item-details * {
            vertical-align: middle;
        }
        .item-details ul li {
            padding: 5px 0;
        }
    `]
})

export class ItemDetailComponent {
    item: Item;

    constructor(private itemService: ItemService,
        private router: Router,
        private activatedRoute: ActivatedRoute) {
    }

    ngOnInit() {
        var id = +this.activatedRoute.snapshot.params["id"];
        if (id) {
            this.itemService.get(id).subscribe(
                item => this.item = item
            );
        }
        else if (id === 0) {

            console.log("id is 0: adding a new item...");
```

```
            this.item = new Item(0, "New Item", null);

        }
        else {
            console.log("Invalid id: routing back to home...");
            this.router.navigate([""]);
        }
    }

    onInsert(item: Item) {

        this.itemService.add(item).subscribe(

            (data) => {

                this.item = data;

                console.log("Item " + this.item.Id + " has been
added.");

                this.router.navigate([""]);

            },

            (error) => console.log(error)

        );

    }

    onBack() {

        this.router.navigate([""]);

    }
}
```

Let's see what we did here:

- In the component's `template` section, we added a `<div>` element with an `*ngIf` condition bound to the `item.Id` being 0 so that it will only be added into the DOM if the user came here with the route designed for adding a new item.
- Within the aforementioned `<div>` element, we added a couple of `<input type="button">` fields, namely `Save` and `Cancel`, which we bound to the new `onInsert` and `onBack` methods (see later).
- We modified the `ngOnInit` method's behavior to support our internal convention that if the `id` is 0, it means that we're adding a new item.
- We implemented a new `onInsert` method, which will call the `add` method of

our `ItemService` class we created earlier for adding the item through our Web API; we also added a subscription that will either return the user to the **Home** route (the **Welcome View**) in the case of success, or output an error to the console in the case of failure.

- We implemented a new `onBack` method, which will just trigger the **Home** route, returning the user back to the **Welcome View** without doing anything.

That's it for the `Add New` command; do not close the `item-detail.component.ts` file, though, as we'll need to update it further in order to implement the next ones.

Update and Delete

As we said earlier, the `Update` and `Delete` commands will take place within the `ItemDetail` component itself. This means that we'll need to keep working with the `item-detail.component.ts` file, adding a couple more buttons to the template and binding them to two new methods; that's basically the same thing we did in the previous section with the `Save`/`onInsert` and `Cancel`/`onBack` buttons/methods.

Here is the updated template (new/modified lines are highlighted):

```
template: `
    <div *ngIf="item" class="item-details">
      <h2>{{item.Title}} - Detail View</h2>
      <ul>
          <li>
              <label>Title:</label>
              <input [(ngModel)]="item.Title"
placeholder="Insert the title..."/>
          </li>
          <li>
              <label>Description:</label>
              <textarea [(ngModel)]="item.Description"
placeholder="Insert a suitable description..."></textarea>
          </li>
      </ul>
      <div *ngIf="item.Id == 0" class="commands insert">
          <input type="button" value="Save"
(click)="onInsert(item)" />
          <input type="button" value="Cancel"
(click)="onBack()" />
      </div>
      <div *ngIf="item.Id != 0" class="commands update">

          <input type="button" value="Update"
(click)="onUpdate(item)" />

          <input type="button" value="Delete"
(click)="onDelete(item)" />
```

```
                <input type="button" value="Back" (click)="onBack()"
/>

        </div>
    </div>
   `,
```

And these are the new methods, to be placed just after the `onInsert` one:

```
onUpdate(item: Item) {
    this.itemService.update(item).subscribe(
        (data) => {
            this.item = data;
            console.log("Item " + this.item.Id + " has been
updated.");
            this.router.navigate([""]);
        },
        (error) => console.log(error)
    );
}

onDelete(item: Item) {
    var id = item.Id;
    this.itemService.delete(id).subscribe(
        (data) => {
            console.log("Item " + id + " has been deleted.");
            this.router.navigate([""]);
        },
        (error) => console.log(error)
    );
}
```

What we did here is quite simple:

- We added another `<div>` element with an `*ngIf` condition that will handle the opposite case as before: `item.Id` is not 0, which means that we're dealing with an already existing item.
- The new `<div>` has three `<input type="button">` fields: `Update`, `Delete`, and `Back`, bound respectively to the `onUpdate`, `onDelete`, and `onBack` methods (see later).
- We implemented the new `onUpdate` and `onDelete` methods, which will call the `update` and `delete` method of our `ItemService` class, respectively; these will update or delete the currently selected item through our Web API. Again, we also added a subscription that will either return the user to the **Welcome View** or output an error to the console log.
- We didn't implement a new method for the **Back** button because we already have the `onBack` method, which does what we need, returning the user back to the **Welcome View** without doing anything.

With these additions, our GUI should be ready; before going further, let's perform a full surface test.

Note

While we don't want to talk too much about Angular 2 and its `http` class, we should definitely spend a few words talking about its new abstraction pattern based upon observables.

One of the most relevant differences with the previous approach is that observables have a lazy behavior by design, meaning that they won't fire unless there is a valid subscription issued by a `.subscribe()` function call. This is a major perspective switch from the AngularJS Promises, which would execute right off the bat regardless of how the client code will use their result afterward.

Another important difference involves the `.subscribe()` function, which will be fired upon completion of the ItemService's `add` task. In Angular 2, subscriptions are designed to work just like a standard `.then()` or `.complete()` function featured in most async-based JavaScript libraries (AngularJS/Promises, JQuery/AJAX, and more), with the key difference that they are also bound to the `Observable` itself; this means that they won't just trigger once and resolve, but they will be executed each and every time the `Observable` completes its task(s) until it ceases to exist, unless they get cancelled by calling the `.unsubscribe()` function method on their `Observable`.

That said, we can easily notice that these HTTP requests are hardly a good example to demonstrate the advantages of this behavior, as these are observables that will fire only once; we'll see it better when we implement some reactive functionalities such as search with autocomplete and similar stuff.

Testing it out

Before hitting *F5* to launch the application in **Debug** mode, ensure that the `Gulp` (or `Grunt`) `default` task is properly running in the **Task Runner Explorer**, otherwise, the compiled scripts in the `www` folder won't be updated and you won't see the new stuff.

This is how our **Welcome View** should appear now:
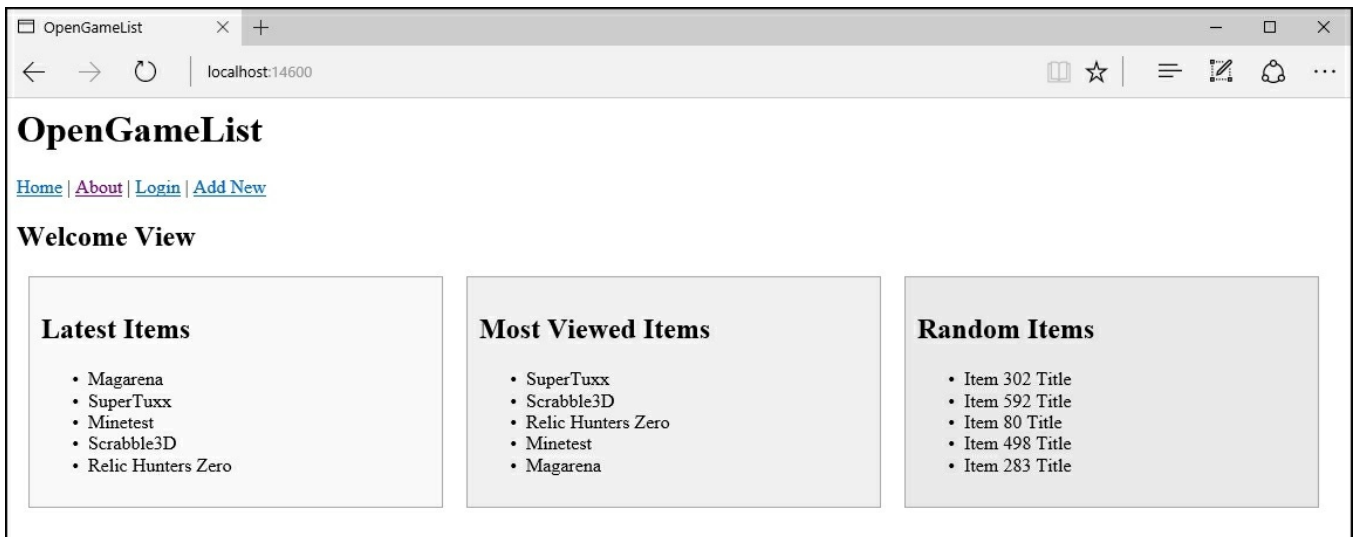
Notice the **Add New** menu command; by clicking on that, we will be brought to the updated **Item - Detail View**:
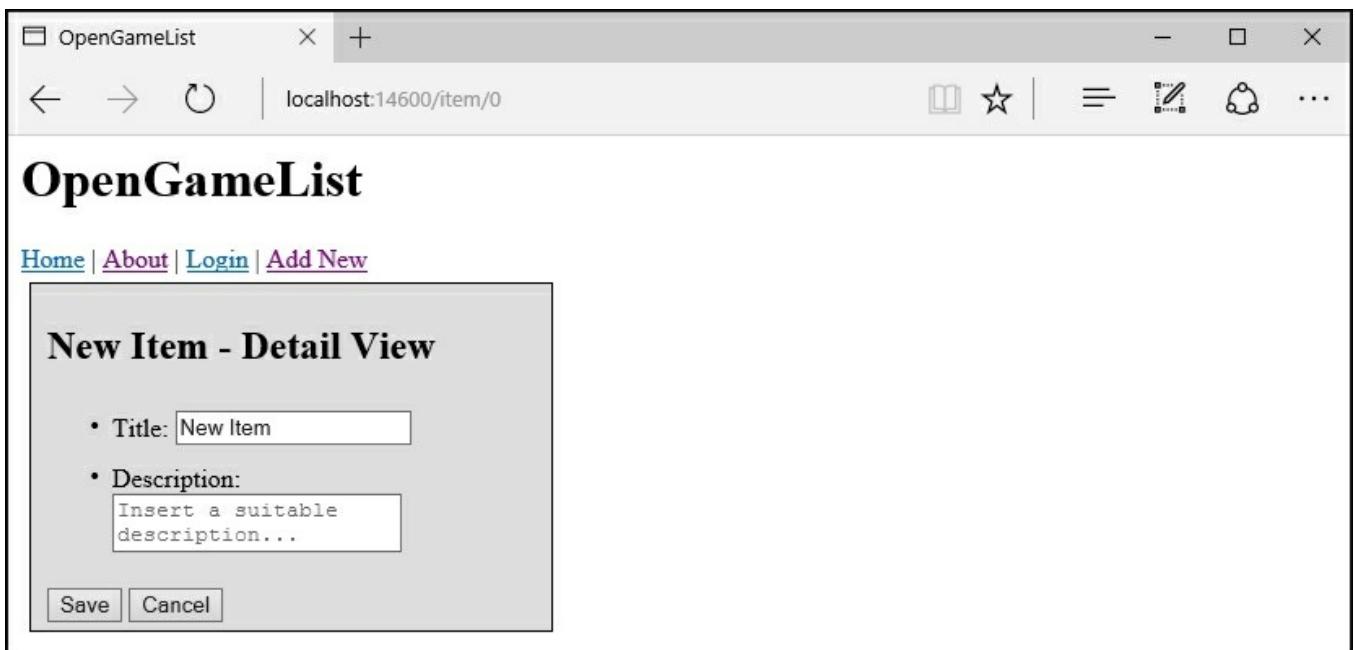
As we can see, the `ItemDetail` component is now aware of the fact that we're in the process of adding a new item; we can tell that by looking at the **Save** and **Cancel** buttons, which we know are only available when we're dealing with a new item, and also by examining the routed URL: `/item/0` matches our internal convention that an item with an `id` of 0 is a new, not-yet-existing item.

So far, so good; let's now click on the **Cancel** button. That should bring us back to the

Welcome View. From there, click on one of the items contained in the **Last Inserted** listing panel, for example, **Magarena**. The **Item - Detail View** should be displayed again, this time in **Update** mode:
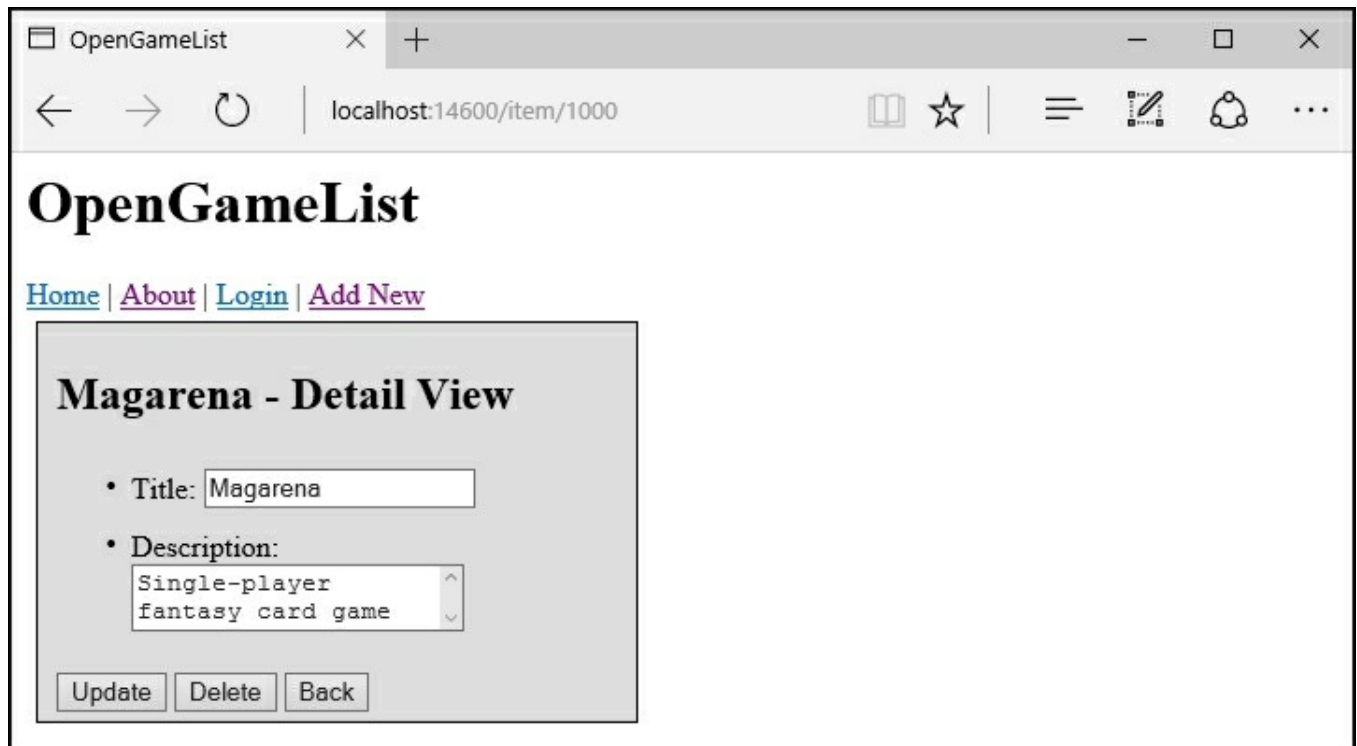
Again, we can confirm that by looking at the button set, **Update**, **Delete**, and **Back** are precisely what we're expecting to see when updating an item.

We're not done yet; now that we have tested the GUI, we need to see whether the `insert`, `update`, and `delete` methods are actually working. Ideally, the changes should be immediately seen by the user, meaning that:

- Whenever we add a new item, we want it to be shown in the **Welcome View** as the first entry of the **Last Inserted Items** panel as soon as we are sent back there
- Whenever we modify an item title, we want to see it updated in the **Welcome View** item listings (if present) as soon as we are sent back there
- Whenever we delete an item, we want it to disappear from the **Welcome View** as soon as we are sent back there

Adding a new test

From the **Welcome View**, click on the **Add New** menu item again to load the **Item - Detail View** in insert mode. Write `Sample Item` in the **Title** textbox and `Sample Description` in the **Description** text area; notice that, thanks to the Angular 2 two-

way binding, the title shown within the view will be updated in *real time* as soon as we start typing, following our changes:
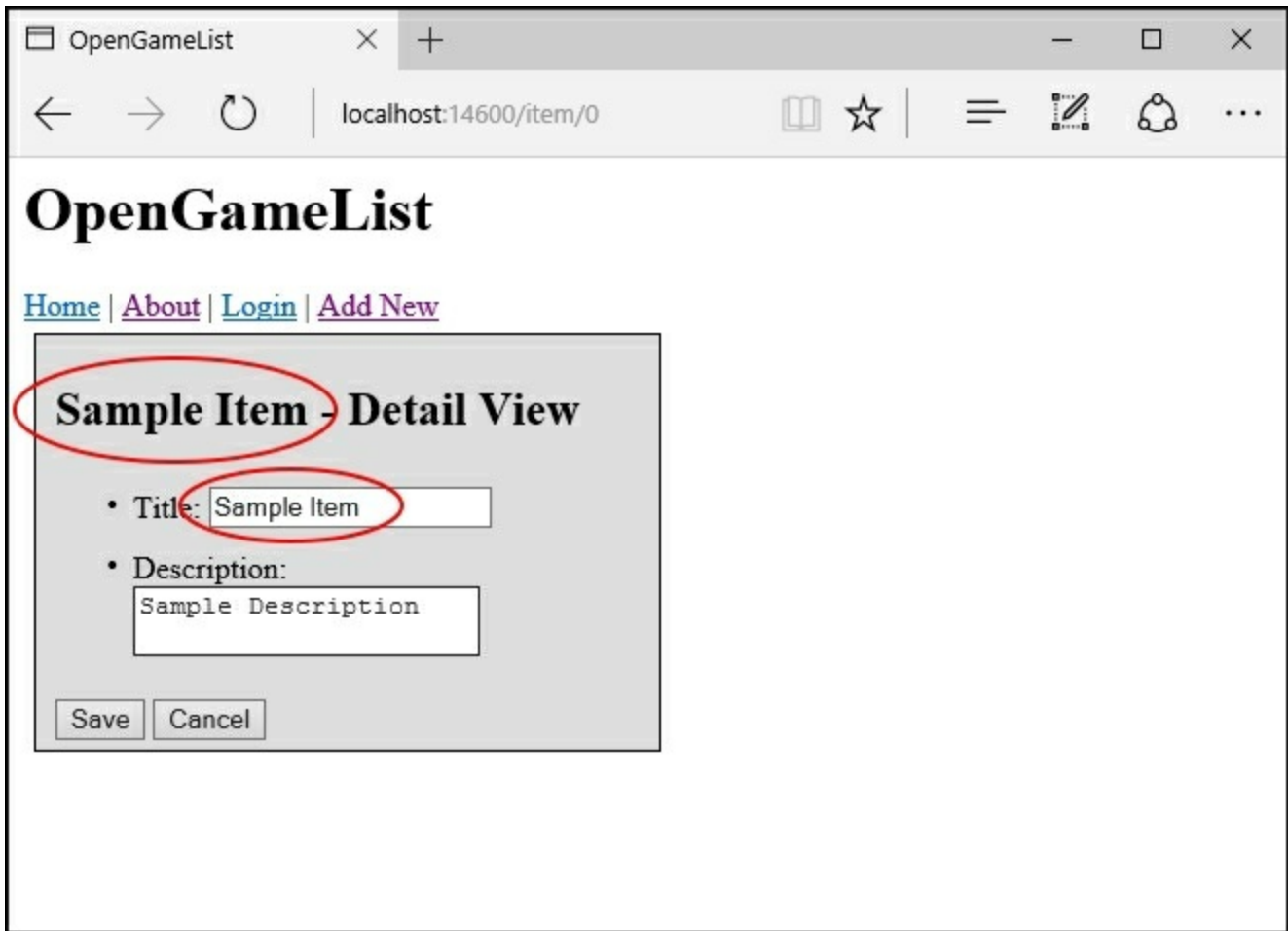
Now press the **Save** button to see whether the changes will be reflected in the **Welcome View**; if we did everything correctly, there's no reason why they shouldn't. We can also get a summary of the tasks performed by our application by looking at the developer **Console** log. In order to show it, press *F12* if we're using Microsoft Edge, or *CTRL + ALT + J* in case we're using Google Chrome or Mozilla Firefox:
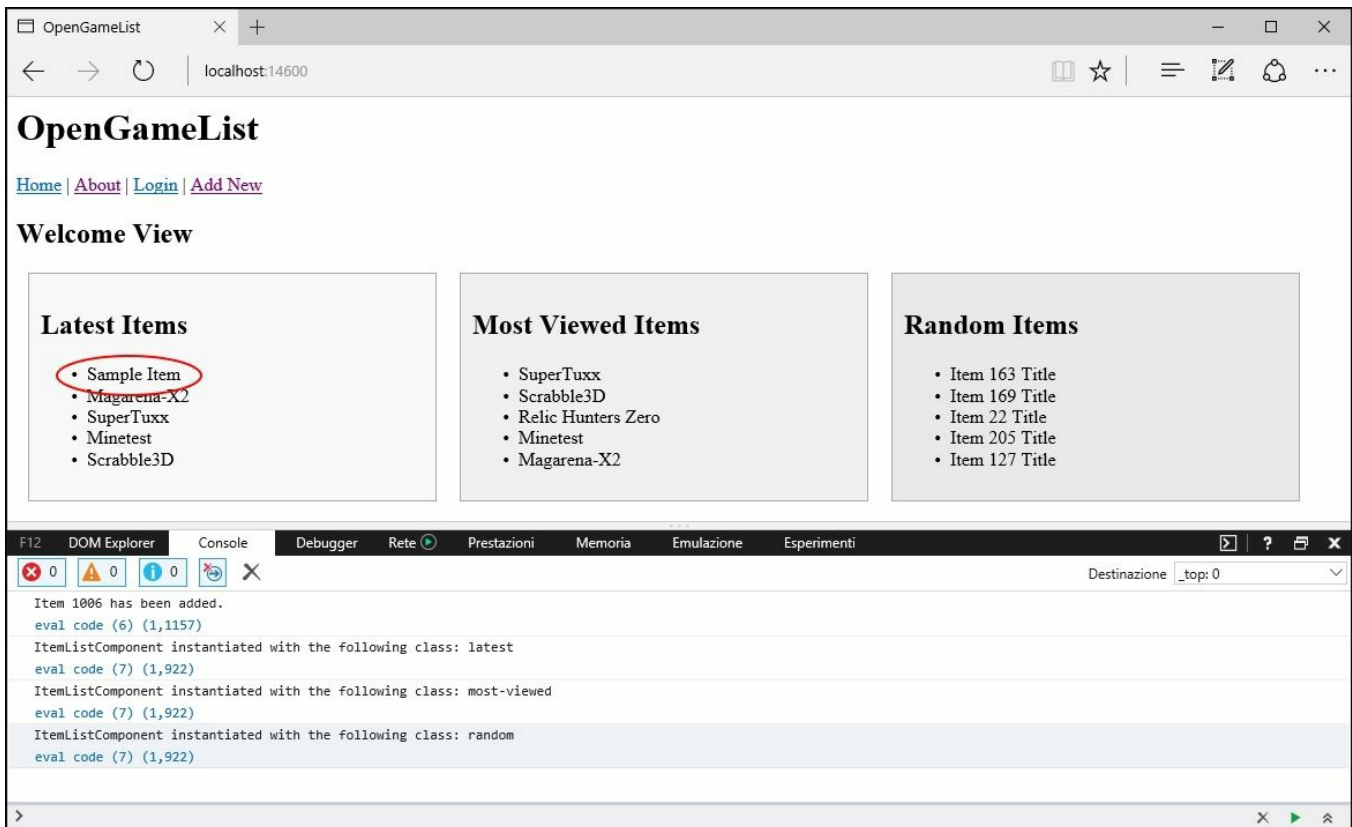
Everything is working as intended. Let's do a quick recap of what is happening under the hood:

- When the user clicks on the **Add New** menu item command, the application will show the **Item - Detail View** with an `id` of 0, which is our internal convention to trigger the insert mode.
- When the user fills up the **Title** and **Description** fields and confirms the insert operation by clicking on the **Save** button, the application will fire the `onInsert` function, which will use the `ItemService` to return an `Observable` object ready to issue an HTTP call to persist these changes to the database using the Web API `Add` method.
- The `Observable` object has a lazy behavior by design, meaning that it won't fire without a subscription. The `onInsert` method handles it fluently with a `.subscribe()` call which will trigger the `Observable` job and also set up a series of tasks to perform upon its completion:
  - In the case of success, update the local item object, output a message in the **Console** log, and re-route the user to the **Welcome View**
  - In the case of failure, output the error in the **Console** log
  - As soon as the user is sent back to the **Welcome View**, all the `ItemListComponents` will get re-instantiated again, so they will always load and display an updated set of lists containing the new item.

Update test

Let's try to modify the title of the new item and see whether our application behaves just as expected.

From the **Welcome View**, click on the **Sample Item** we just created; the **Item - Detail View** should be displayed in update mode, allowing us to change the item's field. Again thanks to the two-way binding, the title will be updated in real time as we type:
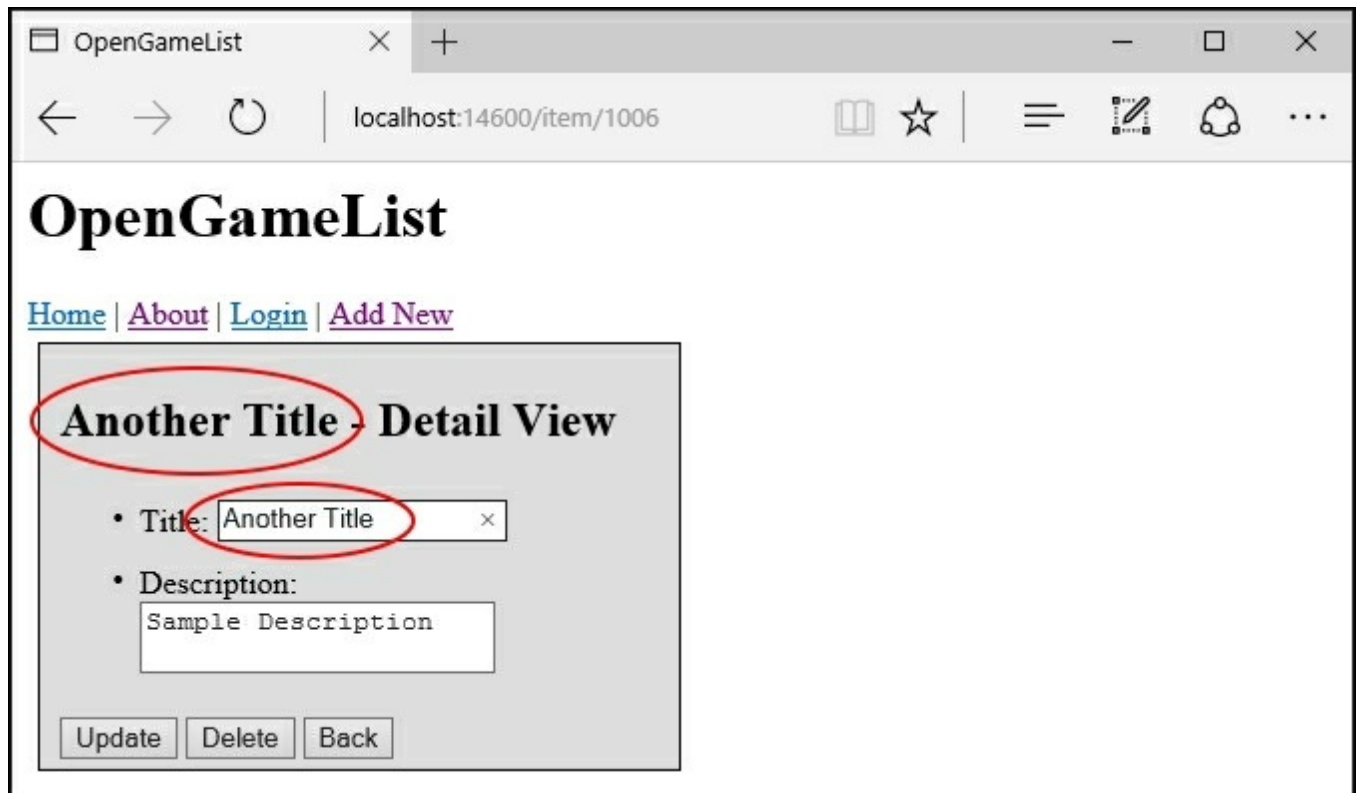
When we're done, we can click on the Update button to verify whether the changes are reflected in the **Welcome View**:
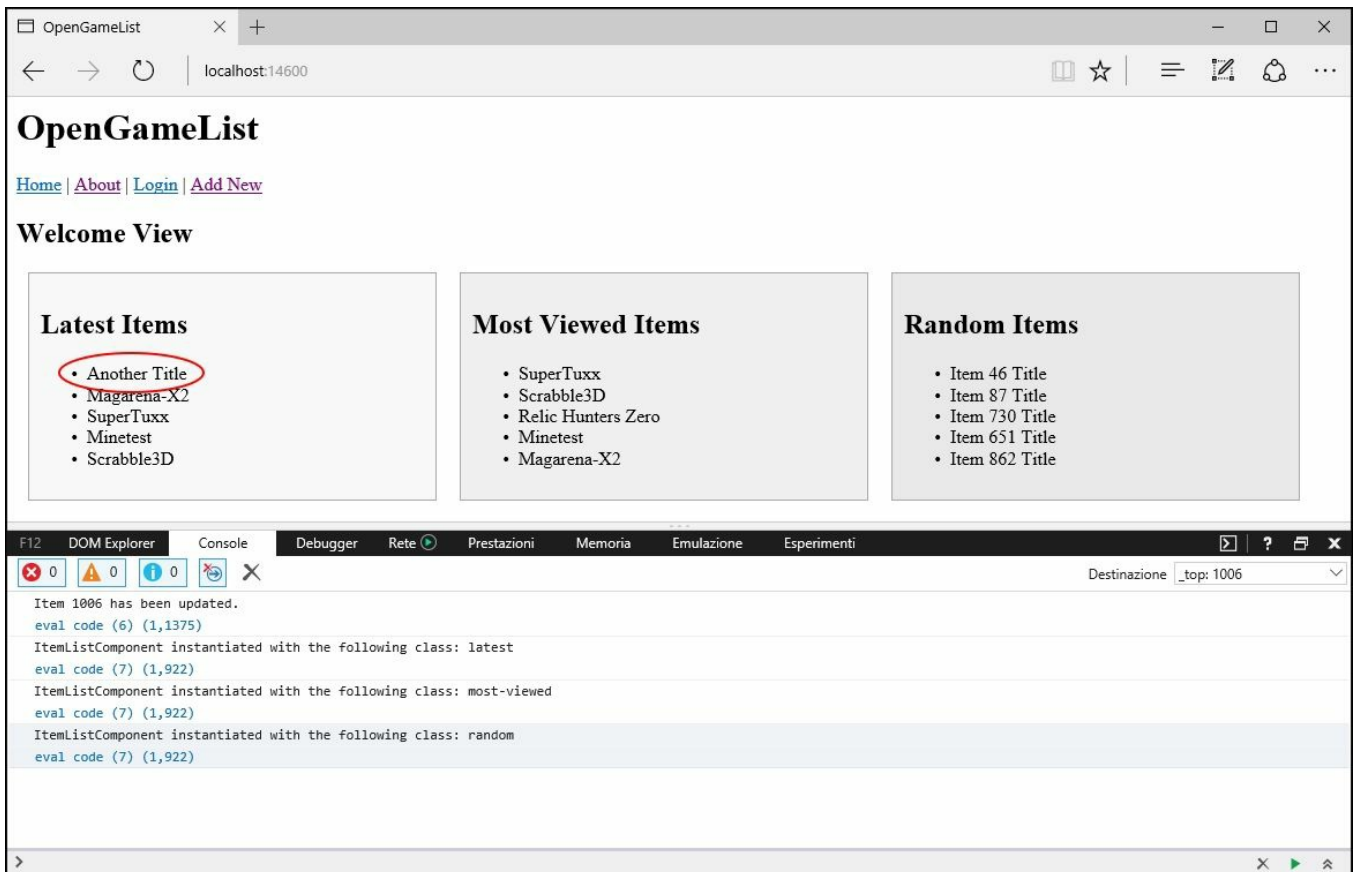
So far, so good. Again, let's see how it worked behind the scenes:

- Whenever the user selects an item from the **Welcome View**, the application will show the **Item - Detail View** in the update mode accordingly.
- Whenever the user performs a change to the item and confirms the operation by clicking on the **Update** button, the application will fire the `onUpdate` function. This will use the `ItemService` to return an `Observable` object ready to issue an HTTP call to persist these changes to the database using the Web API `Update` method.
- The `Observable` object will immediately fire thanks to the fluent `.subscribe()` call issued by the `onUpdate` function, which will also perform the following tasks:
  - In the case of success, update the local item object, output a message in the **Console** log, and re-route the user to the **Welcome View**
  - In the case of failure, output the error in the **Console** log
  - As soon as the user is sent back to the **Welcome View**, all the `ItemListComponents` will get re-instantiated again, so they will always load and display an updated set of item listings.

Delete test

Last but not least, we need to test the `delete` command.

From the **Welcome View**, click on the newly created (and updated) **Another Title** item to show the **Item - Detail View** in update mode. Immediately after clicking on the **Delete** button, we will be routed back to the **Welcome View**, where we can confirm that the item has actually gone:
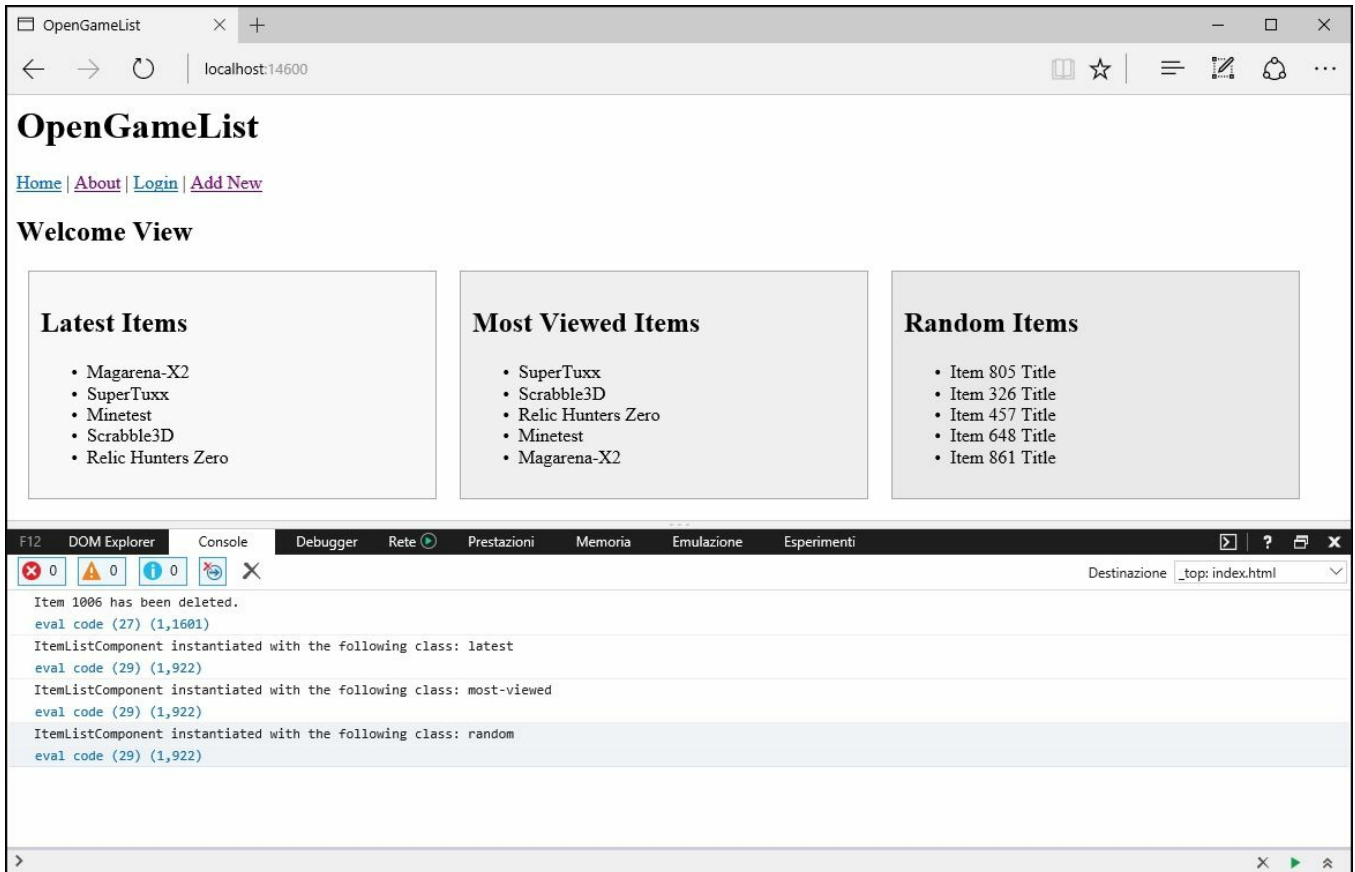
Everyone should have already figured it out by now, yet it's still useful to recap the workflow:

- When the user selects an Item from the **Welcome View**, the application will show the **Item - Detail View** in the update mode.
- When the user clicks on the **Delete** button, the application will fire the `onDelete` function, which will use the `ItemService` to return an **Observable** object ready to issue an HTTP call to delete the item from the database using the Web API `Delete` method.
- The `Observable` object will immediately fire thanks to the fluent `.subscribe()` call issued by the `onDelete` function, which will also perform the following tasks:
    - In the case of success, re-route the user to the **Welcome View**

- In the case of failure, output the error in the Console log
- As soon as the user is sent back to the **Welcome View**, all the `ItemListComponents` will get re-instantiated again, so they will always load and display an updated set of item listings (without the deleted item).

Splitting the ItemDetail component

Our application is going just fine, yet there are still some oddities that we should address as soon as we can. One of the most critical ones is that our `ItemDetail` component is acting like an editor way more than a viewer. Ideally, when the user clicks on an item, they should be presented with a view showing the item data in the display-only mode, with labels and text paragraphs instead of textboxes and text areas.

To better understand it, let's take a look at the following screenshot:
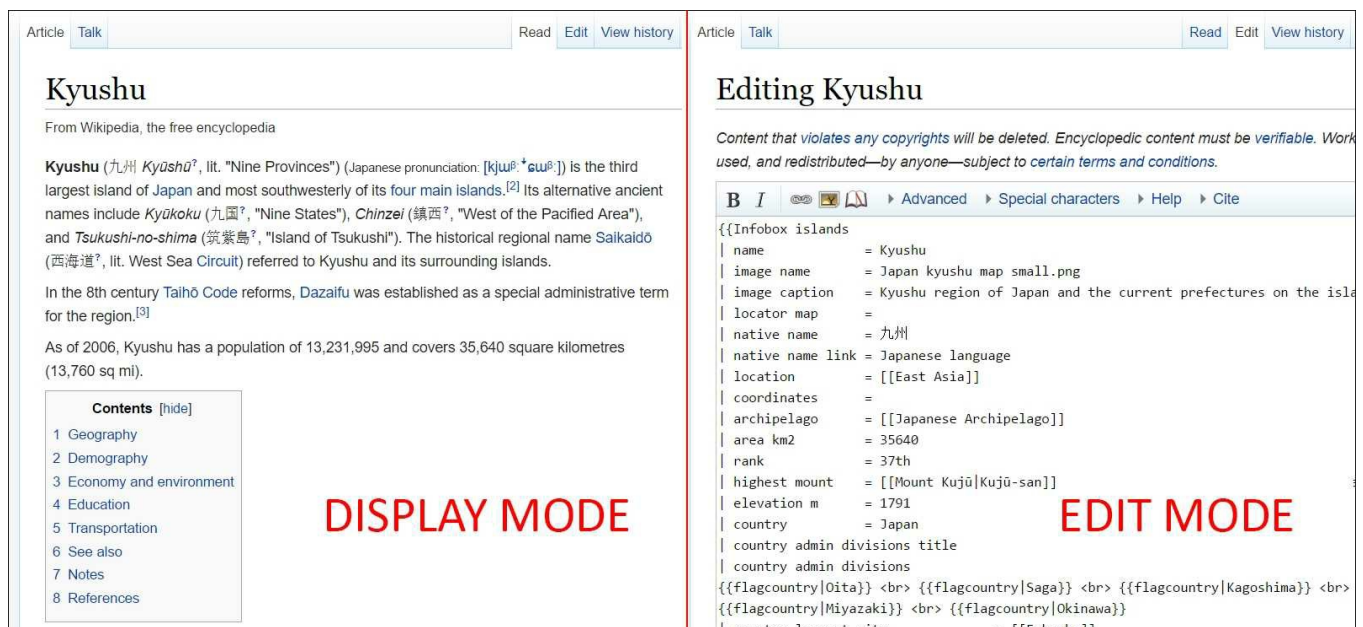
Source: Wikipedia

That's precisely what we're missing right now (and also the result we want to achieve). In order to do that, we need to split our current `ItemDetail` component into two different classes:

- `ItemDetailView`, to show the item just like the left side of the previous screenshot (**DISPLAY MODE**)
- `ItemDetailEdit`, to allow the user to insert or update items (**EDIT MODE**)

Most GUIs start with the former, allowing an authorized user to access the latter by clicking on an **Edit** button, link, or tab. Wikipedia/MediaWiki is no exception, as we can clearly see by looking at the tab menu near the top-right corner.

Note

Notice that we put `Insert` and `Update` together; the reason for that is simple, they usually share the same fields, with very few exceptions, so it's usually fine to handle them within the same component class. As a matter of fact, we already did that within our current `ItemDetail` component, which is basically the `ItemEdit` we're talking about.

That said, here's a brief summary of what we're about to do:

1. Add a new `ItemDetailView` component.
2. Rename the existing `ItemDetail` to `ItemDetailEdit`, because that's what it is.
3. Update the application `Root` module according to these changes.
4. Replace the `ItemDetail` route with two new ones pointing to the new `ItemDetailView` and `ItemDetailEdit` components; the former one will be called upon clicking on an item from the **Welcome View**, while the latter will be activated within it.
5. Change all occurrences pointing to `ItemDetail` to `ItemDetailView` or `ItemDetailEdit` throughout the whole project.
6. Add a tab menu UI element to connect the `ItemDetailView` and `ItemDetailEdit` components, following the same Wikipedia/MediaWiki interface approach.

Let's get to work.

Adding the ItemDetailView component

We'll start by adding a new `item-detail-view.component.ts` TypeScript file into the `/Scripts/app/` folder. The code will be quite similar to the existing `ItemDetail` component, except for the following:

- We will have headers, labels, and paragraphs here, instead of textboxes and text areas.
- We won't have the `onInsert`, `onUpdate`, and `onDelete` functions, as they serve no purpose in a display-only component. We will have an `onEdit` function instead, which will be used to switch to the `ItemEdit` component upon user interaction.

Here's suitable source code for the new `ItemDetailViewComponent` class:

```typescript
import {Component, OnInit} from "@angular/core";
import {Router, ActivatedRoute} from "@angular/router";
import {Item} from "./item";
import {ItemService} from "./item.service";

@Component({
    selector: "item-detail-view",
    template: `
      <div *ngIf="item" class="item-details">
        <h2>{{item.Title}}</h2>
        <p>{{item.Description}}</p>
      </div>
    `,
    styles: [`
        .item-details {
            margin: 5px;
            padding: 5px 10px;
            border: 1px solid black;
            background-color: #dddddd;
            width: 300px;
        }
        .item-details * {
            vertical-align: middle;
        }
        .item-details ul li {
            padding: 5px 0;
        }
    `]
})

export class ItemDetailViewComponent {
    item: Item;

    constructor(private itemService: ItemService,
```

```
        private router: Router,
        private activatedRoute: ActivatedRoute) {
    }

    ngOnInit() {
        var id = +this.activatedRoute.snapshot.params["id"];
        if (id) {
            this.itemService.get(id).subscribe(
                item => this.item = item
            );
        }
        else if (id === 0) {
            console.log("id is 0: switching to edit mode...");
            this.router.navigate(["item/edit", 0]);
        }
        else {
            console.log("Invalid id: routing back to home...");
            this.router.navigate([""]);
        }
    }
}
```

Before going further, there are a couple things worthy of attention:

1. We added a reference to the `item/edit` route, which doesn't exist yet; we did that on purpose because we know that we'll soon implement it.
2. We added an `if` condition within the `ngOnInit()` method to ensure that, if this component is initialized with a non-existing or invalid `id`, we'll re-route the call to the `ItemDetailEdit` component instead of handling it. That's perfectly fine, assuming that this component will be unable to do its display-only job without a valid `id` being passed together with the route; if something like that happens, we could either display a "this item does not exist" error page, throw an exception, or assume that the caller wants to create a new item and route them there accordingly. Although displaying an error page is almost always the best choice in such scenarios, we went for the routing for demonstration purposes.

Refactoring ItemDetail into ItemDetailEdit

The next step is easy: we just have to rename `ItemDetail` to `ItemDetailEdit` within the filesystem and inside the source code.

To complete the first step, right-click on the `item-detail.component.ts` file and rename it to `item-detail-edit.component.ts`.

Right after that, we also need to open it and change a couple of references within the source code.

The first thing we have to do is to change the `selector` from `item-detail` to `item-detail-edit`:

```
selector: "item-detail-edit",
```

Then we need to change the class name from `ItemDetailComponent` to `ItemDetailEditComponent`:

```
export class ItemDetailEditComponent {
```

That's it. We don't need to change anything else within this file for now, yet we're far from being done; we still have to change all the references currently pointing to `ItemDetail` throughout all the other class/components/files, replacing them with `ItemDetailEdit` and/or `ItemDetailView` accordingly.

Updating the Root module

Let's start with the `/Scripts/app/app.module.ts` file. Open it and change the `import` line pointing to the old `ItemDetailComponent` in the following way:

```
import {ItemDetailEditComponent} from "./item-detail-
edit.component";
import {ItemDetailViewComponent} from "./item-detail-
view.component";
```

Then scroll down to the `declarations` section and do the same:

```
    declarations: [
        AboutComponent,
        AppComponent,
        HomeComponent,
        ItemListComponent,
        ItemDetailEditComponent,
        ItemDetailViewComponent,
        LoginComponent,
        PageNotFoundComponent
    ],
```

Updating the Routes

The next thing we need to do is to update the `/Scripts/app/app.routing.ts` file. Open it and change its content accordingly (added/modified parts are highlighted):

```
import {ModuleWithProviders} from "@angular/core";
import {Routes, RouterModule} from "@angular/router";

import {AboutComponent} from "./about.component";
import {HomeComponent} from "./home.component";
import {ItemDetailEditComponent} from "./item-detail-
edit.component";
import {ItemDetailViewComponent} from "./item-detail-
view.component";
import {LoginComponent} from "./login.component";
import {PageNotFoundComponent} from "./page-not-found.component";

const appRoutes: Routes = [
    {
        path: "",
        component: HomeComponent
    },
    {
        path: "home",
        redirectTo: ""
    },
    {
        path: "about",
        component: AboutComponent
    },
    {
        path: "login",
        component: LoginComponent
    },
    {

        path: "item/edit/:id",

        component: ItemDetailEditComponent

    },

    {

        path: "item/view/:id",

        component: ItemDetailViewComponent

    },
    {
```

```
        path: '**',
        component: PageNotFoundComponent
    }
];

export const AppRoutingProviders: any[] = [
];

export const AppRouting: ModuleWithProviders =
RouterModule.forRoot(appRoutes);
```

As we can see, we performed two major changes here:

- Near the top of the file, within the `import` section, we did the same changes we made to the `AppModule` class a moment ago
- We replaced the now obsolete `item` route with two new routes, `item/edit` and `item/view`, respectively pointing to our brand new `ItemDetailEdit` and `ItemDetailView` component classes

Tracking the outdated references

Now we need to find all the references to the outdated `ItemDetail` component class within our application files and update them accordingly. Luckily enough we're using TypeScript, so we don't need to stoop to using find and replace or other loose techniques; we just have to look at our **Task Runner Explorer** window and see if there are compiler **Errors**:
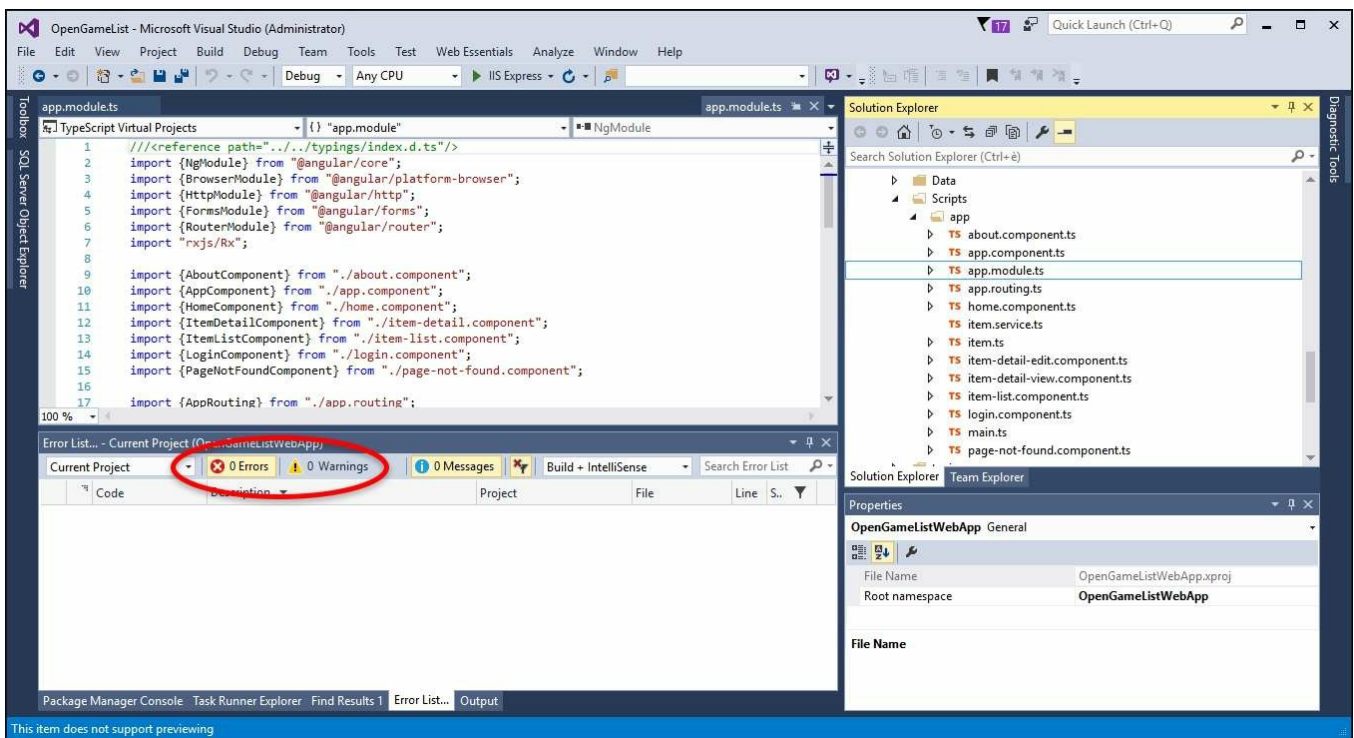
As we can see, we're quite lucky: there are no errors, meaning that there is no other class referencing the old `ItemDetail` class and/or the `item-detail.component.ts` filename.

The routes, however, are a whole different story; they work as literals, so there is no way the compiler will tell us anything. Luckily enough, there are only two components that were using the old `item` route: `AppComponent` and `ItemListComponent`.

To update the former, open the `/Scripts/app/app.component.ts` file and replace the value of the `routerLink` directive for adding a new item (updated lines are highlighted):

```
<h1>{{title}}</h1>
<div class="menu">
```

```
    <a class="home" [routerLink]="['']">Home</a>
    | <a class="about" [routerLink]="['about']">About</a>
    | <a class="login" [routerLink]="['login']">Login</a>
    | <a class="add" [routerLink]="['item/edit', 0]">Add New</a>
</div>
<router-outlet></router-outlet>
```

As for `ItemListComponent`, open the `/Scripts/app/item-list.component.ts`
file, scroll down to the `onSelect` method, and change its contents accordingly
(updated lines are highlighted):

```
    onSelect(item: Item) {
        this.selectedItem = item;
        console.log("Item " + this.selectedItem.Id + " has been
clicked: loading item viewer...");
        this.router.navigate(["item/view", this.selectedItem.Id]);
    }
```

Implementing the tab menu

We're still missing something very important: there's no way we can switch from `ItemDetailView` to `ItemDetailEdit` and vice versa, which is a required condition for our application to work; as a matter of fact, we can't even properly test what we've done up till now until we add something that would allow us to do that.

Without further ado, this means that it's time to implement the Wikipedia-style tab menu we were talking about a while ago. In order to do that, we need to update the following sections of the `item-detail-view.component.ts` and `item-detail-edit.component.ts` files:

- The `template`, because our tab menu will need an HTML structure
- The `class` implementation code, since we will definitely need to implement an event handler method to respond to user interaction (the actual tab click) and put it into action accordingly
- The `styles`, because we want our **tab menu** to blend with the components that will implement it in a nicely fashion

Template

Open the `item-detail-view.component.ts` file and replace the content of the `template` section with the following (new/updated lines are highlighted):

```
<div *ngIf="item" class="item-container">

    <div class="item-tab-menu">

        <span (click)="onItemDetailEdit(item)">Edit</span>

        <span class="selected">View</span>

    </div>

    <div class="item-details">

        <div class="mode">Display Mode</div>
        <h2>{{item.Title}}</h2>
        <p>{{item.Description}}</p>
    </div>
</div>
```

We changed quite a few things here, yet the template is simple enough to easily understand what happened:

- We wrapped everything within a `div` having an `item-container` CSS class. Notice that we also conveniently moved the `*ngIf` expression there, removing it

from the `div.item-details` element as the latter isn't the main container anymore.

- We added a `div` with an `item-tab-menu` CSS class that is going to be our tab menu. It contains two `span` elements representing, respectively, the currently selected tab (**View**) and the tab we can switch to (**Edit**). Notice `onItemDetailEdit(item)` bound to the latter; we'll get to it soon enough.

Now open the `item-detail-edit.component.ts` file and again replace the `template` section content with the following (new/updated lines are highlighted):

```
<div *ngIf="item" class="item-container">

    <div class="item-tab-menu">

        <span class="selected">Edit</span>

        <span *ngIf="item.Id != 0"
(click)="onItemDetailView(item)">View</span>

    </div>

    <div class="item-details">

        <div class="mode">Edit Mode</div>

        <h2>{{item.Title}}</h2>
        <ul>
            <li>
                <label>Title:</label>
                <input [(ngModel)]="item.Title" placeholder="Insert
the title..." />
            </li>
            <li>
                <label>Description:</label>
                <textarea [(ngModel)]="item.Description"
placeholder="Insert a suitable description..."></textarea>
            </li>
        </ul>
        <div *ngIf="item.Id == 0" class="commands insert">
            <input type="button" value="Save"
(click)="onInsert(item)" />
            <input type="button" value="Cancel" (click)="onBack()"
/>
        </div>
        <div *ngIf="item.Id != 0" class="commands update">
            <input type="button" value="Update"
(click)="onUpdate(item)" />
            <input type="button" value="Delete"
(click)="onDelete(item)" />
            <input type="button" value="Cancel"
```

```
(click)="onItemDetailView(item)" />
        </div>
    </div>
</div>
```

As we can see, a lot of stuff happened here as well:

- Again, we wrapped everything into a `div` with an `item-container` CSS class, moving the `*ngIf` expression there.
- Again, we added the HTML structure of our tab menu; the two `span` elements are the same, yet they have obviously switched their respective roles: the **View** one has an `onItemView(item)` event attached, while **Edit** is now the selected one, so isn't expected to do anything. Notice that we added another `*ngIf` expression to the **View** tab menu item, meaning that it will show itself only if the `id` item is not 0; this will prevent the `View` tab from being displayed whenever this component is accessed for adding a new item. That's a good thing to do since we wouldn't be able to "view" an item that doesn't exist in the database yet.
- We renamed the **Back** button **Cancel** and changed its behavior accordingly; instead of going to the home/welcome view by using the `onBack()` method, it will now route to the current item's `ItemDetailView` using the `onItemDetailView(item)` method, which by the way is not implemented yet.

Class code

The next thing we have to do is to implement the missing methods that we already took for granted in the previous paragraph.

Go back to the `item-detail-view.component.ts` file and add a new `onItemDetailEdit(item)` method right after the `ngOnInit` one:

```
onItemDetailEdit(item: Item) {
    this.router.navigate(["item/edit", item.Id]);
}
```

Right after that, switch to the `item-detail-edit.component.ts` file and add the following code after the `onBack` method, which should be the last one:

```
onItemDetailView(item: Item) {
    this.router.navigate(["item/view", item.Id]);
}
```

That's it; both of these routes have already been set, so there is nothing else to do there.

Styles

It's time to pimp our components a little; we won't do anything fancy. We'll leave it to [Chapter 6](), *Applying Styles*, yet we could use a slightly improved GUI to better acknowledge what we're doing.

Let's start with the `item-detail-view.component.ts` file: open it up, then replace the content of the `styles` section with the following:

```
.item-container {
    width: 600px;
}

.item-tab-menu {
    margin-right: 30px;
}

.item-tab-menu span {
    background-color: #dddddd;
    border: 1px solid #666666;
    border-bottom: 0;
    cursor: pointer;
    display: block;
    float: right;
    margin: 0 0 -1px 5px;
    padding: 5px 10px 4px 10px;
    text-align: center;
    width: 60px;
}

.item-tab-menu span.selected {
    background-color: #eeeeee;
    cursor: auto;
    font-weight: bold;
    padding-bottom: 5px;
}

.item-details {
    background-color: #eeeeee;
    border: 1px solid black;
    clear: both;
    margin: 0;
    padding: 5px 10px;
}

.item-details * {
    vertical-align: middle;
}

.item-details .mode {
    font-size: 0.8em;
    color: #777777;
}
```

```
.item-details ul li {
    padding: 5px 0;
}
```

Right after that, do the same with the `styles` section of the `item-detail-edit.component.ts` file:

```
.item-container {
    width: 600px;
}

.item-tab-menu {
    margin-right: 30px;
}

.item-tab-menu span {
    background-color: #dddddd;
    border: 1px solid #666666;
    border-bottom: 0;
    cursor: pointer;
    display: block;
    float: right;
    margin: 0 0 -1px 5px;
    padding: 5px 10px 4px 10px;
    text-align: center;
    width: 60px;
}

.item-tab-menu span.selected {
    background-color: #eeeeee;
    cursor: auto;
    font-weight: bold;
    padding-bottom: 5px;
}

.item-details {
    background-color: #eeeeee;
    border: 1px solid black;
    clear: both;
    margin: 0;
    padding: 5px 10px;
}

.item-details * {
    vertical-align: middle;
}

.item-details .mode {
    font-size: 0.8em;
    color: #777777;
}
```

```css
.item-details ul li {
    padding: 5px 0;
}

.item-details input[type="text"] {
    display: block;
    width: 100%;
}

.item-details textarea {
    display: block;
    width: 100%;
    height: 60px;
}

.commands {
    text-align: right;
    margin: 10px 20px 10px 10px;
}
```

Not much to explain here; we just changed the aspect of these two controls so the **tab menu** can fit into them in a decent-looking way for the upcoming test run.

Testing it out

Now that our splitting job is finally over, it's time to perform an extensive test on what we just did.

Launch the application by hitting *F5* and check that the **Home /welcome** page is still alive and well. We didn't change anything there, so we can go straight on to our brand new `ItemDetailView` component by left-clicking on one of the elements within the **Latest Items** listing, for example **Magarena-X2**.

If everything is working as it should, we should be greeted by something like the following:
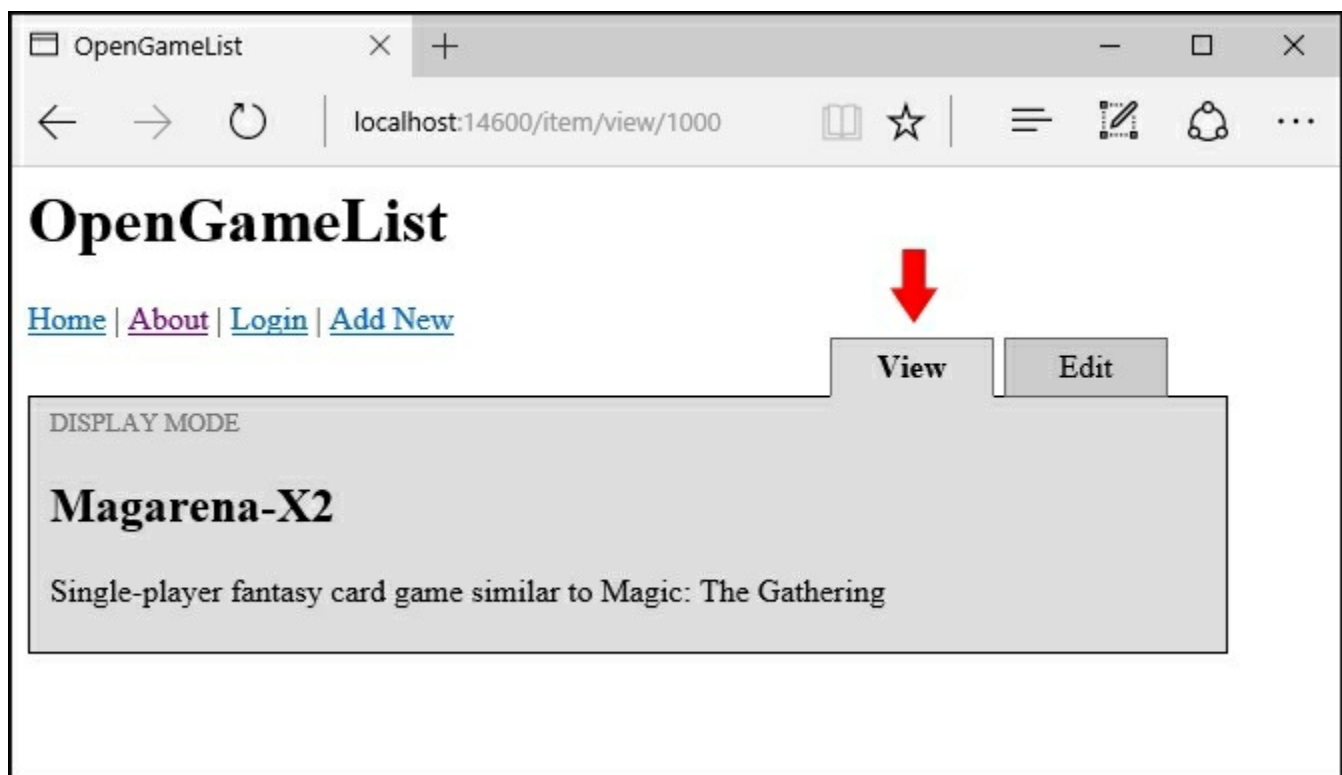
We can easily see that we opened the **ItemDetailView** in **DISPLAY MODE** because:

- The **View** tab is clearly selected
- There's a **DISPLAY MODE** label in the top-left corner of the item containing box
- We don't see any input textboxes/text areas within the view

This is the first time we can see the Wikipedia-style tab menu we implemented in the

previous paragraph; it seems to be working just like it should.

Let's now click on the **Edit** tab and see what happens:
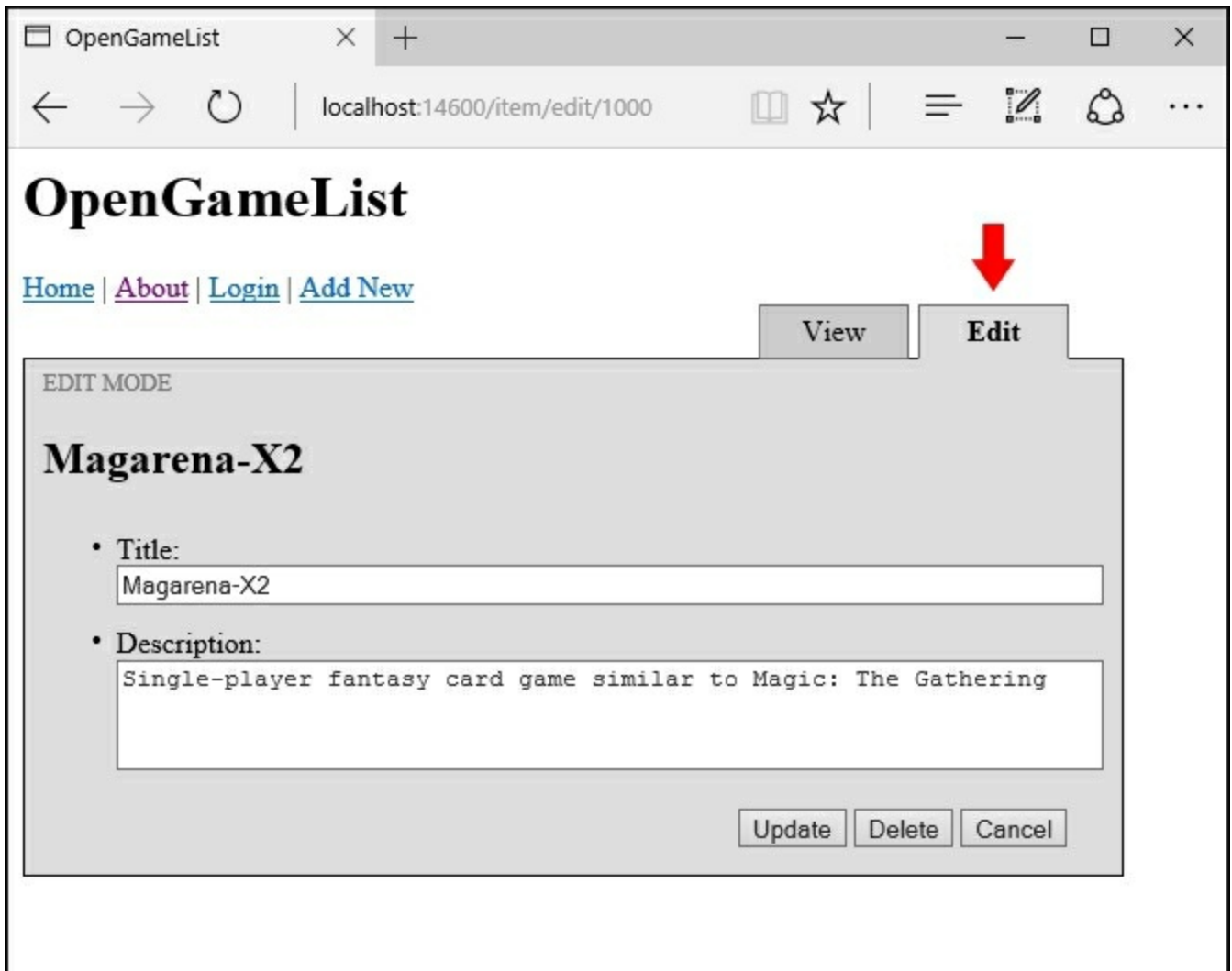
The **EDIT MODE** seems to be looking fine as well. The tab menu is also still working as expected, as the **Edit** tab is now clearly appearing as the one on top.

Let's try to append the following line to the description text area:

> *We can't call this a test without a sample update attempt.*

Then, click on the **Update** button. As soon as we do that, we should see something like this:
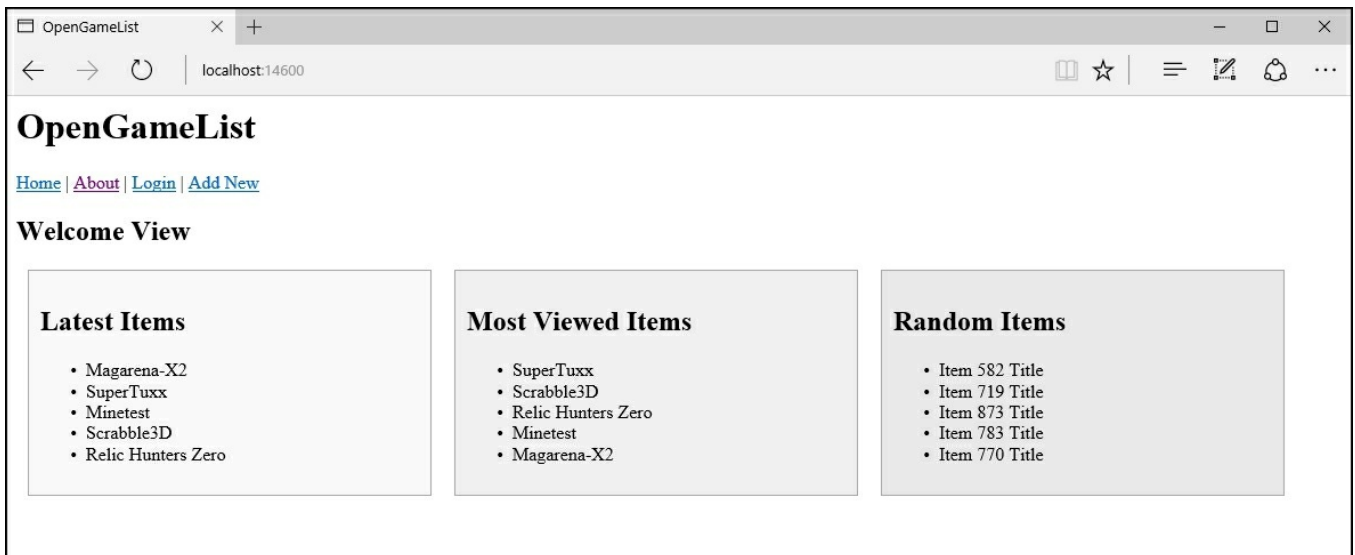
The **Update** button routed us back to the **Home**/**Welcome View;** we never changed its behavior, so it's still working like that. While this may have been viable when we only had a single `ItemDetail` controller, now it most certainly isn't; it should bring us back to `ItemDisplayView`, just like *Wikipedia*/*MediaWiki* works when we update an item there.

In order to fix that, open the `item-detail-edit.component.ts` file and perform the following changes to the `onUpdate` method (modified code is highlighted):

```
onUpdate(item: Item) {
    this.itemService.update(item).subscribe(
        (data) => {
            this.item = data;
            console.log("Item " + this.item.Id + " has been
updated.");
            this.router.navigate(["item/view", this.item.Id]);
        },
        (error) => console.log(error)
    );
}
```

What we did here is quite obvious: We replaced the destination route from root one (bound to `HomeController`) to `item/view`, which is connected to `ItemDetailViewController`, also including the `id` parameter.

Let's test it out; hit *F5* to re-run the application in **Debug** mode, then left-click on the **Magarena-X2** item to launch the **ItemDetailView** controller again. Once there, click on the **Edit** tab and append another test line to the item's **Description** text area:

The previous attempt wasn't good enough; now it should work.

When we're done, click on the **Update** button and see what happens:
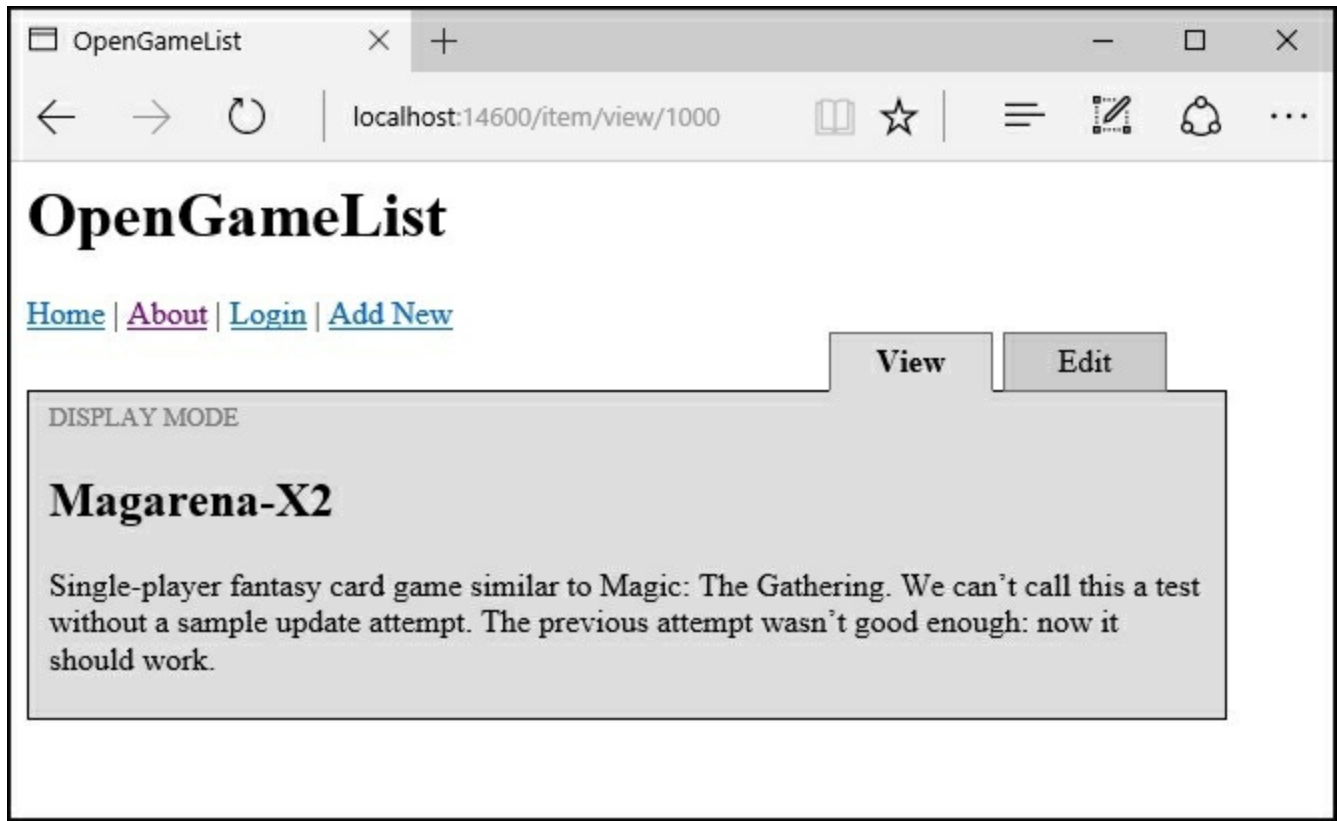
It seems like it worked! We managed to split the `ItemDetail` controller into two, reproducing a MediaWiki-like behavior when it comes to displaying and/or editing our items.

Suggested topics

RESTful conventions, HTTP verbs, HTTP status, Angular 2 observables, Angular 2 observers, Angular 2 subscriptions, MediaWiki, Cascading Style Sheets.

Summary

Before going further, let's do a quick recap of what we did throughout this chapter.

Having replaced the fake data provider with a real one, we made good use of it by implementing **Add**, **Update**, and **Delete** features for our **ItemDetail** controller. In order to do this, we improved our **ItemService**, implementing the corresponding `add`, `update`, and `delete` methods. Then we also updated the client GUI by creating new buttons, event handlers, and routes to properly handle them.

While implementing the process of creating a new item, instead of adding a new route, we chose to exploit the existing one already pointing to `ItemDetail` by passing an `id` with a value of 0, thus creating an internal convention for our own personal usage. Once done, we ran a full test of the updated UI to check that everything worked out well.

It most certainly was, yet we started noticing that our controller looked like an editor way more than a viewer. That didn't match the original plan, so we planned to split the `ItemDetail` controller into two separate classes: `ItemDetailView` to show the item in readable fashion (**DISPLAY MODE**) and `ItemDetailEdit` to handle add, update, and delete commands (**EDIT MODE**). We also thought about making the users able to toggle between them with the help of a dedicated tab menu just like to the one used by MediaWiki.

Splitting the `ItemDetail` controller implied a straightforward, yet rather long, series of tasks: we added a new TypeScript file for the `ItemDetailView` component, filling it with suitable code, then we put in place a full code and filesystem refactoring of the `ItemDetail` component, which we renamed to `ItemDetailEdit` everywhere; the TypeScript compiler saved us some valuable time here, identifying the outdated references and allowing us to promptly fix/update them.

Eventually we implemented the tab menu; in order to do that we had to perform some changes to the controller's `templates` and `styles` sections, as well as implementing the dedicated methods within the class code to handle the click event upon each tab.

After all these changes, we felt the urge to do another round of tests. It turned out it was a good call, as we found a minor issue within the `ItemDetailEdit` controller's `onUpdate` method, which was still calling the **Home** route upon completion instead of the more appropriate `ItemDetailView` one; we fixed it smoothly, making our application ready for the following chapters.

Chapter 6. Applying Styles

Up to this point, we have done our best to keep the layout as simple as we could, so we could focus entirely on the server-side and client-side coding aspects of our app: ASP.NET Core Web API Controllers, Angular 2, C#, and TypeScript. Keeping the layout to a minimum is generally a wise approach when we're learning something new, it also has a few downsides, though, the biggest one being the blatant fact that our application is rather unattractive, to say the least: there is no user, client, or customer that wouldn't say that... or worse.

How bad is it, doc?

It almost entirely depends on what we're planning to do with the project we've been working on; as we just said, while we're working our way through tutorials, demos, or sample projects, it's not bad at all, for at least a couple of good reasons:

- We will greatly benefit from keeping our focus on .NET and Angular 2, leaving the rest for later; applying styles is something that w can easily do whenever we feel like it, even if we don't have a decent grip on stylesheet language already.
- It's generally a good idea to restrain ourselves from doing any relevant style implementation until we can fully understand where and how to do that conveniently; to put it in other words, we shouldn't do styles until we find a suitable approach for doing that within the given scenario and/or environment.

That's why we chose to take this path in the first place; we're definitely in the learning phase, after all. However, since we planned to build a production-ready native web application, we can't restrain ourselves from applying some styling any longer; there's no way that our imaginary product owner would be satisfied otherwise. It's time to dress our (mostly) naked doll and make it as pretty as we can.

Introducing LESS

If we've worked with style sheets within the last few years, there's no chance we won't have heard of **LESS**; however, for the sake of those who didn't, let's take a few words to talk about it. Before getting to that, though, we must briefly introduce the concepts of style sheet language and **Cascading Style Sheets (CSS)**.

Tip

This paragraph is mostly aimed at those who have never used LESS before. If you have some experience with LESS already or feel like you don't need to know anything else about why we're going to use it, you might as well skip it entirely and jump to the next paragraph: **Install and Configure LESS**.

Style sheet languages

A style sheet language, also known as style language, is a programming language used to define the presentation layer's UI design rules of a structured document. We can think of it as a skin or a theme that we can apply to a logical item (the structured document) to change its appearance. For example, we can make it look blue, red, or yellow; we can make the characters bigger or smaller, thinner or wider; we can change the text spacing, alignment, and flow; and so on.

Using dedicated style sheet languages gives developers the chance to separate the presentation layer's **code and structure** (respectively JavaScript and HTML) from the **UI design rules**, thus enforcing the **separation of concerns** (**SoC**) principle within the presentation layer itself.

When it comes to web pages, web applications, and anything else that mostly uses HTML, XHTML, XML, and other markup language-based documents, the most important style sheet language undoubtedly is CSS.

CSS

It was December 17, 1996, when the **World Wide Web Consortium (W3C)** released the official W3C CSS Recommendation for the style sheet language that would be known as CSS1. CSS2 came less than two years later (May 1998), while its revised version, CSS2.1, took considerably more time (June 2011).

Starting from CSS3, things started to become more complex, since the W3C ditched the single, monolithic specification approach by splitting it into separate documents called modules, each one of them following its very own publishing, acceptance, and recommendation history. Starting in 2012, with four of these (Media Queries, Namespaces, Selectors, and Color) being published as formal recommendations and full CSS2.1 backward-compatibility, CSS3 quickly became the most adopted style sheet language standard for the development of new websites.

CSS code sample

Regardless of their version, each adding new features while maintaining backward compatibility with the previous one(s), CSS stick to the following syntax:

```css
.item-details {
    background-color: #dddddd;
    border: 1px solid black;
    margin: 0;
    padding: 5px 10px;
}
```

We've seen this code before; it's a class we added in our application's `item-detail-view.component.ts` file in a previous chapter. It says that any element with the `item-details` class will have a light-grey background color, a black, solid, and pixel-wide border, no margin against the surrounding elements, and a certain amount of padding between its borders and the content. Simple enough, isn't it?

What is LESS and why to use it

LESS is a cascading style sheet pre-processor; we can think of it as a server-side script for CSS files, enabling us to do a number of things that CSS doesn't support (yet), just like PHP and/or ASP can do for an HTML page. The following diagram should help us to better understand the concept:
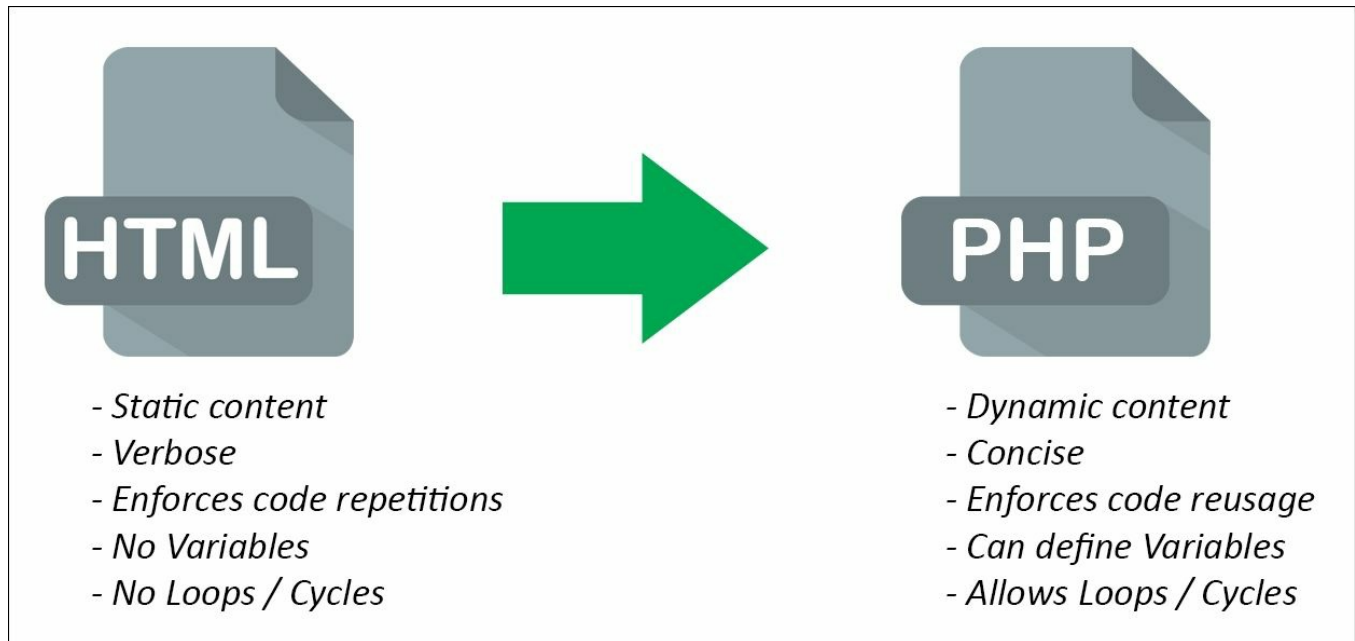
These are the main advantages of using a hypertext pre-processor instead of writing raw **HTML** pages; we're talking about **PHP**, but the same goes for ASP.NET Web Forms, Razor, and basically everything else.

The following are the advantages of using **LESS** instead of writing raw **CSS** files:
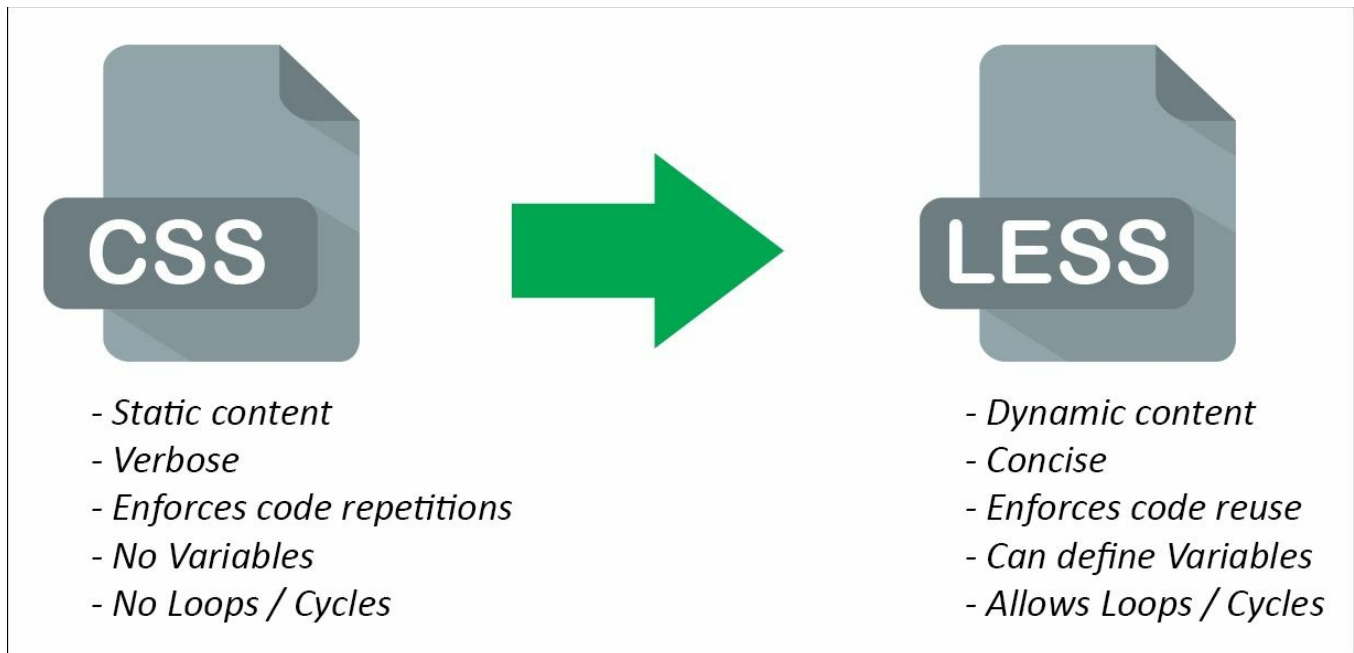
As we can see, they serve the exact same purpose in terms of assisting, improving, and enhancing the development effort.

Making the switch from static stylesheets to dynamic stylesheets is just as easy as switching from static HTML pages to PHP or ASP dynamic pages; they both feature a nested metalanguage that can extend the base static language in a pure backward-compatible fashion. This means that a valid CSS file is also a valid LESS file, just as a valid HTML file is also a valid PHP or ASP file.

There are also some key differences between hypertext pre-processors and stylesheet pre-processors, the most important being how web servers deal with them.

Hypertext pre-processors such as PHP and ASP are compiled by the Web Server upon each request; the Web Server compiles them on the fly and then serves the resulting HTML for each request-response flow. Conversely, Stylesheet pre-processor files are usually compiled into standard CSS files before being published; in other words, the web service doesn't know about the existence of these files, as it just serves the resulting CSS-compiled result.

This also means that using a stylesheet pre-processor will have no performance impact on the server, unless we choose to install some experimental and still highly inefficient handlers, extensions, modules, or client-side scripts that will compile the source files on the fly.

Note

IMPORTANT NOTE

From now on, we'll take for granted that the reader has a decent knowledge of CSS files, syntax, selectors, and their common use within HTML pages.

If this is not the case, we strongly suggest to learn the core CSS concepts before going further, using the Learning CSS website, maintained and hosted by W3C, featuring a massive number of useful guides, tutorials, and articles:

https://www.w3.org/Style/CSS/learning.

Variables

Among the most valuable LESS features, there is variable support. This is a brief example of what we can do with it:

```less
// Variables can be declared as such:
@link-color: #red;
@link-color-hover: lightcoral;

// And then they can be referenced like this:
a, span. link {
    color: @link-color;
}

a:hover, span.link:hover {
    color: @link-color-hover;
}
```

As we might have noticed, double-slash style (`//`) inline comments are supported as well, while CSS only allows the slash-asterisk (`/**/`) syntax.

Import directives

Another LESS key feature is the capability of importing other CSS and LESS files. If we're familiar with the standard CSS `@import`, we know that it can only be used at the beginning of the file to issue the loading of an external CSS file. With LESS, we can do the following:

```less
// look for a style.less file and process + import its contents.
@import "style";

// look for a style.less file and process + import its contents.
@import "style.less";

// look for a style.css file and import its contents (no
processing).
@import "style.css";
```

Notice that the behavior depends on the imported file extension. These defaults can be overridden with the following `options` switches:

```
// link/use a Less file without including it in the output.
@import (reference) "something.less";

// include the file in the output without processing it.
@import (inline) "something.less";

// pretend this is a LESS file, regardless of the extension.
@import (less) "something.css";

// pretend this is a CSS file, regardless of the extension.
@import (css) "something.less";

// never include this file more than once (default behavior).
@import (once) "something.less";

// always include this file in the output, even multiple times.
@import (multiple) "something.less";

// do not break the compile operation if the file is not found.
@import (optional) "something.less";
```

If we need to specify multiple `options` within a single `@import` statement, we can do that by separating each one of them with a comma:

```
// take it as a LESS file, import once, skip if not found.
@import (less,once,optional) "something.css";
```

Nested selectors

We will be able to nest selectors within other selectors, thus making our code more succinct and readable. Just to use a quick example, we can shrink this:

```
item-list {
    border: 0;
    margin: 0;
    padding: 0;
    vertical-align: top;
    display: block;
}

item-list.latest {
        background-color: #f6f6f6;
}

item-list.latest h3 {
    background-image: url(/img/latest-icon.png);
}
```

Into something like the following:

```
item-list {
    border: 0;
    margin: 0;
    padding: 0;
    vertical-align: top;
    display: block;
    &.latest {
        // the & char represents the current selector parent.
        // in this scenario, it stands for: item-list.latest.
         background-color: #f6f6f6;
         h3 {
             background-color: @color-latest;
             background-image: url(/img/latest-icon.png);
         }
    }
}
```

It might not be such a big deal for small-scale CSS files, yet it's a great readability improvement for big ones.

Mixins

Being able to not repeat ourselves is a key principle of all computer programming languages; however, it's not easy to respect that within standard CSS files, because we would often be forced to write something like this:

```
.button-s {
    background-color: blue;
    border: 1px solid black;
    border-radius: 5px;
    font-family: Verdana;
    font-size: 0.8em;
    width: 100px;
}

.button-m {
    background-color: blue;
    border: 1px solid black;
    border-radius: 5px;
    font-family: Verdana;
    font-size: 1em;
    width: 200px;
}

.button-l {
    background-color: blue;
    border: 1px solid black;
    border-radius: 5px;
    font-family: Verdana;
```

```
    font-size: 1.2em;
    width: 300px;
}
```

With LESS, we can shrink it into this:

```
.button-s {
    background-color: blue;
    border: 1px solid black;
    border-radius: 5px;
    font-family: Verdana;
    font-size: 0.8em;
    width: 100px;
}

.button-m {
    .button-s;
    font-size: 1em;
    width: 200px;
}

.button-l {
    .button-s;
    font-size: 1.2em;
    width: 300px;
}
```

In other words, a mixin is a selector reference within another selector. That's another great feature that can save us a lot of time whenever we're dealing with large CSS files.

Extend pseudo-class

Another great feature is the LESS `:extend` pseudo-class, which can be used to apply all properties of a class to another class, optionally including, using the `all` keyword, all the child classes and pseudo-classes. To use a quick example, take the following CSS code:

```
.link {
    color: white;
    background-color: blue;
}

.link:before {
    content: ">";
}

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

```
}

.link-red:before {
    content: ">";
}
```

This could be conveniently written this way using LESS:

```
.link {
    color: white;
    background-color: blue;
    :before {
        content: ">";
    }
}

.link-red {
    &:extend(.link all);
    background-color: red;
}
```

Notice how, since we've used the `all` keyword, we don't have to repeat the `:before` pseudo-class of the base `.link` selector, as it will be applied to `.link-red` as well.

LESS docs and support

We won't go any further than that with LESS, as it would take us far from the scope of this book. From now on, we'll take for granted that everything that we're going to do with it will be acknowledged and understood.

For the sake of simplicity, we won't use anything different from what we briefly introduced in the previous chapters; however, we strongly suggest to take a look at the advanced features (parametric mixins, functions, loops, guards, and more) as soon as we have the chance; they can hardly fail to pay off. We can learn more about them from the LESS official webpage at the following URL: http://lesscss.org/

Systematically Awesome Style Sheets, Stylus, and other alternatives

As most readers probably know, or can easily imagine, LESS is not the only style sheet pre-processor language out there. As a matter of fact, it was released more than two years after **Systematically Awesome Style Sheets (Sass)**, also known as SCSS, which had served the exact same purpose since 2007. Sass can offer basically the same set of features as LESS and came out first, so why shouldn't it be used instead?

The answer is simple and strongly resembles what we have already said when we had to choose between `Gulp` and `Grunt` in [Chapter 1](#), *Getting Ready,*, no one will ever get fired for picking Sass or other viable alternatives, such as Stylus and Switch CSS, instead of LESS. We're free to choose the style sheet pre-processor we like the most, as long as we can use it without issues, meaning that Visual Studio and/or the Task Runner we chose is able to support it. Luckily enough, all of them are now widely supported by many Visual Studio Extensions and `Gulp` plugins (or `Grunt` modules), so it won't make any significant difference. That said, we're going to set up and configure LESS because we find it more straightforward, easy to use, and slightly more suited for a Windows environment than its counterparts, at least for now.

Configuring LESS

LESS happens to be written in JavaScript, so installing it can be as easy as downloading its JavaScript library (`less.js`), linking it to our `index.html` page, and letting its magic work without having to set up anything else.

Should we do that, then? Not a chance. As we said earlier, delegating the compilation task on the client side would be highly inefficient, especially in a client-intensive Angular 2-based native web app. This is what we're going to do instead:

- Add a new `style.less` file to our project.
- Update our **Gulp** script to process and compile it, outputting the resulting `style.css` file into the `/wwwroot/` folder.
- Link the resulting `.css` file to the `/wwwroot/index.html` root application file.
- Make a quick test before starting our styling job.

Let's get to work.

Adding the LESS file

Considering how many files we've added so far, this is going to be a trivial task. Add a new `/less/` folder inside the `/Scripts/` root directory, then right-click on it, choose **Add | New Item**, and select **LESS Style Sheet** from the **client-side** set item tab: name the new file `style.less` and click on **OK**.

Once done, open the newly created file and add the following content:

```less
item-list {
    min-width: 332px;
    border: 1px solid #aaaaaa;
    display: inline-block;
    margin: 0 10px;
    padding: 10px;
    &.latest {
        background-color: #f9f9f9;
    }
    &.most-viewed {
        background-color: #f0f0f0;
    }
    &.random {
        background-color: #e9e9e9;
    }
}
```

We might recognize the preceding lines, as these are the `styles` of our `/Scripts/app/home.component.ts` Angular 2 component file, we just applied some mixins as described previously to shrink the code a bit.

Updating Gulp script

The first thing we need to do is to add the LESS plugin for `Gulp`. Open the `package.json` file, locate the `devDependencies` node and add the following line to the `gulp` section. We'll be placing it right after the `gulp-concat` package to keep the alphabetical sorting:

```
"gulp": "^3.9.1",
"gulp-clean": "^0.3.2",
"gulp-concat": "^2.6.0",
"gulp-less": "^3.1.0",
"gulp-sourcemaps": "^1.6.0",
"gulp-typescript": "^2.13.6",
"gulp-uglify": "^2.0.0",
"typescript": "^1.8.10"
```

As soon as we save the file we'll issue a real-time refresh to the project's NPM dependencies.

Now switch to the `gulpfile.js` file and get ready to add/modify a few lines of code (new/updated lines are highlighted):

```
var gulp = require('gulp'),
    gp_clean = require('gulp-clean'),
    gp_concat = require('gulp-concat'),
    gp_less = require('gulp-less'),
    gp_sourcemaps = require('gulp-sourcemaps'),
    gp_typescript = require('gulp-typescript'),
    gp_uglify = require('gulp-uglify');

/// Define paths
var srcPaths = {
    app: ['Scripts/app/main.ts', 'Scripts/app/**/*.ts'],
    js: [
        'Scripts/js/**/*.js',
        'node_modules/core-js/client/shim.min.js',
        'node_modules/zone.js/dist/zone.js',
        'node_modules/reflect-metadata/Reflect.js',
        'node_modules/systemjs/dist/system.src.js',
        'node_modules/typescript/lib/typescript.js'
    ],
    js_angular: [
        'node_modules/@angular/**'
    ],
    js_rxjs: [
        'node_modules/rxjs/**'
    ],
    less: [
```

```
        'Scripts/less/**/*.less'

    ]
};

var destPaths = {
    app: 'wwwroot/app/',
    css: 'wwwroot/css/',
    js: 'wwwroot/js/',
    js_angular: 'wwwroot/js/@angular/',
    js_rxjs: 'wwwroot/js/rxjs/'
};

// Compile, minify and create sourcemaps all TypeScript files and
place them to wwwroot/app, together with their js.map files.
gulp.task('app', ['app_clean'], function () {
    return gulp.src(srcPaths.app)
        .pipe(gp_sourcemaps.init())

.pipe(gp_typescript(require('./tsconfig.json').compilerOptions))
        .pipe(gp_uglify({ mangle: false }))
        .pipe(gp_sourcemaps.write('/'))
        .pipe(gulp.dest(destPaths.app));
});

// Delete wwwroot/app contents
gulp.task('app_clean', function () {
    return gulp.src(destPaths.app + "*.*", { read: false })
    .pipe(gp_clean({ force: true }));
});

// Copy all JS files from external libraries to wwwroot/js
gulp.task('js', function () {
    gulp.src(srcPaths.js_angular)
        .pipe(gulp.dest(destPaths.js_angular));
    gulp.src(srcPaths.js_rxjs)
        .pipe(gulp.dest(destPaths.js_rxjs));
    return gulp.src(srcPaths.js)
        .pipe(gulp.dest(destPaths.js));
});

// Delete wwwroot/js contents
gulp.task('js_clean', function () {
    return gulp.src(destPaths.js + "*.*", { read: false })
    .pipe(gp_clean({ force: true }));
});

// Process all LESS files and output the resulting CSS in
wwwroot/css

gulp.task('less', ['less_clean'], function () {
```

```
    return gulp.src(srcPaths.less)

        .pipe(gp_less())

        .pipe(gulp.dest(destPaths.css));

});


// Delete wwwroot/css contents

gulp.task('less_clean', function () {

    return gulp.src(destPaths.css + "*.*", { read: false })

    .pipe(gp_clean({ force: true }));

});
// Watch specified files and define what to do upon file changes
gulp.task('watch', function () {
    gulp.watch([
        srcPaths.app,
        srcPaths.js,
        srcPaths.less],
        ['app', 'js', 'less']);
});

// Global cleanup task
gulp.task('cleanup', ['app_clean', 'js_clean', 'less_clean']);

// Define the default task so it will launch all other tasks
gulp.task('default', ['app', 'js', 'less', 'watch']);
```

What we did here was quite straightforward:

- At the beginning of the file, we added a reference to the `gulp-less` plugin, assigning it to the `gp_less` variable just like we did with the other plugins.
- We added the `less` key to the `srcPaths` variable; that key contains a listing of the locations of all the `.less` files we want to process: everything inside `Scripts/less/`, including subfolders, as long as it ends with the `.less` extension.
- We also added the `css` key to the `destPaths` variable, defining the folder where the resulting compiled `.css` files will be generated.
- Then we added two new tasks to instruct `Gulp` how to handle LESS files:
    - The `less` task to compile the `.less` files found inside the `srcPaths.less` folder(s) and then output the result to the `wwwroot/css` target folder.
    - The `less_clean` task to erase the contents of the target folder; since it does the same job as the other existing cleanup tasks, there's no need to

explain how it works. We also added it as a dependent task to the previous one, just like we did with the `app_clean` and `app` tasks back in [Chapter 1](#), *Getting Ready*.

As soon as we perform these changes and save the file, go to the **Task Runner Explorer** panel in Visual Studio 2015. Once there, stop the `Gulpdefault` task, then re-run it again: a new `/wwwroot/css/` folder should be created shortly, together with a freshly-compiled `style.css` file inside. Open it and ensure it has the same content placed into the `/Scripts/less/style.less` file, then proceed to the next part.

Linking the stylesheet

Open the `/wwwroot/index.html` file and add the following line within the `<head>` block, right below the `<meta>` element:

```
<!-- Stylesheets -->
<link rel="stylesheet" type="text/css" href="/css/style.css"
media="screen" />
```

Testing it up

Before going further, it's advisable to run a quick check in order to ensure that everything we made so far is working as it should.

Open the `/Scripts/app/home.component.ts` file and remove everything within the `styles` block so that it will look like the following:

```
styles: []
```

Alternatively, we could entirely remove it, as we're not going to use it anymore.

Once done, run the application in **Debug** mode by hitting *F5* and check if the `item-list` components still have their CSS styles applied to them:
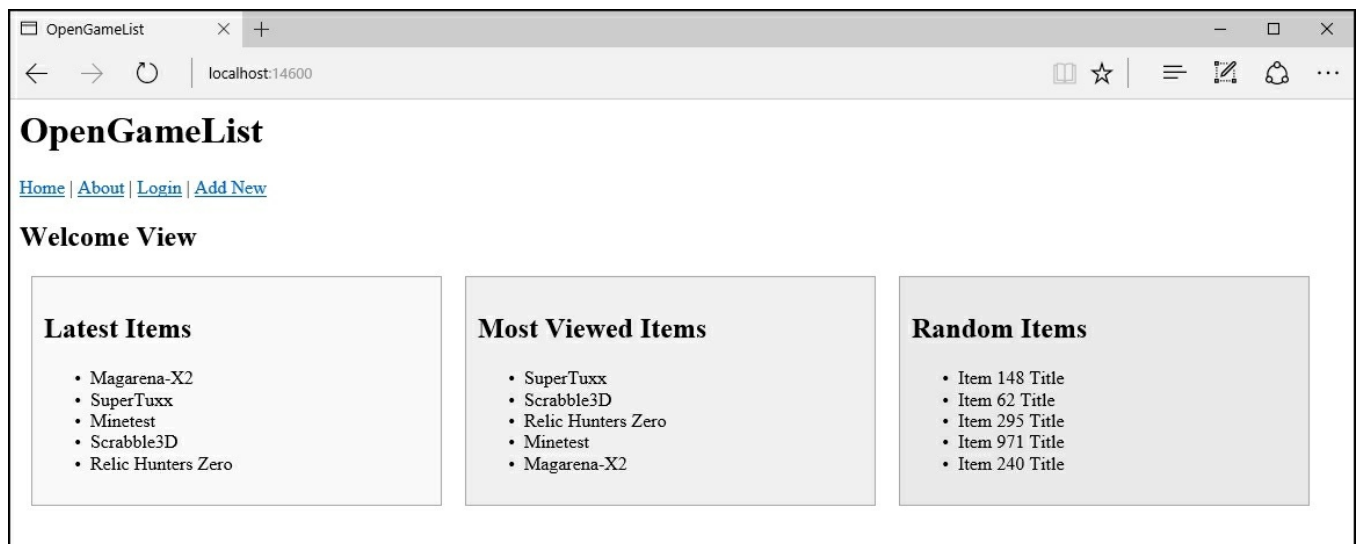
They definitely should, since we placed them on the `style.less` file that, in its `.css` compiled form, should now be linked to our application.

DIY versus framework-based styling

Now that we have added a fully-functional `style.less` stylesheet file, we could also move all our custom styles there, just like we did with the `home.component.ts` file; the question is, are they really worth that much? Hardly, considering that we put them together for demonstration purpose only, a *quick'n'dirty*, temporary solution until we could commit ourselves to the upcoming styling phase. Now that the time has come, we shouldn't bother too much about preserving these samples, we need to think about switching from a full *do-it-yourself* approach to a framework-based one.

Anyone who is into CSS design is well aware of such a debate, which we could summarize in the following single phrase:

> *Should we build our own grid-based responsive layout or use a responsive design framework instead?*

The answer is not that simple because either alternative has their set of advantages. Let's try to perform a quick recap of the most relevant arguments.

Do it yourself approach

The most classic approach: we build our very own grid-based layout, featuring a custom set of resizing raster and/or vector set of images and icon files, following the responsive design good practices and guidelines as issued by the famous Ethan Marcotte 2010 article published in the *A List Apart* blog: http://alistapart.com/article/responsive-web-design

And also follow the subsequent, improved theories and patterns described in his following brief book (*Responsive Web Design*, *A Book Apart*, 2011).

Pros

- Faster loading times, as we will be able to only code, add, and/or include what we need.
- Unique design (all framework-based websites are supposed to look the same).

Cons

- Can be quite hard to handle, unless we're true CSS3/MediaQuery experts.
- **Slow development**, due to the massive amount of required tests for all the existing platforms (browsers, operating systems, mobile devices).
- **Hard to keep it updated** to the latest standards since there will be no one that will bother to test or improve that code other than us.

Framework-based approach

The common approach nowadays: we start from a consolidated, widely-accepted UI frontend framework such as Bootstrap, Foundation, Pure, or YUI and customize it to suit our needs. The word customizing can mean a number of things here, from picking a skin to completely changing the structural behavior of most classes, depending on how much we want to customize the results and/or how much time we are allowed to spend doing that.

Pros

- **Development speed**: These frameworks are a time-saver, as we will be able to use reliable, cross-browser compatible code blocks instead of coding everything from scratch.
- **Consistency**: One of the biggest frontend framework achievements is that they make designers and developers speak the same language, as they will both be able to acknowledge, understand, and apply their changes to the project in a consistent way.
- **Community support**: Each framework has a huge support community, meaning that we'll receive free code samples, support, and updates for as long as we need to. This can be huge, especially if we want to achieve good results without having to commit too much into cross-browser, responsive CSS design.

Cons

- **Limited knowledge**: We didn't write that code, so we won't always be able to understand what we're doing and *why* the stuff we're using behaves like that.
- **Performance heavy**: Even the most *lightweight* and modular framework will undoubtedly be packed with a lot of stuff we won't be using in our project: these contents will be sent by the web server and loaded by the client anyway.
- **Updating issues**: Whenever an improved build of the framework is out, we will have to choose between updating it, and taking the risk of breaking something, and not updating it, and risk losing the bug fixes and the added/improved support for the new CSS standards.

Conclusions

As we can see, both ways could be viable depending on our specific scenario; therefore, the decision between going with a custom grid and adopting a framework-based one should be made on a case-by-case basis.

That said, after our non-exhaustive analysis, we think that adopting a frontend framework might be a good call for our project; we'll also be picking Bootstrap for the task, since it happens to be one of the most suited ones for native web applications based on Angular 2, as we'll be able to see in the following paragraphs.

Note

It's worth noting that by choosing Bootstrap we're ditching a great alternative that will surely pave its way in the upcoming months: we're talking about `material2`, a top-notch component library based upon material design.

The only reason we didn't pick it is that the project is still in alpha, yet there are already a couple of demos out that demonstrate its striking potential. If we're bold enough to try that, we can ditch this chapter entirely and learn our own way by looking at the official project page at the following URL:

https://material.angular.io/ Or look at their source code repository on GitHub: https://github.com/angular/material2

Adding Bootstrap

Installing Bootstrap into our project is just as easy as adding the following line within the `<head>` block of our `wwwroot/index.html` file:

```
<link
href="https://maxcdn.bootstrapcdn.com/bootswatch/3.3.6/yeti/bootstr
ap.min.css" rel="stylesheet" integrity="sha384-
yxFy3Tt84CcGRj9UI7RA25hoUMpUPoFzcdPtK3hBdNgEGnh9FdKgMVM+lbAZTKN2"
crossorigin="anonymous" />
```

Tip

It's advisable to place it right above the `style.css` link, so we'll be able to override the Bootstrap default rules with our own style sheet file.

As we can see, we'll be using the Bootstrap 3 Yeti Theme (actually the 3.3.6 build, the latest stable one at the time of writing), by linking a pre-built, minified `.css` file hosted by MaxCDN, which is the primary **content delivery network** (**CDN**) for Bootstrap and Bootswatch.

Tip

Using a CDN-hosted link for frontend frameworks is often a good practice and can be very advisable in most cases, as it will remove some weight from our web server for a relevant server-side performance benefit: however, it will also have some relevant downsides, such as preventing us from modifying the source CSS (or LESS) file(s). If we feel like we'll be needing that, and/or if we don't want a CDN for other good reasons, we're also free to download the file locally and change the link so it will point to the project filesystem instead. The choice, as always, is entirely up to the developer.

We're choosing to stick with the v3 because Bootstrap 4, although almost ready, is currently still in a beta release. We want to adopt a stable, reliable, and widely-tested framework, we won't push things there, so we'll stick to the latest final instead.

Note

We chose the Yeti theme because it features a minimalistic, yet suitable style that will nicely blend with our project; if you don't like it, you're free to choose another one from the Bootswatch project's page:

http://bootswatch.com/ And here's a link to their primary CDN:
https://www.bootstrapcdn.com/bootswatch/

Choosing a native directives module

If we've already worked with Bootstrap, we know that placing a link to the `.css` file isn't going to be enough: if we want to use transitions, modals, tabs, date/time pickers, and a lot of advanced yet very useful components, we'll also have to add a reference to the plugin's JavaScript code. In standard web projects, it's very common to do that by adding some `<script>` references to the HTML page. We could include each plugin individually, using their own Bootstrap `*.js` file, or all of them at once with a single reference to the pre-compiled `bootstrap.js` or `bootstrap.min.js` file, also available through various CDN repositories (including *MaxCDN*). In either case, we'll also need to add a reference to the *jQuery* JavaScript library, since all these plugins depend on jQuery.

However, since we're using Angular 2, we won't be following that route. We don't need to add jQuery, `bootstrap.js` or any plugin's individual `.js` file since we can install one of the available native directives modules specifically designed for implementing Bootstrap components.

Before going any further, we should spend a bit of time addressing a very reasonable question: why should we do that, instead of sticking to the good old `jquery.js` plus `bootstrap.js` path?

The reason is quite simple, although it might not be easy to understand for those with no Angular experience: we could say that it is strongly related to the unwritten yet very important rule of writing expressive code, preventing ourselves from doing any direct DOM manipulation unless it's unavoidable.

In Angular 2, there are few circumstances where directly manipulating the DOM is really necessary. The framework provides a native set of powerful, high-level APIs that can be used instead. Understanding, using, and leveraging these APIs is the best thing a developer can do to write successful applications because of the following distinctive advantages:

- **Unit testing**: Manipulating the DOM adds browser dependencies and also a level of complexity. Both of them will eventually lead to weaker and less consistent tests.
- **Decoupling**: Removing DOM dependencies will also allow our application to run outside of a browser, for example, in Node.js or any Node.js yet non-web-based environment, such as Electron.
- **Readability**: Using Angular 2's template syntax instead of relying on Bootstrap's attribute-based API or programmatic, JavaScript-based API will eventually lead to smaller, more readable code.

- **Maintainability**: Relying on Angular 2, jQuery, and the Bootstrap plugin script file(s) within the same project can be tricky and hard to maintain. The overhead will also be quite relevant in terms of both size and performance since there is an inevitable amount of repeated stuff in there.

Now that we've cleared our mind, we can choose the Angular 2 native directive modules we'll be using among the available ones. At the time of writing, the Angular 2 and Bootstrap communities seem to have put their hopes into these three very promising GitHub projects:

- https://github.com/valor-software/ng2-bootstrap
- https://github.com/ng-bootstrap/core
- https://github.com/mlaval/angular2-bootstrap

Among these, the first one, formerly `ng2-bootstrap`, seems to have the lead in terms of features and contributors. It's also the only one that comes with an extensive support of both Bootstrap 3 and 4, which is a very relevant thing for us since we'll be using Bootstrap 3.

Installing ng2-bootstrap

The easiest way to install `ng2-bootstrap` is using NPM. However, it also has a dependency on the `moment` JavaScript library, so we're going to install that as well. As usual, we can do that by adding the following lines to the `dependencies` section of our `package.json` file:

```
"moment": "^2.14.1",
"ng2-bootstrap": "^1.0.24",
```

As soon as we save the file, Visual Studio will fetch these NPM packages.

Right after that, we can add a reference to these new libraries to the `srcPaths.js` array variable declared in our `gulpfile.js` file, as we can see in the following excerpt (new lines have been highlighted):

```
var srcPaths = {
    app: ['Scripts/app/main.ts', 'Scripts/app/**/*.ts'],
    js: [
        'Scripts/js/**/*.js',
        'node_modules/core-js/client/shim.min.js',
        'node_modules/zone.js/dist/zone.js',
        'node_modules/reflect-metadata/Reflect.js',
        'node_modules/systemjs/dist/system.src.js',
        'node_modules/typescript/lib/typescript.js',
        'node_modules/ng2-bootstrap/bundles/ng2-bootstrap.min.js',

        'node_modules/moment/moment.js'
    ],
    js_angular: [
        'node_modules/@angular/**'
    ],
    js_rxjs: [
        'node_modules/rxjs/**'
    ],
    less: [
        'Scripts/less/**/*.less'
    ]
};
```

Last but not least, we need to add the `<script>` reference tags within our `wwwroot/index.html` page, right after the other Angular 2 library files:

```
<!-- Step 1. Load libraries -->
<!-- Polyfill(s) for older browsers -->
<script src="js/shim.min.js"></script>
<script src="js/zone.js"></script>
<script src="js/Reflect.js"></script>
<script src="js/system.src.js"></script>
```

```
<!-- Angular2 Native Directives -->
<script src="/js/moment.js"></script>
<script src="/js/ng2-bootstrap.min.js"></script>
```

That's about it.

Applying styles

We're ready to apply some styles to our project. Let's warm ourselves up with some plain Bootstrap class-based styling: while we're there, we'll also add some custom CSS classes and drop a couple of images here and there to make our native web app look a bit more distinctive.

Main navigation bar

Open the `Scripts/app.component.ts` file and, within the `@Component` section, replace the whole `template` with the following content:

```
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container-fluid">
        <input type="checkbox" id="navbar-toggle-cbox">
        <div class="navbar-header">
            <label for="navbar-toggle-cbox" class="navbar-toggle
collapsed" data-toggle="collapse" data-target="#navbar" aria-
expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </label>
            <a class="navbar-brand" href="#">
                <img alt="logo" src="/img/logo.svg" />
            </a>
        </div>
        <div class="collapse navbar-collapse" id="navbar">
            <ul class="nav navbar-nav">
                <li><a class="home" [routerLink]="['']">Home</a>
</li>
                <li><a class="about" [routerLink]="
['about']">About</a></li>
                <li><a class="login" [routerLink]="
['login']">Login</a></li>
                <li><a class="add" [routerLink]="['item/edit',
0]">Add New</a></li>
            </ul>
        </div>
    </div>
</nav>
<h1 class="header">{{title}}</h1>
<div class="main-container">
    <router-outlet></router-outlet>
</div>
```

What we did here was nothing special. We added a couple of Bootstrap classes to our previous `<ul>` menu element and then wrapped it in a more accessible (and also Bootstrap-styled) `<nav>` element. We also introduced two custom CSS classes: `header` for the `<h1>` element and `main-container` for a brand new `<div>` element we used to wrap the `<router-outlet>` container. Finally, we added an image reference pointing to a `logo.svg` vector image file, so our navigation menu bar will have something distinctive. Needless to say, before being able to see how this restyling looks, we need to bring these custom items to life.

Let's start with the CSS classes: open the `/Scripts/less/style.less` file and append the following code:

```less
h1.header {
    font-size: 40px;
    margin: 46px 0 0 0;
    padding: 20px 20px 20px 112px;
    background: url(/img/header-bg.png) no-repeat left top #efefef;
    text-align: left;
}

.navbar {
    margin-bottom: 0;
    img {
        width: 32px;
        height: 32px;
        margin: -4px 0 0 0;
        padding: 0;
    }

    #navbar-toggle-cbox {
        display:none;
        &:checked ~ .collapse {
            display: block;
        }
    }
}
```

We added another image reference here, a raster `png` that will serve as a background for our `<h1 class="header">` file. That's another file we need to add to our project. In order to do that, create a new `/img/` directory within the `/wwwroot/` root folder and place the `logo.svg` and the `header-bg.png` file there.

Tip

This isn't a book about web image design, so we won't cover the image creation process. We'll just add some sample minimalistic images not too different from what we could be using in a real-case scenario. They can be found online at the following URL: http://www.opengamelist.com/img/ Just add the file name to that base URL to view it online and/or download it; you can use this method for every image used within this chapter. If you need a sample favicon, you can find the one we're using here: http://www.opengamelist.com/favicon.ico Alternatively, you can design your own ones or get some from one of the royalty-free image and icon libraries available online.

Before going any further, it's time to run a quick test and see if everything is working properly. As usual, all we need to do is hit *F5* and take a look:
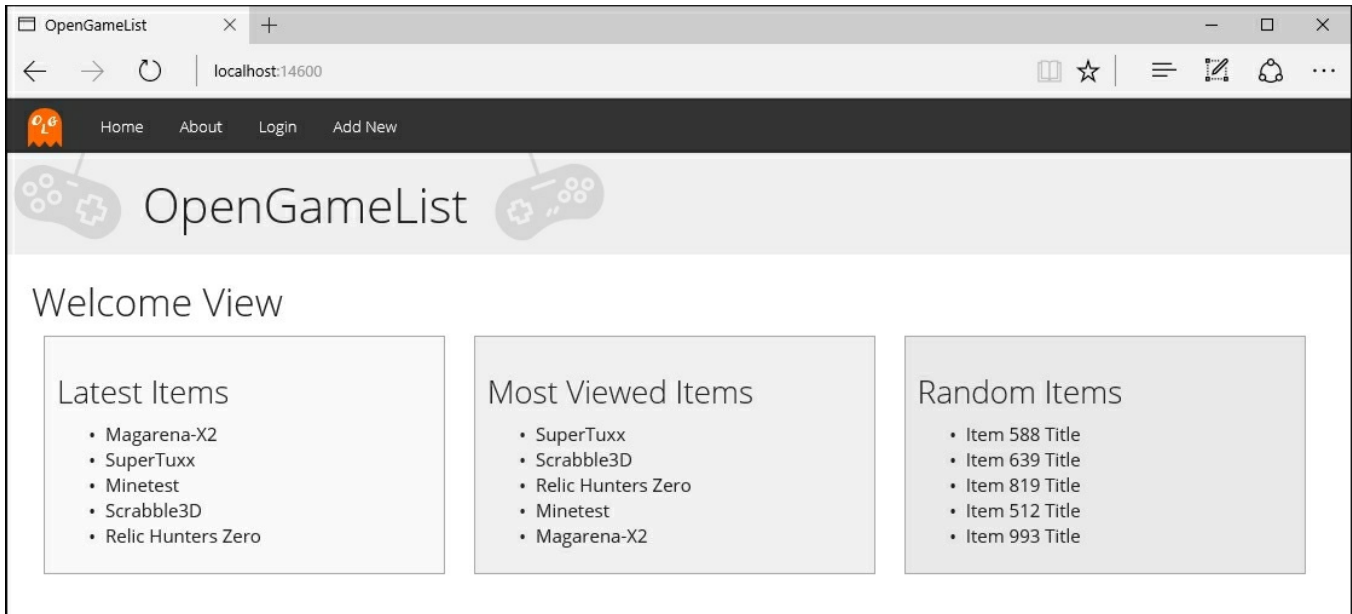
Not that bad, considering where we came from. Nonetheless, we can already see a minor flaw: the navigation bar doesn't tell us where we are, as the active route is not highlighted in any way.

If we inspect the Angular 2-generated HTML code with a DOM inspector, we can see that there actually is something that could help us to identify the active item: it's the `router-link-active` CSS class, which Angular 2 assigns automatically to the anchor who activated the route. Unfortunately, that class name cannot be changed and Bootstrap doesn't support it, as it's expecting the `active` class to be used instead, not to mention the fact that it also wants the class to be applied to the parent `<li>` element.

Taking these facts into account, we have the following options to achieve what we want:

- Completely ditch the Bootstrap standards, together with their `.active` class, and define our own `.router-link-active` class instead with full custom CSS code, logic and behavior.
- Use a CSS3 parent selector such as `li < a.router-link-active` and write our custom CSS code there. That rule will allow us to preserve some of the Bootstrap logic, as we would style the `<li>` element, but we would still be forced to write a lot of custom CSS code.
- Assign the Bootstrap class using JavaScript. We're joking, of course. There's no way we would do that, not after all we said regarding avoiding DOM manipulation earlier.
- Detect the active route and act accordingly using Angular 2 native components.

There's little doubt about which option we should choose.

Detecting the active route

A rather easy way to accomplish our task is using the Angular 2 **Router** class, which is part of the component router we introduced in [Chapter 3](#), *Angular 2 Components and Client-Side Routing*. That class features a convenient `isActive()` method that seems to be perfect for our needs. If we look at the official Angular 2 API documentation at angular.io/docs/ we can read the following:

"Returns if the url is activated or not."

Tip

Source: `https://angular.io/docs/ts/latest/api/router/index/Router-class.html#!#isActive-anchor`.

The method accepts either a URL `string` or a `UrlTree` object, meaning that we need to generate one of them to use it. We can obtain the latter using the `router.createUrlTree` method, both provided by that same class.

To cut it short, we just need to add the `Router` class to our Angular 2 component's constructor, using dependency injection, just as we have done a number of times already, and use the aforementioned methods to understand the currently active route. Once we do that, we can easily add the `active` CSS class to the corresponding `<li>` element using the template syntax class bindings we've already used.

Let's put everything together. Open the `app.component.ts` file again and update it as follows (new/modified code has been highlighted):

```
import {Component} from "@angular/core";
import {Router} from "@angular/router";

@Component({
    selector: "opengamelist",
    template: `
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container-fluid">
        <input type="checkbox" id="navbar-toggle-cbox">
        <div class="navbar-header">
            <label for="navbar-toggle-cbox" class="navbar-toggle
collapsed" data-toggle="collapse" data-target="#navbar" aria-
expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
```

```html
                    <span class="icon-bar"></span>
                </label>
                <a class="navbar-brand" href="#">
                    <img alt="logo" src="/img/logo.svg" />
                </a>
            </div>
            <div class="collapse navbar-collapse" id="navbar">
                <ul class="nav navbar-nav">
                    <li [class.active]="isActive([''])">

                        <a class="home" [routerLink]="['']">Home</a>

                    </li>

                    <li [class.active]="isActive(['about'])">

                        <a class="about" [routerLink]="
['about']">About</a>

                    </li>

                    <li [class.active]="isActive(['login'])">

                        <a class="login" [routerLink]="
['login']">Login</a>

                    </li>

                    <li [class.active]="isActive(['item/edit', 0])">

                        <a class="add" [routerLink]="['item/edit',
0]">Add New</a>

                    </li>
                </ul>
            </div>
        </div>
</nav>
<h1 class="header">{{title}}</h1>
<div class="main-container">
    <router-outlet></router-outlet>
</div>
    `
})

export class AppComponent {
    title = "OpenGameList";

    constructor(public router: Router) { }

    isActive(data: any[]): boolean {
```

```
        return this.router.isActive(

            this.router.createUrlTree(data),

            true);


    }
}
```

Let's see what we did here:

- At the top of the file, we added the `Router` class from the `@angular2/router` package to the `import` list.
- At the bottom of the file, we added the constructor method with a router object instantiated through dependency injection. There's nothing new here, we already did that in [Chapter 3](#), *Angular 2 Components and Client-Side Routing*.
- Right below the `constructor`, we added the `isActive()` internal helper method to avoid repeating some required lines of code more than once. The method accepts a required `data` variable that will be used to generate the required `UrlTree` object using the `router.createUrlTree` method and pass it to the `router.isActive` method. The latter will then return `true` if the given `UrlTree` matches the active route, and `false` otherwise.
- In the `@Componenttemplate` section, we use the `isActive()` internal method result to determine which `<li>` element should have the `active` CSS class bound to itself using template syntax.

Before testing it, we might find it useful to customize the background color of the `active` CSS class to make it more visible than the "black over dark grey" effect featured by our Yeti Bootstrap theme.

Open the `/Scripts/less/style.less` file and append the following:

```
.navbar-default .navbar-nav>.active>a,
.navbar-default .navbar-nav>.active>a:hover,
.navbar-default .navbar-nav>.active>a:focus {
    background-color: #863500;
}
```

Now we can run another quick test to see if the results match the expectations. Hit *F5* again and wait until we see the following:
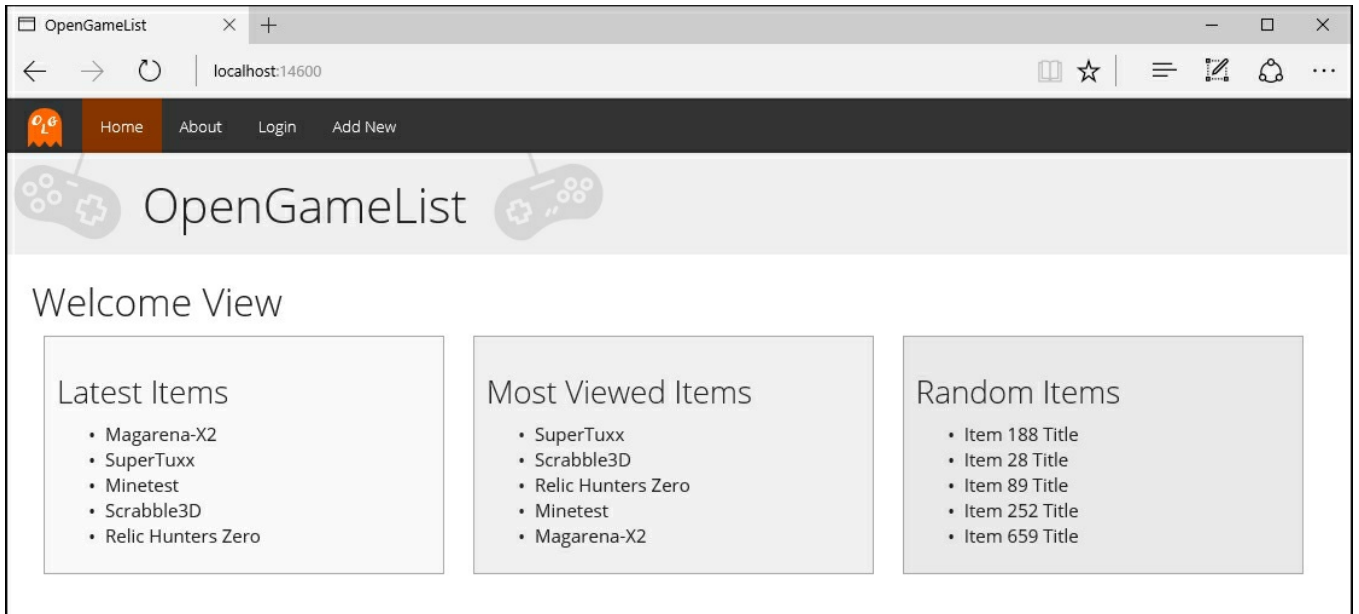
Click on the **About**, **Login**, and **Add New** navigation bar links to see the dark orange background following and highlighting the active route. From now on, our valued users will always know where they are.

Welcome View

It's time to give our **Welcome View** a new and (hopefully) better look.

The `app.component.ts` file is fine as it is, so we won't be touching it again for a while. We'll open the `home.component.ts` instead, and replace the existing template in the following way (new/updated code has been highlighted):

```
<h2>
    A non-comprehensive directory of open-source video games

    available on the web

</h2>

<div class="col-md-4">
    <item-list class="latest"></item-list>
</div>

<div class="col-md-4">
    <item-list class="most-viewed"></item-list>
</div>

<div class="col-md-4">
    <item-list class="random"></item-list>
</div>
```

What we did here was wrap our existing `<item-list>` elements in a Bootstrap grid system made of three `<div>` that will fill the viewport width. We also replaced the `<h2>` text content with a viable payoff for our **Welcome View**, replacing the previous working title.

Tip

For more information about the Bootstrap grid system you can check the official documentation at the following URL:

http://getbootstrap.com/css/#grid

We already removed the `styles` within this file's `@Component` section, so we're done here.

Let's move to the `item-list.component.ts` file. Remove the `styles` section as well, then modify its `template` section in the following way (new/updated code has been highlighted):

```html
<h3>{{title}}</h3>

<ul class="items">
    <li *ngFor="let item of items"
        [class.selected]="item === selectedItem"
        (click)="onSelect(item)">
        <div class="title">{{item.Title}}</div>

        <div class="description">{{item.Description}}</div>
    </li>
</ul>
```

Again, we did nothing special here, we just added the `item.Description` to the template right below the already existing `item.Title` and wrapped them both within some `<div>` elements so we can style them.

Now that we have set up the templates, we can open the `/Scripts/less/style.less` file and create some CSS classes. Here's how the revamped file will look:

```less
// Some Variables that will be used below
@color-latest: #5a4d74;
@color-most-viewed: #4d6552;
@color-random: #703535;

// Header styling
h1.header {
    font-size: 40px;
    margin: 46px 0 0 0;
    padding: 20px 20px 20px 112px;
    background: url(/img/header-bg.png) no-repeat left top #efefef;
    text-align: left;
}

// Navbar styling
.navbar {
    margin-bottom: 0;
    img {
        width: 32px;
        height: 32px;
        margin: -4px 0 0 0;
        padding: 0;
    }

    // Expand/collapse the navbar in mobile-friendly mode using
pure CSS styling.
    // ref.: http://stackoverflow.com/a/31506685/1233379
    #navbar-toggle-cbox {
        display:none;
```

```less
            &:checked ~ .collapse {
                display: block;
            }
        }
    }

    // Improve the visibility of the active navbar item (currently
    active route)
    .navbar-default .navbar-nav>.active>a,
    .navbar-default .navbar-nav>.active>a:hover,
    .navbar-default .navbar-nav>.active>a:focus {
        background-color: #863500;
    }

    h2 {
        margin: 20px;
        padding: 0;
        font-size: 1.4em;
        line-height: 1.4em;
        font-style: italic;
        color: #666666;
    }

    // item-list component(s) styling
    item-list {
        border: 0;
        margin: 0;
        padding: 0;
        vertical-align: top;
        display: block;
        &.latest {
            background-color: #f6f6f6;
            h3 {
                background-color: @color-latest;
                background-image: url(/img/latest-icon.png);
            }
        }
        &.most-viewed {
            background-color: #f0f0f0;
            h3 {
                background-color: @color-most-viewed;
                background-image: url(/img/most-viewed-icon.png);
            }
        }
        &.random {
            background-color: #e9e9e9;
            h3 {
                background-color: @color-random;
                background-image: url(/img/random-icon.png);
            }
        }
        h3 {
```

```
        color: #fefefe;
        margin: 0;
        padding: 10px 15px;
        background-repeat: no-repeat;
        background-position: 97% center;
    }
    ul {
        list-style-type: none;
        padding: 15px;
        li {
            cursor: pointer;
            margin-top: 20px;
            padding-top: 15px;
            border-top: 1px solid #c7c7c7;
            &:first-child {
                margin-top: 0;
                padding-top: 0;
                border-top-width: 0;
            }
            .title {
                font-size: 1.3em;
                font-weight: bold;
            }
            .description {
                margin-top: 3px;
                font-size: 15px;
                line-height: 1.5em;
                height: (15*1.5*2);
                overflow: hidden;
                display: -webkit-box;
                -webkit-line-clamp: 2;
                -webkit-box-orient: vertical;
            }
        }
    }
}
```

As we can see, there's quite a lot of new stuff here. For the sake of readability, we also added some comments in order to explain the role of each CSS class. Let's focus on the most important things:

- We added a unique color and also an icon background image for each of our `<item-list>` component headers.
- We applied some styling to the `<item-list>` element and their children, up to the item's `Text` and `Description` containers.
- We improved the readability of the active `<nav>` element items and we also did what it takes to make the navigation menu work properly in its collapsed, mobile-friendly mode.

What we did was nothing more than a styling sample to give our **Welcome View** a fresh look: while we were there, we also took the chance to show some of the most useful LESS features and capabilities.

We can see the results of our hard work by hitting *F5*:
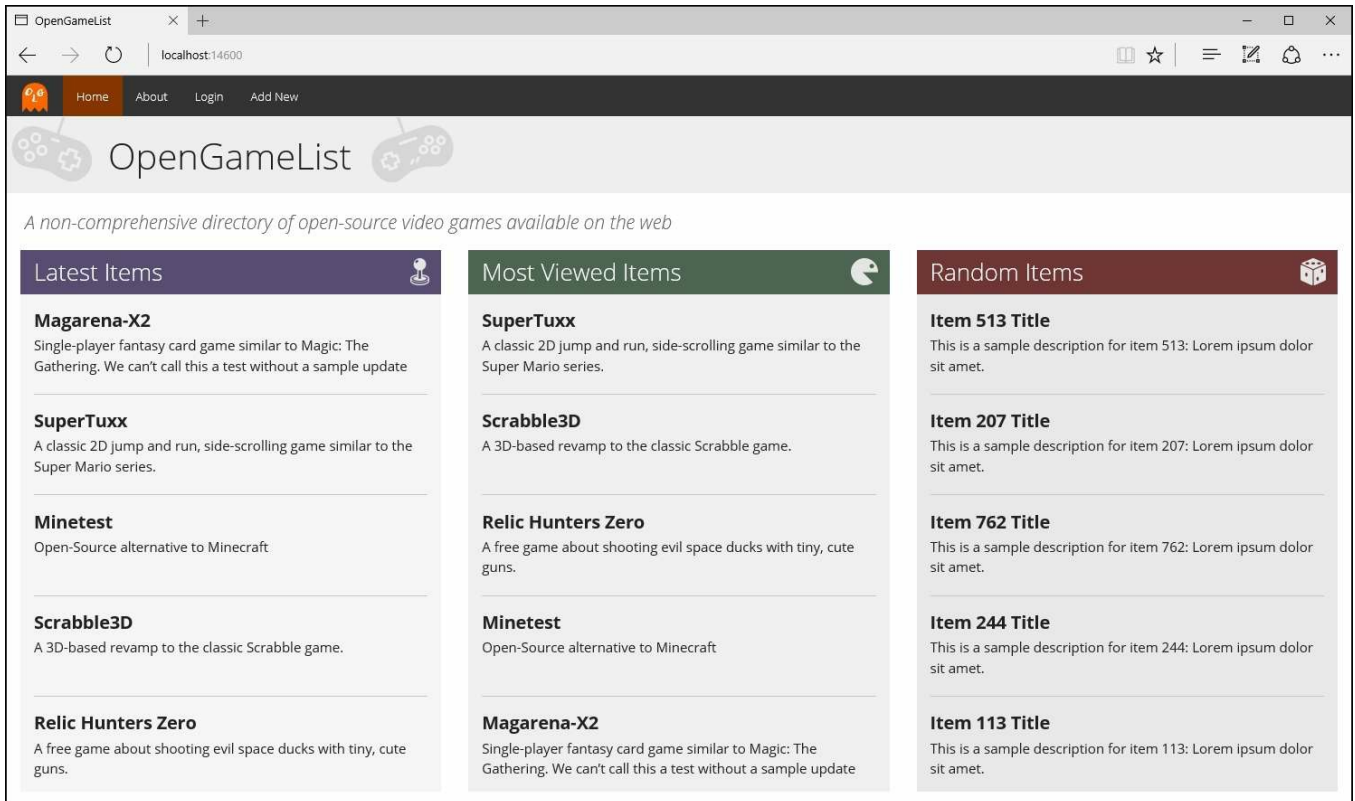
That's another significant improvement: our **Welcome View** is now looking quite good and, thanks to the Bootstrap grid system has gained some mobile-friendliness as well. Here's how it would look on an Apple iPhone 6:

OpenGameList

A non-comprehensive directory of open-source video games available on the web

Latest Items

Magarena-X2
Single-player fantasy card game similar to Magic: The Gathering. We can't call this a test without a sample update attempt. The previous

SuperTuxx
A classic 2D jump and run, side-scrolling game similar to the Super Mario series.

Minetest
Open-Source alternative to Minecraft

Scrabble3D
A 3D-based revamp to the classic Scrabble game.

Relic Hunters Zero
A free game about shooting evil space ducks with tiny, cute guns.

Most Viewed Items

SuperTuxx
A classic 2D jump and run, side-scrolling game similar to the Super Mario series.

Not bad at all, is it? Let's try to keep that pace.

Item Detail View

When we split our **Item Detail View** into two distinctive components (**View** mode and **Edit** mode), we also gave them a minimalistic, wiki-like tabbed interface. Now that we have Bootstrap we can further improve that approach by redesigning it into a more intuitive, visually engaging view-editor interface.

Display mode

Open the `/Scripts/app/item-detail-view.component.ts` and remove the `@Component`'s `styles` section entirely. We won't need it anymore, as we'll be using our `style.less` file from now on.

Right after that, replace the template section HTML content with the following code:

```
<div *ngIf="item">
    <h2>
        <a href="#" (click)="onBack()">&laquo; Back to Home</a>
    </h2>
    <div class="item-container">
        <ul class="nav nav-tabs">
            <li role="presentation">
                <a href="#"
(click)="onItemDetailEdit(item)">Edit</a>
            </li>
            <li role="presentation" class="active">
                <a href="#">View</a>
            </li>
        </ul>
        <div class="panel panel-default">
            <div class="panel-body">
                <div class="item-image-panel">
                    <img src="/img/item-image-sample.png" alt="
{{item.Title}}" />
                    <div class="caption">Sample image with caption.
</div>
                </div>
                <h3>{{item.Title}}</h3>
                <p>{{item.Description}}</p>
                <p>{{item.Text}}</p>
            </div>
        </div>
    </div>
</div>
```

There's a couple of things worth noting here:

- We added a second `<p>` element to display the `item.Text` property
- We don't support item image files yet, so we added a demo image to see how it

might look

- We added a reference to an `onBack()` method to allow the user to navigate back to the **Welcome View** without having to resort to the navigation menu

The aforementioned `onBack()` method isn't there yet, so we need to implement it within the component `class` code section in the following way (added lines have been highlighted):

```
onItemDetailEdit(item: Item) {
    this.router.navigate(["item/edit", item.Id]);
    return false;
}

onBack() {

    this.router.navigate(['']);

}
```

Once we're done with that, we can get back to our styling task. Open the `style.less` file and add the following code:

```
// Item Detail View & Edit components styling
@color-panel: #f9f9f9;

.item-container {
    margin: 0 20px;
    .nav.nav-tabs {
        padding-right: 20px;
        li {
            &.active a {
                font-weight: bold;
                background-color: @color-panel;
            }
            float:right;
        }
    }
    .panel.panel-default {
        background-color: @color-panel;
        border-top-width: 0;
        .item-image-panel {
            width: 170px;
            min-height: 170px;
            float: right;
            padding: 10px;
            background-color: #f2f2f2;
            border: 1px solid #cccccc;
            img {
                width: 150px;
                height: 150px;
```

```
        border: 1px solid #e6e6e6;
    }
    .caption {
        margin-top: 5px;
        text-align: center;
        font-size: 0.8em;
    }
    }
    h3 {
        margin: 10px 0 20px 0;
        font-weight: bold;
    }
    }
}
```

Again, we're just applying some custom spacing, coloring, and aligning here and there, leaving all the rest to the standard Bootstrap `nav` and `panel` classes.

Here's our improved **Item Detail View** in **Display** mode:
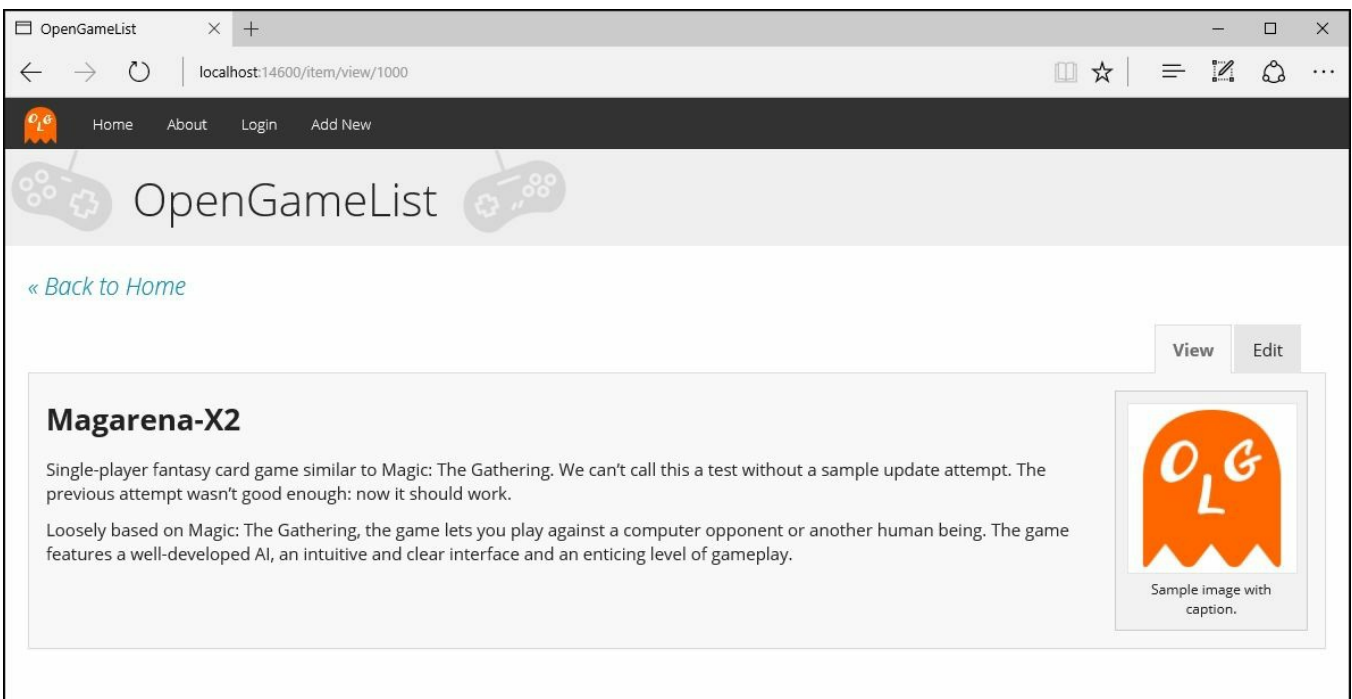
The mobile-friendly viewport mode is looking good as well:
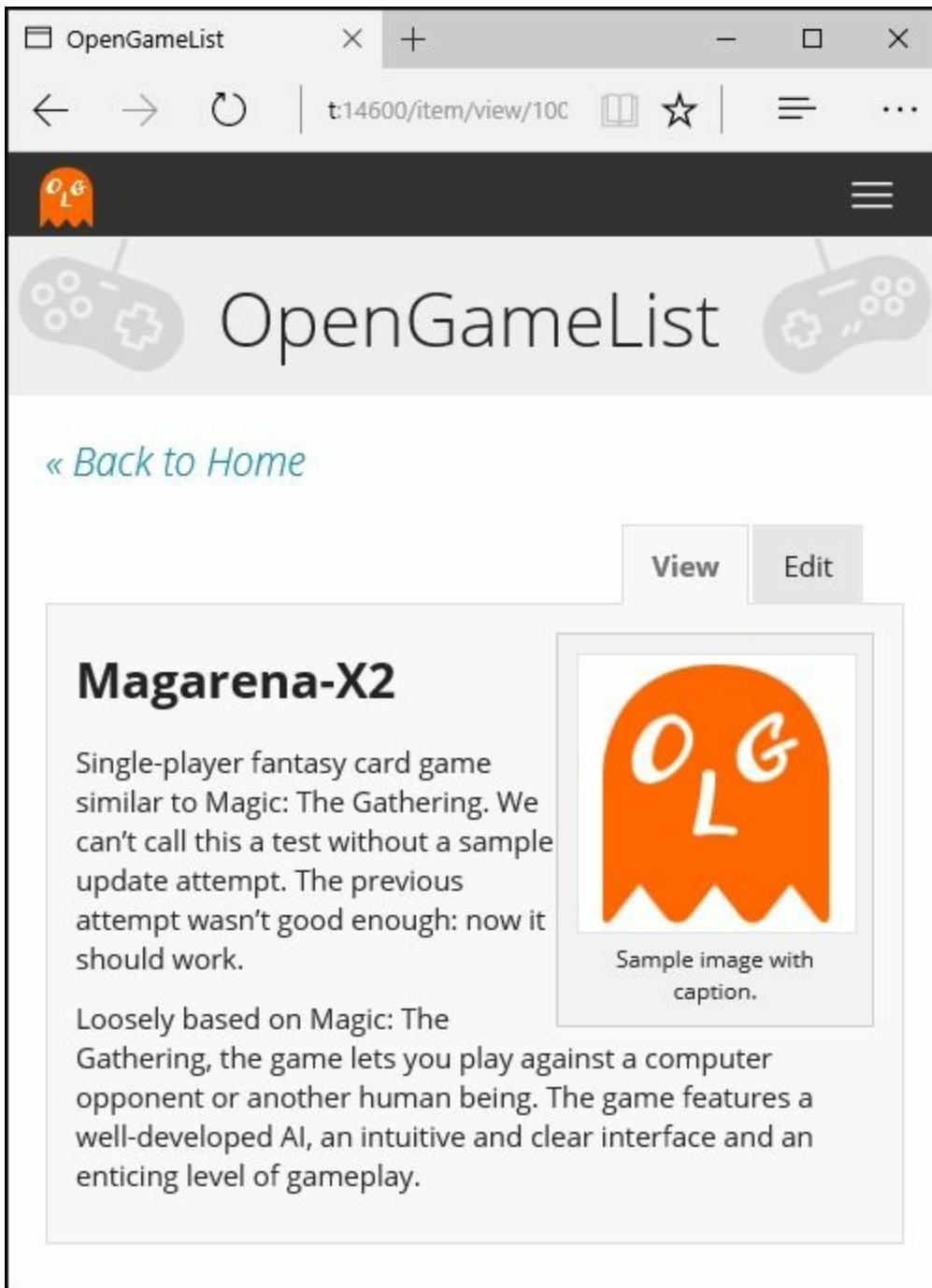
At this point, we can say that the **Display** mode is looking fine. Let's move to the other tab.

Edit mode

When `<form>` elements are involved, UI styling usually gets tricky, as we need to handle things such as form validation, required inputs, and other similar issues that will most likely have some sort of impact on the styling job. However, thanks to Bootstrap and Angular 2, it won't be that hard.

Again, let's start with opening the `/Scripts/app/item-detail-edit.component.ts` and remove the `styles` section of `@Component`.

Tip

To make things clear, we don't have anything against in-component styling. It's just that we wouldn't be able to use LESS syntax sugar while being in there, not to mention the fact that restraining ourselves from using that will also reduce the chance of CSS code repetition between different components.

Right after that, move to the `template` section and replace its contents with the following:

```
<div *ngIf="item">
    <h2>
        <a href="#" (click)="onBack()">
            &laquo; Back to Home
        </a>
    </h2>
    <div class="item-container">
        <ul class="nav nav-tabs">
            <li role="presentation" class="active">
                <a href="#">Edit</a>
            </li>
            <li role="presentation" *ngIf="item.Id != 0">
                <a href="#"
(click)="onItemDetailView(item)">View</a>
            </li>
        </ul>
        <div class="panel panel-default">
            <div class="panel-body">
                <form class="item-detail-edit">
                    <h3>{{item.Title}}</h3>
                    <div class="form-group">
                        <label for="input-title">Title</label>
                        <input id="input-title" name="input-title"
type="text" class="form-control" [(ngModel)]="item.Title"
placeholder="Insert the title..." />
                    </div>
                    <div class="form-group">
                        <label for="input-
description">Description</label>
                        <textarea id="input-description"
name="input-description" class="form-control"
[(ngModel)]="item.Description" placeholder="Insert a suitable
description..." required></textarea>
                    </div>
                    <div class="form-group">
                        <label for="input-text">Text</label>
                        <textarea id="input-text" name="input-text"
```

```
class="form-control" [(ngModel)]="item.Text" placeholder="Insert a
suitable description..."></textarea>
                        </div>
                        <div *ngIf="item.Id == 0" class="commands
insert">
                            <input type="button" class="btn btn-
primary" value="Save" (click)="onInsert(item)" />
                            <input type="button" class="btn btn-
default" value="Cancel" (click)="onBack()" />
                        </div>
                        <div *ngIf="item.Id != 0" class="commands
update">
                            <input type="button" class="btn btn-
primary" value="Update" (click)="onUpdate(item)" />
                            <input type="button" class="btn btn-danger"
value="Delete" (click)="onDelete(item)" />
                            <input type="button" class="btn btn-
default" value="Cancel" (click)="onItemDetailView(item)" />
                        </div>
                    </form>
                </div>
            </div>
        </div>
</div>
```

It seems like we've added a lot of new stuff here. Let's try to shed some light on what
we did:

- The first lines of code are almost identical to the **Display** mode template: we
  have the same `<h2>` pointing back to the **Welcome View** route and also an
  identical `<ul>` element with the `nav-tabs` Bootstrap CSS class to render the
  tabs.
- Inside the panel we've declared the main `<form>` element and added a series of
  `<div>` elements with the `form-group` class replacing our previous list-based
  structure. This is the default CSS class used by Bootstrap to handle the various
  input fields within a form. Inside each one of them, we placed the same
  `<input>` and `<textarea>` we defined before for the `Title` item and
  `Description`, plus a new one for the `Text`.
- We added some `<label>` elements with proper Bootstrap CSS styling. Each one
  of them is linked to their respective input field by using the `for` HTML attribute.
- By looking at the `<input>` elements we can see that we're still using the same
  two-way data binding, `ngModel`-based implementation logic that we put in
  place when we created the template for this component for the first time. There's
  no need to change that, since everything was already working well on that part.
  However, we added an `id` attribute to have them linked to their respective
  labels (as stated previously) and the `form-control` Bootstrap CSS class to
  style them.

Here are the custom classes to add to our `style.less` file. Instead of just appending them at the end of the file, let's place them inside the existing `.item-container` selector since they are only relevant within that scope:

```less
// Form styling for item-detail-edit component
form.item-detail-edit {
    .empty-field {
        color: #f04124;
    }
    .form-group {
        label {
            font-size: 14px;
            font-weight: bold;
            display: block;
            background-color: #eaeaea;
            line-height: 1.4em;
            padding: 5px 10px;
        }
        textarea {
            &#input-description {
                height: 80px;
            }
            &#input-text {
                height: 150px;
            }
        }
    }
    .commands {
        text-align: right;
        input[type="button"] {
            margin-left: 5px;
        }
    }
}
```

Here's how the revamped **Edit** mode will look like after these changes:
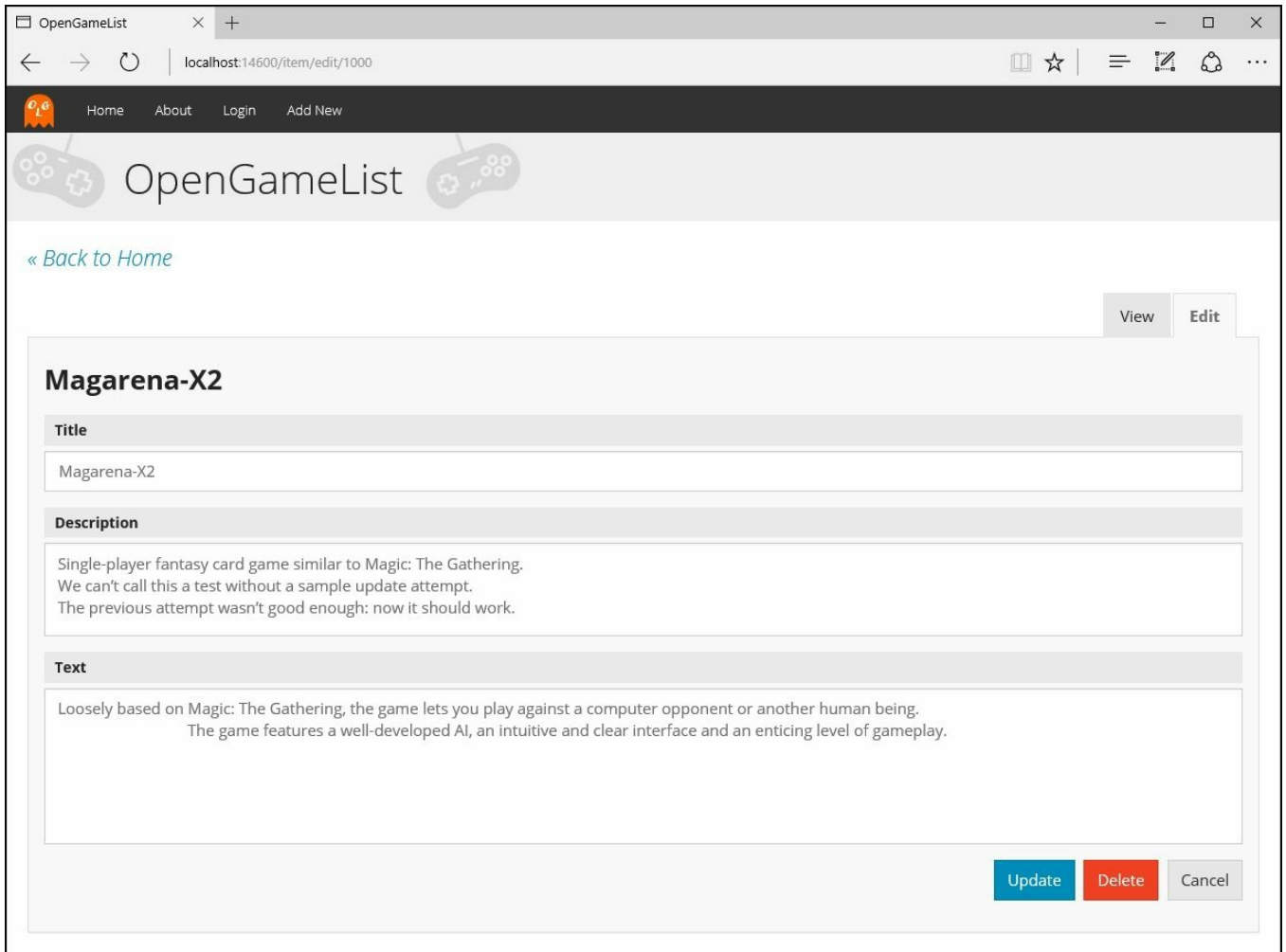
Not bad, but we can do better. We mentioned form validation and required items, yet we have done nothing about that yet. Let's see how we can do that with the item's `Title` property, which is definitely something that should never be empty or null in any given circumstance.

In order to do that, we need to go back to the `item-detail-edit.component.ts` file and apply the following changes.

In the `template` section, at the start of the existing `<form>` element, add the following code:

```
<h3>
    {{item.Title}}
    <span class="empty-field" [hidden]="dTitle.valid">

        Empty Title

    </span>
</h3>
```

Also in the `template` section, within the first `form-group` element, add the following code:

```
<div class="form-group has-feedback" [ngClass]="{'has-success':
dTitle.valid, 'has-error': !dTitle.valid}">
    <label for="input-title">Title</label>
    <input id="input-title" name="input-title" type="text"
class="form-control" [(ngModel)]="item.Title" placeholder="Insert
the title..." required #dTitle="ngModel" />
    <span class="glyphicon form-control-feedback" aria-
hidden="true" [ngClass]="{'glyphicon-ok': dTitle.valid, 'glyphicon-
remove': ! dTitle.valid}"></span>

    <div [hidden]=" dTitle.valid" class="alert alert-danger">

        You need to enter a valid Title.

    </div>
</div>
```

This is a quite complex implementation, so we should take our time to fully understand it.

Let's start with the `<input id="input-title">` element, it being the center of everything. We can say that all the code we added here has the sole purpose of making the GUI react in real time whenever this input field enters in an invalid state. In order to achieve this outcome, we did the following:

- We added the `required` attribute to the input tag to ensure that it will become invalid whenever it's empty
- We assigned its `ngModel` to a template reference variable that we called `dTitle`, so we can track it in real time throughout our template
- We used the `dTitle` variable's `valid` property value to show/hide elements and/or assign CSS classes to impact the GUI, thus improving the user experience in a number of ways

Here's a detailed list of what we did with the `dTitle` variable:

- We added a `<span>` inside the top `<h3>` element, (only visible when the title control is not valid), featuring a short text whenever the item title is invalid.
- We added two new Bootstrap CSS classes to the title's `form-group` element. The first one (`has-feedback`) will always be present, while the second one will vary: `has-success` if the `dTitle.valid` property is `true`, `has-error` otherwise.
- Right after the `<input>` element we added a Bootstrap `glyphicon` component, which is basically a styled `<span>` that will render the icon bound to its given

class. That class will be determined by an `[ngClass]` conditional directive based upon the `dTitle.Valid` property value: a green check if `true`, a red cross otherwise.

- We also added a standard alert panel that will be visible when the `dTitle` is not valid, just like the `<span>` within the `<h3>` mentioned before.

Note

In order to implement form validation, we were forced to introduce some rather advanced Angular 2 concepts such as template reference variables (also known as `ref-vars`), the `ngControl` attribute, and its strictly related `ngControlName` directive.

To better understand these concepts, we strongly suggest reading the official Angular 2 API documentation at the following URLs:

- `https://angular.io/docs/ts/latest/guide/template-syntax.html#!#ref-vars`
- `https://angular.io/docs/ts/latest/api/core/DirectiveMetadata-class.html#!#exportAs-anchor`
- https://angular.io/docs/ts/latest/api/common/index/NgControlName-directive.html

It's time for us to open our `style.less` file again and add the new styling rules. This time we have used a lot of Bootstrap CSS default classes, so the update will be minimal:

```
.empty-field {
    color: #f04124;
    font-weight: normal;
    font-style: italic;
}
```

This selector should be kept inside the `form.item-detail-edit` block, as it's strictly related to that context and won't be used elsewhere.

Once we've done that, we're free to hit *F5* and see how all these efforts look on screen:
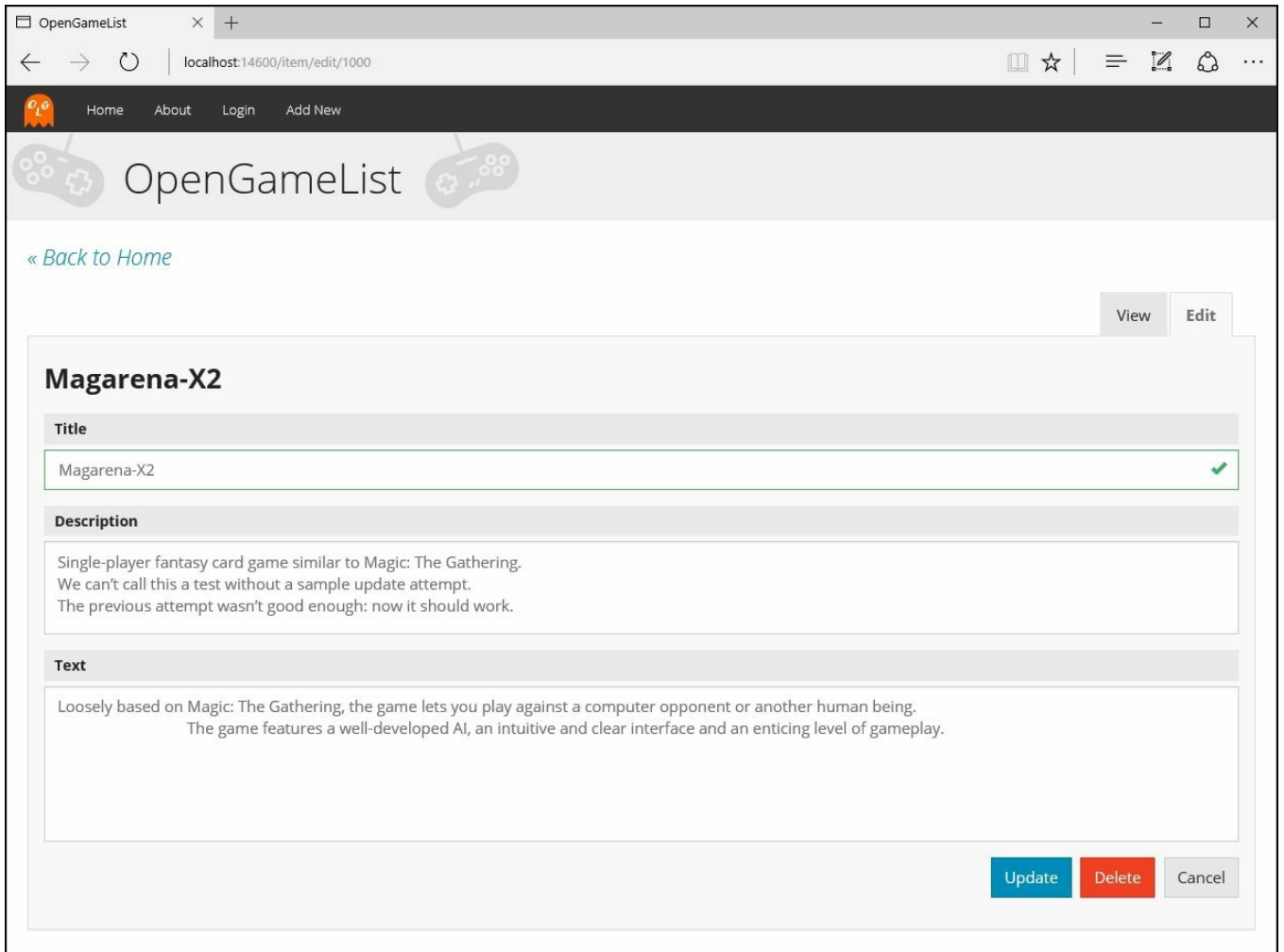
Form featuring a valid title

As we can see, as long as there's a valid `Title`, the input field will now feature a green border and also a green check to the right. Conversely, whenever the `Title` is invalid or becomes empty, a number of UI alerts will come to life: a red border on the input field, a red cross to the right, a red italic title, and also a white-on-red warning panel telling the user that there's a problem with the form:
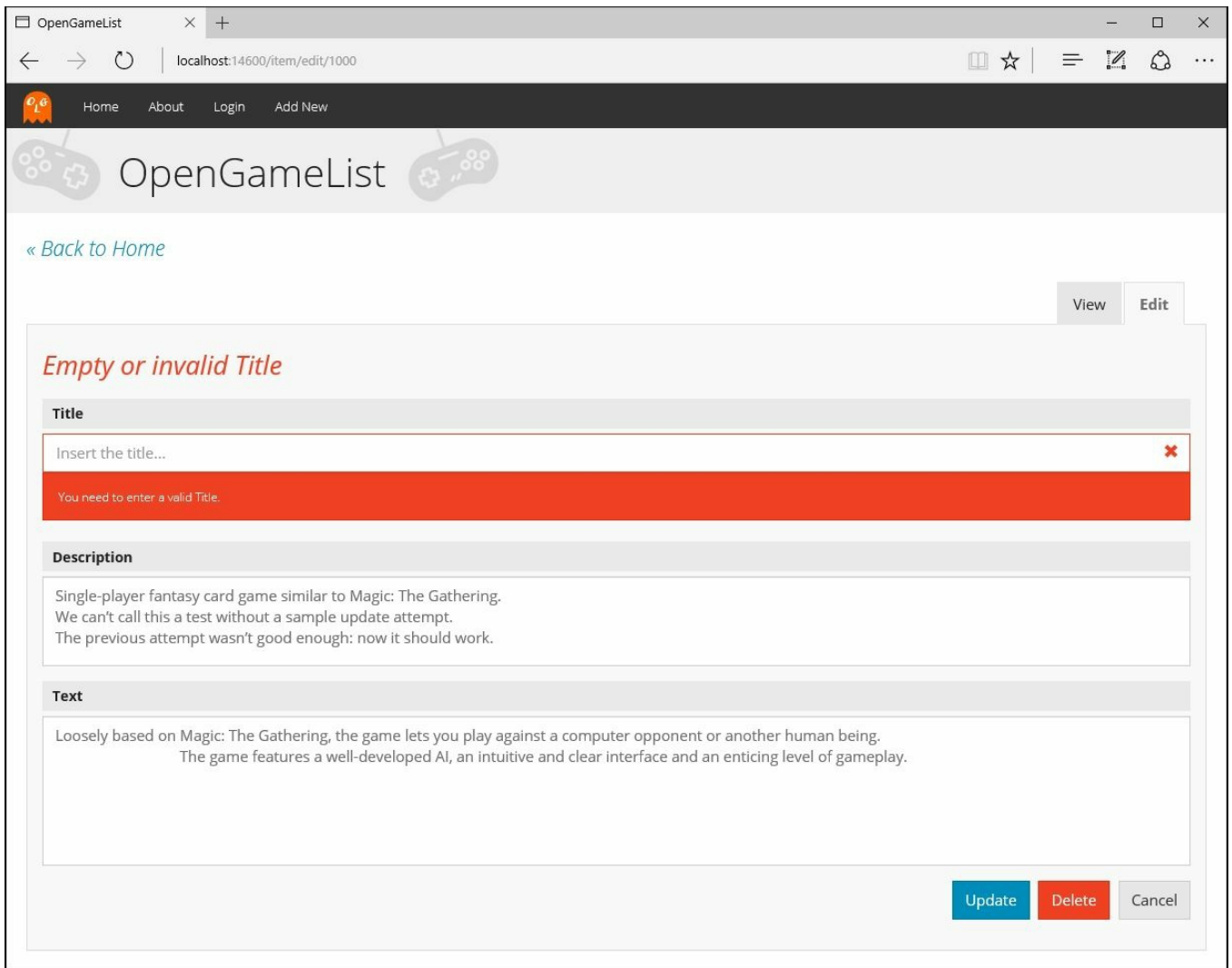
Form featuring an invalid title

It might even be too much for a missing title, yet it has been useful to demonstrate how much control we have upon the layout using Angular 2-plus-Bootstrap only: as a matter of fact, we didn't have to write a single line of JavaScript code.

Conclusions

That's it for now. We just added a simple yet very effective Angular 2 native form validation control. Of course, it has some downsides, such as being client-side only: no server-side feedback is handled in any way. Nonetheless, we can be greatly satisfied with such an outcome.

If we want to further improve what we've done, we can always take a look at the official Bootstrap form-styling documentation at the following URL: http://getbootstrap.com/css/#forms

There are also a lot of examples that will most likely give we good suggestions about how to properly style `<input>` elements and present them to the user in an effective way.

Suggested topics

Style sheet language, SoC, CSS, CSS3, LESS, Sass , Stylus, Switch CSS, Material Design, Material2, Bootstrap, Bootswatch, CDN, expressive Code, unit testing, decoupling, Node.js, Electron, ng2-bootstrap, moment, CSS3 Parent Selector, grid system, Angular 2, NgModel, template reference variables, Angular 2 directives.

Summary

We started this chapter admitting that our ultra-minimalistic UI/UX approach wouldn't work for a potentially shippable product that our Native Web Application should eventually become. Having acknowledged that fact, we added a LESS-based custom stylesheet file to our project. Before doing that, for the benefit of those not familiar with the style sheet pre-processor approach, we spent some time enumerating some of the LESS main advantages.

Right after adding the `style.less` file to our project, we had to choose between adopting one of the popular CSS frameworks such as Bootstrap, YAML, or Foundation, or stick to a full do-it-yourself approach. We briefly enumerated some pros and cons of each alternative, then we opted for Bootstrap 3, mostly because of its great mobile-friendly grid system, saving us the need to write a huge set of layout rules. We chose a suitable theme, then imported it in the `<head>` section of our `index.html` file right before our custom LESS file, so that we could use the latter to apply some custom styling as well.

We then started to apply some Bootstrap and custom styling to the existing components. We started with the navigation menu, replacing our plain list of links with a Bootstrap `navbar` element. We also gave a brand new look and feel to the **Welcome View** and **Item Detail View** (**Display** and **Edit** modes) UI layouts, trying our best to make them prettier, more usable, and also mobile-friendly. While working in edit mode, we introduced some rather advanced Angular 2 concepts in order to implement a rudimental, client-only form validation pattern without writing a single line of JavaScript code.

Chapter 7. Authentication and Authorization

Generally speaking, the term **authentication** refers to any process of verification that someone, be it a human being or an automated system, is who (or what) it claims to be. This is also true within the context of the **World Wide Web** (**WWW**), where that same word is mostly used to denote any technique used by a website or service to collect a set of login info from a user agent, typically a web browser, and authenticate them using a membership and/or **Identity** service.

Authentication should never be confused with **authorization**, as it is a different process and is in charge of a very different task: to give a quick definition, we could say that the purpose of authorization is to confirm that the requesting user is allowed to have access to the action they want to perform.

To better understand the distance between these two apparently similar concepts, we could think of two real-world scenarios:

- A free, yet registered account trying to gain access to a paid or premium only service or feature: this is a common example of *authenticated*, yet not *authorized* access.
- An anonymous user trying to gain access to a publicly available page or file: this is an example of *non-authenticated*, yet *authorized* access.

Do we really need these?

As a matter of fact, implementing **authentication** and/or **authorization** logic isn't mandatory for most web-based applications or services: there are a number of websites that still don't do that, mostly because they serve contents that can be accessed by anyone at any time. This used to be pretty common among most corporate, marketing, and informative websites until some years ago: that was before their owners learned how important it is to build a network of registered users and how much these *loyal* contacts are worth nowadays.

We don't need to be experienced developers to acknowledge how much the World Wide Web has changed in the last few years: each and every website, regardless of its purpose, nowadays has an increasing and more or less legitimate interest in tracking their users, giving them the chance to customize their navigation experience, interacting with their social networks, collecting e-mail addresses, and so on. None of the preceding could be done without an authentication mechanism of some sort.

There are billions of websites and services that require authentication to work properly, as most of their content and/or intents depend upon the actions of registered users: forums, blogs, shopping carts, subscription-based services, and even collaborative tools such as wikis (including ours).

Long story short, the answer is yes: as long as we aim to be a decent wiki, there is no doubt we should implement both an authentication and an authorization procedure. It is the only way to determine who will be able to view, add, update, or delete our valued items, not to mention perform administrative-level tasks, keep track of our users, and handle a lot of important tasks.

Authentication

Since the origin of the World Wide Web, the vast majority of authentication techniques rely upon **HTTP/HTTPS implementation standards**, and all of them work more or less in the following way:

1. A non-authenticated user-agent asks for a content that cannot be accessed without some kind of permissions.
2. The web application returns an authentication request, usually in form of an HTML page containing an empty web form to complete.
3. The user-agent fills up the web form with their credentials, usually a **username** and a **password**, and then sends it back with a `POST` command, which is most likely issued by a click on a **Submit** button.
4. The web application receives the `POST` data and calls the aforementioned server-side implementation that will try to **authenticate** the user with the given input and return an appropriate result.
5. If the result is successful, the web application will authenticate the user and store the relevant data somewhere, depending on the chosen authentication method: sessions/cookies, tokens, signatures, and so on (we'll talk about it later on). Conversely, the result will be presented to the user as a readable outcome inside an error page, possibly asking them to try again, contact an administrator, or something else.

This is still the most common approach nowadays. Almost all websites we can think of are using it, albeit with a number of big or small differences regarding security layers, state management, JWT, or other RESTful tokens, basic or digest access, single sign-on properties, and more.

Third-party authentication

Being forced to have a potentially different username and password for each website visit can be frustrating, other than requiring the users to develop custom *password storage techniques* that might lead to security risks. In order to overcome this issue, we can enhance, or even entirely replace, a standard HTTP-based authentication technique with an authentication protocol based upon third-party providers. The most notable of them is probably `OpenID`, available since 2005 and adopted early by some big players such as Google and StackOverflow, who based their authentication providers upon it. Here's how it works in few words:

- Whenever our application receives an `OpenID` authentication request, it opens a transparent connection interface through the requesting user and a trusted, third-party authentication provider (for example, the **Google Identity Provider**): the interface can be a popup, an AJAX, populated modal windows or an API call, depending on the implementation.
- The user sends his username and password to the aforementioned third-party provider, who performs the *authentication* accordingly and communicates the result to our application by redirecting the user back to where he came, together with a security token that can be used to retrieve the authentication result.
- Our application consumes the token to check the authentication result, authenticating the user in case of success or sending an error response in case of failure.

Authorization

In most standard implementations, including those featured by ASP.NET, the authorization phase kicks in right after the authentication, and it's mostly based on *permissions* or *roles*: any authenticated user might have their own set of permissions and/or belong to one or more roles, and thus be granted access to a specific set of resources. These *role-based* checks are usually set by the developer in a declarative fashion within the application source code and/or configuration files.

Authorization, like we said, shouldn't be confused with authentication, despite the fact it could be easily exploited to perform an implicit authentication as well, especially when it's delegated to a third-party actor.

Third-party authorization

The best known third-party authorization protocol nowadays is **OAuth**, developed by Blaine Cook and Chris Messina in 2006 and widely used by a lot of social networks, including Facebook and Twitter. It basically works like this:

- Whenever an existing user requests a set of permissions to our application via OAuth, we open a transparent connection interface between them and a third-party authorization provider that is trusted by our application (for example, *Facebook*).
- The provider acknowledges the user and, if they have the proper rights, responds entrusting them with a temporary, specific access key.
- The user presents the access key to our application and will be granted access.

Note

We can clearly see how easy it is to exploit this authorization logic for authentication purposes as well; after all, if Facebook says I can do something, shouldn't it also imply that I am who I claim to be? Isn't that enough?

The short answer is no. It might be the case for Facebook, because their OAuth implementation implies that the subscriber receiving the authorization must have authenticated himself to Facebook first; however, this guarantee is not written anywhere, and even if Facebook won't ever change this, considering how many websites are using it for authentication purposes, there is no written guarantee about it. Theoretically speaking, they could split their authorization system from their authentication protocol at any time, thus leading our application's authentication logic to an unrecoverable state of inconsistency.

More generally, we can say that presuming something from something else is almost always a bad practice unless that assumption lies upon very solid, well-documented and (most importantly) highly guaranteed grounds.

Proprietary or third-party?

Theoretically speaking, it's possible to entirely delegate the authentication and/or authorization tasks to existing external, *third-party* providers such as those we mentioned before: there are a lot of web and mobile applications that proudly follow this route nowadays. There are a number of undeniable advantages in using such an approach, including the following:

- **No user-specific DB tables/data models**, just some provider-based identifiers to use here and there as reference keys.
- **Immediate registration**, since there's no need to fill in a registration form and wait for a confirmation e-mail: no username, no password. This will be appreciated by most users and probably increase our conversion rates as well.
- **Little or no privacy issues**, as there's no personal or sensitive data on the application server.
- **No need to handle usernames and passwords** and implement automatic recovery processes.
- **Fewer security-related issues** such as form-based hacking attempts or brute force login attempts.

Of course, there are also some downsides:

- **There won't be an actual user base** so it would be hard to get an overview of active users, get their e-mail address, do statistics, and so on.
- **The login phase might be resource-intensive**, since it will always require an external, back and forth secure connection with a third-party server.
- **All users will need to have (or open) an account with the chosen third-party provider(s)** in order to log in.
- **All users will need to trust our application** because the third-party provider will ask them to authorize it for accessing their data.
- **We will have to register our application with the provider** in order to be able to perform a number of required or optional tasks, such as receive our public and secret keys, authorize one or more URI initiators, and choose the information we want to collect.

Taking all these pros and cons into account, we could say that relying on third-party providers and avoid implementing a proprietary membership provider might be a great time-saving choice for small-scale apps, including ours.

However, we won't be taking that route, because we want the best of both worlds. That's why we'll create an **internal membership provider** that will handle authentication and provide its very own set of authorization rules. Nonetheless, our

users will be also able to log in using their favorite *third-party* provider using the built-in OAuth2 providers support provided by the `AspNetCore.Identity` membership framework.

Choosing an authentication mode

The authentication alternatives made available by ASP.NET Core are basically the same supported by the previous versions of ASP.NET:

- **No authentication**: If we don't feel like implementing anything or if we want to use (or develop) something not relying upon the ASP.NET Identity system
- **Individual user accounts**: When we want to set up an internal database to store user data using the standard ASP.NET Identity interface
- **Azure Active Directory**: Using a token-based set of API calls handled by the **Azure AD Authentication Library** (**ADAL**)
- **Windows authentication**: Viable for local-scope applications only

In Chapter 1, *Getting Ready*, when we created our project, we made the choice to go with an empty project featuring **no authentication**. That was because we didn't want Visual Studio to add `AspNetCore.Identity` support right from the start. Now that we chose to use it, we need to manually add the proper packages.

Installing AspNetCore.Identity

In order to set up and configure the `AspNetCore.Identity` framework, we need to install the required **NuGet** package and perform a number of code changes in some of our project's entity classes.
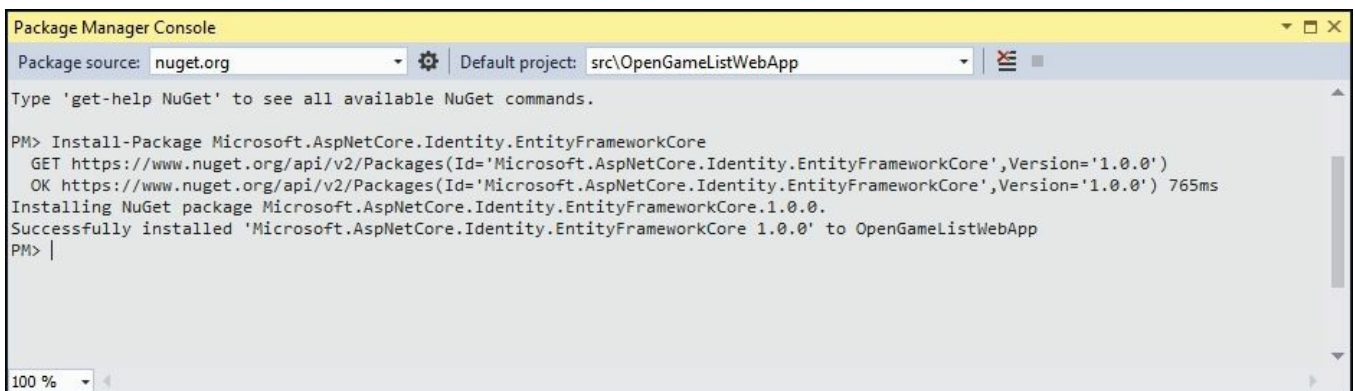
Adding the package

The first thing we're going to do is to check for the existence of the
`Microsoft.AspNetCore.Identity.EntityFrameworkCore`library package, which
we should have already added in Chapter 4, *The Data Model*. If we missed it, we
can fix the issue in a number of ways.

If we like to use the **Package Manager Console**, we can select the appropriate tab
and write the following command:

```
> Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore
```


If we prefer the **Package Manager GUI** interface, right-click in **Solution Explorer**
to the **OpenGameListWebApp** project node, select **Manage NuGet Packages**, and
act accordingly:
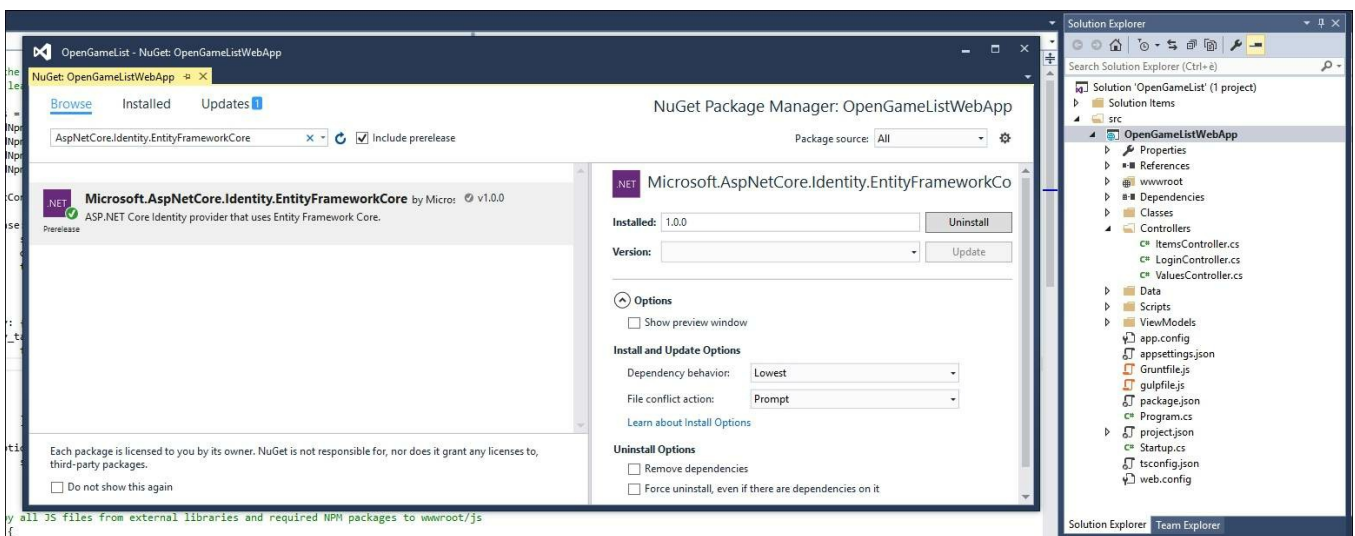
As usual, we can also manage everything directly from the `project.json` file by adding the following line to the `dependencies` section:

```
"Microsoft.AspNetCore.Identity.EntityFrameworkCore": "1.0.0"
```

Updating the project classes

Once done, we need to perform some changes to our project's classes to ensure a proper *Identity* support.

ApplicationDbContext.cs

Open the `Data/ApplicationDbContext.cs` class file and perform the following changes:

1. Add a `using` reference to `Microsoft.AspNetCore.Identity.EntityFrameworkCore`, as required by the new base class:

   ```
           using
   Microsoft.AspNetCore.Identity.EntityFrameworkCore;
   ```

2. Change the base class from `DbContext` to `IdentityDbContext<ApplicationUser>`:

   ```
           public class ApplicationDbContext :
   IdentityDbContext<ApplicationUser>
   ```

3. Remove the `DbSet<ApplicationUser> Users` property, as the `IdentityDbContext` base class already has it built in:

   ```
           #region Properties
           public DbSet<Item> Items { get; set; }
           public DbSet<Comment> Comments { get; set; }
           // public DbSet<ApplicationUser> Users { get; set; }
           #endregion Properties
   ```

ApplicationUser.cs

If we try to compile the project, this file will now produce an error, because our existing `ApplicationUser` class does not extend the `IdentityUser` type, which is a requirement for the `TUser`, generic type required by the `IdentityDbContext` class. To solve the error, switch to the `/ApplicationUsers/ApplicationUser.cs` class and add the `IdentityUser` base class in the following way:

```
namespace OpenGameListWebApp.Data.ApplicationUsers
{
    public class ApplicationUser : IdentityUser
    {
```

Needless to say, we'll have to add a reference to the `AspNetCore.Identity` namespace here as well:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

As soon as we save the file, we'll get three green compilation notices for the `Id`, `Email`, and `UserName` properties, as they are all already present in the `IdentityUser` base class:

We don't need them anymore, so we can comment (or just remove) them as well:

```
//[Key]
//[Required]
//public string Id { get; set; }
//[Required]
//[MaxLength(128)]
//public string UserName { get; set; }
//[Required]
//public string Email { get; set; }
```

That's it! From now on, our `ApplicationUser` entity class is also an `IdentityUser` that can be used by `ASP.NET Identity` for authentication and authorization purposes.

Startup.cs

What we need to do now is to add the Identity-related services to our project's startup class. Open the `Startup.cs` file and add the following to the `ConfigureServices` method, right before the `DbContext` (new lines are highlighted):

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    // Add EntityFramework's Identity support.
    services.AddEntityFramework();

    // Add Identity Services & Stores

    services.AddIdentity<ApplicationUser, IdentityRole>(config => {

        config.User.RequireUniqueEmail = true;

        config.Password.RequireNonAlphanumeric = false;

        config.Cookies.ApplicationCookie.AutomaticChallenge =
false;

    })

        .AddEntityFrameworkStores<ApplicationDbContext>()

        .AddDefaultTokenProviders();

    // Add ApplicationDbContext.
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration["Data:DefaultConnection:Connecti
onString"])
        );

    // Add ApplicationDbContext's DbSeeder
    services.AddSingleton<DbSeeder>();
}
```

In order to make it work, we also need to add the following namespaces:

```
using OpenGameListWebApp.Data.Users;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

DbSeeder.cs

Since we changed our `ApplicationUser` class to make it extend the `IdentityUser`
base class, we most likely broke the seeding mechanism we set up back in Chapter
4, *The Data Model*. On top of that, we should also create some sample roles, since
we now we can make good use of them. These are two good reasons to revise our
current `DbSeeder` class.

Let's open our `/Data/DbSeeder.cs` file and update it accordingly. This is a fat class in terms of source code lines, so we'll just show the relevant changes.

The first thing we need to do is to add a **UserManager** and a **RoleManager**, as they are the required `Asp.NetCore.Identity` handler classes to properly work with users and roles. We can define a private variable for each one of them within the `#Private Members` region (new lines are highlighted):

```
#region Private Members
private ApplicationDbContext DbContext;
private RoleManager<IdentityRole> RoleManager;

private UserManager<ApplicationUser> UserManager;
#endregion Private Members
```

These references will require the following namespaces:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

We can then instantiate these new properties within the `Constructor` using the same dependency injection pattern we already used to instantiate our `ApplicationDbContext`:

```
#region Constructor
public DbSeeder(ApplicationDbContext dbContext,
RoleManager<IdentityRole> roleManager, UserManager<ApplicationUser>
userManager)
    {
        DbContext = dbContext;
        RoleManager = roleManager;

        UserManager = userManager;
    }
#endregion Constructor
```

Right after that, we need to change our `CreateUsers` method to make use of these handlers. Since they all feature methods enforcing `async/await` programming pattern, we also need to make it `async` and change its return type from `void` to `Task`. Therefore, we will also conveniently rename it `CreateUsersAsync` as well. Here's the new method, rewritten from scratch:

```
private async Task CreateUsersAsync()
{
    // local variables
    DateTime createdDate = new DateTime(2016, 03, 01, 12, 30, 00);
    DateTime lastModifiedDate = DateTime.Now;
    string role_Administrators = "Administrators";
```

```csharp
    string role_Registered = "Registered";

    //Create Roles (if they doesn't exist yet)
    if (!await RoleManager.RoleExistsAsync(role_Administrators))
await RoleManager.CreateAsync(new
IdentityRole(role_Administrators));
    if (!await RoleManager.RoleExistsAsync(role_Registered)) await
RoleManager.CreateAsync(new IdentityRole(role_Registered));

    // Create the "Admin" ApplicationUser account (if it doesn't
exist already)
    var user_Admin = new ApplicationUser() {
        UserName = "Admin",
        Email = "admin@opengamelist.com",
        CreatedDate = createdDate,
        LastModifiedDate = lastModifiedDate
    };

    // Insert "Admin" into the Database and also assign the
"Administrator" role to him.
    if (await UserManager.FindByIdAsync(user_Admin.Id) == null)
    {
        await UserManager.CreateAsync(user_Admin, "Pass4Admin");
        await UserManager.AddToRoleAsync(user_Admin,
role_Administrators);
        // Remove Lockout and E-Mail confirmation.
        user_Admin.EmailConfirmed = true;
        user_Admin.LockoutEnabled = false;
    }

#if DEBUG
    // Create some sample registered user accounts (if they don't
exist already)
    var user_Ryan = new ApplicationUser() {
        UserName = "Ryan",
        Email = "ryan@opengamelist.com",
        CreatedDate = createdDate,
        LastModifiedDate = lastModifiedDate,
        EmailConfirmed = true,
        LockoutEnabled = false
    };
    var user_Solice = new ApplicationUser() {
        UserName = "Solice",
        Email = "solice@opengamelist.com",
        CreatedDate = createdDate,
        LastModifiedDate = lastModifiedDate,
        EmailConfirmed = true,
        LockoutEnabled = false
    };
    var user_Vodan = new ApplicationUser() {
        UserName = "Vodan",
        Email = "vodan@opengamelist.com",
```

```
            CreatedDate = createdDate,
            LastModifiedDate = lastModifiedDate,
            EmailConfirmed = true,
            LockoutEnabled = false
        };
    // Insert sample registered users into the Database and also
assign the "Registered" role to him.
    if (await UserManager.FindByIdAsync(user_Ryan.Id) == null)
    {
        await UserManager.CreateAsync(user_Ryan, "Pass4Ryan");
        await UserManager.AddToRoleAsync(user_Ryan,
role_Registered);
        // Remove Lockout and E-Mail confirmation.
        user_Ryan.EmailConfirmed = true;
        user_Ryan.LockoutEnabled = false;
    }
    if (await UserManager.FindByIdAsync(user_Solice.Id) == null)
    {
        await UserManager.CreateAsync(user_Solice, "Pass4Solice");
        await UserManager.AddToRoleAsync(user_Solice,
role_Registered);
        // Remove Lockout and E-Mail confirmation.
        user_Solice.EmailConfirmed = true;
        user_Solice.LockoutEnabled = false;
    }
    if (await UserManager.FindByIdAsync(user_Vodan.Id) == null)
    {
        await UserManager.CreateAsync(user_Vodan, "Pass4Vodan");
        await UserManager.AddToRoleAsync(user_Vodan,
role_Registered);
        // Remove Lockout and E-Mail confirmation.
        user_Vodan.EmailConfirmed = true;
        user_Vodan.LockoutEnabled = false;
    }
#endif
    await DbContext.SaveChangesAsync();
}
```

As we can see, we made some relevant changes here:

- The `DbContext.Add` and `DbContext.AddRange` methods have been replaced by those provided by the `UserManager`. This allow us to specify a password that will be automatically hashed and also to avoid any explicit `Id` assignment, as they will be auto-generated.
- We used `RoleManager` to create two sample roles: administrators and registered.
- We modified our code to add the `admin` user to the administrators role and all the other sample users to the registered role.

Once done, we need to update the `SeedAsync` method to reflect the rename we just

did on `CreateUsersAsync` and also handle the fact that the latter is now asynchronous as well:

```
#region Public Methods
public async Task SeedAsync()
{
    // Create default Users
    if (await DbContext.Users.CountAsync() == 0) await
CreateUsersAsync();
    // Create default Items (if there are none) and Comments
    if (await DbContext.Items.CountAsync() == 0) CreateItems();
}
#endregion Public Methods
```

With this, we're done updating our project's classes.

Before going further, it might be wise to issue a whole project rebuild to make sure we're not getting build errors within our code.

Updating the database
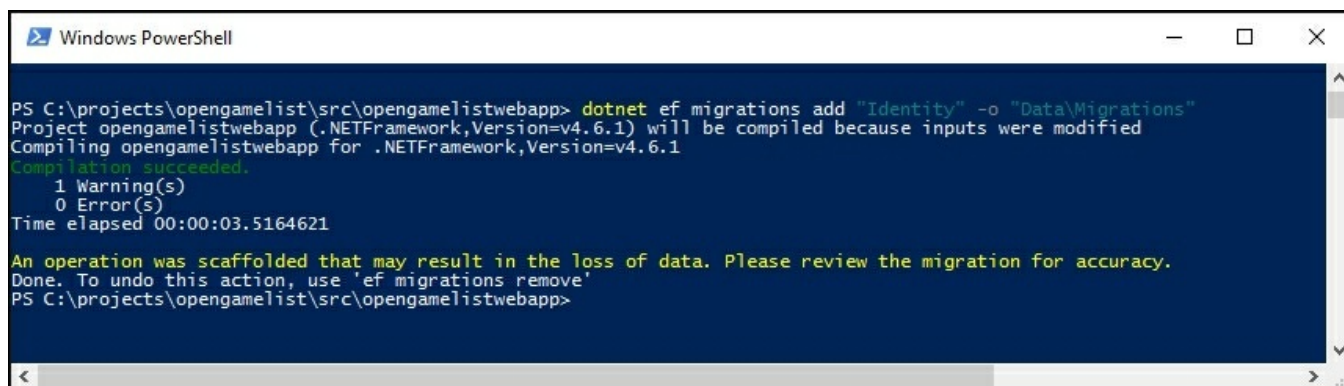
It's time to create a new migration and reflect the code changes to the database by taking advantage of the code-first approach we chose in [Chapter 4](#), *The Data Model*.

It's worth noting that if we were using *Entity Framework 6*, we could entirely skip this step by implementing the **auto-migration** feature it used to have. Unfortunately, there's no such thing in **EF core**, so we must add our migrations manually.

Let's open a Powershell command prompt and go to our project's root folder, then write the following:

```
> dotnet ef migrations add "Identity" -o "Data\Migrations"
```

A new migration will be added to the project. Right after that, we could choose to update our database...except it won't be a good idea. Applying the new migration will most likely cause some data loss or other consistency issues due to the fact that our `ApplicationUser` class experienced some major changes. Such a scenario is also clearly stated by the yellow message shown by the Powershell tool upon completing its given task:

Since we updated our `DbSeeder` class to support the new changes, the best thing we can do would be letting it re-populate our database accordingly. Unfortunately, we know perfectly well that as long as there are some existing users and items in the database tables, it won't even run. This leaves us with nothing but one solution: drop and recreate the database, so the `DbSeeder` will kick in and re-populate everything on the first run.

Note
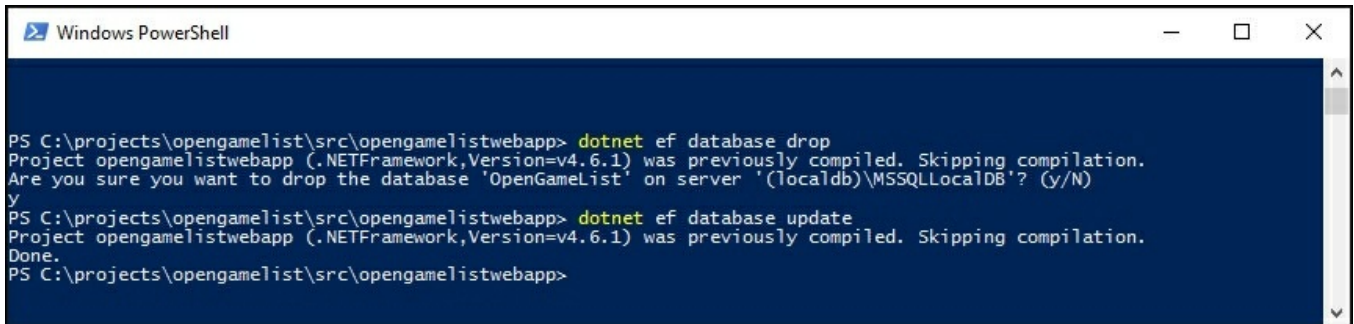
Although it might seem a horrible way to fix things, that's definitely not the case here,

as we're still in development phase. We haven't touched our database contents yet, so we won't mind them being re-seeded from scratch into a new, `Identity-aware` form.

In order to do that, issue the following Powershell commands:

```
> dotnet ef database drop
> dotnet ef database update
```

We'll also have to hit *Y* to confirm the drop.

Once done, hit *F5* and wait for the `DbSeeder` to kick in. After that, it will do its magic. We'll have an updated database with full `AspNetCore.Identity` support.

Authentication methods

Now that we have updated our database to support the `AspNetCore.Identity` authentication workflow and patterns, we should choose which authentication method to implement.

As we most certainly know, the HTTP protocol is *stateless*, meaning that whatever we do during a request/response cycle will be lost before the subsequent request, including the authentication result. The only way we have to overcome this is to store that result somewhere, together with all its relevant data, such as user ID, login date/time, and last request time.

Sessions

Since few years ago, the most common and traditional method to do that was to store this data on the server by using either a memory-based, disk-based, or external session manager. Each session can be retrieved using a unique ID that the client receives with the authentication response, usually inside a *session cookie*, that will be transmitted to the server on each subsequent request.

Here's a brief diagram showing the **Session-Based Authentication Flow**:
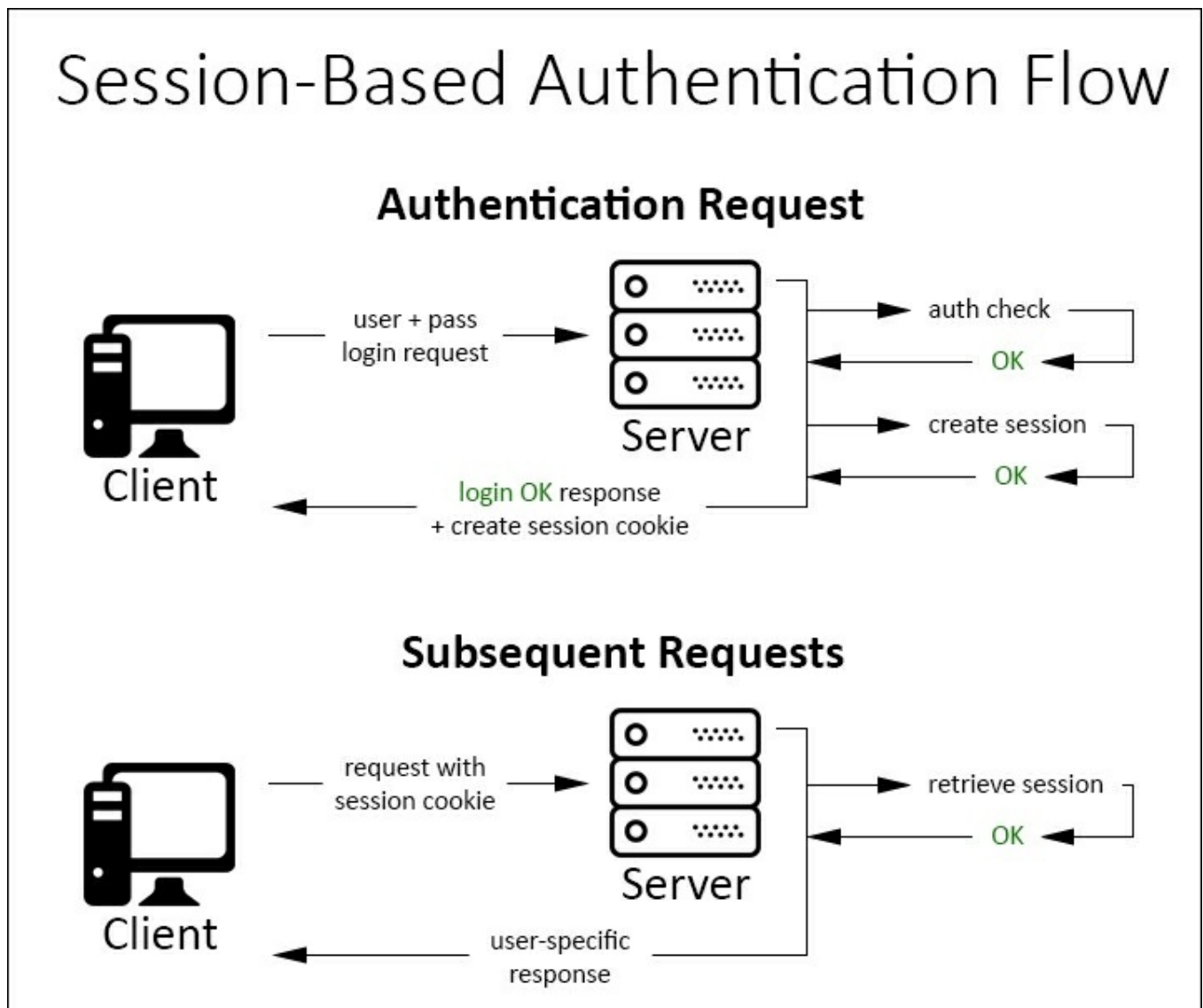
This is still a very common technique used by most web applications. There's nothing wrong with adopting this approach, as long as we are ok with its widely acknowledged downsides, such as the following:

- **Memory issues**: Whenever there are many authenticated users, the web server

will consume more and more memory. Even if we use a file-based or external session provider, there will nonetheless be an intensive IO, TCP, or socket overhead.

- **Scalability issues**: Replicating a session provider in a scalable web farm might not be an easy task and will often lead to bottlenecks or wasted resources.
- **Cross-domain issues**: Session cookies behave just like standard cookies, so they cannot be easily shared among different origins/domains. These kinds of problem can be often solved with some workarounds, yet they will often lead to insecure scenariosto make things work.
- **Security issues**: There is a wide and detailed literature of security-related issues involving sessions and session cookies: XSS attacks, cross-site request forgery, and a number of other threats that won't be covered here for the sake of simplicity. Most of them can be mitigated by some countermeasures, yet they could be difficult to handle for first-hand developers.

As these issues arose over the years, there's no doubt that most analysts and developers put effort into figuring out different approaches.

Tokens

Token-based authentication has been increasingly adopted by single-page applications and mobile apps in the last few years for a number of undeniably good reasons that we'll try to briefly summarize here.

The most important difference between session-based authentication and token-based authentication is that the latter is *stateless*, meaning that we won't be storing any user-specific information on the server memory, database, session provider, or other data containers of any sort.

This single aspect solves most of the downsides that we pointed out earlier for session-based authentication. We won't have sessions, so there won't be an increasing overhead; we won't need a session provider, so scaling will be much easier; plus, for browsers supporting `LocalStorage`, we won't be even using cookies, so we won't get blocked by cross-origin restrictive policies and, hopefully, we'll get around most security issues.

Here's a typical **Token-Based Authentication Flow**:
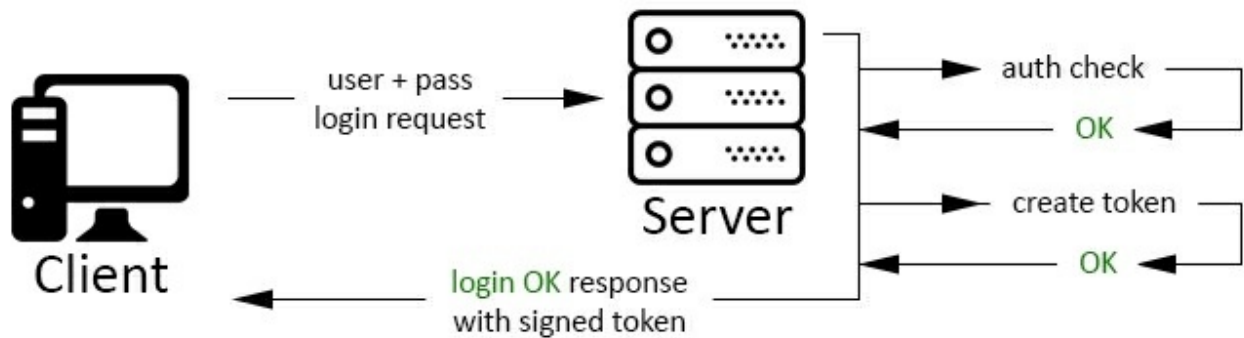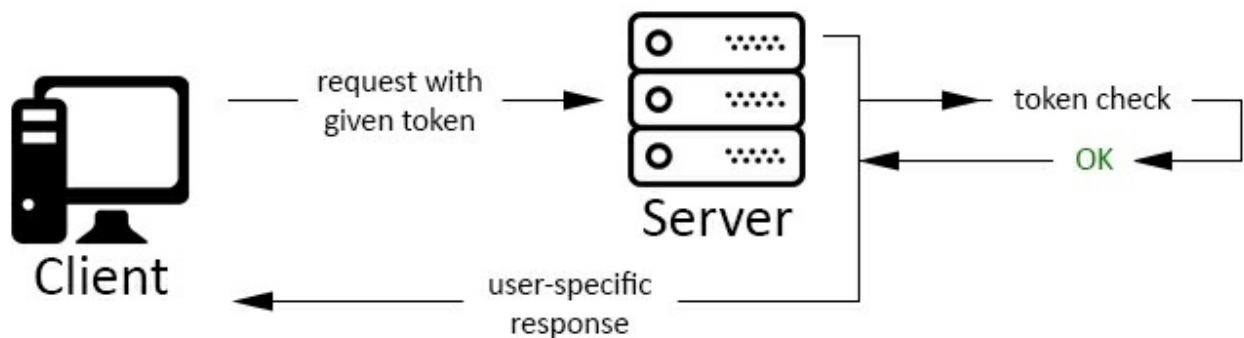
As we can see, the various steps seem very similar. The only big deal is the fact that we create/check tokens instead of creating/retrieving sessions from somewhere.

Signatures

This is a method used by most modern API-based cloud-computing and storage services, including **Amazon Web Services (AWS)**. In contrast with session-based and token-based approaches, which rely upon a transport layer that can be theoretically accessed by/exposed to a third-party attacker, signature-based authentication performs a hash of the whole request using a previously shared `private` key. This ensures that no intruder or man-in-the-middle could ever act as the requesting user, as they won't be able to sign the request.

Two-factor

This is the standard authentication method used by most banking and financial accounts, being arguably the most secure one. The implementation may vary, but it always relies upon the following base workflow:

- The user performs a standard login with a username and password.
- The server identifies the user and prompts them with an additional, user-specific request that can be only satisfied by something obtained or obtainable through a different channel: an OTP password sent by SMS, a unique authentication card with a number of answer codes, a dynamic PIN generated by a proprietary device or a mobile app, and so on.
- If the user gives the correct answer, they get authenticated using a standard session-based or token-based method.

Conclusions

After reviewing all these authentication methods, we're going to use a token-based authentication approach featuring **JSON Web Tokens** (**JWT**), as it seems the most viable one for our specific scenario.

JWT is a JSON-based open standard explicitly designed for native web applications, available in multiple languages, such as .NET, Python, Java, PHP, Ruby, JavaScript/NodeJS, and PERL. We're choosing it because it's becoming a de facto standard for token authentication, as it's natively supported by most technologies.

For specific details about JWT, we recommend reading the following page: https://jwt.io/

Implementing JSON web token authentication

In order to handle JWT-based token authentication, we need to implement the required middleware for doing these tasks:

- **Generating** the JWT tokens upon username/password `POST` requests coming from our client.
- **Validating** any JWT token coming with requests by looking at their `headers` and *cookies*

Although ASP.NET Core natively supports JWT tokens, the only available middleware is the one validating the request `headers` (`JwtBearerMiddleware`). This leaves us with two choices: manually implement what's missing or rely on a third-party library that does just that. We'll try the hand-made route throughout the rest of this chapter, leaving the other alternative to the following chapter.

The first thing to do is define the required steps we need to take care of:

1. Implement a custom **JWT provider** middleware to accept `POST` requests carrying a username and password, and generate JWT tokens accordingly.
2. Add it to the HTTP request pipeline, together with a properly configured `JwtBearerMiddleware` to validate incoming requests containing a JWT in their `headers` block.
3. Create an Angular 2 `Login` form to allow our users to perform the login.
4. Create an Angular 2 `Auth` service that will handle login/logout and store the JWT token so it can be reused.
5. Create an `AuthHttp` wrapper that will add the JWT (if present) to the `headers` block of each request.

Sounds like a plan...let's do this.

JWT provider

The first thing we need to do is to add the following packages to our project:

```
"Microsoft.IdentityModel.Tokens": "5.0.0",
"System.IdentityModel.Tokens.Jwt": "5.0.0"
```

As always, this can be done in a number of ways: **NuGet**, **GUI**, **project.json**, and others. We already know how to do that. The most recent version as we write is `5.0.0` for both packages, but we can expect it to change in the near future.

Once done, right-click to the **OpenGameListWebApp** project and create a `/Classes/` folder. This is where we will put our custom implementations. We could also call it `/AppCode/`, `/Infrastructure/`, or anything else that we like.

Right-click on the new folder, choose the **Add | New Item** option, and add a new **ASP.NET | Middleware Class**, naming it `JwtProvider.cs` just like in the following screenshot:
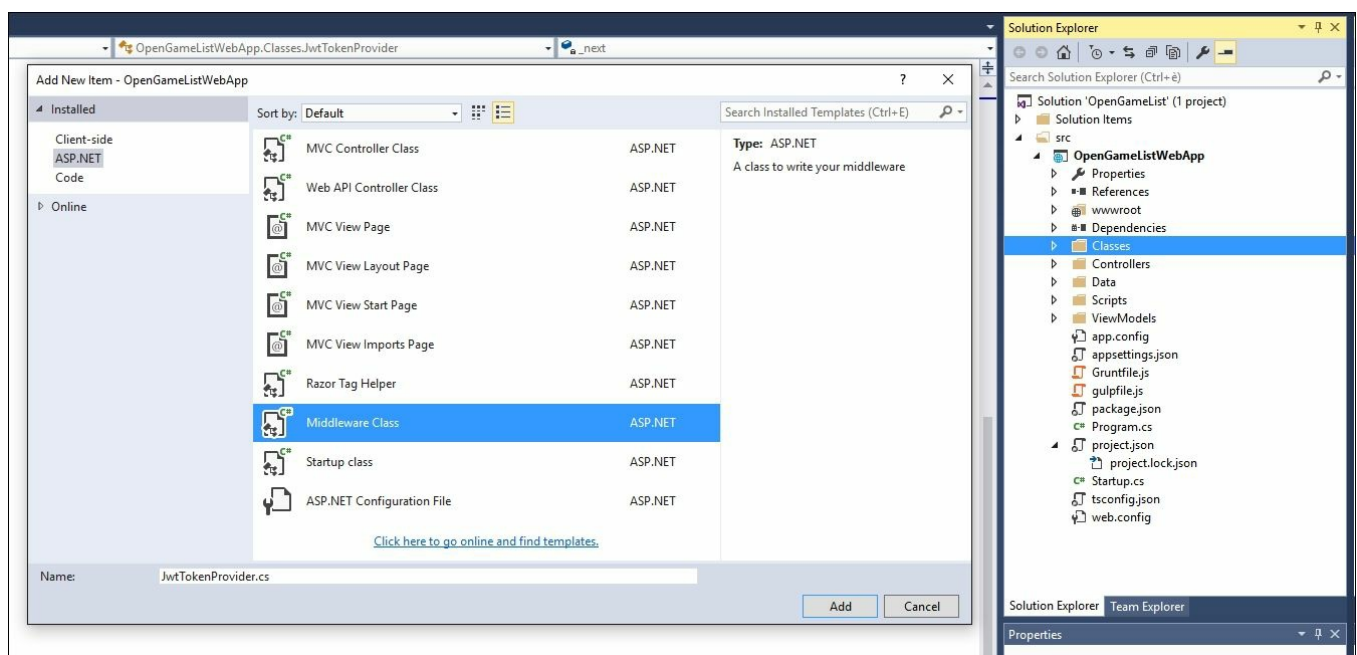
The new class will contain the default code for the ASP.NET core middleware class implementation pattern. We need to implement a lot of stuff here, so we'll split the content into several regions to make it more readable and understandable.

Private members

Let's add a **private members** region, wrapping the existing `_next` variable and

adding the following (new lines highlighted):

```
#region private members
private readonly RequestDelegate _next;

// JWT-related members

private TimeSpan TokenExpiration;

private SigningCredentials SigningCredentials;

// EF and Identity members, available through DI

private ApplicationDbContext DbContext;

private UserManager<ApplicationUser> UserManager;

private SignInManager<ApplicationUser> SignInManager;

#endregion Private Members
```

Don't forget to add the required namespaces as well at the beginning of the file:

```
using Microsoft.IdentityModel.Tokens;
using Microsoft.AspNetCore.Identity;
using OpenGameListWebApp.Data.Users;
using OpenGameListWebApp.Data;
using System.Text;
```

As we can see, we're defining a number of variables here that we'll be using internally. Most of them will be instantiated in the constructor, either programmatically or by using the dependency injection pattern we've already used several times.

Static members

This region includes the minimum amount of info needed to sign in using a JWT token: a `SecurityKey` and an `Issuer`. We also define a `TokenEndPoint` here, which is the URL path that we will use to process the incoming authentication login requests. To put it in other words, it's the route that the `JwtProvider` will have to intercept (right before the standard MVC routing strategy) to properly handle the login requests:

```
#region Static Members
private static readonly string PrivateKey =
"private_key_1234567890";
public static readonly SymmetricSecurityKey SecurityKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(PrivateKey));
public static readonly string Issuer = "OpenGameListWebApp";
```

```
public static string TokenEndPoint = "/api/connect/token";
#endregion Static Members
```

Notice that most of these static members have the `public` access modifier. That's because we'll be using them outside of this class when we'll have to configure the token verification middleware.

Tip

Hardcoding these values in the provider source code is not ideal in production environments. We did it for the sake of simplicity, yet we should remember to adopt better and most secure approaches, such as storing them within an environment variable or a key management tool.

Constructor

Here's what the `Constructor` region looks like:

```
#region Constructor
public JwtProvider(
    RequestDelegate next,
    ApplicationDbContext dbContext,
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager)
{
    _next = next;

    // Instantiate JWT-related members
    TokenExpiration = TimeSpan.FromMinutes(10);
    SigningCredentials =  new SigningCredentials(SecurityKey,
SecurityAlgorithms.HmacSha256);

    // Instantiate through Dependency Injection
    DbContext = dbContext;
    UserManager = userManager;
    SignInManager = signInManager;
}
#endregion Constructor
```

Here, we define the JWT token expiration time and encrypt the symmetrical security key that will be used to validate JWTs using a standard `HmacSha256` encryption algorithm. We're also instantiating the EF/Identity members through DI, like we have done a number of times.

Public methods

Let's move to the `Invoke` method, which we conveniently wrapped inside the `public` methods region:

```
#region public methods
public Task Invoke(HttpContext httpContext)
{
    // Check if the request path matches our TokenEndPoint
    if (!httpContext.Request.Path.Equals(TokenEndPoint,
StringComparison.Ordinal)) return _next(httpContext);

  // Check if the current request is a valid POST with the
appropriate content type (application/x-www-form-urlencoded)
  if (httpContext.Request.Method.Equals("POST") &&
httpContext.Request.HasFormContentType)
  {
      // OK: generate token and send it via a json-formatted string
      return CreateToken(httpContext);
  }
  else
 {
      // Not OK: output a 400 - Bad request HTTP error.
      httpContext.Response.StatusCode = 400;
      return httpContext.Response.WriteAsync("Bad request.");
  }
}
#endregion public methods
```

Here, we need to check whether the request path matches the chosen login path. If it does, we continue execution, otherwise we entirely skip the request. Right after that, we need to check whether the current request is a valid *form-urlencoded* `POST`. If that's the case, we call the `CreateToken` internal method; otherwise, we return a `400` error response.

Private methods

The `CreateToken` method is where most of the magic takes place. We check the given username and password against our internal Identity database and, depending on the result, generate and return either a JWT token or an appropriate error response:

```
#region Private Methods
private async Task CreateToken(HttpContext httpContext)
{
    try
    {
        // retrieve the relevant FORM data
        string username = httpContext.Request.Form["username"];
        string password = httpContext.Request.Form["password"];

        // check if there's an user with the given username
        var user = await UserManager.FindByNameAsync(username);
        // fallback to support e-mail address instead of username
        if (user == null && username.Contains("@")) user = await
```

```
UserManager.FindByEmailAsync(username);

        var success = user != null && await
UserManager.CheckPasswordAsync(user, password);
        if (success)
        {
            DateTime now = DateTime.UtcNow;

            // add the registered claims for JWT (RFC7519).
            // For more info, see https:
            //tools.ietf.org/html/rfc7519#section-4.1
            var claims = new[] {
            new Claim(JwtRegisteredClaimNames.Iss, Issuer),
            new Claim(JwtRegisteredClaimNames.Sub, user.Id),
            new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
            new Claim(JwtRegisteredClaimNames.Iat, new
DateTimeOffset(now).ToUnixTimeSeconds().ToString(),
ClaimValueTypes.Integer64)
            // TODO: add additional claims here
        };

        // Create the JWT and write it to a string
        var token = new JwtSecurityToken(
        claims: claims,
        notBefore: now,
        expires: now.Add(TokenExpiration),
        signingCredentials: SigningCredentials);
        var encodedToken = new
JwtSecurityTokenHandler().WriteToken(token);

        // build the json response
        var jwt = new {
            access_token = encodedToken,
            expiration = (int)TokenExpiration.TotalSeconds
        };

        // return token
        httpContext.Response.ContentType = "application/json";
        await
httpContext.Response.WriteAsync(JsonConvert.SerializeObject(jwt));
        return;
        }
    }
    catch (Exception ex)
    {
        // TODO: handle errors
        throw ex;
    }

    httpContext.Response.StatusCode = 400;
    await httpContext.Response.WriteAsync("Invalid username or
```

```
password.");
}
#endregion Private Methods
```

This will also require the following namespace references:

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Newtonsoft.Json;
```

The code is pretty much self-documented using some inline comments indicating what we're doing here and there. We can see how the username and password are retrieved from the `HttpContext` and checked using the `AspNetCore.Identity` `UserManager` class; if the user exists, we issue a JSON-formatted object containing a JWT token and its expiration time, otherwise we return a HTTP `400` error.

Note

It's also worth noting that, as an additional feature, we configured the method to allow clients to authenticate themselves using their *e-mail* address in place of the *username;* we did that to demonstrate how versatile this implementation actually is, since we do have full control over the whole authentication process.

Extension methods

The sample code provided for middleware classes includes a handy extension method that we can use to add our newborn provider to the request pipeline. We don't need to change it, so we'll just wrap it in an `extension methods` region:

```
#region Extension Methods
// Extension method used to add the middleware to the HTTP request
pipeline.
public static class JwtProviderExtensions
{
    public static IApplicationBuilder UseJwtProvider(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<JwtProvider>();
    }
}
#endregion Extension Methods
```

Full source code

Here's how our `JwtProvider` class will look after all this hard work:

```
using System;
using System.Text;
using System.Threading.Tasks;
```

```csharp
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.IdentityModel.Tokens;
using OpenGameListWebApp.Data;
using Microsoft.AspNetCore.Identity;
using OpenGameListWebApp.Data.Users;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Newtonsoft.Json;

namespace OpenGameListWebApp.Classes
{
    public class JwtProvider
    {
        #region Private Members
        private readonly RequestDelegate _next;

        // JWT-related members
        private TimeSpan TokenExpiration;
        private SigningCredentials SigningCredentials;

        // EF and Identity members, available through DI
        private ApplicationDbContext DbContext;
        private UserManager<ApplicationUser> UserManager;
        private SignInManager<ApplicationUser> SignInManager;
        #endregion Private Members

        #region Static Members
        private static readonly string PrivateKey =
"private_key_1234567890";
        public static readonly SymmetricSecurityKey SecurityKey =
new SymmetricSecurityKey(Encoding.ASCII.GetBytes(PrivateKey));
        public static readonly string Issuer =
"OpenGameListWebApp";
        public static string TokenEndPoint = "/api/connect/token";
        #endregion Static Members

        #region Constructor
        public JwtProvider(
            RequestDelegate next,
            ApplicationDbContext dbContext,
            UserManager<ApplicationUser> userManager,
            SignInManager<ApplicationUser> signInManager)
        {
            _next = next;

            // Instantiate JWT-related members
            TokenExpiration = TimeSpan.FromMinutes(10);
            SigningCredentials =  new
SigningCredentials(SecurityKey, SecurityAlgorithms.HmacSha256);

            // Instantiate through Dependency Injection
```

```csharp
            DbContext = dbContext;
            UserManager = userManager;
            SignInManager = signInManager;
        }
        #endregion Constructor

        #region Public Methods
        public Task Invoke(HttpContext httpContext)
        {
            // Check if the request path matches our LoginPath
            if (!httpContext.Request.Path.Equals(TokenEndPoint,
StringComparison.Ordinal)) return _next(httpContext);

            // Check if the current request is a valid POST with
the appropriate content type (application/x-www-form-urlencoded)
            if (httpContext.Request.Method.Equals("POST") &&
httpContext.Request.HasFormContentType)
            {
                // OK: generate token and send it via a json-
formatted string
                return CreateToken(httpContext);
            }
            else
            {
                // Not OK: output a 400 - Bad request HTTP error.
                httpContext.Response.StatusCode = 400;
                return httpContext.Response.WriteAsync("Bad
request.");
            }
        }
        #endregion Public Methods

        #region Private Methods
        private async Task CreateToken(HttpContext httpContext)
        {
            try
            {
                // retrieve the relevant FORM data
                string username =
httpContext.Request.Form["username"];
                string password =
httpContext.Request.Form["password"];

                // check if there's an user with the given username
                var user = await
UserManager.FindByNameAsync(username);
                // fallback to support e-mail address instead of
username
                if (user == null && username.Contains("@")) user =
await UserManager.FindByEmailAsync(username);

                var success = user != null && await
```

```
UserManager.CheckPasswordAsync(user, password);
            if (success)
            {
                DateTime now = DateTime.UtcNow;

                // add the registered claims for JWT (RFC7519).
                // For more info, see
https://tools.ietf.org/html/rfc7519#section-4.1
                var claims = new[] {
                    new Claim(JwtRegisteredClaimNames.Iss,
Issuer),
                    new Claim(JwtRegisteredClaimNames.Sub,
username),
                    new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
                    new Claim(JwtRegisteredClaimNames.Iat, new
DateTimeOffset(now).ToUnixTimeSeconds().ToString(),
ClaimValueTypes.Integer64)
                    // TODO: add additional claims here
                };

                // Create the JWT and write it to a string
                var token = new JwtSecurityToken(
                    claims: claims,
                    notBefore: now,
                    expires: now.Add(TokenExpiration),
                    signingCredentials: SigningCredentials);
                var encodedToken = new
JwtSecurityTokenHandler().WriteToken(token);

                // build the json response
                var jwt = new {
                    access_token = encodedToken,
                    expiration =
(int)TokenExpiration.TotalSeconds
                };

                // return token
                httpContext.Response.ContentType =
"application/json";
                await
httpContext.Response.WriteAsync(JsonConvert.SerializeObject(jwt));
                return;
            }
        }
        catch (Exception ex)
        {
            // TODO: handle errors
        }

        httpContext.Response.StatusCode = 400;
        await httpContext.Response.WriteAsync("Invalid username
```

```
or password.");
        }
        #endregion Private Methods
    }


    #region Extension Methods
    // Extension method used to add the middleware to the HTTP
request pipeline.
    public static class JwtProviderExtensions
    {
        public static IApplicationBuilder UseJwtProvider(this
IApplicationBuilder builder)
        {
            return builder.UseMiddleware<JwtProvider>();
        }
    }
    #endregion Extension Methods
}
```

Adding the middleware to the pipeline

Now that we have created our `JwtProvider` middleware, we can add it to the request pipeline together with the built-in `JwtBearerMiddleware`. In order to do that, open the `Startup.cs` file and add the following code to the `Configure` method (new lines highlighted):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory, DbSeeder dbSeeder)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
the following URL:
    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = (context) =>
        {
            // Disable caching for all static files.
            context.Context.Response.Headers["Cache-Control"] =
Configuration["StaticFiles:Headers:Cache-Control"];
            context.Context.Response.Headers["Pragma"] =
Configuration["StaticFiles:Headers:Pragma"];
            context.Context.Response.Headers["Expires"] =
Configuration["StaticFiles:Headers:Expires"];
        }
    });

    // Add a custom Jwt Provider to generate Tokens

    app.UseJwtProvider();

    // Add the Jwt Bearer Header Authentication to validate Tokens

    app.UseJwtBearerAuthentication(new JwtBearerOptions()

    {

        AutomaticAuthenticate = true,

        AutomaticChallenge = true,

        RequireHttpsMetadata = false,
```

```
            TokenValidationParameters = new TokenValidationParameters()

            {

                IssuerSigningKey = JwtProvider.SecurityKey,

                ValidateIssuerSigningKey = true,

                ValidIssuer = JwtProvider.Issuer,

                ValidateIssuer = false,

                ValidateAudience = false

            }

        });

        // Add MVC to the pipeline
        app.UseMvc();

        // TinyMapper binding configuration
        TinyMapper.Bind<Item, ItemViewModel>();

        // Seed the Database (if needed)
        try
        {
            dbSeeder.SeedAsync().Wait();
        }
        catch (AggregateException e)
        {
            throw new Exception(e.ToString());
        }
}
```

To avoid compilation errors, be sure to declare the following namespaces to the beginning of the file:

```
using OpenGameListWebApp.Classes;
using Microsoft.IdentityModel.Tokens;
```

It's important to focus on two important things here:

- **Middleware order does indeed count**. Notice how `MVC` gets added after `JwtProvider` and `JwtBearerAuthentication`, so the `MVC` default routing strategies won't interfere with them.
- **There's no AspNetCore.Identity middleware in there**. We purposely avoided calling the `app.UseIdentity()` extension because it internally wraps `app.UseCookieAuthentication()`, which is something we don't need. We

might want to add it if we want to support cookies over `headers`, or even use both of them.

Tip

To know more about what's under the hood of `app.UseIdentity()`, it can be useful to take a look at the extension's source code, which is publicly available on **GitHub** at the following URL:

https://github.com/aspnet/Identity/blob/dev/src/Microsoft.AspNetCore.Identity/Build

With this, we're done with the server-side part of our job. Let's switch to the client side.

Angular 2 login form

Remember that `/Scripts/app/login.component.ts` sample we created back in [Chapter 3](#), *Angular 2 Components and Client-Side Routing*. The time has come to update it into a proper `login` form.

Open that file and modify the existing, almost empty `template` with the following code:

```
<div class="login-container">
    <h2 class="form-login-heading">Login</h2>
    <div class="alert alert-danger" role="alert"
*ngIf="loginError">
        <strong>Warning:</strong> Username or Password mismatch
    </div>
    <form class="form-login" [formGroup]="loginForm"
(submit)="performLogin($event)">
        <input formControlName="username" type="text" class="form-
control" placeholder="Your username or e-mail address" required
autofocus />
        <input formControlName="password" type="password"
class="form-control" placeholder="Your password" required />
        <div class="checkbox">
            <label>
                <input type="checkbox" value="remember-me">
                Remember me
            </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Sign in</button>
    </form>
</div>
```

That's a simple login form with some Bootstrap and custom classes. Notice that we also defined an `ngFormModel` and an event handler method called `performLogin` that will trigger on each submit. Both should be added within the component's `class` implementation in the following way (new lines highlighted):

```
export class LoginComponent {
    title = "Login";
    loginForm = null;

    constructor(private fb: FormBuilder) {

        this.loginForm = fb.group({

            username: ["", Validators.required],

            password: ["", Validators.required]
```

```
        });

    }

    performLogin(e) {

        e.preventDefault();

        alert(JSON.stringify(this.loginForm.value));

    }
}
```

We're introducing two new classes here:

- **FormBuilder**: This is a factory class for creating instances of type `FormGroup`, which is how Angular 2 handles model-driven (or reactive) forms, we'll say more regarding this topic in a short while.
- **Validators**: Angular has three built-in form validations that can be applied using this class. These are `Validators.required`, `Validators.minLength(n)`, and `Validators.maxLength(n)`. The names are self-explanatory, so we'll just say that we're using the first one, at least for now.

In order to use these classes, we need to add the following `import` statement at the beginning of the file:

```
import {FormBuilder, Validators} from "@angular/forms";
```

As we can see, there's also a `performLogin` method that we didn't implement much. We're just opening a UI alert to ensure us that everything is working so far, then bring the user back to our welcome view.

While we're here, let's take the chance to also add the `Router` component, so we'll be able to send the user somewhere right after the login. We can easily do that using the same DI technique we've already used a number of times.

This is how the `login.component.ts` will look after these changes:

```
import {Component} from "@angular/core";
import {FormBuilder, Validators} from "@angular/forms";
import {Router} from "@angular/router";

@Component({
    selector: "login",
    template: `
    <div class="login-container">
      <h2 class="form-login-heading">Login</h2>
```

```
      <form class="form-login" [ngFormModel]="loginForm"
(submit)="performLogin($event)">
        <input ngControl="username" type="text" class="form-
control" placeholder="Your username or e-mail address" required
autofocus />
        <input ngControl="password" type="password" class="form-
control" placeholder="Your password" required />
        <div class="checkbox">
          <label>
            <input type="checkbox" value="remember-me"> Remember me
          </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Sign in</button>
      </form>
    </div>
    `
})

export class LoginComponent {
    title = "Login";
    loginForm = null;

    constructor(
        private fb: FormBuilder,
        private router: Router) {
        this.loginForm = fb.group({
            username: ["", Validators.required],
            password: ["", Validators.required]
        });
    }

    performLogin(e) {
        e.preventDefault();
        alert(JSON.stringify(this.loginForm.value));
    }
}
```

Adding styles

As for the custom CSS classes, we can add them to our `Scripts/less/style.less`
file:

```
.login-container {
    max-width: 330px;
    padding: 15px;
    .form-login {
        margin: 0 0 10px 20px;
        .checkbox {
            margin-bottom: 10px;
        }
```

```
        input {
            margin-bottom: 10px;
        }
    }
}
```

Updating the root module file

Our renewed `LoginComponent` should compile just fine. However, if we try to run the app now, we would get a full-scale Angular 2 runtime error in the browser's console log:
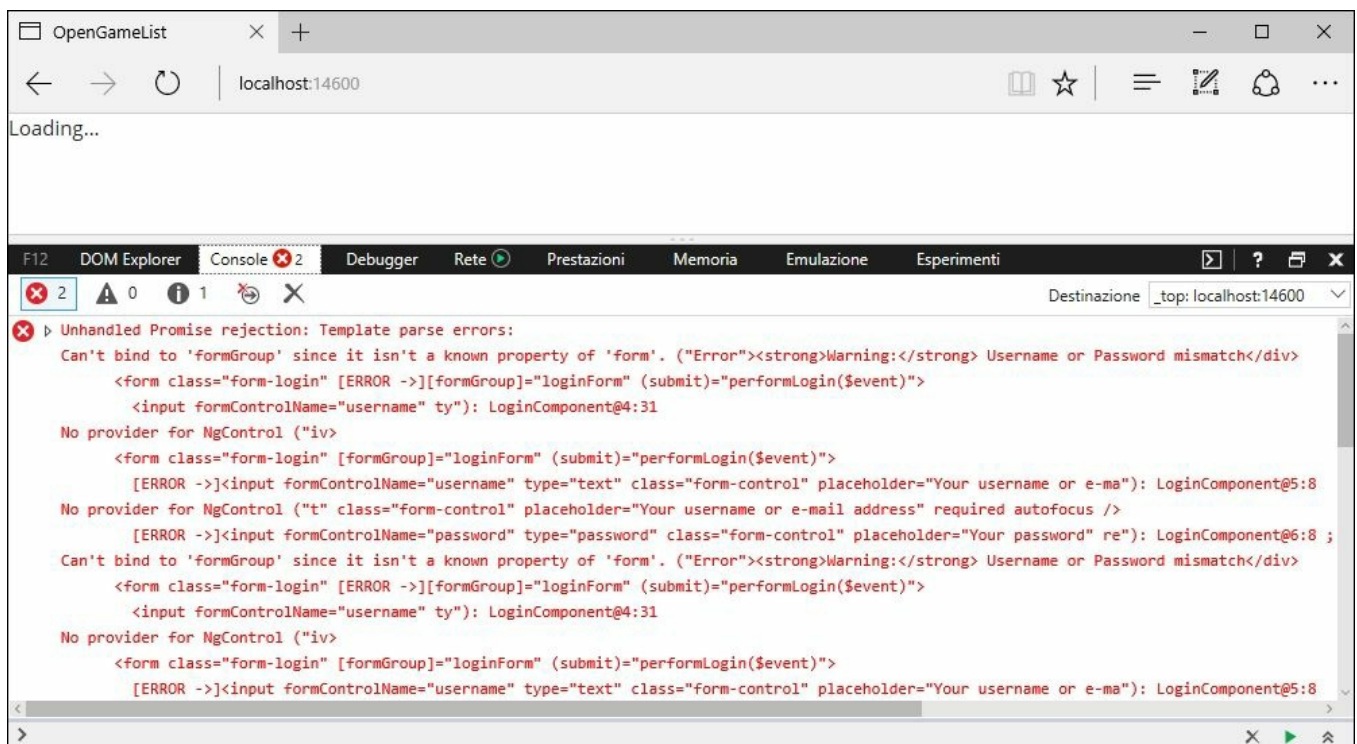
Pretty scary, isn't it?

When we see something like that in Angular 2, it usually means that we're missing a required module. That's exactly the case. In order to use reactive forms classes, we need to open our `/Scripts/app/app.module.ts` file and append `ReactiveFormsModule` to the following existing `import` statement, near the beginning of the file:

```
import {FormsModule, ReactiveFormsModule} from "@angular/forms";
```

And also add it to the `imports` array as follows:

```
imports: [
    BrowserModule,
```

```
    HttpModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule,
    AppRouting
],
```

Once done, our application will be able to run without errors.

Wait a minute...`FormsModule` has been there since Chapter 3, *Angular 2 Components and Client-Side Routing*! On top of that, we even used it to build the `ItemDetailEditComponent` form, which happens to work just fine! Why do we need `ReactiveFormsModule` now?

As a matter of fact, we don't; we could stick to the `FormsModule` and build another template-driven form just like the one we already did. As a matter of fact, since this is a tutorial application, we took the chance to use the alternative strategy provided by Angular 2 to build forms: the model-driven (or reactive) forms approach.

This clarification raises a predictable question: which one of them is better? The answer is not easy, as both techniques have their advantages. To keep it extremely simple, we can say that template-driven forms are generally simpler to pull off, but they're rather difficult to test and validate as they become complex; conversely, model-driven forms do have an harder learning curve but they usually perform better when dealing with large forms, as they allow us to unit test their whole validation logic.

Note

We won't explore these topics further, as they would take us way beyond the scope of this book. For more info regarding template-driven and model-driven forms, we strongly suggest reading the following article from the Angular 2 developers blog:

http://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/

And also check out the official Angular 2 documentation regarding forms:

https://angular.io/docs/ts/latest/guide/forms.html

UI and validation test

Let's do a quick test right now. Hit *F5* and click on the **Login** top navigation bar. We should be welcomed by something like this:
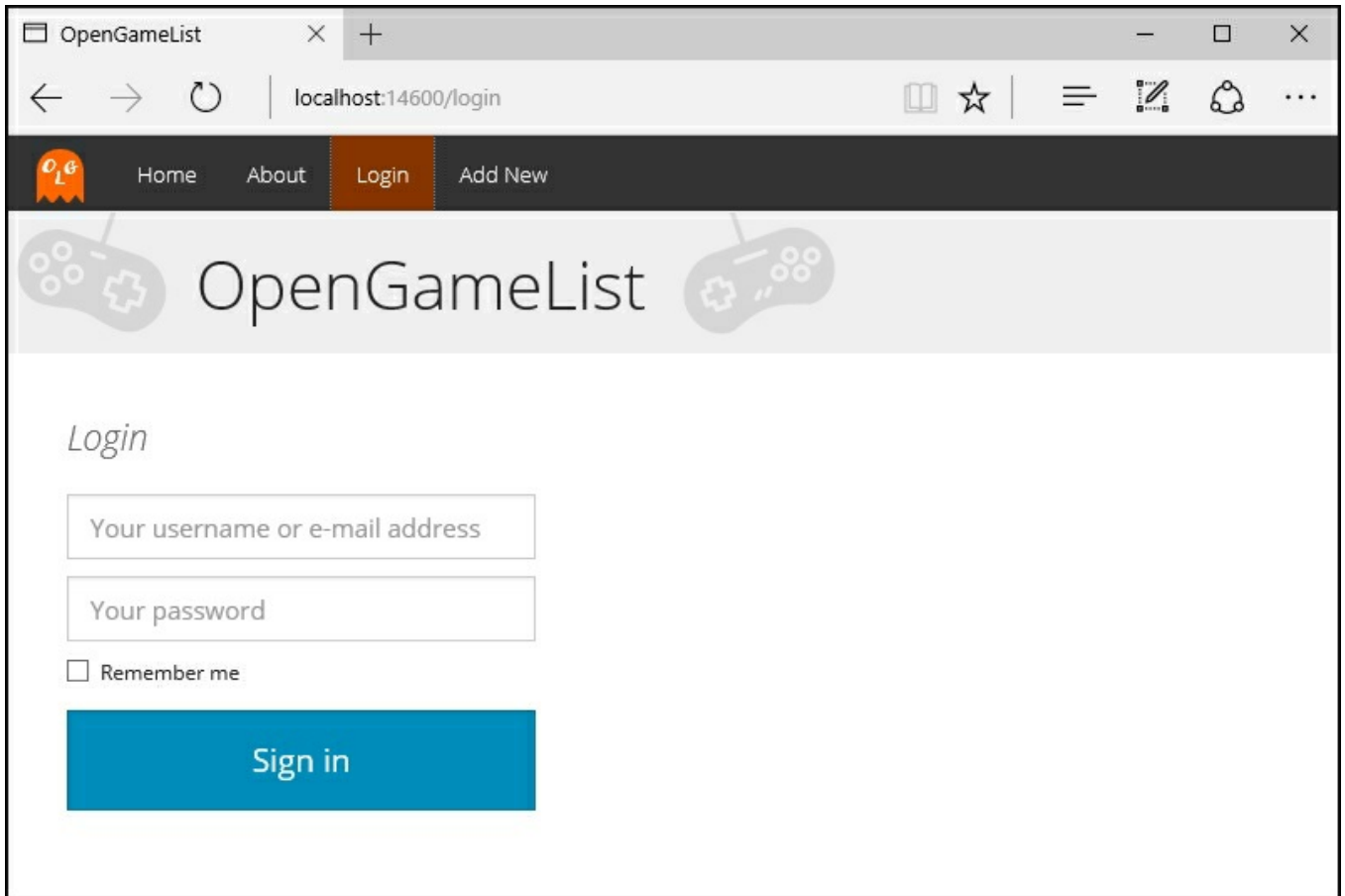
Let's now check the *validators* by hitting the **Sign in** button, leaving the input fields empty. We can see the two textboxes react accordingly, since they're both expecting a required value:
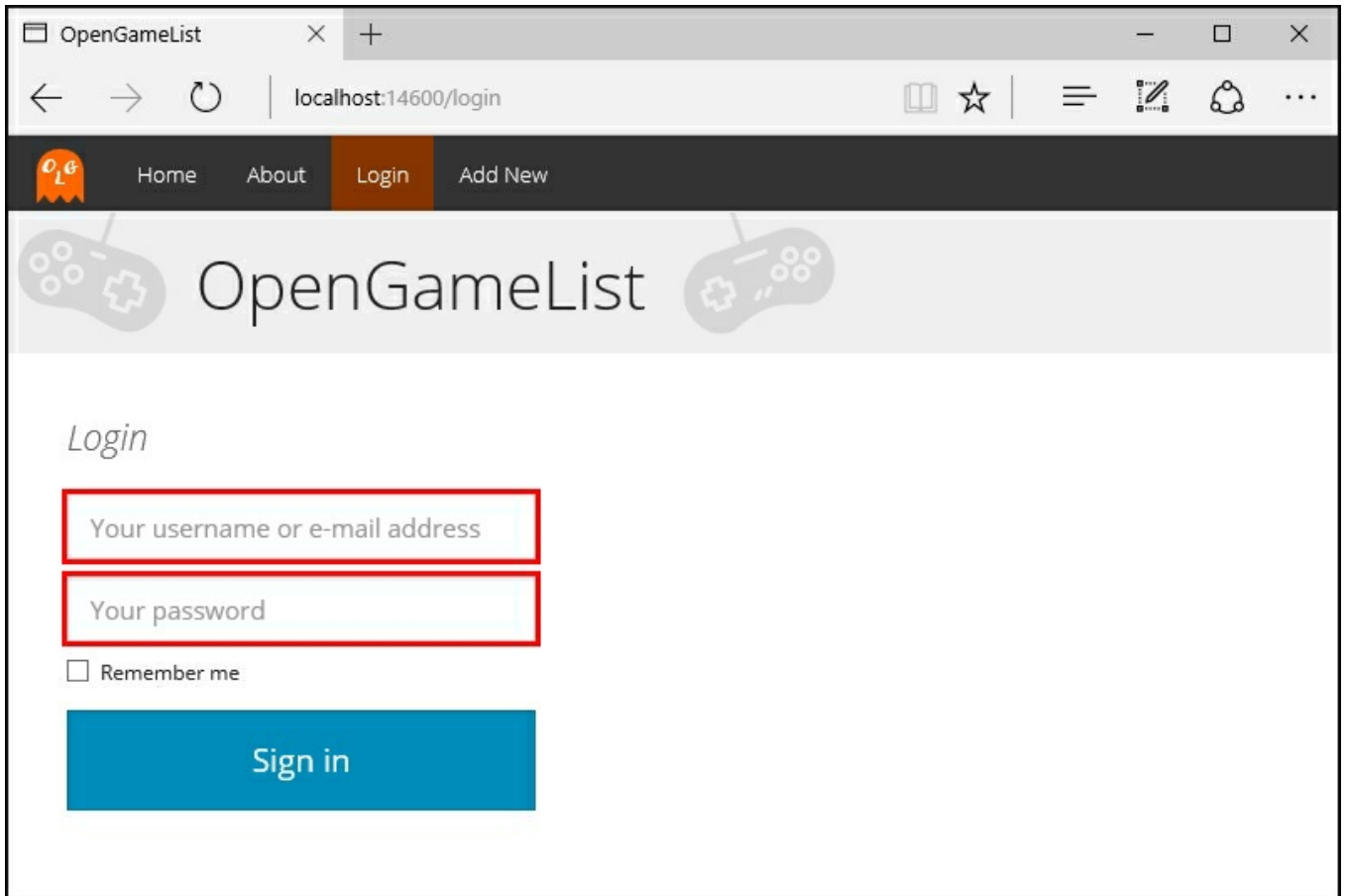
Finally, let's test the outcome JSON by filling up the input fields with some random values and pressing the **Sign in** button again:
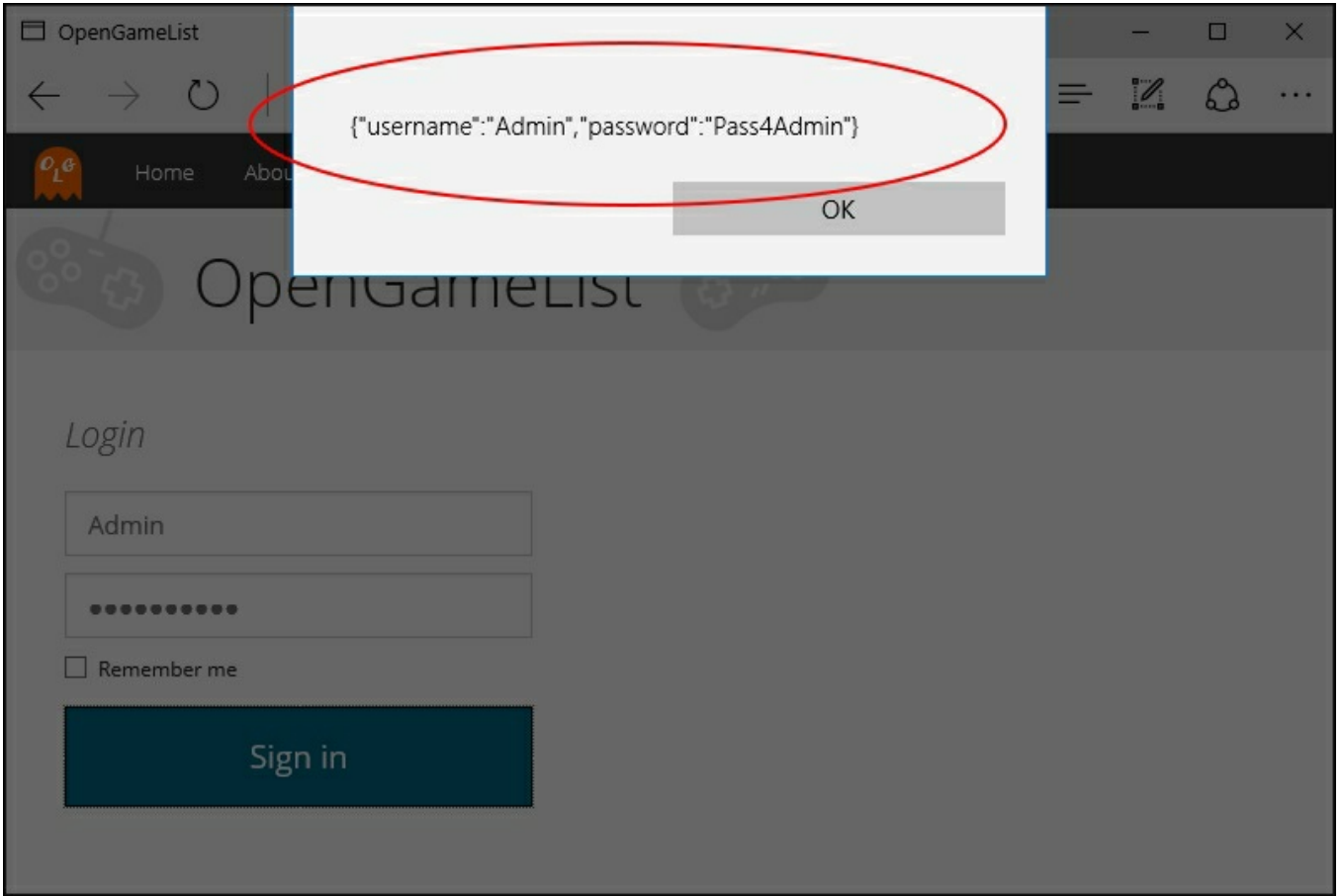
That's it. It seems that our login form is working fine.

AuthService component

Now we need to create a dedicated service to handle the login and logout operations.

Right-click on the `/Scripts/app/` folder, select **Add | New Item** and add a new `auth.service.ts` file to the project, then fill it with the following code:

```
import {Injectable, EventEmitter} from "@angular/core";
import {Http, Headers, Response, RequestOptions} from
"@angular/http";
import {Observable} from "rxjs/Observable";

@Injectable()
export class AuthService {
    authKey = "auth";

    constructor(private http: Http) {
    }

    login(username: string, password: string): any {
        var url = "api/connect/token";  // JwtProvider's LoginPath

        var data = {
            username: username,
            password: password,
            client_id: "OpenGameList",
            // required when signing up with username/password
            grant_type: "password",
            // space-separated list of scopes for which the token
is issued
            scope: "offline_access profile email"
        };

        return this.http.post(
            url,
            this.toUrlEncodedString(data),
            new RequestOptions({
                headers: new Headers({
                    "Content-Type": "application/x-www-form-
urlencoded"
                })
            }))
            .map(response => {
                var auth = response.json();
                console.log("The following auth JSON object has
been received:");
                console.log(auth);
                this.setAuth(auth);
                return auth;
            });
```

```
    }

    logout(): boolean {
        this.setAuth(null);
        return false;
    }

    // Converts a Json object to urlencoded format
    toUrlEncodedString(data: any) {
        var body = "";
        for (var key in data) {
            if (body.length) {
                body += "&";
            }
            body += key + "=";
            body += encodeURIComponent(data[key]);
        }
        return body;
    }

    // Persist auth into localStorage or removes it if a NULL
argument is given
    setAuth(auth: any): boolean {
        if (auth) {
            localStorage.setItem(this.authKey,
JSON.stringify(auth));
        }
        else {
            localStorage.removeItem(this.authKey);
        }
        return true;
    }

    // Retrieves the auth JSON object (or NULL if none)
    getAuth(): any {
        var i = localStorage.getItem(this.authKey);
        if (i) {
            return JSON.parse(i);
        }
        else {
            return null;
        }
    }

    // Returns TRUE if the user is logged in, FALSE otherwise.
    isLoggedIn(): boolean {
        return localStorage.getItem(this.authKey) != null;
    }
}
```

This code has some resemblance to the one we used in the `item.service.ts` class.

This can be expected, since both are Angular 2 service-type components used to instantiate service accessor objects, with the purpose of sending and receiving data to and from the web APIs. However, there are some key differences that might be worthy of attention:

- The content-type set for the `Login` method's `POST` request has been set to `application/x-www-form-urlencoded` instead of `application/json` to comply with the requirements set in the `JwtProvider` class.
- We store the result locally by making use of the `localStorage` object, which is part of HTML5's Web Storage API. This is a local caching object that keeps its content with no given expiration date. That's a great way to store our JWT-related JSON response, as we want to keep it even when the browser is closed. Before doing that, we choose to convert it into a `string` using `JSON.stringify`, since not all `localStorage` browser implementations can store JSON-type objects flawlessly.

Tip

Alternatively, in case we were to delete the token whenever the user closes the specific browser tab, we could use the `sessionStorage` object, which stores data only until the currently active session ends.

It's worth noting that we defined three methods to handle `localStorage`: `setAuth()`, `getAuth()`, and `isLoggedIn()`. The first one is in charge of `insert`, `update`, and `delete` operations; the second will retrieve the `auth` JSON object (if any); and the last one can be used to check whether the current user is authenticated or not, without having to `JSON.parse` it.

Updating the AppModule

In order to test our new `AuthService` component, we need to hook it up to the `AppModule` and to the `LoginComponent` we created a short while ago.

Open the `/Scripts/app/app.module.ts` file and add the following `import` line between the `AppRouting` and `ItemService` lines:

```
import {AppRouting} from "./app.routing";
import {AuthService} from "./auth.service";
import {ItemService} from "./item.service";
```

Then scroll down to the providers array and add it there too:

```
providers: [
    AuthService,
    ItemService
],
```

Updating the LoginComponent

Once done, switch back to the `Scripts/app/login.component.ts` file and replace the content of the source code as follows (new/updated lines are highlighted):

```
import {Component} from "@angular/core";
import {FormBuilder, Validators} from "@angular/forms";
import {Router} from "@angular/router";
import {AuthService} from "./auth.service";

@Component({
    selector: "login",
    template: `
    <div class="login-container">
      <h2 class="form-login-heading">Login</h2>
      <div class="alert alert-danger" role="alert"
*ngIf="loginError"><strong>Warning:</strong> Username or Password
mismatch</div>
      <form class="form-login" [formGroup]="loginForm"
(submit)="performLogin($event)">
        <input formControlName="username" type="text" class="form-
control" placeholder="Your username or e-mail address" required
autofocus />
        <input formControlName="password" type="password"
class="form-control" placeholder="Your password" required />
        <div class="checkbox">
          <label>
            <input type="checkbox" value="remember-me">
            Remember me
          </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Sign in</button>
      </form>
    </div>
    `
})

export class LoginComponent {
    title = "Login";
    loginForm = null;
    loginError = false;

    constructor(
        private fb: FormBuilder,

private router: Router,
  private authService: AuthService) {
        this.loginForm = fb.group({
            username: ["", Validators.required],
            password: ["", Validators.required]
```

```
        });
    }

    performLogin(e) {
        e.preventDefault();
        var username = this.loginForm.value.username;

        var password = this.loginForm.value.password;

        this.authService.login(username, password)

            .subscribe((data) => {

                // login successful

                this.loginError = false;

                var auth = this.authService.getAuth();

                alert("Our Token is: " + auth.access_token);

                this.router.navigate([""]);

            },

            (err) => {

                console.log(err);

                // login failure

                this.loginError = true;

            });
    }
}
```

What we did here is pretty straightforward:

- We included the import reference for our `AuthService` component and added it to the constructor, so we can have it available using DI.
- We added a new `loginError` local variable that will reflect the outcome of the last login attempt.
- We added a `<div>` element acting as an alert, to be shown whenever the `loginError` becomes `true`.
- We modified the `performLogin` method to make it send the username and password values to the `AuthService` component's `login` method, so it can perform the following tasks:
  - Issue an HTTP request to the `JwtProvider` middleware

- Receive a valid JWT accordingly and persist it into the `localStorage` object cache
- Return `true` in case of success or `false` in case of failure
- If everything goes as expected, we'll be shown a confirmation popup alert and route the user back to the welcome view; otherwise, we'll show the wrong username or password alert above the form

Login test

Let's run a quick test to see whether everything is working as expected. Hit *F5*, then navigate through the login view using the top navigation menu. Once there, fill in the login form with some incorrect data to test the **Wrong Username or Password** alert and left-click on the **Sign in** button:
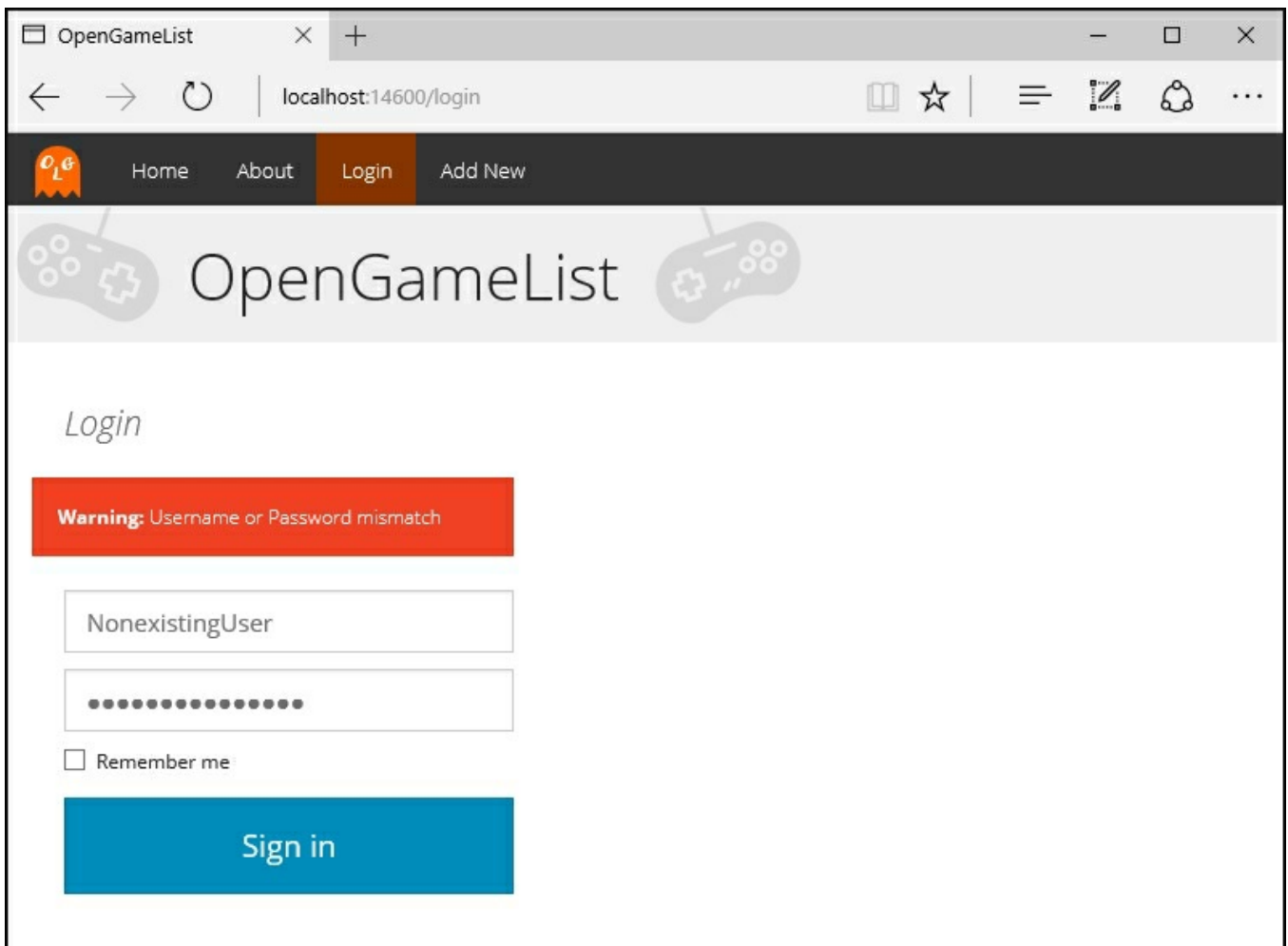
Now, let's test a successful login attempt by filling in the form again, this time using the actual Admin user credentials as defined within the `DbSeeder` class:

- E-mail: **Admin**

- Password: **Pass4Admin**

Then, left-click on the **Sign in** button.

If everything has been set up properly, we should receive the following response:
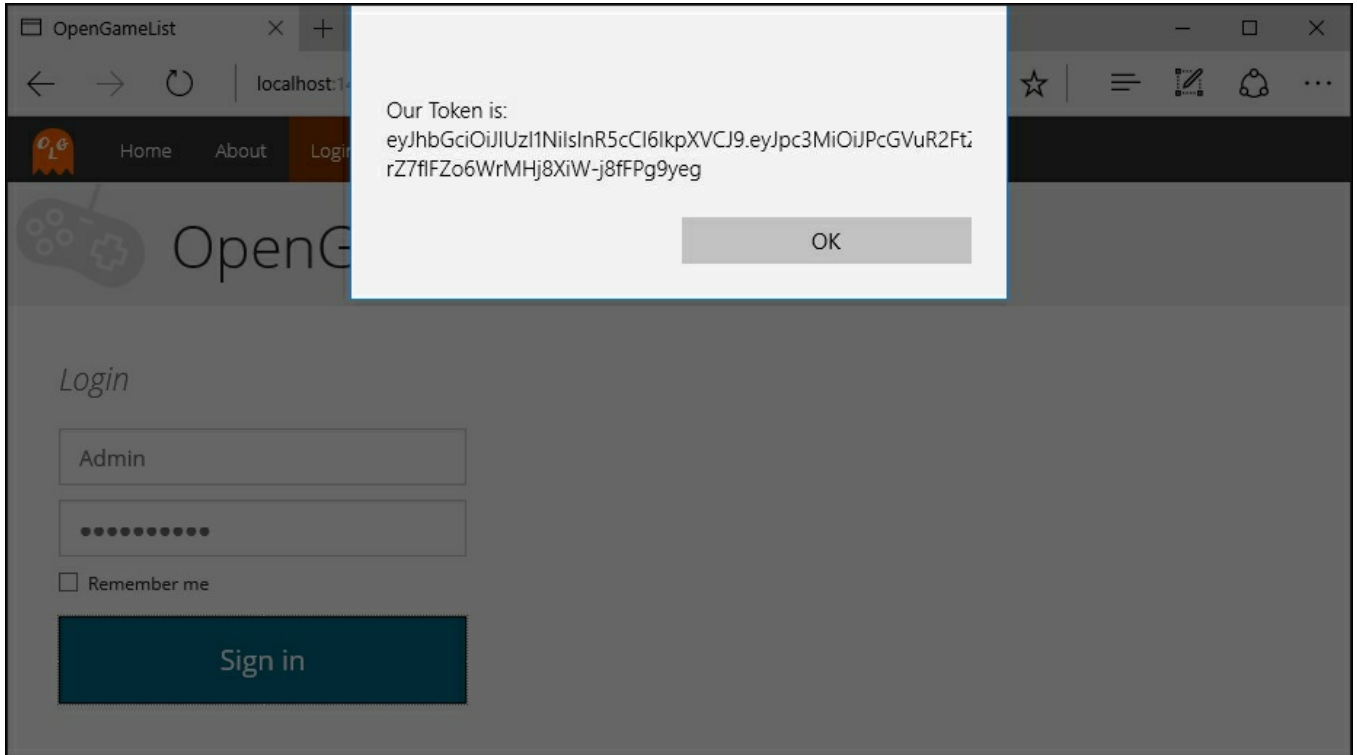
If we see something like this, it means that our `JwtProvider` works!

All we need to do now is to find a way to put that token inside the `headers` of all our subsequent requests, so we can check the token validation as well and complete our authentication cycle.

AuthHttp wrapper

A rather easy way to do that with Angular 2 is create a wrapper class that will internally use the standard `Http` component right after having it configured to suit our needs.

Right-click on the `/Scripts/app/` folder, then select **Add** | **New Item**. Add a new `auth.http.ts` file to the project and fill it with the following code:

```
import {Injectable} from '@angular/core';
import {Http, Headers} from '@angular/http';

@Injectable()
export class AuthHttp {
    http = null;
    authKey = "auth";

    constructor(http: Http) {
        this.http = http;
    }

    get(url, opts = {}) {
        this.configureAuth(opts);
        return this.http.get(url, opts);
    }

    post(url, data, opts = {}) {
        this.configureAuth(opts);
        return this.http.post(url, data, opts);
    }

    put(url, data, opts = {}) {
        this.configureAuth(opts);
        return this.http.put(url, data, opts);
    }

    delete(url, opts = {}) {
        this.configureAuth(opts);
        return this.http.delete(url, opts);
    }

    configureAuth(opts: any) {
        var i = localStorage.getItem(this.authKey);
        if (i != null) {
            var auth = JSON.parse(i);
            console.log(auth);
            if (auth.access_token != null) {
                if (opts.headers == null) {
                    opts.headers = new Headers();
                }
```

```
                opts.headers.set("Authorization", `Bearer
${auth.access_token}`);
            }
        }
    }
}
```

There's not much to say here, it's just a wrapper that calls the `configureAuth`
method internally to add the JWT token stored in the browser's `localStorage`, if
any, to each request's `headers`.

Since we'll be using the `AuthHttp` wrapper anywhere in our application, the first
thing we need to do is add it to the application's `root` module, just like we did with
the `AuthService` a short while ago. Open the `Scripts/app/app.module.ts` file
and add the usual import line between `AppRouting` and `AuthService`:

```
import {AppRouting} from "./app.routing";
import {AuthHttp} from "./auth.http";
import {AuthService} from "./auth.service";
```

And also add it to the `providers` array as follows:

```
providers: [
    AuthHttp,
    AuthService,
    ItemService
],
```

Now we can update each and every `Http` reference included in our other Angular 2
files and replace them with `AuthHttp`. As we can easily guess, the affected
components are the two service classes we're using to connect through the web API
interface: `auth.service.ts` and `item.service.ts`.

For both of them, we need to add the following line at the beginning of the file:

```
import {AuthHttp} from "./auth.http";
```

And change the constructor parameters in the following way:

```
constructor(private http: AuthHttp) {
```

Adding authorization rules

It's time to see whether our manual JWT-based `auth` implementation is working as expected. Before doing that, though, we need to define some testable navigation patterns that will allow us to differentiate the logged-in user from the anonymous one. It's actually easy to do that, since we already have some content that should be made accessible to authenticated users only. We need to handle them on the client side and also on the server side.

Adapting the client

Let's start by updating the main menu navigation bar. Open the `Scripts/app/app.component.ts` file and add the following `import` reference near the top:

```
import {AuthService} from "./auth.service";
```

Right after that, change the `template` section in the following way (new/updated lines are highlighted):

```
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container-fluid">
        <input type="checkbox" id="navbar-toggle-cbox">
        <div class="navbar-header">
            <label for="navbar-toggle-cbox" class="navbar-toggle
collapsed" data-toggle="collapse" data-target="#navbar" aria-
expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </label>
            <a class="navbar-brand" href="javascript:void(0)">
                <img alt="logo" src="/img/logo.svg" />
            </a>
        </div>
        <div class="collapse navbar-collapse" id="navbar">
            <ul class="nav navbar-nav">
                <li [class.active]="isActive([''])">
                    <a class="home" [routerLink]="['']">Home</a>
                </li>
                <li [class.active]="isActive(['about'])">
                    <a class="about" [routerLink]="
['about']">About</a>
                </li>
                <li *ngIf="!authService.isLoggedIn()"
                [class.active]="isActive(['login'])">
```

```
                <a class="login" [routerLink]="
['login']">Login</a>

            </li>

        <li *ngIf="authService.isLoggedIn()">

            <a class="logout" href="javascript:void(0)"
            (click)="logout()">Logout</a>

        </li>

        <li *ngIf="authService.isLoggedIn()"
        [class.active]="isActive(['item/edit', 0])">

            <a class="add" [routerLink]="['item/edit',
0]">Add New</a>

        </li>
        </ul>
    </div>
    </div>
</nav>
<h1 class="header">{{title}}</h1>
<div class="main-container">
    <router-outlet></router-outlet>
</div>
```

What we did here is pretty easy to understand:

- We added a `ngIf` built-in directive to the **Login** menu element, since we don't want it to appear if the user is already logged in.
- We also added another **Logout** menu element with similar yet opposing behavior, as we don't want it to be seen if the user is not logged in. Clicking on this element will trigger the `logout()` method, which we'll be adding shortly.
- We added another `ngIf` condition to the **New Item** menu element, as it should be seen by logged-in users only.

In order to use the `authService` object, we also need to instantiate it through *dependency injection* within the class constructor, which is another thing we have to change (new/updated lines highlighted):

```
constructor(public router: Router, public authService: AuthService)
{ }
```

Finally, we need to implement that `logout()` method we talked about earlier:

```
logout(): boolean {
    // logs out the user, then redirects him to Welcome View.
```

```
    if (this.authService.logout()) {
        this.router.navigate([""]);
    }
    return false;
}
```

Nothing odd here, just a standard logout and redirect behavior to adopt when the user chooses to perform a `logout`.

The changes we applied to the `AppComponent` template should also be performed in the `ItemDetailViewComponent` templates as well. Open `Scripts/app/item-detail-view.component.ts` and add the import line:

```
import {AuthService} from "./auth.service";
```

Then move to the constructor and add the `AuthService` reference there for DI (new code highlighted):

```
constructor(
    private authService: AuthService,
    private itemService: ItemService,
    private router: Router,
    private activatedRoute: ActivatedRoute) { }
```

And finally, update the `template` section accordingly, using the same `ngIf` built-in directive we used before to show/hide the **Edit** tab accordingly to the current user's logged in status:

```
<li *ngIf="authService.isLoggedIn()" role="presentation">
    <a href="javascript:void(0)"
(click)="onItemDetailEdit(item)">Edit</a>
</li>
```

Testing the client

Let's hit *F5* and see whether everything is working as it should. We should start as anonymous users and see something like this:
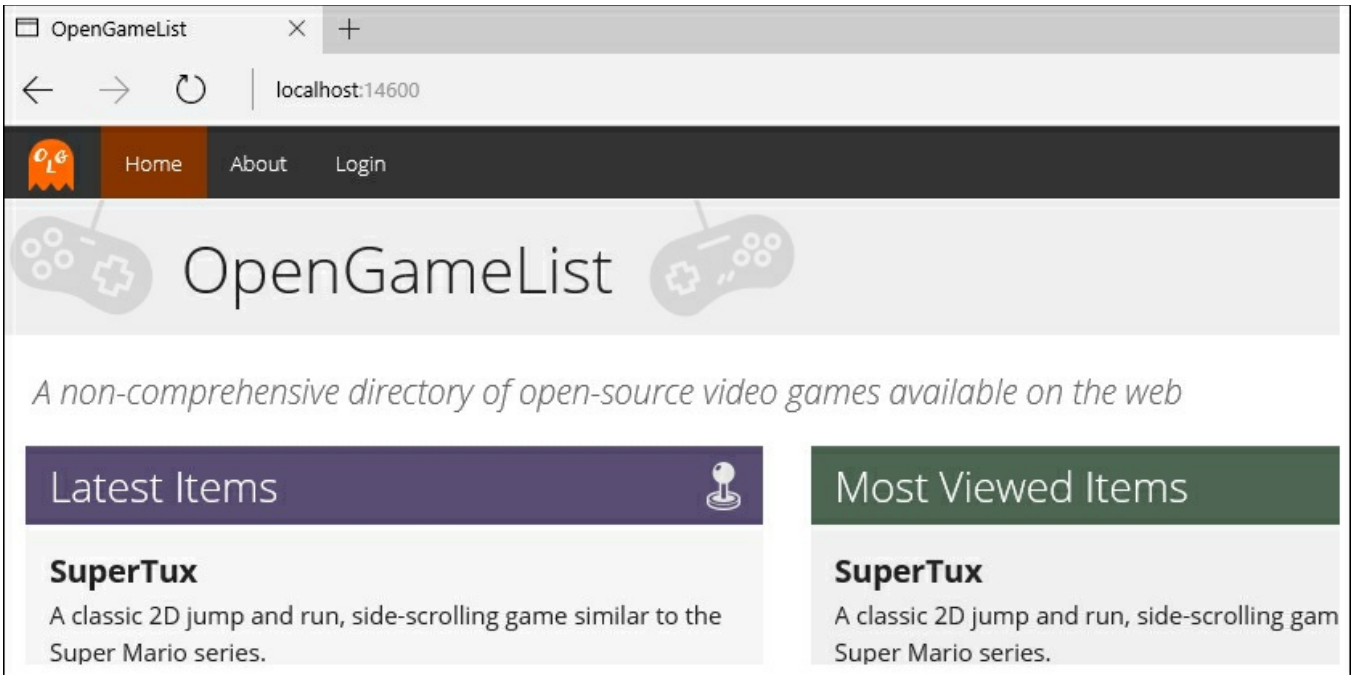
We can see that the **New Item** menu element is gone. That's expected; we're not logged in, so we shouldn't be able to add a new item.

From there, we can click the **Login** menu element and be brought to the login view, where we can input the admin credentials (`admin`/`pass4admin`, in case we forgot). As soon as we hit the **Sign In** button, we will be routed back to the welcome view, where we should be greeted by something like the following screenshot:
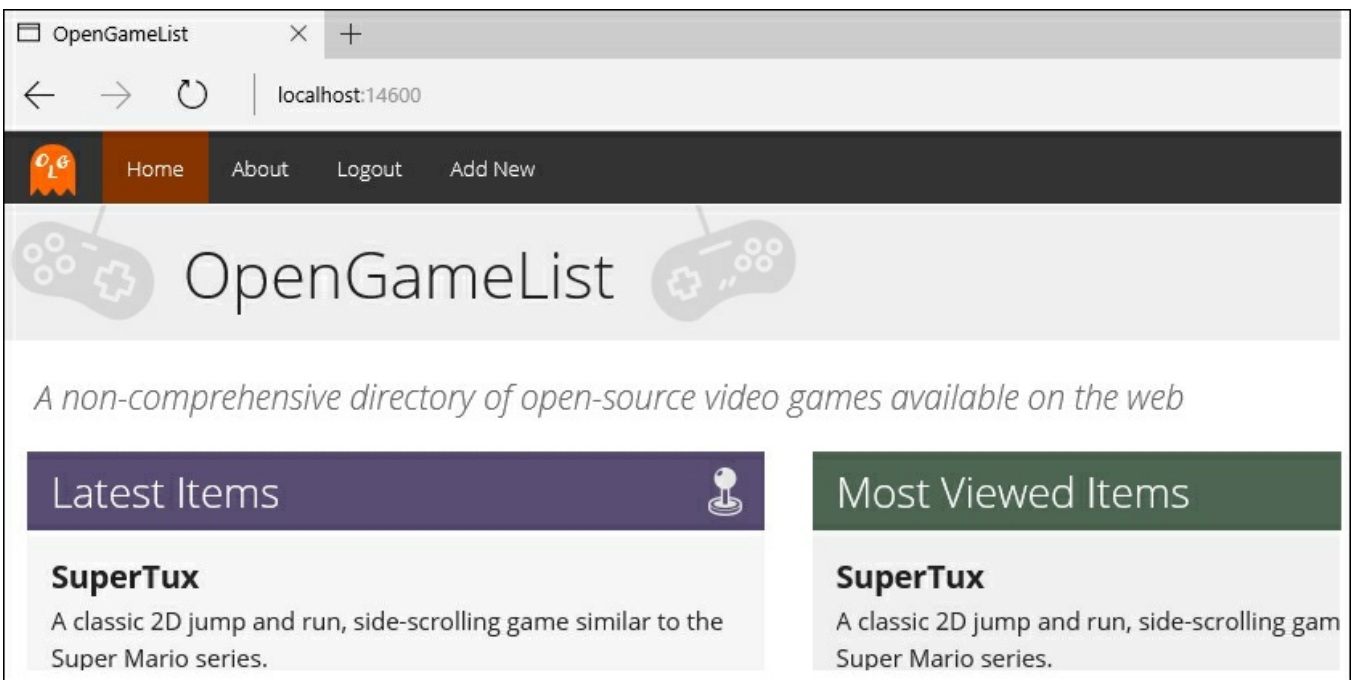
The **Login** menu element is gone, replaced by **Logout** and **Add New**. We can then click on **Logout** and see both of them replaced by the former again.

So far, so good. However, we're not done with the client yet. These modifications prevent the user from clicking some links they're not allowed to see, yet they are unable to stop the user from going to their given destinations. For example, the user could manually input the routes within the browser's navigation bar and go to the login view while being already logged in, or even worse access the add/edit item view despite being anonymous.

In order to avoid that, we can add a login status check within the `login.component.ts` constructor (new lines highlighted):

```
constructor(
    private fb: FormBuilder,
    private router: Router,
    private authService: AuthService) {
    if (this.authService.isLoggedIn()) {

        this.router.navigate([""]);

    }
    this.loginForm = fb.group({
        username: ["", Validators.required],
        password: ["", Validators.required]
    });
}
```

Also add it to the `ngOnInit` startup method within the `item-detail-edit.component.ts` file:

```
ngOnInit() {
    if (!this.authService.isLoggedIn()) {

        this.router.navigate([""]);

    }
    var id = +this.activatedRoute.snapshot.params["id"];
    if (id) {
        this.itemService.get(id).subscribe(
            item => this.item = item
        );
    }
    else if (id === 0) {
        console.log("id is 0: adding a new item...");
        this.item = new Item(0, "New Item", null);
    }
    else {
```

```
        console.log("Invalid id: routing back to home...");
        this.router.navigate([""]);
    }
}
```

Doing this will also require adding the corresponding `import` reference line near the topmost section of the `item-detail-edit.component.ts` file:

```
import {AuthService} from "./auth.service";
```

And the DI injection in the constructor method:

```
constructor(
    private authService: AuthService,
    private itemService: ItemService,
    private router: Router,
    private activatedRoute: ActivatedRoute) { }
```

That way, any unauthorized user will be bounced back whenever they try to manually hack our route mechanism by issuing a direct request to these views.

Protecting the server

Now that our client is more or less ready, it's time to shield our web API interface from unauthorized requests as well. We can easily do that using the `[Authorize]` attribute, which can be used to restrict access to any controller and/or controller method we don't want to open to unauthorized access.

To implement the required authorization behavior, it could be wise to use it on the `Add`, `Update`, and `Delete` methods of our `ItemsController` class (new lines are highlighted):

```
[HttpPost()]
[Authorize]
public IActionResult Add([FromBody]ItemViewModel ivm)
{
    [...]
}


[HttpPut("{id}")]
[Authorize]
public IActionResult Update(int id, [FromBody]ItemViewModel ivm)
{
    [...]
}


[HttpDelete("{id}")]
[Authorize]
public IActionResult Delete(int id)
```

```
{
    [...]
}
```

In order to use the `[Authorize]` attribute, we also need to declare the following namespace reference at the beginning of the file:

```
using Microsoft.AspNetCore.Authorization;
```

Now these methods are protected against unauthorized access, as they will accept only requests coming from logged-in users/clients with a valid `JWT` token. Those who don't have it will receive a `401 - Unauthorized` HTTP error response.

Retrieving the user ID

Before closing the `ItemsController` class file, we should take the chance to remove the `item.UserId` value override we defined back in Chapter 5, *Persisting Changes*, when we had no authentication mechanism in place:

```
// TODO: replace the following with the current user's id when
authentication will be available.
item.UserId = DbContext.Users.Where(u => u.UserName ==
"Admin").FirstOrDefault().Id;
```

Now that we're working with real users, we definitely have to remove this ugly workaround and find a way to retrieve the actual user ID. Luckily enough, when we implemented our very own JWT provider earlier, we did actually put it in the claims JWT token (`JwtProvider` class, `CreateToken` method):

```
new Claim(JwtRegisteredClaimNames.Sub, user.Id),
```

This means that we can retrieve it in the following way (updated code is highlighted):

```
item.UserId = User.FindFirst(ClaimTypes.NameIdentifier).Value;>
```

Let's perform this change and move on.
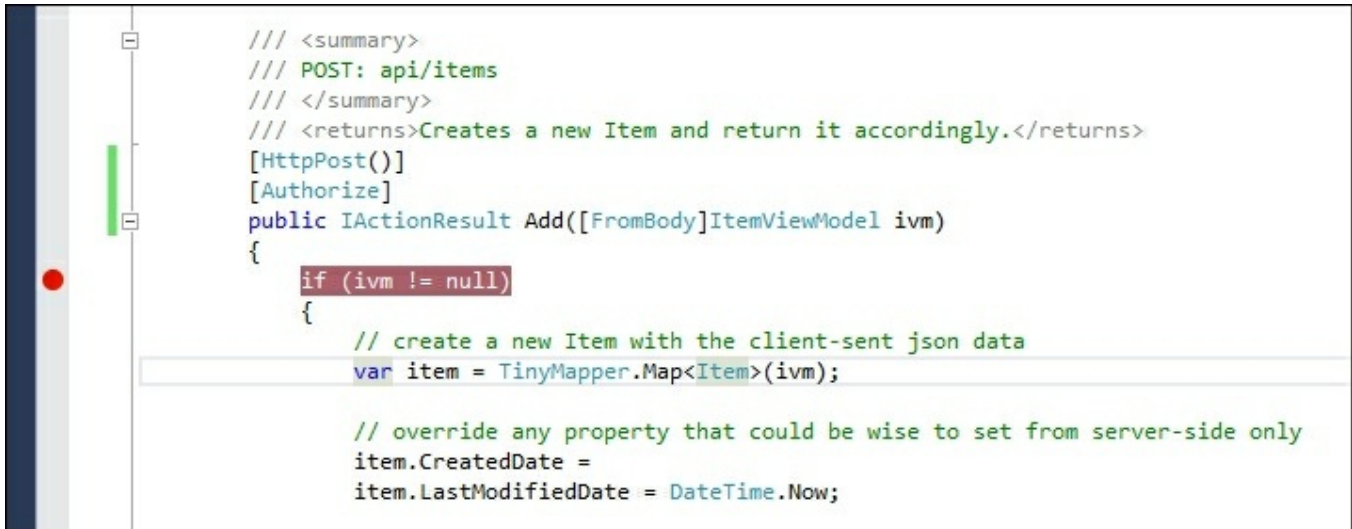
Note

This minor update should be enough for now. However, it won't work when dealing with external OpenId and/or OAuth2 providers, as they will put their own data in these claims. Retrieving our local `UserId` in such scenarios will require some additional work, such as querying a dedicated lookup table. We'll see more about this during Chapter 8, *Third-Party Authentication and External Providers*.

Authorization test

Before going further, it's definitely time to perform a client/server interaction test to ensure that our authorization pattern is working as expected.

From the Visual Studio source code editing interface, we can put a breakpoint right below the `ItemsControllerAdd` method:

Once done, we can hit *F5*, navigate from the welcome view to the login view, and authenticate ourselves. Right after that, we'll be able to click upon the **Add New** menu element.

From there, we can fill in the form with some random text and click on the **Save** button. The form will consequently call the `Add` method of `ItemsController`, hopefully triggering our breakpoint.

Open a **Watch** window (**Debug** | **Windows** | **Watch** | **Watch 1**) and check the `HttpContext.User.Identity.IsAuthenticated` property value:

```
/// <summary>
/// POST: api/items
/// </summary>
/// <returns>Creates a new Item and return it accordingly.</returns>
[HttpPost()]
[Authorize]
public IActionResult Add([FromBody]ItemViewModel ivm)
{
    if (ivm != null)
    {
        // create a new
        var item = TinyN

        // override any
        item.CreatedDate
        item.LastModifie

        // set a suitabl
        // NOTE: we'll
        item.UserId = "

        // add the new item
        DbContext.Items.Add(item);
```

Watch 1	
Name	Value
HttpContext.User.Identity.IsAuthenticated	true

If it's `true`, it means that we've been successfully authenticated. That shouldn't be surprising, since our request already managed to get inside a method protected by an `[Authorize]` attribute.

Suggested topics

Authentication, authorization, HTTP protocol, Secure Socket Layer, Session State Management, Indirection, Single Sign-On, Azure AD Authentication Library (ADAL), AspNetCore Identity, OpenID, OAuth, Conversion Rate, Code-First Migrations, IdentityUser, Stateless, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), LocalStorage, Web Storage API, Generic Types, JWT Tokens, Claims, Refresh Tokens, Sliding Sessions.

Summary

At the start of this chapter, we introduced the concepts of authentication and authorization, acknowledging the fact that most applications, including ours, do require a mechanism to properly handle authenticated and non-authenticated clients, as well as authorized and unauthorized requests.

We took some time to properly understand the similarities and differences between authentication and authorization, as well as the pros and cons of handling these tasks using our own internal provider or delegating them to third-party providers such as Google, Facebook, and Twitter. We also found out that, luckily enough, the `AspNetCore.Identity` framework can be configured to achieve the best of both worlds. To be able to use it we added the required packages to our project and did what was needed to properly configure them, such as performing some changes in our `ApplicationUser` and `ApplicationDbContext` classes and then adding a new `EntityFrameworkCore` migration to update our database accordingly.

We briefly enumerated the various web-based authentication methods available nowadays: sessions, tokens, signatures, and two-factor strategies of various sorts. After careful consideration, we chose to implement a token-based approach using Json Web Token (JWT), a solid and well-known standard for native web applications.

Implementing JWT within our application took us some time, as we had to take care of a number of steps: writing our own `JwtProvider` to generate the tokens; adding them to the HTTP request pipeline, together with the AspNetCore-native `JwtBearerMiddleware` needed to validate them; and finally, moving to our Angular 2 client app, creating a `login` form, an `AuthService`, and an `AuthHttp` wrapper class to handle everything on the client side.

Right after that, we implemented the required server-side and client-side authorization rules to protect some of our application views, routes, and APIs from unauthorized access.

Chapter 8. Third-Party Authentication and External Providers

The hand-made authentication and authorization flow we put together in [Chapter 7](#), *Authentication and Authorization*, is pretty much working. However, it lacks some very important features required for a production-ready environment, the most important ones being token expiration, token refresh, and sliding session support. Implementing them from scratch won't be easy and would take us far from the scope of this book. Luckily enough, there are a number of third-party packages that already went down that route with great results. Among them, the most promising one seems to be `OpenIddict`, an open-source project featuring an OAuth2/OpenID Connect provider based on **ASP.NET Core Identity** and **AspNet.Security.OpenIdConnect.Server** (also known as **ASOS**).

In this chapter, we'll learn how to properly install and configure it, as well as implement support for external authentication/authorization providers such as Google, Facebook, and Twitter.

Introducing OpenID connect servers

As the name suggests, `OpenIddict` is basically a (mostly) full-featured OAuth2/OpenID connect server that can be easily plugged into any ASP.NET Core application.

Note

In case we need a quick recap regarding OAuth2 and/or OpenID connect before going further, we can check a lot of useful info by visiting the following URLs:

- https://tools.ietf.org/html/rfc6749
- http://openid.net/connect/faq/

The main purpose of OAuth2/OpenID connect server interfaces such as `OpenIddict` is to support a wide amount of modern authentication standards, including, yet not limited to, JWT. As a matter of fact, they're not that different from the `JwtServer` we put together during the previous chapter, except that they come bundled with a lot of additional features we're still missing, such as the aforementioned token refresh and sliding expiration, thus releasing us from the need to manually implement them.

Choosing between replacing our custom `JwtProvider` with `OpenIddict` (or other similar third-party solutions) or not is mostly a matter of personal choice, depending on whether we feel more like coding our very own authentication layer or switching to a community-oriented approach. It won't even impact what we'll do in this chapter, as it will work with both.

Tip

If you choose to keep using your own `JwtProvider`, you might want to skip the next paragraph entirely.

Installing OpenIddict

Let's start with the good news! Adding `OpenIddict` won't be hard, as we already did most of the required work. It's also worth noting that doing it is not a required step, as our hand-made alternative is working perfectly fine and will be fully compatible with the rest of the chapters.

Note

The `OpenIddict` project is actively maintained by Kévin Chalet and licensed under the Apache License: http://www.apache.org/licenses/LICENSE-2.0.html

This means that we are able to use, modify, and distribute it freely. For a better overview of the product, we strongly suggest taking a look at the official GitHub project page at the following URL:

https://github.com/openiddict/openiddict-core

Adding MyGet to the package sources

The first thing we need to do is to add a new package source to the Visual Studio **NuGet package manager**:

Note

At the time of writing, this is a required step, as`OpenIddict` has not been released to the official NuGet package archive. This will most likely change in the future, to the point that it could be useful to perform a quick search through the official packages before doing the following steps.

1. Right-click on the **OpenGameListWebApp** project node.
2. Select **Manage NuGet Packages** and open the pop-up window.
3. Locate the **Package Source** drop-down list to the upper-right corner and set it to **All**, then click on the cogwheel icon to its immediate right to access the **Options** modal window.
4. Left-click on the plus icon near the upper-right corner and add the following package source:

Name : `myget.org`

Source: `https://www.myget.org/F/aspnet-contrib/api/v3/index.json`

Once done, click the Update button. Right after doing that, we should be able to see something very similar to the following screenshot:
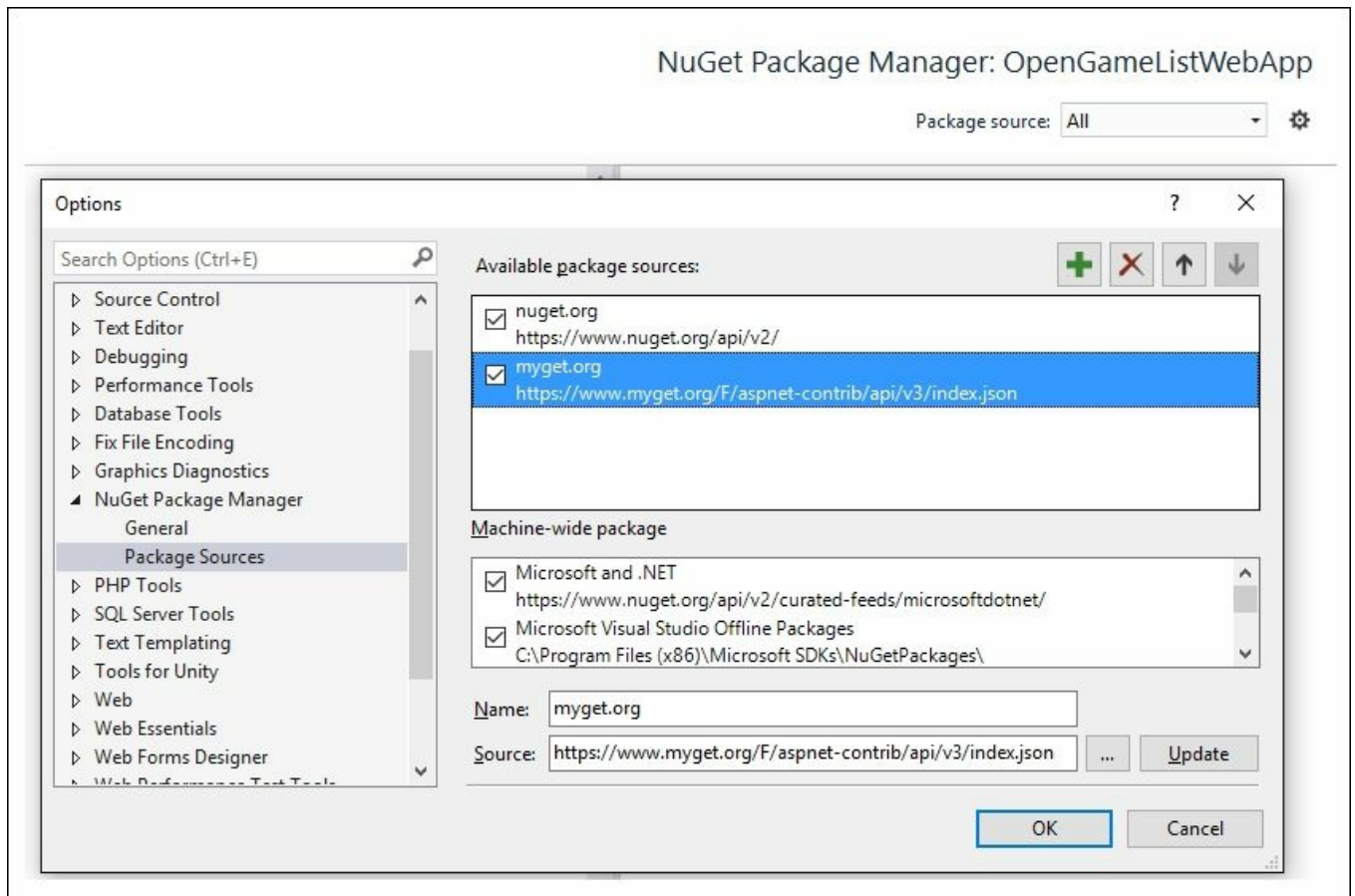
Ensure that the checkbox to the left of the new package source is checked, then click **Ok** to close the modal window.

Installing the OpenIddict package

Now we can issue a search for **OpenIddict** and be sure to actually find something relevant:
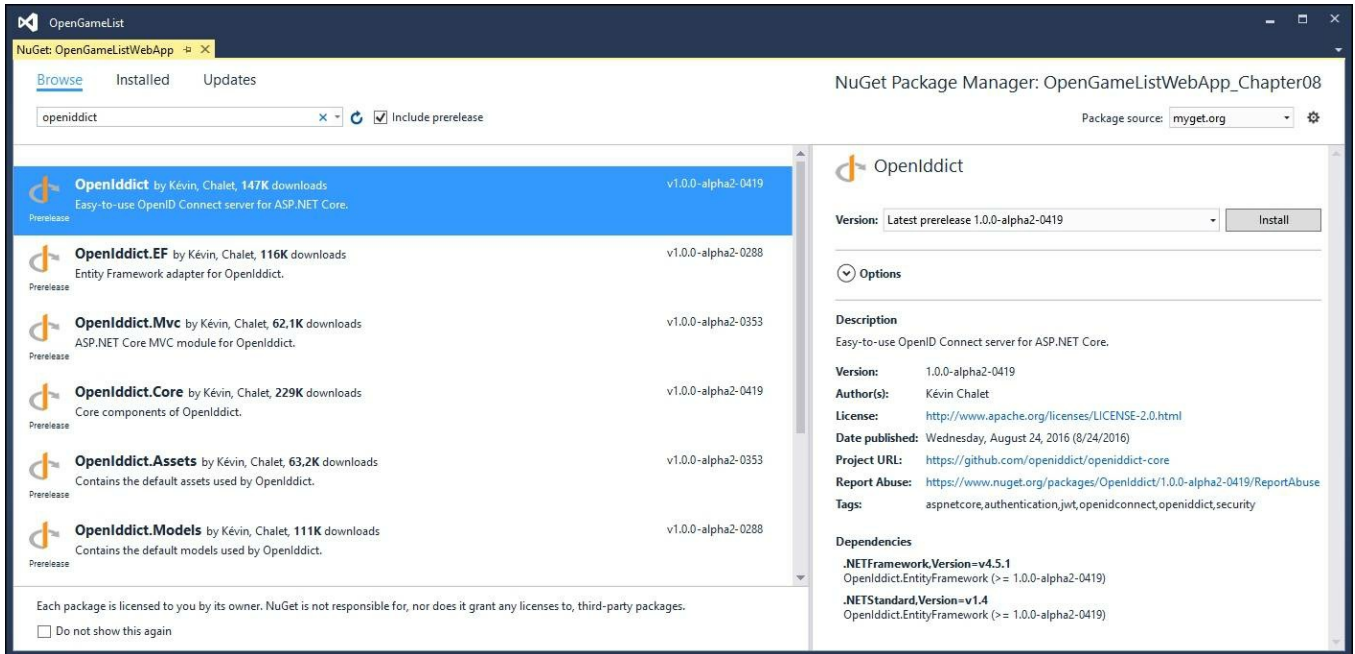
We're looking for the one called **OpenIddict**, which should be the first entry.

As always, ensure that the **Include prerelease** checkbox is checked to include the latest versions of the packages. We'll be using the **1.0.0-alpha2-0419** build, since it's the most recent one at the time of writing.

Tip

IMPORTANT It's worth noting that, being still in alpha stage, the upcoming releases of **OpenIddict** might bring major changes to the interface, thus requiring some changes/updates to the source code we'll be using throughout this chapter.

Updating ApplicationUser and ApplicationDbContext

This step might look familiar, as we already did something very similar when we added `AspNetCore.Identity`. We basically need to change the base classes, replacing `IdentityUser` and `IdentityDbContext` with `OpenIddictUser` and `OpenIddictDbContext`.

These are the relevant code changes (new/updated code highlighted):

In the `/Data/Users/ApplicationUser.cs` file:

```
public class ApplicationUser: OpenIddictUser
```

In the `/Data/ApplicationDbContext.cs` file:

```
public class ApplicationDbContext :
OpenIddictDbContext<ApplicationUser>
```

We also need to declare the following namespace in both of these files:

```
using OpenIddict;
```

This concludes the `OpenIddict` package implementation. However, we still need to update our database accordingly.

Note

We're about to venture again into migrations territory, so it might be wise to issue a full rebuild of our project to ensure that our code will compile without errors.

Adding a new migration

The `OpenIddict` module comes with a set of built-in entities that need to be added to our existing database. Luckily enough, since we're using code-first migrations, this is not a big deal.

Open a **PowerShell** command prompt and navigate through the project's root folder:

`C:\Projects\OpenGameList\src\OpenGameListWebApp\`

Once there, type the following command to add the new migration:

`dotnet ef migrations add "OpenIddict" -o "Data\Migrations"`

Wait for the migration to be created, then type the following to execute it:

`dotnet ef database update`

This will create the following new tables in our local database:

```
OpenIddictApplications
OpenIddictAuthorizations
OpenIddictScopes
OpenIddictTokens
```

Tip

If the tables are not there or you run into some other issues, it could mean that EF Core is unable to handle a previously created migration. The best workaround to solve these kinds of problems is to delete the `Data\Migrations` folder and then run the preceding commands again. Alternatively, you can try the `database drop` and `database update` commands, just like we did in Chapter 7, *Authentication and Authorization*.

Updating the DbSeeder

One of the new tables, `OpenIddictApplications`, needs to be populated with a single row corresponding to our web application. The task can be easily performed by our `DbSeeder` class, which happens to do just that.

Open the `Data/DbSeeder.cs` file and add the following code to the constructor method (new lines are highlighted):

```
public async Task SeedAsync()
{
    // Create the Db if it doesn't exist
    DbContext.Database.EnsureCreated();
    // Create default Application

    if (!DbContext.Applications.Any()) CreateApplication();
    // Create default Users
    if (!DbContext.Users.Any()) await CreateUsersAsync();
    // Create default Items (if there are none) and Comments
    if (!DbContext.Items.Any()) CreateItems();
}
```

Right after that, locate the beginning of the `Seed` methods region and add the following method:

```
private void CreateApplication()
{
    DbContext.Applications.Add(new OpenIddictApplication
    {
        Id = "OpenGameList",
        DisplayName = "OpenGameList",
        RedirectUri = "/api/connect/token",
        LogoutRedirectUri = "/",
        ClientId = "OpenGameList",
        ClientSecret =
Crypto.HashPassword("1234567890_my_client_secret"),
        Type = OpenIddictConstants.ClientTypes.Public
    });
    DbContext.SaveChanges();
}
```

Also, add the following required namespaces at the beginning of the file:

```
using OpenIddict;
using CryptoHelper;
```

Moving literal values to appsettings.json

As we can see, there are a lot of literal values here. Instead of having them hanging

there, it could be wise to move them into the project's configuration file instead. Remember the `appsettings.json` file that we used back in Chapter 4, *The Data Model*, to add the database connection string? Open it, then add a new `Authentication` root key with the following content (new lines are highlighted):

```json
{
  "Authentication": {

    "OpenIddict": {

      "ApplicationId": "OpenGameList",

      "DisplayName": "OpenGameList",

      "TokenEndPoint": "/api/connect/token",

      "ClientId": "OpenGameList",

      "ClientSecret": "1234567890_my_client_secret"

    }

  },
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Data Source=
(localdb)\\MSSQLLocalDB;Initial Catalog=OpenGameList;Integrated
Security=True; MultipleActiveResultSets=True"
    }
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "StaticFiles": {
    "Headers": {
      "Cache-Control": "no-cache, no-store",
      "Pragma": "no-cache",
      "Expires": "-1"
    }
  }
}
```

We already know how to retrieve these values programmatically. As a matter of fact, we already did that a number of times, using the `Configuration` property defined within the `Startup` class. What we need to do now is to find a way to make it

available through the `DbSeeder` class as well.

The best way to achieve that is via dependency injection (DI), just like we have already done a number of times. However, in order to make it work, we need to register a generic `IConfiguration` singleton object within the application's DI system beforehand.

To do that, open the `Startup.cs` file and add the following lines at the beginning of the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add a reference to the Configuration object for DI
    services.AddSingleton<IConfiguration>(

        c => { return Configuration; }

        );
```

This will allow us to retrieve that singleton instance anywhere we want using DI.

Note

If we don't do that, we will get an `InvalidOperationException` error at runtime because the DI manager won't be able to find any suitable `IConfiguration` instances.

Let's make good use of what we just did. Open the `DbSeeder.cs` file and perform the following changes (new lines highlighted):

```
#region Private Members
private ApplicationDbContext DbContext;
private RoleManager<IdentityRole> RoleManager;
private UserManager<ApplicationUser> UserManager;

private IConfiguration Configuration;

#endregion Private Members

#region Constructor
public DbSeeder(
    ApplicationDbContext dbContext,
    RoleManager<IdentityRole> roleManager,
    UserManager<ApplicationUser> userManager,
    IConfiguration configuration)
{
    DbContext = dbContext;
    RoleManager = roleManager;
    UserManager = userManager;
```

```
    SignInManager = signInManager;
    Configuration = configuration;
}
#endregion Constructor
```

In order to use the `IConfiguration` interface, we also need to add the following namespace reference at the beginning of the file:

```
using Microsoft.Extensions.Configuration;
```

Now that we have a `IConfiguration` instance, we can easily use it to replace the literal values within the `CreateApplication` method:

```
private void CreateApplication()
{
    DbContext.Applications.Add(new OpenIddictApplication
    {
        Id =
Configuration["Authentication:OpenIddict:ApplicationId"],
        DisplayName =
Configuration["Authentication:OpenIddict:DisplayName"],
        RedirectUri =
Configuration["Authentication:OpenIddict:TokenEndPoint"],
        LogoutRedirectUri = "/",
        ClientId =
Configuration["Authentication:OpenIddict:ClientId"],
        ClientSecret =
Crypto.HashPassword(Configuration["Authentication:OpenIddict:Client
Secret"]),
        Type = OpenIddictConstants.ClientTypes.Public
    });
    DbContext.SaveChanges();
}
```

That's it.

Configuring the Startup class

Last but not least, we need to add the `OpenIddict` service and middleware to our application's `Startup` class.

Open the `Startup.cs` file, locate the `ConfigureServices` method and add the following (new lines are highlighted):

```
public void ConfigureServices(IServiceCollection services)
{
    // Add a reference to the Configuration object for DI
    services.AddSingleton<IConfiguration>(
        c => { return Configuration; }
        );

    // Add framework services.
    services.AddMvc();

    // Add EntityFramework's Identity support.
    services.AddEntityFramework();

    // Add Identity Services & Stores
    services.AddIdentity<ApplicationUser, IdentityRole>(config => {
        config.User.RequireUniqueEmail = true;
        config.Password.RequireNonAlphanumeric = false;
        config.Cookies.ApplicationCookie.AutomaticChallenge =
false;
    })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add ApplicationDbContext.
    services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration["Data:DefaultConnection:Connecti
onString"])
        );

    // Register the OpenIddict services, including the default
Entity Framework stores.

    services.AddOpenIddict<ApplicationUser, ApplicationDbContext>()

        // Integrate with EFCore

        .AddEntityFramework<ApplicationDbContext>()

        // Use Json Web Tokens (JWT)

        .UseJsonWebTokens()
```

```
        // Set a custom token endpoint (default is /connect/token)


.EnableTokenEndpoint(Configuration["Authentication:OpenIddict:Token
EndPoint"])

        // Set a custom auth endpoint (default is
/connect/authorize)

        .EnableAuthorizationEndpoint("/api/connect/authorize")

        // Allow client applications to use the grant_type=password
flow.

        .AllowPasswordFlow()

        // Enable support for both authorization & implicit flows

        .AllowAuthorizationCodeFlow()

        .AllowImplicitFlow()

        // Allow the client to refresh tokens.

        .AllowRefreshTokenFlow()

        // Disable the HTTPS requirement (not recommended in
production)

        .DisableHttpsRequirement()

        // Register a new ephemeral key for development.

        // We will register a X.509 certificate in production.

        .AddEphemeralSigningKey();

    // Add ApplicationDbContext's DbSeeder
    services.AddSingleton<DbSeeder>();
}
```

We can easily see how moving the `TokenEndPoint` value into the
`appsettings.json` file was well worth the effort, as we can easily retrieve it from
there. While we're here, we should take the chance to do the same with the
`AuthorizationEndPoint` value as well.

Open the `appsettings.json` file and add the following line right above the
`TokenEndPoint` key:

```
"AuthorizationEndPoint": "/api/connect/authorize",
```

Then go back to the `Startup.cs` file and reference it accordingly:

```
.EnableAuthorizationEndpoint(Configuration["Authentication:OpenIdd
ict:AuthorizationEndPoint"])
```

So far, so good. Let's now scroll the `Startup.cs` file down to the `Configure` method and change it like this (new/updated lines are highlighted):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory, DbSeeder dbSeeder)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
the following URL:
    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = (context) =>
        {
            // Disable caching for all static files.
            context.Context.Response.Headers["Cache-Control"] =
Configuration["StaticFiles:Headers:Cache-Control"];
            context.Context.Response.Headers["Pragma"] =
Configuration["StaticFiles:Headers:Pragma"];
            context.Context.Response.Headers["Expires"] =
Configuration["StaticFiles:Headers:Expires"];
        }
    });

    // Add a custom Jwt Provider to generate Tokens
    // app.UseJwtProvider();

    // Add OpenIddict middleware

    // Note: UseOpenIddict() must be registered after
app.UseIdentity() and the external social providers.

    app.UseOpenIddict();

    // Add the Jwt Bearer Header Authentication to validate Tokens
    app.UseJwtBearerAuthentication(new JwtBearerOptions()
    {
```

```
            AutomaticAuthenticate = true,
            AutomaticChallenge = true,
            RequireHttpsMetadata = false,

            Authority = "http://localhost:14600/",

            TokenValidationParameters = new TokenValidationParameters()
            {
                //IssuerSigningKey = JwtProvider.SecurityKey,

                //ValidateIssuerSigningKey = true,

                //ValidIssuer = JwtProvider.Issuer,
                ValidateIssuer = false,
                ValidateAudience = false
            }
        });

        // Add MVC to the pipeline
        app.UseMvc();

        // TinyMapper binding configuration
        TinyMapper.Bind<Item, ItemViewModel>();

        // Seed the Database (if needed)
        try
        {
            dbSeeder.SeedAsync().Wait();
        }
        catch (AggregateException e)
        {
            throw new Exception(e.ToString());
        }
}
```

Notice that we commented out the `JwtProvider`, together with some
`TokenValidationParameters` within the `JwtBearerAuthentication`
initialization; none of them are needed by the `OpenIddict` middleware. Conversely,
we had to specify an explicit `Authority` property value to allow the JWT bearer
middleware to download the signing key. Again, this is not something that should be
kept in literal form, as we'll need to change that value when we push everything into
production. The best thing to do is to move it into the configuration file, just like we
did with the other OpenIddict-related values.

Open the `appsettings.json` file and add the following key to the
`Authentication:OpenIddict` section:

```
"Authentication": {
  "OpenIddict": {
```

```
    "ApplicationId": "OpenGameList",
    "DisplayName": "OpenGameList",
    "TokenEndPoint": "/api/connect/token",
    "ClientId": "OpenGameList",
    "ClientSecret": "1234567890_my_client_secret",
    "Authority": "http://localhost:14600/"
  }
}
```

Right after that, replace the literal value in `Setup.cs` accordingly:

```
Authority = Configuration["Authentication:OpenIddict:Authority"],
```

Updating the Web.Config rewrite rules

It's time to update those rewrite rules that we pulled off back in [Chapter 3](#), *Angular 2 Components and Client-Side Routing*. We need to do that to ensure that all the OpenIddict's endpoint URIs will be reachable by any internal and external actor involved in the authentication process.

To do that, open the `web.config` file and add the following lines to the `system.webServer > Rewrite` section (new lines highlighted):

```xml
<rewrite>
  <rules>
    <rule name="Angular 2 pushState routing" stopProcessing="true">
      <match url=".*" />
      <conditions logicalGrouping="MatchAll">
        <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />
        <add input="{REQUEST_FILENAME}" pattern=".*\.[\d\w]+$"
negate="true" />
        <add input="{REQUEST_URI}" pattern="^/(api)" negate="true"
/>

        <add input="{REQUEST_URI}" pattern="^/(.well-known)"
negate="true" />


        <add input="{REQUEST_URI}" pattern="^/(signin)"
negate="true" />
      </conditions>
      <action type="Rewrite" url="/index.html" />
    </rule>
  </rules>
</rewrite>
```

These lines will ensure that all the required `OpenIddict` endpoint URIs won't be rewritten to the `index.html` file, just like it is for the URLs starting with `/api`.

Note

It's worth saying that we could also change all the endpoint URIs programmatically instead of updating the rewrite rule. However, it would imply changing a lot of default values, which is something that should be normally avoided when dealing with complex package libraries such as *EF Core*, *Identity Core*, and *OpenIddict*. Compared to that, the rewrite update we just made is definitely a simpler yet also more secure approach.

Seeding the database

We're ready to populate our database and test our new authentication provider. We can do that by hitting *F5*, let the `DbSeeder` middleware fill the database with the new required info, and checking out the resulting outcome.

If everything has been made correctly, we shouldn't be able to detect changes of any sort in our application's behavior, as the new provider is expected to hook on the same routes and use the same naming conventions as the previous one. This is hardly a coincidence; we purposely implemented both of them using the same criteria in order to make them interchangeable to the extent of our limited scenario.

External authentication providers

Now that we finally have a fully-featured, JWT-based internal authentication provider, we can start thinking about supporting other login alternatives. Hooking up some external authentication providers such as Google, Facebook, and Twitter will undoubtedly please our users and increase our application's overall appeal.

Luckily enough, AspNetCore.Identity comes bundled with a built-in set of middleware classes for the most used OAuth2 providers that will simplify each of these tasks.

OAuth2 authorization flow

Before we start, let's do a quick recap of how the OAuth2 authorization flow actually works:
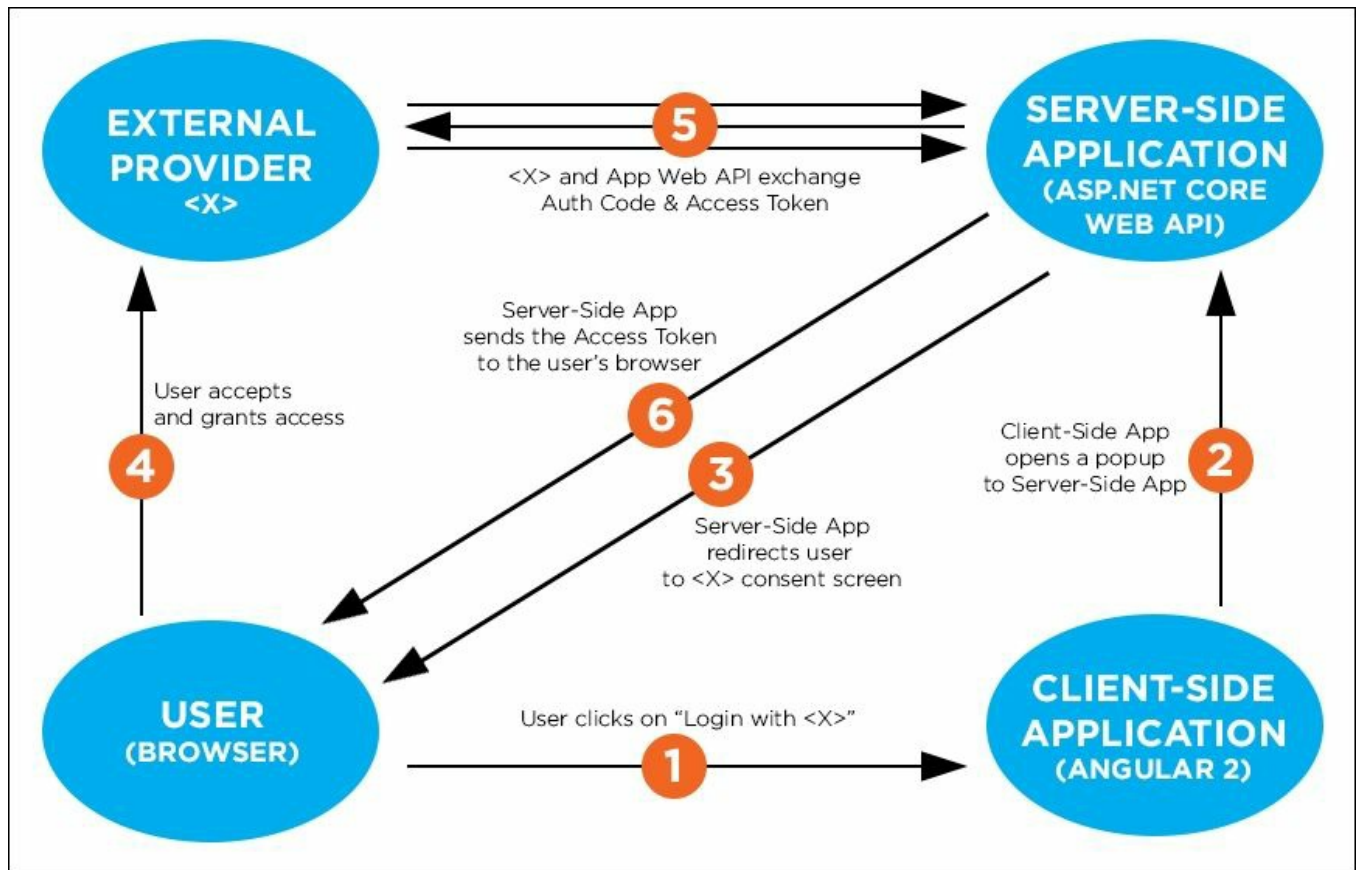
This, in short, is what will happen under the hood. Luckily enough, the AspNetCore middleware will do most of the job, as long as we provide a minimal amount of infrastructural logic to our Web API.

Facebook

Let's start with the big whale of the social networks.

The first thing we need to do is to obtain an `AppId` and an `AppSecret` that we can use to perform our first request against Facebook's OAuth2 authentication workflow.

Note
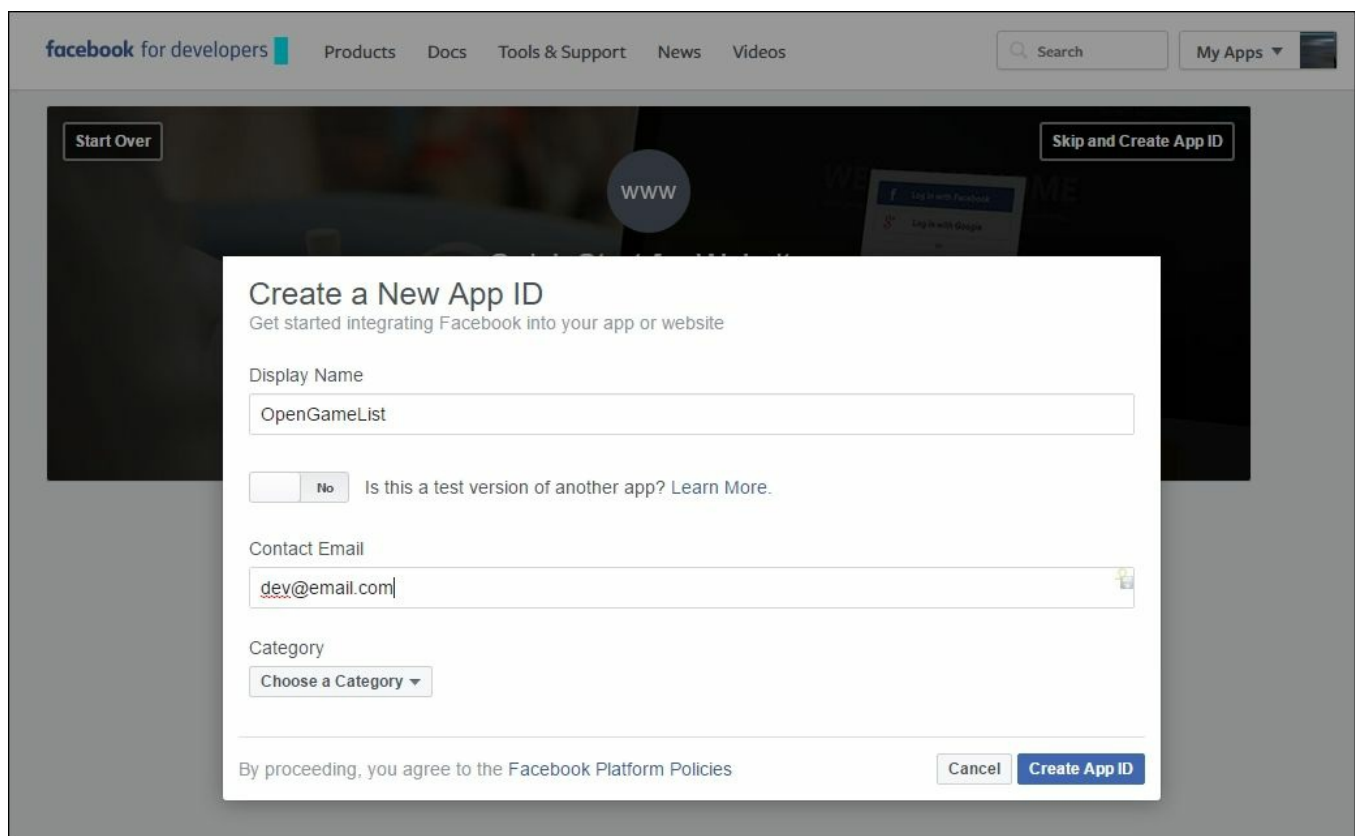
To learn more about the OAuth2 authorization framework, we strongly suggest reading the following URLs:

- **Official Page**: http://oauth.net/2/
- **RFC 6749**: https://tools.ietf.org/html/rfc6749

Creating a Facebook App

Go to the Facebook Developer page at the following URL: https://developers.facebook.com/.

In order to use it, we need to log in using a **Facebook Developer Account**. Once inside, click **Add a New App:,** select **Website**, fill in the required fields, and click **Create App ID**:

Note

It's worth saying that external provider web platforms are subject to frequent and major changes. The actual Google, Facebook, and Twitter pages and forms might be different from those depicted by the screenshots made at the time of writing.

As soon as we get past the **CAPTCHA**, a new Facebook app will be added to our account and we'll be automatically brought to the **Add Products** selection screen. Once there, we should be able to add a new **Facebook Login** product by clicking on the **Get Started** button to the right:
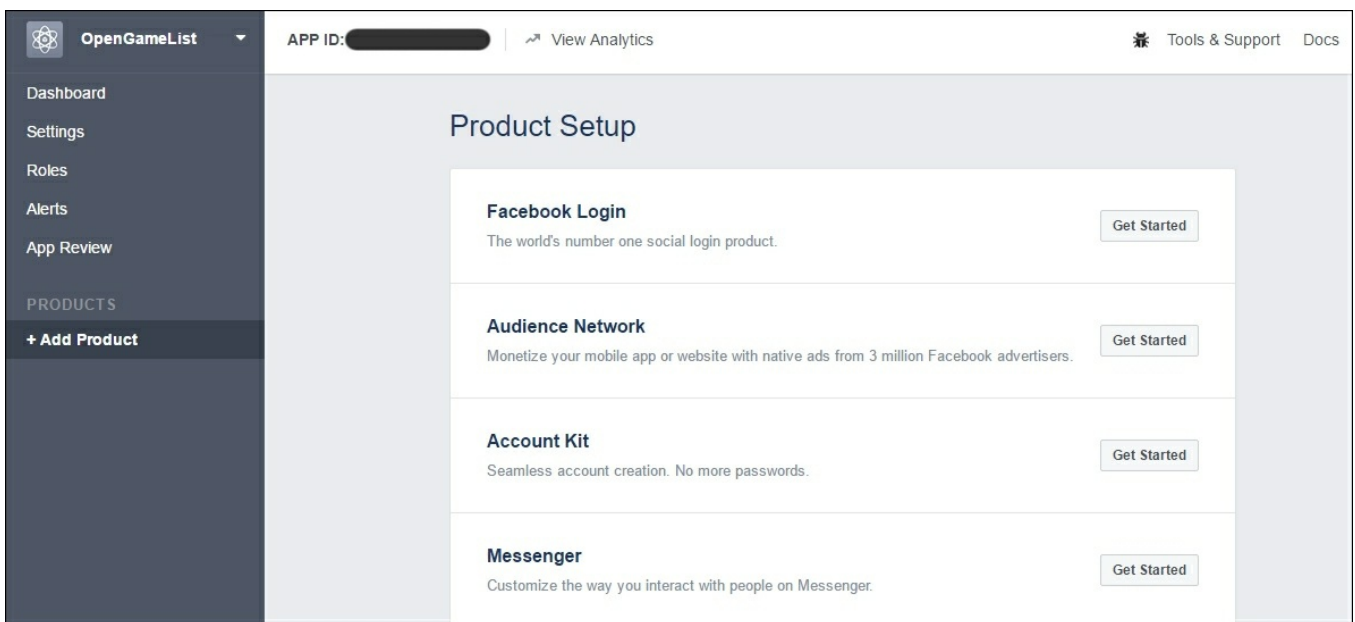
A modal window will open with a number of OAuth2-related options. The default values are OK, with the sole exception of the **Valid OAuth redirect URIs** textbox, which needs to be updated with our current URL endpoint. Since we're in development, we need to allow our local address and also the developer port we're using:
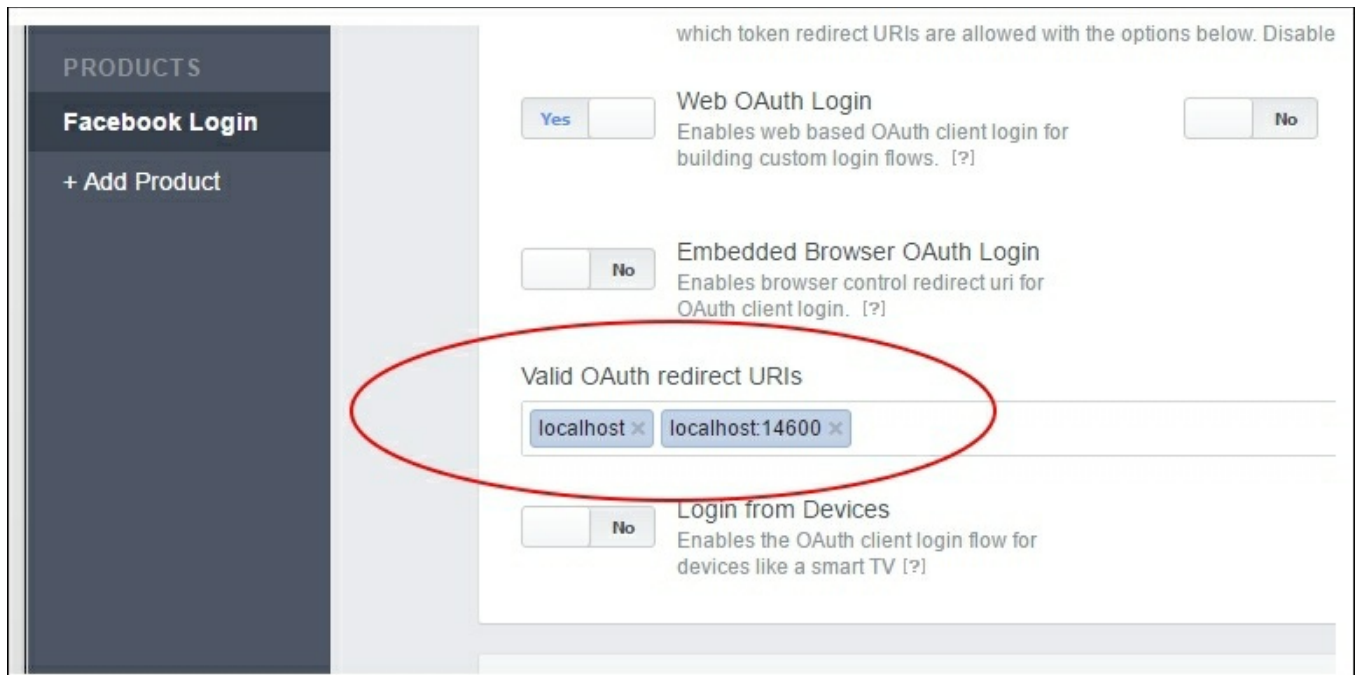
In the preceding example, we specified two valid URIs: `localhost` (which defaults to port 80) and `localhost:14600`, which is the one used by Visual Studio during our debug runs.

We don't need to configure anything else. Before leaving, we need to go to the **Dashboard** and retrieve the **App ID** and **App Secret** values, as we'll need them soon enough.

Tip

If you want to apply further access restrictions to your Facebook App, you can go to **Settings** | **Advanced** and configure the **Server IP whitelist** on the **Security** panel. However, once you input something there, the whitelist logic will immediately kick in, blocking every other IP.

Storing the keys in App settings

Open the `appsettings.json` file, locate the `Authentication` root key we created earlier, and add a new `Facebook` key right below `OpenIddict`. Once done, use it to store the **App ID** and **App Secret** values of the previously-created Facebook app in the following way:

```
"Authentication": {
  "OpenIddict": {
    "ApplicationId": "OpenGameList",
    "DisplayName": "OpenGameList",
    "TokenEndPoint": "/api/connect/token",
```

```
    "ClientId": "OpenGameList",
    "ClientSecret": "1234567890_my_client_secret",
    "Authority": "http://localhost:14600/"
  },
  "Facebook": {

    "AppId": "__INSERT_APP_ID_HERE__",

    "AppSecret": "__INSERT_APP_SECRET_HERE__"

  }
}
```

Note

Storing these values in plain text inside the `appsettings.json` file is not recommended, because they can be easily accessed by unauthorized people (network admins, server admins, and so on) or even checked into some public source control repositories by some developer's mistake. There are better alternatives nowadays, such as the **Secret Manager Tool**, granting a better level of security.

For more info about how to use it, it's highly advisable to carefully read the following guide from the official ASP.NET Core documentation website:

https://docs.asp.net/en/latest/security/app-secrets.html.

Adding the Facebook middleware

Now we need to add the Facebook middleware to the HTTP request pipeline.

Open the `project.json` file and add the following package:

```
 "Microsoft.AspNetCore.Authentication.Facebook": "1.0.0"
```

Once done, open up the `Startup.cs` file, locate the `Configure` method, and add the following right above the **JwtProvider**/OpenIddict middleware, as follows (new lines are highlighted):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment
env, ILoggerFactory loggerFactory, DbSeeder dbSeeder)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // Configure a rewrite rule to auto-lookup for standard default
files such as index.html.
    app.UseDefaultFiles();

    // Serve static files (html, css, js, images & more). See also
```

```
the following URL:
    // https://docs.asp.net/en/latest/fundamentals/static-
files.html for further reference.
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = (context) =>
        {
            // Disable caching for all static files.
            context.Context.Response.Headers["Cache-Control"] =
Configuration["StaticFiles:Headers:Cache-Control"];
            context.Context.Response.Headers["Pragma"] =
Configuration["StaticFiles:Headers:Pragma"];
            context.Context.Response.Headers["Expires"] =
Configuration["StaticFiles:Headers:Expires"];
        }
    });
    // Add a custom Jwt Provider to generate Tokens
    // app.UseJwtProvider();

    // Add the AspNetCore.Identity middleware (required for
external auth providers)

    // IMPORTANT: This must be placed *BEFORE* OpenIddict and any
external provider's middleware

    app.UseIdentity();

    // Add external authentication middleware below.

    // To configure them please see
http://go.microsoft.com/fwlink/?LinkID=532715

    app.UseFacebookAuthentication(new FacebookOptions()

    {

        AutomaticAuthenticate = true,

        AutomaticChallenge = true,

        AppId = Configuration["Authentication:Facebook:AppId"],

        AppSecret =
Configuration["Authentication:Facebook:AppSecret"],

        CallbackPath = "/signin-facebook",

        Scope = { "email" }

    });
    // Add OpenIddict middleware
    // NOTE: UseOpenIddict() must be registered after
```

```
app.UseIdentity()
    // and all the external social provider middlewares (if any).
    app.UseOpenIddict();

    // Add the Jwt Bearer Header Authentication to validate Tokens
    app.UseJwtBearerAuthentication(new JwtBearerOptions()
    {
        AutomaticAuthenticate = true,
        AutomaticChallenge = true,
        RequireHttpsMetadata = false,
        Authority =
Configuration["Authentication:OpenIddict:Authority"],
        TokenValidationParameters = new TokenValidationParameters()
        {
            //IssuerSigningKey = JwtProvider.SecurityKey,
            //ValidateIssuerSigningKey = true,
            //ValidIssuer = JwtProvider.Issuer,
            ValidateIssuer = false,
            ValidateAudience = false
        }
    });

    // Add MVC to the pipeline
    app.UseMvc();

    // TinyMapper binding configuration
    TinyMapper.Bind<Item, ItemViewModel>();

    // Seed the Database (if needed)
    try
    {
        dbSeeder.SeedAsync().Wait();
    }
    catch (AggregateException e)
    {
        throw new Exception(e.ToString());
    }
}
```

As we already said, the middleware order is very important. Any internal, external, or third-party authentication middleware coming after `UseMvc` won't be able to hook up on any route.

Adding the AccountsController

Since this is our first external provider, we need to do some extra work here, adding a new controller to deal with a number of account-related tasks to use for this provider and also for those that will come next. We'll also extensively use it during Chapter 9, *User Registration and Account Edit*, when we'll be dealing with **User Registration** and **Edit Account** functions. Its name will be `AccountsController`

and it will have a lot of features in common with the already existing `ItemsController`, such as an `AccountDbContext` DI instance to access the data provider, a `DefaultJsonSettings` property to format JSON data, the code required to fetch the authenticated user details, and so on.

Wait a minute... are we really going to duplicate all these code lines? We clearly shouldn't, unless we want to completely ditch the **DRY** principle, which is something that any developer should try to follow whenever they can.

Note

DRY stands for **Don't Repeat Yourself** and is a widely achieved principle of software development. Whenever we violate it we fall into a **WET** approach, which could mean *Write Everything Twice*, *We Enjoy Typing* or *Waste Everyone's Time,* depending on what you like the most.

As a matter of fact, we can avoid a lot of code repetition by adding a `BaseController` that will host these reusable objects, methods, and properties. This has always been a rather common DRY pattern for MVC and Web API and it's still very doable in ASP.NET Core.

BaseController

Let's do this. Right-click on the Controllers folder, select **Add** | **New Item** and add a new **Web API Controller** class. Name it `BaseController.cs` and click **OK** to add it to our project's tree:
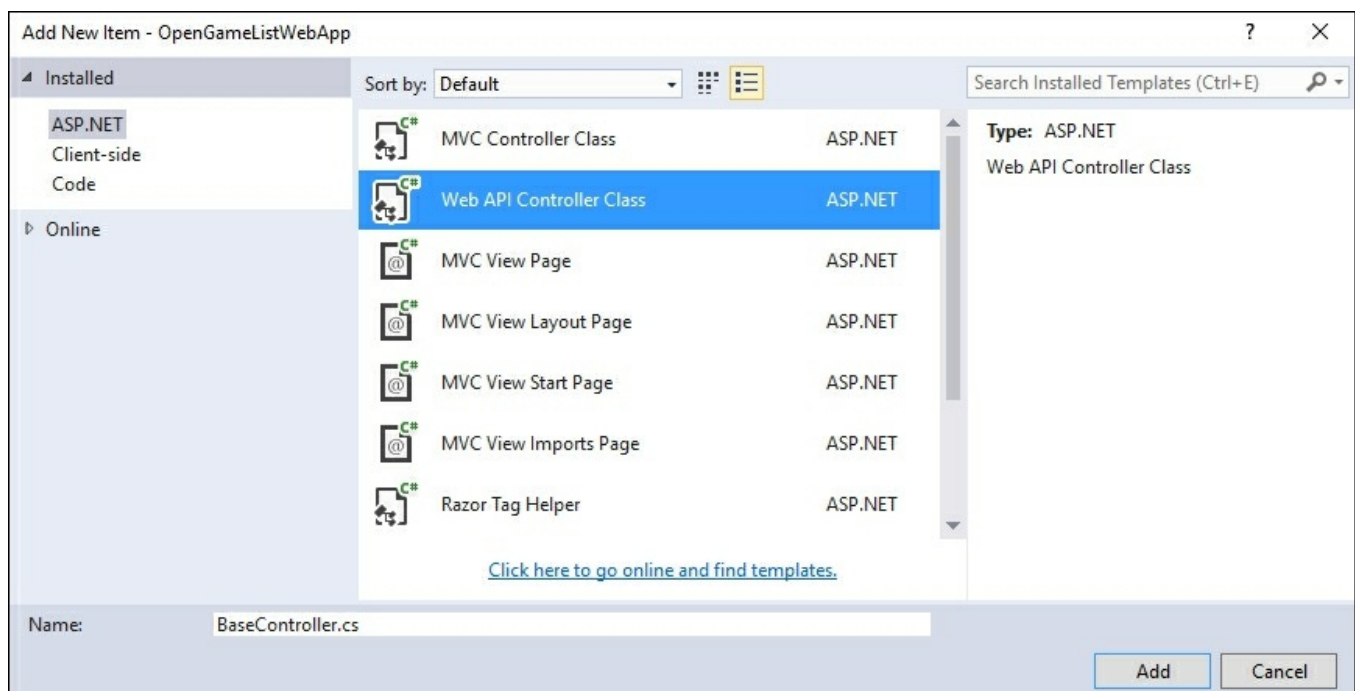
The `BaseController` will be the base class of our existing `ItemsController` and also of the new `AccountsController` we're about to create, so we have to put there everything that we reasonably expect both of them will use. With that in mind, this could be a good start:

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
using OpenGameListWebApp.Data;
using OpenGameListWebApp.Data.Users;
using Newtonsoft.Json;

namespace OpenGameListWebApp.Controllers
{
    [Route("api/[controller]")]
    public class BaseController : Controller
    {
        #region Common Fields
        protected ApplicationDbContext DbContext;
        protected SignInManager<ApplicationUser> SignInManager;
        protected UserManager<ApplicationUser> UserManager;
        #endregion Common Fields

        #region Constructor
        public BaseController(ApplicationDbContext context,
            SignInManager<ApplicationUser> signInManager,
            UserManager<ApplicationUser> userManager)
        {
            // Dependency Injection
            DbContext = context;
            SignInManager = signInManager;
            UserManager = userManager;
        }
        #endregion Constructor

        #region Common Methods
        /// <summary>
        /// Retrieves the .NET Core Identity User Id
        /// for the current ClaimsPrincipal.
        /// </summary>
        /// <returns></returns>
        public async Task<string> GetCurrentUserId()
        {
            // if the user is not authenticated, throw an exception
            if (!User.Identity.IsAuthenticated)
                throw new NotSupportedException();

            var info = await
```

```
SignInManager.GetExternalLoginInfoAsync();
            if (info == null)
                // internal provider
                return
User.FindFirst(ClaimTypes.NameIdentifier).Value;
            else
            {
                // external provider
                var user = await UserManager.FindByLoginAsync(
                    info.LoginProvider,
                    info.ProviderKey);
                if (user == null) throw new
NotSupportedException();
                return user.Id;
            }
        }
        #endregion Common Methods

        #region Common Properties
        /// <summary>
        /// Returns a suitable JsonSerializerSettings object
        /// that can be used to generate the JsonResult return
value
        /// for this Controller's methods.
        /// </summary>
        protected JsonSerializerSettings DefaultJsonSettings
        {
            get
            {
                return new JsonSerializerSettings()
                {
                    Formatting = Formatting.Indented
                };
            }
        }
        #endregion Common Properties
    }
}
```

Let's see what we put in there:

- Above the controller's `class` definition there's the `[Route]` attribute, which
  defines the default routing scheme. Nothing new there, just the standard
  api/[controller] attribute-based rule we've been using since Chapter 2, *ASP.NET
  Controllers and Server-Side Routes*. Since we want it for all Controllers, we
  chose to put it here so we won't have to type this single line of code more than
  once.
- Right before the constructor, we declared the local instances of the ASP.NET
  Identity handlers we'll be using throughout all our Controllers. Since we're
  going to need these everywhere, declaring them here is indeed the right thing to

do to.

- The `GetCurrentUserId` method is something we've already seen before. It contains the *one-liner* we used back in Chapter 7, *Authentication and Authorization*, to retrieve the authenticated user's `Id`. That time, we said that we were going to change it in the near future and we actually did that: now it supports both internal and external providers, which is precisely what we need. We also took the chance to centralize it here, since we're going to use it more than once.
- The `DefaultJsonSettings` property is an old friend. We coded it back in Chapter 2, *ASP.NET Controllers and Server-Side Routes*. We put it there for obvious reasons, as this is something each Controller will most likely use.

AccountsController

Time to finally add the `AccountsController` to the loop. Once again, right-click on the **Controllers** folder, select **Add** | **New Item**, and add another **Web API Controller** class file. Name it `AccountsController.cs` and click **OK**.

Since we're going to write a good amount of code, we'll split it into multiple sections, wrapping them into regions as much as we can.

Namespaces

Let's start with the namespaces that we'll be using throughout the controller:

```
using System;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using OpenGameListWebApp.Classes;
using OpenGameListWebApp.Data;
using OpenGameListWebApp.Data.Users;
using OpenGameListWebApp.ViewModels;
```

As usual, we need to add them to the beginning of the file.

Class declaration

We don't need to define a default routing scheme since we already put that in the `BaseController` class. We just need to extend it within the class declaration, like the following:

```
public class AccountsController : BaseController
```

Constructor

We can use the constructor of the base class since we made it to suit our needs.

```
#region Constructor
public AccountsController(
    ApplicationDbContext context,
    SignInManager<ApplicationUser> signInManager,
    UserManager<ApplicationUser> userManager) : base(
    context,
    signInManager,
    userManager)
{ }
#endregion Constructor
```

Again, no big news here. This is the same DI pattern we've already used a number of times before. We'll be able to use these instances by accessing the `protected` local variables defined in the base class, which will be available here.

ExternalLogin

This is the method that our Angular 2 client will call whenever the user initiates an external login request:

```
#region External Authentication Providers
// GET: /api/Accounts/ExternalLogin
[HttpGet("ExternalLogin/{provider}")]
public IActionResult ExternalLogin(string provider, string
returnUrl = null)
{
    switch (provider.ToLower())
    {
        case "facebook":
        case "google":
        case "twitter":
            // Request a redirect to the external login provider.
            var redirectUrl = Url.Action("ExternalLoginCallback",
"Accounts", new { ReturnUrl = returnUrl });
            var properties =
SignInManager.ConfigureExternalAuthenticationProperties(provider,
redirectUrl);
            return Challenge(properties, provider);
        default:
            return BadRequest(new { Error = String.Format("Provider
'{0}' is not supported.", provider) });
    }
}
```

Tip

Notice that we started a new region here: it will contain the whole set of methods required to properly handle the external authentication provider's authentication flow.

As we can see, it will configure the external request and initiates the authentication workflow against the given provider, assuming it's among the supported ones. If it's not, it will issue a `400 - Bad Request` HTTP error response instead.

ExternalLoginCallBack

This method will be executed at the end of the OAuth2 workflow to handle the authentication success or failure scenarios:

```
[HttpGet("ExternalLoginCallBack")]
public async Task<IActionResult> ExternalLoginCallback(string
returnUrl = null, string remoteError = null)
{
    try
    {
        // Check if the External Provider returned an error and act
accordingly
        if (remoteError != null)
        {
            throw new Exception(remoteError);
        }

        // Extract the login info obtained from the External
Provider
        ExternalLoginInfo info = await
SignInManager.GetExternalLoginInfoAsync();
        if (info == null)
        {
            // if there's none, emit an error
            throw new Exception("ERROR: No login info available.");
        }

        // Check if this user already registered himself with this
external provider before
        var user = await
UserManager.FindByLoginAsync(info.LoginProvider, info.ProviderKey);
        if (user == null)
        {
            // If we reach this point, it means that this user
never tried to logged in
            // using this external provider. However, it could have
used other providers
            // and /or have a local account.
            // We can find out if that's the case by looking for
his e-mail address.

                // Retrieve the 'emailaddress' claim
```

```csharp
            var emailKey =
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress
";
            var email = info.Principal.FindFirst(emailKey).Value;

            // Lookup if there's an username with this e-mail
address in the Db
            user = await UserManager.FindByEmailAsync(email);
            if (user == null)
            {
                // No user has been found: register a new user
using the info retrieved from the provider
                DateTime now = DateTime.Now;

                // Create a unique username using the
'nameidentifier' claim
                var idKey =
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifi
er";
                var username = String.Format("{0}{1}",
info.LoginProvider, info.Principal.FindFirst(idKey).Value);

                user = new ApplicationUser()
                {
                    UserName = username,
                    Email = email,
                    CreatedDate = now,
                    LastModifiedDate = now
                };

                // Add the user to the Db with a random password
                await UserManager.CreateAsync(user,
"Pass4External");

                // Assign the user to the 'Registered' role.
                await UserManager.AddToRoleAsync(user,
"Registered");

                // Remove Lockout and E-Mail confirmation
                user.EmailConfirmed = true;
                user.LockoutEnabled = false;
            }
            // Register this external provider to the user
            await UserManager.AddLoginAsync(user, info);

            // Persist everything into the Db
            await DbContext.SaveChangesAsync();
        }

        // create the auth JSON object
        var auth = new
        {
```

```
            type = "External",
            providerName = info.LoginProvider
        };

        // output a <SCRIPT> tag to call a JS function registered
into the parent window global scope
        return Content(
            "<script type="text/javascript">" +
            "window.opener.externalProviderLogin(" +
JsonConvert.SerializeObject(auth) + ");" +
            "window.close();" +
            "</script>",
            "text/html"
            );
    }
    catch (Exception ex)
    {
        // return a HTTP Status 400 (Bad Request) to the client
        return BadRequest(new { Error = ex.Message });
    }
}
```

This is where all the magic takes place, as we'll be checking for a number of things and take action accordingly:

1. Before anything else, we wrap all the method into a `try`/`catch` block, so any given `Exception` would result in a `400 - Bad Request` HTTP error response.

2. We check the external provider error message (if any) by looking at the `remoteError` parameter value. If something went bad, we throw an `Exception` here, otherwise, we go ahead.

3. We extract the `ExternalLoginInfo` object using the `SignInManager`. This is a strongly-typed .NET object containing the response data sent by the external provider and decrypted by the *Facebook middleware*. In the unlikely case it happens to be `null`, we throw an `Exception`, otherwise, we go ahead.

4. We check whenever the user already authenticated himself with this external provider before using the `UserManager.FindByLoginAsync` method. If that's the case, we skip to step 8; otherwise, we need to do additional checks.

5. We need to check whether the user registered himself before using different providers. To do so, we retrieve the user e-mail from the `ExternalLoginInfo` object, so we can perform a database lookup to see whether we already have it. If that's the case, we skip to step 7; otherwise, we need to create it.

6. We create a new user using the data we can retrieve from the relevant `ExternalLoginInfo` claims, including a temporary (yet unique) username and a random password that they'll be able to change in the future. We also assign them the registered user role.

7. We associate the user with this external provider, so we'll be ready to handle

further authentication attempts (skipping steps *5-7*).

8. We create an `auth` JSON object with some useful data.

9. Finally, we output a `text/html` response containing a `<SCRIPT>` tag that will be executed by the client pop-up window to handle the external login on the client side.

The last step is very important and deserves some explanation. As we might already know, the OAuth2 authorization workflow is an interactive process where the user has to manually accept a consent form. In order to support that, we'll need to call these controller routes from a popup window. That's why we need to call a function registered within the parent window (`window.opener`) and also close the current one using `window.close()`.

Logout

Since the external provider implementation is based on cookies, we need to create a server-side method that will remove them (if present) when the user performs the logout. The `SignInManager.SignOutAsync` method automatically handles that:

```
[HttpPost("Logout")]
public IActionResult Logout()
{
    if (HttpContext.User.Identity.IsAuthenticated)
    {
        SignInManager.SignOutAsync().Wait();
    }
    return Ok();
}
#endregion External Authentication Providers
```

Tip

We're closing the external authentication providers region right after this method.

Updating the ItemsController

Before moving to Angular 2, we should really bring the `ItemsController` up to date with our recent changes. Open the `/Controllers/ItemsController.cs` file and append the following namespaces to the already existing `using` list:

```
using Microsoft.AspNetCore.Identity;
using OpenGameListWebApp.Data.Users;
```

Right after that, change the class declaration and the constructor to extend the `BaseController` in the following way:

```
public class ItemsController : BaseController
{
    #region Constructor
    public ItemsController(

        ApplicationDbContext context,

        SignInManager<ApplicationUser> signInManager,

        UserManager<ApplicationUser> userManager) : base(

        context,

        signInManager,

        userManager)

    { }
    #endregion Constructor
```

The `[Route]` attribute can be deleted, as well as the whole private fields `region` and the `DefaultJsonSettings` property near the end of the file. We'll get them all from the `BaseController`.

We still need to perform some changes to the `Add` method, as it still contains the outdated `UserId` retrieval method with no external provider support that we implemented in [Chapter 7](#), *Authentication and Authorization*. We need to replace it with the updated version we implemented within the `BaseController` earlier:

```
item.UserId = await GetCurrentUserId();
```

However, since this is an `async` method, doing that will also require setting the `async` method:

```
public async Task<IActionResult> Add([FromBody]ItemViewModel ivm)
```

The server-side implementation tasks are finally done. Now we can switch back to Angular 2 and configure our client to properly handle what we just did.

Note

Before we continue, it might be wise to issue a full project **Rebuild** to ensure that there are no compile errors up to this point.

Configuring the client

Let's do a quick inventory of what we need to do within our Angular 2 client app to allow our users to log in with Facebook:

1. Add a **Login with Facebook** button to our login view, bound to a method that will fire the **OAuth2** initial request to our Web API. We also need to do that within a popup, since the user will be prompted by the Facebook consent form.
2. Change the `logout()` method within the `AuthService` class. We need it to issue a call to the new `Logout` web API to remove the authentication cookie placed by external providers (if any).
3. Update the `AppComponent` class with the new changes. Specifically, we need to do the following:
   - Change the internal `AppComponent.logout()` handler to match the new `AuthService.logout()` implementation.
   - Add an `externalProviderLogin` method to handle the external authentication outcome response. We also need to make it available in the global scope, as we know it will be called from a popup using the `window.opener` interface.

Challenge accepted. Let's do this.

Updating the LoginComponent

Open the `/Scripts/app/login.component.ts` file, locate the `template` section, and add the following (new lines are highlighted):

```
<div class="login-container">
    <h2 class="form-login-heading">Login</h2>
    <div class="alert alert-danger" role="alert"
*ngIf="loginError">
        <strong>Warning:</strong> Username or Password mismatch
    </div>
    <form class="form-login" [formGroup]="loginForm"
(submit)="performLogin($event)">
        <input formControlName="username" type="text" class="form-
control" placeholder="Your username or e-mail address" required
autofocus />
        <input formControlName="password" type="password"
class="form-control" placeholder="Your password" required />
        <div class="checkbox">
            <label>
                <input type="checkbox" value="remember-me">
                Remember me
            </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Sign in</button>
    </form>
    <button class="btn btn-sm btn-default btn-block" type="submit"
(click)="callExternalLogin('Facebook')">
```

```
        Login with Facebook

    </button>
</div>
```

Once done, scroll down to the `class` section and add the following code (new lines are highlighted):

```
export class LoginComponent {
    title = "Login";
    loginForm = null;
    loginError = false;
    externalProviderWindow = null;

    constructor(
        private fb: FormBuilder,
        private router: Router,
        private authService: AuthService) {
        this.loginForm = fb.group({
            username: ["", Validators.required],
            password: ["", Validators.required]
        });
    }

    performLogin(e) {
        e.preventDefault();
        var username = this.loginForm.value.username;
        var password = this.loginForm.value.password;
        this.authService.login(username, password)
            .subscribe((data) => {
                // login successful
                this.loginError = false;
                var auth = this.authService.getAuth();
                alert("Our Token is: " + auth.access_token);
                this.router.navigate([""]);
            },
            (err) => {
                console.log(err);
                // login failure
                this.loginError = true;
            });
    }


    callExternalLogin(providerName: string) {

        var url = "api/Accounts/ExternalLogin/" + providerName;

        // minimalistic mobile devices support
```

```
        var w = (screen.width >= 1050) ? 1050 : screen.width;

        var h = (screen.height >= 550) ? 550 : screen.height;

        var params =
"toolbar=yes,scrollbars=yes,resizable=yes,width=" + w + ", height="
+ h;

        // close previously opened windows (if any)

        if (this.externalProviderWindow) {

            this.externalProviderWindow.close();

        }

        this.externalProviderWindow = window.open(url,
"ExternalProvider", params, false);

    }
}
```

That's it.

Note

Even if we added some logic to make things work on mobile devices, manually invoking `window.open` is not something we should be proud of. As a matter of fact, it's never a good practice for mobile-friendly web applications. There are a lot of existing client libraries, including Facebook and Google's OAuth2 JavaScript SDK, that can show, resize, and handle pop up and modal windows in a much better way than we can do in a bunch of JavaScript lines.

We're not using these to keep things as simple as possible, so we can focus on the core aspect of the process, which relieves us from adding external libraries and explaining how to properly use them. However, it's definitely advisable to migrate to a more robust solution before going live.

Updating the AuthService

When we built the `AuthService` class back in Chapter 7, *Authentication and Authorization*, we were thinking about supporting JWT tokens only. That's why we implemented a simple `logout()` method that basically just deletes the JWT tokens from `localStorage` and returns `true`. Now that we're adding the ASP.NET Core middleware for external providers, we need to delete their cookies as well. That's the reason why we added the `Logout()` Web API within our `AccountsController`, which now we need to call from Angular 2.

To do that, just open the `Scripts/app/auth.service.ts` file and change the existing `logout()` method in the following way:

```
logout(): any {
    return this.http.post(
        "api/Accounts/Logout",
        null)
        .map(response => {
            this.setAuth(null);
            return true;
        })
        .catch(err => {
            return Observable.throw(err);
        });
}
```

That's pretty straightforward. Instead of just removing the JSON web token from `localStorage`, we issue a call to the `api/Accounts/Logout` web API to delete the cookies. The `localStorage` cleansing will happen right after that. That way, we'll remove both the cookie-based external provider support and the JWT-based implementation that handles our internal accounts.

Updating the AppComponent

We're not done yet. The modifications we made must be also properly handled by the `AppComponent`, since it is our Angular 2 application's entry point. To be more specific, we need to change the internal `logout()` method to match the changes we made within the `AuthService` component and also implement the `externalProviderLogin` method.

Open the `/Scripts/app/app.component.ts` file and add change the following `import` line at the beginning of the file (new code is highlighted):

```
import {Component, NgZone} from "@angular/core";
```

Then change its `class` section just like the following (new/updated lines are highlighted):

```
export class AppComponent {
    title = "OpenGameList";

    constructor(
        public router: Router,
        public authService: AuthService,
        public zone: NgZone) {
        if (!(<any>window).externalProviderLogin) {

            var self = this;
```

```
            (<any>window).externalProviderLogin = function (auth) {

                self.zone.run(() => {

                    self.externalProviderLogin(auth);

                });

            }

        }
    }

    isActive(data: any[]): boolean {
        return this.router.isActive(
            this.router.createUrlTree(data),
            true);
    }

    logout(): boolean {
        // logs out the user, then redirects him to Welcome View.
        this.authService.logout().subscribe(result => {

            if (result) {

                this.router.navigate([""]);

            }

        });

        return false;
    }

    externalProviderLogin(auth: any) {

        this.authService.setAuth(auth);

        console.log("External Login successful! Provider: "

            + this.authService.getAuth().providerName);

        this.router.navigate([""]);

    }
}
```

The update within the `logout()` method is not a big deal. We just encapsulated its previous logic within a subscription to the updated `AuthService.logout()` return value, so we can be sure that the routing call won't be issued before the Web API call

has been completed.

However, the new code we wrote to properly implement the `externalProviderLogin` method might be more difficult to understand, at least at first glance. Let's take a look at the new stuff, starting from the bottom.

The `externalProviderLogin` method itself is actually pretty straightforward: we receive the JSON `auth` object as input parameter and pass it to the `authService` that will store it in `localStorage`, then we redirect the (externally) logged-in user back to the welcome view.

Plain and simple, isn't it? If we could only call this method from outside our Angular 2 app, we would be done. That's precisely what we're doing within the `constructor` method, which now contains a short, yet quite obscure, piece of code. Let's try to understand what we're doing there.

Understanding zones

If we take another look at the `constructor` updated source code, we can see that we're injecting something new here: a `zone` instance of `NgZone` type. What are these zones and how do they work in Angular 2?

To get straight to the point, **zones** are execution contexts for encapsulating and intercepting JavaScript-based asynchronous activities. Each zone acts as a separate, persisting execution context that can be used to trace each asynchronous activity back to its originating source.

For a short yet enlightening definition of what zones are, we can use the words of Brian Ford of the Angular team during the presentation of the `Zone.js` library at Ng-Conf 2014:

"You can think of it as thread-local storage for JavaScript VMs."

Note

The full talk is available at the following URL:

https://www.youtube.com/watch?v=3IqtmUscE_U.

As we already know, most modern JavaScript libraries execute a lot of asynchronous activities, such as DOM events, promises, and XHR calls. Being able to track these activities back to their issue would allow them to take action before and after each activity completes, thus providing great control over the whole execution flow.

This is most likely the reason that led Angular 2 developers to integrate the `zone.js` within their framework. As a matter of fact, Angular 2 runs the application and all of its components in a specific zone, so it can listen to its own asynchronous events and react accordingly, checking for data changes, updating the information shown on screen via data binding, and so on.

We won't go further than that, as it would take us far from the scope of this book. The only thing we need to understand here is that whenever we need to call one of our application's methods from outside, we also need to run it within the Angular 2 zone; if we don't do that, Angular 2 won't be able to track the originating source, meaning that it won't react to model changes.

This is what would happen if we were to do that:

```
if (!(<any>window).externalProviderLogin) {
    (<any>window).externalProviderLogin = function (auth) {
        self.externalProviderLogin(auth);
    }
}
```

As we can see, there's no zone encapsulation anymore, so the method will be fired from a global context and Angular 2 event hooks will be cut out. This doesn't necessarily mean it won't work, but it would be a fire and forget way of exposing that method.

In our specific scenario, since routing is involved, it won't be the proper way to perform our task. We need to run our job within the same execution context used by our application.

This is precisely what we did within our `constructor` method (zone-encapsulation lines are highlighted):

```
if (!(<any>window).externalProviderLogin) {
    var self = this;
    (<any>window).externalProviderLogin = function (auth) {
        self.zone.run(() => {
            self.externalProviderLogin(auth);
        });
    }
}
```

Testing it out

It's time to run a full-surface test. Hit *F5* and wait for the welcome screen, then click the **Login** menu item to go to the updated `login` view:
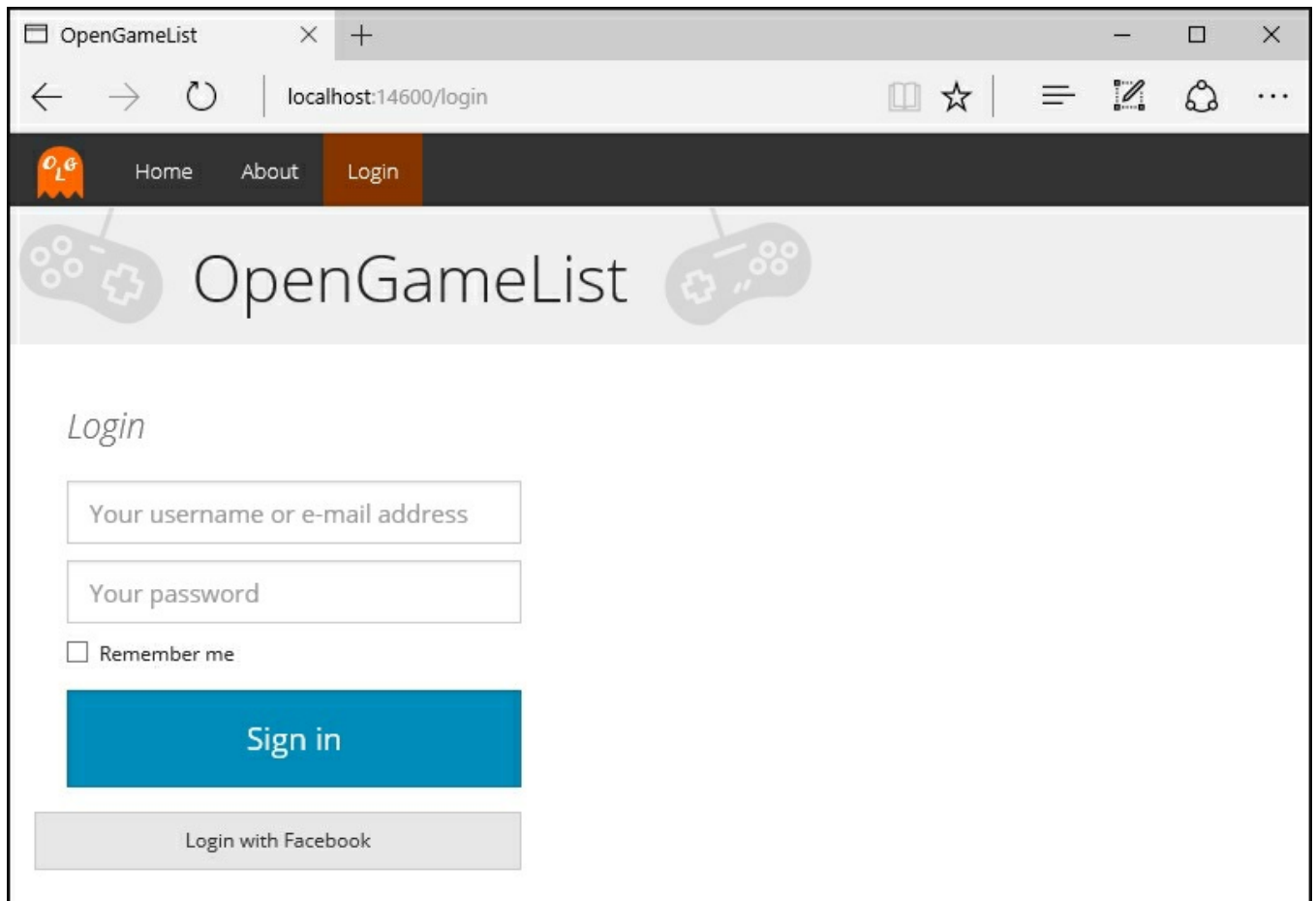
Click the **Login with Facebook** button. A pop-up window should appear shortly, containing the Facebook consent form:
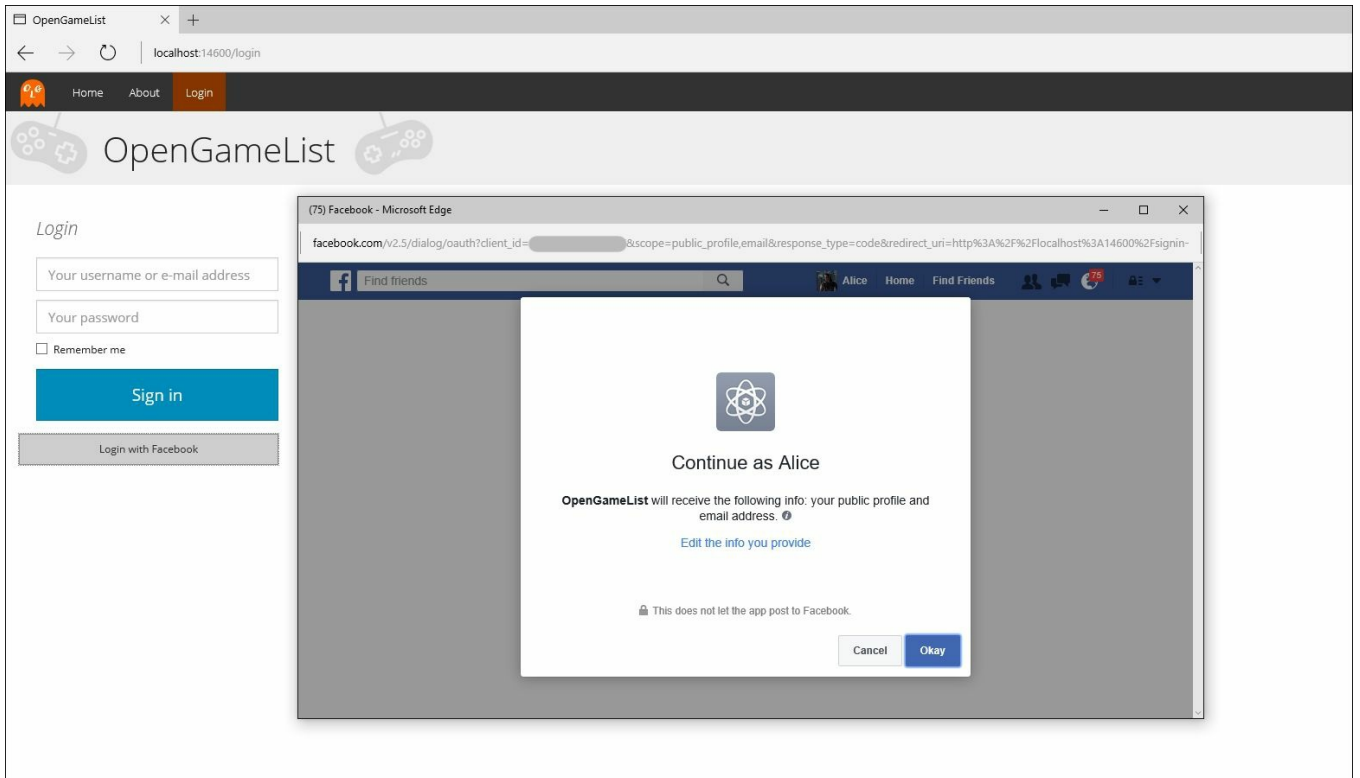
Tip

If we have never authenticated ourselves with Facebook using this browser, a login screen will be shown right before the consent form.

As soon as we click **Okay**, the OAuth2 authorization flow will continue. If we set everything up correctly, the pop-up screen should eventually close, bringing us back to the welcome view with an authenticated status. We can easily confirm our status by checking the presence of the **Logout** and **Add New** menu items since we previously set them to appear only if the user has been authenticated.

Google
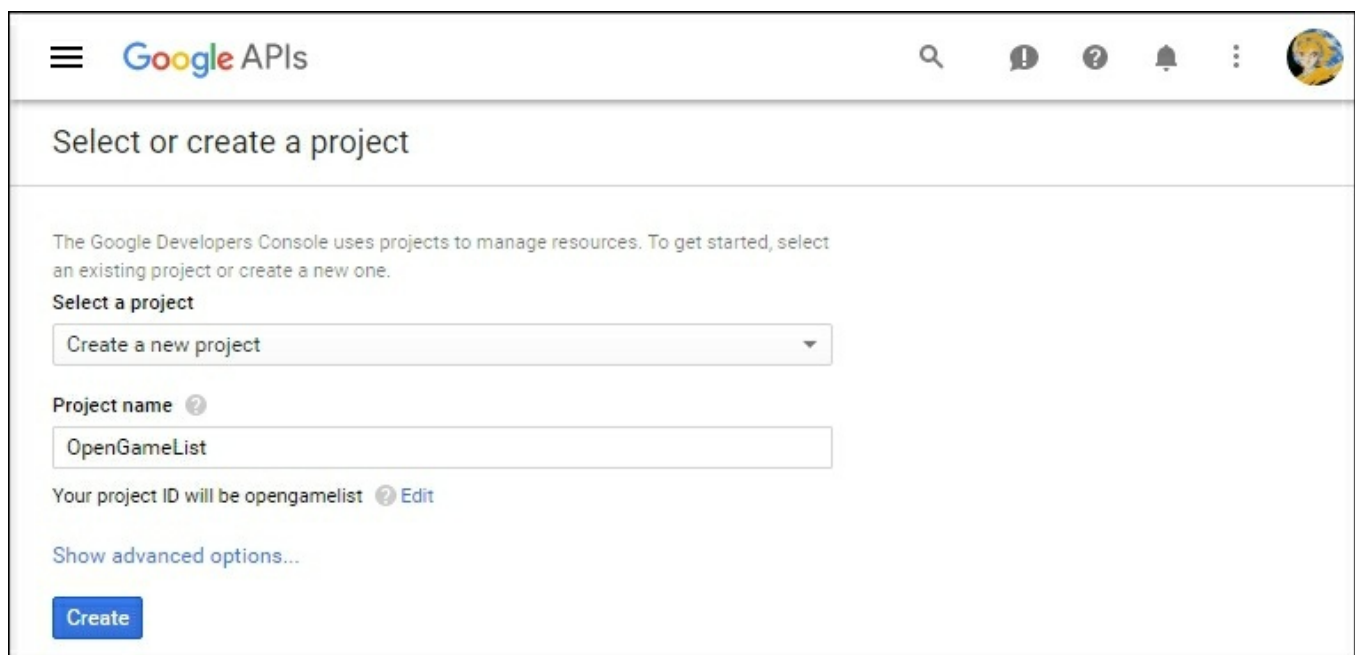
As we said earlier, implementing additional providers will be much easier, as the external authentication infrastructure we built for Facebook will also take care of them.

Let's start with the big G. Again, the first thing we need to do is to configure our web application on Google servers to obtain the required credentials to configure the ASP.NET middleware.

Open the following URL:

https://console.developers.google.com/projectselector/apis/credentials?pli=1

Select **Create a new Project** from the drop-down list, then give it a suitable name and click the **Create** button:

On the following screen, click the **Create credentials** button and select **OAuth client ID**:
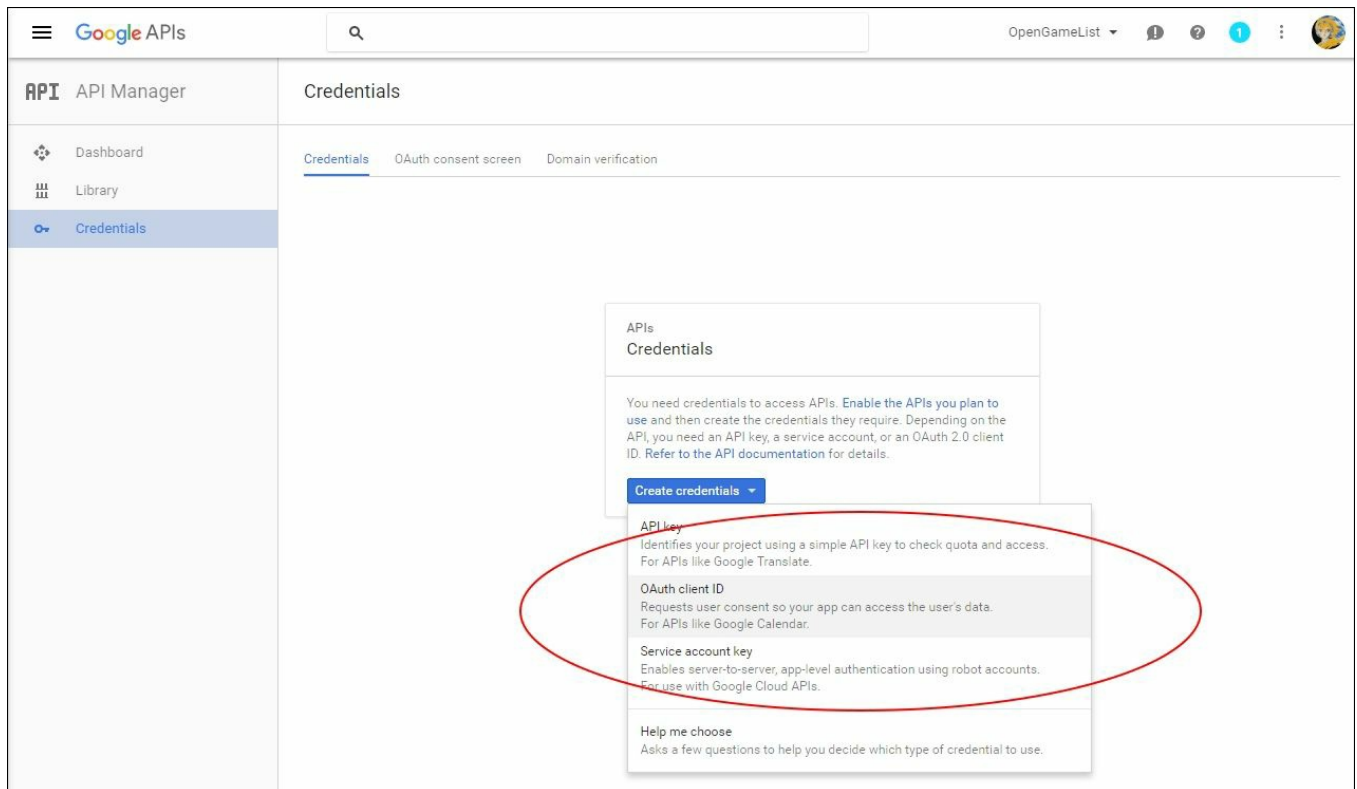
As soon as you confirm your choice, the Google platform will then warn you that you must create an **OAuth consent screen** for your application. We can do that by choosing an e-mail address and a product name shown to users (the remaining fields are optional).

Once done, we'll be allowed to add the **OAuth client ID**. Choose **Web Application** and fill in the form that will open with your application name and one (or more) JavaScript origins. It's pretty much the same thing we did with Facebook, with the only difference that we also have to specify the **Authorized redirect URIs** to match the format that will be internally used by ASP.NET Core, which defaults to the following: `http://<hostname>:<port>/signin-google`

When we're ready, we can click on the Create button to get our new application's **ClientId** and **ClientSecret**. We can then put them in our project's `appsettings.json` file, right below the Facebook keys:

```json
{
  "Authentication": {
    "Facebook": {
      "AppId": "___FB_APP_ID___",
      "AppSecret": "___FB_APP_SECRET___"
    },
    "Google": {

      "ClientId": "___GOOGLE_CLIENT_ID___",

      "ClientSecret": "___GOOGLE_CLIENT_SECRET___"

    }
```

```
    }
```

Before leaving the Google platform, we also need to add support for the **Google+ API**, otherwise, the OAuth2 login won't be able to work. To do that, go to **Library**, click the **Google APIs** tab, and look for **Google+ API** using the search textbox:
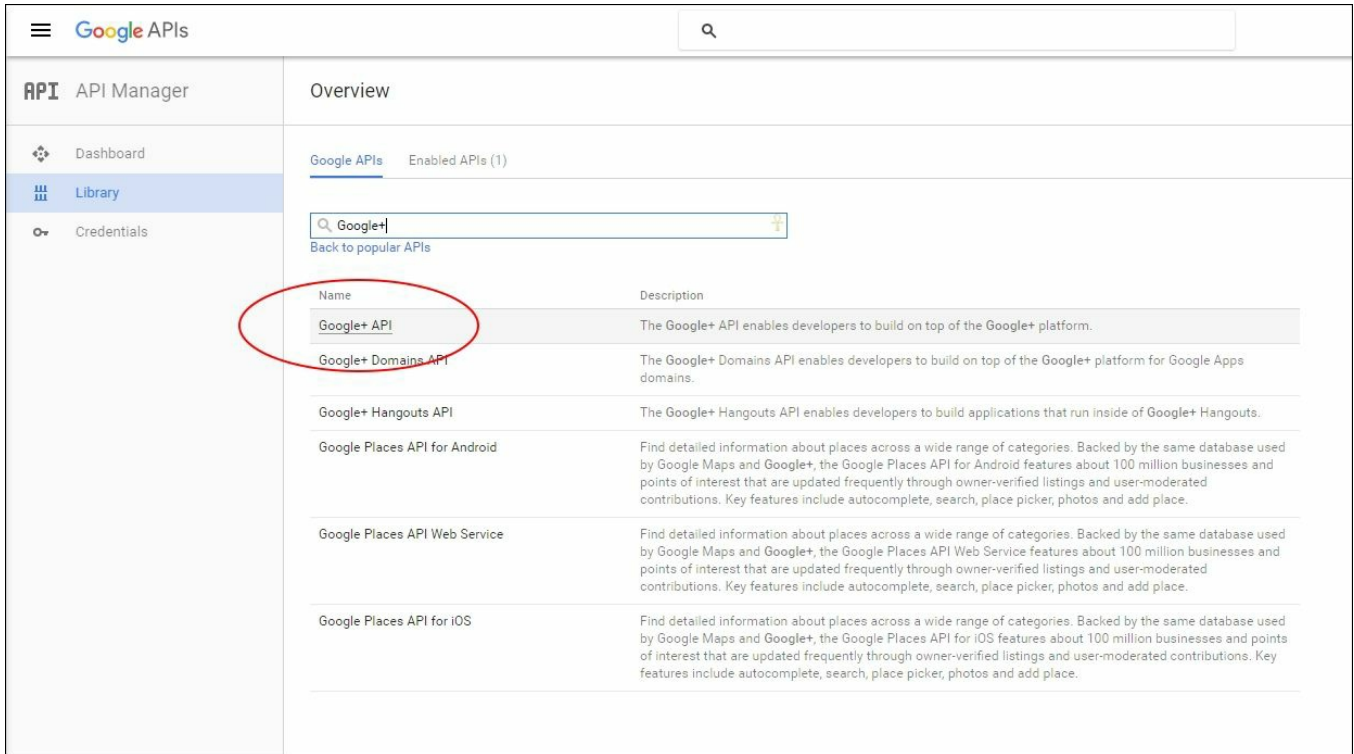
Select it, then click **Enable** to add the **Google OAuth2 authorization flow** support to our application.

Once done, we have to add the Google authentication package to our `project.json` file, right below the Facebook one:

```
"Microsoft.AspNetCore.Authentication.Google": "1.0.0"
```

Now we can open the `Startup.cs` file and add the Google Authentication middleware to the `Configure` method, right below the Facebook one:

```
app.UseGoogleAuthentication(new GoogleOptions()
{
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    ClientId = Configuration["Authentication:Google:ClientId"],
    ClientSecret =
Configuration["Authentication:Google:ClientSecret"],
    CallbackPath = "/signin-google",
    Scope = { "email" }
```

```
});
```

Last but not least, we need to add the Login with Google button within the HTML `template` of our Angular 2 application's `AppComponent` class. Open the `/Scripts/app/app.component.ts` file and add the following right below the existing Facebook button:

```
<button class="btn btn-sm btn-default btn-block" type="submit"
(click)="callExternalLogin('Google')">
    Login with Google
</button>
```

... and we're done.

Twitter

Adding Twitter to the loop is relatively easy, as the Twitter API is way simpler to use than Facebook's and Google's. However, it also has some pretty serious limitations when compared to their big brothers:

- It doesn't natively allow us to request the user's e-mail address, so we'll need to do an additional step.
- It requires a public facing URL to authenticate users. It won't work with `localhost` or unreachable hostnames.

To add it to our supported external providers list, go to https://apps.twitter.com/, sign in using your Twitter account, click on the **Add Application** button, then fill in the relevant data on the **Create an application** form that will appear:

Create an application

Application Details

Name *

    OpenGameList

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

    A non-comprehensive directory of open-source video games available on the web

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

    http://www.opengamelist.com/

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.

(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

As soon as we accept the **Developer Agreement**, we'll be redirected to our new Twitter application's settings page. There we can retrieve the application's `ConsumerKey` and `ConsumerSecret` and paste them into our `appsettings.json` file, right below the Facebook and Google ones:

```
"Twitter": {
  "ConsumerKey": "___TWITTER_CONSUMER_KEY___",
  "ConsumerSecret": "___TWITTER_SECRET_KEY___"
```

```
}
```

Right after that, just like we did with Facebook and Google, we need to add the Twitter authentication package to our `project.json` file:

```
"Microsoft.AspNetCore.Authentication.Twitter": "1.0.0"
```

We also need to add the corresponding middleware to the `Startup.cs` file:

```
app.UseTwitterAuthentication(new TwitterOptions()
{
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    ConsumerKey =
Configuration["Authentication:Twitter:ConsumerKey"],
    ConsumerSecret =
Configuration["Authentication:Twitter:ConsumerSecret"],
    CallbackPath = "/signin-twitter"
});
```

And we also add the HTML button to the `/Scripts/app/app.component.ts` file:

```
<button class="btn btn-sm btn-default btn-block" type="submit"
(click)="callExternalLogin('Twitter')">
    Login with Twitter
</button>
```

Elevated permissions request

We're done coding, but we're not done yet because Twitter, just as we said at the beginning of this paragraph, won't natively send us the user's e-mail address.

To fix that, we need to manually fill in an elevated permissions request form where we ask Twitter to give to our application the chance to request e-mail addresses via OAuth2.

In order to do that, we need to perform the following steps:

1. Visit the following URL: https://support.twitter.com/forms/platform.
2. Locate the radio button list and select **I need access to special permissions**.
3. Enter the same **Application Name** we used when we registered our application with Twitter.
4. Enter the **Application ID**. This can be obtained by going to https://apps.twitter.com/, selecting our Twitter app, and then looking at the numeric parameter contained at the end of the URL itself.
5. Write the following within the **Authorization Request** text area: **E-Mail Address for OAuth2 sign-in**.

6. Click **Submit** button and wait for an e-mail response that will hopefully come in few hours. The e-mail will contain the instruction we need to follow to be able to request the user e-mail address. Until then the Twitter authentication won't work, as our `AccountsControllerExternalLoginCallback` method will throw an exception.

Troubleshooting

Learning how to properly handle external authentication providers is one of the most difficult aspects of this whole book. There are a lot of things that can go wrong, most of them not even under our control: `OpenIddict` behavior, Facebook/Google/Twitter configuration, permission handling, missing or incorrect endpoint URIs, security issues, and so on.

Understanding everything will require time and can be frustrating at times. That said, these are some useful suggestions that could save you a headache or two:

- We should always check our development browser's console log and network tabs, as these are an endless source of info regarding our web application's overall status: **HTTP 404** and/or **500** errors, missing resources, invalid JavaScript code, and so on, not to mention the console messages sent by our very own code.
- Every time we get an **Access Denied** exception, we should clean all browser cookies for our web application, especially the one called `Identity.External`, and also the relevant entry in our browser's `localStorage`. Both of them can be easily managed using the **Application** tab of the *Google Chrome* development tools, accessible by pressing *Shift + Ctrl + J*.
- Every time our .NET application crashes without satisfying browser output documenting the exception being thrown, we should check the **Output** tab of Visual Studio 2015. A lot of .NET Core libraries write their exceptions there.
- External OAuth2 providers will always give us the option to delete/deauthorize our web application from their web interface. Whenever we encounter problems with authentication, the best thing we can do to retrace our steps is to do precisely that and start over.

Conclusions

Here's how our application's login view should look after all this effort:
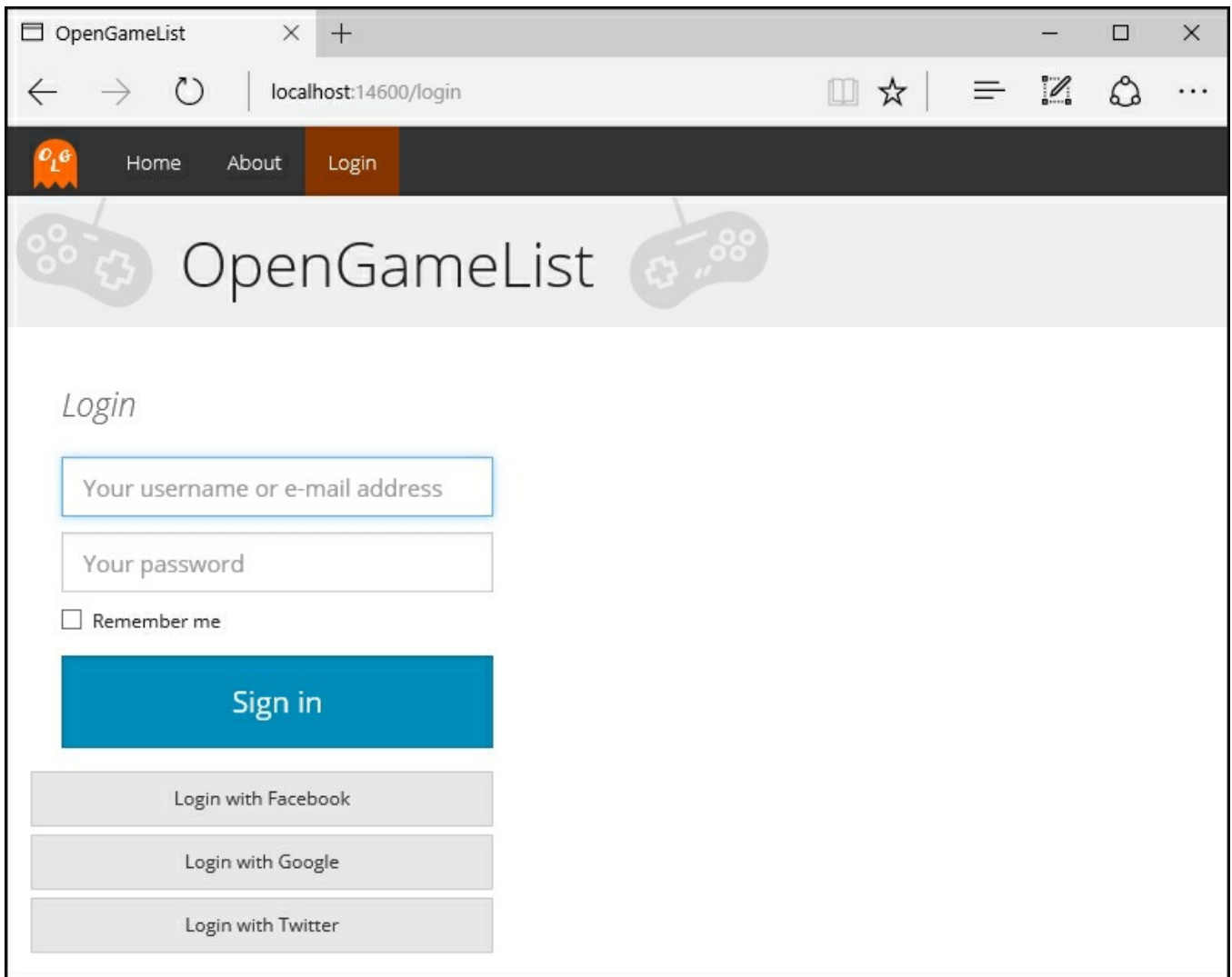
With this new authentication system, our application is now starting to become a potentially shippable product. However, we don't have a user profile view to handle user registrations, e-mail/password changes, and so on.

This feature, together with some other features, will be addressed in the following chapter.

Suggested topics

OpenID Connect, OpenIddict, OAuth2, OpenID Connect, RFC 6749, Secret Manager, DRY, WET, Zones, Zone.js, LocalStorage, Cookies.

Summary

Our `JwtProvider` was working fine, but it lacked some quite important features that had to be implemented as well in order to use it in a production-ready application such as the one we're aiming to build. To enable us to do that, we also identified **OpenIddict**, a viable open-source OAuth2/OpenID Connect provider that leverages ASP.NET Core Identity and ASOS, as a viable alternative, thus giving ourselves the choice between going for it and sticking with our handmade solution.

As soon as we chose our path, we implemented some external OAuth2 authentication providers such as Facebook, Google, and Twitter. The first was also the toughest one, as we needed to create our web API interface and understand how to properly handle the various scenarios for the first time. We definitely had an easier time with the other two, even if each one of them presented their own set of issues we had to address.

Chapter 9. User Registration and Account Edit

In this chapter, we'll be adding a couple more account-related features that are still missing: user registration and edit user info.

User Registration

To implement a **User Registration** logic, we need to take care of the following tasks:

- Create a `UserViewModel` to send and receive data from and to the Web API, just like we did with items back in [Chapter 2](#), *ASP.NET Controllers and Server-Side Routes*
- Add the `Get`, `Add`, `Update`, and `Delete` RESTful methods to the `AccountsController` Web API interface
- Handle all these methods in Angular 2 by updating the `AuthService` class
- Create a `UserEditComponent` class in our Angular 2 client to host the registration form, hooking it to the `AuthService` accordingly
- Update the other Angular 2 components, together with the existing client-side routing structure, in order to properly integrate it within the application workflow

Let's get it done.

UserViewModel

Right-click on the `/ViewModels/` folder and add a new C# class, naming it `UserViewModel.cs`. Replace the default content with the following code:

```csharp
using System;
using System.ComponentModel;
using Newtonsoft.Json;

namespace OpenGameListWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class UserViewModel
    {
        #region Constructor
        public UserViewModel()
        {

        }
        #endregion Constructor

        #region Properties
        public string UserName { get; set; }
        public string Password { get; set; }
        public string PasswordNew { get; set; }
        public string Email { get; set; }
        public string DisplayName { get; set; }
        #endregion Properties
    }
}
```

We only declared the properties we're going to send to and/or receive from our Angular 2 client, just like we did with the `ItemViewModel`. The `PasswordNew` property is the only special case since it doesn't exist in the `ApplicationUser` class. We're going to use it by the end of this chapter, when we'll give our users some account-editing capabilities, such as being able to change their password.

AccountsController

Open the `/Controllers/AccountsController.cs` file we created in [Chapter 8](), *Third-Party Authentication and External Providers*, then add the following `region` between the existing constructor and external authentication providers regions:

```
#region RESTful Conventions
#endregion RESTful Conventions
```

This is where we're going to implement the `Get`, `Add`, `Update`, and `Delete` methods to properly handle all requests regarding user accounts.

Get

The `Get` request is commonly used any time the client wants to retrieve updated info regarding a given user.

To properly handle such requests, we need to define at least two implementations: a parameterless one, returning the currently active/logged in account (if any), and another one that will return the user corresponding to the requested ID.

The former is often used when the client needs to show the user their data, usually within a standard *view/edit account info* screen; the latter is required whenever we have a clickable user listing of any sort, or any other control that enables a visitor to see their fellow user's info:

```csharp
/// <summary>
/// GET: api/accounts
/// </summary>
/// <returns>A Json-serialized object representing the current
account.</returns>
[HttpGet()]
public async Task<IActionResult> Get()
{
    var id = await GetCurrentUserId();
    var user = DbContext.Users.Where(i => i.Id ==
id).FirstOrDefault();
    if (user != null) return new JsonResult(new UserViewModel()
    {
UserName = user.UserName,
Email = user.Email,
DisplayName = user.DisplayName
    }, DefaultJsonSettings);
    else return NotFound(new { error = String.Format("User ID {0}
has not been found", id) });
}
```

```
/// <summary>
/// GET: api/accounts/{id}
/// ROUTING TYPE: attribute-based
/// </summary>
/// <returns>A Json-serialized object representing a single
account.</returns>
[httpget("{id}")]
public IActionResult Get(string id)
{
    return BadRequest(new { error = "not implemented (yet)." });
}
```

That's it. We don't have to implement the latter now, as we won't be using it by our application in this phase. We don't plan to add a clickable list of active and/or registered users, so we'll just emit a bad request HTTP error for now.

Add

This is a standard request to create a new user. We expect it to come right after the (new) user submitted a properly filled in registration form, so we restrict the method to POST requests only by using the `HttpPost` attribute:

```
/// <summary>
/// POST: api/accounts
/// </summary>
/// <returns>Creates a new User and return it accordingly.
</returns>
[HttpPost()]
public async Task<IActionResult> Add([FromBody]UserViewModel uvm)
{
    if (uvm != null)
    {
        try
        {
            // check if the Username/Email already exists
            ApplicationUser user = await
UserManager.FindByNameAsync(uvm.UserName);
            if (user != null) throw new Exception("UserName already
exists.");
            user = await UserManager.FindByEmailAsync(uvm.Email);
            if (user != null) throw new Exception("E-Mail already
exists.");

            var now = DateTime.Now;

            // create a new Item with the client-sent json data
            user = new ApplicationUser()
            {
                UserName = uvm.UserName,
                Email = uvm.Email,
                CreatedDate = now,
```

```
                LastModifiedDate = now
            };

            // Add the user to the Db with a random password
            await UserManager.CreateAsync(user, uvm.Password);

            // Assign the user to the 'Registered' role.
            await UserManager.AddToRoleAsync(user, -Registered");

            // Remove Lockout and E-Mail confirmation
            user.EmailConfirmed = true;
            user.LockoutEnabled = false;

            // persist the changes into the Database.
            DbContext.SaveChanges();

            // return the newly-created User to the client.
            return new JsonResult(new UserViewModel()
            {
                UserName = user.UserName,
                Email = user.Email,
                DisplayName = user.DisplayName
            }, DefaultJsonSettings);
        }
        catch (Exception e)
        {
            // return the error.
            return new JsonResult(new { error = e.Message });
        }
    }

    // return a generic HTTP Status 500 (Not Found) if the client
payload is invalid.
    return new StatusCodeResult(500);
}
```

The implementation is quite long, but the code should be understandable enough. We check if the given username and/or e-mail address exist in our users archive. If they do, we emit an error message; otherwise, we create a new user account and return its relevant info in JSON format.

E-mail confirmation

We won't hide the fact that, for the sake of simplicity, we're purposely skipping the e-mail confirmation step, which has always been a distinctive feature of ASP.NET Identity as well as any existing web application since the dawn of time.

Luckily enough, implementing such a mechanism in ASP.NET Core isn't going to be hard at all. The task could be basically split into three relevant steps:

1. Instead of setting the user's `EmailConfirmed` property to `true` like we did, we should issue a call to the `GenerateEmailConfirmationTokenAsync()` method of `UserManager`.
2. The confirmation token should then be sent to the user's e-mail address so they can send it back to our web application as a URL parameter or inside an e-mail activation form.
3. Eventually, we should be able to validate it via a dedicated `AccountController` Web API using the `UserManager.ConfirmEmailAsync()` method.

Pretty straightforward, isn't it? Implementing it before going further is a great exercise to test our skills up to this point.

Update

The `Update` method is used to alter the data of an existing user. It's often called at the end of an edit profile info form of any sort, assuming that the currently active/logged in user is updating their own data. It can also be used by high-privileged users, such as administrators, to modify other user's data. However, we won't be supporting that within our application's first release, so we might as well skip that:

```
/// <summary>
/// PUT: api/accounts/{id}
/// </summary>
/// <returns>Updates current User and return it accordingly.
</returns>
[HttpPut]
[Authorize]
public async Task<IActionResult> Update([FromBody]UserViewModel
uvm)
{
    if (uvm != null)
    {
        try
        {
            // retrieve user
            var id = await GetCurrentUserId();
            ApplicationUser user = await
UserManager.FindByIdAsync(id);
            if (user == null) throw new Exception("User not
found");

            // check for current password
            if (await UserManager.CheckPasswordAsync(user,
uvm.Password))
            {
                // current password ok, perform changes (if any)
```

```csharp
                bool hadChanges = false;

                if (user.Email != uvm.Email)
                {
                    // check if the Email already exists
                    ApplicationUser user2 = await
UserManager.FindByEmailAsync(uvm.Email);
                    if (user2 != null && user.Id != user2.Id) throw
new Exception("E-Mail already exists.");
                    else await UserManager.SetEmailAsync(user,
uvm.Email);
                    hadChanges = true;
                }

                if (!string.IsNullOrEmpty(uvm.PasswordNew))
                {
                    await UserManager.ChangePasswordAsync(user,
uvm.Password, uvm.PasswordNew);
                    hadChanges = true;
                }

                if (user.DisplayName != uvm.DisplayName)
                {
                    user.DisplayName = uvm.DisplayName;
                    hadChanges = true;
                }

                if (hadChanges)
                {
                    // if we had at least 1 change:
                    // update LastModifiedDate
                    user.LastModifiedDate = DateTime.Now;
                    // persist the changes into the Database.
                    DbContext.SaveChanges();
                }

                // return the updated User to the client.
                return new JsonResult(new UserViewModel()
                {
                    UserName = user.UserName,
                    Email = user.Email,
                    DisplayName = user.DisplayName
                }, DefaultJsonSettings);
            }
            else throw new Exception("Old password mismatch");
        }
        catch (Exception e)
        {
            // return the error.
            return new JsonResult(new { error = e.Message });
        }
    }
```

```
    // return a HTTP Status 404 (Not Found) if we couldn't find a
suitable item.
    return NotFound(new { error = String.Format("Current User has
not been found") });
}
```

Despite the fair amount of code, we should be able to understand everything. The first thing we do is check for the user's current password, which is required to perform any changes. If the password matches, we update the user fields accordingly with the new data. If the e-mail has been changed, we also check within our database to see if it already belongs to anyone else; if it does, we emit an error message. Otherwise, we go ahead.

If all checks pass, we update the `LastModifiedDate`, persist the new user info within our database, and return an updated set of data in JSON format; otherwise, we return a HTTP `Error 500`.

Delete

The `Delete` method is almost always a prerogative of administrators unless we want to allow our users to delete themselves. We're not planning to do that, neither are we building an administration panel in this phase, so we can safely skip both implementations for the time being:

```
/// <summary>
/// DELETE: api/accounts/
/// </summary>
/// <returns>Deletes current User, returning a HTTP status 200 (ok)
when done.</returns>
[HttpDelete()]
[Authorize]
public IActionResult Delete()
{
    return BadRequest(new { error = "not implemented (yet)." });
}

/// <summary>
/// DELETE: api/accounts/{id}
/// </summary>
/// <returns>Deletes an User, returning a HTTP status 200 (ok) when
done.</returns>
[HttpDelete("{id}")]
[Authorize]
public IActionResult Delete(string id)
{
    return BadRequest(new { error = "not implemented (yet)." });}
```

Now our Web API is ready to handle the most basic requests for adding and updating

user accounts. We're going to call these new methods with our Angular 2 client in a short while.

User class

Time to switch to Angular 2. The first thing we need to do here is to add an Angular 2 `User` class. Right-click on `/Scripts/app/`, create a new `user.ts` TypeScript file, and fill it with the following code:

```
export class User {
    constructor(
        public UserName: string,
        public Password: string,
        public PasswordNew: string,
        public Email: string,
        public DisplayName: string) {}
}
```

This will be the client-side counterpart of the `UserViewModel` Web API class we just made.

AuthService

Open the `/Scripts/app/auth.service.ts` file and add the following import reference:

```
import {User} from "./user";
```

Then, implement the following methods:

```
get() {
    return this.http.get("api/Accounts")
        .map(response => response.json());
}

add(user: User) {
    return this.http.post(
        "api/Accounts",
        JSON.stringify(user),
        new RequestOptions({
            headers: new Headers({
                "Content-Type": "application/json"
            })
        }))
        .map(response => response.json());
}

update(user: User) {
    return this.http.put(
        "api/Accounts",
        JSON.stringify(user),
        new RequestOptions({
            headers: new Headers({
                "Content-Type": "application/json"
            })
        }))
        .map(response => response.json());
}
```

There's nothing special to explain here; we're just calling the Web APIs we added before, returning the resulting JSON objects.

UserEditComponent

Now we can create our **User Registration** form. Right-click on the `/Scripts/app/` folder and add a new `user-edit.component.ts` TypeScript file. Since the code is a bit complex, we'll split it up into parts.

Let's start with the `import` statements:

```
import {Component, OnInit} from "@angular/core";
import {FormBuilder, FormControl, FormGroup, Validators} from
"@angular/forms";
import {Router} from "@angular/router";
import {AuthService} from "./auth.service";
import {User} from "./user";
```

As we can see, we're referencing a lot of stuff here. By looking at the components we're pulling off from the `@angular2/forms` package, we can already guess that we're going to build a form using the model-driven pattern. We already used that when we built our minimalistic login form, yet we'll definitely see more about it here.

Here's the `@Component` section source code:

```
@Component({
    selector: "user-edit",
    template: `
<div class="user-container">
    <form class="form-user" [formGroup]="userForm"
(submit)="onSubmit()">
        <h2 class="form-user-heading">{{title}}</h2>
        <div class="form-group">
            <input formControlName="username" type="text"
class="form-control" placeholder="Choose an Username" autofocus />
            <span class="validator-label valid"
*ngIf="this.userForm.controls.username.valid">
                <span class="glyphicon glyphicon-ok" aria-
hidden="true"></span>
                valid!
            </span>
            <span class="validator-label invalid"
*ngIf="!this.userForm.controls.username.valid &&
!this.userForm.controls.username.pristine">
                <span class="glyphicon glyphicon-remove" aria-
hidden="true"></span>
                invalid
            </span>
        </div>
        <div class="form-group">
```

```html
        <input formControlName="email" type="text" class="form-
control" placeholder="Type your e-mail address" />
        <span class="validator-label valid"
*ngIf="this.userForm.controls.email.valid">
            <span class="glyphicon glyphicon-ok" aria-
hidden="true"></span>
            valid!
        </span>
        <span class="validator-label invalid"
*ngIf="!this.userForm.controls.email.valid &&
!this.userForm.controls.email.pristine">
            <span class="glyphicon glyphicon-remove" aria-
hidden="true"></span>
            invalid
        </span>
    </div>
    <div class="form-group">
        <input formControlName="password" type="password"
class="form-control" placeholder="Choose a Password" />
        <span class="validator-label valid"
*ngIf="this.userForm.controls.password.valid &&
!this.userForm.controls.password.pristine">
            <span class="glyphicon glyphicon-ok" aria-
hidden="true"></span>
            valid!
        </span>
        <span class="validator-label invalid"
*ngIf="!this.userForm.controls.password.valid &&
!this.userForm.controls.password.pristine">
            <span class="glyphicon glyphicon-remove" aria-
hidden="true"></span>
            invalid
        </span>
    </div>
    <div class="form-group">
        <input formControlName="passwordConfirm"
type="password" class="form-control" placeholder="Confirm your
Password" />
        <span class="validator-label valid"
*ngIf="this.userForm.controls.passwordConfirm.valid &&
!this.userForm.controls.password.pristine &&
!this.userForm.hasError('compareFailed')">
            <span class="glyphicon glyphicon-ok" aria-
hidden="true"></span>
            valid!
        </span>
        <span class="validator-label invalid" *ngIf="
(!this.userForm.controls.passwordConfirm.valid &&
!this.userForm.controls.passwordConfirm.pristine) ||
this.userForm.hasError('compareFailed')">
            <span class="glyphicon glyphicon-remove" aria-
hidden="true"></span>
```

```
                        invalid
                </span>
            </div>
            <div class="form-group">
                <input formControlName="displayName" type="text"
class="form-control" placeholder="Choose a Display Name" />
            </div>
            <div class="form-group">
                <input type="submit" class="btn btn-primary btn-block"
[disabled]="!userForm.valid" value="Register" />
            </div>
        </form>
</div>
    `
})
```

As expected, there's a lot of new stuff here. We made extensive use of the `valid`
property exposed by the `FormGroup` container and also by their inner `FormControl`
input controls, as it's a convenient way to check the status of these items in real time.

We're using these values to change the GUI behavior in a number of ways, including
disabling the **Register** button until the form is valid:

```
<input type="submit" class="btn btn-primary btn-block"
[disabled]="!userForm.valid" value="Register" />
```

Both the `Form` components and their `Validators` are set in the `class` source code,
which we split into three parts for better reading. The first one features the
`constructor`, which instantiates the services we'll be using via dependency
injection:

```
export class UserEditComponent {
    title = "New User Registration";
    userForm: FormGroup = null;
    errorMessage = null;

    constructor(
        private fb: FormBuilder,
        private router: Router,
        private authService: AuthService) {
        if (this.authService.isLoggedIn()) {

            this.router.navigate([""]);

        }
    }
```

We know these objects already, so there's no need to explain them again. We can even
recognize the highlighted part, as we already used it in our `LoginComponent` for the

same purpose. If the user is already logged in, they shouldn't be allowed to fill in a registration form, so we're redirecting them to the **Welcome View**.

Here's the `ngOnInit` method, where we initialize the form using `FormBuilder`:

```
ngOnInit() {
    this.userForm = this.fb.group(
        {
            username: ["", [
                Validators.required,
                Validators.pattern("[a-zA-Z0-9]+")
                ]],
            email: ["", [
                Validators.required,
                Validators.pattern("[a-z0-9!#$%&'*+/=?^_`{|}~-]+
(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?")
                ]],
            password: ["", [
                Validators.required,
                Validators.minLength(6)]],
            passwordConfirm: ["", [
                Validators.required,
                Validators.minLength(6)]],
            displayName: ["", null]
        },
        {
            validator: this.compareValidator('password',
'passwordConfirm')
        }
    );
}
```

As we can see, we're using a wide set of `Validators` here. Most of them are shipped with the `@angular2/forms` package:

- `Validators.required` is an old friend, as we already used it for our login form. Its behavior is quite self-explanatory: it will invalidate the control as long as its value is empty. We're using this on all our input controls except for `displayName`, since it's not a required field.
- `Validators.pattern` is basically a **regular expression (regex)** validator. It will check the control value against the given **regex** and invalidate it until it matches. We used it together with a very basic regex to allow only alphanumeric characters for the `username`, and then with a **RFC 2822** compliant regex to check for invalid e-mail addresses. Although it won't save us from scammers, it will do a decent job to save our real users from the most common typing errors.
- `Validators.minLength` does what its name suggests: it will invalidate the control until its value reaches the specified minimum character count.

The last validator we used, `this.compareValidator`, is a custom one: we made it to check if the `password` and `passwordConfirm` input values are the same or not and let the user know in real time, which is a common feature of modern registration forms. Technically, it's nothing more than a standard function that we can append right after the `ngOnInit` method:

```
compareValidator(fc1: string, fc2: string) {
    return (group: FormGroup): { [key: string]: any } => {
        let password = group.controls[fc1];
        let passwordConfirm = group.controls[fc2];
        if (password.value === passwordConfirm.value) {
            return null;
        }
        return { compareFailed: true }
    }
}
```

Note

It's worth noting that form-level validators such as this are required to either return `null` if there are no errors, or a `{ [key: string]: any }` object in case of failure. The used `key` can be checked within the `template` by using the `FormControl.hasError(key)` method, just like we did in the preceding form.

Right after that, we can implement the `onSubmit` function, which will be executed when the **Submit** button is clicked:

```
onSubmit() {
    this.authService.add(this.userForm.value)
        .subscribe((data) => {
            if (data.error == null) {
                // registration successful
                this.errorMessage = null;
                this.authService.login(
                    this.userForm.value.username,
                    this.userForm.value.password)
                    .subscribe((data) => {
                        // login successful
                        this.errorMessage = null;
                        this.router.navigate([""]);
                    },
                    (err) => {
                        console.log(err);
                        // login failure
                        this.errorMessage =
                            "Warning: Username or Password
mismatch";
                    });
            }
```

```
        else {
            // registration failure
            this.errorMessage = data.error;
        }
    },
    (err) => {
        // server/connection error
        this.errorMessage = err;
    });
}
```

As we can see, if the `onSubmit` method is completed without errors we will issue a call to `authService.login` to authenticate the user and, right after that, redirect them back to the **Welcome View**.

Tip

Don't forget to add a closing parenthesis right after the `onSubmit` method to close the `class` block.

Connecting the dots

Now that we have set up our controller/service/component chain, we need to wrap everything up within the client application's loop.

Updating the root module

Let's start by adding the new-born class to the application's root module. Open the `/Scripts/app/app.module.ts` file, then add the following line to the `import` references right after the `PageNotFoundComponent`:

```
import {UserEditComponent} from "./user-edit.component";
```

Don't forget to update the declarations array accordingly:

```
declarations: [
    AboutComponent,
    AppComponent,
    HomeComponent,
    ItemListComponent,
    ItemDetailEditComponent,
    ItemDetailViewComponent,
    LoginComponent,
    PageNotFoundComponent,
    UserEditComponent
],
```

Implementing the route

The next step will be adding the `register` route to the routing class file. Open the `/Scripts/app/app.routing.ts` file and add this to the `import` references:

```
import {UserEditComponent} from "./user-edit.component";
```

Then scroll down to the routing rules and add the following, right after the `Login` one:

```
{
    path: "register",
    component: UserEditComponent
},
```

That's it. Now that we've defined the route, we need to make good use of it.

Adding the Register link

Open the `/Scripts/app/login.component.ts` file and add this to the `import` references:

```
import {UserEditComponent} from "./user-edit.component";
```

Then scroll down to the `template` section and add the following (new lines highlighted):

```
<div class="login-container">
    <h2 class="form-login-heading">Login</h2>
    <div class="alert alert-danger" role="alert"
*ngIf="loginError">
        <strong>Warning:</strong> Username or Password mismatch
    </div>
    <form class="form-login" [formGroup]="loginForm"
(submit)="performLogin($event)">
        <input formControlName="username" type="text" class="form-
control" placeholder="Your username or e-mail address" required
autofocus />
        <input formControlName="password" type="password"
class="form-control" placeholder="Your password" required />
        <div class="checkbox">
            <label>
                <input type="checkbox" value="remember-me">
                Remember me
            </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Sign in</button>
    </form>
    <div class="register-link">

        Don't have an account yet?

        <a (click)="onRegister()">Click here to register!</a>

    </div>
    <button class="btn btn-sm btn-default btn-block" type="submit"
(click)="callExternalLogin('Facebook')">
        Login with Facebook
    </button>
    <button class="btn btn-sm btn-default btn-block" type="submit"
(click)="callExternalLogin('Google')">
        Login with Google
    </button>
</div>
```

Keep scrolling down and implement the `onRegister()` method within the class structure, right after the `performLogin()` method:

```
onRegister() {
    this.router.navigate(["register"]);
}
```

With this, we're done here as well.

Defining the styles

Open the `/Scripts/less/style.less` file, then append the following to style up the `UserEditComponent` we just made:

```less
.user-container {
    max-width: 500px;
    padding: 15px;
    .form-user {
        margin-left: 20px;
        input {
            margin-bottom: 10px;
            margin-right: 5px;
            max-width: 330px;
            display: inline;
        }
        .validator-label {
            &.valid {
                color: green;
            }
            &.invalid {
                color: red;
            }
        }
    }
    .register-link {
        cursor:pointer;
        font-size: 0.8em;
        margin: 10px 0;
    }
}
```

Right after that, add the CSS class we attached to the link panel that we put in the `UserLoginComponent` file:

```less
.register-link {
    cursor:pointer;
    font-size: 0.8em;
    margin: 10px 0;
}
```

Updating the menu

Last but not least, we should find a way to let our users know that our **Login** view now also features a link to the brand new **User Registration** feature we just made. The best thing we can do is to bring the information up to the menu itself.

Switch back to the `/Scripts/app/app.component.ts` file and modify the

navigation menu in the following way (updated code is highlighted):

```
<div class="collapse navbar-collapse" id="navbar">
    <ul class="nav navbar-nav">
        <li [class.active]="isActive([''])">
            <a class="home" [routerLink]="['']">Home</a>
        </li>
        <li [class.active]="isActive(['about'])">
            <a class="about" [routerLink]="['about']">About</a>
        </li>
        <li *ngIf="!authService.isLoggedIn()"
[class.active]="isActive(['login']) || isActive(['register'])">
            <a class="login" [routerLink]="['login']">Login /
Register</a>
        </li>
        <li *ngIf="authService.isLoggedIn()">
            <a class="logout" href="javascript:void(0)"
(click)="logout()">Logout</a>
        </li>
        <li *ngIf="authService.isLoggedIn()"
[class.active]="isActive(['item/edit', 0])">
            <a class="add" [routerLink]="['item/edit', 0]">Add
New</a>
        </li>
    </ul>
</div>
```

Testing it out

This is what our new user **Login** view will look like:
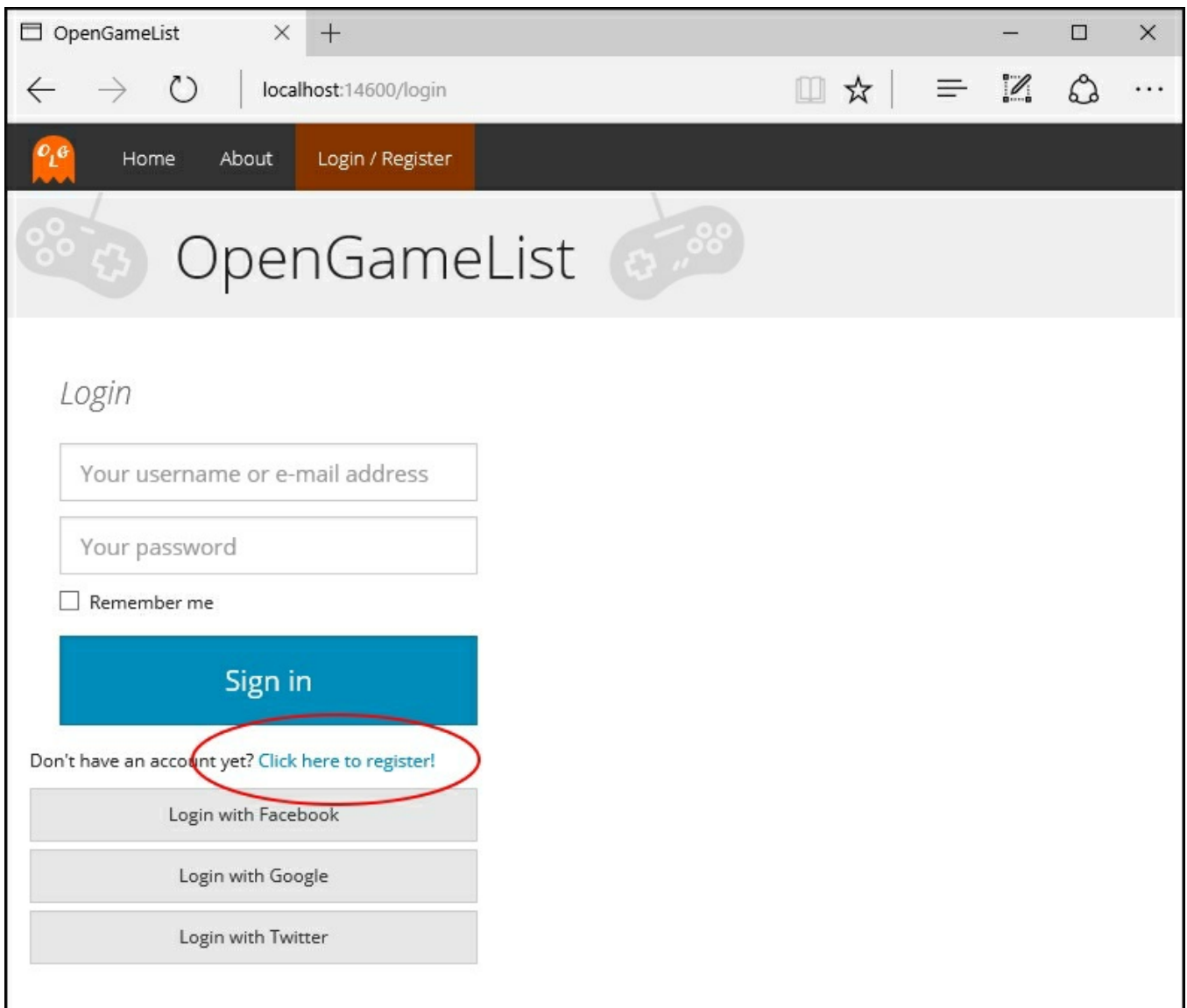
And here's the **User Registration** view, which is accessible by clicking on the **Click here to register!** link:
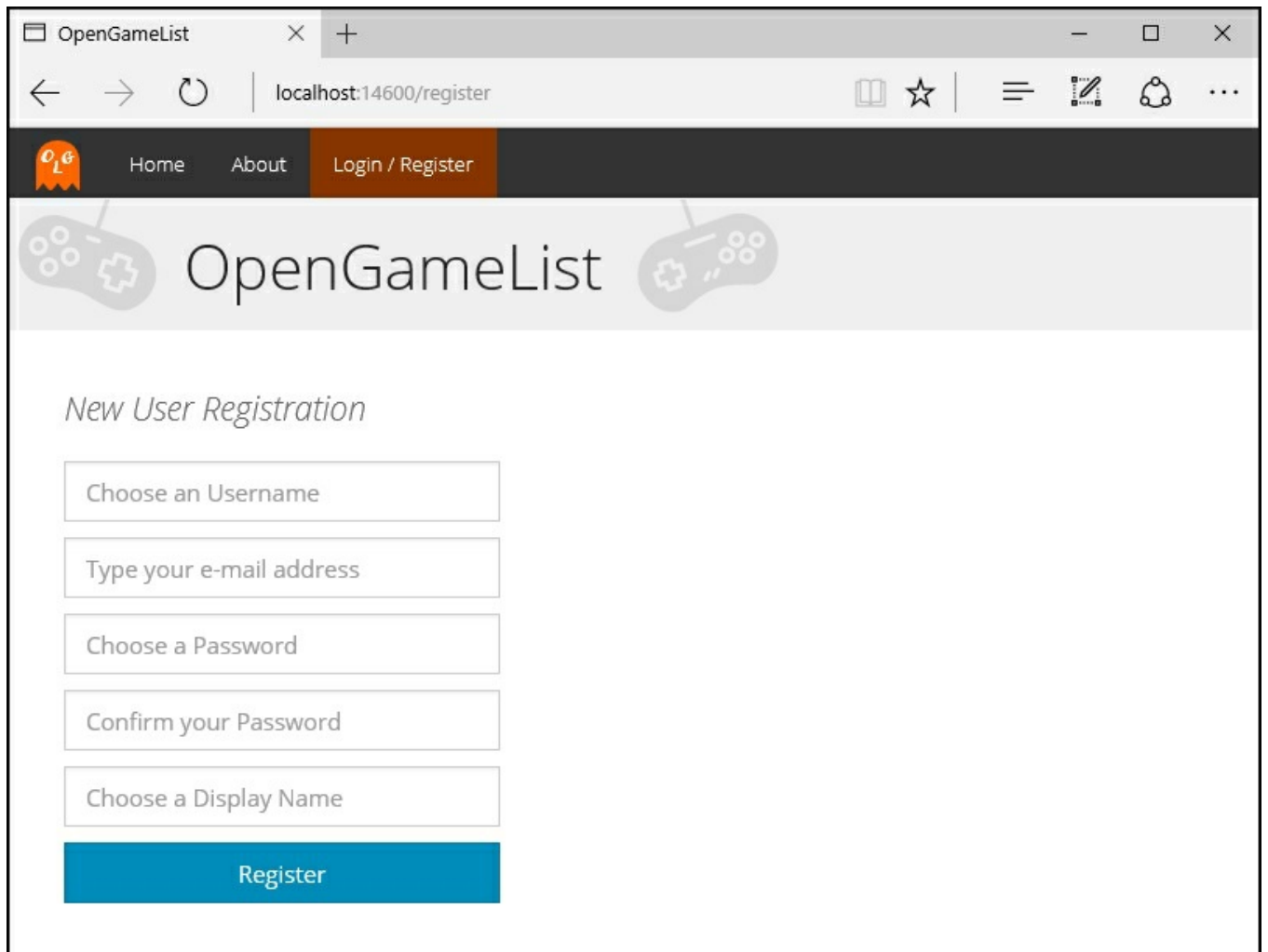
We can play a bit with the form to see how our module-driven approach actually works. As soon as we start typing something, we will get an instant feedback that will guide us to compile each field properly:
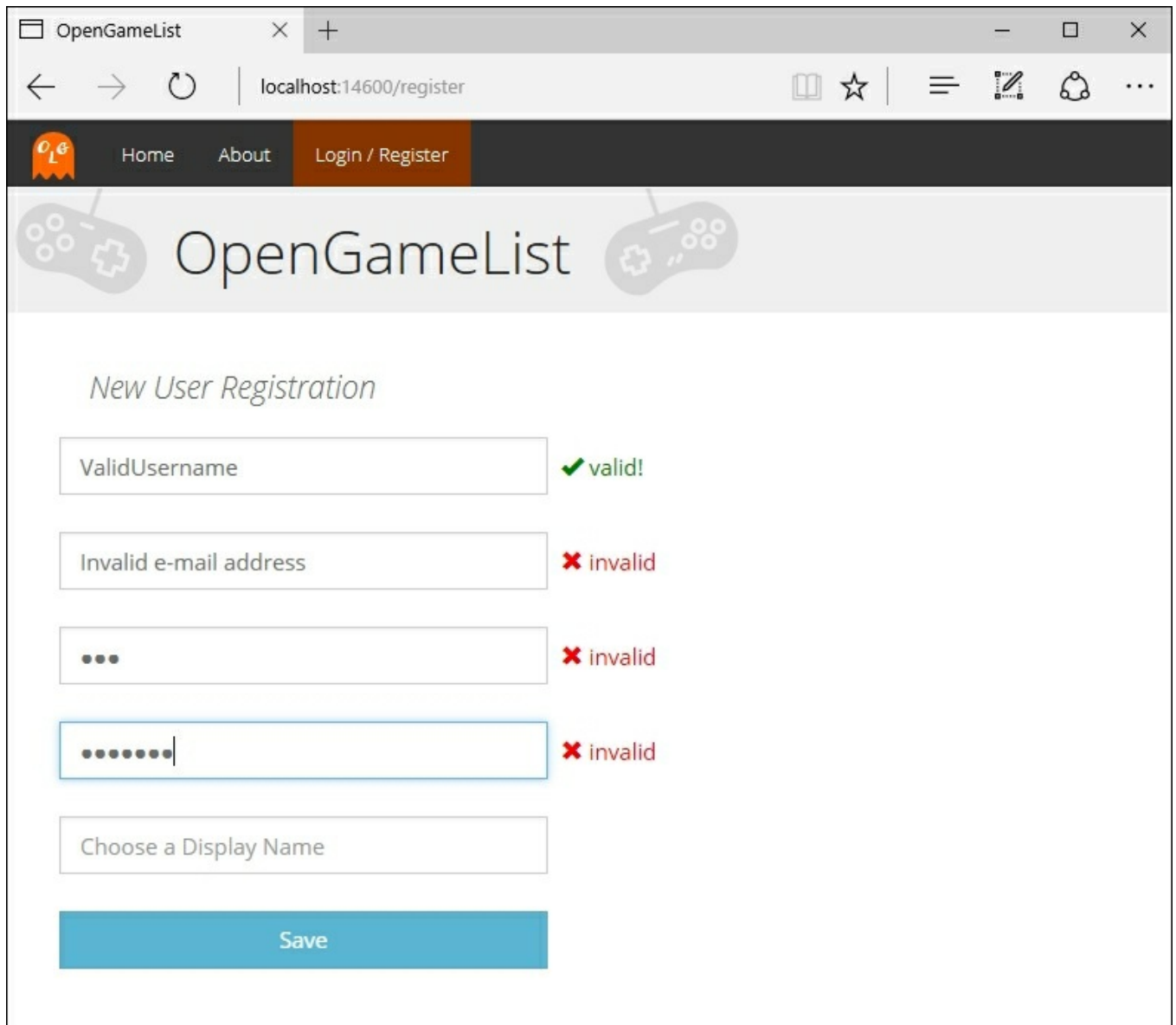
When we are ready to create our new account, we can click on the **Register** button and submit the form. A new user will be created and authenticated, then we will be redirected to the **Welcome View**.

Edit Account

Out of the three RESTful methods, we implemented in our Web API and `AuthService` interfaces, the whole **User Registration** view only took one. The `Get` and the `Update` methods are still there, waiting for us to make use of them.

The best thing we can do to test them both is to implement an **Edit Account** view to give our users the chance to change a limited set of their account info, such as e-mail address, password, and display name.

Luckily enough, we already did most of the job: the `UserEditComponent` already does most of what we need; we just have to tweak it a bit more so it can be used to update an existing account as well as register a new one.

Two routes, one component

In contrast with what we did for the `ItemDetailEditComponent`, we don't need to handle a `get` parameter because the account we want to update cannot be chosen: users will only be able to edit their own one. However, we still need to add an additional route, so we'll be able to distinguish between **User Registration** and an account update requests.

Open the `/Scripts/app/app.routing.ts` file and add the following route, right after the `register` one:

```
{
    path: "account",
    component: UserEditComponent
},
```

Adapting the UserEditComponent

The next thing we need to do is to fetch the currently active route within the `UserEditController` and act accordingly. Go back to the `/Scripts/app/user-edit.component.ts` file and add the `ActivatedRoute` class reference to the `@angular/router`import line:

```
import {Router, ActivatedRoute} from "@angular/router";
```

This will allow us to define `isRegister` Boolean variable and set it accordingly to the active route (new/updated lines are highlighted):

```
export class UserEditComponent {
    title = "New User Registration";
    userForm: FormGroup = null;
    errorMessage = null;
    isRegister: boolean;

    constructor(
        private fb: FormBuilder,
        private router: Router,
        private activatedRoute: ActivatedRoute,
        private authService: AuthService) {
        // determine behavior by fetching the active route

        this.isRegister = (activatedRoute.snapshot.url.toString()
=== "register");

        if ((this.isRegister && this.authService.isLoggedIn())

            || (!this.isRegister &&
!this.authService.isLoggedIn())) {

            this.router.navigate([""]);

        }

        if (!this.isRegister) {

            this.title = "Edit Account";

        }
    }
```

Once assigned, we immediately used the `isRegister` variable to improve our redirect strategy: as we don't want registered users to play with our registration form, we also can't allow unregistered users to edit their (non-existing) account. Right after that, we use it again to conditionally change the title to a most suited one.

The next change we need to do is at the end of the `ngOnInit` method (new lines are highlighted):

```
ngOnInit() {
    this.userForm = this.fb.group(
        {
            username: ["", [
                Validators.required,
                Validators.pattern("[a-zA-Z0-9]+")
            ]],
            email: ["", [
                Validators.required,
                Validators.pattern("[a-z0-9!#$%&'*+/=?^_`{|}~-]+
(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?")
            ]],
            password: ["", [
                Validators.required,
                Validators.minLength(6)]],
            passwordConfirm: ["", [
                Validators.required,
                Validators.minLength(6)]],
            displayName: ["", null]
        },
        {
            validator: this.compareValidator('password',
'passwordConfirm')
        }
    );

    if (!this.isRegister) {

        this.userForm.addControl("passwordCurrent",

            new FormControl("", Validators.required));

        var password = this.userForm.find("password");

        password.clearValidators();

        password.setValidators(Validators.minLength(6));

        var passwordConfirm =

            this.userForm.find("passwordConfirm");

        passwordConfirm.clearValidators();

        passwordConfirm.setValidators(Validators.minLength(6));

        this.authService.get().subscribe(
```

```
                    user => {

                        this.userForm.find("username")

                            .setValue(user.UserName);

                        this.userForm.find("email")

                            .setValue(user.Email);

                        this.userForm.find("displayName")

                            .setValue(user.DisplayName);

                    }

                );

            }
}
```

These few lines of code depict a number of tasks we need to perform whenever `this.isRegister` is `false`:

- Adding a new `passwordCurrent` control with a `required` validator as we'll use it to request the user's current password, which will be required to perform any change to their account.
- Removing the `required` validator from the `password` and `passwordConfirm` controls: the reason for this is pretty simple; when we're in **Edit Account** mode, these controls will be used to change the current password, which should never be a required operation. The only validator we're keeping for these controls is the `minLength(6)`. The user won't be forced to change it; however, if he chooses to do that, we won't accept anything less than the six-character length.
- Retrieving the current `UserName`, `Email`, and `DisplayName` values, so the user will be able to review them before performing any change. It's worth noting that they will be set asynchronously, as soon as the `authService` method `get` will fetch the data from the `AccountsController`.

Right after that, we need to perform the following changes to the `onSubmit` method (new/updated lines are highlighted):

```
onSubmit() {
    if (this.isRegister) {
        this.authService.add(this.userForm.value)
            .subscribe((data) => {
                if (data.error == null) {
                    // registration successful
                    this.errorMessage = null;
```

```
                    this.authService.login(
                        this.userForm.value.username,
                        this.userForm.value.password)
                        .subscribe((data) => {
                            // login successful
                            this.errorMessage = null;
                            this.router.navigate([""]);
                        },
                        (err) => {
                            console.log(err);
                            // login failure
                            this.errorMessage =
                                "Warning: Username or Password
mismatch";
                        });
                }
                else {
                    // registration failure
                    this.errorMessage = data.error;
                }
            },
            (err) => {
                // server/connection error
                this.errorMessage = err;
            });
    }

    else {

        let user = new User(

            this.userForm.value.username,

            this.userForm.value.password,

            this.userForm.value.passwordNew,

            this.userForm.value.email,

            this.userForm.value.displayName);

        this.authService.update(user)

            .subscribe((data) => {

                if (data.error == null) {

                    // update successful

                    this.errorMessage = null;

                    this.router.navigate([""]);
```

```
                    }

            else {

                // update failure

                this.errorMessage = data.error;

            }

        },

        (err) => {

            // server/connection error

            this.errorMessage = err;

        });

    }
}
```

There's nothing special here. We wrapped the `authService.add` method into an `if` condition to ensure that it will be executed only when the form is being used in **User Registration** mode; we then added a call to the `authService.update` method for the new **Edit Account** scenario.

We're not done yet. Before being able to test it, we need to make some changes to the `template` section too. Scroll up to it, look for the `<input>` HTML element that handles the username, and conditionally disable it in the following way (updated code is highlighted):

```
<input [disabled]="!this.isRegister" formControlName="username"
type="text" class="form-control" placeholder="Choose an Username"
autofocus />
```

This is required to make the user aware of the fact that won't be able to change its username.

Once done, scroll down to the `<div class="form-group">` containing the `password` input control and place the following code right before it:

```
<div *ngIf="!this.isRegister" class="form-group">
    <input formControlName="passwordCurrent" type="password"
class="form-control" placeholder="Current Password" />
    <span class="validator-label invalid"
*ngIf="!this.userForm.controls.passwordCurrent.valid">
```

```
        <span class="glyphicon glyphicon-remove" aria-
hidden="true"></span>
        required
    </span>
</div>
```

This is the HTML code for the `passwordCurrent` control we added programmatically in the `ngOnInit` method a short while ago. We already explained why we need this, so we can go ahead.

The last thing we need to do here is to apply a minor update to the submit button text value. We certainly don't want our users to click on **Register** when they are editing their account!

Scroll down to that HTML code and change it as follows (updated code is highlighted):

```
<div class="form-group">
    <input type="submit" class="btn btn-primary btn-block"
[disabled]="!userForm.valid" value="{{this.isRegister ? 'Register'
: 'Save'}}" />
</div>
```

That's much better. On top of that, we're now aware of the fact that Angular 2 Template Syntax allows the usage of ternary operators. That's great to know!

Updating the navigation menu

We're almost done. We just need to connect the `account` route with our navigation menu so that our users will be able to access it.

Open the `/Scripts/app/app.component.ts` file and add the following `<li>` element to the `<ul>` container in the `template` section, right under the **Add New** one:

```
<li *ngIf="authService.isLoggedIn()" class="right"
[class.active]="isActive(['account'])">
    <a [routerLink]="['account']">Edit Account</a>
</li>
```

Adding styles

We added the `right` CSS class to our new `<li>` element for a reason: we want our **Edit Account** menu item to appear near the upper-right corner of the browser screen, just like the majority of websites actually do. Needless to say, in order to make it happen, we also need to define a `.class` selector and configure it with the appropriate style sheet rules.

Open the `/Scripts/less/style.less` file and append the following code:

```
@media (min-width: 768px) {
    .nav.navbar-nav {
        float: none;
        li.right {
            float: right;
        }
    }
}
```

That's it. As we can see, we added some floating rules to put the menu item to the rightmost part of the screen. These rules are conveniently wrapped into a Media Query that will ensure they will be applied only when we have enough width (`768px` or more). This means that our desired right-most effect will work on a desktop environment only, leaving the mobile browsers layout unaffected.

Note

Media Queries are a powerful CSS3 feature. UI designers can use them to tailor their presentations to a specific range of output devices without changing the content of the page itself. To know more about them, we strongly suggest reading the following URL:

https://www.w3.org/TR/css3-mediaqueries/

Before going further, we should check that the **Task Runner** is actually running, so the compiled client code will be updated with our latest changes.

Final test

Now we can hit *F5* and see what our new **Edit Account** view looks like:
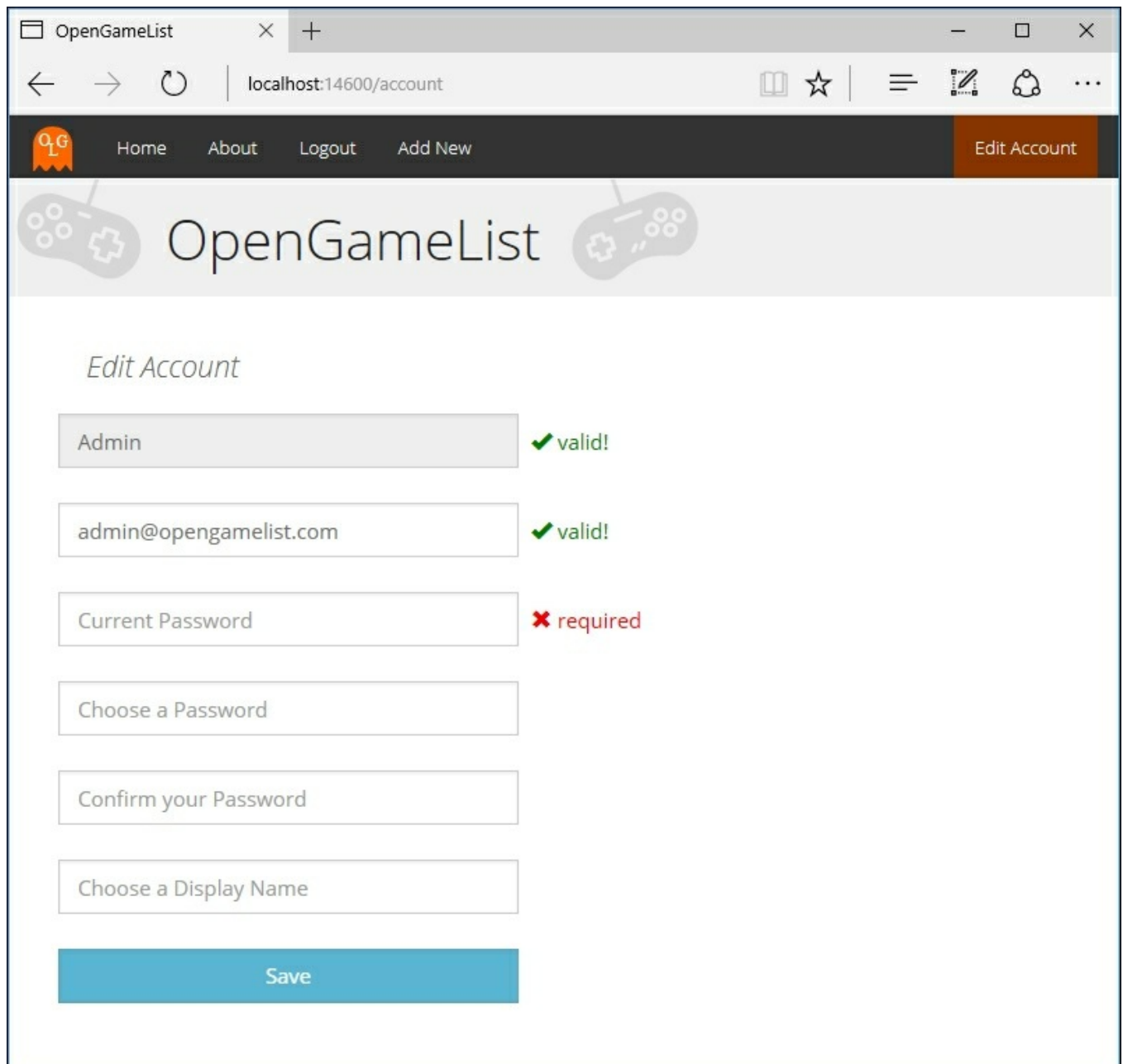
As we can see, the **Edit Account** menu item is shown on the *upper-right* corner of the screen. This is definitely a better place for all the options that controls the user account status, as they won't be confused with the *navigation* part of the main menu.

Wouldn't it be great if the **Login/Register** and **Logout** commands could be there too?

Let's take this chance to do that.

Open the `/Scripts/app/app.component.ts` file and rearrange the navigation menu `<li>` elements in the following way:

```
<ul class="nav navbar-nav">

    <!-- Navigation commands for everyone -->
    <li [class.active]="isActive([''])">
        <a class="home" [routerLink]="['']">Home</a>
    </li>
    <li [class.active]="isActive(['about'])">
        <a class="about" [routerLink]="['about']">About</a>
    </li>

    <!-- Navigation commands for authenticated users -->
    <li *ngIf="authService.isLoggedIn()"
[class.active]="isActive(['item/edit', 0])">
        <a class="add" [routerLink]="['item/edit', 0]">Add New</a>
    </li>

    <!-- Account-related commands -->
    <li *ngIf="!authService.isLoggedIn()" class="right"
[class.active]="isActive(['login']) || isActive(['register'])">
        <a class="login" [routerLink]="['login']">Login /
Register</a>
    </li>
    <li *ngIf="authService.isLoggedIn()" class="right">
        <a class="logout" href="javascript:void(0)"
(click)="logout()">Logout</a>
    </li>
    <li *ngIf="authService.isLoggedIn()" class="right"
[class.active]="isActive(['account'])">
        <a [routerLink]="['account']">Edit Account</a>
    </li>

</ul>
```

All we did here was reposition the content we already had add a couple HTML comments to split these elements into three groups: the navigation commands accessible to everyone, those restricted to authenticated users only, and the account/related ones. We also added the `right` CSS class to the **Login/Register** and **Logout** elements as we moved them into the latter group.

Before moving out, let's hit *F5* one last time to see what our improved menu looks like:

Conclusions

Our sample SPA application is mostly done. We are perfectly aware of the fact that a number of relevant features are still missing, such as the inability to refresh JWT Tokens, the lack of View Comments/Insert Comment components, the missing support for user-uploaded item image files, and so on. However, it is undoubtedly a potentially shippable product that will most likely meet our product owner's expectations.

All we have to do now is add some finishing touches and then publish it in a production environment, which is precisely what we're going to do in the next chapter.

Suggested topics

Template-driven forms, model-driven forms, FormBuilder, regex, Angular 2
Template Syntax, ternary operators, Media Queries.

Summary

Our native web application is slowly coming to an end, with most of its expected features ready and working. In this chapter, we implemented two of them that were still missing: a **User Registration** view, where new users can register themselves, and an **Edit Account** view they can use to change their e-mail address, password, and/or display name.

For each one of them, we added a Web API interface using standard RESTful methods, a set of Angular 2 `AuthService` methods, and a client-side routing strategy. While we were there, we also made some cosmetic changes to the navigation menu items and their corresponding style sheets to grant our visitors a better user experience.

Chapter 10. Finalization and Deployment

Our valuable journey through ASP.NET Core Web API and Angular 2 development is coming to an end. The native web application we've been working on since [Chapter 2](), *ASP.NET Controllers and Server-Side Routes*, is now a potentially shippable product, ready to be published in a suitable environment for evaluation purposes.

However, in order to do that, we need to give our project some finishing touches.

Switching to SQL Server

Although `localDB` proved itself to be a great development choice, it's not a good idea to use it in production as well. That's why we'll replace it with **SQL Server**. As for the chosen edition, we can either go for Express, Web, Standard, or Enterprise depending on what we need and/or can afford.

For the sake of simplicity, we'll be using SQL Server 2016 Express Edition, which can be downloaded for free from the following Microsoft official URL: https://www.microsoft.com/en-US/server-cloud/products/sql-server-editions/sql-server-express.aspx

Needless to say, we need to install it on a machine that is reachable from our web server via a **Local Area Network (LAN)**.

Installing SQL Server 2016 Express

The installation process is pretty straightforward. Unless we don't need anything specific, we can just go for the basic type:
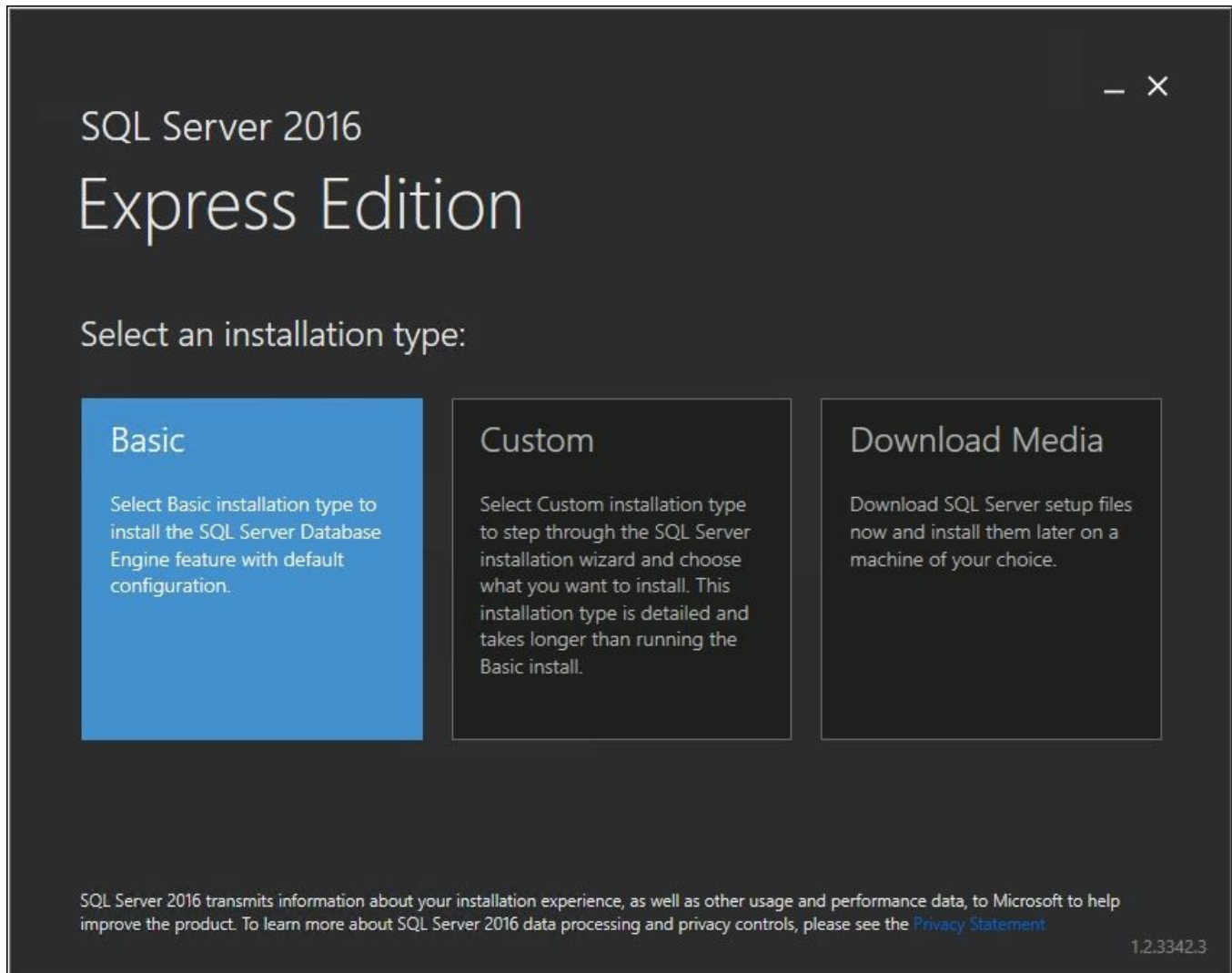
Eventually, we'll be prompted with an **Installation Complete** window, which will also give us some useful info, including the database instance name and a default connection string ready for a connection test:

Installing SQL Server Management Studio

From here, we can click the **Install tools** button and download SQL Server Management Studio, a tool that we can use to create the `OpenGameList` database and also a dedicated user that can access it.

Note

SQL Server Management Studio is a separate product available for free download at the following URL:

https://msdn.microsoft.com/en-us/library/mt238290.aspx

Configuring the database

Once we've downloaded and installed it, launch the SQL Server Management Studio. We will be prompted by a **Connect to Server** modal window that will allow us to connect to our local SQL Server instance.

To do this, select the **Database Engine** server type and then, from the **Server name** combo box, choose **<Browse for more...>**. Another pop-up window will appear, from which we'll be able to select the database engine we just installed on our server:
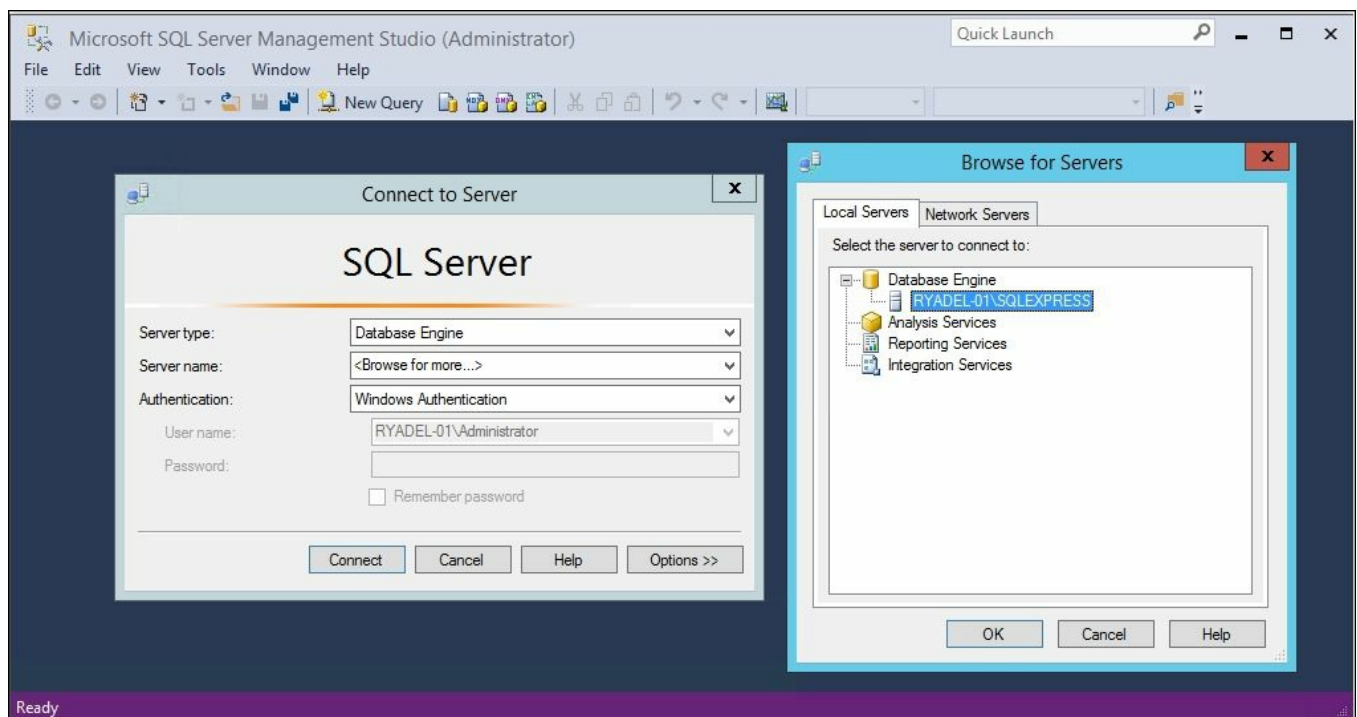
As for the **Authentication** part, we can leave **Windows Authentication**, the default SQL Server authentication mode.

When we're done, click on the **Connect** button and a **Server Explorer** window will appear, containing a tree view representing the structure of your SQL Server instance. This is the interface we'll use to create our database and also the user/password that our application will use to access it.

Tip

If you have a strong knowledge of SQL Server, you might want to skip the following steps and configure your instance as you prefer; otherwise, keep reading.

Changing the authentication mode

The first thing we need to do is to change the default SQL Server authentication mode, so we won't be forced to use an existing Windows account. To do so, right-click on the root tree view node, which represents our SQL Server instance, and select **Properties** from the contextual menu. From the modal window that appears, select the **Security** page, then switch from **Windows Authentication mode** to **SQL Server and Windows Authentication mode**:
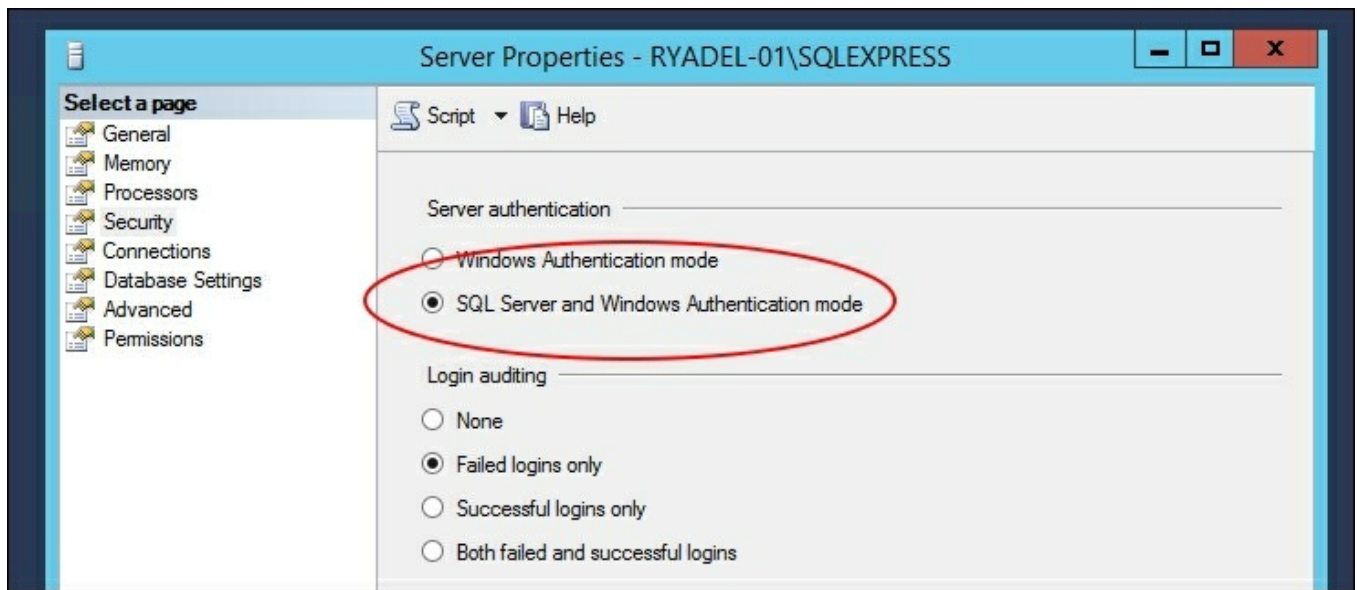
Adding the OpenGameList database

Now we can create the database that will host our application's tables. Right-click on the **Databases** folder and choose **Add Database** from the contextual menu. Give it the **OpenGameList** name and click on **OK**.

Adding the OpenGameList login

Go back to the root **Databases** folders, then expand the **Security** folder, which should be just below it. From there, right-click on the **Logins** subfolder and choose **New Login**. Again, give it the `OpenGameList` name. From the radio button list below, select **SQL Server Authentication** and set a suitable password (for example, `SamplePassword123`), then click on **OK**.

Tip

If you want a simpler password, such as `OpenGameList`, you might have to also disable the **enforce password policy** option.

Mapping the login to the database

The next thing we need to do is to properly map this login to the `OpenGameList` database we added earlier. From the navigation menu to the left, switch to the **User Mapping** tab. Click on the checkbox right to the `OpenGameList` database, then write `OpenGameList` in the **User** cell and assign the **db_owner** membership role:
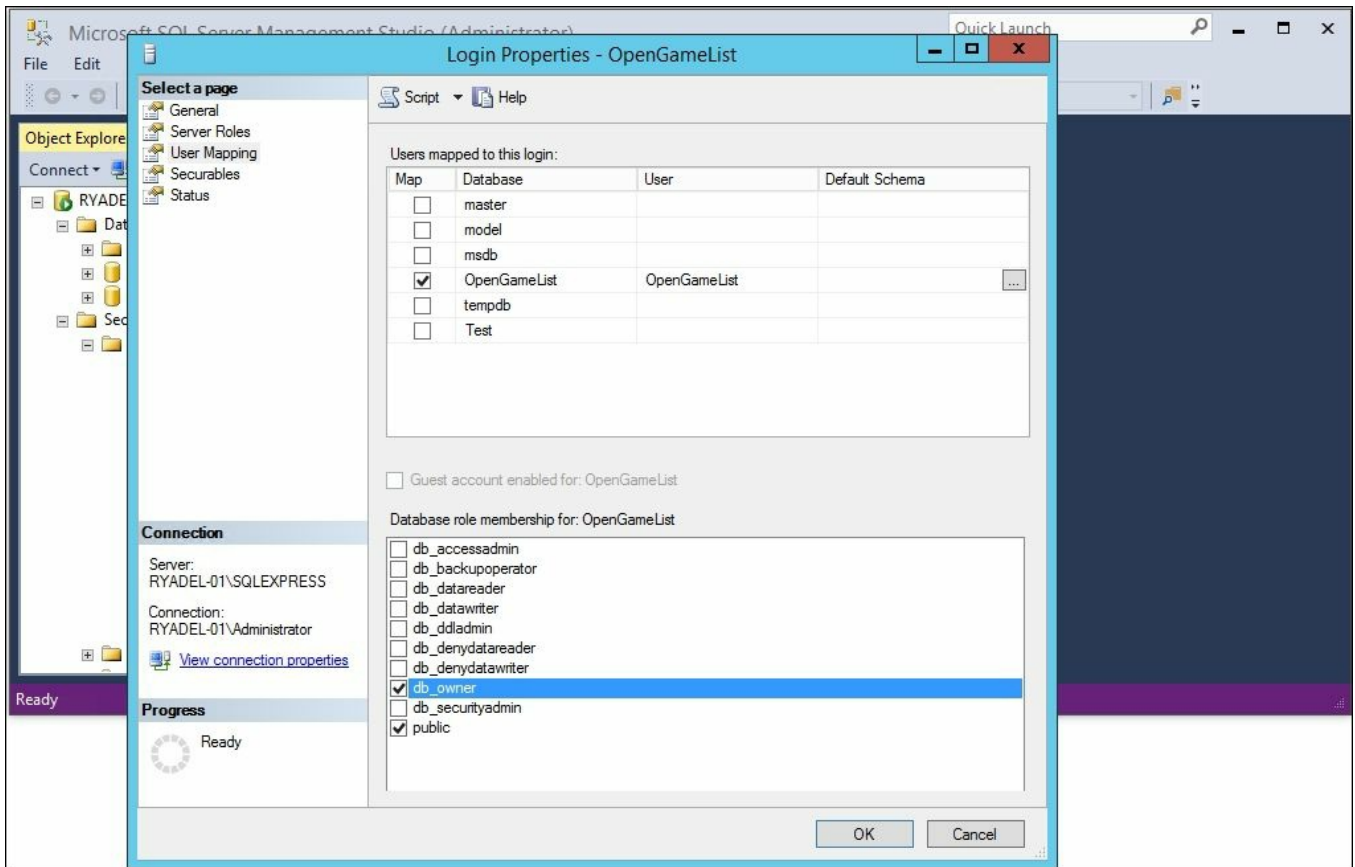
As soon as we click on the **OK** button, a new `OpenGameList` user will be added to the `OpenGameList` database with full administrative rights.

We can easily confirm that by going back to the root **Databases** folder and expanding it to **OpenGameList** | **Security** | **Users**:
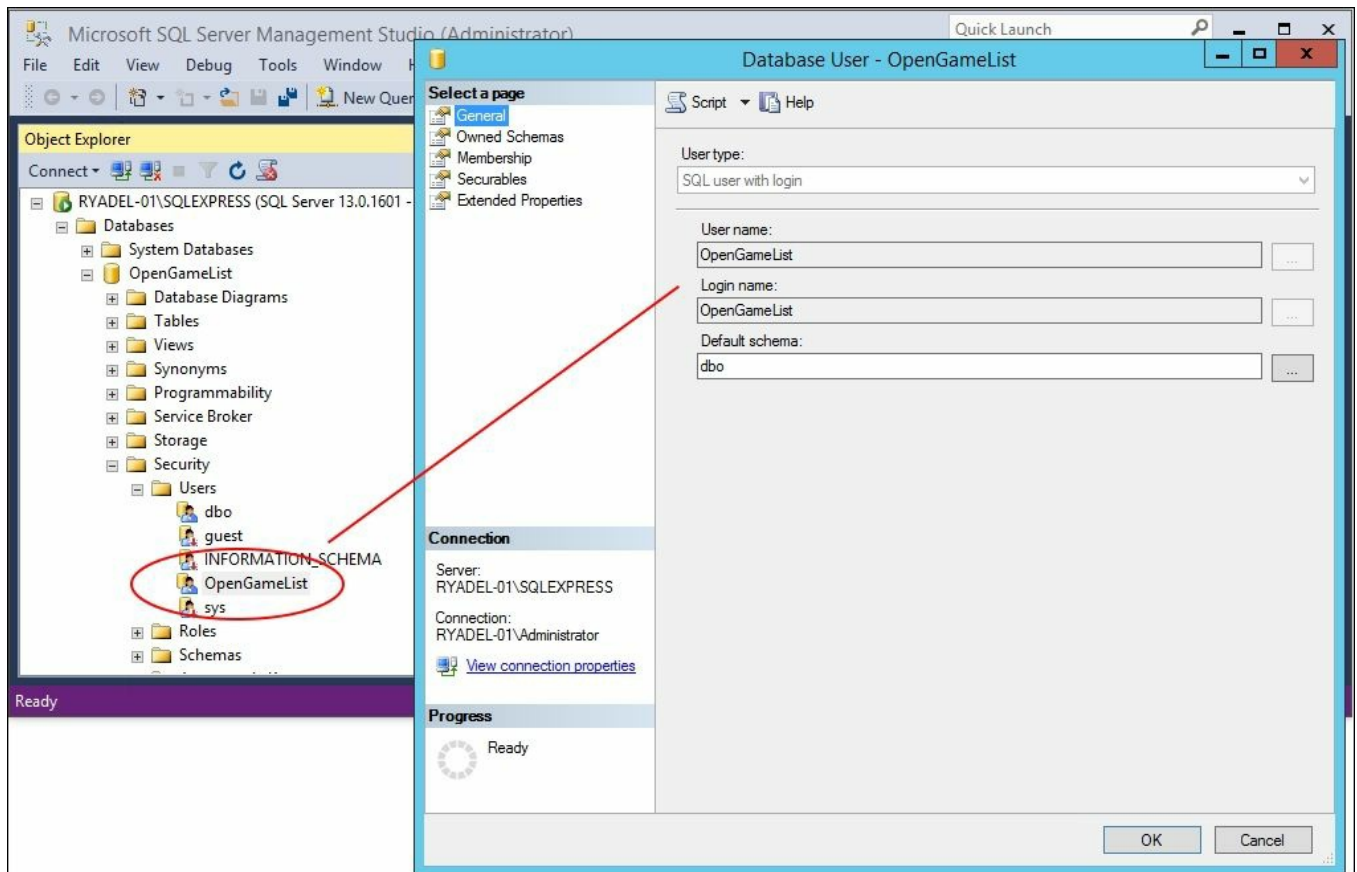
That's it! Now we'll be able to access our brand new `OpenGameList` database with a standard connection string using the credentials we just created.

Adding a SQL Server connection string

Now that the SQL Server database has been set up, we need to tell our application to use it instead of `localDb` while in production. We can easily do that by adopting the ASP.NET Core default pattern for configuring application behavior across multiple environments.

To implement it within our project, we need to perform the following steps:

1. Create an `appsettings.production.json` file to override the `localDb` connection string with the SQL Server one when the application runs in a production environment.
2. Configure the `publishOptions` within the `project.json` file to publish these kinds of file.
3. Check that our `Startup` class is properly configured to load the application settings files using a cascading logic that privileges the current running environment.
4. Update the `launchSettings.json` file to ensure that the production

environment will be set whenever we publish our project.

Creating an application settings file for production

Right-click to the project's root and select **Add | New Item**. Choose **ASP.NET Configuration File**, name it `appsettings.production.json`, and click on **OK**.

An environment-specific version of the `appsettings.json` file will be created accordingly.

Visual Studio should also nest it below the main configuration file, as can be seen in the following screenshot:
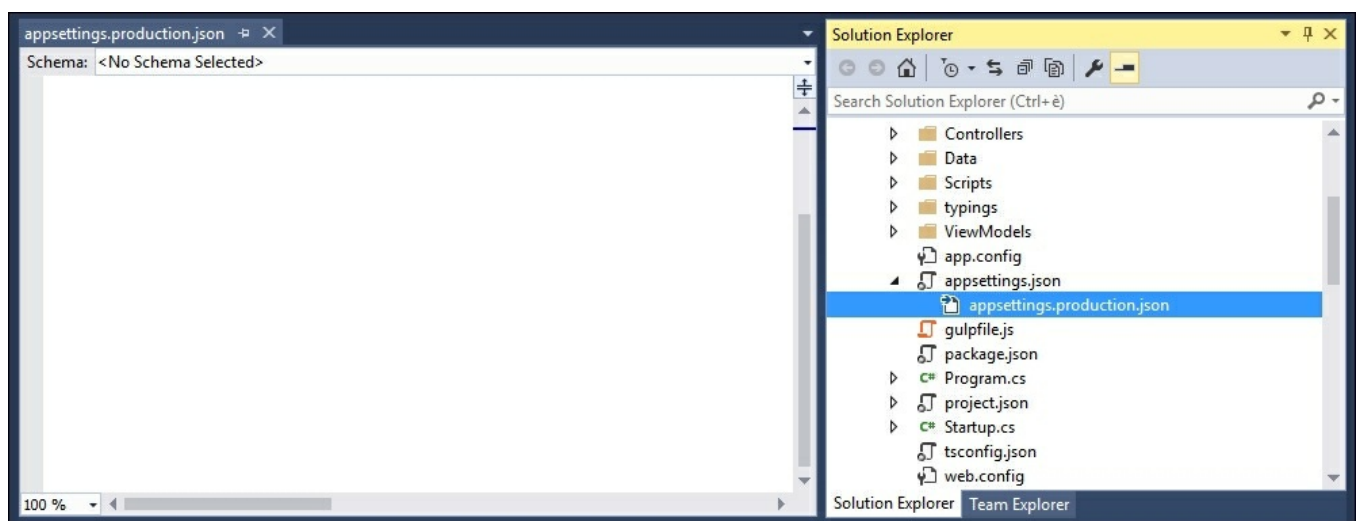
From now on, any key/value pair we'll include within the `appsettings.production.json` file will override the corresponding key/value in the main configuration file whenever the application will run within a production environment. It means that we can use it to redefine a number of production-specific values there, including (yet not limited to) the default connection string.

As a matter of fact, that's precisely what we need to do. Open the new `appsettings.production.json` file and replace the sample contents with the following code:

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString":
"Server=localhost\\SQLEXPRESS;Database=OpenGameList;User
Id=OpenGameList;Password=SamplePassword123;Integrated
Security=False;MultipleActiveResultSets=True"
```

```
      }
   }
}
```

Updating AppSettings

Another important thing we definitely need to override within the
`appsettings.production.json` file is the `Authority` that will be used to issue
(and check) our JWT Tokens. To do that, add the following lines right before the
`data` key we just added:

```
"Authentication": {
  "OpenIddict": {
    "Authority": "http://www.your-website-url.com/"
  }
},
```

Be sure to check the other settings before closing the file, changing them accordingly
to a production environment. For example, we might want to switch to a less
resource-intensive debugging behavior:

```
"Logging": {
  "IncludeScopes": false,
  "LogLevel": {
    "Default": "Warning",
    "System": "Error",
    "Microsoft": "Error"
  }
},
```

We might also want to increase our caching policy for local files:

```
"StaticFiles": {
  "Headers": {
    "Cache-Control": "max-age=3600",
    "Pragma": "cache",
    "Expires": null
  }
}
```

This will tell the browser client (and/or a proxy) to keep all the static files for one
hour.

Updating external providers

If we implemented one or more external providers (see Chapter 8, *Third-Party
Authentication and External Providers*), we'll probably need to replace the redirect
URIs we configured earlier and also add our public facing URL (`www.our-website-
url.com`) to the allowed JavaScript origins URL list. For a detailed guide about how

to do that, we can refer to [Chapter 8](#), *Third-Party Authentication and External Providers.*

Tip

Although most providers will allow to set multiple values for the allowed origin URIs, it's strongly advisable to add a whole new app, such as `OpenGameList_Production`, to use for production purposes. If you do that you will also have to override the public and private keys, but you won't compromise your development environment.

Configuring the publishOptions

We now need to update the root `project.json` file to include these files whenever we publish our application. Within that file, look for the `publishOptions` key and change the `include` array values to the following (updated lines are highlighted):

```
"publishOptions": {
  "include": [
    "wwwroot",
    "Views",
    "Areas/**/Views",
    "appsettings*.json",
    "web.config"
  ]
},
```

This small change ensures that, whenever we publish our application, each and every `appsettings` file will be included as well.

Checking the Startup class

The application settings file(s) loading logic can be customized within the `Startup` method of the `Startup` class.

The `Startup.cs` file bundled with the default ASP.NET Core project template that we chose back in [Chapter 1](#), *Getting Ready*, already features a cascading logic in place that perfectly suits our needs. All we need to do is to ensure that it's still there. We can easily check that by opening the `Startup.cs` file and having a look at the two `AddJsonFile` method calls within the constructor:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile(
"appsettings.json",
```

```
        optional: true,
        reloadOnChange: true)
    .AddJsonFile(
$"appsettings.{env.EnvironmentName}.json",
        optional: true)
    .AddEnvironmentVariables();
  Configuration = builder.Build();
}
```

We can see how the first highlighted line will load the default `appsettings.json` file, while the second will look for an (optional) environment-specific version containing the relevant value overrides. If everything is still in place, we don't need to change anything, since the existing behavior perfectly fits what we want.

Updating the launchSettings.json

Last but not least, we need to set up our app so that it will run in a production environment whenever we publish it.

To do that, open the `/properties/launchSettings.json` file and change the `ASPNETCORE_ENVIRONMENT` variable within our application's profile from `Development` to `Production` in the following way:

```
"OpenGameListWebApp": {
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "http://localhost:5000/api/values",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Production"
  }
}
```

This will ensure the proper loading of the `appsettings.production.json` file.

Publishing our native web application

Installing and/or configuring a production-ready web server, such as **Internet Information Services** (**IIS**) or Apache, can be a very complex task depending on a number of things that we can't address now, as they would easily bring us far beyond the scope of this book.

To keep things simple, we'll just assume that we already have access to an up and running, physical or virtualized Windows Server machine featuring a running IIS instance (we'll call it web server from now on) that we can configure to suit our needs. Ideally, we'll be able to do that via a dedicated management interface such as Remote Desktop, IIS Remote Configuration, Plesk, or any other remote administration mechanism made available by our web farm and/or service provider.

Note

Windows 2008 R2 (or newer) and IIS 7.5 (or above) will be required to host a .NET Core web application, as stated by the official Microsoft publishing and deployment documentation available at the following URL:

https://docs.asp.net/en/latest/publishing/iis.html

Last but not least, we'll also assume that our web server provides FTP-based access to the `/inetpub/` folder that we can use to publish our web projects.

Tip

If we're facing a different scenario, it could be advisable to skip this chapter entirely and follow the instructions given by our chosen web hosting provider instead.

Creating a Publish profile

The most convenient way to deploy a web-based project in Visual Studio is creating one or more **Publish** profiles. Each one of them is basically an XML configuration file with a `.pubxml` extension that contains a set of deployment-related information, most of which depends on the server/target we're deploying our application into, Azure, FTP, filesystem and more.

We can easily set up one or more **Publish** profile files using the **Publish** profile wizard. As soon as we have at least one ready, we'll be able to execute it with a single mouse click and have our application published.

To open the **Publish** profile wizard, right-click on the project's root node (`OpenGameListWebApp` in our case) and select the **Publish...** option from the contextual menu. A pop-up window should open showing the following welcome screen:
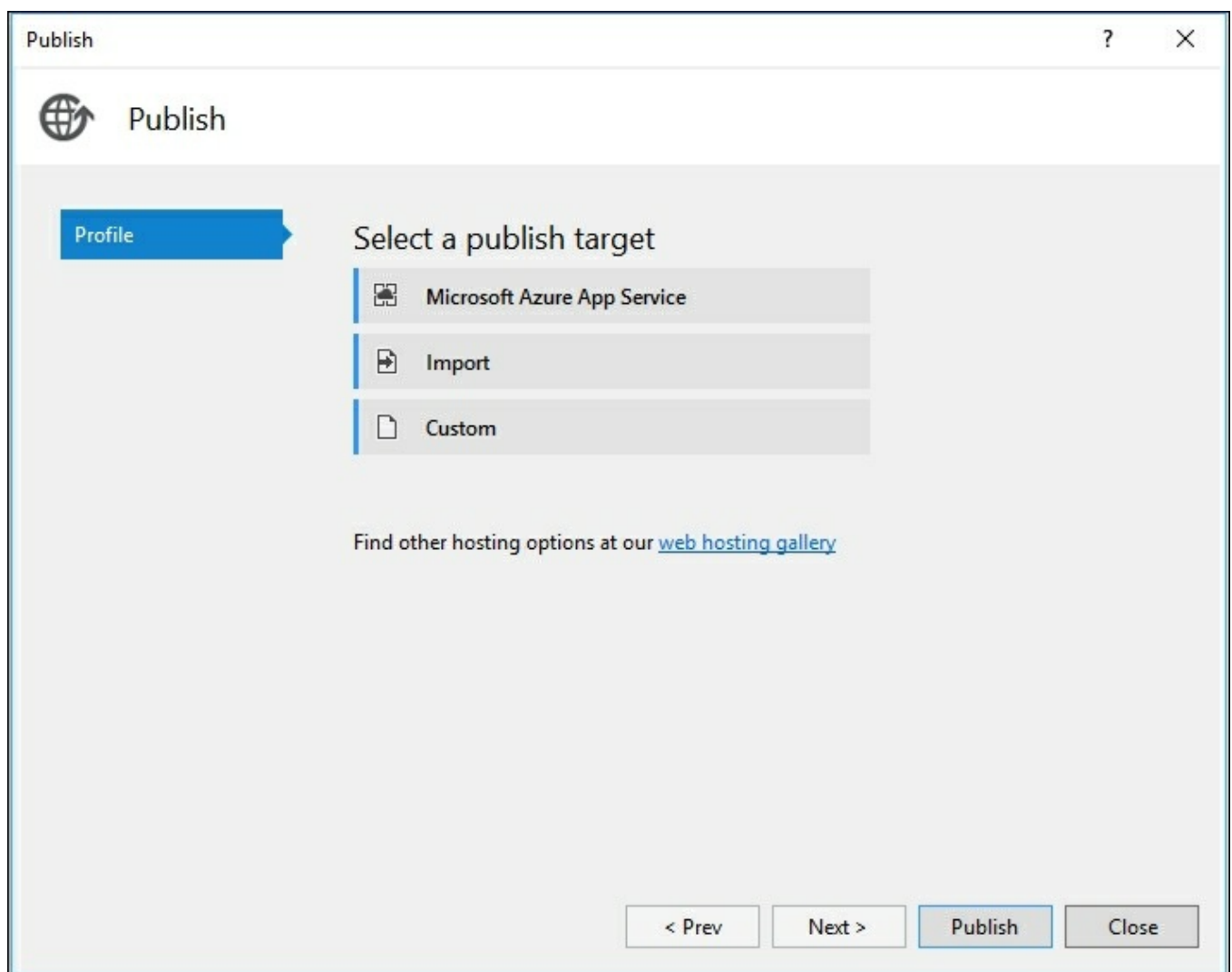
File System Publish profile

Among the various available profiles, the easiest to configure is the one that builds everything into a dedicated folder within our local File System. To create it, follow these steps:

1. Select the **Custom** publish target.
2. Give a suitable name to the profile, such as `Production-FileSystem`.
3. Select the **File System** publish method.
4. When prompted for a **Target Location**, specify the path of the folder that will contain the published application. Visual Studio will suggest a path located within the application's `/bin/Release/` subfolder. However, it's strongly advisable to change it and to choose an external directory instead.
5. Set **Configuration** to **Release**.
6. Set the **Target Framework** accordingly to what we used for our project (**.NET Framework 4.6.1** in our example).

We can then click on the **Publish** button to start the publishing process. Once done, we'll have to manually upload the generated contents to the web server.

FTP Publish profile

As a viable alternative we can also create a **Publish** profile that will automatically upload our web project to our web server using a properly configured FTP connection endpoint: we will then link the remote destination folder to a new website project using IIS.

Note

As we said earlier, we're doing all that assuming that we have a web server accessible through FTP, since it's one of the most common deployment scenarios. If that's not the case, we might as well skip this paragraph and configure a different **Publish** profile.

We already know how to launch the wizard. Right-click on the project's root node and select **Publish**. Wait for it to start, then do the following:

1. Select the **Custom** publish target.
2. Give a suitable name to the profile, such as `Production-FTP`.
3. Select the **FTP** publish method.
4. When prompted for a **Server**, specify the FTP server URL, such as `ftp.your-ftp-server.com`. In the **Site Path** option, insert the target folder from the FTP

server root, such as `/OpenGameList/`.

5. Set the **Passive Mode**, **Username**, and **Password** according to our FTP server settings.
6. Set **Configuration** to **Release**.
7. Set the **Target Framework** accordingly to what we used for our project (**.NET Framework 4.6.1** in our example).

Right after that, the wizard's **Connection** tab should eventually look not too different from the following screenshot:

Configuring IIS

We should now connect to our web server and set up our web application within IIS.

Note

As we said earlier, configuring a web application can be either a very easy or an insanely complex task depending on a number of things, such as caching, load balancing, CPU optimization, database load, and security issues.

Although the most common issues will be briefly handled within this chapter, it's advisable to follow a dedicated guide to properly handle each one of them.

Installing the ASP.NET Core module for IIS

We might think that IIS is the ideal platform to host ASP.NET Core applications, as it always has been since the first release of ASP.NET. As a matter of fact, it's not. ASP.NET Core web applications run via the highly optimized Kestrel server. Whenever we choose to host one of them with IIS, we basically need it to act as a reverse proxy for the underlying Kestrel server.

The good news is that we don't need to configure anything by ourselves, because the ASP.NET Core template we used back in Chapter 1, *Getting Ready*, provided us with a root `web.config` file containing all the settings, to do just that.

The relevant configuration lines should be contained within the `<system.webServer>` element, which should resemble the following XML code:

```
<system.webServer>
  <handlers>
    <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModule" resourceType="Unspecified"/>
  </handlers>
  <aspNetCore processPath="%LAUNCHER_PATH%"
arguments="%LAUNCHER_ARGS%" stdoutLogEnabled="false"
stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false"/>
</system.webServer>
```

We can see that we have a dedicated `aspNetCore` handler and some related configuration placeholders that will be transformed into actual values upon publication.

As we said earlier, we don't need to change anything, as the handler will do everything by itself, assuming that it is installed on the Web Server. Since ASP.NET Core is a rather new technology, this might as well not be the case, so we could need to download and install it.

At the time of writing, we need to obtain the .NET Core Windows Server Hosting bundle, which conveniently includes all the required packages to host a .NET Core application on a IIS powered server machine: the .NET Core Runtime, the .NET Core Library, the ASP.NET Core module, and also the required reverse proxy between IIS and the Kestrel server.

The bundle can be downloaded from the following URL:

https://go.microsoft.com/fwlink/?LinkId=817246

Note

For further references regarding ASP.NET Core IIS publishing settings, it's strongly advised to check out this official guide:

https://docs.asp.net/en/latest/publishing/iis.html#iis-configuration

A (mostly) complete list of all the available .NET Core related packages (SDK, IIS module, and more) is also available at the following URL:

https://www.microsoft.com/net/download

Adding the website

As soon as we install the .NET Core Windows Server Hosting bundle, we'll be able to configure our IIS instance to host our application.

Note

As we said earlier in this chapter, to host ASP.NET Core web applications, we're going to need IIS 7.5 or above.

From the **IIS Manager** interface, right-click on **Sites** and choose the **Add New Website** option. Name it `OpenGameList`. By looking at the read-only textbox to the immediate right, we can see that a new **Application Pool** will also be created with that same name. Take a mental note of it, as we'll need to configure it soon enough.

Set the physical path of the **Content Directory** to the folder we targeted for FTP publishing.

In our previous example, it should be something like `C:\inetpub\OpenGameList\`, assuming that the FTP root for the web admin user points to `C:\inetpub\`.

Be sure to target the application's root folder, not the `\wwwroot\` one.

Tip

Needless to say, we need to grant read/write permissions for that folder to the `IUSR` and/or `IIS_IUSRS` accounts, or any other identity our `Application Pool` is using.

As for the bindings, either choose a specific IP address or leave the **All Unassigned** option and choose a **Host name** that is already configured to redirect to our web server via DNS:
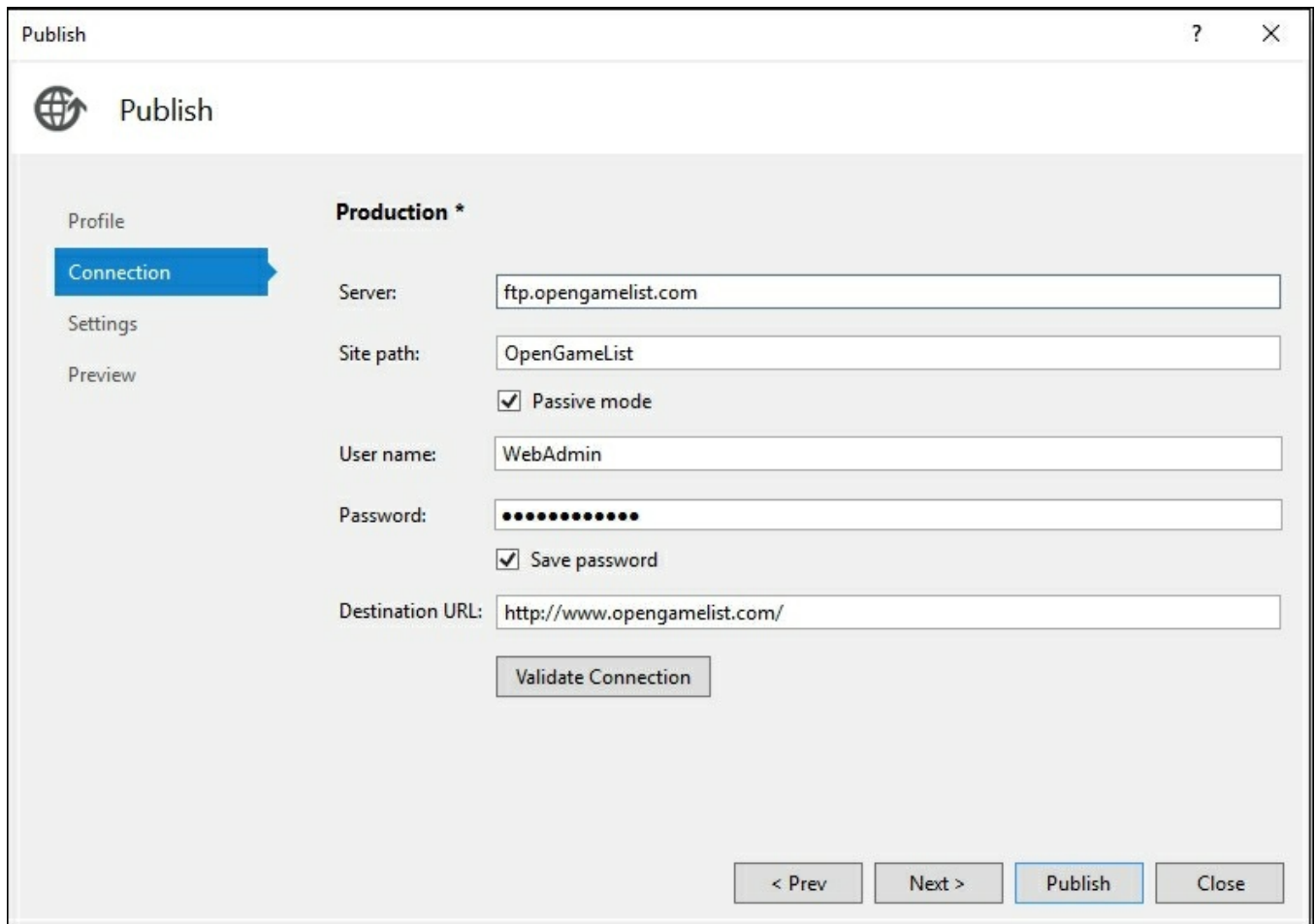
In our example, we already set up http://www.opengamelist.com , so we'll just use that.
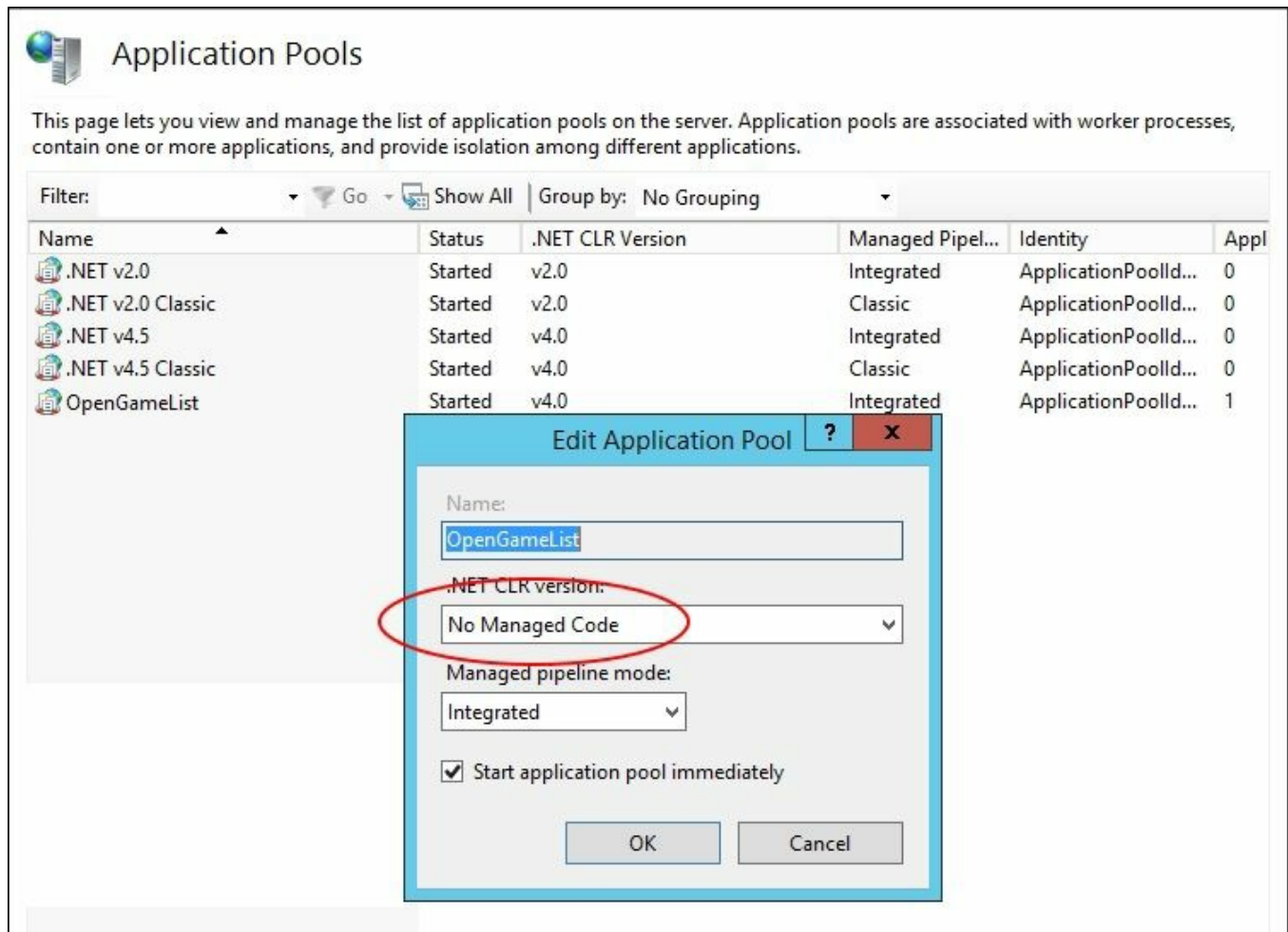
Before clicking on the **OK** button, ensure that the **Start Website immediately** option is checked, so the website will be immediately available.

Note

We're assuming that the server comes with the .NET Framework installed, as it's a default package with all the latest Windows Server versions. In case it doesn't, we can manually install it either via Server Manager, Web Platform Installer, or Windows Update.

Configuring the Application Pool

We can now switch to the **Application Pools** node. Select the **OpenGameList** one, the same that we created a short while ago, and set the **.NET CLR version** to **No Managed Code**:

This might seem rather counterintuitive, as it looks like we're ruling out ASP.NET. As a matter of fact, that's the exact opposite: since we're publishing an ASP.NET Core application, we need to ensure that the soon-to-be-outdated .NET 4 CLR won't get in the way. Since the former is still (at the time of writing) in a pre-release state, it isn't available yet within the GUI, leaving us with the only option to remove any reference here. We already configured the .NET Core module for IIS to do the required job anyway.

Note

This is one of the many things that will surely change in the future. There is a good chance that, by the time you're reading this book, the new CLR will be integrated
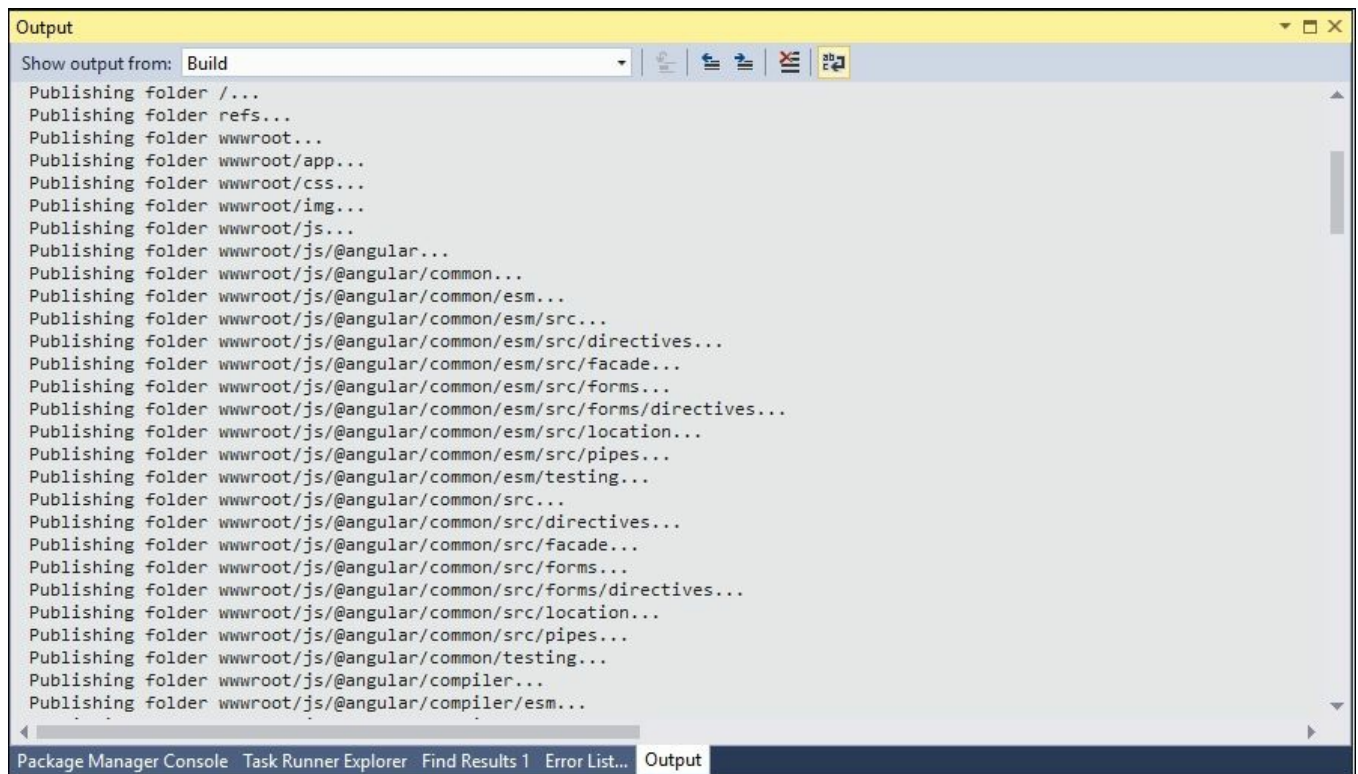
within the **Application Pool** GUI settings.

Firing up the engine

It's time to publish our native web application. Before doing that, ensure that the **Task Runner** `default` task is running, as we want to upload the latest version of our client files.

Right-click on the project's root node, then left-click on **Publish**. Select the **Production-FTP** profile and hit the **Publish** button to start the build and upload process.

The whole publishing process flow can be checked in real time within the Visual Studio **Output** window. As soon as the FTP connection will be attempted, we'll be asked for username and password, unless we gave our consent to store our login credentials within the `Production-FTP.pubxml` file.

The publishing task will require a lot more time than it used to be for ASP.NET 4 applications, because the whole set of ASP.NET Core libraries will be published in the destination folder, together with all the required Angular 2 modules and dependencies.

Tip

There's nothing to worry about, as this is another thing that will surely change once the ASP.NET Core framework reaches its final stage.

Once done, our default web browser will be automatically launched against the URL we specified within the **Publish** profile settings.

If everything has been set up properly, our native web application will show itself in all its splendor:
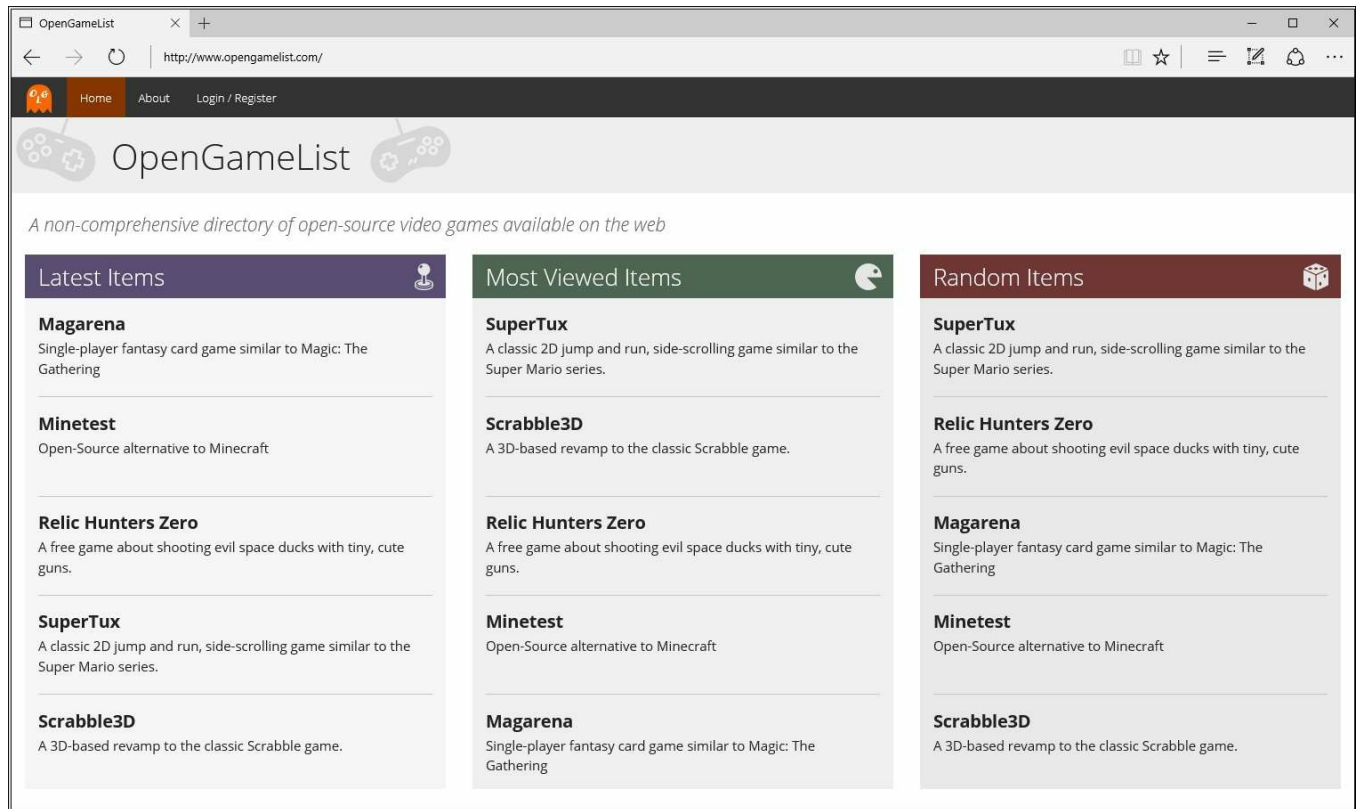
Troubleshooting

...Or maybe not. The deployment task isn't always easy, as there could be a number of issues (mostly depending on the server machine state) that can prevent it from going well. This statement is particularly true for ASP.NET Core application IIS-based deployments, as the reverse proxy mechanism undeniably adds an additional level of complexity.

These are the three best things we can do to diagnose the most common problems:

1. Read the browser's output messages.
2. Examine the Event Viewer's application log.
3. Enable the ASP.NET Core module `stdout` logging feature.

The first one is rather obvious: who doesn't look at the browser output? However, for ASP.NET Core applications, it's far less effective than it used to be since most errors are still unhandled and won't appear there.

The Event Viewer is often underestimated, yet it's very important for debugging ASP.NET Core application for the same reason as before: we will find most of the relevant stuff there.

The ASP.NET Core Module Log is a new feature brought by the new CLR. However, when it comes to troubleshooting issues, it happens to be the real deal. Activating it is just as easy as opening the root `web.config` file and change the `stdoutLogEnabled` attribute from `false` to `true`.

We also need to manually create a `/logs/` folder inside the root application folder on the web server, otherwise the logs won't be generated.

Tip

The log folder, location, and filename prefix can be configured by changing the `stdoutLogFile` attribute value. Remember to manually create the chosen folder whenever you change it and also to grant read/write permissions to the identity used by the **Application Pool**.

The Kestrel test

A quick and effective way to check if the application is working properly is to entirely skip IIS and run it directly on Kestrel. Doing this is just as easy as opening the application folder on the Web Server, locating the `OpenGameList.exe` file, and executing it with (or without) administrative rights.

Doing this will open a command prompt where we'll be able to see the whole application bootstrap process. Once it completes, we should be able to test the application by opening a web browser and pointing it to `http://localhost:5000/`, `5000` being the default TCP listening port for Kestrel as defined in the `/settings/launchSettings.json` file:

If the application completes its boot phase and starts running, the issue is most likely related to the IIS configuration and/or the ASP.NET Core module; otherwise, there's a good chance that our problem lies within the application code itself.

If that's the case, checking the Event Viewer and the aforementioned `stdout` logs will be our best weapons to identify and overcome the issue.

Suggested topics

LAN, SQL Server 2016, SQL Server Management Studio, Windows Server, IIS, Apache, FTP server, **Publish** profile, ASP.NET Core module for IIS, ASP.NET 5, .NET CLR v4, Kestrel, `stdout` log.

Summary

Eventually, our journey through ASP.NET Core MVC/Web API and Angular 2 has come to an end. Our last effort was getting our native web application ready for being published into a production environment, where it can be checked by the product owner as the potentially shippable product it now definitely is.

The first thing we did was changing the underlying database from `localDb` to a real SQL Server instance. For the sake of simplicity, we chose to install SQL Server 2016 Express, which is freely available for download from the Microsoft Download Center. We briefly installed it, together with the SQL Server Management Studio tools, then we used the latter to properly configure the former: creating the database, adding the login credentials, and doing what it takes to make our application able to connect using a standard connection string. We also took advantage of the ASP.NET Core default pattern to handle multiple environments by defining an application settings file for production, which we used to override the `localDb` connection string with the SQL Server one.

The next step was creating a **Publish** profile for our Visual Studio project. We evaluated two alternatives, File System and FTP, each one of them being viable or not depending on our own deployment scenario.

Eventually, we switched to the **Web Server**, where we found out that configuring IIS was not as easy as it used to be for ASP.NET 4 and below because the new CLR isn't fully integrated within the GUI yet. We had to install the ASP.NET Core module for IIS which does all the required jobs, making IIS act like a reverse proxy for the underlying Kestrel server. Right after that, we were able to create our **website** entry together with its related **Application Pool**.

Once we did all that, we were able to actually publish our native web application and watch the result on the web browser. In the event that something didn't go as expected, we took some time to analyze the most common troubleshooting issues and give some useful advice to overcome them.

Part 3. Module 3

ASP.NET Core 1.0 High Performance

Create fast, scalable, and high performance applications

Chapter 1. Why Performance Is a Feature

This is an exciting time to be a **C#** developer. Microsoft is in the middle of one of the biggest changes in its history, and it is embracing open source software. The ASP.NET and .NET frameworks are being rebuilt from the ground up to be componentized, cross-platform, and open source.

ASP.NET Core 1.0 and **.NET Core 1.0** (previously called **ASP.NET 5** and **.NET Core 5**) embrace many ideas from popular open source projects, such as **Go**'s ability to produce a statically-linked, standalone binary. You can now compile a single native executable that is free of any external dependencies and run it on a system without **.NET** installed.

The ASP.NET **Model View Controller** (**MVC**) web application framework, which is now part of ASP.NET Core 1.0, borrows heavily from **Ruby on Rails** and Microsoft is keen in promoting tools, such as **Node.js**, **Grunt**, **gulp**, and **Yeoman**. There is also **TypeScript**, which is a statically-typed version of JavaScript that was developed at Microsoft.

By reading this book, you will learn how to write high-performance software using these new .NET Core technologies. You'll be able to make your web applications responsive to input and scalable to demand.

We'll focus on the latest Core versions of .NET. Yet, many of these techniques also apply to previous versions, and they will be useful for web application development in general (in any language or framework).

Understanding how all of these new frameworks and libraries fit together can be a bit confusing. We'll present the various available options while still using the newest technology, guiding you down the path to high-speed success, and avoiding performance pitfalls.

After finishing this book, you will understand what problems can occur when web applications are deployed at scale (to distributed infrastructure) and know how to avoid or mitigate these issues. You will gain the experience of how to write high-performance applications without learning about issues the hard way.

In this chapter, we will cover the following topics.

- Performance as a feature

- The common classes of performance issues
- Basic hardware knowledge
- Microsoft tools and alternatives
- New .NET naming and compatibility

Performance as a feature

You may have previously heard about the practice of treating performance as a first-class feature. Traditionally, performance (along with things such as security, availability and uptime) was only considered a **Non-Functional Requirement (NFR)** and usually had some arbitrary made-up metrics that needed to be fulfilled. You may have heard the term "performant" before. This is the quality of performing well and, often, is captured in requirements without quantification, providing very little value. It is better to avoid this sort of corporate jargon when corresponding with clients or users.

Using the outdated waterfall method of development, these NFRs were inevitably left until the end, and dropped from an over-budget and late project in order to get the functional requirements completed. This resulted in a substandard product that was unreliable, slow, and often insecure (as reliability and security are also often neglected NFRs). Think about how many times you're frustrated at software that lags behind in responding to your input. Perhaps, you used a ticket-vending machine or a self-service checkout that is unresponsive to the point of being unusable.

There is a better way. By treating performance as a feature and considering it at every stage of your **agile** development process, you can get users and customers to love your product. When software responds quicker than a user can perceive, it is a delight to use, and this doesn't slow them down. When there is noticeable lag, then users need to adjust their behavior to wait for the machine instead of working at their own pace.

Computers have incredible amounts of processing power today, and they now possess many more resources than they did even just a few years ago. So, why do we still have software that is noticeably slow at responding, when computers are so fast and can calculate much quicker than people can? The answer to this is poorly written software that does not consider performance. Why does this happen? The reason is that often the signs of poor performance are not visible in development, and they only appear when deployed. However, if you know what to look for, then you can avoid these problems before releasing your software to the production environment.

This book will show you how to write software that is a joy to use and never keeps the user waiting or uninformed. You will learn how to make products that users will love instead of products that frustrate them all the time.

Common classes of performance problems

Let's take a look at some common areas of performance problems and see whether they matter or not. We will also learn why we often miss these issues during development.

Language considerations

People often focus on the speed of the programming language that is used. However, this often misses the point. This is a very simplistic view that glosses over the nuances of technology choices. It is easy to write slow software in any language.

With the huge amounts of processing speed that is available today, relatively "slow" interpreted languages can often be fast enough, and the increase in development speed is worth it. It is important to understand the arguments and the trade-offs involved even if by reading this book you have already decided to use C# and .NET.

The way to write the fastest software is to get down to the metal and write in assembler (or even machine code). This is extremely time consuming, requires expert knowledge, and ties you to a particular processor architecture and instruction set; therefore, we rarely do this these days. If this happens, then it's only done for very niche applications (such as virtual reality games, scientific data crunching, and sometimes embedded devices) and usually only for a tiny part of the software.

The next level of abstraction up is writing in a language, such as Go, C, or C++, and compiling the code to run on the machine. This is still popular for games and other performance-sensitive applications, but you often have to manage your own memory (which can cause memory leaks or security issues, such as buffer overflows).

A level above is software that compiles to an intermediate language or byte code and runs on a virtual machine. Examples of this are **Java**, **Scala**, **Clojure**, and, of course, C#. Memory management is normally taken care of, and there is usually a **Garbage Collector** (**GC**) to tidy up unused references (Go also has a GC). These applications can run on multiple platforms, and they are safer. However, you can still get near to native performance in terms of execution speed.

Above these are interpreted languages, such as **Ruby**, **Python**, and JavaScript. These languages are not usually compiled, and they are run line-by-line by an interpreter. They usually run slower than a compiled language, but this is often not a problem. A more serious concern is catching bugs when using dynamic typing. You won't be able to see an error until you encounter it, whereas many errors can be caught at compile time when using statically-typed languages.

It is best to avoid generic advice. You may hear an argument against using Ruby on Rails, citing the example of Twitter having to migrate to Java for performance reasons. This may well not be a problem for your application, and indeed having the popularity of Twitter would be a nice problem to have. A bigger concern when running Rails may be the large memory footprint, making it expensive to run on cloud

instances.

This section is only to give you a taste, and the main lesson is that normally, language doesn't matter. It is not usually the language that makes a program slow, it's poor design choices. C# offers a nice balance between speed and flexibility that makes it suitable for a wide range of applications, especially server-side web applications.

Types of performance problems

There are many types of performance problems, and most of them are independent of the programming language that is used. A lot of these result from how the code runs on the computer, and we will cover the impact of this later on in the chapter.

We will briefly introduce common performance problems here and will cover them in more detail in later chapters of this book. Issues that you may encounter will usually fall into a few simple categories, including the following:

- Latency:
    - Memory latency
    - Network latency
    - Disk and I/O latency
    - Chattiness / handshakes
- Bandwidth:
    - Excessive payloads
    - Unoptimized data
    - Compression
- Computation:
    - Working on too much data
    - Calculating unnecessary results
    - Brute forcing algorithms
- Doing work in the wrong place:
    - Synchronous operations that could be done offline
    - Caching and coping with stale data

When writing software for a platform, you are usually constrained by two resources. These are the computation processing speed and accessing remote (to the processor) resources.

Processing speed is rarely a limiting factor these days, and this could be traded for other resources, for example, compressing some data to reduce the network transfer time.

Accessing remote resources, such as main memory, disk, and the network will have various time costs. It is important to understand that speed is not a single value, and it has multiple parameters. The most important of these are bandwidth and, crucially, **latency**.

Latency is the lag in time before the operation starts, whereas bandwidth is the rate at which data is transferred once the operation starts. Posting a hard drive has a very

high bandwidth, but it also has very high latency. This would make it very slow to send lots of text files back and forth, but perhaps, this is a good choice to send a large batch of 3D videos (depending on the Weissman score). A mobile phone data connection may be better for the text files.

Although this is a contrived example, the same concerns are applicable to every layer of the computing stack often with similar orders of magnitude in time difference. The problem is that the differences are too quick to perceive, and we need to use tools and science to see them.

The secret to solving performance problems is in gaining a deeper understanding of the technology and knowing what happens at the lower levels. You should appreciate what the framework is doing with your instructions at the network level. It's also important to have a basic grasp of how these commands run on the underlying hardware, and how they are affected by the infrastructure that they are deployed to.

When performance matters

Performance is not always important in every situation. Learning when performance does and doesn't matter is an important skill to acquire. A general rule of thumb is that if the user has to wait for something to happen, then it should perform well. If this is something that can be performed asynchronously, then the constraints are not as strict, unless an operation is so slow that it takes longer than the time window for it; for example, an overnight batch job on an old financial services mainframe.

A good example from a web application standpoint is rendering user view versus sending e-mail. It is a common, yet naïve, practice to accept a form submission and send an e-mail (or worse, many e-mails) before returning the result. Yet, unlike a database update, an e-mail is not something that happens almost instantly. There are many stages over which we have no control that will delay an e-mail in reaching a user. Therefore, there is no need to send an e-mail before returning the result of the form. You can do this offline and asynchronously after the result of the form submission is returned.

The important thing to remember here is that it is the perception of performance that matters and not absolute performance. It can be better to not do some work (or at least defer it) rather than speed it up.

This may be counterintuitive, especially considering how individual computer operations can be too quick to perceive. However, the multiplying factor is scale. One operation may be relatively quick, but millions of them may accumulate to a visible delay. Optimizing these will have a corresponding effect due to the magnification. Improving code that runs in a tight loop or for every user is better than

fixing a routine that runs only once a day.

Slower is sometimes better

In some situations, processes are designed to be slow, and this is essential to their operation and security. A good example of this, which may be hit in profiling, is **password hashing** or **key stretching**. A secure password hashing function should be slow so that the password, which (despite being bad practice) may have been reused on other services, is not easily recovered.

We should not use generic hashing functions, such as **MD5**, **SHA1**, and **SHA256**, to hash passwords because they are too quick. Some better algorithms that are designed for this task are **PBKDF2** and **bcrypt**, or even **Argon2** for new projects. Always remember to use a unique salt per password too. We won't go into any more details here, but you can clearly see that speeding up password hashing would be bad, and it's important to identify where to apply optimizations.

Why issues are missed

One of the main reasons that performance issues are not noticed in development is that some problems are not perceivable on a development system. Issues may not occur until latency increases. This may be because a large amount of data was loaded into the system and retrieving a specific record takes longer. This may also be because each piece of the system is deployed to a separate server, increasing the network latency. When the number of users accessing a resource increases, then the latency will also increase.

For example, we can quickly insert a row into an empty database or retrieve a record from a small table, especially when the database is running on the same physical machine as the web server. When a web server is on one virtual machine and the big database server is on another, then the time taken for this operation can increase dramatically.

This will not be a problem for one single database operation, which appears just as quick to a user in both cases. However, if the software is poorly written and performs hundreds or even thousands of database operations per request, then this quickly becomes slow.

Scale this up to all the users that a web server deals with (and all of the web servers) and this can be a real problem. A developer may not notice that this problem exists if they're not looking for it, as the software performs well on their workstation. Tools can help in identifying these problems before the software is released.

Measuring

The most important takeaway from this book is the importance of measuring. You need to measure problems or you can't fix them. You won't even know when you have fixed them. Measurement is the key to fixing performance issues before they become noticeable. Slow operations can be identified early on, and then they can be fixed.

However, not all operations need optimizing. It's important to keep a sense of perspective, but you should understand where the chokepoints are and how they will behave when magnified by scale. We'll cover measuring and profiling in the next chapter.

The benefits of planning ahead

By considering performance from the very beginning, it is cheaper and quicker to fix issues. This is true for most problems in software development. The earlier you catch a bug, the better. The worst time to find a bug is once it is deployed and then being reported by your users.

Performance bugs are a little different when compared to functional bugs because often, they only reveal themselves at scale, and you won't notice them before a live deployment unless you go looking for them. You can write integration and load tests to check performance, which we will cover later in this book.

Understanding hardware

Remember that there is a computer in computer science. It is important to understand what your code runs on and the effects that this has, this isn't magic.

Storage access speeds

Computers are so fast that it can be difficult to understand which operation is a quick operation and which is a slow one. Everything appears instant. In fact, anything that happens in less than a few hundred milliseconds is imperceptible to humans. However, certain things are much faster than others are, and you only get performance issues at scale when millions of operations are performed in parallel.

There are various different resources that can be accessed by an application, and a selection of these are listed, as follows:

- CPU caches and registers:
    - L1 cache
    - L2 cache
    - L3 cache
- RAM
- Permanent storage:
    - Local **Solid State Drive (SSD)**
    - Local **Hard Disk Drive (HDD)**
- Network resources:
    - **Local Area Network (LAN)**
    - Regional networking
    - Global internetworking

Virtual Machines (**VM**s) and cloud infrastructure services could add more complications. The local disk that is mounted on a machine may in fact be a shared network disk and respond much slower than a real physical disk that is attached to the same machine. You may also have to contend with other users for resources.

In order to appreciate the differences in speed between the various forms of storage, consider the following graph. This shows the time taken to retrieve a small amount of data from a selection of storage mediums:

Approximate access times for various forms of storage
Logarithmic scale (lower is better)
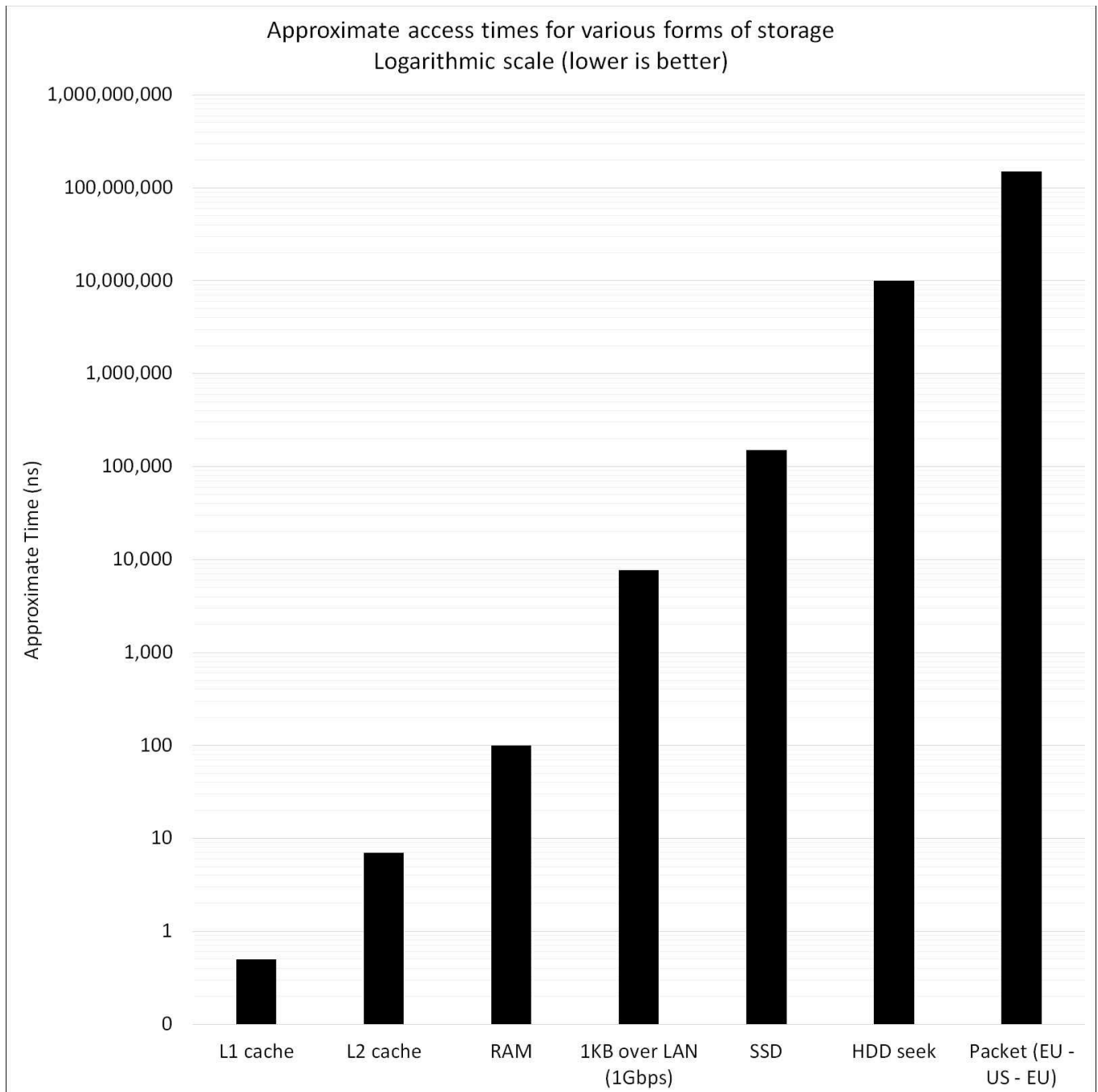
This graph has a logarithmic scale, which means that the differences are very large. The top of the graph represents one second or one billion nanoseconds. Sending a packet across the Atlantic Ocean and back takes roughly 150 milliseconds (ms) or 150 million nanoseconds (ns), and this is mainly limited by the speed of light. This is still far quicker than you can think about, and it will appear instantaneous. Indeed, it can often take longer to push a pixel to a screen than to get a packet to another continent.

The next largest bar is the time that it takes a physical HDD to move the read head into position to start reading data (10 ms). Mechanical devices are slow.

The next bar down is how long it takes to randomly read a small block of data from a local SSD, which is about 150 microseconds. These are based on Flash memory technology, and they are usually connected in the same way as a HDD.

The next value is the time taken to send a small datagram of 1 KB (1 kilobyte or 8 kilobits) over a gigabit LAN, which is just under 10 microseconds. This is typically how servers are connected in a data center. Note how the network itself is pretty quick. The thing that really matters is what you are connecting to at the other end. A network lookup to a value in memory on another machine can be much quicker than accessing a local drive (as this is a log graph, you can't just stack the bars).

This brings us on to main memory or RAM. This is fast (about 100 ns for a lookup), and this is where most of your program will run. However, this is not directly connected to the CPU, and it is slower than the on die caches. RAM can be large, often large enough to hold all of your working dataset. However, it is not as big as disks can be, and it is not permanent. It disappears when the power is lost.

The CPU itself will contain small caches for data that is currently being worked on, which can respond in less than 10 ns. Modern CPUs may have up to three or even four caches of increasing size and latency. The fastest (less than 1 ns to respond) is the Level 1 (L1) cache, but this is also usually the smallest. If you can fit your working data into these few MB or KB in caches, then you can process it very quickly.

Scaling approach changes

For many years, the speed and processing capacity of computers increased at an exponential rate. This was known as Moore's Law, named after Gordon Moore of Intel. Sadly, this era is no Moore (sorry). Single-core processor speeds have flattened out, and these days increases in processing ability come from scaling out to multiple cores, multiple CPUs, and multiple machines (both virtual and physical). Multithreaded programming is no longer exotic, it is essential. Otherwise, you cannot hope to go beyond the capacity of a single core. Modern CPUs typically have at least four cores (even for mobiles). Add in a technology such as **hyper-threading**, and you have at least eight logical CPUs to play with. Naïve programming will not be able to fully utilize these.

Traditionally, performance (and redundancy) was provided by improving the hardware. Everything ran on a single server or mainframe, and the solution was to use faster hardware and duplicate all components for reliability. This is known as vertical scaling, and it has reached the end of its life. It is very expensive to scale this way and impossible beyond a certain size. The future is in distributed-horizontal scaling using commodity hardware and cloud computing resources. This requires that we write software in a different manner than we did previously. Traditional software can't take advantage of this scaling like it can easily use the extra capabilities and speed of an upgraded computer processor.

There are many trade-offs that have to be made when considering performance, and it can sometimes feel like more of a black art than a science. However, taking a scientific approach and measuring results is essential. You will often have to balance memory usage against processing power, bandwidth against storage, and latency against throughput.

An example is deciding whether you should compress data on the server (including what algorithms and settings to use) or send it raw over the wire. This will depend on many factors, including the capacity of the network and the devices at both ends.

Tools and costs

Licensing of Microsoft products has historically been a minefield of complexity. You can even sit for an official exam on it and get a qualification. Microsoft's recent move toward open source practices is very encouraging, as the biggest benefit of open source is not the free monetary cost but that you don't have to think about the licensing costs. You can also fix issues, and with a permissive license (such as **MIT**), you don't have to worry about much. The time costs and cognitive load of working out licensing implications now and in the future can dwarf the financial sums involved (especially for a small company or startup).

Tools

Despite the new .NET framework being open source, many of the tools are not. Some editions of Visual Studio and SQL Server can be very expensive. With the new licensing practice of subscriptions, you will lose access if you stop paying, and you are required to sign in to develop. Previously, you could keep using existing versions licensed from a **Microsoft Developer Network** (**MSDN**) or BizSpark subscription after it expired and you didn't need to sign in.

With this in mind, we will try to stick to the free (community) editions of Visual Studio and the Express version of SQL Server unless there is a feature that is essential to the lesson, which we will highlight when it occurs. We will also use as many free and open source libraries, frameworks, and tools as possible.

There are many alternative options for lots of the tools and software that augments the ASP.NET ecosystem, and you don't just need to use the default Microsoft products. This is known as the ALT.NET (alternative .NET) movement, which embraces practices from the rest of the open source world.

Looking at some alternative tools

For version control, git is a very popular alternative to **Team Foundation Server** (**TFS**). This is integrated into many tools (including Visual Studio) and services, such as GitHub or GitLab. Mercurial (hg) is also an option. However, git has gained the most developer mindshare. Visual Studio Online offers both git and TFS integration.

PostgreSQL is a fantastic open source relational database, and it works with many **Object Relational Mappers** (**O/RMs**), including **Entity Framework** (**EF**) and NHibernate. Dapper is a great, and high-performance, alternative to EF and other bloated O/RMs. There are plenty of NoSQL options that are available too; for example, Redis and MongoDB.

Other code editors and **Integrated Development Environments** (**IDE**s) are available, such as Visual Studio Code by Microsoft, which also works on Apple Mac OS X. ASP.NET Core 1.0 (previously ASP.NET 5) runs on Linux (on Mono and CoreCLR). Therefore, you don't need Windows (although Nano Server may be worth investigating).

RabbitMQ is a brilliant open source message queuing server that is written in Erlang (which WhatsApp also uses). This is far better than **Microsoft Message Queuing** (**MSMQ**), which comes with Windows. Hosted services are readily available, for example, CloudAMQP.

The author has been a long time Mac user (since the PowerPC days), and he has run Linux servers since well before this. It's positive to see OS X become popular and to observe the rise of Linux on Android smartphones and cheap computers, such as the Raspberry Pi. You can run Windows 10 on a Raspberry Pi 2 and 3, but this is not a full operating system and only meant to run **Internet of Things** (**IoT**) devices. Having used Windows professionally for a long time, developing and deploying with Mac and Linux, and seeing what performance effects this brings is an interesting opportunity.

Although not open source (or always free), it is worth mentioning JetBrains products. TeamCity is a very good build and **Continuous Integration** (**CI**) server that has a free tier. ReSharper is an awesome plugin for Visual Studio, which will make you a better coder. They're also working on a C# IDE called Project Rider that promises to be good.

There is a product called Octopus Deploy, which is extremely useful for the deployment of .NET applications, and it has a free tier. Regarding cloud services, **Amazon Web Services** (**AWS**) is an obvious alternative to Azure, even if the AWS

Windows support leaves something to be desired. There are many other hosts available, and dedicated servers can often be cheaper for a steady load if you don't need the dynamic scaling of the cloud.

Much of this is beyond the scope of this book, but you would be wise to investigate some of these tools. The point is that there is always a choice about how to build a system from the huge range of components available, especially with the new version of ASP.NET.

The new .NET

The new ASP.NET and the .NET Framework that it relies upon were rewritten to be open source and cross-platform. This work was called ASP.NET 5 while in development, but this has since been renamed to ASP.NET Core 1.0 to reflect that it's a new product with a stripped down set of features. Similarly, .NET Core 5 is now .NET Core 1.0, and Entity Framework 7 is now Entity Framework Core 1.0.

The web application framework that was called ASP.NET MVC has been merged into ASP.NET Core, although it's a package that can be added like any other dependency. The latest version of MVC is 6 and, along with **Web API 2**, this has been combined into a single product, called ASP.NET Core. MVC and Web API aren't normally referred to directly any more as they are simply NuGet packages that are used by ASP.NET Core. Not all features are available in the new Core frameworks yet, and the focus is on server-side web applications to start with.

All these different names can be perplexing, but naming things is hard. A variation of Phil Karlton's famous quote goes like this:

> *"There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors."*

We've looked at naming here, and we'll get to caching later on in this book.

It can be a little confusing understanding how all of these versions fit together. This is best explained with a diagram like the following, which shows how the layers interact:
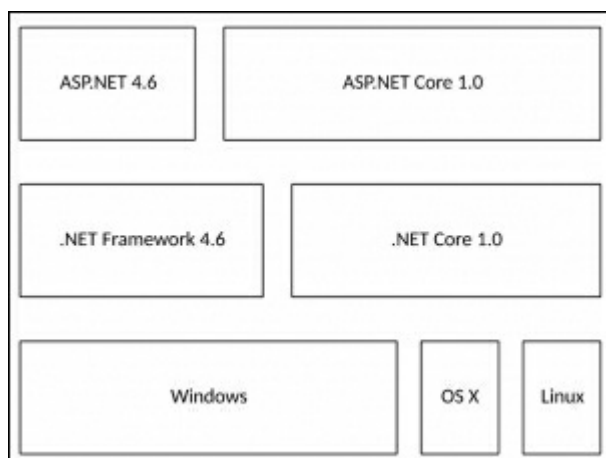
ASP.NET Core 1.0 can run against the existing .NET Framework 4.6 or the new .NET Core 1.0 framework. Similarly, .NET Core can run on Windows, Mac OS X, and Linux, but the old .NET only runs on Windows.

There is also the Mono framework, which has been omitted for clarity. This was a previous project that allowed .NET to run on multiple platforms. Mono was recently acquired by Microsoft, and it was open sourced (along with other Xamarin products). Therefore, you should be able to run ASP.NET Core using Mono on any supported operating system.

.NET Core focuses on web-application development and server-side frameworks. It is not as feature filled as the existing .NET Framework. If you write native-graphical desktop applications, perhaps using **Windows Presentation Foundation (WPF)**, then you should stick with .NET 4.6.

As this book is mainly about web-application development, we will use the latest Core versions of all software. We will investigate the performance implications of various operating systems and architectures. This is particularly important if your deployment target is a computer, such as the Raspberry Pi, which uses a processor with an ARM architecture. It also has limited memory, which is important to consider when using a managed runtime that includes garbage collection, such as .NET.

Summary

Let's sum up what we covered in this introductory chapter and what we will cover in the next chapter. We introduced the concept of treating performance as a feature, and we covered why this is important. We also briefly touched on some common performance problems and why we often miss them in the software development process. We'll cover these in more detail later on in this book.

We showed the performance differences between various types of storage hardware. We highlighted the importance of knowing what your code runs on and, crucially, what it will run on when your users see it. We talked about how the process of scaling systems has changed from what it used to be, how scaling is now performed horizontally instead of vertically, and how you can take advantage of this in the architecting of your code and systems.

We showed you the tools that you can use and the licensing implications of some of them. We also explained the new world of .NET and how these latest frameworks fit in with the stable ones. We touched upon why measurement is vitally important. In the next chapter, we'll expand on this and show you how to measure your software to see whether it's slow.

Chapter 2. Measuring Performance Bottlenecks

Measurement is the most crucial aspect of building high performance systems. You can't change what you can't measure, because you won't know what effect your change has had, if any. Without measuring your application you won't know if it's performing well.

If you only go by when your software feels slow then you have left it too late. You are reactively fixing a problem rather than proactively avoiding one. You must measure to achieve good performance even though it's the feel that matters to a user.

Some books leave measurement, analysis, and profiling until the end. Yet this is the first thing that should be considered. It's easy to fix the wrong problem and optimize areas that are not having performance difficulties.

In this chapter we will cover the following topics.

- **Structured Query Language (SQL)** database profiling
  - SQL Server Profiler
  - MiniProfiler
- Web application profiling
  - Glimpse
  - **Integrated Development Environment (IDE)** profilers
- HTTP monitoring
  - Browser developer tools
  - Fiddler proxy
- Network monitoring
  - Microsoft message analyzer
  - Wireshark
- Scientific method and repeatability

This chapter will show you how to measure if there are performance issues and where they are occurring. We will describe the tools that can give you this information and demonstrate how to use them effectively and correctly. We'll also show you how to repeat your experiments consistently so that you can tell if you have fixed a problem once you've identified it.

We will cover measurement again towards the end of the book, but there we'll focus on continuous automated monitoring to avoid regressions. This chapter will focus more on manual testing to identify potential performance problems during

development and debugging.

Tools

Good debugging tools are essential in discovering where problems lie. You can write your own crude timing code and we will show you how. However, purpose built tools are much nicer to work with.

Many of the tools in this chapter help examine areas external to your code. We will cover profiling of code too, but it's hard to identify problems this way unless the work is purely computational. Slowdowns often happen because of actions your app initiates outside of its immediate stack, and these can be hard to debug by simply stepping through the code.

Moving through your program line-by-line slows down execution so much that it can be difficult to identify which lines are fast and which are slow. The same approach taken for fixing functional bugs cannot always be applied to fix performance issues.

One of the problems with adopting a new framework (such as ASP.NET Core) early is that it can take a while for the existing tools to be updated to work with it. We will point out when this is the case but these compatibility problems should improve over time. As many of the tools are open source, you could help out and contribute to the community.

SQL

First off we will cover SQL related issues, so if you're not using a relational database then you can skip this bit, perhaps if you're using a NoSQL store or a document database instead. Relational databases are a very mature technology and are flexible in their uses. However, it is essential to have a basic knowledge of the SQL syntax and how databases work in order to use them effectively.

It can be tempting when using an O/RM such as **Entity Framework (EF)** to ignore SQL and stay in a C# world, but a competent developer should be able to write a high performance SQL query. Ignoring the realities of how a database engine works will often lead to performance issues. It's easy to write code with an O/RM that's too chatty with the database and issues far too many queries for an operation. Not having the correct indexes on a table will also result in poor performance.

During development you may not notice these mistakes, unless you use tools to identify the inefficient events occurring. Here we will show you a couple of ways of doing this.

SQL Server Profiler

SQL Server Profiler is a tool that allows you to inspect what commands are being executed on your SQL Server database. If you have the management tools installed then you should already have it on your machine.

1. If you are using Microsoft SQL Server, then **SQL Server Profiler** can be accessed from the **Tools** menu of **SQL Server Management Studio** (**SSMS**).

2. Load SQL Server Profiler and connect to the database that you are using. Name your trace and select the **Tuning** profile template.

3. Click **Run** and you should see that the trace has begun. You are now ready to run your code against the database to see what queries it's actually executing.

Executing a simple query

As a test you can execute a simple select query with Management Studio and the profile window should be flooded with entries. Locate the query that you just executed among the noise. This is easier if it starts with a comment, as shown in the

following screenshot.

The **Duration** column shows the time taken for the query in milliseconds (ms). In this example, selecting the top 1000 rows from a table containing over a million entries took **246** ms. This would appear very quick, almost instantaneous, to the user. Modifying the query to return all rows makes it much slower, as shown in the following screenshot:

The query has now taken over 13 seconds (**13455** ms) to complete, which is noticeably slow. This is an extreme example, but it is quite a common mistake to request more data than needed and to filter or sort it in application code. The database is often much better suited to this task and is usually the correct location to select the data you want.

We will cover specific SQL mistakes and remedies in the following chapters. However, the first step is knowing how to detect what is going on with the communication between your application and the database. If you can't detect that your app is making inefficient queries then it will be difficult to improve performance.

MiniProfiler

MiniProfiler is an excellent open source tool for debugging data queries. It supports SQL databases and some NoSQL databases, such as Mongo and Raven. It came out of Stack Exchange, the people who run the Stack Overflow Q&A site. It embeds a

widget into your web pages that shows you how long they take to get their data. It also shows you the SQL queries and warns you about common mistakes. Although MiniProfiler started with .NET, it is also available for Ruby, Go, and Node.js.

The biggest benefit of MiniProfiler over SQL Server Profiler is that it's always there. You don't need to explicitly go looking for SQL problems and so it can highlight issues much earlier. It's even a good idea to run it in production, only visible to logged in admins. This way, every time you work on the website you will see how the data access is performing. Make sure you test for the performance impacts of this before deploying it though.

Unfortunately MiniProfiler doesn't yet support ASP.NET Core 1.0. It relies on the `System.Web` library and doesn't tie into the **Open Web Interface for .NET (OWIN)** style lifecycle used by the new framework. Hopefully it will soon support ASP.NET Core (and may do by the time you read this) as it's a very useful tool.

Tip

If you're using a supported version of ASP.NET then MiniProfiler is highly recommended. ASP.NET Core support is planned, so keep an eye on ANCLAFS.com for the latest support details.

Due to these limitations, we won't cover MiniProfiler in too much detail but you can get more information at miniprofiler.com . If you are using a previous version of ASP.NET then you can install it into your project with the NuGet package manager. Type `Install-Package MiniProfiler` into the package manager PowerShell console window in Visual Studio. Then follow the rest of the setup instructions from the website.

Application profiling

Often you will want a breakdown of where all the time is being taken up within your application. There are various tools available for this and we will cover a couple of them here.

Glimpse

Glimpse is a fantastic open source add-on for your web application. Like MiniProfiler, it adds a widget to your web pages so that you can see problems as you navigate around and work on your site. It provides information similar to your browser developer tools but also delves inside your server side application to show you a trace of what actions are taking the most time.

Glimpse is available from getglimpse.com and you can install it with NuGet for the web framework and O/RM you're using. For ASP.NET Core we will need to use Glimpse version 2. This is currently a beta prerelease, but by the time you read this it may be stable.

Using Glimpse

Installing Glimpse is really simple, there are only three steps.

1. Install with NuGet.
2. Add lines to `Startup.cs`.
3. Build and run the app.

Let's have a look at these steps.

Installing the package

Right-click on your web application project in the Solution Explorer and select **Manage NuGet Packages...** to open the graphical package manager window. Search for Glimpse, select it, and then click on the **Install** button. If you want to install the beta version of Glimpse 2, then make sure the **Include prerelease** checkbox is selected:
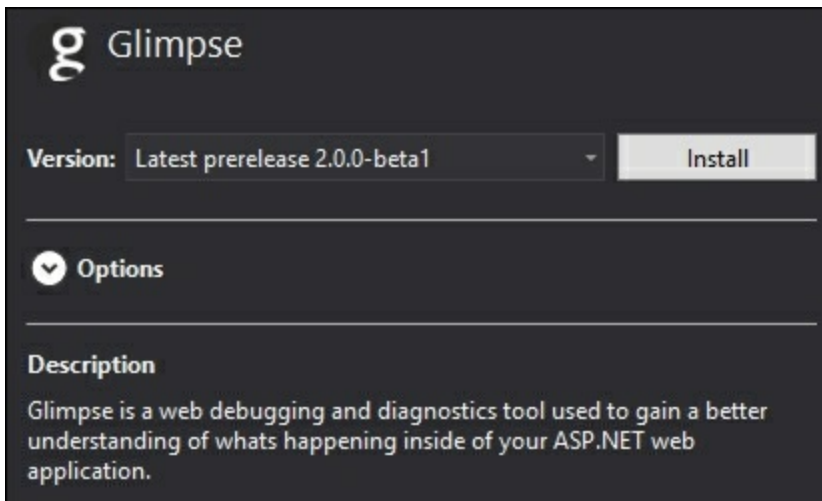
Add code

You need to add three snippets of code to your `Startup.cs` file. In the `using` directives at the top of the file, add the following:

```
using Glimpse;
```

In the `ConfigureServices` function, add the following:

```
services.AddGlimpse();
```

In the `Configure` function, add the following:

```
app.UseGlimpse();
```

Running your web application

Build and run your web application by pressing **F5**. If you encounter a duplicate type error when running then simply clean the solution and do a full rebuild. You may need to apply migrations to the database, but this is just a button click in the browser. However, if you add this to an existing project with data, then you should take more care.

You should then look at your web application with the Glimpse bar at the bottom of the browser window, which looks like the following screenshot. The bar is one long strip, but it has been broken up in this screenshot to display it more clearly:
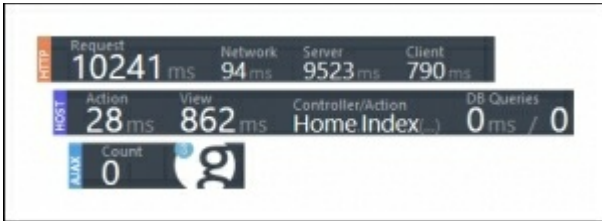
Note

The first page load may take much longer than subsequent ones, so you should refresh for more representative data. However, we won't worry about this for the purposes of this demo.

Mouse over each section in the Glimpse toolbar to expand it for more information. You can minimize and restore each section (HTTP, HOST, or AJAX) by clicking on its title.

If you used the default web application template (and left the authentication as the default individual user accounts), then you can register a new user to cause the application to perform some database operations. You will then see some values in the **DB queries** field, as shown in the following screenshot:
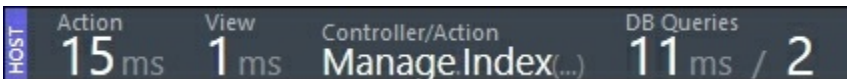
Click on the Glimpse logo to see a history of all the page requests. Select one to view the details and expand the SQL queries by clicking on them, which is displayed in the following screenshot:
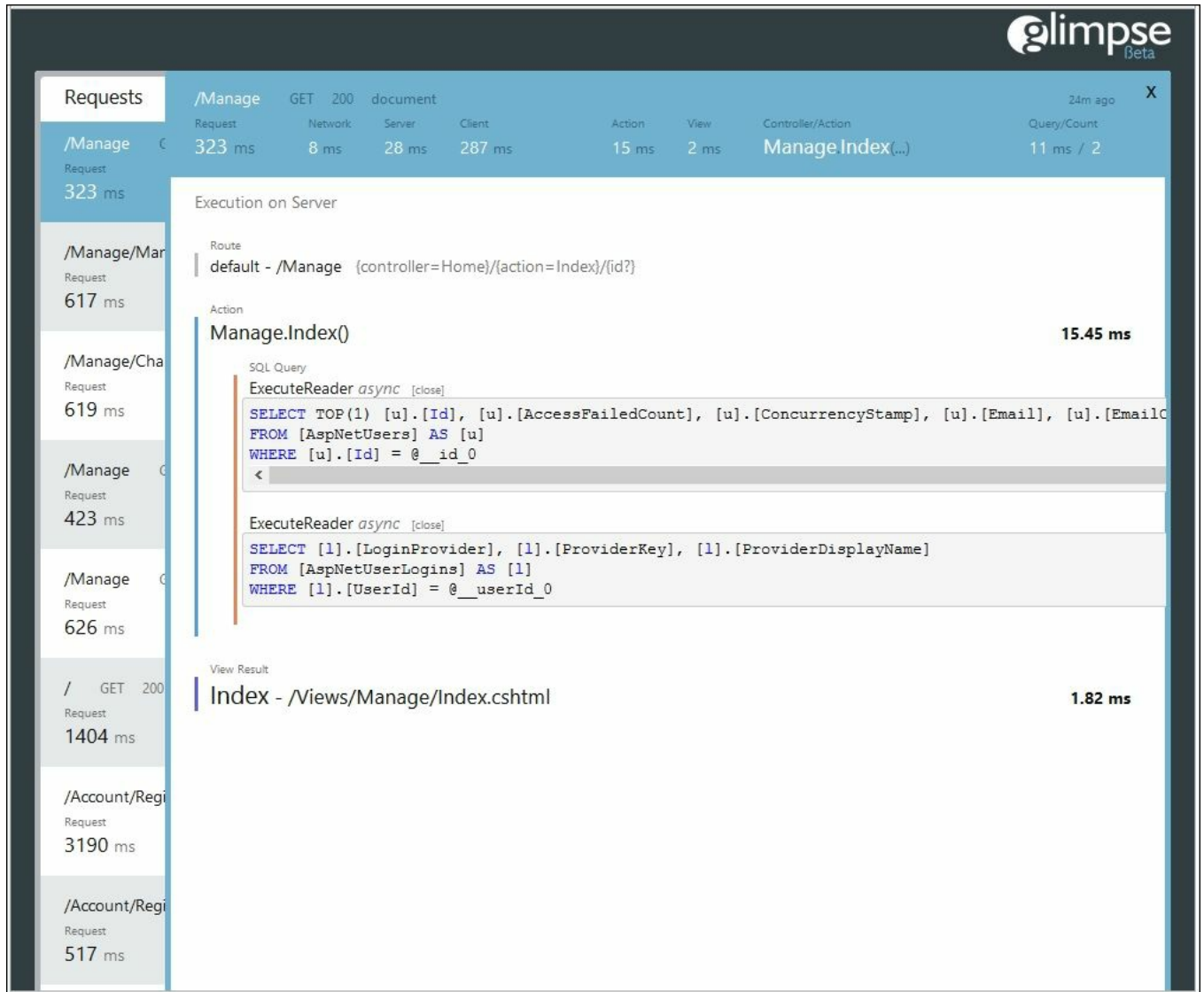
Glimpse shows you how much time is spent in each layer of your application and exactly which SQL queries are being run. In this case, EF Core 1.0 generates the SQL.

Glimpse is very useful to track down where performance problems lie. You can easily see how long each part of the page pipeline takes and identify slow parts.

IDE

Using the profiling tools that are built into **Visual Studio (VS)** can be very informative to understand the CPU and memory usage of your code. You can also see the time taken for certain operations.

When running your application, open the diagnostic tools window in VS, as shown in the following screenshot:
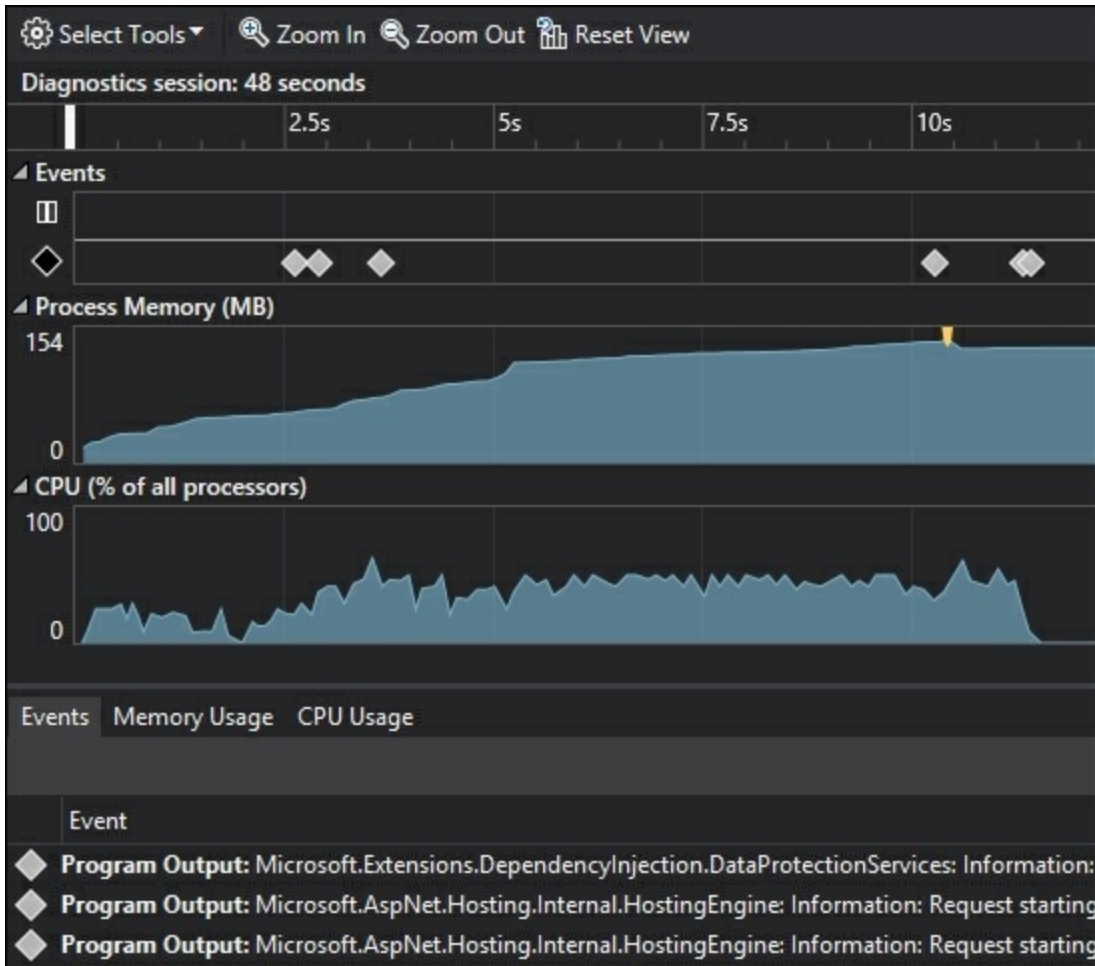
You can see the CPU and memory profiles, including the automatic Garbage Collection events that are used to free up memory (the marker just prior to memory use decreasing). You will be able to see breakpoint events and, if you have the enterprise version of VS, IntelliTrace events as well.

Note

IntelliTrace is only available in the enterprise edition of Visual Studio. However, you can still use the performance and diagnostic tools in the free community edition.

If you have **IntelliTrace**, then you can find it in the VS options, as shown in the following screenshot. However, the diagnostic tools are still useful without this premium feature:
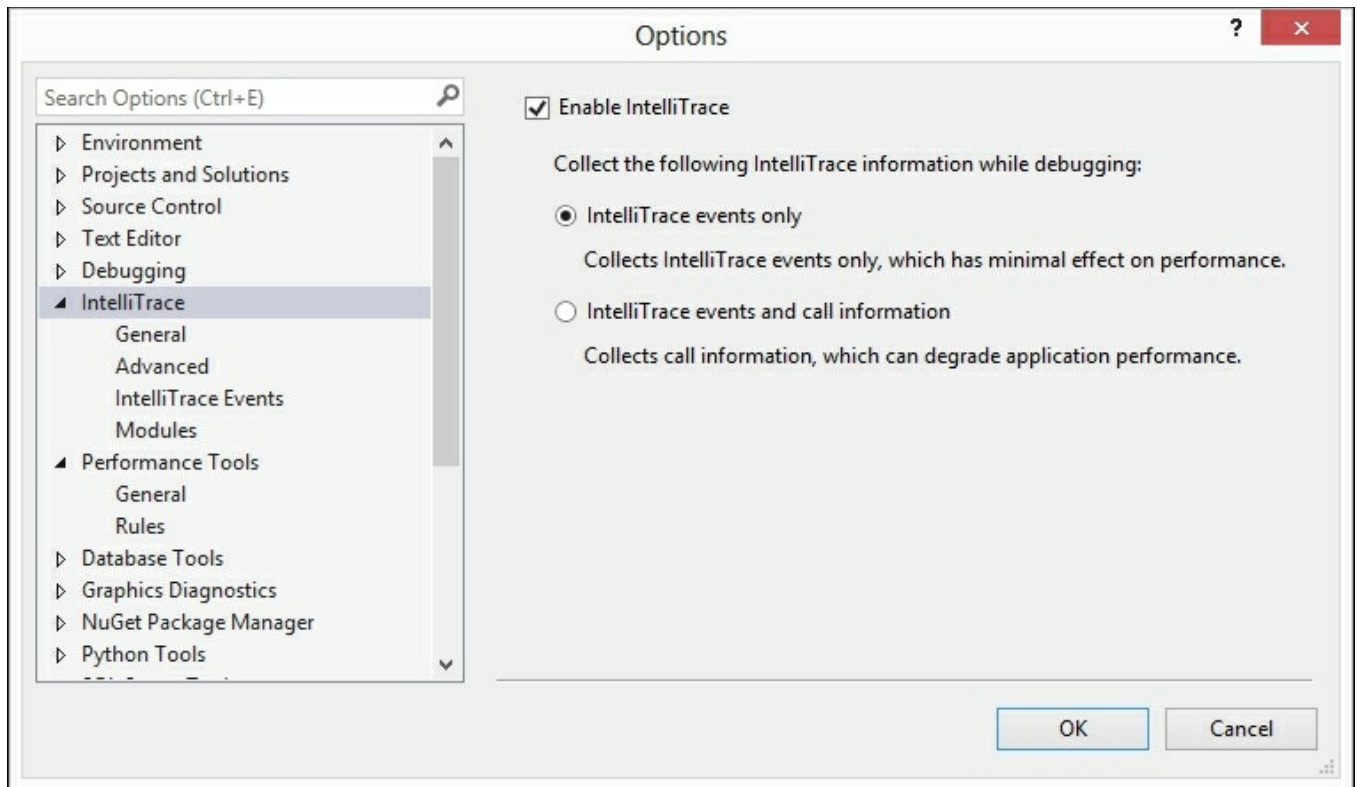
When you put in a breakpoint and your code hits it, then VS will tell you how long it was since the previous event. This is shown in the events list and also overlaid near the breakpoint.

Tip

You can't install the community edition and enterprise edition of Visual Studio on the same machine. Use a **Virtual Machine (VM)**, for example, with **Hyper-V** or VirtualBox, if you wish to run both.

Alternatives to VS tools are Redgate ANTS and Telerik JustTrace. JetBrains also have dotTrace and dotMemory. However, all of these tools can be quite expensive and we won't cover them here.

Monitoring HTTP

When dealing with a web application, you will normally use HTTP as the application protocol. It's very useful to know what requests occur between the browsers and your servers.

Browsers

Most modern browsers have excellent developer tools, which include a network tab to monitor requests to and responses from the web server. You can normally access the dev tools by pressing **F12**. These are handy to view the web traffic and you can still see encrypted communications without messing about with certificates.

The dev tools in both Chrome and Firefox are superb. We'll focus on the network and timing component, but we highly recommend that you learn all of the features. If you know how to get the best out of them, then web development is much easier.

Chrome

The network dev tools in Chrome are very useful. They provide a good visualization of the request timings, and you can throttle the connection to see how it behaves over various different internet connections. This is particularly important for mobile devices.

A selection of network requests from the Chrome dev tools are shown in the following screenshot. Additional columns are available if you right-click on the **Headers** bar:

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
localhost	GET	200 OK	document	Other	2.8 KB 8.5 KB	49 ms 43 ms
bootstrap.css /lib/bootstrap/dist/css	GET	200 OK	stylesheet	(index):9 Parser	31.8 KB 151 KB	397 ms 147 ms
site.css /css	GET	200 OK	stylesheet	(index):10 Parser	873 B 618 B	140 ms 139 ms
ASP-NET-Banners-01.png /images	GET	200 OK	png	(index):52 Parser	8.5 KB 8.1 KB	214 ms 150 ms

You can disable the cache with a simple checkbox so that the browser will always load assets fresh from your web server. You can also click on a resource for more information and the timing tab is very useful to provide a breakdown of the

connection components, for example, **Time To First Byte** (**TTFB**). Some basic timing details from a local web server are shown in the following screenshot:
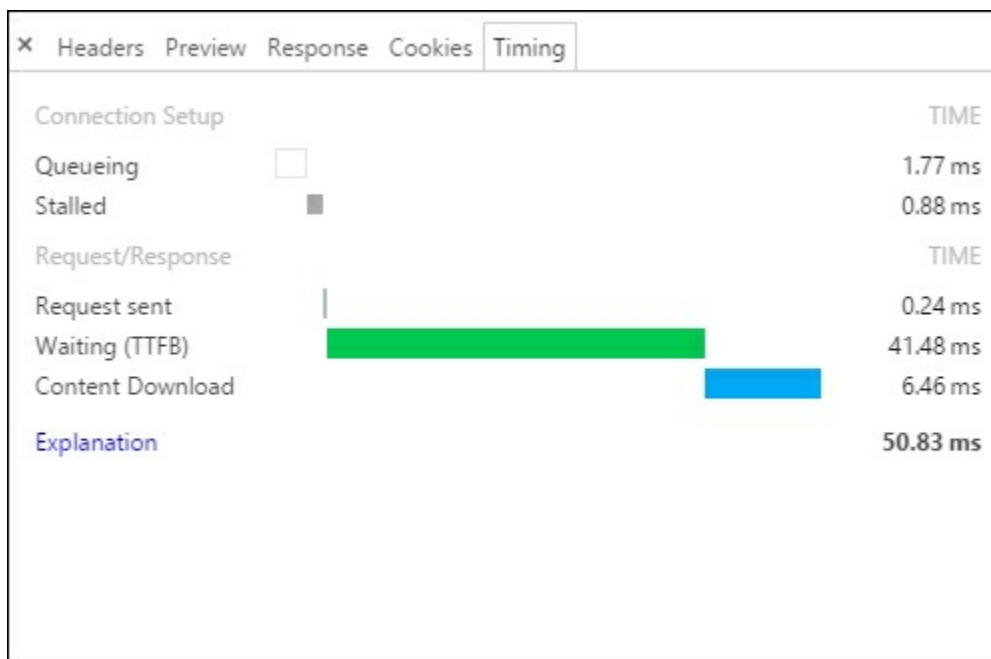
On a local web server, this breakdown won't contain much extra information, but on a remote server it will display other things, such as the **Domain Name System** (**DNS**) hostname lookup, and SSL/TLS handshake. These additional components are shown in the next screenshot:
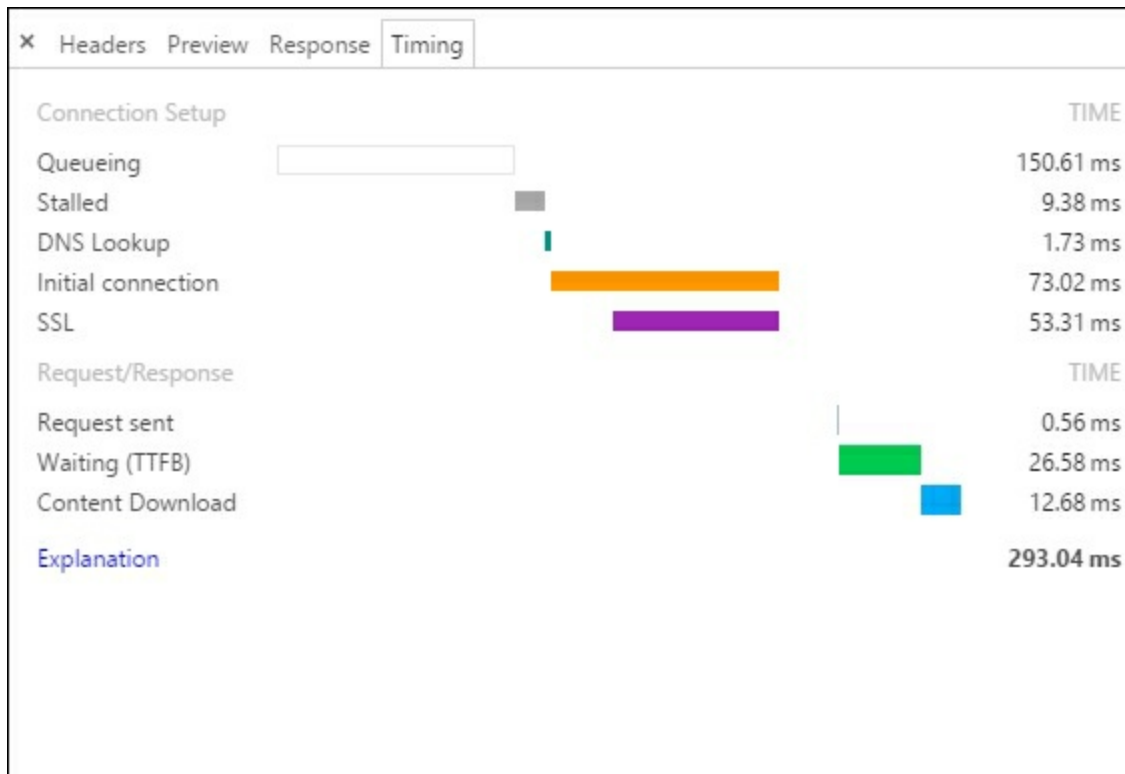
Firefox

Firefox has similar dev tools to Chrome but with some added features. For example, you can edit a request and resend it to the web server. The **Network** tab presents the same sort of information as Chrome does, as shown in the following screenshot:

The detail view is also very similar, including the **Timings** tab. This tab is shown in the following screenshot:
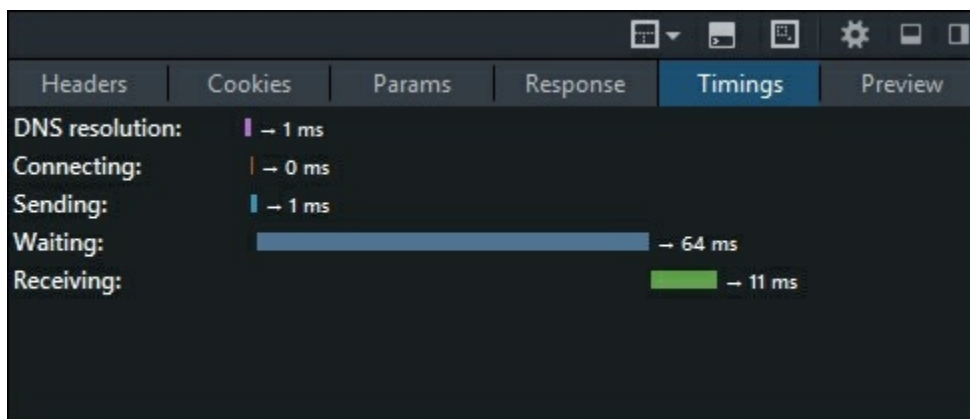
Fiddler

Sometimes, browser tools aren't enough. Perhaps, you are debugging a native application, backend web client, or mobile browser. Fiddler is a free debugging proxy that can capture all HTTP traffic between the client and server. With a little bit of work, it can also intercept HTTPS traffic. Fiddler is available at www.telerik.com/fiddler .

As this book focuses on web development we won't go into more detail. The browser dev tools are suitable for most work these days. They now fulfil a large part of the role that Fiddler used to play before they acquired the same features. Fiddler is still there if you need it and it can be handy if your web server calls an external HTTP API. Although this can also often be debugged directly inside VS.

Network

Occasionally, you will need to debug at a lower level than HTTP or SQL. This is where network monitors or packet sniffers come in. Perhaps, you want to debug a **Tabular Data Stream** (**TDS**) message to a SQL Server DB or a TLS handshake to an SSL terminating load balancer. Or maybe you want to analyze a SOAP web service envelope or **Simple Mail Transfer Protocol** (**SMTP**) e-mail connection to see why it's not working correctly.

Microsoft Message Analyzer

Microsoft Message Analyzer supersedes Microsoft Network Monitor (**Netmon**) and is a tool to capture network traffic on Windows systems. Netmon requires you to log out and back in again after installation, whereas Wireshark doesn't. You can read more about these two Microsoft tools online, but, for clarity and brevity, we will focus on Wireshark for low-level network monitoring.

Wireshark

Wireshark (previously called Ethereal) is an open source and cross-platform packet capture and network analysis application. It is probably the most popular tool of its kind and has many uses. You can install Wireshark without needing to restart or log out which makes it great to debug a live problem that's hard to recreate. You can download Wireshark from www.wireshark.org and it runs on Windows, Mac OS X, and Linux.

Wireshark isn't particularly useful for local development as it only captures traffic going over the network, not to localhost. The only thing you are likely to see if you run it against a local web application is VS reporting what you do back to Microsoft.

Tip

You can turn off the **Customer Experience Improvement Program** (**CEIP**) by clicking on the button next to the quick launch box and selecting **Settings...**. By default, it is on (it's now opt-out rather than the opt-in of previous products).

Click on the fin icon button at the top-left of Wireshark to start a trace. Then perform some network actions, such as loading a webpage from a test server. Once you capture some packets, click on the stop button and you can then examine the collected data at your leisure.

Note

Ask your IT administrator before running Wireshark. It can often pick up sensitive information off of the LAN, which may be against your IT acceptable use policy.

The following screenshot shows part of the results from a Wireshark capture.
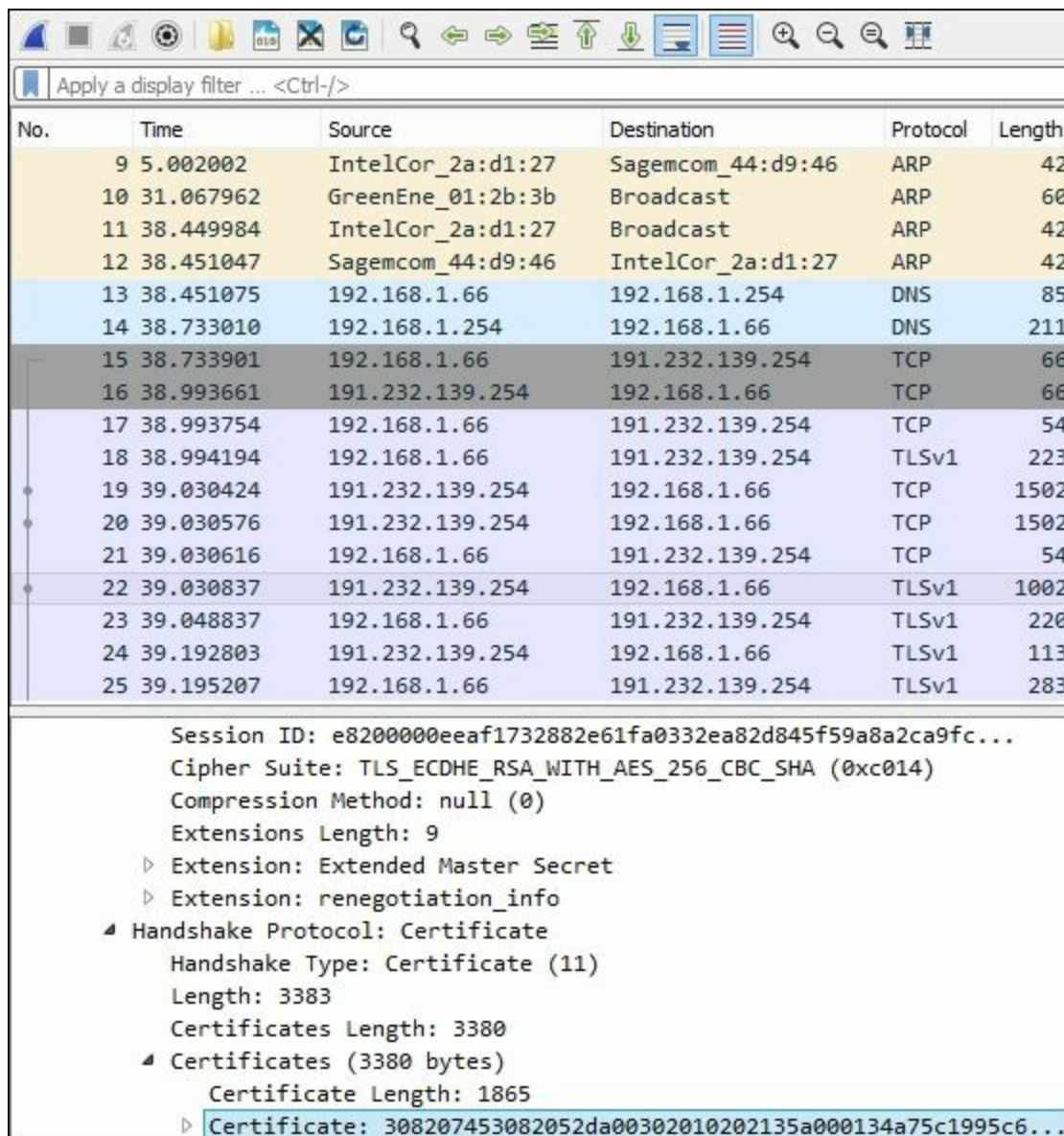
There can be a lot of noise when using Wireshark. You will see low-level network packets, such as **Address Resolution Protocol** (**ARP**) traffic, which you can filter out or ignore. You may also see data from VoIP phones and other networked devices or data that is intended for other computers on the network.

Select the packet that you are interested in from the top pane. The middle pane will display the contents of the packet. You can normally ignore the lower layers of the network stack, such as Ethernet and TCP/IP (presented at the top of the list).

Dive straight into the application layer that is listed last. If this is a protocol that Wireshark recognizes, then it will break it down into the fields that it's made up of.

The bottom pane displays a hex dump of the raw packet. This is not normally as useful as the parsed data in the middle pane.

If you use TLS/SSL (hint: you should), then you won't see the contents of HTTP traffic. You would need a copy of the server's private key to see inside the TLS connection, which wraps and encrypts the HTTP application data. You will only be able to see the domain that was connected to via the DNS lookup and TLS certificate, not the full URL or any payload data.

Using Wireshark is a huge topic, and there are many great resources available to learn about it, both online and offline. We won't go into much more detail here because it's usually not necessary to go down to this low level of network scrutiny. However, this is a good tool to have in your back pocket.

Roll your own

Sometimes, you may want to write your own performance measurement code. Make sure that you have exhausted all other options and investigated the available tools first.

Note

Perhaps, you want to record the execution time of some process to write to your logs. Maybe you want to send this to a system such as Logstash then visualize the changes over time with **Kibana**. Both are great open source products from Elastic, and they store data in the Elasticsearch search server. You can read more about both of these tools at elastic.co

You can easily record the length of time for a task by storing the current time before it starts and comparing this to the current time after it finishes, but this approach has many limitations. The act of measuring will affect the result to some extent, and the clock is not accurate for short durations. It can be useful for really slow events if you need to share states between processes or multiple runs, but, to benchmark, you should normally use the `Stopwatch` class.

Tip

It is usually best to store timestamps in **Coordinated Universal Time** (**UTC**), which is otherwise known as **Greenwich Mean Time** (**GMT**), and name timestamps in UTC. You will avoid many issues with time zones and daylight saving if you use `DateTimeOffset.UtcNow` (or at least `DateTime.UtcNow`). Name variables and columns to indicate this, for example, `TimestampUtc`.

Use `TimeSpan` for lengths of time, but, if you must use primitives (such as integers), then include the units in the variable or column name. For example, `DurationInMilliseconds` or `TimeoutInSeconds`. This will help to avoid confusion when another developer (or your future self) comes to use them.

However, to benchmark quick operations, you should use a `Stopwatch`. This class is in the `System.Diagnostics` namespace.

If you try to measure a single quick event, then you will not get accurate results. A way around this is to repeat the task many times and then take an average. This is useful to benchmark, but it is not usually applicable to real applications. However, once you identify what works quickest with a test, then you can apply it to your own programs.

Let's illustrate this with a small experiment to time how long it takes to hash a password with the PBKDF2 algorithm (in the `System.Security.Cryptography` namespace). In this case, the operation under test is not important, as we are simply interested in the timing code. A Naïve approach may look like the following code:

```
var s = Stopwatch.StartNew();
pbkdf2.GetBytes(2048);
s.Stop();
Console.WriteLine($"Test duration = {s.ElapsedMilliseconds} ms");
```

This code will output a different value every time it is run, due to the flaws in the measurement process. A better way would be to repeat the test many times and average the output, like the following code:

```
var s = Stopwatch.StartNew();
for (var ii = 0; ii < tests; ii++)
{
    pbkdf2.GetBytes(2048);
}
s.Stop();
var mean = s.ElapsedMilliseconds / tests;
Console.WriteLine($"{tests} runs mean duration = {mean} ms");
```

This code will output very similar results every time. The higher the value of `tests`, the more accurate it will be, but the longer the test will take.

Note

We use the new concise string-formatting method here, but you can use the traditional overloads to `Console.WriteLine` if you prefer.
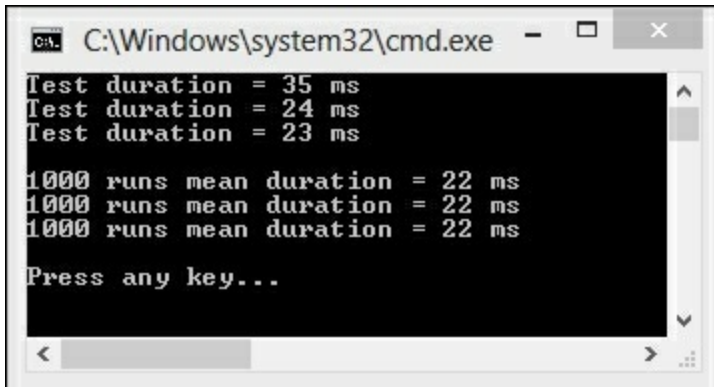
Let's write a quick example application that demonstrates the differences by running these two different versions multiple times. We'll extract the two tests into methods and call them each a few times:

```
var pbkdf2 = new Rfc2898DeriveBytes("password", 64, 256);
SingleTest(pbkdf2);
SingleTest(pbkdf2);
SingleTest(pbkdf2);

Console.WriteLine();
var tests = 1000;
AvgTest(pbkdf2, tests);
AvgTest(pbkdf2, tests);
AvgTest(pbkdf2, tests);

Console.WriteLine();
Console.WriteLine("Press any key...");
Console.ReadKey(true);
```

The output will look something like the following screenshot. You can find the full application listing in the code that accompanies this book if you want to run it for yourself:

You can see that the three individual tests can give wildly different results and yet the averaged tests are identical.
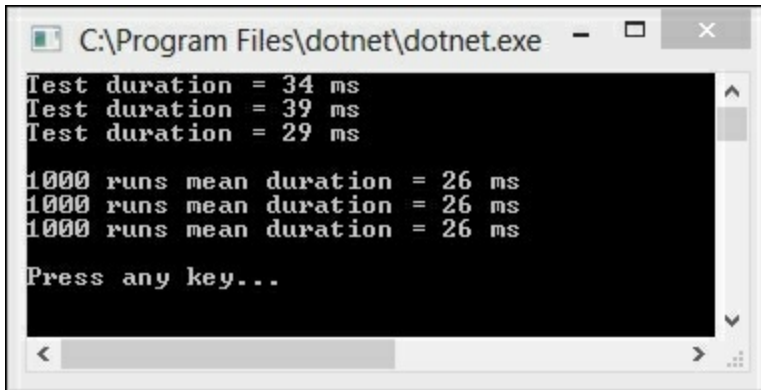
Tip

Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look. The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/ASP.NET-Core-1.0-High-Performance. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Your results will vary. This is due to the inherent variability in computer systems. Embedded systems that are time sensitive usually use a real time OS. Normal software systems typically run on a time-sharing OS, where your instructions can easily get interrupted, and VMs make the problem even worse.

You will get different results, depending on whether you build in debug or release mode and if you run with or without debugging. Release mode without debugging (*Ctrl + F5*) is the fastest.

The following screenshot shows the same benchmarking demo application running with debugging. You can tell because the `dotnet` executable is shown in the title bar of the command prompt. If it ran without debugging, then this would display `cmd.exe` (on Windows), as in the previous screenshot.

Note

Unit testing is very valuable and you may even practice **Test-Driven Development (TDD)**, but you should be careful about including performance tests as unit tests. Unit tests must be quick to be valuable, and tests that accurately measure the time taken for operations are often slow. You should set a timeout on your unit tests to make sure that you don't write slow ones with external dependencies. You can still test performance, but you should do it in the integration testing stage along with tests that hit an API, DB, or disk.

Science

We dealt with the computer in computer science by showcasing some hardware in the previous chapter. Now, it's time for the science bit.

It's important to take a scientific approach if you wish to achieve consistently reliable results. Have a methodology or test plan and follow it the same way every time, only changing the thing that you want to measure. Automation can help a lot with this.

It's also important to always measure for your use case on your systems with your data. What worked well for someone else may not work out great for you.

We will talk more about science and statistics later in the book. Taking a simple average can be misleading but it's fine to use it as a gentle introduction. Read Chapter 8, *The Downsides of Performance Enhancing Tools,* for more on concepts such as medians and percentiles.

Repeatability

Results need to be repeatable. If you get wildly different results every time you test, then they can't be relied upon. You should repeat tests and take the average result to normalize out any variability in the application or hardware under test.

It is also important to clearly record the units of measurement. When you compare a new value to a historic one, you need to know this. NASA famously lost a Mars probe because of unit confusion.

Only change one thing

When testing, you aim to measure the impact of a single change. If you change more than one thing at a time, then you cannot be sure which one has made the difference.

The aim is to minimize the effects of any other changes apart from the one you are interested in. This means keeping external factors as static as possible and performing multiple tests, taking the average result.

Summary

Let's sum up what we covered about measurement in this chapter and what we'll cover in the next chapter. We covered the importance of measurement in solving performance problems. Without measuring, you cannot hope to write high-performance software; you will be coding in the dark.

We highlighted some of the tools that you can use to measure performance. We showed you how to use a selection of these and how to write your own.

We also covered the value of taking a scientific approach to measurement. Making sure that your results are repeatable and that you record the correct units of measurement are important concerns.

In the next chapter, we will learn how to fix common performance problems. You will gain the skills to speed up the low-hanging fruit and make yourself look like a performance wizard to your colleagues. No longer will it be a case of it worked in test, it's an operations problem now.

Chapter 3. Fixing Common Performance Problems

This chapter gets into the meat of optimization, once you identify and locate performance problems. It covers a selection of the most common performance issues across a variety of areas and explains simple solutions to some of the mistakes people often make. When using these techniques, you'll look like a wizard to your clients and colleagues by quickly speeding up their software.

Topics covered in this chapter include the following:

- Network latency
- **Select N+1** problems
- Disk I/O issues on virtual machines
- Asynchronous operations in a web application
- Performing too many operations in one web request
- Static site generators
- Pragmatic solutions with hardware
- Shrinking overly-large images

Most of the problems in this chapter center on what happens when you add latency to common operations or when throughput is reduced from what it was in development. Things that worked fine in test when everything was on one physical machine with minimal data are now no longer quite as speedy when you have an API on a different continent, a full database on a different machine to your web server, and its virtual disk somewhere else on the network entirely.

You will learn how to identify and fix issues that are not always apparent when everything is running on a single machine. You'll see how to identify when your O/RM or framework behaves badly and is too chatty with the database, which can easily happen if it's not used correctly.

We will see how to ensure that work is performed in the most appropriate place, and we'll look at some ways of keeping your images small using the correct resolution and format. These techniques will ensure that your application is efficient and that data is not sent over the wire unnecessarily.

We'll also discuss how to mitigate performance issues with an alternative approach by improving the underlying hardware to reduce the factors that amplify issues in bad software. This can be a good temporary measure if the software is already deployed to production and in use. If you already have live performance problems then this can

buy you some time to engineer a proper fix.

Latency

As covered in previous chapters, latency is the delay that occurs before an operation can complete, sometimes also known as **lag**. You may not be able to control the latency of the infrastructure that your software runs on, but you can write your application in such a way that it can cope with this latency in a graceful manner.

The two main types of latency that we will discuss here are **network latency** and **disk latency**. As the names suggest these are, respectively, the delay in performing an operation over the network and the delay to read from or write to a persistent storage medium. You will often deal with both at the same time, for example, a **database (DB)** query to a server on a remote virtual machine will require the following operations:

- A network operation from web server to DB server
- A network operation from DB server to remote disk on a **Storage Area Network (SAN)**
- A disk operation to look up data on the physical drive

Note

Although **Solid State Drives (SSDs)** have much lower latency than spinning platter disks, they are still relatively slow. When we talk about disk I/O here, we refer to both types of drive.

You can clearly see that, if you issue too many DB operations, the latency present in typical production infrastructure will compound the problem. You can fix this by minimizing the number of DB operations so that they can't be amplified as much.

Let's illustrate this with an example. Let's suppose you wish to return 200 records from your DB and the round trip latency is 50 milliseconds (ms). If you retrieve all of the records at once, then the total time will be 50 ms plus the time to transfer the records. However, if you first retrieve a list of the record identifiers and then retrieve all of them individually, the total time will be at least *201 * 50 ms = 10.05 seconds*!

Unfortunately, this is a very common mistake. In a system where latency dominates throughput, it is important to keep requests to a minimum.

Asynchronous operations

Most new .NET framework APIs that have significant latency will have **asynchronous** (**async**) methods. For example, the .NET HTTP client (superseding the web client), SMTP client, and **Entity Framework** (**EF**) all have async versions of common methods. In fact, the async version is usually the native implementation and the non-async method is simply a blocking wrapper to it. These methods are very beneficial and you should use them. However, they may not have the effect that you imagine when applied to web application programming.

Note

We will cover async operations and asynchronous architecture later in this book. We'll also go into **Message Queuing** (**MQ**) and worker services. This chapter is just a quick introduction and we will simply show you some tools to go after the low-hanging fruit on web applications.

An async API returns control to the calling method before it completes. This can also be awaited so that on completion, execution resumes from where the asynchronous call was made. With a native desktop or mobile application, awaiting an async method returns control to the **user interface** (**UI**) thread, which means that the software remains responsive to user input. The app can process user interactions rather than blocking on your method. Traditionally, you may have used a background worker for these tasks.

You should never perform expensive work on the UI thread. Therefore, this technique does increase performance for native applications. However, for a web application this UI blocking problem does not exist because the browser is the UI. Therefore, this technique will not increase performance for a single user in isolation.

Awaiting asynchronous API methods in a web application is still good practice, but it only allows the software to scale better and handle more concurrent users. A web request typically cannot complete until the async operation also completes. Therefore, although the thread is surrendered back into the thread pool and you can use it for other requests, the individual web request will not complete quicker.

Simple asynchronous tools

As this book deals with web application programming, we won't go into much more detail on native application UIs in this chapter. Instead, we will showcase some simple tools and techniques that can help with async tasks in web applications.

The tools we are about to cover offer some simple solutions that are only suitable for very small applications. They may not always be reliable, but sometimes they can be good enough. If you are after a more robust solution, then you should read the later chapters about distributed architecture.

Background queuing

Background queuing is a useful technique when you have an operation that does not need to occur straight away. For example, logging stats to a database, sending an e-mail, or processing a payment transaction. If you perform too much work in a single web request, then background queuing may offer a convenient solution, especially if you don't require the operation to always succeed.

If you use ASP.NET 4.6 (or any version from 4.5.2 onwards), then you can use `HostingEnvironment.QueueBackgroundWorkItem` to run a method in the background. This is preferable to simply setting a task running, as if ASP.NET shuts down then it will issue a cancellation request and wait for a grace period before killing the item. However, this still does not guarantee completion because the application can die at any point due to an unexpected reboot or hardware failure. If the task needs to complete, then it should be **transactional** and make a record of success upon completion. It can then be retried if it failed. Queuing a background work item is okay for fire-and-forget events, if you genuinely don't care whether they succeed or not.

Unfortunately, `HostingEnvironment.QueueBackgroundWorkItem` is not part of ASP.NET Core. Therefore, if you want to use this, then you will have to simply queue a job. We will show you how to do this later, but if you use the full version of ASP.NET, then you can do the following to send an e-mail in the background:

```
var client = new SmtpClient();
HostingEnvironment.QueueBackgroundWorkItem(ct =>
    client.SendMailAsync(message));
```

Assuming that you already have your message, this will create an SMTP client and send the e-mail message in the background without blocking further execution. This does not use the `ct` (cancellation token) variable. Keep in mind that the e-mail is not guaranteed to be sent. Therefore, if you need to definitely dispatch it, then consider using another method.

If you use ASP.NET Core, then this functionality is not available. However, you can manually create something similar with `Task.Run` as in the following example. However, this is probably not the best approach for anything nontrivial:

```
Task.Run(() => asyncMethod(cancellationToken));
```

If you can cancel your task, then you can get the `ApplicationStopping` token from an injected instance of the `IApplicationLifetime` interface to pass in as your cancellation token. This will let your task know when the application is about to stop,

and you can also block shutdown with it while you gracefully clean up.

You should use this technique with caution, so we won't give you a full example here. Although, you should now have enough pointers to dig deeper and understand the ASP.NET Core application lifecycle if you wish.

Hangfire

Hangfire is an excellent library to run simple background jobs. It does not support ASP.NET Core yet, but this is planned (refer to ANCLAFS.com for the latest information). If you use the full .NET, then you should consider it and you can read more at hangfire.io .

You need persistent storage, such as SQL Server, to use Hangfire. This is required so that it can ensure that tasks are completed. If your tasks are very quick, then this overhead can outweigh the benefits. You can reduce the latency using **message queues** or the in-memory store **Redis**, but these are advanced topics.

As this book focuses on ASP.NET Core and Hangfire does not support it yet, we won't cover its use in more detail. It is also beyond the scope of this chapter in terms of quick and easy solutions.

Select N+1 problems

You may have heard of Select N+1 problems before. It's a name for a class of performance problems that relate to inefficient querying of a DB. The pathological case is where you query one table for a list of items and then query another table to get the details for each item, one at a time. This is where the name comes from. Instead of the single query required, you perform N queries (one for the details of each item) and the one query to get the list to begin with. Perhaps a better name would be Select 1+N.

You will hopefully not write such bad-performing queries by hand, but an O/RM can easily output very inefficient SQL if used incorrectly. You might also use some sort of business objects abstraction framework, where each object lazily loads itself from the DB. This can become a performance nightmare if you want to put a lot of these objects in a list or calculate some dashboard metrics from a large set.

Note

We will go into detail about SQL and O/RM optimization in Chapter 5, *Optimizing I/O Performance*. This chapter will simply offer some quick fixes to common problems.

If you have a slow application that has performance issues when retrieving data, then Select N+1 may be the problem. Run a SQL profiler tool, as described in the previous chapter, to discover if this is the case. If you see lots of SQL queries for your data instead of just one, then you can move on to the solution stage. For example, if your screen fills with queries on a page load, then you know you have a problem.

In the following example, we will use the micro-O/RM Dapper (made by the team at Stack Overflow) to better illustrate what occurs. However, you are more likely to encounter these problems when using a large lazy loading library or O/RM (such as EF or NHibernate).

Note

Entity Framework Core 1.0 (previously EF 7) does not support lazy loading, so you are unlikely to encounter Select N+1 problems when using it. Previous versions of EF do support this and it may be added to EF Core in the future.

You currently need to use a beta pre-release version of Dapper to work with ASP.NET Core. As with Glimpse, this may be stable by the time you read this (check

). To use Glimpse with Dapper, you need to use the `Glimpse.ADO` package, which unfortunately does not yet support .NET Core.

Consider a simple blog website. On the home page, we would like a list of the posts along with the number of comments each post has. Our blog post model may look something like the following:

```
namespace SelectNPlusOne.Models
{
    public class BlogPost
    {
        public int BlogPostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }
        public int CommentCount { get; set; }
    }
}
```

We also have a model for a comment, which may look like this.

```
namespace SelectNPlusOne.Models
{
    public class BlogPostComment
    {
        public int BlogPostCommentId { get; set; }
        public string CommenterName { get; set; }
        public string Content { get; set; }
    }
}
```

Note

As this is an example, we kept things simple and only used a single set of models. In a real application, you will typically have separate **view models** and **data access layer models**. The controller will map between these, perhaps assisted by a library, such as AutoMapper (automapper.org).

Our view to render this into HTML may look something like the following:

```
@model IEnumerable<SelectNPlusOne.Models.BlogPost>
<table class="table">
    <tr>
        <th>Title</th>
        <th># Comments</th>
    </tr>
    @foreach (var post in Model)
    {
        <tr>
            <td>@post.Title</td>
```

```
            <td>@post.CommentCount</td>
        </tr>
    }
</table>
```

We want to populate these models and view from our database. We have two tables, which look like this:
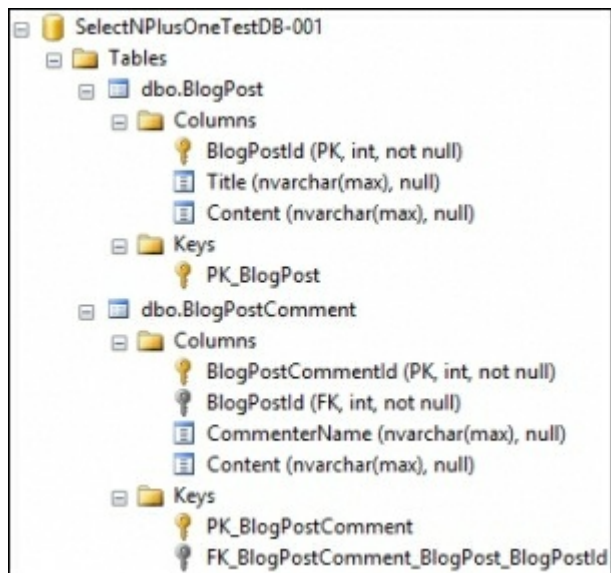
The relationship between the two tables in question looks like the following:

In our controller, we can write code, such as the following to query the database,

populate the model from the database results, and return the view to render it:

```
using (var connection = new SqlConnection(connectionString))
{
    await connection.OpenAsync();
    var blogPosts = await connection.QueryAsync<BlogPost>(@"
        SELECT * FROM BlogPost");
    foreach (var post in blogPosts)
    {
        var comments = await
            connection.QueryAsync<BlogPostComment>(@"
            SELECT * FROM BlogPostComment
            WHERE BlogPostId = @BlogPostId",
            new { BlogPostId = post.BlogPostId });
        post.CommentCount = comments.Count();
    }
    return View(blogPosts);
}
```

We test this and it works! We feel pretty happy with ourselves. It completes quickly on our local test database that contains a handful of rows. We used the async methods everywhere, which must be what makes this so quick. We even only get the comments for each blog in question, not all comments everytime. We also used a parameterized query to avoid **SQL injection**, and everything looks good. Ship it!

Note

As this is an example, we kept it simple for clarity. In a real application, you will want to use techniques such as dependency injection (such as the DI built into ASP.NET Core) to make it more flexible.

Unfortunately, when the data starts to increase (as posts and comments are added), the blog starts to get much slower with pages taking a longer time to load. Our readers get bored waiting and give up. Audience figures drop along with revenue.

Let's profile the database to see what the problem might be. We run SQL Server Profiler filtering on the database in question, and look at the SQL being executed.

The following screenshot shows the filter dialog in SQL Server Profiler:
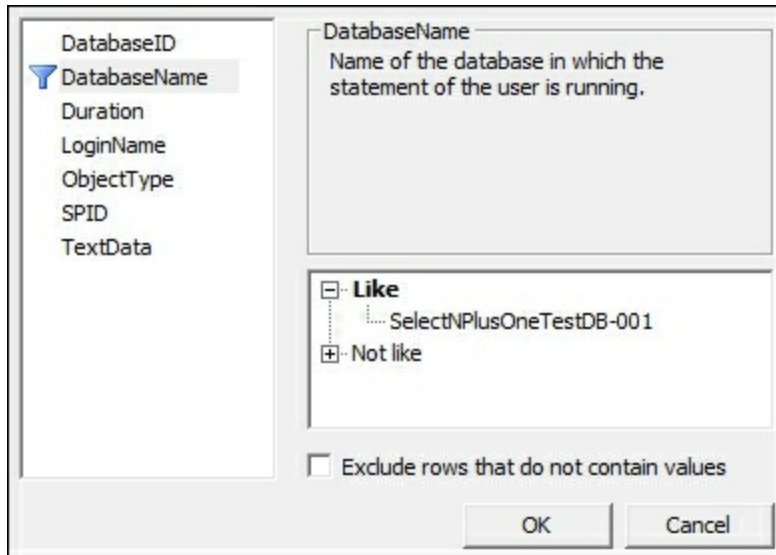
The trace that we capture reveals that lots of queries are being executed; far too many for the data that we need. The problem is that our code is not very efficient because it uses multiple simple queries rather than one slightly more complicated one.

Our code first gets a list of blog posts and then gets the comments for each post, one post at a time. We also bring back way more data than we need. `Async` does not speed up an individual request because we still need all of the data before we can render the page.

Note

The bad coding is obvious in this example because Dapper has the SQL right in your code. However, if you use another O/RM, then you wouldn't typically see the SQL in Visual Studio (or your editor of choice). This is an additional benefit of using Dapper because you see the SQL where it's used, so there are no surprises.

However, the main benefit of Dapper is that it's fast, very fast, and much faster than EF. It's a great choice for performance and you can read more about it at github.com/StackExchange/dapper-dot-net.

We only want a count of the comments for each post and we can get everything that we need (and only what we need) in one query. Let's alter our previous code to use a slightly more complicated SQL query rather than two simpler queries, one of which was inside a `foreach` loop:

Tip
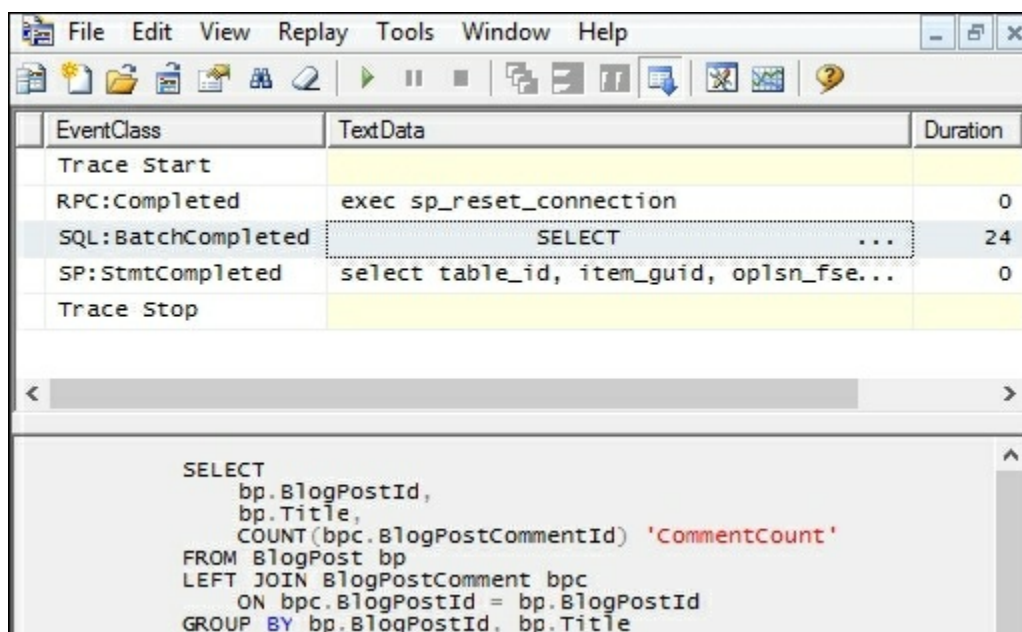
A SQL query inside a loop is an obvious **code smell** that indicates things may not be

as well thought-out as they can be.

```
using (var connection = new SqlConnection(connectionString))
{
    await connection.OpenAsync();
    var blogPosts = await connection.QueryAsync<BlogPost>(@"
        SELECT
            bp.BlogPostId,
            bp.Title,
            COUNT(bpc.BlogPostCommentId) 'CommentCount'
        FROM BlogPost bp
        LEFT JOIN BlogPostComment bpc
            ON bpc.BlogPostId = bp.BlogPostId
        GROUP BY bp.BlogPostId, bp.Title");
    return View(blogPosts);
}
```

This more efficient code only performs a single query against the database and gets all of the information that we need. We join the comments table to the posts in the database and then aggregate by grouping. We only request the columns that we need and add the count of the comments to our selection.

Let's profile the new code to see whether we fixed the problem. The following image shows that we now only have a single query being executed rather than the thousands being executed before:

The number of queries has been dramatically reduced. Therefore, the page loads much faster. However, the page is still very big because all the blog posts are listed

on it and there are a lot. This slows down rendering and increases the time to deliver the page to a browser.

Efficient paging

In a real application, you want to implement paging so that your list is not too long when a lot of data is in the table. It's a bad idea to list thousands of items on a single page.

You may want to do this with LINQ commands because they are very convenient. However, you need to be careful. If your O/RM is not LINQ aware or you accidentally cast to the wrong type a little too early, then the filtering may occur inside the application when the best place to perform this filtering is actually in the database. Your code may be retrieving all of the data and throwing most of it away without you realizing it.

Perhaps you are tempted to modify the action method `return` statement to look something like the following:

```
return View(blogPosts.OrderByDescending(bp => bp.CommentCount)
                     .Skip(pageSize * (pageNumber - 1))
                     .Take(pageSize));
```

This works and will speed up your application considerably. However, it may not have the effect that you think it has. The application is quicker because the view rendering is speedier due to generating a smaller page. This also reduces the time to send the page to the browser and for the browser to render the HTML.

Yet, the application still gets all of the blog posts from the database and loads them into memory. This can become a problem as the amount of data grows. If you want to use LINQ methods such as this, then you need to check that they are handled all the way to the database. It's a very good idea to read the documentation for your O/RM or framework and double-check the SQL that is generated using a profiler.

Let's have a look at what the SQL should look like. For example, if you use SQL Server, starting with the preceding query, you can take only the top ten most-commented posts by altering it like the following:

```
SELECT TOP 10
    bp.BlogPostId,
    bp.Title,
    COUNT(bpc.BlogPostCommentId) 'CommentCount'
FROM BlogPost bp
LEFT JOIN BlogPostComment bpc
    ON bpc.BlogPostId = bp.BlogPostId
GROUP BY bp.BlogPostId, bp.Title
ORDER BY COUNT(bpc.BlogPostCommentId) DESC
```

We order by comment count in descending order. However, you can sort by descending ID to get a rough reverse chronological order if you like. From this ordered set, we select (or take) only the top ten records.

If you want to skip records for paging, the `SELECT TOP` clause is not good enough. In SQL Server 2012 and onward, you can use the following instead:

```
SELECT
    bp.BlogPostId,
    bp.Title,
    COUNT(bpc.BlogPostCommentId) 'CommentCount'
FROM BlogPost bp
LEFT JOIN BlogPostComment bpc
    ON bpc.BlogPostId = bp.BlogPostId
GROUP BY bp.BlogPostId, bp.Title
ORDER BY COUNT(bpc.BlogPostCommentId) DESC
OFFSET 0 ROWS
FETCH NEXT 10 ROWS ONLY
```

You can adjust the value for `OFFSET` to get the correct entries for your page number. The `FETCH NEXT` value will change the page size (the number of entries on a page). You can pass these values in with a parameterized query, as follows:

```
using (var connection = new SqlConnection(connectionString))
{
    await connection.OpenAsync();
    var blogPosts = await connection.QueryAsync<BlogPost>(@"
        SELECT
            bp.BlogPostId,
            bp.Title,
            COUNT(bpc.BlogPostCommentId) 'CommentCount'
        FROM BlogPost bp
        LEFT JOIN BlogPostComment bpc
            ON bpc.BlogPostId = bp.BlogPostId
        GROUP BY bp.BlogPostId, bp.Title
        ORDER BY COUNT(bpc.BlogPostCommentId) DESC
        OFFSET @OffsetRows ROWS
        FETCH NEXT @LimitRows ROWS ONLY", new
        {
            OffsetRows = pageSize * (pageNumber - 1),
            LimitRows  = pageSize
        }
    );
    return View(blogPosts);
}
```

You can pass in the page size and number as URL parameters if you update your action method signature to the following:

```
public async Task<IActionResult> Index(int pageNumber = 1,
```
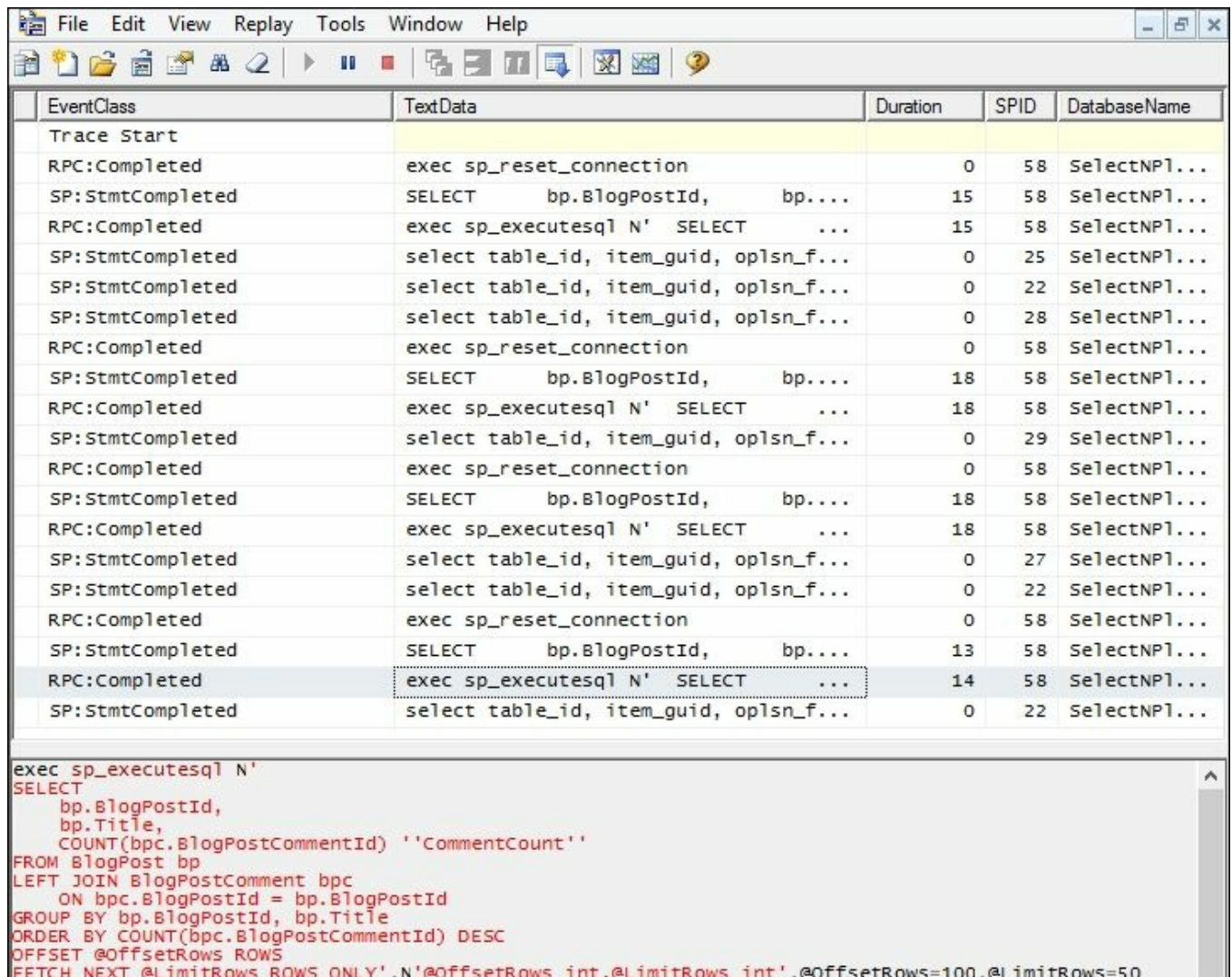
```
                                                         int pageSize  = 10)
```

Here, we provided default values for both parameters, so they are optional. When no parameters are provided, then the first page of ten results is shown. We need to multiply the page size by the zero-indexed page number to calculate the correct offset. It should be zero for the first page so that no records are skipped.

Tip

It would be a very good idea to apply some validation to the paging parameters. Don't allow users to set them to anything outside of a reasonable range. This is left as an exercise to the reader.

If we look in the profiler at the queries being executed on the database server, then we can see what SQL is now being run. We can also see the time taken and compare this to our results from previously:

EventClass	TextData	Duration	SPID	DatabaseName
Trace Start				
RPC:Completed	exec sp_reset_connection	0	58	SelectNP1...
SP:StmtCompleted	SELECT bp.BlogPostId, bp....	15	58	SelectNP1...
RPC:Completed	exec sp_executesql N' SELECT ...	15	58	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	25	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	22	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	28	SelectNP1...
RPC:Completed	exec sp_reset_connection	0	58	SelectNP1...
SP:StmtCompleted	SELECT bp.BlogPostId, bp....	18	58	SelectNP1...
RPC:Completed	exec sp_executesql N' SELECT ...	18	58	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	29	SelectNP1...
RPC:Completed	exec sp_reset_connection	0	58	SelectNP1...
SP:StmtCompleted	SELECT bp.BlogPostId, bp....	18	58	SelectNP1...
RPC:Completed	exec sp_executesql N' SELECT ...	18	58	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	27	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	22	SelectNP1...
RPC:Completed	exec sp_reset_connection	0	58	SelectNP1...
SP:StmtCompleted	SELECT bp.BlogPostId, bp....	13	58	SelectNP1...
RPC:Completed	exec sp_executesql N' SELECT ...	14	58	SelectNP1...
SP:StmtCompleted	select table_id, item_guid, oplsn_f...	0	22	SelectNP1...

```
exec sp_executesql N'
SELECT
    bp.BlogPostId,
    bp.Title,
    COUNT(bpc.BlogPostCommentId) ''CommentCount''
FROM BlogPost bp
LEFT JOIN BlogPostComment bpc
    ON bpc.BlogPostId = bp.BlogPostId
GROUP BY bp.BlogPostId, bp.Title
ORDER BY COUNT(bpc.BlogPostCommentId) DESC
OFFSET @OffsetRows ROWS
FETCH NEXT @LimitRows ROWS ONLY',N'@OffsetRows int,@LimitRows int',@OffsetRows=100,@LimitRows=50
```

The query in the screenshot gets the data for the third page with the page size set to `50` entries. Therefore, it used an offset of `100` (to skip the first two pages of 50) and fetched the next 50 rows. The URL query string for this can look something like the following:
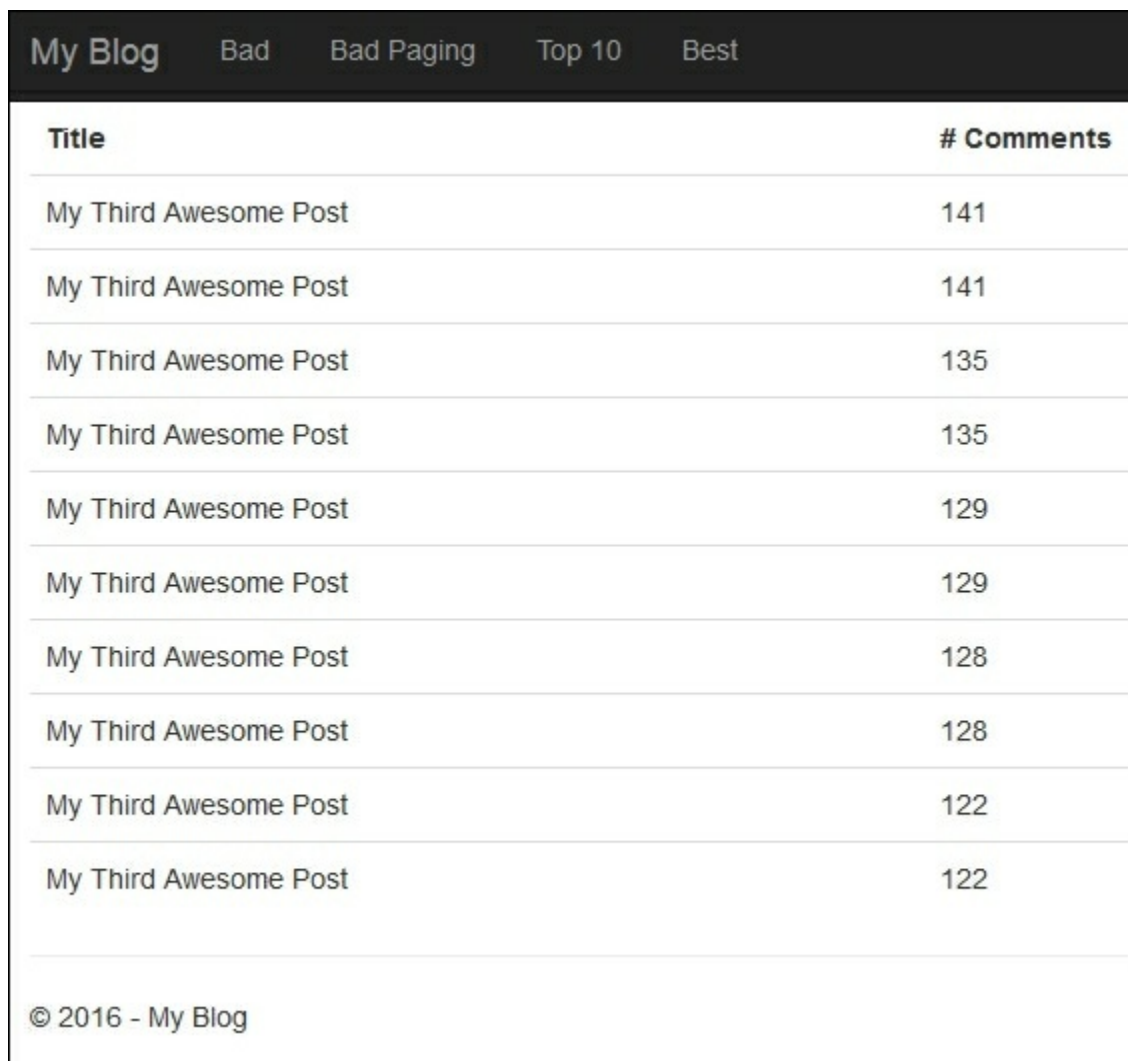
```
/?pagenumber=3&pagesize=50
```

We can see that the duration of the query has decreased from **24** ms previously to **14** ms now.

Tip

Note how the SQL executes differently when parameters are passed into the query. This is much safer than concatenating user-supplied values directly into a SQL command.

If you do not use any parameters, then the default values are used and the home page shows only ten entries, which looks something like the following screenshot, depending on the data in the database:

My Blog Bad Bad Paging Top 10 Best	
Title	**# Comments**
My Third Awesome Post	141
My Third Awesome Post	141
My Third Awesome Post	135
My Third Awesome Post	135
My Third Awesome Post	129
My Third Awesome Post	129
My Third Awesome Post	128
My Third Awesome Post	128
My Third Awesome Post	122
My Third Awesome Post	122
© 2016 - My Blog	

By default, the home page only displays the top 10 most commented posts, but you can easily add page navigation with hyperlinks. Simply add the `pagenumber` and `pagesize` query string parameters to the URL.

You can use the example URL query string shown previously on either the home page or the bad paging path, for example, `/Home/BadPaging/?pagenumber=3&pagesize=50`.

The links in the navigation bar load the examples that we just walked through. The best is the same as the home page and is the default. Top 10 and bad paging should be fairly self-explanatory. Bad will take a long time to load, especially if you use the DB creation script included with the project. You can time it with your browser developer tools.

For previous versions of SQL Server (prior to 2012), there are paging workarounds using `ROW_NUMBER()` or nested `SELECT` statements, which invert the sort order. If you use another database, such as PostgreSQL, MySQL, or SQLite, then you can easily implement paging with the `LIMIT` clause. These free databases often have more features than SQL Server, such as the `Limit` clause mentioned here and a **concatenation aggregator** to name just one other.

Note

One of the touted benefits of big O/RMs is the layer of abstraction that they offer. This allows you to change the database that you use. However, in practice, it is rare to change something as core as a database. As you can see from the simple paging example, syntax varies between databases for anything other than simple standard SQL. To get the best performance, you really need to understand the features and custom syntax of the database that you use.

Static site generators

The database is the logical place to perform any work to filter and sort data. Doing this in the application is a waste. However, for a simple blog that is updated infrequently, a database may be unnecessary in the first place. It may even become the bottleneck and slow the whole blog down. This is a typical problem of blog engines, such as **WordPress**. A better approach may be to use a static site generator.

A static site generator prerenders all of the pages and saves the resulting HTML. This can easily be served by a simple web server and scales well. When a change is made and pages need updating, then this site is regenerated and a new version is deployed. This approach doesn't include dynamic features, such as comments, but third-party services are available to provide these added extras.

A popular static site generator is **Jekyll**, which is written in **Ruby**. **GitHub** provides a free static site-hosting service called **GitHub Pages**, which supports Jekyll, and you can read more about it at pages.github.com . Another static site generator (written in **Go**) is **Hugo**, which you can read about at gohugo. io . These tools are basically a form of extreme caching. We'll cover caching in the next section and later on in this book.

It's often worth taking a step back to see whether the problem that you're trying to solve is even a problem. You may well improve database performance by removing the database.

Pragmatic solutions with hardware

The best approach to take with a poorly performing application is usually to fix the software. However, it is good to be pragmatic and try to look at the bigger picture. Depending on the size and scale of an application, it can be cheaper to throw better hardware at it, at least as a short term measure.

Hardware is much cheaper than developer time and is always getting better. Installing some new hardware can work as a quick fix and buy you some time. You can then address the root causes of any performance issues in software as part of the ongoing development. You can add a little time to the schedule to **refactor** and improve an area of the code base as you work on it.

Once you discover that the cause of your performance problem is latency, you have two possible approaches:

- Reduce the number of latency-sensitive operations
- Reduce the latency itself using faster computers or by moving the computers closer together

With the rise of cloud computing, you may not need to buy or install new hardware. You can just pay more for a higher-performing instance class or you can move things around inside your cloud provider's infrastructure to reduce latency.

A desktop example

To borrow an example from native desktop applications, it is quite common to have poorly-performing **Line of Business (LoB)** applications on corporate desktops. The desktop will probably be old and underpowered. The networking back to the central database may be slow because the connection might be over a remote link to a regional office.

With a badly-written application that is too chatty with the DB, it can be better, performance-wise, to run the application on a server close to (or on the same server as) the DB. Perhaps the application workspace and DB servers can be in the same **server rack** at the **data center** and connected by Gigabit (or ten Gigabit) **Ethernet**.

The user can then use a remote desktop connection or **Citrix** session to interact with the application. This will reduce the latency to the DB and can speed things up, even taking into consideration the lag of the remote UI. This effectively turns the desktop PC into a **thin client**, similar to how old mainframes are used.

For example, you can build a high-performance server with RAID SSDs and lots of RAM for much less than the cost of the developer time to fix a large application. Even badly-written software can perform well if you run the application and DB together on the same machine, especially if you run it on **bare metal** with no virtual machines in the way. This tactic would buy you time to fix things properly.

These remote application and virtualization technologies are usually sold as tools to aid deployment and maintenance. However, the potential performance benefits are also worth considering.

Due to the rise of web applications, **thick client** desktop applications are now less common. Architecture seems to oscillate between computation on the server and doing work on the client, as the relative progress of networking speed and processing power race each other.

Web applications

The same relocation approach does not typically work as well for web applications, but it depends on the architecture used. The good news is that, for web applications, you usually control the infrastructure. This is not normally the case for native application hardware.

If you use a **three-tier architecture**, then you can move the application servers closer to the DB server. Whether this is effective or not depends on how chatty the web servers are with the application servers. If they issue too many web API requests, then this won't work well.

A **two-tier architecture** (where the web servers talk directly to the database) is more common for typical web applications. There are solutions using **clustered** databases or read-only mirrors to place the data close to the web servers, but these add complexity and cost.

What can make a significant difference are **proxy servers**. A popular open source proxy server is Varnish and you can also use the NGINX web server as a proxy. Proxy servers cache the output of web servers so that a page doesn't have to be regenerated for each user. This is useful for shared pages but caching is hard; typically, you should not cache personalized pages. You don't want to accidentally serve someone's authenticated private information to another user.

Proxies such as Varnish can also route different parts of your website to different servers. If you have a small area of your site that performs badly due to DB chatter, then you could host that part from web servers on (or very close to, such as on the same VM host) the DB machines and route requests for it to there. The rest of the site could remain on the existing web server farm.

This isn't a long term solution, but it allows you to split off a poorly performing part of your program so that it doesn't impact the rest of your system. You're then free to fix it once it's been decoupled or isolated. You can even split off the data required to a separate DB and synchronize it with a background process.

There are also **Content Delivery Networks (CDNs)**, such as **CloudFlare**, **Amazon CloudFront**, and **Azure CDN**, which are good to cache static assets. CDNs cache parts of your site in data centers close to your users, reducing the latency. CloudFlare can even add HTTPS to your site for free, including automatically issuing certificates.

Note

You can read more about the CDN offerings of CloudFlare (including HTTP/2 server push and WebSockets on the free plan) at www.cloudflare.com .

We will cover caching in more detail in [Chapter 7](), *Learning Caching and Message Queuing*, so we won't go into more detail here. Caching is a challenging subject and needs to be understood well so that you can use it effectively.

Oversized images

While we're on the subject of static assets, we should briefly mention image optimization. We'll cover this in much more detail in the next chapter, but it's worth highlighting some common problems here. As you have very little control over network conditions between your infrastructure and the user, low throughput may be a problem in addition to high latency.

Web applications heavily use images, especially on landing pages or home pages, where they might form a fullscreen background. It is regrettably common to see a raw photo from a camera simply dropped straight in. Images from cameras are typically many megabytes in size, far too big for a web page.

You can test whether there are problems on a web page using a tool, such as Google's PageSpeed Insights. Visit developers.goog le.com/speed/pagespeed/insights , enter a URL, and click on ANALYZE to view the results. Google use this information as part of their search engine ranking, so you would do well to take its advice up to a point. Slow sites rank lower in search results.

You can also use the browser developer tools to view the size of images. Press *F12* and look at the network tab after a page load to view how much data was transferred and how long it took. You can often miss these performance issues on a local machine or test server because the image will load quickly. After the first load, it will also be stored in the browser's cache, so make sure you do a full hard reload and empty or disable the cache. In Chrome (when the dev tools are open), you can right-click or long click on the reload button for these extra options. It's also a good idea to use the built-in throttling tools to see how a user will experience the page loading.

The most basic image optimization problems typically fall into two categories:

- Images that are overly large for the display area they are displayed in
- Images that use the wrong compression format for the subject matter

Image resolution

The most common issue is that an image has too high a resolution for the area that displays it. This forces the browser to resize the image to fit the screen or area. If the size of the file is unnecessarily large, then it will take longer to be transferred over an internet connection. The browser will then throw away most of the information. You should resize the image ahead of time before adding it to the site.

There are many image manipulation tools available to resize pictures. If you run Windows, then **Paint.NET** (www.getpaint.net) is an excellent free piece of software. This is much better than the Paint program that comes with Windows (although this will work if you have no other option).

For other platforms, GIMP (www.gimp.org) is very good. If you prefer using the command line, then you may like ImageMagick (imagemagick.org), which can perform lots of image manipulation tasks programmatically or in batches. There are also cloud-hosted image management services, such as **Cloudinary** (cloudinary.com).

You should shrink images to the actual size that they will be displayed on the user's screen. There can be complications when dealing with responsive images, which scale with the size of the user's screen or browser window. Also keep in mind **high DPI** or Retina displays, which may have more than one physical pixel to every logical pixel. Your images may have to be bigger to not look blurry, but the upper bound is still likely to be lower than the raw size. It is rare to need an image at more than twice the size of the displayed resolution. We will discuss responsive images in more detail later in this book, but it is worth keeping them in mind.

The following image displays the resizing dialog from Paint.NET:
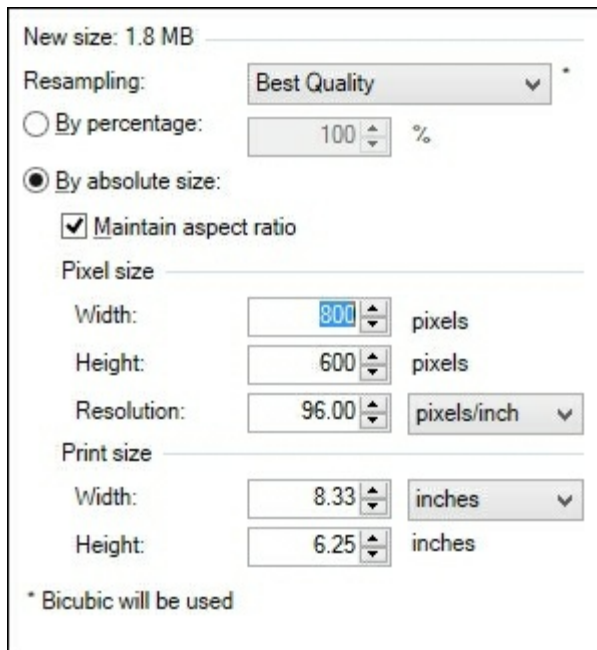
When resizing an image, it is usually important to keep the **aspect ratio** of the image the same. This means changing the horizontal and vertical sizes in proportion to each other.

For example, reducing the **Height** from **600** px to **300** px and reducing the **Width** from **800** px to **400** px (meaning both dimensions are reduced by 50%) keeps the image looking the same, only smaller. Most image-manipulation software will assist with this process. Keeping the aspect ratio the same will avoid images looking stretched. If images need to fit a different shape, then they should be cropped instead.

Image format

The next most common problem is using images in the wrong file format for the content. You should never use raw uncompressed images, such as **bitmap** (**BMP**), on the web.

For natural images such as photos, use the JPEG file format. JPEG is a lossy codec, which means that information is lost when using it. It is very good for pictures with a lot of gradients in them, such as images of natural scenes or people. JPEG looks poor if there is any text in the image because there will be **compression artifacts** around the edges of the letters. Most mid- and low-end cameras natively save images as JPEG, so you do not lose anything by staying with it. However, you should resize the images to make them smaller, as mentioned previously.

For artificial images such as diagrams or icons, use **PNG**. PNG is a lossless codec, which means that no information is discarded. This works best for images with large blocks of solid color, such as diagrams drawn in painting software or screenshots. This also supports transparency, so you can have images that don't appear rectangular or are translucent. You can also have animated PNGs, which are of superior quality to **GIFs**, but we won't go into the details of them in this chapter.

You can alter the format of images using the same tools that you use to resize them, as mentioned previously, by simply changing the file format when saving the image. As always, you should test for what works best in your specific use case. You can perform experiments by saving the same image in different formats (and different resolutions) then observing the sizes of the files on disk.

The following image displays the available image options in Paint.NET. Additional formats are available and we will go into more detail in Chapter 4, *Addressing Network Performance*:
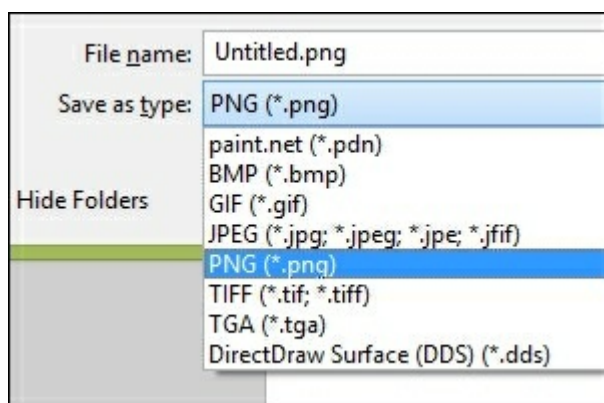
Even when you only choose between **JPEG** and **PNG**, you can still make a significant difference. The following screenshot displays the difference in the size of the file of the same image in two resolutions and two formats.

The following test image is the one used in the experiment. Due to the hard edges it looks best as a PNG, but the gradient background makes it more difficult to compress:

Note

The test images used here are available for download with this book, so you can try the experiment for yourself.

In a web context, this particular image may be best served with a transparent background using CSS for the gradient. However, simple shapes, such as these can better be represented as **Scalable Vector Graphics** (**SVG**) or with a HTML5 canvas.

Summary

In this chapter, you learned about some common performance problems and how to fix them. We covered asynchronous operations, Select N+1 problems, pragmatic hardware choices, and overly-large images.

In the next chapter, we will expand on image optimization and extend this to other forms of compression for different types of resources. We'll look at the new process for the bundling and minification of static assets in ASP.NET Core using open source tools.

Additionally, we will introduce networking topics, such as TCP/IP, HTTP, WebSockets, and encryption. We'll also cover caching, including another look at CDNs.

Chapter 4. Addressing Network Performance

This chapter builds on a subset of the problems that were discussed in the previous chapter but in more detail. It deals with latency, or lag, which originates at the networking level between the user and the application. This is mostly applicable to web applications where the user interacts with the application via a web browser. You will learn how to optimize your application to cater for bandwidth and latency that is unknown and outside of your control. You'll compress your payloads to be as small as possible, and then you will deliver them to the user as quickly as possible. You will learn about the tools and techniques that can be used to achieve a fast and responsive application. You'll also see the trade-offs involved and be able to calculate whether these methods should be applied to your software.

The topics that we will cover in this chapter include the following:

- TCP/IP
- HTTP and HTTP/2
- HTTPS (TLS/SSL)
- WebSockets and push notifications
- Compression
- Bundling and minification
- Caching and CDNs

This chapter deals with how to speed up the experience for a user using your application. The skills in this chapter are just as applicable to a static site or client-side web app as they are to a dynamic web application.

These topics apply to any web application framework. However, we will focus on how they are implemented with ASP.NET—in particular with the new ASP.NET Core implementation, and how this differs from the existing full ASP.NET.

Internet protocols

It's important to know about how your HTML and other assets are delivered from the web server to your user's browser. Much of this is abstracted away and transparent to web development, but it's a good idea to have at least a basic understanding in order to achieve high performance.

TCP/IP

Transmission Control Protocol / Internet Protocol (**TCP/IP**) is the name for a pair of communication protocols that underpin the internet. **IP** is the lower-level protocol of the two, and this deals with routing *packets* to their correct destinations. IP can run on top of many different lower-level protocols (such as Ethernet), and this is where IP addresses come from.

TCP is a layer above IP, and it is concerned with the reliable delivery of packets and *flow control*. TCP is where ports come from, such as port 80 for HTTP, and port 443 for HTTPS. There is also the **User Datagram Protocol** (**UDP**), which can be used instead of TCP, but it provides fewer features.

HTTP runs on top of TCP, and it is usually what you will deal with in a web application. You may occasionally need to directly use **Simple Mail Transfer Protocol** (**SMTP**) to send e-mails, the **Domain Name System** (**DNS**) to resolve hostnames to IP addresses, or **File Transfer Protocol** (**FTP**) to upload and download files.

The basic unencrypted versions of these protocols run directly on TCP, but the secure encrypted versions (HTTPS, SMTPS, and FTPS) have a layer in between them. This layer is called **Transport Layer Security** (**TLS**), and this is the modern successor to the **Secure Sockets Layer** (**SSL**). SSL is insecure and deprecated, and it should no longer be used. However, the term SSL is still commonly and confusingly used to describe TLS. All browsers require the use of TLS encryption to support HTTP/2.

You may not often think about the lower-level protocols when you build web applications. Indeed, you may not need to consider even HTTP/HTTPS that much. However, the protocol stack below your application can have significant performance implications.

The following diagram shows how the protocols are typically stacked:

```
┌─────────────────────────────────────────────┐
│                                             │
│              HTTP or HTTP/2                 │
│                                             │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│                                             │
│                 TLS (SSL)                   │
│                                             │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│                                             │
│                   TCP                       │
│                                             │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│                                             │
│                    IP                       │
│                                             │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│                                             │
│     Link Layer and physical (e.g. Ethernet) │
│                                             │
└─────────────────────────────────────────────┘
```
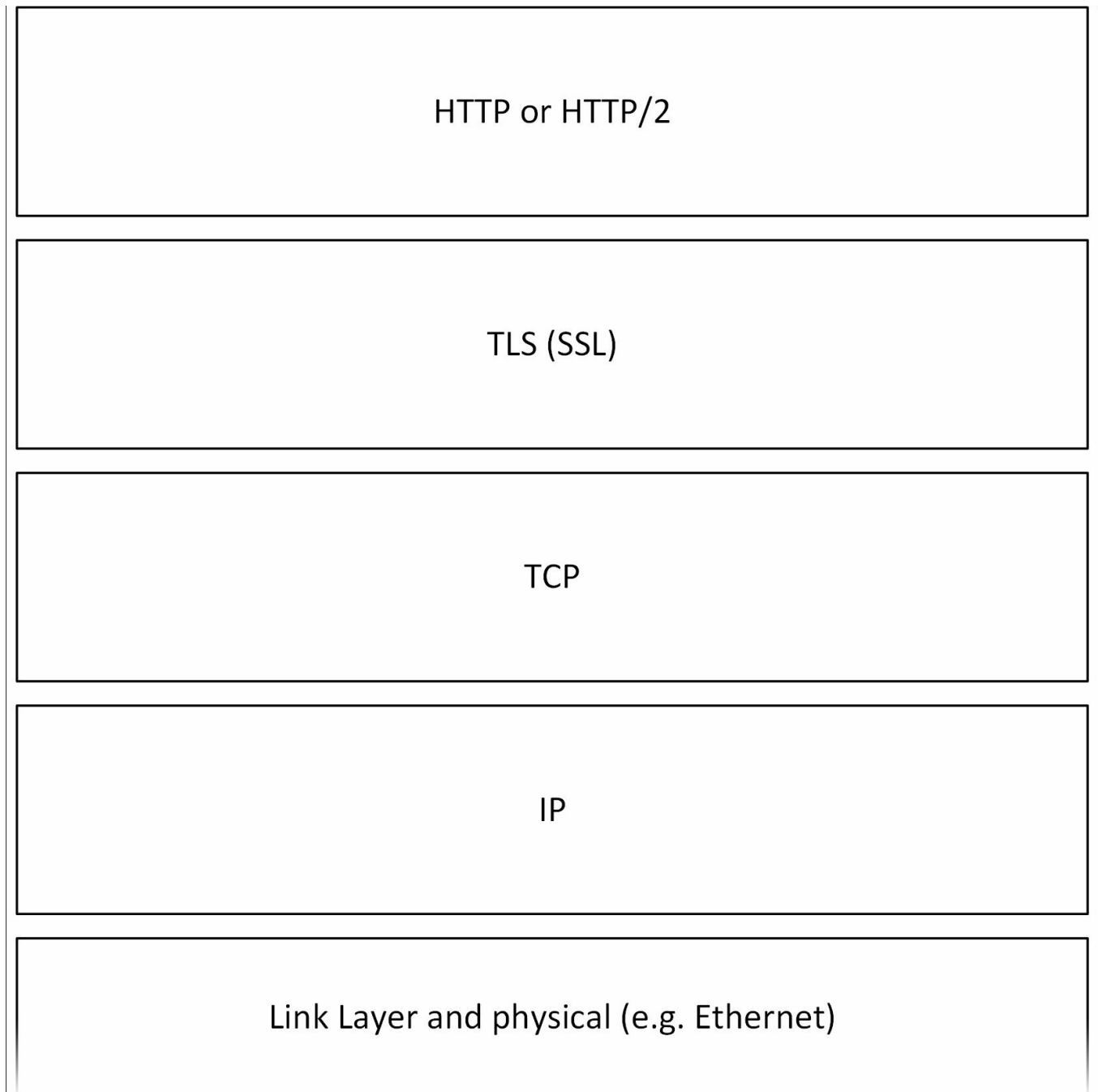
Slow-start

TCP implements an algorithm called *slow-start* for congestion-control purposes. This means that the connection from a browser to a web server is initially slow and ramps up over time to discover the available bandwidth. You can alter the settings for this so that the ramp up is more aggressive, and connections get quicker faster. If you increase the initial *congestion window*, then performance can improve, especially on connections with good bandwidth but high latency, such as mobile **4G** or servers on other continents.

As usual, you should test for your use case perhaps using Wireshark, as described previously in [Chapter 2](#), *Measuring Performance Bottlenecks* . There are downsides to altering this window, and it should be considered carefully. Although this may speed up websites, it can cause buffers in networking equipment to fill, which can generate latency problems for VoIP applications and games if no **Quality of Service (QoS)** is in use end to end.

You can change this value on Windows Server 2008 R2 with a hotfix (KB2472264) and higher. You can also easily adjust this on Linux, and ASP.NET Core enables you to run your .NET web app on Linux (and Mac OS X) in addition to Windows.

We won't provide detailed instructions here because this should be a cautiously considered decision, and you shouldn't apply advice blindly. You can easily find instructions online for the operating system that you use on your web server.

TCP slow-start is just one example of why you can't ignore the lower levels of Internet technology on the shoulders of which web applications stand. Let's move up the stack a little to the application layer.

HTTP

As a web application developer who wants to deliver high performance, it's important to understand Hypertext Transfer Protocol. You should know what version of HTTP you use, how it works, and how this affects things, such as **request pipelining** and encryption.

HTTP/1.1 is the version that you will probably be most familiar with today because it has been in use for some time. HTTP/2 is becoming more popular, and it changes the best way to do many things.

Headers

HTTP uses headers to provide metadata about a request along with the main payload in the body of the message, much like e-mails do. You won't see these headers when you view the source, but you can observe them using the browser developer tools. You can use headers for many things, such as cache control and authentication. Cookies are also sent and received as headers.

Browsers will only open a limited number of HTTP/1.1 connections at one time to a single host. If you require a lot of requests to retrieve all the assets for a page, then they are queued, which increases the total time taken to fully load it. When combined with the TCP slow-start mentioned previously, this effect can be amplified, degrading the performance. This is less of a problem with HTTP/2, which we will cover shortly. You can reduce the impact of this problem by allowing the browser to reuse connections. You can do this by ensuring that your web server doesn't send a `Connection: close` header with HTTP responses.

HTTP methods

There are multiple methods (or verbs) that HTTP uses. The most common are `GET` and `POST`, but there are many more. Typically, we use `GET` requests to retrieve data from a server, and we use `POST` to submit data and make changes. `GET` should not be used to alter data.

Other useful verbs are `HEAD` and `OPTIONS`. `HEAD` can check the headers for a `GET` request without the overhead of downloading the body. This is useful to check caching headers to see whether the resource has changed. `OPTIONS` is commonly used for **Cross Origin Resource Sharing (CORS)** to perform a preflight check to validate a domain.

Other often used verbs are `PUT`, `DELETE`, and `PATCH`. We mainly use these for **Representational State Transfer (REST)** APIs because they can mimic operations

on resources or files. However, not all software (such as some proxy servers) understands them, so sometimes, we emulate them using `POST`. You may even have problems with `OPTIONS` being blocked by proxies and web servers.

Status codes

HTTP uses numeric response codes to indicate a status. You are probably familiar with 200 (OK) and 404 (Not Found), but there are many others. For example, 451 indicates that the content has been blocked by a government-mandated censorship filter.

Note

The 451 status code is in reference to the book *Fahrenheit 451* (whose title is the purported temperature at which paper burns). You can read the official document (RFC 7725) at tools.ietf.org/html/rfc7725. If this code is not used, then it can be tricky to discover if and why a site is unavailable. For example, you can find out whether the UK government is blocking your site at blocked.org.uk, but this is just a volunteer effort run by the Open Rights Group the British version of the **Electronic Frontier Foundation** (**EFF**).

We commonly use 3xx codes for redirection (perhaps to HTTPS). There are various forms of redirection with different performance characteristics (and other effects). You can use a 302 to temporarily redirect a page, but then the browser has to request the original page every time to see whether the redirect has ended. It also has bad implications for **Search Engine Optimization** (**SEO**), but we won't discuss these here.

A better approach is to use a 301 to indicate a permanent redirect. However, you need to be careful, as this can't be undone and clients won't look at the original URL again. If you use redirects to upgrade users from HTTP to HTTPS, then you should also consider using **HTTP Strict Transport Security** (**HSTS**) headers. Again, do this carefully.

Encryption

HTTP encryption is very important. It not only secures data in transit to prevent eavesdropping, but it also provides authentication. This ensures that users actually connect to the site that they think they are and that the page wasn't tampered with. Otherwise, unscrupulous internet connection providers can inject or replace adverts on your site, or they can block internet access until you have opted out of a parental filter. Or, these can be worse things, such as stealing your user's data, which you are usually required by law to protect.

There is really no good reason today to not use encryption everywhere. The overheads are tiny, although we will still consider them and show you how to avoid potential issues. Arguments against using HTTPS are usually hangovers from a time long ago when computation was expensive.

Modern computing hardware is very capable and often has special acceleration for common encryption tasks. There are many studies that show that the processing overheads of encryption are negligible. However, there can be a small delay in initially setting up a secure connection for the first time. In order to understand this, it is useful to illustrate a simple model of how TLS works.

There are two parts to secure communication: the initial key exchange and the ongoing encryption of the channel. Session ciphers, such as the **Advanced Encryption Standard** (**AES**), can be very quick, and they can operate at close to line speed. However, these ciphers are *symmetrical* and both parties need to know the key. This key needs to be distributed securely so that only the two communicating parties possess it. This is called *key exchange*, and it uses asymmetric encryption. This usually also requires a third party to vouch for the server, so we have a system of certificates. This initial setup is the slow part, although we will show you an alternative for devices that lack the AES acceleration later.

Key exchange

As mentioned previously, key exchange is the process of securely sharing an encryption key between two parties without being intercepted. There are various methods of doing this, which mostly rely on asymmetric encryption. Unlike symmetric encryption (that we exchange this key for), this can only be performed in one direction with a single key. In other words, the key that is used to encrypt cannot be used to decrypt, and a different key is required. This is not the case for the majority of the data once we have shared a key. The reason that we do this is that symmetric encryption is faster than asymmetric encryption. Therefore, it is not used for everything and is only needed to encrypt another key.

In addition to exchanging a key, the browser (or other HTTPS client) should check the certificate to ensure that the server belongs to the domain that it claims to. Some programmatic clients fail to do this by default, so this is worth checking out. You can also implement certificate pinning (even in the browser with **HTTP Public Key Pinning**) to improve security, but this is beyond the scope of this book.

We will illustrate two variations of key exchange by analogy in simplified forms to show you how the process works. You can look up the technical details if you wish.

RSA

RSA is traditionally the most common key-exchange mechanism that is used for TLS. Until recently, we used it on most HTTPS connections.

RSA stands for Rivest-Shamir-Adleman after the names of its creators, and it is probably the most popular form of public key cryptography. The British snooping agency, **Government Communications Headquarters** (**GCHQ**), supposedly conceived public key cryptography at around the same time, but as it was only made public in 1997, it's impossible to prove this. The invention credit goes to Whitfield Diffie and Martin Hellman, who recently picked up a Turing Award for it. We'll talk more about their work shortly.

Note

The Turing Award is the Nobel Prize of computing. It's named after Alan Turing, the legendary computing pioneer who helped the allies win WWII while working for the nascent GCHQ, but who was later betrayed by the British government.

RSA uses large prime numbers to generate a public and private key pair. The public key can be used to encrypt information that can only be decrypted with the private key. In addition to this, the private key can be used to sign information (usually a **hash** of it), which can be verified with the public key. RSA is often used to sign TLS certificates even if another algorithm is used as the key exchange mechanism (this is negotiated during the initial TLS handshake).

Tip

This hashing and signing of certificates is where you may have heard of **SHA-1** certificates being deprecated by browsers. SHA-1 is no longer considered secure for hashing and, like **MD5** before it, should not be used. Certificate chains must now use at least an **SHA-2** hashing algorithm (such as **SHA-256**) to sign.

An analogy to help explain how RSA works is to think of sending a lock instead of sending a key. You can post an open padlock to someone, retaining the key to it. They can then use your lock to secure a case with a key of theirs inside and send it back to you. Now, only you can open the case to get the new key.

In reality, this is more complicated. You can't be sure that someone didn't intercept your lock and then use their own lock to get the key and copy it before sending it on to you. Typically, we solve this with **Public Key Infrastructure** (**PKI**). A trusted third party will sign your certificate and verify that it is indeed your public key and that you own the lock. Browsers typically display a warning if a **Certificate Authority** (**CA**) does not countersign the certificate in this way.

Diffie-Hellman (**D-H**) key exchange is another method of gaining a shared key. Invented shortly before RSA, it has only recently become popular on the web. This is partly due to the reduced computational cost of the **elliptic curve** variant. However, another reason is that the ephemeral versions provide a quality called **Perfect Forward Secrecy** (**PFS**). Unlike RSA, the session key for the symmetric encryption never needs to be transmitted. Both parties can calculate the shared key without it needing to be sent on the wire (even in an encrypted state) or permanently stored. This means that an eavesdropped encrypted exchange cannot be decrypted in the future if the keys were recovered. With RSA key exchange, you can recover a recorded communication in plain text if you obtain the private keys later. PFS a is useful countermeasure against mass surveillance, where all communication is caught in a dragnet and permanently stored.

D-H is better explained with a color mixing analogy, where paints represent numbers. The two parties choose a shared color, and each party chooses a secret color of their own. Both mix the shared color with their secret color and send the result to the other. Each party then mixes the color that they received with their secret color again. Both parties now have the same color without ever having to send this color anywhere where it could be observed.

As this is not a book about security, we won't go into any more detail on encryption algorithms. If you are interested, there is a huge amount of information available that we can't cover here. Encryption is a large subject, but security is an even broader concern.

TLS handshake

The point of briefly explaining how TLS key exchange works for various methods is to show that it is complex. Many messages need to be sent back and forth to establish a secure connection, and no matter how fast the connection, latency slows down every message. This all occurs in the TLS handshake, where the client (usually a web browser) and server negotiate common capabilities and agree on what ciphers they should use. There is also **Server Name Indication** (**SNI**) to consider, which is similar to the HTTP host header in that it allows multiple sites to use the same IP address. Some older clients don't support SNI.

We can observe the TLS handshake using Wireshark. We won't go into a huge amount of detail, but you can see that at least four messages are exchanged between the client and server. These are client hello, server hello (including the certificate and server key exchange), client key exchange, and cipher agreement. The browser may send more messages if we do not optimally configured things, such as requesting an intermediate certificate. This may also check a revocation list to see whether the

certificate was revoked.

The following screenshot shows a TLS handshake captured with Wireshark:

```
TLSv1     223 Client Hello
TCP      1502 [TCP segment of a reassembled PDU]
TCP      1502 [TCP segment of a reassembled PDU]
TCP        54 2697 → 443 [ACK] Seq=170 Ack=2897 Win=66048 Len=0
TLSv1    1002 Server Hello, Certificate, Server Key Exchange, Server Hello Done
TLSv1     220 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
TLSv1     113 Change Cipher Spec, Encrypted Handshake Message
TLSv1     283 Application Data
```

All these network operations happen quickly. However, if the connection has a high latency, then these extra messages can have an amplified effect on performance. The computational delays are typically much smaller than the network delays, so we can discount these, unless you use very old hardware. Fortunately, there are some simple things you can do that will help speed things up and let you enjoy high performance while still being secure.

Delay diagnostics

There are various mechanisms that are built into TLS that can you can use to speed it up. However, there are also things that will slow it down if you don't do them correctly. Some great free online tools to assess your TLS configuration are available from Qualys SSL Labs at ssllabs.com . The server test at ssllabs.com/ssltest is very useful. You enter a URL, and they give you a grade along with lots of other information.

For example, if we analyze the packtpub.com site, we can see that on the date of the test it got a **B** grade. This is due to supporting weak Diffie-Hellman parameters and the obsolete and insecure **RC4** cipher. However, it is not always as simple as removing old ciphers. You can have a very secure site, but you might exclude some of your customers, who use older clients that don't support the latest standards. This will, of course, vary depending on the nature of your client base, and you should measure your traffic and consider your options carefully.

The following screenshot shows some of the report from SSL Labs for packtpub.com .

If we have a look at a site with a better configuration (emoncms.org), we can see that it gets an **A** grade. You can get an A+ grade using HSTS headers. Additionally, these headers avoid the overhead of a redirect. You may also be able to get your site embedded in a preloaded list shipped with browsers if you submit the domains to the vendors.

The following screenshot shows some of the report from SSL Labs for emoncms.org :

The options chosen by modern browsers would typically be an **Elliptic Curve Diffie-Hellman Ephemeral key exchange** (**ECDHE**) with an RSA SHA-256 signature and AES session cipher. The ephemeral keys provide PFS because they are only held in memory for the session. You can see what connection has been negotiated by looking in your browser.

In Firefox, you can do this by clicking on the lock icon in the URL bar and then clicking on the **More Information** button, as shown in the following image:

In the **Technical Details** section, you will see the cipher suite used. The following image from Firefox shows ECDHE key exchange and RSA certificate signing:

Technical Details
Connection Encrypted (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)

You can also view the certificate details by clicking on the **View Certificate** button. The domain is usually included as the **Common Name (CN)** in the **Subject** field. Alternative domains can also be included under the **Certificate Subject Alt Name** extension:
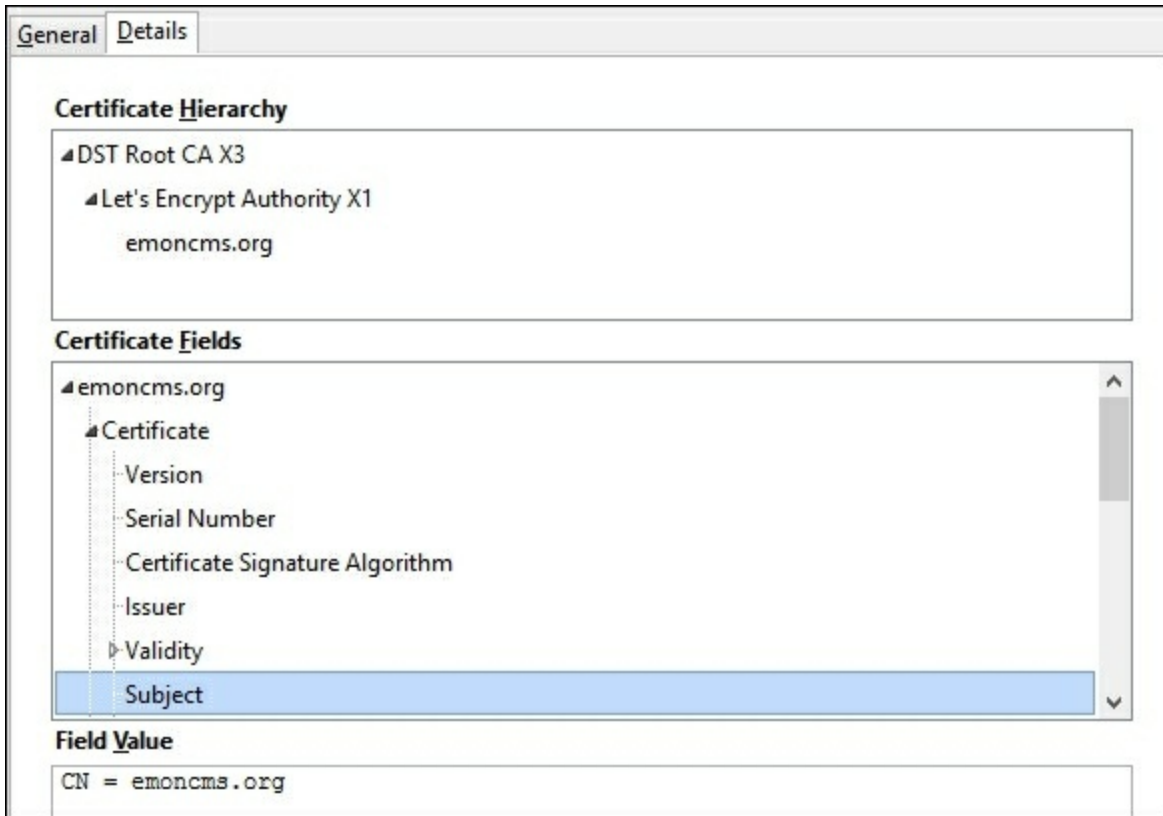
In Chrome, you can look at the TLS connection information in the **Security** tab of the developer tools. For example, the following image displays the security details for huxley.unop.uk :
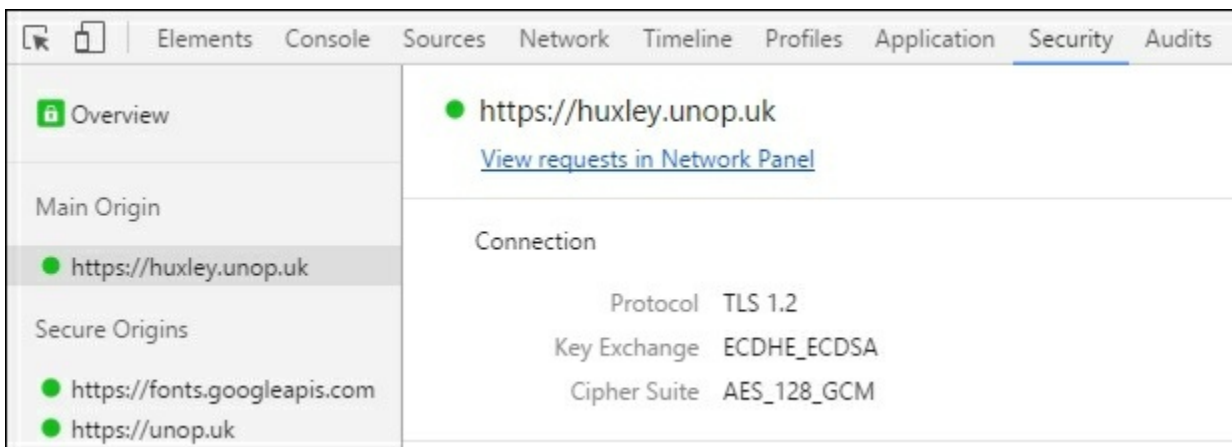
The following screenshot displays the same window for emoncms.org :
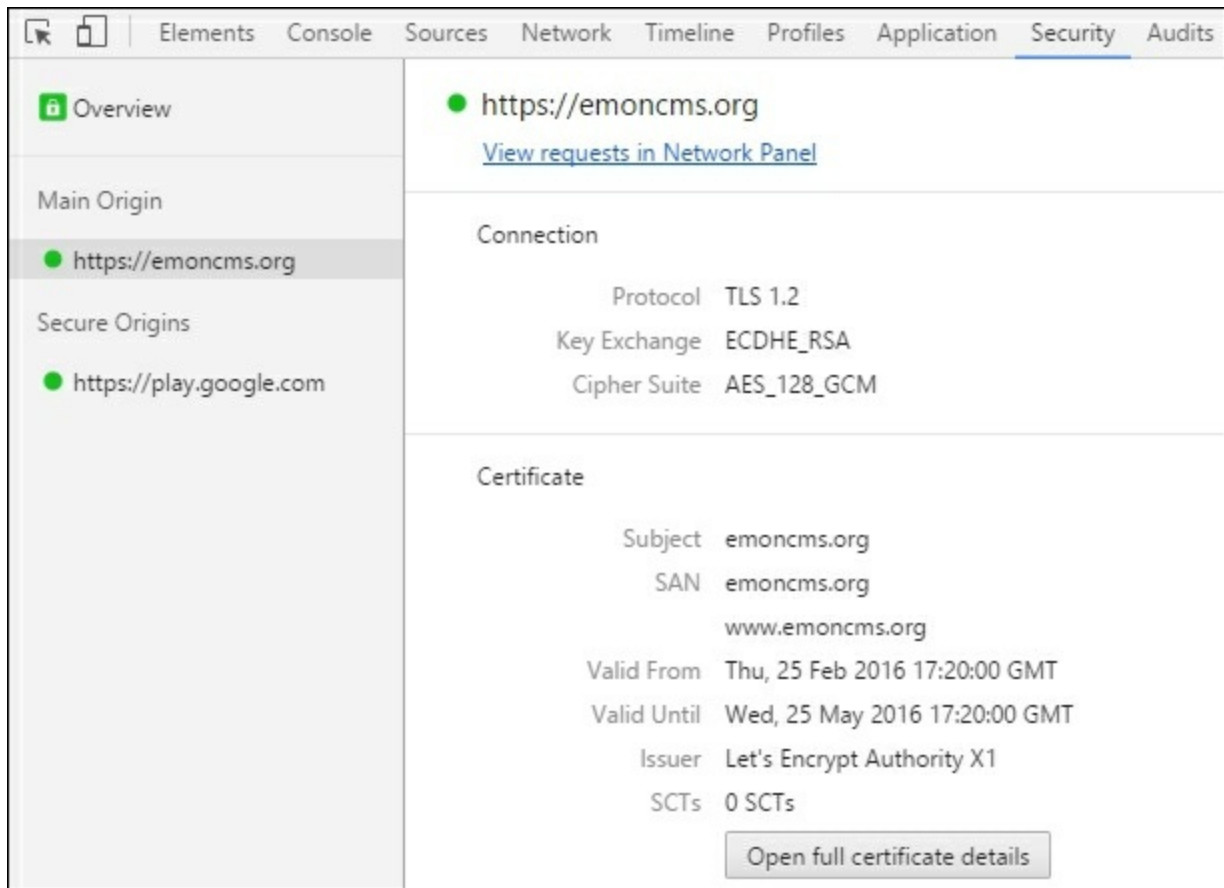
Tip

You may need to refresh the page to see TLS information if the site was already loaded when you opened the developer tools. You can access the same tab by clicking on the padlock in the Chrome URL bar and then clicking on the **Details** link.

You can view the certificate (in the native operating system certificate store) by clicking on the **Open full certificate details** button. A link with the same function exists on the equivalent screen of Chrome for Android, although the certificate information is reduced.

Performance tweaks

We already discussed the most important performance tweak for TLS because it is not about TLS. You should ensure that your HTTP connections are reusable, because if this is not the case, then you will incur the added penalty of the TLS negotiation along with the TCP overhead. Caching is also very important, and we will talk more about this later.

Note

TLS and HTTP both support compression, but these have security implications. Therefore, consider them carefully. They can leak information, and a determined adversary can use an analysis of them to recover encrypted data. TLS compression is deprecated, and it will be removed in TLS 1.3. Therefore, do not use it. We will discuss HTTP compression later on in this chapter.

In regard to specific advice for TLS, there are a few things, which you can do to improve performance. The main technique is to ensure that you use **session resumption**. This is different to reusing HTTP connections, and this means that clients can reuse an existing TLS connection without having to go through the whole key exchange.

Tip

You can implement sessions with IDs on the server or with encrypted tickets (in a similar manner to ASP.NET cookies that are encrypted with the machine key). There was a bug in the Microsoft client implementation around ticket encryption key rotation, but the **KB3109853** patch fixed it. Make sure that you install this update, especially if you see exceptions thrown when connecting to secure endpoints from your .NET code.

It is important to not overdo things and bigger is not always better, especially when it comes to key size. It is a trade-off between performance and security, and this will depend on your specific situation. In general, a good balance is not using 256 bit AES keys when 128 bit will do.

A 2048 bit RSA key is big enough, lower is insecure and larger is too slow. You can use the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) to sign instead of RSA, as it is much quicker. However, support is limited, so you would need to deploy RSA in parallel.

If you use ECDSA, then a 256 bit key is sufficient. For ECDHE, 256 bit is also fine, and for the slower version without elliptic curves (DHE), 2048 bit is sufficient. If you use ECDSA, then you will see this listed instead of the RSA signing in the connection details. For example, when visiting huxley.unop.uk , the details in the following screenshots are displayed in Firefox. This difference is also displayed in the previous Chrome screenshots:

Technical Details
Connection Encrypted (TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)

Additionally, it is important to include the full certificate chain with your certificate. If you fail to include all intermediate certificates, then the browser will need to download them until it finds one in its trusted root store. You can also use a technique called **Online Certificate Status Protocol** (**OCSP**) stapling, by embedding revocation data so browsers don't need to check a certificate revocation list.

Both of these certificate techniques may increase the size of payloads, which can be an issue if bandwidth is a concern. However, they will reduce the number of messages, which will increase performance if latency is the main problem, which is usually the case. Keeping key sizes small also helps a little with bandwidth. It is hard to recommend one generic approach. Therefore, as always, test for your unique situation.

There is also an alternative stream cipher called **ChaCha/Poly**, which is especially useful for mobile devices. This uses the **ChaCha20** stream cipher and the **Poly1305 Message Authentication Code** (**MAC**) algorithm to create a secure alternative to RC4. AES is a block cipher and is fast with hardware support, but many mobile devices and some older computers don't have this acceleration. ChaCha/Poly is faster when using just software. Therefore, this is better for battery life. This is supported in Chrome, including Chrome for Android, and in Firefox (from version 47).

Note

As all algorithms are different, you can't directly compare key sizes as a measure of how secure they are. For example, a 256 bit ECDHE key is equivalent to a 3072 bit RSA key. AES is very secure with relatively small keys, but you cannot use it for key exchange. ChaCha/Poly is more comparable to the security of AES 256 than AES 128.

In the following screenshot of Chrome on Android, you can see that when connecting to huxley.unop.uk , Chrome uses **CHACHA20_POLY1305** as the stream cipher, ECDHE for the key exchange, and ECDSA for the signature:

🔒 huxley.unop.uk

Chrome verified that COMODO ECC Domain Validation Secure Server CA 2 issued this website's certificate. The server did not supply any Certificate Transparency information.

Certificate information

🔒 Your connection to huxley.unop.uk is encrypted using a modern cipher suite.

The connection uses TLS 1.2.

The connection is encrypted and authenticated using CHACHA20_POLY1305 and uses ECDHE_ECDSA as the key exchange mechanism.

What do these mean?

Huxley

Note

The new version of TLS (1.3) is still a draft, but it may be finalized soon. It looks like it will only allow **Authenticated Encryption with Additional Data** (**AEAD**) ciphers. AES-GCM and ChaCha/Poly are the only two ciphers that currently meet these criteria. It will also remove some other obsolete features, such as TLS compression.

It may sometimes sound like using TLS is not always worth it, but it is an excellent idea to use HTTPS on your entire site, including any third-party resources that you load in. By doing this, you will be able to take advantage of the performance enhancing features of HTTP/2, which include techniques that mean that it is no longer crucial to serve resources (such as JavaScript libraries) from multiple domains. You can securely host everything yourself and avoid the DNS, TCP, and TLS overhead of additional requests. All of this can also be free because *Let's Encrypt* and *CloudFlare* provide certificates at zero cost. Let's look at HTTP/2 in detail now.

HTTP/2

As the name suggests, HTTP/2 is the new version of HTTP. It contains some significant performance improvements for the modern web. It was predated by **SPDY**, which has since been deprecated in favor of HTTP/2.

As mentioned previously, the first step toward using HTTP/2 is to use HTTPS on your entire site. Although not technically required, most clients (all the major browsers) mandate the use of TLS to enable HTTP/2. This is mainly due to the **Application-Layer Protocol Negotiation** (**ALPN**) that TLS provides, which allows easy support for HTTP/2. It also stops proxy servers from messing up the data, which many ISPs use to reduce their costs and record what their customers do online.

HTTP/2 improves performance in a number of ways. It uses compression even for the headers, and multiplexing, which allows multiple requests to share the same connection. It also allows the server to push resources that it knows the client will need before the client has realized it needs them. Although, this requires some configuration to set the correct headers and it can waste bandwidth if it is overused.

Multiplexing has implications for bundling and image concatenation (**sprites**), which we will talk about in the compression section later on in this chapter. This also means that you don't need to split assets over multiple domains (**shards**), where the extra overheads may even slow things down. However, you may still wish to use a

cookie-free subdomain to serve static assets without cookies, even though the new header compression means that the bandwidth savings will be smaller. If you use a naked domain (without a www), then you may need a new domain name for cookie-less use.

You can identify what version of HTTP is used to deliver your assets using the browser developer tools. In Firefox, you can see this on the details panel of the network tab. You will see the version listed as HTTP/1.1 when the old protocol is in use.

The following screenshot shows that packtpub.com uses **HTTP/1.1**:

```
  Headers        Cookies        Params
Request URL: https://www.packtpub.com/
Request method: GET
Remote address: 83.166.169.231:443
Status code: ● 200 OK
Version: HTTP/1.1
```
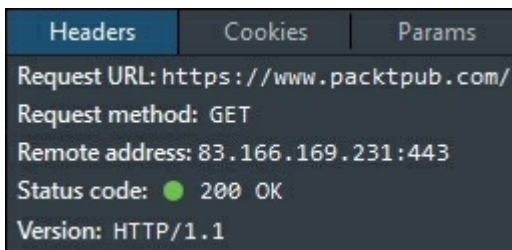
Tip

In Chrome, you can right-click on the column headers in the network inspector and add a **Protocol** column. You can also see more detailed network information by entering `chrome://net-internals` into the address bar. This displays things, such as sessions for HTTP/2 and **Quick UDP Internet Connections** (**QUIC**)—an experimental multiplexed stream transport.

The following screenshot shows that emoncms.org also uses **HTTP/1.1**, even though TLS is configured differently. The encrypted transport layer is transparent to HTTP:

```
  Headers        Cookies        Params
Request URL: https://emoncms.org/
Request method: GET
Remote address: 80.243.190.58:443
Status code: ● 200 OK
Version: HTTP/1.1
```
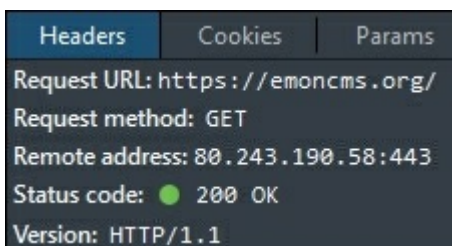
When HTTP/2 is used, you will see the version listed as **HTTP/2.0**. The following screenshot displays this for huxley.unop.uk , and it also displays CORS, caching, and

content compression headers:

WebSockets

WebSockets is a different protocol to HTTP. However, HTTP initiates it and it uses the same ports, so we'll discuss it briefly here. This HTML5 feature is useful for **push notifications** and **Real Time Communication** (**RTC**) applications. WebSockets use the `ws://` and `wss://` protocol prefixes instead of `http://` and `https://`. Once established by an existing HTTP connection, the protocol is full-duplex and binary in contrast to HTTP/1.

Before WebSockets, if your web server wanted to notify a client of a change, then you would have to use a technique, such as long polling. This is where a web request is held open by the server in case it wants to send something. When the request gets a response or it times-out, it is re-established. Needless to say, polling is never very efficient.

Push notifications can improve performance from a user's point of view because they receive updates as soon as they occur. They don't need to refresh anything or keep checking. You can immediately respond to the user when a long running process starts, run it asynchronously, and notify them immediately upon its completion.

Socket.IO is a popular WebSocket library for Node.js. To see it in action, you can look in the browser developer tools on a site that uses it. For example, if you open the dev tools and go to https://www.opentraintimes.com/maps/signalling/staines, you will see the connection being upgraded from HTTPS to WSS (or from HTTP to WS if you use the insecure version).

WebSockets predate HTTP/2, but they are still relevant despite the new server push technology. These two features appear similar, but they serve different purposes. WebSockets are for real-time and two-way data transfers, and server push is currently just to preload.

Note

In addition to HTTP/2 server push preloading, there is a new browser feature that is currently supported in Android, Chrome, and Opera, which allows you to declare resource preloading in markup using `rel="preload"` on a `link` tag. You can read the spec at w3c.github.io/preload and check the current state of browser support at caniuse.com/#feat=link-rel-preload .

In Chrome, the protocol switch will look something like the following screenshot. You can't see the contents of a WebSocket connection, so you won't be able to view the data being transferred from within the dev tools:

```
▼ General
    Request URL: wss://data.opentraintimes.com/socket.io/?EIO=3&transport=websocket
    Request Method: GET
    Status Code: ● 101 Switching Protocols
▼ Response Headers      view source
    Connection: upgrade
    Date: Sat, 27 Feb 2016 16:19:09 GMT
    Sec-WebSocket-Accept: GfOqW7FtWTX9yflqYhgJ1z2VJ8g=
    Sec-WebSocket-Extensions: permessage-deflate
    Server: nginx/1.8.1
    Upgrade: websocket
▼ Request Headers       view source
    Accept-Encoding: gzip, deflate, sdch
    Accept-Language: en-US,en;q=0.8,en-GB;q=0.6
    Cache-Control: no-cache
    Connection: Upgrade
```

There is a Microsoft library for ASP.NET, which is called `SignalR`. This library allows you to perform push notifications with WebSockets. It also falls back to long polling if the client or server does not support them. You will need a fairly recent version of Windows Server (2012 or later) and IIS (8.0 and above) to use WebSockets.

Unfortunately, the latest stable version (`SignalR 2`) does not support .NET Core. The new version (`SignalR 3`) is not planned for release until after the **Release to manufacturing** (**RTM**) of ASP.NET Core, but this is a top priority. You can try a

beta version, but it may be stable by the time you read this.

Tip

You may also wish to look at `StackExchange.NetGain` as a WebSocket server.

Compression

Data compression is a broad topic, and we can't hope to cover it all. Here, we will learn about lossless compression and how to use it in HTTP. We will also cover lossy image compression of pictures later in the chapter. Compression is important because if we can make files smaller, we can shorten the time that it takes to transfer them over a network.

Lossless compression algorithms

You may have noticed HTTP headers from some of the previous screenshots were related to encoding. The most common compression algorithms for HTTP are gzip and **DEFLATE**, which are very similar. These are both related to the algorithm used in ZIP files. If you do not already use HTTP compression, then this is a quick win, and it will improve the performance of your site if you enable it.

There are many other more advanced compression algorithms, such as **xz**, which is similar to the **7-Zip** (**7z**) format and uses the **Lempel-Ziv-Markov chain Algorithm (LZMA/LZMA2)**. However, there are currently only two additional algorithms in common use in major browsers. These are **Brotli** and **Shared Dictionary Compression for HTTP (SDCH)**. Both are from Google, but only Chrome supports SDCH, and it requires a large dictionary.

Brotli is more interesting, and Opera and Chrome (currently behind a `chrome://flags/#enable-brotli` flag), and Firefox by default (version 44 or higher) support it. Both browsers require the use of HTTPS to support Brotli (yet another good reason to use TLS), and the encoding token used in the headers is `br`. Brotli promises significant performance improvements, especially for mobile devices on slow connections.

If you access a site over HTTP, you will see the following for the request headers in the Chrome dev tools network inspector in the details of a request:

```
Accept-Encoding: gzip, deflate, sdch
```

However, if you use HTTPS then you will see this instead (after enabling the flag):

```
Accept-Encoding: gzip, deflate, sdch, br
```

The server can then respond with Brotli-encoded content using this response header:

```
Content-Encoding: br
```

For example, if you visit https://www.bayden.com/test/brotliimg.aspx in a supported browser, then Brotli will deliver the content (an image of a star). Here is a subset (for clarity and brevity) of the request headers from Chrome:

```
GET /test/brotliimg.aspx HTTP/1.1
Host: www.bayden.com
Connection: keep-alive
Accept-Encoding: gzip, deflate, sdch, br
```

This is a subset of the corresponding response headers:

```
HTTP/1.1 200 OK
Content-Type: image/png
Content-Encoding: br
Server: Microsoft-IIS/7.5
X-AspNet-Version: 4.0.30319
YourAcceptEncoding: gzip, deflate, sdch, br
```

Fiddler (the awesome HTTP debugging proxy by Eric Lawrence that we mentioned previously) also supports Brotli with a simple add-on (drop https://bayden.com/dl/Brotli.exe into `fiddler2\tools` and restart it). You can use this to easily test the impact on your site without deploying anything to your web servers.

Bundling and minification

Bundling and minification are techniques that you may already be familiar with. They speed up the delivery of static assets. They are usually used for text files, such as JavaScript and CSS content.

Bundling

Bundling is the technique of combining or concatenating multiple files together so that they can be delivered as one. This is a good idea when using HTTP/1.1 because the number of concurrent connections is limited. However, bundling is less necessary with HTTP/2, and in fact, it can reduce performance. The new multiplexing in HTTP/2 means that there is no longer a large penalty when you request many files instead of one that contain all of the same content. You can take advantage of this by only delivering what is needed for a page rather than the entire client side codebase for every page. Even if you selectively bundle per page, this could be inefficient.

For example, you may include a validation library for use with forms. However, because this is bundled, it will be sent to all pages, including the ones with no forms to validate. If you have a separate bundle for validated pages, then there may be duplication in the common core code that is also sent. By keeping things separated, the client can cache them individually and reuse components. This also means that if you change something, you only need to invalidate the cache of this one part. The client can keep using the other unmodified parts and not have to redownload them.

As always, you should measure for your particular use case. You may find that bundling still reduces the total file size. The overheads for HTTP/2 are much lower but still not zero, and compression can work better on larger files. However, keep in mind the implications for caching and reusability.

Minification

Minification is the process of reducing the file size of a textual static asset. We do this by various means, including stripping out comments, removing whitespace, and shortening variable names. It can also be useful to *obfuscate* code to make it harder to reverse engineer. Minification is still useful when you use HTTP/2, but you should be careful when testing to compare preminified and postminified files size after the lossless compression has also been applied.

As discussed previously, you should use HTTP content compression with at least the gzip or DEFLATE algorithms. These are pretty efficient; so, you may find that when compressed, your minified file is not much smaller than the compressed raw source

file.

Changes in ASP.NET Core

In the full .NET Framework and previous versions of MVC, there was an integrated bundling and minification system. This has changed for ASP.NET Core, and there are new tools to perform this work.

The new tools that were adopted include the task runner gulp, although you can use Grunt if you prefer to. Also, a generator tool called Yeoman is used for scaffolding. There are now package managers, such as **Bower** and **npm**, which are similar to NuGet, but for frontend libraries. For example, NuGet no longer delivers jQuery and Twitter Bootstrap, and they use Bower instead by default.

Most of these tools are written in JavaScript, and they run on Node.js. The package manager for Node.js is npm. These tools are popular in other open source web frameworks, and they are well established. They're not new to the scene, only new to .NET.

Gulp packages come from npm and are responsible for the minification of your static assets. This is now done at build time, as opposed to request time, as was previously the case. It works much more like a static site generator than a dynamic web application. A gulp file (`gulpfile.js`) in the root of your project configures these tasks using JavaScript.

The new tooling is not only restricted to ASP.NET Core, and you can use these features with traditional ASP.NET applications in Visual Studio. This is a good example of the cross-pollination and benefits that the new frameworks can provide to the existing ones.

Image optimization

Digital media compression is much more complicated than the lossless file compression that we talked about previously even if we just stick to images. We briefly mentioned when to use PNG and when to use JPEG in the previous chapter. Here, we'll go into much more detail and explore some other exotic options.

We covered the rule of thumb, which says that PNG is the best image format for icons and JPEG is better for photos. These two formats are the most common for lossless and lossy image compression, respectively.

We will talk more about other image formats later, but you are usually constrained to the popular formats by what browsers support. So, how can you get more out of the common choices?

PNG

Portable Network Graphics (**PNG**) is a lossless image compression format that internally works similarly to a ZIP file (using the DEFLATE algorithm). It's a good choice for images that contain solid blocks of color, and it has a better quality (with more colors) than the old **Graphics Interchange Format** (**GIF**).

PNG supports transparency in all modern browsers, so you should use it instead of GIF for static images. This is not a problem unless you need to support Internet Explorer 6, in which case this is probably the least of your troubles. PNG also supports animation with **Animated PNG** (**APNG**) files. These are like animated GIFs but of a much higher quality. Unfortunately, only Firefox and Safari support APNGs.

Tip

A great site to look up which browsers support a particular feature is caniuse.com . You can search for feature support, then check this against the user agent analytics of your site. For example, you could search for PNG-alpha, Brotli, or APNG.

Some ZIP algorithm implementations are better than others, and they produce smaller files that can still be decoded by everyone. For example, 7-Zip is much more efficient than most other zip compression software on Windows, even when using the ZIP format, not its native 7z format. Likewise, you can compress a PNG more compactly without losing any data and still have it work in all browsers. This usually comes with a higher upfront computational cost. However, if you compress static assets, which rarely change, then it can be well worth the effort.

You may already use the PNGOUT tool to losslessly reduce the size of your PNG images. If you're not, then you probably should. You can read more about it and download it at advsys.net/ken/utils.htm .

However, there is a new algorithm called Zopfli that offers better compression, but it is very slow to compress. Decompression is just as quick, so it's only a single optimization cost for precompiled resources. Zopfli is a precursor to Brotli, but it's compatible with DEFLATE and gzip, as it's not a new format.

You can get Zopfli from github.com/google/zopfli , but you should always test with your images and verify that there is indeed a file size reduction. You will find that these tools can help you deliver your assets quicker and achieve higher performance.

You may also use the practice of combining many sprites into one image. As with bundling, this is less necessary when using HTTP/2. However, the same caveats apply as with compression, and you should always test for your set of images.

JPEG

JPEG is a lossy image compression format, which means that it usually discards data from the picture to make it smaller. It is best suited to natural gradients and continuous tones, such as those found in photographs. JPEG does not support transparency like PNG does, so if you want to use one on different backgrounds, then you will need to prerender them.

Tip

It's a good space saving idea to remove the **Exchangeable image file format** (**Exif**) metadata from your JPEG files for the web. This contains information about the camera used and geographic data of where the photo was taken.

JPEG has a quality setting, which affects the image file size and the level of detail. The lower the quality, the smaller the file, but the worse it will look. You can perform tests on your images to see what settings provide an acceptable trade-off. Crucially, the best value for this quality setting will vary per image, depending on the content. There are tools that allow you to automatically detect the optimal quality level, such as Google's butteraugli.

There is an interesting project from Mozilla (the makers of the Firefox browser) called **mozjpeg**. This aims to better compress JPEG images and is similar to what PNGOUT and Zopfli do for PNG images. You can use mozjpeg to compress your JPEG images to a smaller size than normal, without affecting decompression or quality. It is available at github.com/mozilla/mozjpeg , but you will need to compile

it yourself. As always, results may vary, so test it for the photos on your site.

JPEG Archive (github.com/danielgtaylor/jpeg-archive) is a handy tool that uses mozjpeg to compress JPEG images, using various comparison metrics. Another similar tool is imgmin (github.com/rflynn/imgmin), which is slightly older.

Other image formats

Many other image formats are available, but you are usually limited on the web by what browsers support. As discussed in the previous chapter, you shouldn't scale images in the browser, or you will get poor performance. This usually means having to save multiple separate copies of smaller images, for example, when displaying thumbnails. Clearly this results in duplication, which is inefficient. Some of these new image formats have clever solutions to the problem of responsive and scalable images.

BPG is an image format by the talented Fabrice Bellard, and you can read more about it at bellard.org/bpg . It has a JavaScript polyfill to support browsers before native support is added to any of them.

WebP is an image format from Google, and only Chrome, Android, and Opera support it. It has impressive space savings over JPEG, and it will be a good choice if it becomes more widely supported, so check caniuse.com for the latest adoption stats.

JPEG2000 is an improved version of JPEG, although it may be encumbered by software patents, so it hasn't seen widespread adoption outside of medical imaging. Only Safari supports JPEG2000, and there is also JPEG XR, which is only supported in IE.

Whereas JPEG uses a **Discrete Cosine Transform** (**DCT**), JPEG2000 is based on a Wavelet Transform. One of the properties this provides is a *progressive* download. This means that the image is stored in such a way that if you download a small part from the beginning of the file, then you have a smaller and lower quality version of the full image. This has obvious applications for responsive and scalable images. The browser would only need to download enough of the image to fill the area it is rendering to, and the file need only be stored once. No resizing and no duplication for thumbnails would be required. This technique is also used in the **Free Lossless Image Format** (**FLIF**).

FLIF is one of the more exciting upcoming image formats, as it is progressive and responsive, but free and not patented. FLIF is still in development, but it promises to be very useful if browsers support it, and you can read more about it at flif.info.

Note

JPEG and PNG can support progressive download, but this isn't normally useful for responsive images. Progressive JPEG subjectively loads more gracefully and can even make files smaller, but interlaced PNG usually makes files bigger.

The problem is that most of these progressive image formats are not yet ready for the mainstream because all of the major browsers do not support them. It's a good idea to keep an eye on the future, but for now, we need to resize images for high performance.

Resizing images

Until new image formats gain widespread adoption, resizing is still required, and you may need to do this dynamically for different devices. Perhaps, you also have user-submitted image content, although you need to be very careful with this from a security point of view. Some image libraries are not safe, and a specially-crafted image can exploit your system. In fact, many image-processing libraries have issues when they are used in a web context.

If you are not extremely diligent and careful, then you can easily end up with memory leaks, which can take down your web server. It is always a good idea to separate and sandbox a process that deals with large media files.

Coming from a .NET standpoint, it can be tempting to use WinForms `System.Drawing` or its WPF successor (`System.Windows.Media`). However, these were designed for desktop software, and Microsoft strongly recommends against using them in a service or web application. Microsoft recommeds the **Windows Imaging Component** (**WIC**), but this is a **Component Object Model** (**COM**) API that is meant for use from C or C++ apps. In addition to this, none of these imaging libraries are cross-platform, so they are not suitable for use in .NET Core.

If you use Windows, then you could try using ImageResizer by Imazen (imazen.io), from imageresizing.net. While it still uses the **GDI+** `System.Drawing`, it is pretty battle hardened, so most of the bugs should have been worked out. There's also **DynamicImage**, which wraps the newer WPF image functions and uses shaders. You can read more about it at dynamicimage.apphb.com , although it hasn't been updated in a while, and it doesn't support .NET Core.

A popular option in open source circles is ImageMagick, which we've mentioned previously, and a fork called **GraphicsMagick**, which claims to be more efficient. Another popular image library is **LibGD**, and it's suitable for server use. You can read more at libgd.github.io . Although it's written in C, there are wrappers for other

programming languages, for example, **DotnetGD** targeting .NET Core.

One of the features that .NET Core lacks is that there is not yet a compelling option for image processing. ImageResizer 5 may help with this when released, so it is worth keeping an eye on it. Native code support is now much better in .NET Core, as it was a pain to do in classic .NET, which may help with integrating native cross-platform imaging libraries.

There is also a new cross-platform version of the open source **ImageProcessor** libraries (imageprocessor.org), called **ImageProcessorCore**, which shows promise. However, this is still a work in progress, and it is not yet stable. If you want to try it out, then you can get the nightly packages from MyGet or build it from source.

Note

Platform support and compatibility changes rapidly, so check ANCLAFS.com for the latest information. Feel free to contribute to this list or to the projects.

For now, it may be easier to install an open source service, such as Thumbor, or use a cloud-based imaging service, such as ImageEngine (WURFL.io) or Cloudinary, which we've already mentioned. Image manipulation is a common task, and it is effectively a solved problem. It may be better to use an existing solution and not reinvent the wheel, unless it's part of your core business or you have very unusual requirements.

Tip

Once you have your resized images, you can load them responsively with the `picture` and `source` tags using the `srcset` and `sizes` attributes. You can also use this technique to provide newer image formats (such as WebP), with a fallback for browsers that don't yet support them. Or you can use **Client Hints** (refer to httpwg.org/http-extensions/client-hints.html and caniuse.com/#feat=client-hints-dpr-width-viewport).

Caching

It is often said (originally by Phil Karlton) that caching is one of the hardest problems in computer science, along with naming things. This may well be an exaggeration, but caching is certainly difficult. It can also be very frustrating to debug if you are not methodical and precise in your approach.

Caching can apply at various different levels from the browser to the server using many diverse technologies. You rarely use just a single cache even if you don't realize it. Multiple caches don't always work well together, and it's vexing if you can't clear one.

We briefly touched upon caching in the previous chapter, and we'll go into much more detail in [Chapter 7](#), *Learning Caching and Message Queuing* . However, as caching has an impact on network performance, we'll cover it here as well.

Browser

A lot of caching happens in the web browser, which is inconvenient because as you don't have direct control over it (unless it's your browser). Asking users to clear their cache is unsatisfactory and confusing to many. Yet, you can exert influence on how browsers cache resources by carefully controlling the HTTP headers that you set and the URLs that you use.

If you fail to declare what resources are cacheable and for how long, then many browsers will just guess this. The heuristics for this can be wildly different between implementations. Therefore, this will result in suboptimal performance. You should be explicit and always declare cache information even (and especially) for assets that shouldn't be cached by marking them as noncacheable.

Tip

You need to be vigilant with what you advertise as cacheable because if you are careless, then you can get yourself into a situation where you're unable to update a resource. You should have a cache-busting strategy in place, and tested, before using caching.

There are various technologies that are used are to cache in browsers. Many different HTTP headers can be set, such as `Age`, `Cache-Control`, `ETag` (Entity Tag), `Expires`, and `Last-Modified`. These come from a few different standards, and the interactions can be complex, or they vary between browsers. We will explain these in more detail in Chapter 7, *Learning Caching and Message Queuing* .

Another technique is to use a unique URL for content. If a URL changes, then a browser will treat it as a different resource, but if it is the same, then it may load it from its local cache. Some frameworks calculate a hash of the file contents, and then they use this as a query string parameter. This way, when the contents of the file changes, so does the URL.

There are other and more modern features that you can use to cache, such as the **HTML5 Application Cache** (or **AppCache**). This was designed for offline web applications and wasn't very flexible. Busting the cache was complicated to put it mildly. AppCache is already deprecated, and you should use **Service Workers** instead. These provide much more flexibility, although support is pretty recent.

There are many improvements coming, in the latest browsers that give you more control, and we'll also show you how to use them in Chapter 7, *Learning Caching and Message Queuing* .

Server

The web server is a great place to cache because it is usually under your complete control. However, it's not really part of network performance, apart from generating the correct headers. There can be other great performance benefits with server-side caching in terms of improving the speed to generate pages, but we will cover these in later chapters.

If you use the traditional .NET Framework on Microsoft's **Internet Information Services** (**IIS**) web server, then you can use **output caching** from within your application. This will take care of setting the correct headers and sending 304 (Not Modified) responses to browser requests. It will also cache the output on the server in memory, on disk or using Memcached/Redis. You can add attributes to your controller action methods to control the caching options, but other ways of doing this are available, for example, in the configuration files.

`OutputCache` is not available in ASP.NET Core, but you can use `ResponseCache` to set the correct headers. The output is not cached on the server, but you can install a caching proxy in front of it. Again, we will cover this more and demonstrate server-side caching in [Chapter 7](), *Learning Caching and Message Queuing* .

If you want to disable caching on an ASP.NET Core page, then add this annotation to your controller action:

```
[ResponseCache(NoStore = true, Duration = 0)]
```

This will set the following header on the HTTP response and ensure that it is not cached:

```
Cache-Control: no-store
```

To cache a page for an hour, add the following instead, `Duration` is in seconds:

```
[ResponseCache(Duration = 3600, VaryByHeader = "Accept")]
```

The cache control header will then look like the following:

```
Cache-Control: public,max-age=3600
```

There's plenty more to say about other caching configuration options and profiles. Therefore, if you're interested, then read the later chapters. It's a complex topic, and we've only scratched the surface here.

Note

You can read documentation about response caching in ASP.NET Core at docs.asp.net/en/latest/performance/caching/response.html .

These caching directives not only instruct the browser, but they also instruct any proxies on the way. Some of these may be in your infrastructure if you have a caching proxy, such as Squid, Varnish, or HAProxy. Or perhaps, you have a TLS-terminating load balancer (such as **Azure Application Gateway**) to reduce the load on your web servers that also caches. You can forcibly flush the caches of servers that you control, but there may be other caches in between you and your users where you can't do this.

Proxy servers between you and your users

There can be many proxy servers between you and your users over which you have no direct control. They may ignore your caching requests, block parts of your site, or even modify your content. The way to solve these problems is to use TLS, as we have already discussed. TLS creates a secure tunnel so that the connection between your infrastructure and the browser can't easily be tampered with.

Corporate proxies commonly **Man in the Middle attack** (**MitM**) your connection to the user so that they can spy on what employees are doing online. This involves installing a custom-trusted root certificate on users' workstations so that your certificate can be faked. Unfortunately, there isn't much you can do about this, apart from educating users. **Certificate pinning** is effective in native apps, but it's not so useful for web applications. **HTTP Public Key Pinning** (**HPKP**) is available but, as it is a **Trust on First Use** (**TOFU**) technique, the initial connection could be intercepted. Client certificates are another option, but they can be difficult to distribute, and they aren't commonly used.

MitM can be useful if you trust the third party and remain in control. This is used by some **Content Delivery Networks** (**CDNs**) to speed up your site.

CDNs

CDNs can improve the performance of your site by storing copies of your content at locations closer to your users. Services, such as the ones provided by CloudFlare, perform a MitM on your connection and save copies at data centers around the world. The difference from an unauthorized proxy is that you control the configuration, and you can purge the cache whenever you like.

You should be careful because if you don't use the caching features, then this can reduce the responsiveness of your site due to the extra hops involved. Make sure that you monitor the response times with and without a CDN, and you need a fallback plan in case they go down.

Another common use case for CDNs is to distribute popular libraries, for example, the jQuery JavaScript library. There are free CDNs from jQuery (MaxCDN), Google, Microsoft, and cdnjs (CloudFlare) that do this. The hypothesis is that a user may already have the library from one of these in their cache. However, you should be extremely careful that you trust the provider and connection. When you load a third-party script into your site, you are effectively giving them full control over it or at least relying on them to always be available.

If you choose to use a CDN, then ensure that it uses HTTPS to avoid tampering with scripts. You should use explicit `https://` URLs on your secure pages or at least protocol agnostic URLS (`//`), and never `http://`. Otherwise, you will get mixed content warnings, which some browsers display as totally unencrypted or even block.

You will need a fallback that is hosted on your own servers anyway in case the CDN goes down. If you use HTTP/2, then you may find that there is no advantage to using a CDN. Obviously, always test for your situation.

There are some useful new features in ASP.NET Core views to easily enable local fallback for CDN resources. We'll show you how to use them and other features in later chapters.

Summary

In this chapter, you learned how to improve performance at the network level between the edge of your infrastructure and your users. You now know more about the internet protocols under your application and how to optimize your use of them for best effect.

You learned how to take advantage of compression to shrink text and image files. This will reduce bandwidth and speed up delivery of assets. We also highlighted caching, and you should now see how important it is. We'll cover caching more in [Chapter 7](), *Learning Caching and Message Queuing* .

In the next chapter, you will learn how to optimize the performance inside your infrastructure. You will see how to deal with I/O latency, and how to write well-performing SQL.

Chapter 5. Optimizing I/O Performance

This chapter addresses issues that often occur when you take your functionally tested application and split it up into parts for deployment. Your web servers host the frontend code, your database is somewhere else in the data center, you may have a **Storage Area Network (SAN)** for centralized files, an app server for APIs, and the virtual disks are all on different machines as well.

These changes add significant latency to many common operations and your application now becomes super slow, probably because it's too chatty over the network. In this chapter, you will learn how to fix these issues by batching queries together, and performing work on the best server for the job. Even if everything runs on one machine, the skills that you'll learn here will help to improve performance by increasing efficiency.

The topics covered in this chapter include the following:

- The operations that can be slow
- Select N+1 problems in detail
- Returning only what you need
- Writing high-performance SQL

You will learn about the operations that you shouldn't use synchronously, and how to query for getting only the data that you need in an efficient manner. You'll also see how to tame your O/RM, and learn to write high-performance SQL with **Dapper**.

We briefly covered some of these topics in Chapter 3 , *Fixing Common Performance Problems*, but here we'll dive into greater detail. The first half of this chapter will focus on background knowledge and using diagnostic tools, while the second half will show you solutions to issues you may come across. You'll also learn about some more unusual problems, and how to fix or alleviate them.

We'll initially focus on understanding the issues, because if you don't appreciate the root cause of a problem, then it can be difficult to fix. You shouldn't blindly apply advice that you read, and expect it to work successfully. Diagnosing a problem is normally the hard part, and once this is achieved, it is usually easy to fix it.

Input/output

I/O is a general name for any operation in which your code interacts with the outside world. There are many things that count as I/O, and there can be plenty of I/O that is internal to your software, especially if your application has a distributed architecture.

Note

The recent rise in popularity of the `.io` **Top Level Domain** (**TLD**) can be partly attributed to standing for I/O, but that is not its real meaning. As is the case for some other TLDs, it is actually a country code. Other examples include `.ly` for Libya and `.tv` for Tuvalu (which, like the neighboring Kiribati, may soon be submerged beneath the Pacific Ocean due to climate change).

The TLD `.io` is intended for the **British Indian Ocean Territory** (**BIOT**), a collection of tiny but strategic islands with a shameful history. The `.io` TLD is therefore controlled by a UK-based registry. BIOT is nothing more than a military base, and also happens to be a hop on the proposed AWE fiber optic cable between Australia and Djibouti.

In this chapter, we will focus on improving the speed of I/O, not on avoiding it. Therefore, we won't cover caching here. Both I/O optimizing and caching are powerful techniques on their own, and when they're combined, you can achieve impressive performance. See Chapter 7, *Learning Caching and Message Queuing* for more on caching.

Categories of I/O

The first challenge is to identify the operations that trigger I/O. A general rule of thumb in .NET is that if a method has an asynchronous API (`MethodAsync()` variants), then it is declaring that it can be slow, and may be doing I/O work. Let's take a closer look at some of the different kinds of I/O.

Disks

The first type of I/O we will cover is reading from, and writing to, persistent storage. This will usually be some sort of a disk drive such as a spinning platter **Hard Disk Drive** (**HDD**), or as is more common these days, a flash memory-based **Solid State Drive** (**SSD**).

HDDs are slower than SSDs for random reads and writes, but are competitive for large block transfers. The reason for this is that the arm on the HDD has to physically move the head to the correct location on the magnetic platter before it can begin the read or write operations. If the disk is powered down, then it can take even longer, as the platters will have to **Spin-up** from a stationary position to the correct **revolutions per minute** (**rpm**) beforehand.

Note

You may have heard the term "Spin-up" in reference to provisioning a generic resource. This historically comes from the time taken to spin the platters on a rotating disk up to the operational speed. The term is still commonly used, even though these days there may not be any mechanical components present.

Terminology like this often has a historical explanation. As another example, a floppy disk icon is normally used to represent the save function. Yet floppy disks are no longer in use, and many younger users may never have encountered one.

Knowing what type of drive your code is running on is important. HDDs perform badly if you make lots of small reads and writes. They prefer to have batched operations, so writing one large file is better than many smaller ones.

The performance of disks is similar to that of a network, in that there is both latency and throughput, often called bandwidth in networking parlance. The latency of an HDD is high, as it takes a relatively long time to get started, but once started, the throughput can be respectable. You can read data rapidly if it's all in one place on the disk, but it will be slower if it is spread all over, even if the total data is less. For example, copying a single large file disk-to-disk is quick, but trying to launch many programs simultaneously is slow.

SSDs experience fewer of these problems as they have lower latency, but it is still beneficial to keep random writes to a minimum. SSDs are based on **flash memory** (similar to the chips used in memory cards for phones and cameras), and they can only be written to a fixed number of times. The controller on the SSD manages this for you, but the SSD's performance degrades over time. Aggressive writing will accelerate this degradation.

Multiple disks can be combined to improve their performance and reliability characteristics. This is commonly done using a technology called **Redundant Array of Independent Disks** (**RAID**). Data is split across multiple disks to make it quicker to access, and more tolerant to hardware failures. RAID is common in server hardware, but can increase the startup time, as Spin-up is sometimes staggered to reduce the peak power draw.

HDDs offer much larger capacity than SSDs, and so are a good choice for storage of infrequently used files. You can get hybrid drives, which combine an HDD with an SSD. These claim to offer the best of both worlds, and are cheaper than SSDs of an equivalent size. However, if you can afford it, and if you can fit all of your data on an SSD, then you should use one. You will also decrease your power and cooling requirements, and you can always add an additional HDD for mass storage or backups.

Virtual file systems

File access can be slow at the best of times due to the physical nature of the disks storing the data, as mentioned previously. This problem can be compounded in a virtualized environment such as a cloud-hosted infrastructure. The storage disks are usually not on the same host server as the virtual machine, and will generally be implemented as network shares even if they appear to be mounted locally. In any case, there is always an additional problem, which is present whether the disk is on the VM host or somewhere else on the network, and that is contention.

On a virtualized infrastructure, such as those provided by AWS and Azure, you share the hardware with other users, but a physical disk can only service a single request at once. If multiple tenants want to access the same disk simultaneously, then their operations will need to be queued and timeshared. Unfortunately, this abstraction has much the same detrimental effect on performance as reading lots of random files. Users are likely to have their data on disk stored in locations different from other customers. This will cause the arm on the drive to frequently move to different sectors, reducing throughput and increasing latency for everyone on the system.

All this means that on shared virtual hosting, using an SSD can have a bigger positive

performance impact than normal. Even better is to have a local SSD, which is directly attached to the VM host, and not to another machine on the network. If disks must be networked, then the storage machine should be as close as possible to the VM using it.

You can pay extra for a dedicated VM host where you are the only tenant. However, you may as well then be running on bare metal, and reaping the benefits of reduced costs and higher performance. If you don't require the easy provisioning and maintenance of VMs, then a bare metal dedicated server may be a good option.

Many cloud hosting providers now offer SSDs, but most only offer ephemeral local disks. This means that the local disk only exists while your VM is running, and vanishes when it is shutdown, making it unsuitable for storing the OS if you want to bring a VM back up in the same state.

You have to write your application in a different way to take advantage of an ephemeral local drive, as it could disappear at any time, and so can only be used for temporary working storage. This is known as an **immutable server**, which means it doesn't change and is disposable. This normally works better when the OS is Linux, as it can be tricky to bootstrap new instances when running Windows.

Databases

Databases can be slow, because they rely on disks for the storage of their data, but there are other overheads as well. However, DBs are usually a better way of storing significant data than flat files on disk. Arbitrary data can be retrieved quickly if it is indexed, much quicker than scanning a file by brute force.

Relational databases are a mature and very impressive technology. However, they only shine when used correctly, and how you go about querying them makes a massive difference to performance. DBs are so convenient that they're often overused, and are typically the bottleneck for a web application.

An unfortunately common *anti-pattern* is requiring a database call in order to render the homepage of a website. An example is when you try to visit a website mentioned on live TV, only to discover that it has crashed due to the MySQL DB being overloaded. This sort of a website would be better architected as a static site with the client-side code hitting cached and queued web APIs.

The pathological case for a slow DB is where the web server is in one data center, the database server is in another, and the disks for the DB are in a third. Also, all the servers may be shared with other users. Obviously, it's best not to end up in this situation, and to architect your infrastructure in a sane way, but you will always have

some latency.

There are application programming techniques that allow you to keep your network and DB chatter to a minimum. These help you to improve the performance and responsiveness of your software, especially if it is hosted in a high-latency virtualized environment. We will demonstrate some of these skills later on in this chapter.

APIs

Modern web application programming generally involves using third-party services and their associated APIs. It's beneficial to know where these APIs are located, and what the latency is. Are they in the same data center, or are they on the other side of the planet? Unless you've discovered some exciting new physics, then light only travels so fast.

Note

Today, almost all intercontinental data travels by fiber optics cables. Satellites are rarely used anymore, as the latency is high, especially for geostationary orbits. Many of these cables are under the oceans, and are hard to fix. If you rely on an API on a different continent, not only can it slow you down, but it also exposes you to additional risk.

You probably shouldn't build an important workflow that can be disrupted by a fisherman trawling in the wrong place. You also need to further secure your data, as some countries (such as the UK) are known to tap cables and store the communications, if they cross their borders.

One of the issues with APIs is that latency can compound. You may need to call many APIs, or maybe an API calls another API internally. These situations are not normally designed this way, but can grow organically as new features are added, especially if no refactoring is performed periodically to tidy up any mess.

One common form of latency is startup time. Websites can go to sleep if not used, especially if using the default **Internet Information Services** (**IIS**) settings. If a website takes a non-negligible amount of time to wake up, and all the required APIs also need to wake up, then the delays can quickly add up to a significant lag for the first request. It may even time-out.

There are a couple of solutions to this initial lag problem. If you use IIS, then you can configure the application pool to not go to sleep. The defaults in IIS are optimized for shared hosting, so they will need tweaking for a dedicated server. The second option

is to keep the site alive by regularly polling it with a health check or uptime monitoring tool. You should be doing this anyway so that you know when your site goes down, but you should also ensure that you are exercising all the required dependencies (such as APIs and DBs). If you are simply retrieving a static page or just checking for a 200 status code, then services may go down without you realizing.

Similarly, scaling can have a lag. If you need to scale up, then you should preheat your load balancers and web servers. This is especially important if using an AWS **Elastic Load Balancer** (**ELB**). If you're expecting a big peak in traffic, then you can ask AWS to have your ELBs prewarmed. An alternative would be using **Azure Load Balancer**, **Azure Application Gateway**, or running HAProxy yourself so that you have more control. You should also be running load tests, which we'll cover in Chapter 9 , *Monitoring Performance Regressions* .

Network diagnostics tools

As we discovered earlier, practically all I/O operations in a virtualized or cloud-hosting infrastructure are now network operations. Disks and databases are rarely local, as this would prevent scaling out horizontally. There are various command-line tools that can help you discover where the API, DB, or any other server you're using is located, and how much latency is present on the connection.

While all of these commands can be run from your workstation, they are most useful when run from a server via a **Secure Shell** (**SSH**) or **Remote Desktop Protocol** (**RDP**) connection. This way, you can check where your databases, APIs, and storage servers are, in relation to your web servers. Unfortunately, it is common for hosting providers to geographically separate your servers, and put them in different data centers.

For example, if using AWS, then you would want to configure your servers to be in at least the same region, and preferably in the same Availability Zone (**AZ**), which usually means the same data center. You can replicate (cluster) your DB or file server across AZs (or even across regions) so that your web servers are always talking to a server on their local network. This also adds redundancy, so in addition to increasing performance, it will make your application more resilient to hardware faults or power supply failures.

Ping

Ping is a simple networking diagnostics tool, available on almost all operating systems. It operates at the IP level and sends an **Internet Control Message Protocol** (**ICMP**) echo message to the host specified.

Not all machines will respond to pings, or requests may be blocked by firewalls. However, it's good netiquette to allow servers to respond for debugging purposes, and most will oblige. For example, open a command prompt or terminal, and type the following:

```
ping ec2.eu-central-1.amazonaws.com
```

This will ping an **Amazon Web Services** (**AWS**) data center in Germany. In the response, you will see the time in milliseconds. From the UK, this **round-trip time** (**RTT**) may be something like `33ms`, but your results will vary.

Tip

On Windows, by default, ping performs four attempts, then exits. On a Unix-like OS (such as Mac OS X, BSD, or Linux), by default, it continues indefinitely. Press **Ctrl+C** to stop and quit.

Try this command next, which will do the same, but for an AWS data center in Australia:

```
ping ec2.ap-southeast-2.amazonaws.com
```

From the UK, the latency now goes up, by almost an order of magnitude, to around 300 ms. AWS doesn't have any data centers in the UK, and neither do Microsoft and Google (read into that what you will). So to ping a UK hosting provider, enter the following:

```
ping bytemark.co.uk
```

The latency now decreases to an average of `23ms`, as our connection has (probably) not left the country. Obviously, your results will vary depending on where you are. Next we'll see how to discover what route our data is taking, as it's not always only distance that's important. The number of hops can likewise be significant.

The following image shows the output of the three `ping` operations that we have just performed to Germany, the UK, and Australia. Note the difference in the timings; however, your results will be different, so try this for yourself.

```
C:\Users\James>ping ec2.eu-central-1.amazonaws.com

Pinging ec2.eu-central-1.amazonaws.com [54.239.54.36] with 32 bytes of data:
Reply from 54.239.54.36: bytes=32 time=33ms TTL=241
Reply from 54.239.54.36: bytes=32 time=33ms TTL=241
Reply from 54.239.54.36: bytes=32 time=33ms TTL=241
Reply from 54.239.54.36: bytes=32 time=33ms TTL=241

Ping statistics for 54.239.54.36:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 33ms, Maximum = 33ms, Average = 33ms

C:\Users\James>ping ec2.ap-southeast-2.amazonaws.com

Pinging ec2.ap-southeast-2.amazonaws.com [54.240.195.29] with 32 bytes of data:
Reply from 54.240.195.29: bytes=32 time=297ms TTL=237
Reply from 54.240.195.29: bytes=32 time=298ms TTL=237
Reply from 54.240.195.29: bytes=32 time=297ms TTL=237
Reply from 54.240.195.29: bytes=32 time=297ms TTL=237

Ping statistics for 54.240.195.29:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 297ms, Maximum = 298ms, Average = 297ms

C:\Users\James>ping bytemark.co.uk

Pinging bytemark.co.uk [80.68.81.80] with 32 bytes of data:
Reply from 80.68.81.80: bytes=32 time=22ms TTL=53
Reply from 80.68.81.80: bytes=32 time=22ms TTL=53
Reply from 80.68.81.80: bytes=32 time=27ms TTL=53
Reply from 80.68.81.80: bytes=32 time=22ms TTL=53

Ping statistics for 80.68.81.80:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 22ms, Maximum = 27ms, Average = 23ms
```

Tip

IPv4 addresses starting with 54 (the ones in the form `54.x.x.x`) are a clue that the server may be running on an AWS **Elastic Compute Cloud** (**EC2**) virtual server. Perform a reverse DNS lookup with `nslookup` or `ping` (covered later in this chapter) to confirm if this is the case. AWS provides IP address ranges at the following link: docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html

Tracert

Tracert (or traceroute on a Unix-like OS) is a tool which, as the name suggests, traces the route to a destination host. Enter the following command:

```
tracert www.google.com
```

You should be able to see the connection leaving the network of your **Internet Service Provider** (**ISP**), and entering the domain `1e100.net`, which is Google's domain. 1.0×10^{100} is a **googol**, which is their namesake. The following image shows the output that you might see for this trace:

```
C:\Users\James>tracert www.google.com

Tracing route to www.google.com [173.194.112.180]
over a maximum of 30 hops:

  1    1 ms     1 ms     1 ms  BThomehub.home [192.168.1.254]
  2    *        *        *     Request timed out.
  3   16 ms    18 ms    15 ms  31.55.186.181
  4   16 ms    16 ms    17 ms  31.55.186.180
  5   17 ms    16 ms    16 ms  core4-hu0-6-0-3.faraday.ukcore.bt.net [195.99.127.202]
  6   19 ms    17 ms    17 ms  peer1-xe0-1-0.faraday.ukcore.bt.net [213.121.193.173]
  7   17 ms    16 ms    16 ms  109.159.253.67
  8   16 ms    16 ms    16 ms  209.85.244.182
  9   17 ms    17 ms    17 ms  209.85.250.184
 10   22 ms    23 ms    23 ms  209.85.253.108
 11   26 ms    27 ms    28 ms  74.125.37.102
 12   75 ms    74 ms    73 ms  66.249.95.38
 13  231 ms    29 ms    28 ms  216.239.57.148
 14   37 ms    28 ms    38 ms  72.14.238.57
 15   68 ms    71 ms    70 ms  fra07s32-in-f20.1e100.net [173.194.112.180]

Trace complete.
```

Next, let's trace a route to Australia by running the following command with the same AWS host name as our earlier example, as follows:

```
tracert ec2.ap-southeast-2.amazonaws.com
```

This may take some time to run, especially if some hosts don't respond to pings and traceroute has to timeout. If you get asterisks (* * *), then this could indicate the presence of a firewall. Your results may look something like the following image:

```
C:\Users\James>tracert ec2.ap-southeast-2.amazonaws.com

Tracing route to ec2.ap-southeast-2.amazonaws.com [54.240.195.144]
over a maximum of 30 hops:

  1    1 ms     3 ms     3 ms  BThomehub.home [192.168.1.254]
  2    *        *        *     Request timed out.
  3    *        *        *     Request timed out.
  4   16 ms    16 ms    16 ms  31.55.186.180
  5   16 ms    16 ms    16 ms  195.99.127.42
  6   26 ms   235 ms    17 ms  peer2-et-9-3-0.faraday.ukcore.bt.net [62.6.201.195]
  7  198 ms    19 ms    15 ms  213.137.183.98
  8   16 ms    16 ms    16 ms  82.112.115.185
  9  175 ms   174 ms   174 ms  ae-1.r03.londen05.uk.bb.gin.ntt.net [129.250.6.230]
 10  173 ms   173 ms   171 ms  ae-15.r03.amstnl02.nl.bb.gin.ntt.net [129.250.6.25]
 11   22 ms    23 ms    23 ms  ae-4.r25.amstnl02.nl.bb.gin.ntt.net [129.250.2.146]
 12  100 ms   100 ms   101 ms  ae-5.r23.asbnva02.us.bb.gin.ntt.net [129.250.6.162]
 13  101 ms   101 ms   102 ms  ae-0.r22.asbnva02.us.bb.gin.ntt.net [129.250.3.84]
 14  167 ms   166 ms   166 ms  ae-5.r23.lsanca07.us.bb.gin.ntt.net [129.250.3.189]
 15  176 ms   175 ms   176 ms  ae-2.r00.lsanca07.us.bb.gin.ntt.net [129.250.3.238]
 16  167 ms   168 ms   168 ms  ae-1.amazon.lsanca07.us.bb.gin.ntt.net [129.250.198.98]
 17    *        *        *     Request timed out.
 18    *        *        *     Request timed out.
 19    *        *        *     Request timed out.
 20  305 ms   305 ms   304 ms  54.240.203.85
 21  305 ms   305 ms   305 ms  54.240.192.115
 22  305 ms   305 ms   305 ms  54.240.192.181
 23  305 ms   305 ms   305 ms  54.240.195.144

Trace complete.
```

In the preceding example, we can see the connection leaving the **British Telecom** (**BT**) network, and entering the **Nippon Telegraph and Telecom** (**NTT**) Global IP Network. We can even see the route taken from London to Sydney, via Amsterdam, Ashburn (east US, in Virginia), and Los Angeles. The hostnames suggest that the connection has gone via the Faraday telephone exchange building, near St. Paul's Cathedral in London (named after electrical pioneer Michael Faraday), and entered Amazon's network in LA.

Note

This isn't the whole story as it only shows the IP level. At the physical level, the fiber likely comes back to the UK from the Netherlands (possibly via Porthcurno, Goonhilly Satellite Earth Station, or more likely, Bude where GCHQ conveniently have a base). Between LA and Australia, there will also probably be a stopover in Hawaii (where the NSA base that Edward Snowden worked at is located).

There are maps of the connections available at submarinecablemap.com and www.us.ntt.net/about/network-map.cfm. It's good idea to have at least a basic understanding of how the internet is physically structured, in order to achieve high performance.

If we now trace the routes to the AWS data centers in Korea and Japan, we can see that, initially, they both take the same route as each other. They go from London to New York and then to Seattle, before reaching Osaka in Japan. The Korean trace then carries on for another eleven hops, but the Japanese trace is done in six, which makes logical sense.

The following image shows the typical results of a trace to Korea first, then the results of a second trace to Japan.

```
C:\Users\James>tracert ec2.ap-northeast-2.amazonaws.com

Tracing route to ec2.ap-northeast-2.amazonaws.com [52.95.192.85]
over a maximum of 30 hops:

  1     1 ms     1 ms     1 ms  BThomehub.home [192.168.1.254]
  2     *        *        *     Request timed out.
  3     *        *        *     Request timed out.
  4    36 ms    16 ms    16 ms  31.55.186.188
  5    16 ms    16 ms    15 ms  core4-hu0-6-0-1.faraday.ukcore.bt.net [195.99.127.200]
  6    25 ms    15 ms    15 ms  peer2-et-10-3-0.faraday.ukcore.bt.net [62.6.201.199]
  7    19 ms    15 ms    15 ms  213.137.183.100
  8    17 ms    17 ms    16 ms  82.112.115.185
  9   273 ms   274 ms   273 ms  ae-13.r02.londen03.uk.bb.gin.ntt.net [129.250.2.118]
 10    17 ms    16 ms    16 ms  ae-4.r22.londen03.uk.bb.gin.ntt.net [129.250.5.24]
 11    87 ms    87 ms    88 ms  ae-5.r24.nycmny01.us.bb.gin.ntt.net [129.250.2.18]
 12   172 ms   179 ms   157 ms  ae-1.r21.sttlwa01.us.bb.gin.ntt.net [129.250.4.13]
 13   250 ms   250 ms   250 ms  ae-2.r20.osakjp02.jp.bb.gin.ntt.net [129.250.3.86]
 14   252 ms   248 ms   311 ms  ae-4.r22.osakjp02.jp.bb.gin.ntt.net [129.250.6.188]
 15   280 ms   275 ms   274 ms  ae-1.r00.osakjp02.jp.bb.gin.ntt.net [129.250.2.253]
 16   255 ms   256 ms   254 ms  ae-0.amazon.osakjp02.jp.bb.gin.ntt.net [61.200.82.122]
 17   261 ms   261 ms   261 ms  54.239.52.142
 18   286 ms   286 ms   286 ms  54.239.52.149
 19     *        *        *     Request timed out.
 20   294 ms   291 ms   298 ms  54.239.122.238
 21   322 ms   289 ms   295 ms  54.239.122.245
 22   422 ms   335 ms   295 ms  54.239.122.38
 23     *        *        *     Request timed out.
 24     *        *        *     Request timed out.
 25     *        *        *     Request timed out.
 26     *        *        *     Request timed out.
 27   292 ms   296 ms   295 ms  52.95.192.85

Trace complete.

C:\Users\James>tracert ec2.ap-northeast-1.amazonaws.com

Tracing route to ec2.ap-northeast-1.amazonaws.com [27.0.1.195]
over a maximum of 30 hops:

  1     2 ms     1 ms     1 ms  BThomehub.home [192.168.1.254]
  2     *        *        *     Request timed out.
  3     *        *        *     Request timed out.
  4    16 ms    16 ms    16 ms  31.55.186.180
  5    16 ms    16 ms    16 ms  core3-hu0-6-0-1.faraday.ukcore.bt.net [195.99.127.192]
  6    15 ms    16 ms    16 ms  peer2-et-9-3-0.faraday.ukcore.bt.net [62.6.201.195]
  7    15 ms    15 ms    16 ms  213.137.183.96
  8    16 ms    16 ms    16 ms  82.112.115.185
  9   273 ms   274 ms   274 ms  ae-13.r02.londen03.uk.bb.gin.ntt.net [129.250.2.118]
 10    16 ms    17 ms    17 ms  ae-4.r22.londen03.uk.bb.gin.ntt.net [129.250.5.24]
 11    82 ms    83 ms    83 ms  ae-5.r24.nycmny01.us.bb.gin.ntt.net [129.250.2.18]
 12   153 ms   153 ms   152 ms  ae-1.r21.sttlwa01.us.bb.gin.ntt.net [129.250.4.13]
 13   242 ms   243 ms   242 ms  ae-2.r20.osakjp02.jp.bb.gin.ntt.net [129.250.3.86]
 14   294 ms   309 ms   283 ms  ae-4.r23.osakjp02.jp.bb.gin.ntt.net [129.250.6.90]
 15   273 ms   273 ms   273 ms  ae-2.r00.osakjp02.jp.bb.gin.ntt.net [129.250.3.197]
 16   247 ms   246 ms   246 ms  ae-0.amazon.osakjp02.jp.bb.gin.ntt.net [61.200.82.122]
 17   259 ms   259 ms   259 ms  27.0.0.250
 18   255 ms   412 ms   306 ms  54.239.52.135
 19   261 ms   262 ms   260 ms  27.0.0.67
 20   254 ms   254 ms   254 ms  27.0.0.155
 21   255 ms   255 ms   255 ms  27.0.0.175
 22   255 ms   255 ms   255 ms  27.0.1.195

Trace complete.
```

You can use the difference in time between hops to work out the approximate geographic distance. However, there are occasionally anomalies if some systems respond quicker than others.

Tip

If you're on a flight with free Wi-Fi, then a traceroute is an interesting exercise to perform. The internet connection is likely going via satellite, and you'll be able to tell

the orbit altitude from the latency. For example, a geostationary orbit will have a large latency of around 1,000 ms, but a **Low Earth Orbit** (**LEO**) will be much smaller. You should also be able to work out where the ground station is located.

Nslookup

Nslookup is a tool for directly querying a DNS server. **Dig** is another similar tool, but we won't cover it here. Both `ping` and traceroute have performed DNS lookups, but you can do this directly with `nslookup`, which can be very useful. You can call `nslookup` with command-line parameters, but you can also use it interactively. To do this, simply type the name of the tool into a console or a command prompt, as follows:

```
nslookup
```

You will now get a slightly different command prompt from within the program. By default, the DNS name servers of the computer you're on are used, and it bypasses any entries in your local *hosts file*.

Tip

A hosts file can be very useful for testing changes prior to adding them to DNS. It can also be used as a crude blocker for adverts and trackers by setting their addresses to `0.0.0.0`. There are local DNS servers you can run to do this for your whole network. One such project is pi-hole.net , which is based on **dnsmasq**, but which simplifies setting it up and updating the hosts on **Raspberry Pi**.

Enter the hostname of a server to resolve its IP address; for example, type the following:

```
polling.bbc.co.uk
```

The results show that this hostname is a CNAME (an alias for the real Canonical Name) of `polling.bbc.co.uk.edgekey.net`, which resolves to `e3891.g.akamaiedge.net`, and this currently has an IP address of `23.67.87.132`. We can perform a reverse DNS lookup on the IP address by entering it:

```
23.67.87.132
```

We then get the hostname of that machine, conveniently containing the IP address, which is `a23-67-87-132.deploy.static.akamaitechnologies.com`. The domain name is owned by Akamai, which is a **Content Delivery Network (CDN)** used to distribute load.

If you are using a DNS server on your local network, possibly with your router

running dnsmasq, then it may cache results, and give you stale data. You can see more up-to-date information by changing the server to a core one that propagates changes quicker. For example, to use one of Google's public DNS servers, enter the following (but be aware that Google will log all the internet sites you visit if you use this normally):

```
server 8.8.8.8
```

Then run the same query again. Notice how the hostname now resolves to a different IP address. This is a common behavior for CDNs, and the record will change over time, even on the same DNS server. It is often used as a technique to balance network load. Changing DNS servers can also sometimes be used to get around Naïve content filters or location restrictions.

To exit the `nslookup` interactive mode, type `exit` and press **return**. The following image shows the output for the previous commands:

```
C:\Users\James>nslookup
Default Server:  BThomehub.home
Address:  192.168.1.254

> polling.bbc.co.uk
Server:  BThomehub.home
Address:  192.168.1.254

Non-authoritative answer:
Name:    e3891.g.akamaiedge.net
Address:  23.67.87.132
Aliases:  polling.bbc.co.uk
          polling.bbc.co.uk.edgekey.net

> 23.67.87.132
Server:  BThomehub.home
Address:  192.168.1.254

Name:    a23-67-87-132.deploy.static.akamaitechnologies.com
Address:  23.67.87.132

> server 8.8.8.8
Default Server:  google-public-dns-a.google.com
Address:  8.8.8.8

> polling.bbc.co.uk
Server:  google-public-dns-a.google.com
Address:  8.8.8.8

Non-authoritative answer:
Name:    e3891.g.akamaiedge.net
Address:  23.65.37.43
Aliases:  polling.bbc.co.uk
          polling.bbc.co.uk.edgekey.net

> exit
```

Note

IPv4 is the version of IP that you will probably be most familiar with. It uses 32-bit addresses that are usually represented as four dotted decimal octets, such as

`192.168.0.1`. However, we have run out of IPv4 addresses, and the world is (slowly) moving to the new version of IP, called **IPv6**. The address length has increased fourfold, to 128-bit, and is usually represented in hexadecimal such as `2001:0db8:0000:0000:0000:ee00:0032:154d`. Leading zeros can be omitted, like `2001:db8::ee00:32:154d`, to make them easier to write. Localhost loopback (`127.0.0.1`) is now simply `::1` in IPv6.

On Windows, you can use `ping -a x.x.x.x` to do a reverse DNS lookup, and resolve IP addresses to hostnames. On Linux (and on other Unix-like systems such as OS X), this feature is not available, and the `-a` flag serves a different purpose. You will have to use `nslookup`, or dig for reverse DNS on these OSs.

Build your own

You can build your own tools using C#, and the functions you need are provided by .NET. These are built into the full .NET Framework, but for .NET Core, you will need to add the NuGet package `System.Net.Ping` for ping, although the DNS name resolution is built-in. The underlying implementations are platform-specific, but the framework will take care of calling the native code on the OS you're using.

You won't normally need to programmatically resolve a hostname, as most networking commands will do this automatically. However, it can occasionally be useful, especially if you want to perform a reverse DNS lookup, and find the hostname for an IP address.

Tip

The .NET `Dns` class differs from `nslookup`, as it includes entries from the local hosts file rather than just querying a DNS server. It is, therefore, more representative of the IP addresses that other processes are resolving.

To programmatically perform a DNS lookup, follow these steps:

1. Add the `System.Net` namespace.

   ```
   using System.Net;
   ```

2. To resolve a hostname to an IP address, use the following static method. This will return an array of IP addresses, although there will usually only be one.

   ```
   var results = await Dns.GetHostAddressesAsync(host);
   ```

3. To resolve an IP address to a hostname, use the following method instead:

   ```
   var revDns = await Dns.GetHostEntryAsync(result);
   ```

4. If successful, then the hostname will be available as `revDns.HostName`.

To programmatically ping a host, follow these steps:

1. Add the `System.Net.NetworkInformation` namespace.

```
using System.Net.NetworkInformation;
```

2. You can then instantiate a new `Ping` object.

```
var ping = new Ping();
```

3. With this object, you can now ping an IP address or hostname (that will perform a DNS lookup internally) by using the following method:
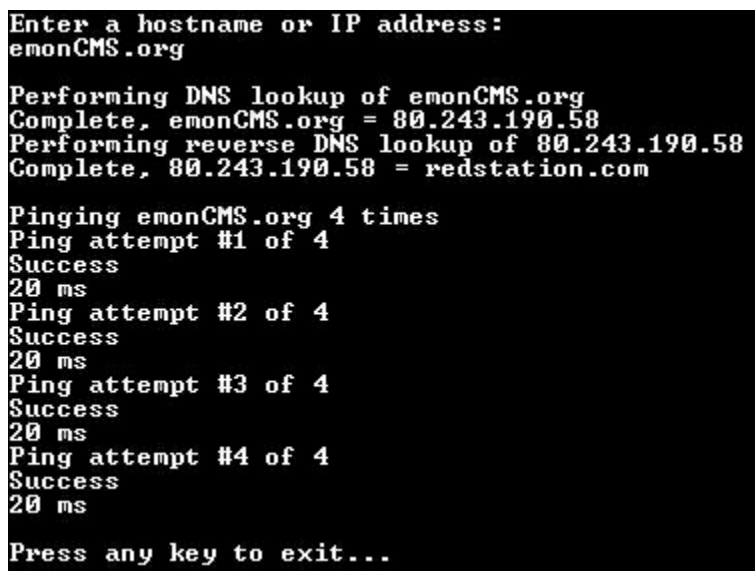
```
var result = await ping.SendPingAsync(host);
```

4. You can get the status of the ping with `result.Status`, and if successful, then you can get the RTT in milliseconds with `result.RoundtripTime`.

Note

The source code for our console application, illustrating how to use the .NET Core `Dns` and `Ping` classes, is available for download along with this book.

A reverse DNS lookup can usually reveal the hosting company being used by a website. Usually, there is only one IP address per hostname, as shown in the following image of our .NET Core console app output:

```
Enter a hostname or IP address:
emonCMS.org

Performing DNS lookup of emonCMS.org
Complete, emonCMS.org = 80.243.190.58
Performing reverse DNS lookup of 80.243.190.58
Complete, 80.243.190.58 = redstation.com

Pinging emonCMS.org 4 times
Ping attempt #1 of 4
Success
20 ms
Ping attempt #2 of 4
Success
20 ms
Ping attempt #3 of 4
Success
20 ms
Ping attempt #4 of 4
Success
20 ms

Press any key to exit...
```

In the preceding screenshot, we can see that emoncms.org is using redstation.com as a host. The low latency suggests that the server is located in the same country as our

computer.

DNS is often used for load balancing. In this case, you will see many IP addresses returned for a single domain name, as shown in the following screenshot:

```
Enter a hostname or IP address:
DuckDuckGo.com

Performing DNS lookup of DuckDuckGo.com
Complete, DuckDuckGo.com = 54.229.105.92
Performing reverse DNS lookup of 54.229.105.92
Complete, 54.229.105.92 = ec2-54-229-105-92.eu-west-1.compute.amazonaws.com

Complete, DuckDuckGo.com = 54.229.105.203
Performing reverse DNS lookup of 54.229.105.203
Complete, 54.229.105.203 = ec2-54-229-105-203.eu-west-1.compute.amazonaws.com

Complete, DuckDuckGo.com = 176.34.131.233
Performing reverse DNS lookup of 176.34.131.233
Complete, 176.34.131.233 = ec2-176-34-131-233.eu-west-1.compute.amazonaws.com

Complete, DuckDuckGo.com = 176.34.155.20
Performing reverse DNS lookup of 176.34.155.20
Complete, 176.34.155.20 = ec2-176-34-155-20.eu-west-1.compute.amazonaws.com

Complete, DuckDuckGo.com = 46.51.197.89
Performing reverse DNS lookup of 46.51.197.89
Complete, 46.51.197.89 = ec2-46-51-197-89.eu-west-1.compute.amazonaws.com

Complete, DuckDuckGo.com = 176.34.135.167
Performing reverse DNS lookup of 176.34.135.167
Complete, 176.34.135.167 = ec2-176-34-135-167.eu-west-1.compute.amazonaws.com

Pinging DuckDuckGo.com 4 times
Ping attempt #1 of 4
Success
27 ms
Ping attempt #2 of 4
Success
27 ms
Ping attempt #3 of 4
Success
27 ms
Ping attempt #4 of 4
Success
27 ms

Press any key to exit...
```

We can see in the preceding screenshot that the privacy-focused search engine **DuckDuckGo** (which doesn't track its users like Google does) is using AWS. DNS is being used to balance the load across various instances—in this case, they're all in the Dublin data center, because that's the closest one. Notice how the ping times are now slightly higher than the UK-based host in the previous example.

It's likely that they're using the AWS DNS service **Route 53** (so named because DNS uses port 53). This can balance load across regions, whereas an ELB (which DuckDuckGo doesn't appear to be using) can only balance inside a region (but both inside and across AZs). Azure offers a similar service called **Traffic Manager** for DNS load balancing.

Solutions

Now that you understand a bit more about the causes of latency-based problems, and how to analyze them, we can demonstrate some potential solutions. The measurements that you have taken using the previously illustrated tools will help you quantify the scale of the problems, and choose the appropriate fixes to be applied.

Batching API requests

Rendering a typical web page may require calls to many different APIs (or DB tables) to gather the data required for it. Due to the style of object-oriented programming encouraged by C# (and many other languages), these API calls are often performed in series. However, if the result of one call does not affect another, then this is suboptimal, and the calls could be performed in parallel. We'll cover DB tables later in this chapter, as there are better approaches for them.

Concurrent calls can be more pertinent if you implement a **microservices** architecture (as opposed to the traditional monolith, or big ball of mud), and have lots of different distributed endpoints. Message queues are sometimes a better choice than HTTP APIs in many cases, perhaps using a publish and subscribe pattern. However, maybe you're not responsible for the API, and are instead integrating with a third party. Indeed, if the API is yours, then you could alter it to provide all the data that you need in one shot anyway.

Consider the example of calling two isolated APIs that have no dependencies on each other. The sequence diagram for this may look something like the following:
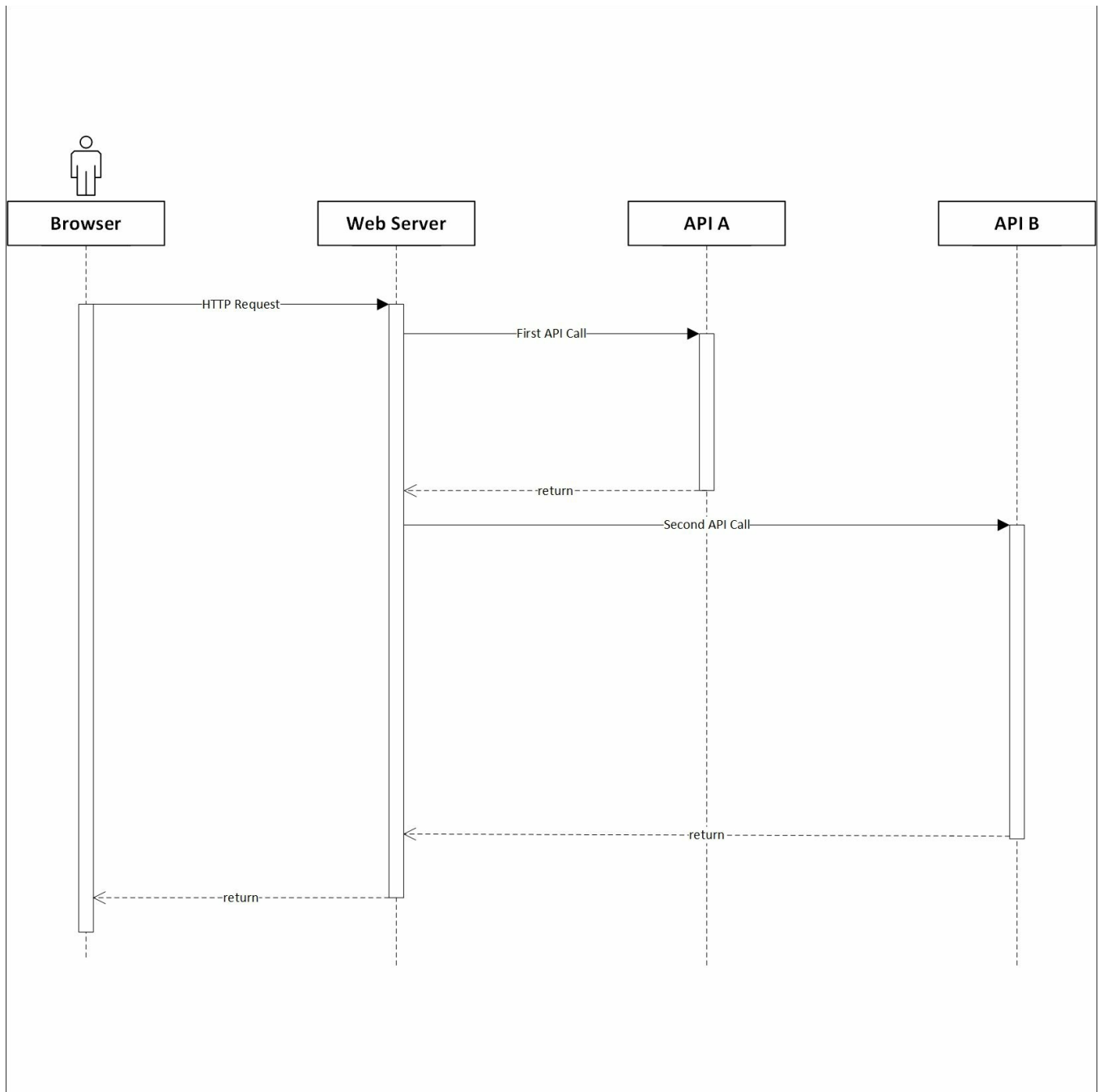
This linear process of calling A, and then calling B when A is done, is simple, as it requires no complex orchestration to wrangle the results of the API calls, but it is slower than necessary. By calling both APIs synchronously and in sequence, we waste time waiting for them to return. A better way of doing this, if we don't require the result of the first API for the request to the second, may be to call them together asynchronously. The sequence diagram for this new flow may look something like the following:
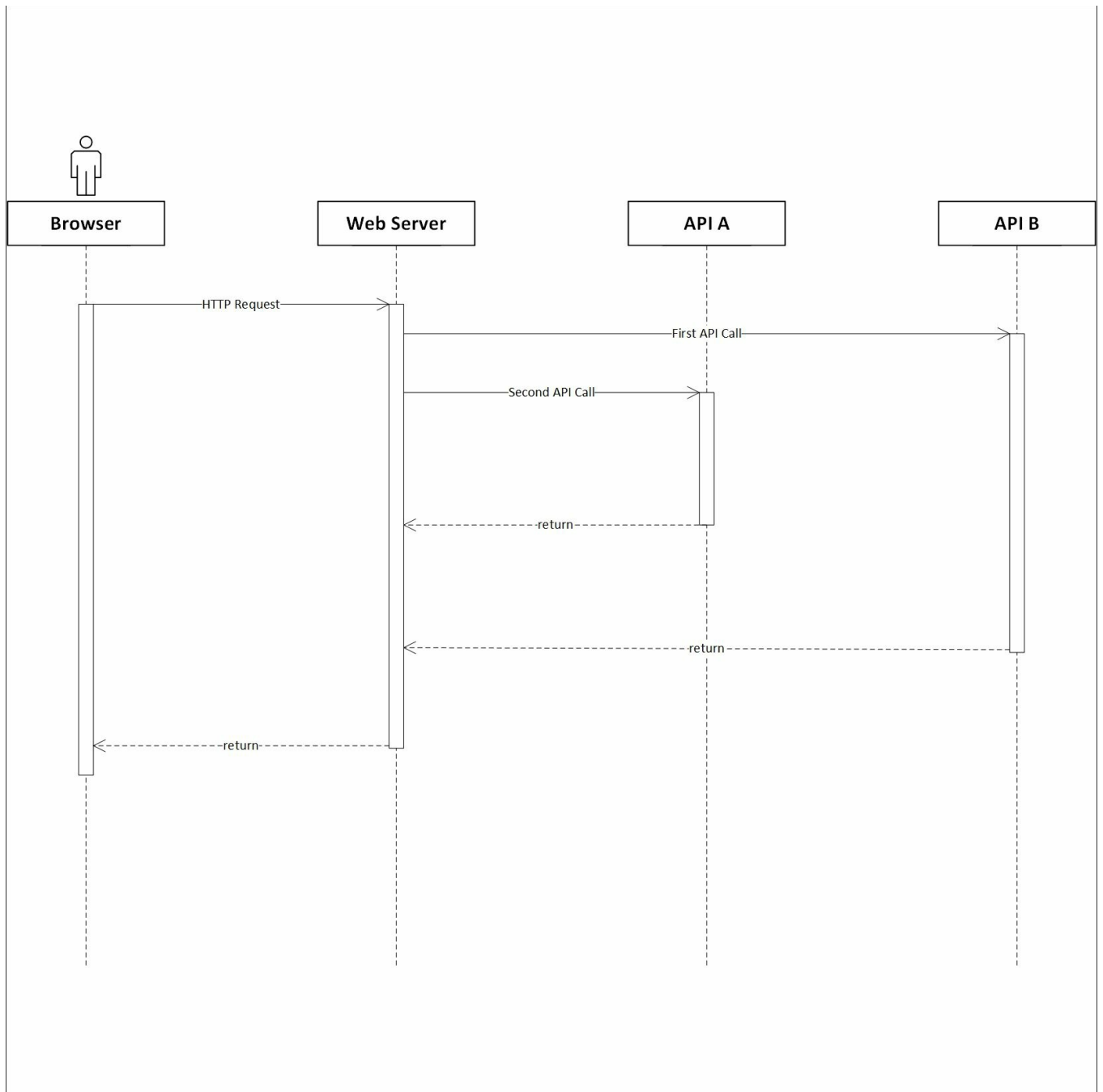
There are two changes in this new sequence, one obvious and the other subtle. We call both APIs asynchronously and in parallel so that they are simultaneously in flight. This first change will have the greatest impact, but there is a smaller tweak that can also help, which is calling the slowest API first.

In the original sequence diagram, we call **API A** and then **API B**, but B is slower. Calling A and B at the same time will have a big impact, but calling B and then A is slightly better. B dominates the timeline, and we will be killing a (relatively) large amount of time waiting for it.

We can use this downtime to call A, as there will be some small fixed overhead in any API method call. Both of these changes combined mean that we are now only waiting for B, and effectively get the call to A for free, in terms of time.

We can illustrate this principle with a simple console application. There are two methods that simulate the APIs, and both are similar. API A has the following code:

```
private static async Task CallApiA()
{
    Thread.Sleep(10);
    await Task.Delay(100);
}
```

`Thread.Sleep` simulates the fixed overhead, and `Task.Delay` simulates waiting for the API call to return. API B takes twice as long to return as API A, but the fixed overhead is the same, and it has the following code:

```
private static async Task CallApiB()
{
    Thread.Sleep(10);
    await Task.Delay(200);
}
```

Now, if we synchronously call the methods in sequence, we discover that all of the delays add up, as expected.

```
CallApiA().Wait();
CallApiB().Wait();
```

These operations take a total of around **332 ms** on average, as there is about 10 ms of additional intrinsic overhead in the method invocation. If we call both methods simultaneously, the time reduces significantly.

```
Task.WaitAll(CallApiA(), CallApiB());
```

The operations now take an average total of **233 ms**. This is good, but we can do better if we swap the order of the methods.

```
Task.WaitAll(CallApiB(), CallApiA());
```

This now takes, on an average, a total of **218 ms**, because we have swallowed the fixed overheads of API A into the time we are waiting for API B.

Note

The full console application which benchmarks these three variants is available for download with this book. It uses the `Stopwatch` class to time the operations, and

averages the results over many runs.

The results of these three different approaches are shown in the following image:

```
Benchmarking over an average of 20 runs
A then B mean elapsed time: 332 ms
A and B mean elapsed time: 233 ms
B and A mean elapsed time: 218 ms
Press any key to exit...
```

As is always the case, measure your results to make sure there really is an improvement. You may find that if you are making many calls to one API, or reusing the same API client, then your requests are processed sequentially, regardless.

This parallelizing of tasks not only works for I/O, but can also be used for computation, as we will discover in the next chapter. For now, we will move on, and take a deeper look at database query optimization.

Efficient DB operations

Although this isn't a book aimed at **Database Administrators** (**DBAs**), it's advantageous for you to appreciate the underlying technology that you're using. As is the case with networking, if you understand the strengths and weaknesses, then you can achieve high performance more easily.

This is additionally relevant for the recent trend of developers doing more work outside of their specialization, particularly in small organizations or startups. You may have heard the buzzwords full-stack developer or **DevOps**. These refer to roles that blur the traditional boundaries, and merge development (both frontend and backend), quality assurance, networking, infrastructure, operations, and database administration into one job. Being a developer today is no longer a case of just programming or shipping code. It helps to know something about the other areas too.

Previously, in Chapter 3, *Fixing Common Performance Problems* , we covered using the micro O/RM Dapper, and how to fix Select N+1 problems. Now, we'll build on that knowledge, and highlight a few more ways that you can consolidate queries and improve the performance of your DB operations.

As detailed in Chapter 2, *Measuring Performance Bottlenecks* , the first step is to profile your application and discover where the issues lie. Only then should you move on to applying solutions to the problem areas.

Database tuning

We're going to focus more on improving poorly performing queries, so we won't go into DB tuning too much. This is a complex topic, and there is a lot you can learn about indexes and how data is stored.

However, if you haven't run the **Database Engine Tuning Advisor** (**DETA**), then this is an easy step to identify if you have any missing indexes. Yet, you should be careful when applying the recommended changes, as there are always downsides and tradeoffs to consider. For example, indexes make retrieving data quicker, but also make it slower to add and update records. They also take up additional space, so it is best not to overdo it. It will depend on your particular business use case: if you wish to optimize upfront, or take the hit later on retrieval.

The first step is to capture a trace with **SQL Server Profiler**. See Chapter 2, *Measuring Performance Bottlenecks* for how to do this, and save the resulting file. This is one of the reasons that the tuning profile is a good choice. You should make sure that the trace is representative of a genuine DB use to get good results. You can

launch the DETA from the same menu as the profiler in SSMS.

You can then load in your trace as a workload, and after processing, it will give you some recommendations. We won't cover the details of how to do this here for space reasons, and it is pretty self-explanatory. There are many good guides available online, if required.

Reporting

A common use case, which can often bring a database to its knees, is reporting. Reports usually run on large amounts of data, and can take considerable time if poorly written.

If you already have a method for retrieving a single item from a DB, then it can be tempting to reuse this for reporting purposes and for calculating metrics inside of the application. However, this is best avoided; reports should generally have their own dedicated queries.

Even better is to have another database dedicated to reporting, which is populated from the main DB or from backups. This way, intensive reports won't affect the performance of the main application at the expense of not including real-time data. This is sometimes known as a **data warehouse** or **data mart**, and the data is occasionally denormalized for simplicity or performance reasons. Populating another DB can also be a convenient way to test your backups, as database backups should always be tested to ensure that they can be restored correctly and contain all of the required data.

Note

In the ensuing examples, we will focus on **Microsoft SQL Server (MS SQL)** and the **Transact-SQL (T-SQL)** dialect of SQL. Other databases, such as PostgreSQL,

are available as well. PostgreSQL has its own dialect called **Procedural Language / PostgreSQL (PL/pgSQL)**, which is similar to **Oracle PL/SQL**. These dialects all support basic SQL commands, but use different syntax for some operations, and contain different or additional features.

The latest version of MS SQL Server is SQL Server 2016, and it's the first edition to support running on Linux in addition to Windows. It is based on the **SQL Azure** service code, so there should now be fewer differences between a self-hosted, on-premise DB and Microsoft's cloud offering.

Aggregates

Aggregate functions are an incredibly useful feature of a RDBMS. You can compute values that summarize many records in the database, and only return the result keeping the source data rows in the DB.

You will be familiar with the `COUNT` aggregate function from earlier in the book, if not before. This gives you the total number of records returned by your query, but without returning them. You may have read advice that `COUNT(1)` performs better than `COUNT(*)`, but this is no longer the case, as SQL Server now optimizes the latter to perform the same as the former.

Tip

By default, SQL Server will return a message detailing the count of the records returned, along with the result of every query. You can turn this off by prefixing your query with the `SET NOCOUNT ON` command, which will save a few bytes in the **Tabular Data Stream** (**TDS**) connection, and increase performance slightly. This is significant only if a **cursor** is being used, which is bad practice anyway for locking reasons. It's good practice to re-enable row count after the query, even though it will be reset outside of the local scope anyway.

In Chapter 3, *Fixing Common Performance Problems*, we solved our Select N+1 problem by joining tables in the DB, and using `COUNT` instead of performing these calculations in our application code. There are many other aggregate functions available, that can be useful in improving performance, especially for reporting purposes.
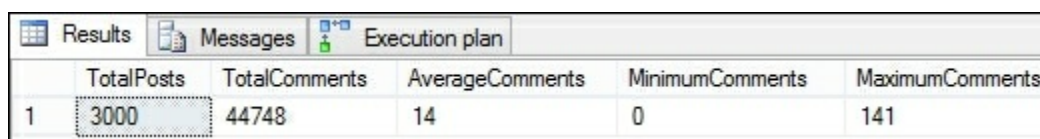
Going back to our earlier example, suppose we now want to find out the total number of blog posts. We also want to find the total number of comments, the average comment count, the lowest number of comments for a single post, and the highest number of comments for a single post.

The first part is easy—just apply COUNT to the posts instead of the comments—€"but the second part is harder. As we already have our list of posts with a comment count for each, it may be tempting to reuse that, and simply work everything out in our C# code.

This would be a bad idea, and the query would perform poorly, especially if the number of posts is high. A better solution would be to use a query such as the following:

```
;WITH PostCommentCount AS(
SELECT
    bp.BlogPostId,
    COUNT(bpc.BlogPostCommentId) 'CommentCount'
FROM BlogPost bp
LEFT JOIN BlogPostComment bpc
    ON bpc.BlogPostId = bp.BlogPostId
GROUP BY bp.BlogPostId
) SELECT
    COUNT(BlogPostId) 'TotalPosts',
    SUM(CommentCount) 'TotalComments',
    AVG(CommentCount) 'AverageComments',
    MIN(CommentCount) 'MinimumComments',
    MAX(CommentCount) 'MaximumComments'
FROM PostCommentCount
```

This query uses a **Common Table Expression** (**CTE**), but you could also use a nested SELECT to embed the first query into the FROM clause of the second. It illustrates a selection of the aggregate functions available, and the results on the test database from before look like the following:

	TotalPosts	TotalComments	AverageComments	MinimumComments	MaximumComments
1	3000	44748	14	0	141
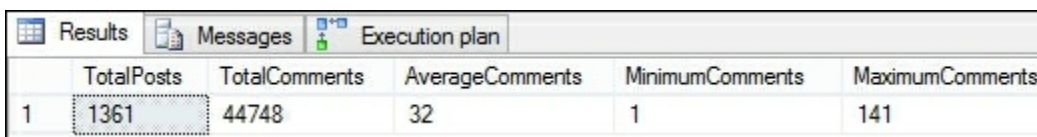
Note

The semicolon at the start is simply a good practice to avoid errors and remove ambiguity. It ensures that any previous command has been terminated, as WITH can be used in other contexts. It isn't required for the preceding example, but might be if it were part of a larger query. CTEs can be very useful tools, especially if you require recursion. However, they are evaluated every time, so you may find that temporary tables perform much better for you, especially if querying one repeatedly in a nested SELECT.

Now suppose that we wish to perform the same query, but only include posts that have at least one comment. In this case, we could use a query like the following:

```sql
;WITH PostCommentCount AS(
SELECT
    bp.BlogPostId,
    COUNT(bpc.BlogPostCommentId) 'CommentCount'
FROM BlogPost bp
LEFT JOIN BlogPostComment bpc
    ON bpc.BlogPostId = bp.BlogPostId
GROUP BY bp.BlogPostId
HAVING COUNT(bpc.BlogPostCommentId) > 0
) SELECT
    COUNT(BlogPostId) 'TotalPosts',
    SUM(CommentCount) 'TotalComments',
    AVG(CommentCount) 'AverageComments',
    MIN(CommentCount) 'MinimumComments',
    MAX(CommentCount) 'MaximumComments'
FROM PostCommentCount
```

We have added a HAVING clause to ensure that we only count posts with more than zero comments. This is similar to a WHERE clause, but for use with a GROUP BY. The query results now look something like the following:

	TotalPosts	TotalComments	AverageComments	MinimumComments	MaximumComments
1	1361	44748	32	1	141

Sampling

Sometimes, you don't need to use all of the data, and can sample it. This technique is particularly applicable to any time-series information that you may wish to graph. In SQL Server, the traditional way to perform random sampling was using the NEWID() method, but this can be slow. For example, consider the following query:

```sql
SELECT TOP 1 PERCENT *
FROM [dbo].[TrainStatus]
ORDER BY NEWID()
```

This query returns exactly 1% of the rows with a random distribution. When run against a table with 1,205,855 entries, it returned 12,059 results in about four seconds, which is slow. A better way may be to use TABLESAMPLE, which is available in any reasonably recent version of SQL Server (2005 onwards), as follows:

```
SELECT *
FROM [dbo].[TrainStatus]
TABLESAMPLE (1 PERCENT)
```

This preceding query is much quicker, and when run against the same data as the previous example, it completes almost instantly. The downside is that it's cruder than the earlier method, and it won't return exactly 1% of the results. It will return roughly 1%, but the value will change every time it is run. For example, running against the same test database, it returned 11,504, 13,441 and 11,427 rows when executing the query three times in a row.

Inserting data

Querying databases may be the most common use case, but you will usually need to put some data in there in the first place. One of the most commonly used features when inserting records into a relational DB is the identity column. This is an auto-incrementing ID that is generated by the database, and doesn't need to be supplied when adding data.

For example, in our `BlogPost` table from earlier, the `BlogPostId` column is created as `INTIDENTITY(1,1)`. This means that it's an integer value starting at one and increasing by one for every new row. You `INSERT` into this table, without specifying `BlogPostId`, like so:

```
INSERT INTO BlogPost (Title, Content)
VALUES ('My Awesome Post', 'Write something witty here...')
```

Identities can be very useful, but you will usually want to know the ID of your newly created record. It is typical to want to store a row, then immediately retrieve it so that the ID can be used for future editing operations. You can do this in one shot with the `OUTPUT` clause, like so;

```
INSERT INTO BlogPost (Title, Content)
OUTPUT INSERTED.BlogPostId
VALUES ('My Awesome Post', 'Write something witty here...')
```

In addition to inserting the row, the ID will be returned.

Note

You may see `SCOPE_IDENTITY()` (or even `@@IDENTITY`) advocated as ways of retrieving the identity, but these are outdated. The recommended way of doing this, on modern versions of SQL Server, is to use `OUTPUT`.

`OUTPUT` works on the `INSERT`, `UPDATE`, `DELETE`, and `MERGE` commands. It even works

when operating on multiple rows at a time, as in this example of bulk inserting two blog posts:

```
INSERT INTO BlogPost (Title, Content)
OUTPUT INSERTED.BlogPostId
VALUES ('My Awesome Post', 'Write something witty here...'),
       ('My Second Awesome Post', 'Try harder this time...')
```

The preceding query will return a result set of two identities, for example, 3003 and 3004. In order to execute these inserts with Dapper, you can use the following method for a single record. First, let us create a blog post object, which we'll hard code here, but which would normally come from user input:

```
var post = new BlogPost
{
    Title = "My Awesome Post",
    Content = "Write something witty here..."
};
```

To insert this one post, and set the ID, you can use the following code. This will execute the insert statement, and return a single integer value.

```
post.BlogPostId = await connection.ExecuteScalarAsync<int>(@"
    INSERT INTO BlogPost (Title, Content)
    OUTPUT INSERTED.BlogPostId
    VALUES (@Title, @Content)",
    post);
```

You can then assign the returned ID to the post, and return that object to the user for editing. There is no need to select the record again, assuming the insert succeeds and no exceptions are thrown.

You can insert multiple records at once with the same SQL by using the execute method. However, the SQL will be executed multiple times, which may be inefficient, and you only get back the number of inserted rows, not their IDs. The following code supplies an array of posts to the same SQL used in the previous example:

```
var numberOfRows = await connection.ExecuteAsync(@"
    INSERT INTO BlogPost (Title, Content)
    OUTPUT INSERTED.BlogPostId
    VALUES (@Title, @Content)",
    new[] { post, post, post });
```

If you want multiple IDs returned, then you will need to use the query method to return a collection of values, as we have done previously. However, it is difficult to make this work for a variable number of records, without dynamically building SQL.

The following code performs a bulk insert, using multiple separate parameters for the query:

```
var ids = await connection.QueryAsync<int>(@"
    INSERT INTO BlogPost (Title, Content)
    OUTPUT INSERTED.BlogPostId
    VALUES (@Title1, @Content1),
           (@Title2, @Content2)", new
    {
        Title1 = post.Title,
        Content1 = post.Content,
        Title2 = post.Title,
        Content2 = post.Content
    });
```

This will perform all the work in a single command, and return an enumerable collection of the identities. However, this code is not very scalable (or even elegant).

Tip

There are many helper methods in the `Dapper.Contrib` package which can assist you with inserting records and other operations. However, they suffer from the same limitations as the examples here, and you can only return a single identity or the number of rows inserted.

GUIDs

Integer identities can be useful to a database internally, but perhaps you shouldn't be using an identifying key externally, especially if you have multiple web servers or expose the IDs to users. An alternative is to use a **Globally Unique Identifier (GUID)**, referred to as `UNIQUEIDENTIFIER` in SQL Server. We have already touched on these, as they are generated by the `NEWID()` function used in the suboptimal sampling example.

GUIDs are used ubiquitously, and are 16 bytes long—four times bigger than an integer. The size of GUIDs means that you are unlikely to get a **unique constraint** conflict when inserting a random GUID into an already populated table.

Note

People sometimes worry about GUID collisions, but the numbers involved are so staggeringly huge that collisions are incredibly unlikely. The GUIDs generated by .NET (using a `COM` function on Windows) are, specifically, **Universally Unique Identifiers (UUIDs)** version 4. These have 122 bits of randomness, so you would need to generate nearly three billion GUIDs before having even half a chance of a collision.

At one new record per second, that would take over 90 billion years. Even at two million GUIDs per second (what an average computer today could produce), it would still take over 45 thousand years (more time than human civilization has been around). If you think that this will be an issue for your software, then you should probably be future-proofing it by using five-digit years, such as 02016, to avoid the *Y10K* (or *YAK* in hexadecimal) problem.

This uniqueness property allows you to generate a GUID in your application code and store it in a distributed database, without worrying about merging data later. It also allows you to expose IDs to users, without caring if someone will enumerate them or try and guess a value. Integer IDs often expose how many records there are in a table, which could be sensitive information depending on your use case.

One thing you have to watch out for with random GUIDs is using them as keys, as opposed to IDs. The difference between these two terms is subtle, and they are often one and the same. The key (**primary** or **clustered**) is what the DB uses to address a record, whereas the ID is what you would use to retrieve one. These can be the same value but they don't have to be. Using a random GUID as a key can cause performance issues with indexing. The randomness may cause fragmentation, which will slow down queries.

You can use sequential GUIDs, for example `NEWSEQUENTIALID()` can be the `DEFAULT` value for a column, but then you lose most of the beneficial qualities of GUIDs that mainly come from the randomness. You now effectively just have a really big integer, and if you're only concerned with the maximum number of rows in a table, and require more than two billion, then a `big int` (twice an `int`, but half a GUID) should suffice.

A good compromise is to have an `int` identity as the primary key, but to use a GUID as the ID (and enforce this with a unique constraint on the column). It's best if you generate the GUID in application code with `Guid.NewGuid()`, as using a default column value in the DB means you have to retrieve the ID after an insert, as shown previously.

A table using this strategy may partially look something like the following image. This is a screen capture of part of a table from the **MembershipReboot** user management and authentication library:
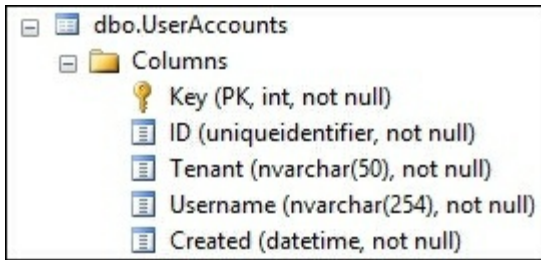
Advanced DB topics

There are many other database performance enhancing techniques that we could cover here, but don't have space for. We'll briefly mention some additional topics in case you want to look into them further.

A common requirement when saving data is to insert a record if it doesn't yet exist, or to update an existing record if there is already a row containing that ID. This is sometimes referred to as an **upsert**, which is a portmanteau word combining update and insert.

You can use the `MERGE` command, to do this in one operation and not worry about choosing between `UPDATE` or `INSERT` (wrapped in a **transaction** inside a **stored procedure**). However, `MERGE` does not perform as well as `UPDATE` or `INSERT`, so always test the effects for your use case, and use it sparingly.

Stored procedures, transactions, and **locking** are all big subjects on their own. They are important to understand, and not only from a performance perspective. We can't fit all of these in here, but we will touch upon maintaining stored procedures later in the book.

Some other techniques you could look into are **cross-database joins**, which can save you from querying multiple DBs. There is also the practice of **denormalizing** your data, which involves flattening your relational records ahead of time to provide a single row rather than needing to join across many tables for a query. We briefly mentioned this in relation to data warehouses earlier.

Note

You can find lots of useful SQL documentation and information on MSDN (msdn.microsoft.com) and TechNet (technet.microsoft.com). Additionally, there are many good books on advanced SQL techniques.

Finally, a note on housekeeping. It's important to have a plan for managing the size of

a database. You can't simply keep inserting data into a table, and hope it will continue to perform well forever. At some point, it will get so large that even deleting records from it will cause your DB to grind to a crawl. You should know ahead of time how you will remove or archive rows, and it is better to do this regularly in small batches.

Simulation and testing

To wrap up this chapter, let's reiterate the importance of being able to test your application on realistic infrastructure. Your test environments should be as live-like as possible. If you don't test on equivalent DBs and networks, then you may get a nasty surprise come deployment time.

When using a cloud hosting provider (and if you automate your server builds), then this is easy: you can simply Spin-up a staging system that matches production. You don't have to provision it to the exact same scale as long as all the parts are there and in the same place. To reduce costs further, you only need to keep it around for as long as your test.

Alternatively, you could create a new live environment, deploy and test it, then switch over, and destroy or reuse the old live environment. This swapping technique is known as **blue-green deployment**. Another option is to deploy new code behind a **feature switch**, which allows you to toggle the feature at runtime, and only for some users. Not only does this allow you to functionally verify features with test users, you can also gradually roll out a new feature, and monitor the performance impact as you do so. We will cover both of these techniques in Chapter 9, *Monitoring Performance Regressions* .

Summary

We've covered a lot in this chapter, and it should hopefully be clear that being a high performance developer requires more than simply being a capable programmer. There are many externalities to consider around your code.

You now understand the basics of I/O, and the physical causes of the intrinsic limitations present. You can analyze and diagnose how a network logically fits together. We have covered how to make better use of the network to reduce overall latency, which is important as more I/O operations now happen over a network. You have also seen how to make better use of a database, and how to do more with fewer operations.

In the next chapter, we will dig into code execution in detail, and show you how to speed up your C#. We'll see how .NET Core and ASP.NET Core perform so much better than the previous versions, and how you can take advantage of these improvements.

Chapter 6. Understanding Code Execution and Asynchronous Operations

This chapter covers solving performance problems in the execution of your code, which is not normally the first location that speed issues occur, but can be a good additional optimization. We'll discuss the areas where performance is needed, and where it is okay (or even required) to be slow. The merits of various data structures will be compared from the standard built-in generic collections to the more exotic. We will demonstrate how to compute operations in parallel, and how to take advantage of extra instruction sets that your CPU may provide. We'll dive into the internals of ASP.NET Core and .NET Core to highlight the improvements that you should be aware of.

The topics covered in this chapter include the following:

- .NET Core and the native tool chain
- **Common Language Runtime** (**CLR**) services such as GC, JIT, and NGen
- ASP.NET Core and the Kestrel web server
- Generic collections and Bloom filters
- Serialization and data representation formats
- Relative performance of hashing functions
- Parallelization (SIMD, TPL, and PLINQ)
- Poorly performing practices to avoid

You will learn how to compute results in parallel and combine the outputs at the end. This includes how to avoid incorrect ways of doing this, which can make things worse. You'll also learn how to reduce server utilization, and to choose the most appropriate data structures in order to process information efficiently for your specific situation.

Getting started with the core projects

There are many benefits of using .NET Core and ASP.NET Core over the old full versions of the frameworks. The main enhancements are the open source development and cross platform support, but there are also significant performance improvements. Open development is important and the source code is not only available, the development work happens in the open on GitHub. The community is encouraged to make contributions and these may be merged in upstream if they pass a code review; the flow isn't just one way. This has led to increased performance and additional platform support coming from outside of Microsoft. If you find a bug in a framework, you can now fix it rather than work around the problem and hope for a patch.

The multiple projects that make up the frameworks are split across two organizations on GitHub. One of the driving principles has been to split the frameworks up into modules, so you can just take what you need rather than the whole monolithic installation. The lower level framework, .NET Core, can be found, along with other projects, under github.com/dotnet. The higher level web application framework, ASP.NET Core, can be found under github.com/aspnet. The reason for this split is that .NET Core is managed by the *.NET Foundation* which is an independent organization, although most of the members are Microsoft staff. ASP.NET Core is managed directly by Microsoft.

Let's have a quick look at some of the various .NET Core repositories and how they fit together.

.NET Core

There are a couple of projects that form the main parts of .NET Core and these are CoreCLR and **CoreFX**. CoreCLR contains the .NET Core CLR and the base core library, `mscorlib`. The CLR is the virtual machine that runs the .NET code. CoreCLR contains a **just-in-time (JIT)** compiler, **Garbage Collector** (GC), and also base types and classes in `mscorlib`. CoreFX includes the foundational libraries and sits on top of CoreCLR. This includes most built-in components that aren't simple types.

Note

You may be familiar with the `<gcServer>` element used to set a GC mode, more suitable for server use, with shorter pauses. You can read more about the original version at msdn.microsoft.com/en-us/library/ms229357 . In .NET Core you can set the `COMplus_gcServer` environment variable to `1` or set `System.GC.Server` to `true` as a runtime option in a JSON runtime configuration file.

It's worth highlighting another project called **CoreRT**, which is a runtime that allows **ahead-of-time (AOT)** compilation, instead of the JIT of CoreCLR. This means that you can compile your C# code to native machine code, and run it without any dependencies. You end up with a single (statically linked) binary and don't need .NET installed, which is very similar to how **Go** operates. With the Roslyn compiler, you no longer always need to compile to **Common Intermediate Language** (**CIL**) bytecode then have the CLR JIT compile to native instructions at runtime with **RyuJIT**.

This native tool chain allows for performance improvements, as the compilation does not have to happen quickly, in real time, and can be further optimized. For example, a build can be tuned for execution speed at the expense of compilation speed. It's conceptually similar to the **Native Image Generator** (**NGen**) tool, which has been in the full .NET Framework since its inception.

ASP.NET Core

ASP.NET Core runs on top of .NET Core, although it can also run on the full .NET Framework. We will only cover running on .NET Core, as this performs better and ensures enhanced platform support. There are many projects that make up ASP.NET Core and it's worth briefly mentioning some of them.

Tip

It's useful to not include any framework references to .NET 4.5/4.6 in your project, so that you can be sure you're only using .NET Core and don't accidentally reference any dependencies that are not yet supported.

ASP.NET Core includes **Model View Controller** (**MVC**), **Web API,** and **Web Pages** (a way to make simple pages with **Razor**, similar to PHP, and the spiritual successor to classic ASP). These features are all merged together, so you don't need to think of MVC and Web API as separate frameworks anymore. There are many other projects and repositories, including **EF Core**, but the one we will highlight here is the Kestrel HTTP server.

Kestrel

Kestrel is a new web server for ASP.NET Core and it performs brilliantly. It's based on **libuv**, which is an asynchronous I/O abstraction and support library that also runs below Node.js. Kestrel is blazingly fast, and the benchmarks are very impressive. However, it is pretty basic, and for production hosting, you should put it behind a reverse proxy, so that you can use caching and other features to service a greater number of users. You can use many HTTP or proxy servers for this purpose such as IIS, NGINX, Apache, or HAProxy.

Tip

You should be careful with your configuration if using NGINX as a reverse proxy, as by default, it will retry `POST` requests if the response from your web server times out. This could result in duplicate operations being performed, especially if used as a load balancer across multiple backend HTTP servers.

Data structures

Data structures are objects that you can use to store the information you're working on. Choosing the best implementation for how your data is used can have dramatic effects on the speed of execution. Here, we'll discuss some common and some more interesting data structures that you might like to make use of.

As this is a book about web application programming and not one on complex algorithm implementation or micro-optimization, we won't be going into a huge amount of detail on data structures and algorithms. As briefly mentioned in the introduction, other factors can dwarf the execution speed in a web context, and we assume you're not writing a game engine. However, good algorithms can help speed up applications, so, if you are working in an area where execution performance is important, you should read more about them.

We're more interested in the performance of the system as a whole, not necessarily the speed the code runs at. It is often more important to consider what your code expresses (and how this affects other systems) than how it executes. Nevertheless, it is still important to choose the right tool for the job, and you don't want to have unnecessarily slow code. Just be careful of over-optimizing when it already executes fast enough, and above all, try to keep it readable.

Lists

A .NET `List<T>` is a staple of programming in C#. It's type safe, so you don't have to bother with casting or boxing. You specify your generic type and can be sure that only objects of that class (or primitive) can be in your list. Lists implement the standard list and enumerable interfaces (`IList` and `IEnumerable`), although you get more methods on the concrete list implementation, for example, adding a range. You can also use **Language-Integrated Query** (**LINQ**) expressions to easily query them, which is trivial when using **fluent lambda functions**. However, although LINQ is often an elegant solution to a problem, it can lead to performance issues, as it is not always optimized.

A list is really just an enhanced, one-dimensional, array. In fact, it uses an array internally to store the data. In some cases, it may be better to directly use an array for performance. Arrays can occasionally be a better choice, if you know exactly how much data there will be, and you need to iterate through it inside a tight loop. They can also be useful if you need more dimensions, or plan to consume all of the data in the array.

You should be careful with arrays; use them sparingly and only if benchmarking shows a performance problem when iterating over a list. Although they can be faster, they also make parallel programming and distributed architecture more difficult, which is where significant performance improvements can be made.

Modern high performance means scaling out to many cores and multiple machines. This is easier with an **immutable** state that doesn't change, which is easier to enforce with higher level abstractions than with arrays. For example, you can help enforce a read-only list with an interface.

If you are inserting or removing a lot of items in the middle of a large list, then it may be better to use the `LinkedList<T>` class. This has different performance characteristics to a list, as it isn't really a list–it's more of a chain. It may perform better than a list for some specialized cases, but, in most common cases, it will be slower. For example, access by index is quick with a list (as it is array backed), but slow with a linked list (as you will have to walk the chain).

It is normally best to initially focus on the what and why of your code rather than the how. LINQ is very good for this, as you simply declare your intentions, and don't need to worry about the implementation details and loops. It is a bad idea to optimize prematurely, so do this only if testing shows a performance problem. In most common cases, a list is the correct choice unless you need to select only a few values from a

large set, in which case, dictionaries can perform better.

Dictionaries

Dictionaries are similar to lists, but excel at quickly retrieving a specific value with a key. Internally, they are implemented with a hash table. There's the legacy `Hashtable` class (in the `System.Collections.NonGeneric` package) but this is not type safe, whereas `Dictionary<T>` is a generic type, so you probably shouldn't use `Hashtable` unless you're porting old code to .NET Core. The same applies to `ArrayList`, which is the legacy, non-generic version of `List`.

A dictionary can look up a value with a key very quickly, whereas a list would need to go through all the values until the key was found. You can, however, still enumerate a dictionary as it is ordered, although this isn't really how it is intended to be used. If you don't need ordering, then you can use a `HashSet`. There are sorted versions of all of these data structures, and you can again use read-only interfaces to make them difficult to mutate.

Collection benchmarks

Accurate benchmarking is hard and there are lots of things which can skew your results. Compilers are very clever and will optimize, which can make trivial benchmarks less valuable. The compiler may output very similar code for different implementations.

What you put in your data structures will also have a large effect on their performance. It's usually best to test or profile your actual application and not optimize prematurely. Readability of code is very valuable and shouldn't be sacrificed for runtime efficiency unless there is a significant performance problem (or if it's already unreadable).

There are benchmarking frameworks that you can use to help with your testing, such as BenchmarkDotNet which is available at github.com/PerfDotNet/BenchmarkDotNet . However, these can be an overkill, and are sometimes tricky to set up, especially on .NET Core. Other options include **Simple Speed Tester** (which you can read more about at theburningmonk.github.io/SimpleSpeedTester) and **MiniBench** (available from github.com/minibench).

We'll perform some simple benchmarks to show how you might go about this. However, don't assume that the conclusions drawn here will always hold true, so test for your situation. First, we will define a simple function to run our tests:

```
private static long RunTest(Func<double> func, int runs = 1000)
{
    var s = Stopwatch.StartNew();
    for (int j = 0; j < runs; j++)
    {
        func();
    }
    s.Stop();
    return s.ElapsedMilliseconds;
}
```

We use `Stopwatch` here as using `DateTime` can cause problems, even when using UTC, as the time could change during the test and the resolution isn't high enough. We also need to perform many runs to get an accurate result. We then define our data structures to test and prepopulate them with random data.

```
var rng = new Random();
var elements = 100000;
var randomDic = new Dictionary<int, double>();
for (int i = 0; i < elements; i++)
```

```
{
    randomDic.Add(i, rng.NextDouble());
}
var randomList = randomDic.ToList();
var randomArray = randomList.ToArray();
```

We now have an array, list, and dictionary containing the same set of 100,000 key/value pairs. Next, we can perform some tests on them to see what structure performs best in various situations.

```
var afems = RunTest(() =>
{
    var sum = 0d;
    foreach (var element in randomArray)
    {
        sum += element.Value;
    }
    return sum;
}, runs);
```

The preceding code times how long it takes to iterate over an array in a `foreach` loop and sums up the double precision floating point values. We can then compare this to other structures and the different ways of accessing them. For example, iterating a dictionary in a `for` loop is done as follows:

```
var dfms = RunTest(() =>
{
    var sum = 0d;
    for (int i = 0; i < randomDic.Count; i++)
    {
        sum += randomDic[i];
    }
    return sum;
}, runs);
```

This performs very badly, as it is not how dictionaries are supposed to be used. Dictionaries excel at extracting a record by its key very quickly. So quickly in fact that you will need to run the test many more times to get a realistic value.
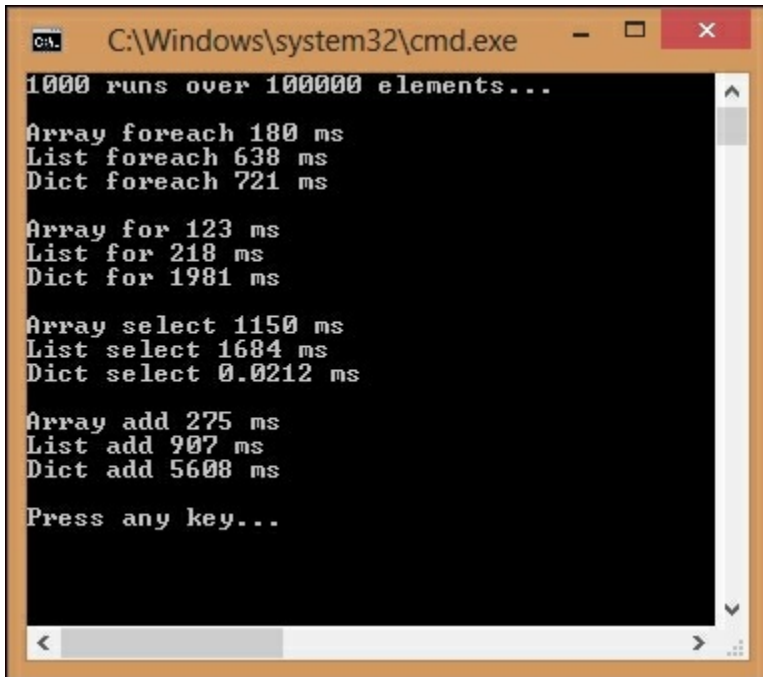
```
var lastKey = randomList.Last().Key;
var dsms = RunTest(() =>
{
    double result;
    if (randomDic.TryGetValue(lastKey, out result))
    {
        return result;
    }
    return 0d;
}, runs * 10000);
Console.WriteLine($"Dict select {(double)dsms / 10000} ms");
```

Getting a value from a dictionary with `TryGetValue` is extremely quick. You need to pass the variable to be set into the method as an `out` parameter. You can see if this was successful and if the item was in the dictionary by testing the Boolean value returned by the method.

Conversely, adding items to a dictionary one by one can be slow, so it all depends on what you're optimizing for. The following image shows the output of a very simple console application that tests various combinatorial permutations of data structures and their uses:

These results, shown in the preceding image, are informative, but you should be skeptical, as many things can skew the output, for example, the ordering of the tests. If the gap is small, then there is probably not much to choose between two variants, but you can clearly see the big differences.

Tip

To get more realistic results, be sure to compile in release mode and run without debugging. The absolute results will depend on your machine and architecture, but the relative measures should be useful for comparisons. You may get higher performance if compiling to a single executable binary with the .NET native tool chain or with a different release of the .NET Core framework.

The main lesson here is to measure for your specific application and choose the most appropriate data structure for the work that you are performing. The collections in the

standard library should serve you well for most purposes, and there are others that are not covered here, which can sometimes be useful, such as a `Queue` or `Stack`.

You can find more information about the built-in collections and data structures on MSDN (msdn.microsoft.com/en-us/library/system.collections.generic). You can also read about them on the .NET Core documentation site on the GitHub pages (dotnet.github.io/docs/essentials/collections/Collections-and-Data-Structures.html).

However, there are some rarer data structures, not in the standard collection, that you may occasionally wish to use. We'll show an example of one of these now.

Bloom filters

Bloom filters are an interesting data structure that can increase performance for certain use cases. They use multiple overlaid hashing functions and can quickly tell you if an item definitely does not exist in a set. However, they can't say with certainty if an item exists, only that it is likely to. They are useful as a pre-filter, so that you can avoid performing a lookup, because you know for sure that the item won't be there.

The following image shows how a Bloom filter works. **A**, **B**, and **C** have been hashed and inserted into the filter. **D** is hashed to check if it is in the filter but, as it maps to a zero bit, we know that it is not in there.
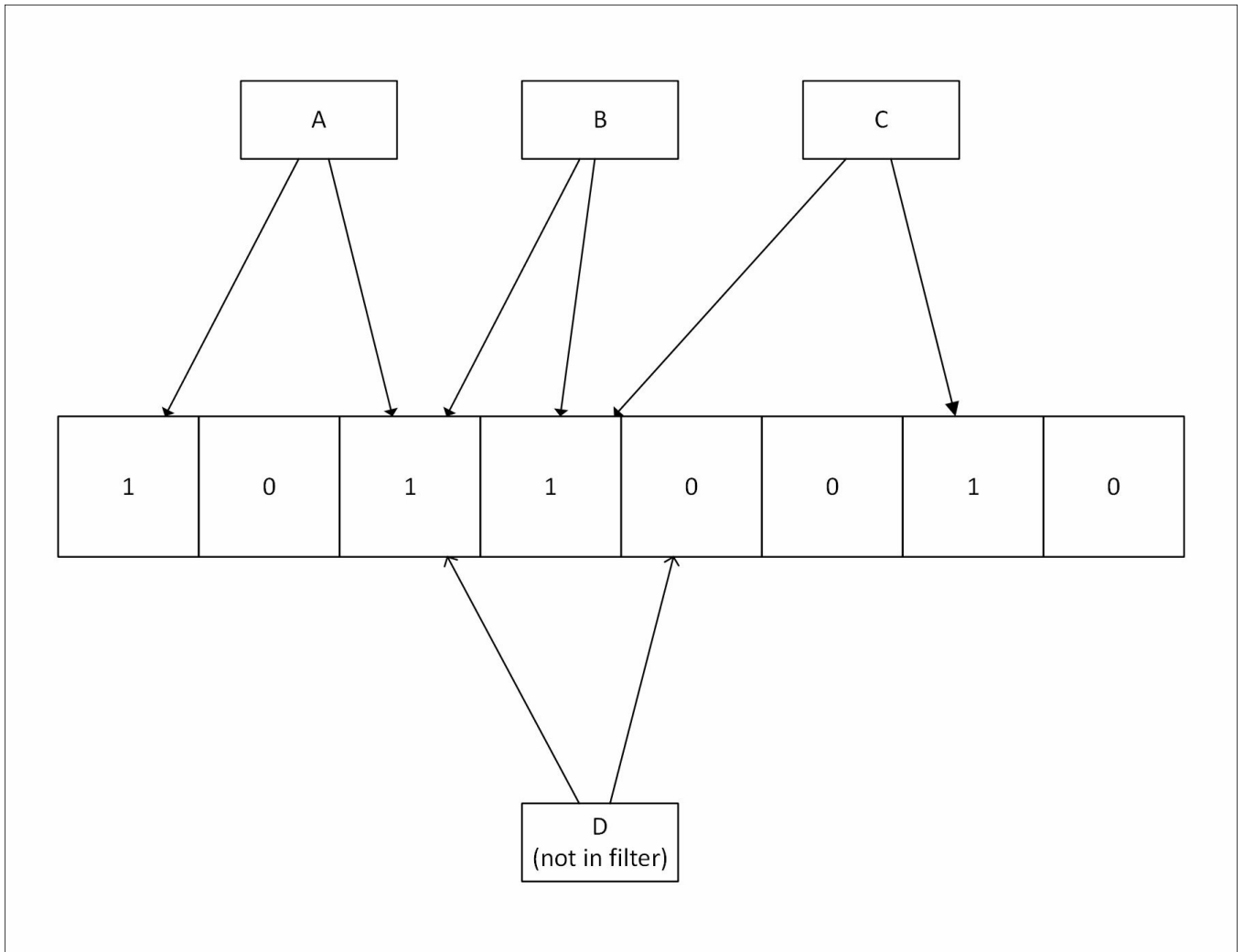
Bloom filters are much smaller than holding all the data or even a list of hashes for each item in the set. They can also be much quicker, as the lookup time is constant for any size of set. This constant time can be lower than the time to look up an item in a large list or dictionary, especially if it is on the file system or in a remote database.

An example application for a Bloom filter could be a local DNS server, which has a list of domains to override, but forwards most requests to an upstream server. If the list of custom domains is large, then it may be advantageous to create a Bloom filter from the entries and hold this in memory.

When a request comes in, it is checked against the filter, and if it doesn't exist, then the request is forwarded to the upstream server. If the entry does exist in the filter, then the local hosts file is consulted; if the entry is there, its value is used. There is a small chance that the entry will not be in the list, even if the filter thinks it is. In this case, when it isn't found, the request is forwarded, but this approach still avoids the need to consult the list for every request.

Another example of using a Bloom filter is in a caching node, perhaps as part of a proxy or CDN. You wouldn't want to cache resources that are only requested once, but how can you tell when the second request occurs if you haven't cached it? If you add the request to a Bloom filter, you can easily tell when the second request occurs and then cache the resource.

They are also used in some databases to avoid expensive disk operations and in **Cache Digests**, which allow an agent to declare the contents of its cache. HTTP/2 may support Cache Digests in the future, but this will probably use **Golomb-coded sets** (**GCS**), which are similar to Bloom filters, but smaller, at the expense of slower queries.

There is an open source implementation of a Bloom filter in .NET available at bloomfilter.codeplex.com among others. You should test the performance for yourself to make sure they offer an improvement.

Hashing and checksums

Hashing is an important concept, which is often used to ensure data integrity or lookup values quickly and so it is optimized to be fast. This is why general hashing functions should not be used on their own to securely store passwords. If the algorithm is quick, then the password can be guessed in a reasonably short amount of time. Hashing algorithms vary in their complexity, speed of execution, output length, and collision rate.

A very basic error detection algorithm is called a **parity check**. This adds a single bit to a block of data and is rarely used directly in programming. It is, however, extensively used at the hardware level, as it is very quick. Yet, it may miss many errors where there are an even number of corruptions.

A **Cyclic Redundancy Check** (**CRC**) is a slightly more complex error detecting algorithm. The **CRC-32** (also written **CRC32**) version is commonly used in software, particularly in compression formats, as a **checksum**.

Tip

You may be familiar with the built-in support for hash codes in .NET (with the `GetHashCode` method on all objects), but you should be very careful with this. The only function of this method is to assist with picking buckets in data structures that use hash tables internally, such as a dictionary, and also in some LINQ operations. It is not suitable as a checksum or key, because it is not cryptographically secure and it varies across frameworks, processes, and time.

You may have used the **Message Digest 5** (**MD5**) algorithm in the past, but today its use is strongly discouraged. The security of MD5 is heavily compromised and collisions can be produced easily. Since it is very quick, it may have some non-secure uses, such as non-malicious error checking but there are better algorithms that are fast enough.

If you need a strong but quick hashing function, then the **Secure Hash Algorithm** (**SHA**) family is a good choice. However, **SHA-1** is not considered future proof, so for a new code, **SHA-256** is generally a better choice.

When signing messages, you should use a dedicated **Message Authentication Code** (**MAC**), such as a **Hash-based MAC** (**HMAC**), which avoids vulnerabilities in a single pass of the hashing function. A good option is the `HMACSHA256` class built into .NET. Various APIs, such as some of the AWS REST APIs, use **HMAC-SHA256** to authenticate requests. This ensures that, even if the request is performed over an

unencrypted HTTP channel, the API key can't be intercepted and recovered.

As briefly mentioned in Chapter 1 , *Why Performance Is a Feature* , password hashing is a special case and general purpose hashing algorithms are not suitable for it–they are too fast. A good choice is **Password-Based Key Derivation Function 2** (**PBKDF2**), which we used as an example in Chapter 2 , *Measuring Performance Bottlenecks* . PBKDF2 is a particularly popular choice for .NET, as it is built into the framework, and so, the implementation is more likely to be correct. It has been built against an RFC and reviewed by Microsoft, which you can't say for any random piece of code found online. For example, you could download an implementation of **bcrypt** for .NET, but you have to trust that it was coded correctly or verify it yourself.

Hashing benchmarks

Let's do some simple benchmarking of various hash functions to see how they
compare performance wise. In the following code snippet, we define a method for
running our tests, similar to the one for the previous collection benchmarks, but taking
an `Action` parameter, rather than a `Func<double>`, as we don't want to return a
value:

```
private static long RunTest(Action func, int runs = 1000)
{
    var s = Stopwatch.StartNew();
    for (int j = 0; j < runs; j++)
    {
        func();
    }
    s.Stop();
    return s.ElapsedMilliseconds;
}
```

We include the following using statement:

```
using System.Security.Cryptography;
```

Next, we define a short private constant string (`hashingData`) to hash in the class
and get the bytes for it in an 8-bit Unicode (`UTF8`) format.

```
var smallBytes = Encoding.UTF8.GetBytes(hashingData);
```

We also want to get a larger block of bytes to hash to see how it compares
performance wise. For this, we use a cryptographically secure random number
generator.

```
var largeBytes = new byte[smallBytes.Length * 100];
var rng = RandomNumberGenerator.Create();
rng.GetBytes(largeBytes);
```

We need a `key` for some of our functions, so we use the same technique to generate
this.

```
var key = new byte[256];
var rng = RandomNumberGenerator.Create();
rng.GetBytes(key);
```

Next, we create a sorted list of the algorithms to test and execute the tests for each
one:

```
var algos = new SortedList<string, HashAlgorithm>
{
```

```
    {"1.          MD5", MD5.Create()},
    {"2.        SHA-1", SHA1.Create()},
    {"3.      SHA-256", SHA256.Create()},
    {"4.   HMAC SHA-1", new HMACSHA1(key)},
    {"5. HMAC SHA-256", new HMACSHA256(key)},
};
foreach (var algo in algos)
{
    HashAlgorithmTest(algo);
}
```

Our test method runs the following tests on each algorithm. They all inherit from `HashAlgorithm`, so we can run the `ComputeHash` method on each of them for the small and large byte arrays.

```
var smallTimeMs = RunTest(() =>
{
    algo.Value.ComputeHash(smallBytes);
}, runs);
var largeTimeMs = RunTest(() =>
{
    algo.Value.ComputeHash(largeBytes);
}, runs);
```

We then calculate the average (mean) time for both sizes. We cast the long integer to a double precision floating point number so that we can represent small values between one and zero.

```
var avgSmallTimeMs = (double)smallTimeMs / runs;
var avgLargeTimeMS = (double)largeTimeMs / runs;
```

The preceding method then outputs these mean times to the console. We need to test PBKDF2 separately, as it doesn't inherit from `HashAlgorithm`.
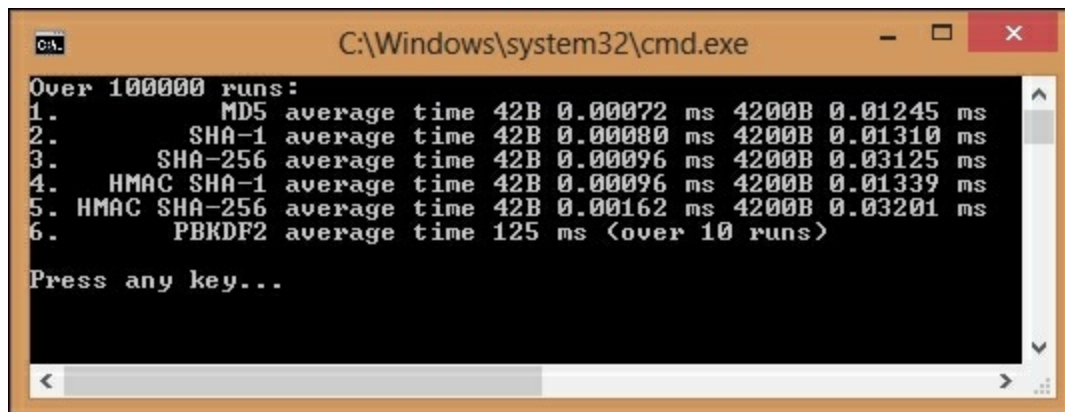
```
var slowRuns = 10;
var pbkdf2 = new Rfc2898DeriveBytes(hashingData, key, 10000);
var pbkdf2Ms = RunTest(() =>
{
    pbkdf2.GetBytes(256);
}, slowRuns);
```

PBKDF2 is so slow that it would take a considerable amount of time to perform 100,000 runs (this is the point of using it). Internally, this RFC2898 implementation of the key-stretching algorithm runs HMAC SHA-1 10,000 times. The default is 1,000, but due to the computing power available today, it is recommended to set this to at least an order of magnitude higher. For example, **Wi-Fi Protected Access II (WPA2)** uses 4,096 rounds of iterations to produce a 256-bit key with the **Service Set Identifier (SSID)** as the **salt**.

The output will look something like the following:

From the preceding output, you can see that the time taken for one hash varies from about 720 nanoseconds for a small MD5 to 32 microseconds for a large HMAC SHA-256 and **125** milliseconds for a small PBKDF2 with typical parameters.

Benchmarking results can vary dramatically, so you shouldn't read too much into absolute values. For example, the output from the BenchmarkDotNet tool comparing MD5 and SHA-256 on the same machine looks like this:

```
// ***** BenchmarkRunner: Finish  *****

// * Export *
  Algo_Md5UsSha256-report.csv
  Algo_Md5UsSha256-report-stackoverflow.md
  Algo_Md5UsSha256-report-default.md
  Algo_Md5UsSha256-report-github.md
  Algo_Md5UsSha256-report.txt
  Algo_Md5UsSha256-measurements.csv
  BuildPlots.R
  Algo_Md5UsSha256-report.html

// * Detailed results *
Algo_Md5UsSha256_Md5
Mean = 31.2803 us, StdError = 0.0880 us (0.28%); N = 20, StdDev = 0.3936 us
Min = 30.5574 us, Q1 = 31.0145 us, Median = 31.2130 us, Q3 = 31.6598 us, Max = 31.9157 us
IQR = 0.6453 us, LowerFence = 30.0466 us, UpperFence = 32.6277 us
ConfidenceInterval = [31.1078 us; 31.4528 us] (CI 95%)


Algo_Md5UsSha256_Sha256
Mean = 201.0418 us, StdError = 1.5320 us (0.76%); N = 33, StdDev = 8.8008 us
Min = 186.5222 us, Q1 = 193.0558 us, Median = 199.2981 us, Q3 = 208.7261 us, Max = 221.1175 us
IQR = 15.6703 us, LowerFence = 169.5503 us, UpperFence = 232.2315 us
ConfidenceInterval = [198.0391 us; 204.0446 us] (CI 95%)


Total time: 00:00:31 (31.33 sec)

// * Summary *
BenchmarkDotNet-Dev=v0.9.3.0+
OS=Microsoft Windows NT 6.2.9200.0
Processor=Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz, ProcessorCount=4
Frequency=1656392 ticks, Resolution=603.7218 ns
HostCLR=MS.NET 4.0.30319.42000, Arch=64-bit DEBUG

Type=Algo_Md5UsSha256   Mode=Throughput

 Method |      Median |     StdDev |
------- |------------ |----------- |
    Md5 |  31.2130 us |  0.3936 us |
 Sha256 | 199.2981 us |  8.8008 us |

// * Warnings *
Benchmark was built in DEBUG configuration. Please, build it in RELEASE.

// ***** BenchmarkRunner: End *****

Global total time: 00:00:32 (32 sec)
```

You can see in the last image that the results are different to our homemade
benchmark. However, this uses the full .NET Framework, calculates median rather
than mean for the average time, and runs in debug mode (which it helpfully warns us
of) among other things.

A faster machine will have a higher throughput (as can be seen in the
BenchmarkDotNet README.md on GitHub). Dedicated hardware such as **Graphics
Processing Units (GPUs)**, **Field-Programmable Gate Arrays (FPGAs)**, and
Application-Specific Integrated Circuits (ASICs) can be much faster. These tend to
be used in the mining of bitcoin (and other crypto currencies), as these are based on
hashing as a proof-of-work. Bitcoin uses SHA-256, but other currencies use different
hashing algorithms.

The same algorithms form the basis of TLS, so faster hardware can handle a greater
number of secure connections. As another example, Google built a custom ASIC
called a **Tensor Processing Unit (TPU)** to accelerate their machine learning cloud

services.

Other benchmarking samples are available in BenchmarkDotNet, and when you first run it, you will be presented with the following menu:

```
Available Benchmarks:
  #0   Algo_BitCount
  #1   Algo_Md5VsSha256
  #2   Algo_MostSignificantBit
  #3   Cpu_Atomics
  #4   Cpu_BranchPerdictor
  #5   Cpu_Ilp_Inc
  #6   Cpu_Ilp_Max
  #7   Cpu_Ilp_RyuJit
  #8   Cpu_Ilp_VsBce
  #9   Cpu_MatrixMultiplication
  #10  Framework_DateTime
  #11  Framework_DictionaryVsIDictionary
  #12  Framework_SelectVsConvertAll
  #13  Framework_StackFrameVsStackTrace
  #14  Framework_Stopwatch
  #15  Framework_StringConcatVsStringBuilder
  #16  IL_Loops
  #17  IL_ReadonlyFields
  #18  IL_Switch
  #19  IntroBaseline
  #20  IntroBasic
  #21  IntroColumns
  #22  IntroCommandStyle
  #23  IntroConfigSource
  #24  IntroConfigUnion
  #25  IntroDefaultToolchain
  #26  IntroJobsFull
  #27  IntroMultipleRuntimes
  #28  IntroParams
  #29  IntroTags
  #30  Jit_ArraySumLoopUnrolling
  #31  Jit_AsVsCast
  #32  Jit_Bce
  #33  Jit_BoolToInt
  #34  Jit_GenericsMethod
  #35  Jit_Inlining
  #36  Jit_InterfaceMethod
  #37  Jit_LoopUnrolling
  #38  Jit_RegistersVsStack
  #39  Jit_RotateBits
  #40  Array_AccessNormalRefUnsafe
  #41  Array_HeapAllocVsStackAlloc
  #42  Math_DoubleSqrt
  #43  Math_DoubleSqrtAvx
  #44  Os_Sleep
```

The previous benchmark was the second option, (number **#1 Algo_Md5VsSha256**).

Benchmarking is hard, so it's a good idea to use a library such as BenchmarkDotNet if you can. The only conclusion we can draw from our benchmarks is that SHA-256 is slower than MD5. However, SHA-256 should be fast enough for most applications and it's more secure for integrity checking. However, it is still not suitable for password storage.

SHA-256 can be used to provide signatures for verifying downloaded files, which

must be retrieved over HTTPS to be safe, and for signing certificates. When used as part of an HMAC, it can also be used to securely authenticate messages–API requests, for example. You will only connect successfully if you know the correct API key to hash with.

Serialization

Serialization is the process of turning objects into data suitable for transmission over a network or for storage. We also include deserialization, which is the reverse, under this umbrella. Serialization can have significant performance implications, not only on the network transmission speed but also on computation, as it can make up most of the expensive processing on a web server. You can read more about serialization on MSDN (msdn.microsoft.com/en-us/library/mt656716).

Serialization formats can be text-based or binary. Some popular text-based formats are Extensible Markup Language (**XML**) and **JavaScript Object Notation** (**JSON**). A popular binary format is **Protocol Buffers**, which was developed at Google. There's another binary serialization format (`BinaryFormatter`) built into the full .NET, but this is not in .NET Core.

XML has fallen out of fashion with developers, and JSON is now generally preferred. This is partly due to the smaller size of equivalent JSON payloads, but it may also be due to the use of XML in the originally named **Simple Object Access Protocol** (**SOAP**). This is used in **Windows Communication Foundation** (**WCF**), but SOAP is no longer an acronym, as developers discovered it is far from simple.

JSON is popular due to being compact, human-readable, and because it can easily be consumed by JavaScript, particularly in web browsers. There are many different JSON serializers for .NET, with different performance characteristics. However, because JSON is not as rigidly defined as XML, there can be differences in implementations, which make them incompatible, especially when dealing with complex types such as dates. For example, the very popular **Json.NET** represents dates in the **International Organization for Standardization** (**ISO**) format, whereas the JSON serializer used in old versions of ASP.NET MVC represented dates as the number of milliseconds since the Unix **epoch**, wrapped in a JavaScript date constructor.

The .NET developer community has converged on Json.NET, and compatibility is always preferable to performance. ASP.NET Web API has used Json.NET as the default for a while now, and ASP.NET Core also uses Json.NET. There is a serializer that's part of the **ServiceStack** framework called **ServiceStack.Text**, which claims to be faster, but you should probably value compatibility and documentation over speed. The same applies to other JSON libraries such as Jil (github.com/kevin-montrose/Jil) and **NetJSON** (github.com/rpgmaker/NetJSON), which can be even faster than ServiceStack in benchmarks.

If you are after pure performance, and you control all of the endpoints, then you probably would want to use a binary protocol. However, this may limit future interoperability with third-party endpoints which you don't control. Therefore, it's best to only use these internally.

It would be a bad idea to build your own custom message protocol on top of UDP. So, if you want to use binary serialization, you should look at something like **protobuf-net**, which is a Protocol Buffers implementation for .NET. You may also wish to consider Microsoft's Bond framework (github.com/Microsoft/bond) or Amazon's Ion (amznlabs.github.io/ion-docs). You may need to tune these tools for best performance, for example, by changing the default buffer size.

SIMD CPU instructions

Single Instruction Multiple Data (**SIMD**) is a technique that is available on many modern processors and can speed up execution by parallelizing calculations even in a single thread on one core. SIMD takes advantage of additional instructions available on CPUs to operate on sets of values (vectors) rather than just single values (scalars).

The most common instruction set for this is called **Streaming SIMD Extensions 2 (SSE2)** and it has been around for over 15 years since its debut with the Pentium 4. A newer instruction set called **Advanced Vector Extensions** (**AVX**) offers superior performance over SSE2 and has been around for over five years. So, if you're using a reasonably recent x86-64 CPU, then you should have access to these extra instructions.

Note

Some ARM CPUs (such as those in the Raspberry Pi 2 and 3) contain a similar technology called **NEON**, officially known as **Advanced SIMD**. This is not currently supported in .NET, but may be in the future. An official open source library project in C is hosted at projectne10.org .

You can use the following Boolean property to test if SIMD is supported:

```
Vector.IsHardwareAccelerated
```

This property is JIT intrinsic, and the value is set by RyuJIT at runtime.

You can instantiate a generic typed `Vector` or use one of the two/three/four dimensional convenience classes. For example, to create a single precision floating point vector, you could use the following generic code:

```
var vectorf = new Vector<float>(11f);
```

To create a single precision floating point 3D vector instead, you could use this code:

```
var vector3d = new Vector3(0f, 1f, 2f);
```

Tip

A two dimensional double precision floating point vector can be a good substitute for a `Complex` structure. It will give higher performance on hardware accelerated systems. `Vector2` only supports single precision floating point numbers, but you can use the generic type to specify the real and imaginary components of the complex

number as a double. `Complex` only supports double precision floating point numbers, but, if you don't need high precision, you could still use the `Vector2` convenience class. Unfortunately, this means that it's not simply a drop in replacement, but the math is different anyway.

You can now use standard vector mathematics, but modifying your algorithms to take advantage of vectors can be complex and isn't something you should typically be doing in a web application. It can be useful for desktop applications, but, if a process takes a long time in a web request, it's often best to run it in the background and then it doesn't need to be as quick.

We will cover this distributed architecture approach in the next chapter. For this reason, we won't be going into any more detail on SIMD, but you can read more on it if you wish, now that you have a taste of it. You can read some background information at wikipedia.org/wiki/SIMD and you can find the documentation for the .NET implementation on MSDN at msdn.microsoft.com/en-us/library/dn858218 . You could also take a look at the example console application, which is available for download along with this book, as a simple starter for ten.

Parallel programming

While SIMD is good at increasing the performance of a single thread running on one core, it doesn't work across multiple cores or processors and its applications are limited. Modern scaling means adding more CPUs, not simply making a single thread faster. We don't just want to parallelize our data as SIMD does; we should actually focus more on parallelizing our processing, as this can scale better.

There are various .NET technologies available to help with parallel processing so that you don't have to write your own threading code. Two such parallel extensions are **Parallel LINQ (PLINQ)**, which extends the LINQ operations you're familiar with, and the **Task Parallel Library (TPL)**.

Task Parallel Library

One of the main features of the TPL is to extend loops to run in parallel. However, you need to be careful with parallel processing, as it can actually make your software slower while doing simple tasks. The overheads involved with marshalling the multiple tasks can dwarf the execution of the workload for trivial operations.

For example, take the following simple `for` loop:

```
for (int i = 0; i < array.Length; i++)
{
    sum += array[i];
}
```

The array in the preceding `for` loop contains 100,000 integers, but adding integers is one of the easiest things a CPU can do and using a `for` loop on an array is a very quick way of enumerating. This accumulation will complete in under a tenth of a millisecond on a reasonably modern machine.

You may think that you would be able to speed this up by parallelizing the operation. Perhaps you could split the array, run the summation on two cores in parallel and add the results.

You might use the following code to attempt this:

```
Parallel.For(0, array.Length, i =>
{
    Interlocked.Add(ref sum, array[i]);
});
```

Tip

You must use an interlocked add or you will get an incorrect summation result. If you don't, the threads will interfere with each other, corrupting the data when writing to the same location in memory.

However, this code actually runs over 42 times slower than the first example. The extra overhead, complexity of running many threads, and locking the variable so that only one thread can write to it at a time is just not worth it in this case.

Parallelization can be useful for more complex processes, especially if the body of the loop performs some slow operation such as accessing the file system. However, blocking I/O operations may be better dealt with by using asynchronous access. Parallel access can cause contention, because access may eventually have to be performed in series at some point, for example, at the hardware level.

If we want to perform a more processor-intensive operation, such as hashing multiple passwords, then running the tasks in parallel can be beneficial. The following code performs a PBKDF2 hash on each password in a list and then calculates the Base64 representation of the result:

```
foreach (var password in passwords)
{
    var pbkdf2 = new Rfc2898DeriveBytes(password, 256, 10000);
    Convert.ToBase64String(pbkdf2.GetBytes(256));
}
```

We're not using the output in this example, but you may be doing this to upgrade the security of the passwords in your database by migrating them to a more resilient key-stretching algorithm. The input may be plaintext passwords or the output of a legacy one-way hash function, for example MD5 or an unsalted SHA.

We can improve the speed of this on a multicore system by using the `Parallel.ForEach` loop, using code such as the following:

```
Parallel.ForEach(passwords, password =>
{
    var pbkdf2 = new Rfc2898DeriveBytes(password, 256, 10000);
    Convert.ToBase64String(pbkdf2.GetBytes(256));
});
```

This will speed up the process, but by how much will depend on many factors, such as the number of passwords in the list, the number of logical processors and the number of CPU cores. For example, on a Core i5 CPU with two cores, but four logical processors, having only two passwords in the list does not result in a massive improvement (only 1.5 times quicker). With four passwords (or more) in the list, the improvement is better (about 1.9 times quicker). There is still some overhead, so you can't get double the speed with twice the CPU cores.

We can see the reason for this difference by looking at the CPU utilization in the task manager during benchmarking. With only two passwords to hash, the CPU graph looks like the following:
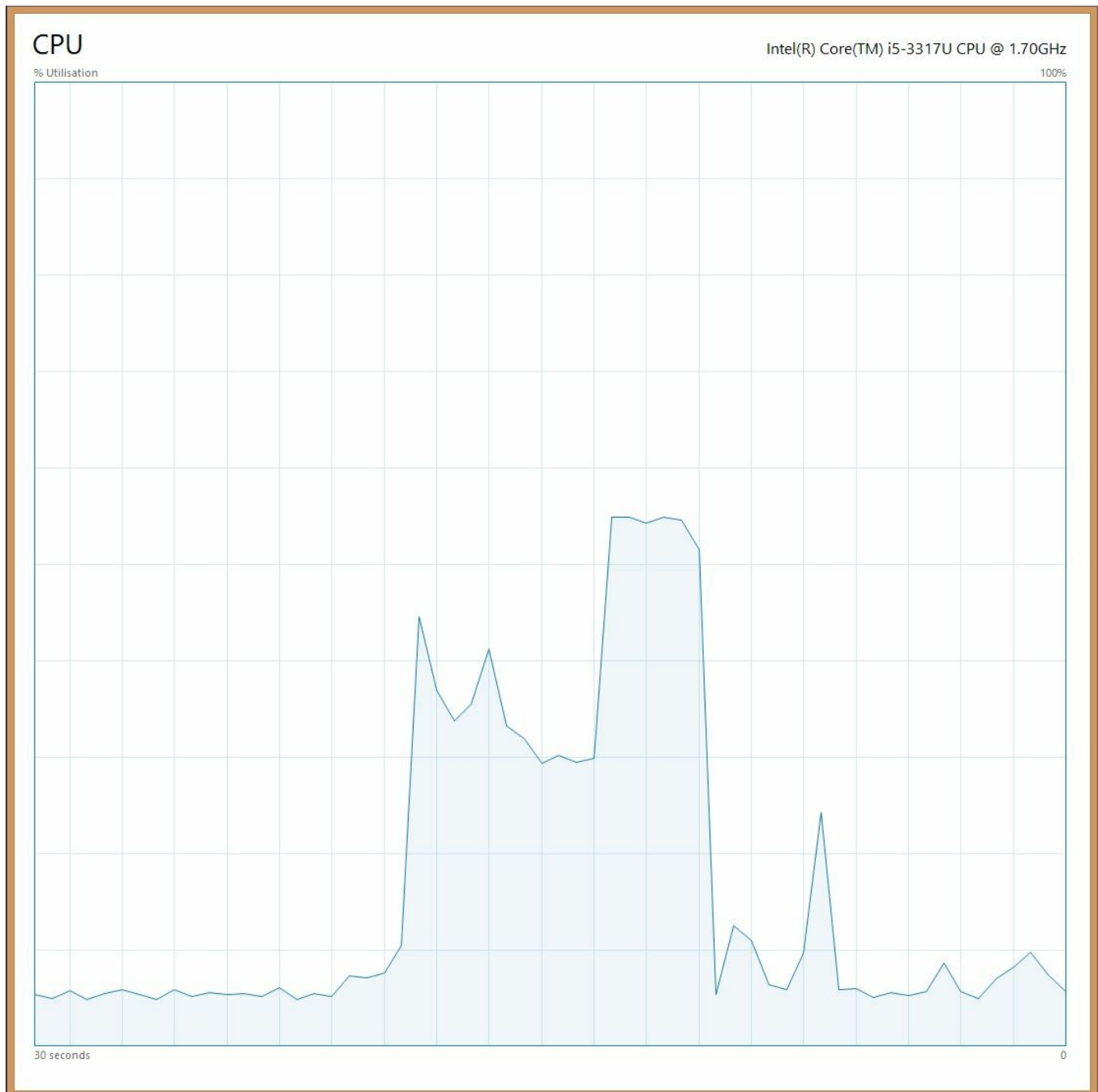
In the preceding graph, we can see that initially, when hashing in series, the CPU is running at about 25%, fully using one logical CPU. When hashing two passwords in parallel, it uses 50%, running on two logical processors. This doesn't translate into a twofold increase in speed due to the intrinsic overheads and the nature of **hyper-threading**.

Note

Hyper-threading is a technology that typically exposes two logical processors to the OS for each physical core. However, the two logical CPUs still share the execution resources of their single core.

Although there are two cores on the CPU die, hyper-threading exposes four logical CPUs to the OS. As we only have two threads, because we are hashing two passwords, we can only use two processors. If the threads are executed on different cores, then the speed increase can be good. But if they are executed on processors sharing the same core, then performance won't be as impressive. It is still better than single-threaded hashing due to scheduling improvements, which is what hyper-threading is designed for.

When we hash four passwords at the same time, the CPU graph looks like the following:
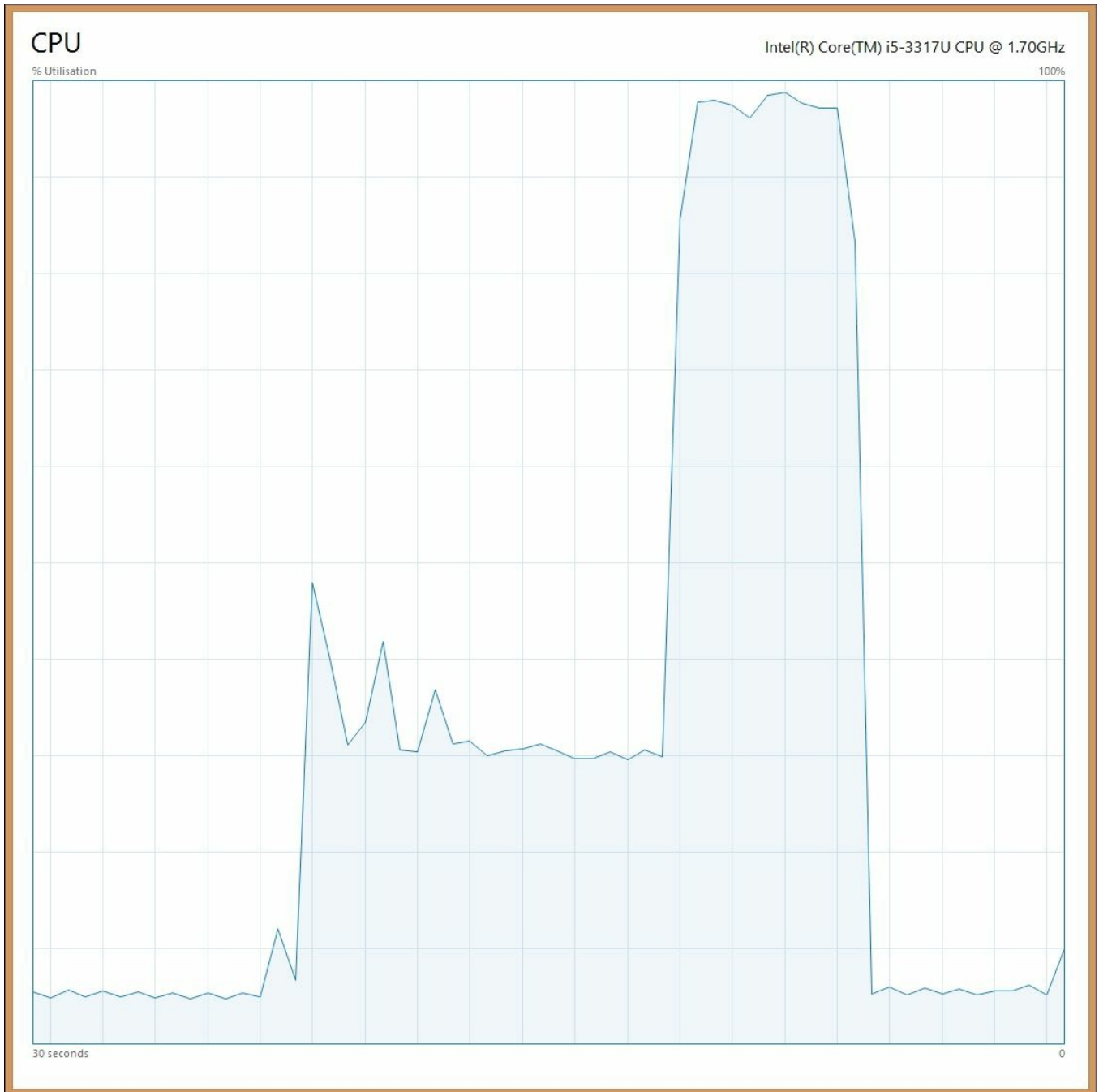
We can see that now the initial 25% usage jumps to almost full utilization and we are making use of most of the processor. This translates to just under a doubling of the performance as compared to hashing in sequence. There are still significant overheads involved, but, as the main operation is now so much quicker, the tradeoff is worth it.

Parallel LINQ

There are other ways to take advantage of parallel programming, such as LINQ expressions. We could rewrite the previous example as a LINQ expression and it may look something like the following:

```
passwords.AsParallel().ForAll(p =>
{
    var pbkdf2 = new Rfc2898DeriveBytes(p, 256, 10000);
    Convert.ToBase64String(pbkdf2.GetBytes(256));
});
```

You can enable these features with the `AsParallel()` method. The `ForAll()` method has the same purpose as the loops in previous examples and is useful if the order is unimportant. If ordering is important, then there is an `AsOrdered()` method, which can help solve this. However, this can reduce the performance gains due to the extra processing involved.

This example performs similarly to the previous one that used a parallel loop, which is unsurprising. We can also limit the number of operations that can occur in parallel, using the `WithDegreeOfParallelism()` method as follows:

```
passwords.AsParallel().WithDegreeOfParallelism(2).ForAll(p =>
{
    var pbkdf2 = new Rfc2898DeriveBytes(p, 256, 10000);
    Convert.ToBase64String(pbkdf2.GetBytes(256));
});
```

This preceding example limits the hashes to two at a time and performs similarly to when we only had two passwords in the list, which is to be expected. This can be useful if you don't want to max out the CPU, because there are other important processes running on it.

You can achieve the same effect with the TPL by setting the `MaxDegreeOfParallelism` property on an instance of the `ParallelOptions` class. This object can then be passed into overloaded versions of the loop methods as a parameter, along with the main body.
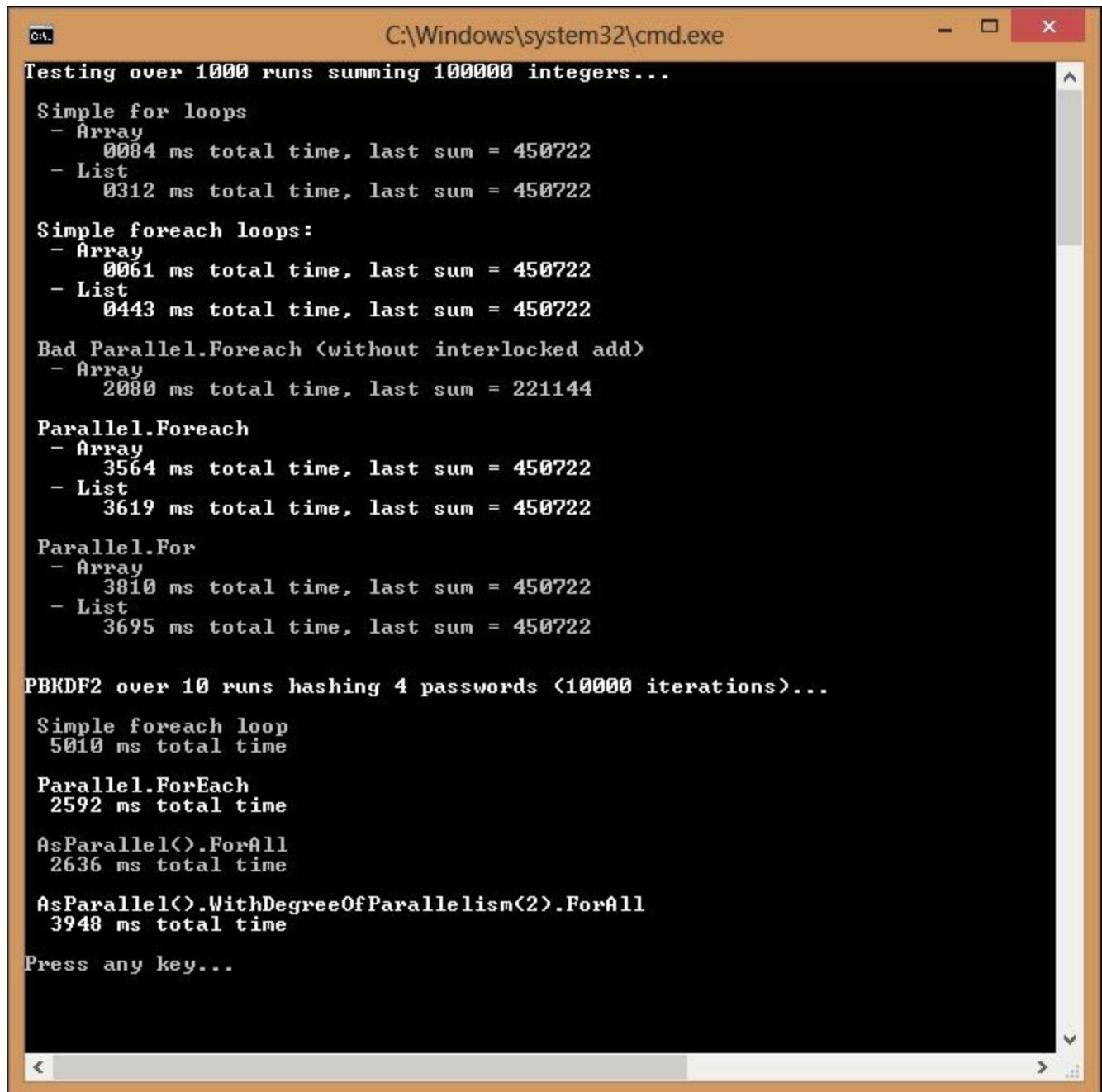
Tip

It's important, when you're using parallel LINQ to query datasets, that you don't lose sight of the best place for the query to be performed. You may speed up a query in the application with parallelization, but the best place for the query to occur may still be in the database, which can be even faster. To read more on this topic, refer back to Chapter 5, *Optimizing I/O Performance* , and Chapter 3, *Fixing Common*

Performance Problems .

Parallel benchmarking

Let's have a look at the output of a simple .NET Core console application, which benchmarks the techniques that we have just discussed. It shows one situation where parallelization doesn't help and actually makes things worse. It then shows another situation where it does help.

When calculating a sum, by accumulating 100,000 random integers between zero and ten, the quickest way is to use an array in a simple `foreach` loop. Using parallelization here makes the process much slower, and if used Naïvely, without

locking, will give an incorrect result, which is much worse.

When performing a more computationally intensive workload, such as a PBKDF2 hashing function on multiple passwords, parallelization can help significantly. The time is almost halved, as it is running across two cores. The final operation, which limits the number of threads, can take a varying amount of time on different runs. This is likely due to the threads sometimes sharing a core and sometimes running on different cores. It can be almost as quick as using all logical cores, depending on the work.

The CPU graph for the benchmark looks like the following:
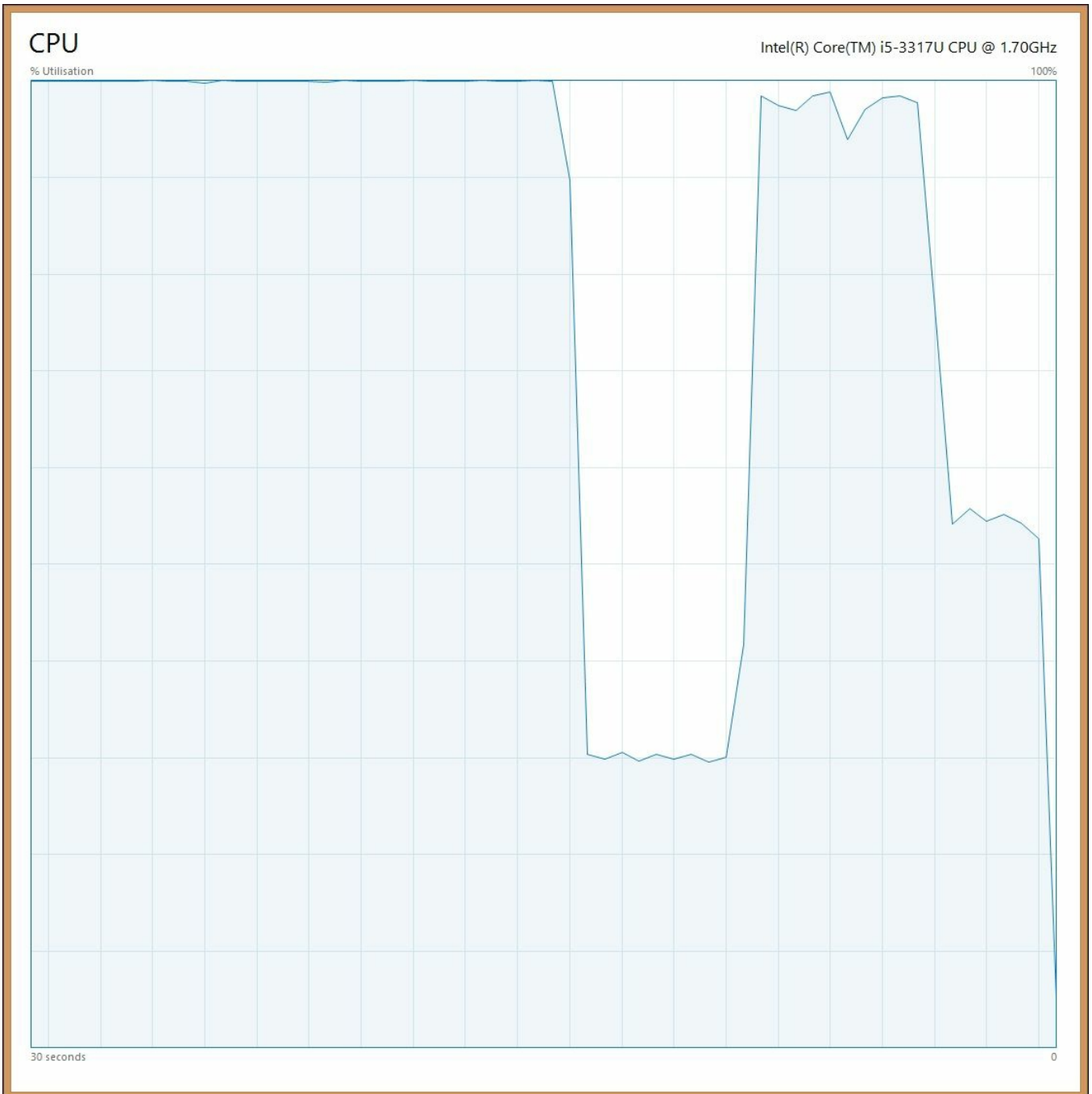
The initial parallel summations max out the CPU and are very inefficient. Next, the single-threaded hashing uses only one logical processor (25%), but the other, later hashes make almost full use of both the cores. The final hashing, limited to two passwords at a time, only makes use of half the CPU power.

Parallel programming limitations

Performance problems with web applications don't typically mean increasing speed for a single user on the system in isolation. It's easy to make a web app perform when there's only one user, but the challenge lies in maintaining that single user performance as the number of users scales up and you have wall-to-wall requests.

The parallel programming techniques discussed in this section have limited use in scaling web applications. You already have a parallelized system, as each user request will be allocated its own resources. You can simply add more instances, agents, or machines to meet demand. The problem then is in distributing work efficiently between them and avoiding bottlenecks for shared resources. We'll cover this more in the next chapter, but first let's look at some things that you should be avoiding.

Practices to avoid

We've shown some ways of speeding up software, but it's often better to illustrate
what not to do and how things can go wrong. Web applications generally perform
well if no bad practices have been followed and here we'll highlight a few things you
should watch out for.

Reflection

Reflection is the process of programmatically inspecting your code with other code, and digging into its internals at runtime. For example, you could inspect an assembly when it is loaded to see what classes and interfaces it implements so that you can call them. It is generally discouraged and should be avoided if possible. There are usually other ways to achieve the same result that don't require reflection, although it is occasionally useful.

Reflection is often bad for performance, and this is well-documented, but, as usual, it depends on what you're using it for. What is new is that there are significant changes to reflection for .NET Core. The API has changed and it is now optional. So, if you don't use reflection, you don't have to pay the performance penalty. This allows the native tool chain to optimize compilation better, as reflection adds restrictions to what can be done with static linking.

There is an extra method on the reflection API now, so, whereas previously you would have called something like `myObject.GetType().GetMembers()`, you now need to call it as `myObject.GetType().GetTypeInfo().GetMembers()` by inserting the new `GetTypeInfo()` method, which is in the `System.Reflection` namespace.

If you must use reflection, then it is best not to perform it repeatedly or in a tight loop. However, it would be fine to use it once during the startup of your application. Yet, if you can avoid using it entirely, you can benefit from some of the new improvements in .NET Core, for example, native compilation and the performance boost that it brings.

Regular expressions

A **regular expression** (**regex**) can be very useful, but can perform badly and is typically misused in situations where another solution would be better. For example, a regex is often used for e-mail validation when there are much more reliable ways to do this.

If reusing a regex repeatedly, you may be better off compiling it for performance by specifying the `RegexOptions.Compiled` option in the constructor. This only helps if you're using the regex a lot and involves an initial performance penalty. So, ensure that you check if there is actually an improvement and it isn't now slower.

The `RegexOptions.IgnoreCase` option can also affect performance, but it may in fact slow things down, so always test for your inputs. Compiling has an effect on this too and you may want to use `RegexOptions.CultureInvariant` in addition, to avoid comparison issues.

Be wary of trusting user input to a regex. It is possible to get them to perform a large amount of backtracking and use excessive resources. You shouldn't allow unconstrained input to a regex, as they can be made to run for hours.

Regexes are often used for e-mail address validation, but this is usually a bad idea. The only way to fully validate an e-mail address is to send an e-mail to it. You can then have the user click a link in the e-mail to indicate that they have access to that mailbox and have received it. E-mail addresses can vary a lot from the common ones that people are regularly exposed to, and this is even truer with the new top-level domains being added.

Many of the regexes for e-mail address validation found online will reject perfectly valid e-mail addresses. If you want to assist the user, and perform some basic e-mail address validation on a form, then all you can sensibly do is check that there is an `@` symbol in it (and a `.` after that) so that the e-mail is in the form `x@y.z`. You can do this with a simple string test and avoid the performance penalty and security risk of a regular expression.

String concatenation in tight loops

As strings are immutable and can't change, when you concatenate a string, a new object is created. This can cause performance problems and issues with memory use if you do it a lot inside a tight loop.

You may find it better to use a string builder or another approach. However, don't fret too much about this, as it only applies at a large scale. Always test to see if it is genuinely a problem and don't micro-optimize where you don't need to.

It's good general advice to work out where your code is spending most of its time, and focus your optimization there. It's obviously much better to optimize code executed millions of times inside a loop than code that only runs occasionally.

Dynamic typing

C# is a statically typed language and variable types are checked at compile time, but it does have some dynamic features. You can use the `dynamic` type and objects such as `ExpandoObject` to get some of the features of a dynamically typed language. The `var` type is not in fact dynamic, and is simply inferred at compile time.

Dynamic typing has a performance and safety penalty, so it is best avoided if you can find another way to solve your problem. For example, the `ViewBag` in ASP.NET MVC is dynamic, so it is best not to use `ViewBag`, and use a well-defined view model instead. This has many other benefits apart from performance, such as safety and convenience.

Synchronous operations

Synchronous methods block execution, and should be avoided if possible, especially if they are slow or access I/O. We've covered asynchronous (async for short) operations in previous chapters. Understanding how to use async is important for modern high performance programming, and new language features make it more accessible than ever. If an async method is available, then it should generally be used in preference to the synchronous blocking version.

The `async` and `await` keywords make asynchronous programming much easier than it used to be, but, as covered in [Chapter 3](#) , *Fixing Common Performance Problems* , the effects on web applications are not always visible for a lone user. These convenient features allow you to serve more users simultaneously by returning threads to the pool during downtime, while waiting for operations to complete. The threads can then be used to service other users' requests, which allows you to handle more users with less servers than otherwise.

Async methods can be useful, but the big gains come not from writing asynchronous code, but from having an asynchronous architecture. We will cover distributed architecture in the next chapter, when we discuss message queuing.

Exceptions

Exceptions should, as the name suggests, be reserved for exceptional circumstances. Exceptions are slow and expensive and shouldn't be used as flow control in business logic if you know that an event is likely to occur.

This isn't to say that you shouldn't use exception handling–you should. However, it should be reserved for events that are genuinely unexpected and rare. If you can predict ahead of time that a condition may occur then you should handle it explicitly.

For example, the disk becoming full and your code not being able to write a file because there is no space is an exceptional situation. You would not expect this to normally happen, and you can just `try` the file operation and `catch` any exceptions. However, if you are trying to parse a date string or access a dictionary, then you should probably use the special `TryParse()` and `TryGetValue()` methods and check for null values rather than just relying on exception handling.

Summary

In this chapter, we discussed some techniques that can improve the performance of code execution and dug into the projects that make up .NET Core and ASP.NET Core. We explored data structures, serialization, hashing, and parallel programming and how to benchmark for measuring relative performance.

Linear performance characteristics are easier to scale and code that does not exhibit this behavior can be slow when the load increases. Code that has an exponential performance characteristic or has erratic outliers (which are rare but very slow when they occur) can cause performance headaches. It is often better to aim for code, that while being slightly slower in normal cases is more predictable and performs consistently over a large range of loads.

The main lesson here is to not blindly apply parallel programming and other potentially performance-enhancing techniques. Always test to make sure that they make a positive impact, as they can easily make things worse. We aim for the situation where everything is awesome, but, if we're not careful, we can make everything awful by mistake.

In the next chapter, you'll learn about caching and message queuing–two advanced techniques that can significantly improve the performance of a system.

Chapter 7. Learning Caching and Message Queuing

Caching is incredibly useful and can be applied to almost all layers of an application stack. However, it's hard to always get caching working correctly, so, in this chapter, we will cover caching at the web, application, and database levels. We will show you how to use a reverse proxy server to store the results of your rendered web pages and other assets. We'll also cover caching at lower levels, using an in-memory data store to speed up access. You will learn how to ensure that you can always flush (or bust) your cache if you need to force the propagation of updates.

This chapter also covers asynchronous architecture design using message queuing and abstractions that encapsulate various messaging patterns. You will learn how to perform a long running operation (such as video encoding) in the background, while keeping the user informed of its progress.

You will learn how to apply caching and message queuing software design patterns to slow operations so that they don't have to be performed in real time. You'll also learn about the complexity that these patterns can add, and understand the tradeoffs involved. We'll see how to combat these complexities and mitigate the downsides, in Chapter 8, *The Downsides of Performance-Enhancing Tools* .

The topics covered in this chapter include the following:

- Web caching background
- JavaScript service workers
- Varnish proxy and IIS web servers
- Redis and Memcached in-memory application caching
- Message Queuing and messaging patterns
- RabbitMQ and its various client libraries

Why caching is hard

Caching is hard, but it's not hard because it's difficult to cache something. Caching indefinitely is easy, the hard part is invalidating the cache when you want to make an update. There's a well-used quote from the late Phil Karlton of Netscape that goes:

> *"There are only two hard things in Computer Science: cache invalidation and naming things."*

There are also many humorous variants on it, as used previously throughout this book. This sentiment may be a slight exaggeration, but it highlights how complex removing your "done-computer-stuff ^{TM}" from your "quick-things-box 2.0 ^{TM}" is perceived to be. Naming things is genuinely very hard though.

Caching is the process of storing a temporary snapshot of some data. This temporary cache can then be used instead of regenerating the original data (or retrieving it from the canonical source) every time it is required. Doing this has obvious performance benefits, but it makes your system more complicated and harder to conceptualize. When you have many caches interacting, the results can appear almost random unless you are disciplined in your approach.

When you are reasoning about caching (and message queuing), it is helpful to dispel the idea that data only exists in a single consistent state. It is easier if you embrace the concept that data has a freshness, and is always stale by some amount. This is in fact always the case, but the short timeframes involved in a small system mean that you can typically ignore it in order to simplify your thinking. However, when it comes to caching, the timescales are longer and so, freshness is more important. A system at scale can only be eventually consistent, and various parts of it will have a different temporal view of the data. You need to accept that data can be in motion, otherwise you're just not thinking four-dimensionally!

As a trivial example, consider a traditional static website. A visitor loads a page in their browser, but this page is now instantly out-of-date. The page on the server could have been updated just after the visitor retrieved it, but they will not know, as the old version will remain in their browser until they refresh the page.

If we extend this example to a database-backed web application, such as an ASP.NET or WordPress website, then the same principle applies. A user retrieves a web page generated from data in the database, but it could be out-of-date as soon as it is loaded. The underlying data could have changed, but the page containing the old data remains in the browser.

By default, web apps typically regenerate HTML from the DB for every page load, but this is incredibly inefficient if the data has not changed. It is only done like this so that when a change is made, it shows up immediately as soon as the page is refreshed.

However, a user may have an old page in their browser, and you have limited control over this. So you may as well cache this page on the server as well and only remove it when the underlying data in the database changes. Caching the rendered HTML like this is often essential for maintaining performance at a scale beyond simply a small number of users.

Web caching

The first category of caching that we'll discuss is at the web level. This involves storing the final output of your web stack as it would be sent to users so that, when requested again, it's ready to go and doesn't need to be regenerated. Caching at this stage removes the need for expensive database lookups and CPU-intensive rendering at the application layer. This reduces latency and decreases the workload on your servers, allowing you to handle more users and serve each user rapidly.

Web caching typically occurs on your web servers or on reverse proxy servers, which you have put in front of your web servers to shield them from excessive load. You might also choose to hand this task over to a third party, such as a CDN. Here we will cover two pieces of web server and proxy server software, IIS and Varnish. However, many more web caching and load balancing technologies are available, for example, NGINX or HAProxy.

Caching at the web layer works best for static assets and resources such as JavaScript, CSS, and images. Yet it can also work for anonymous HTML that is rarely updated but regularly accessed, such as a homepage or landing page, which is unauthenticated and not customized for a user.

We touched upon proxy servers in Chapter 3, *Fixing Common Performance Problems*, and covered web layer caching a little in Chapter 4, *Addressing Network Performance* . However, in this chapter, we'll go into more detail on web caching.

Caching background

Before we delve into the implementation details, it helps to understand a little about how caching works on the web. If you take the time to study the mechanisms at work, then caching will be less confusing and less frustrating than if you just dived straight in.

It is helpful to read and understand the relevant HTTP specifications. However, don't assume that software always strictly adheres to these web standards even if it claims to.

First, let's look at a typical network setup which you may be traversing with your HTTP traffic. The following diagram illustrates an example of a common configuration for a web application:
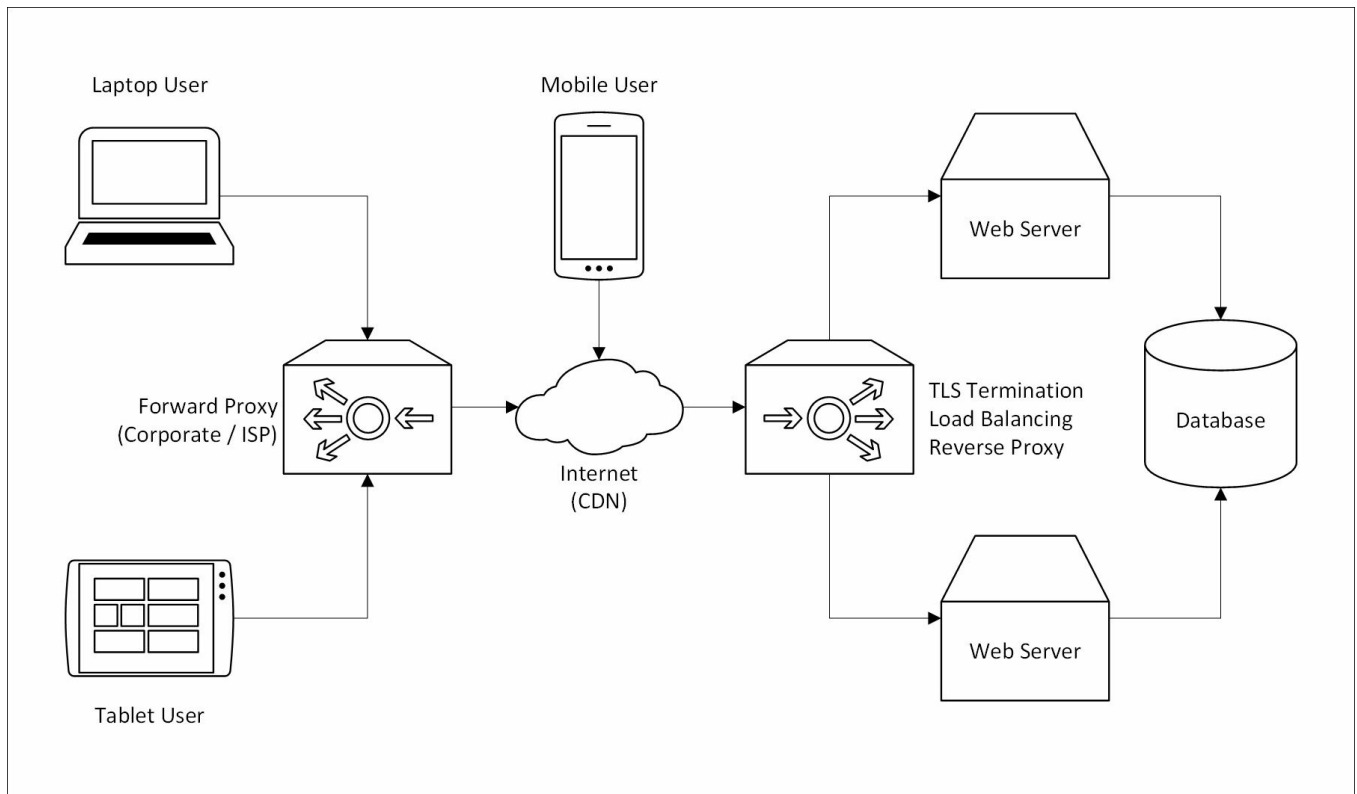
As seen in the preceding diagram, the laptop and tablet users are connecting through a caching forward proxy server (which may be on a corporate network or at an ISP). The mobile user is connecting directly over the internet. However, all users are going through a CDN before reaching your infrastructure.

After your firewall (not shown), there is an appliance which terminates the TLS connections, balances the load between web servers, and acts as a caching reverse proxy. These functions are often performed by separate devices, but we've kept things simple here.

Copies of your resources will be kept on your web servers, your reverse proxy, your CDN, any forward proxies, and in the browsers on all user devices. The simplest way to control the caching behavior of these resources is to use in-band signaling and add HTTP headers to your content, declaring cache control metadata only in a single place.

It's good practice to apply the same standard HTTP caching techniques to your own web servers and proxies even though you could customize them and flush their caches at will. This not only cuts down on the amount of configuration that you have to do, and avoids duplicated work, but it also ensures that any caches that you don't control should behave correctly too. Even when using HTTPS, the browser will still perform caching and there may also be transparent corporate proxies or meddling ISP captive

portals in the way.

HTTP headers

HTTP caching involves setting cache control headers in your responses. There are many of these headers, which have been added over the years from different standards and various versions of the protocol. You should know how these are used, but you should also understand how the uniqueness of a cacheable resource is determined–for example, by varying the URL or by altering only a part of it, such as query string parameters.

Many of these headers can be categorized by function and the version of HTTP that they were introduced with. Some headers have multiple functions and some are non-standard, yet are almost universally used. We won't cover all of these headers, but we will pick out some of the most important ones.

There are broadly two types of caching header categories. The first defines an absolute time during which the cache can be reused, without checking with the server. The second defines rules which the client can use to test with the server if the cache is still valid.

Most instructional headers (those that issue caching commands) fit into one of these two header categories. In addition to these, there are many purely informational headers, which provide details about the original connection and client, which may otherwise be obscured by a cache (for example, the original client IP address).

Some headers, such as `Cache-Control`, are part of the latest standard, but others, such as `Expires`, are typically used only for backwards compatibility, in case there is an ancient browser or an old proxy server in the way. However, this practice is becoming increasingly unnecessary as infrastructure and software is upgraded.

Note

The latest caching standard in this case is HTTP/1.1 as HTTP/2 uses the same caching directives (RFC 7234). Some headers date from HTTP/1.0, which is considered a legacy protocol. Very old software may only support HTTP/1.0.

Standards may not be implemented correctly in all applications. It is a sensible idea to test that any observed behavior is as expected.

The `Age` header is used to indicate how long (in seconds) a resource has been in a cache. On the other hand, the `ETag` header is used to specify an identifier for an individual object or a particular unique version of that object.

The `Cache-Control` header tells caches if the resource may be cached. It can have many values including a `max-age` (in seconds) or `no-cache` and `no-store` directives. The confusing, yet subtle, difference between `no-cache` and `no-store` is that `no-cache` indicates that the client should check with the server before using the resource, whereas `no-store` indicates that the resource shouldn't be cached at all. To prevent caching, you should generally use `no-store`.

The ASP.NET Core `ResponseCache` action attribute sets the `Cache-Control` header and is covered in [Chapter 4](), *Addressing Network Performance*. However, this header may be ignored by some older caches. `Pragma` and `Expires` are older headers used for backward compatibility and they perform some of the same functions that the `Cache-Control` header now handles.

The `X-Forwarded-*` headers are used to provide more information about the original connection to the proxy or load balancer. These are non-standard, but widely-used, and are standardized as the combined `Forwarded` header (**RFC 7239**). The `Via` header also provides some proxy information, and `Front-End-Https` is a non-standard Microsoft header, which is similar to `X-Forwarded-Proto`. These protocol headers are useful for telling you if the original connection used HTTPS when this is stripped at the load balancer.

Tip

If you are terminating the TLS connections at a load balancer or proxy server, and are also redirecting users to HTTPS at the application level, then it is important to check the `Forwarded` headers. You can get stuck in an infinite redirection loop if your web servers desire HTTPS but only receive HTTP from the load balancer. Ideally, you should check all varieties of the headers, but, if you control the proxy, you can decide what headers to use.

There are lots of different HTTP headers that are involved in caching. The following list includes some of the ones that we haven't covered here. The large quantity of headers should give you an idea of just how complicated caching can be.

- `If-Match`
- `If-Modified-Since`
- `If-None-Match`
- `If-Range`
- `If-Unmodied-Since`
- `Last-Modified`
- `Max-Forwards`
- `Proxy-Authorization`
- `Vary`

Cache busting

Cache busting (also known as cache bursting, cache flushing, or cache invalidation) is the hard part of caching. It is easy to put an item in a cache, but, if you don't have a strategy ahead of time to manage the inevitable change, then you may come unstuck.

Getting cache busting correct is usually more important with web-level caching. This is because, with server side caching (which we'll discuss later in this chapter), you are in full control and can reset if you get it wrong. A mistake on the web can persist and be difficult to remedy.

In addition to setting the correct headers, it is helpful to vary the URL of resources when their content changes. This can be done by adding a timestamp, but often a convenient solution is to use a hash of the resource content and append this as a parameter. Many frameworks, including ASP.NET Core, use this approach. For example, consider the following JavaScript tag in a web page:

```
<script src="js/site.js"></script>
```

If you make a change to `site.js`, then the browser (or proxy) won't know that it has altered and may use a previous version. However, it will re-request it if the output is changed to something like the following:

```
<script src="js/site.js?v=EWaMeWsJBYWmL2g_KkgXZQ5nPe-
a3Ichp0LEgzXczKo">
</script>
```

Here the `v` (version) parameter is the **Base64 URL encoded**, SHA-256 hashed content of the `site.js` file. Making a small change to the file will radically alter the hash due to the **avalanche effect**.

Note

Base64 URL encoding is a variant on standard Base64 encoding. It uses different non-alphanumeric characters (+ becomes - while / changes to _) and percent encodes the = character (which is also made optional). Using this safe alphabet (from RFC 4648) makes the output suitable for use in URLs and filenames.

In ASP.NET Core, you can easily use this feature by adding the `asp-append-version` attribute with a value of `true` in your Razor views like so:

```
<script src="~/js/site.js" asp-append-version="true"></script>
```

Service workers

If you are writing a client-side web app, rather than a simple dynamic website, then you may wish to exert more control over caching using new browser features. You can do this by writing your cache control instructions in JavaScript (technically **ECMAScript 6** (**ES6**)). This gives you many more options when it comes to a visitor using your web app offline.

A **service worker** gives you greater control than the previous AppCache API. It also opens the door to features such as mobile web app install banners (which prompt a user to add your web app to their home screen). However, it is still a relatively new technology.

Tip

Service workers are a new experimental technology, and as such, are currently only supported in some recent browsers (partially in Chrome, Firefox, and Opera). You may prefer to use the previous deprecated AppCache method (which is almost universally supported) until adoption is more widespread.

Information on current browser support is available at caniuse.com/#feat=serviceworkers and caniuse.com/#feat=offline-apps (for AppCache). A more detailed service worker breakdown is available at jakearchibald.github.io/isserviceworkerready.

A service worker can do many useful things (such as background synchronization and push notifications), but the interesting parts, from our point of view, are the scriptable caches, which enable offline use. It effectively acts as an in-browser proxy server and can be used to improve the performance of a web application in addition to allowing interaction without an internet connection (after initial installation, of course).

Note

There are other types of **web workers** apart from service workers (for example, audio workers, dedicated workers, and shared workers), but we won't go into these here. All web workers allow you to offload work to a background task so that you don't make the browser unresponsive (by blocking the main UI thread with your work).

Service workers are asynchronous and rely heavily on JavaScript **promises**, which we'll assume you are familiar with. If you're not, then you should read up on them, as they're useful in many other contexts involving asynchronous and parallel scripting.

Service workers require the use of HTTPS (yet another good reason to use TLS on

your entire site). However, there is an exception for `localhost`, so you can still develop locally.

Service worker example

To install a service worker, first create a file for it (which is served over HTTPS). In the following example, this file is called `service-worker.js`. Then inside a `<script>` tag on your HTML page (also served over HTTPS), add the following JavaScript code:

```
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('service-worker.js', {
        scope: '/'
    });
}
```

The preceding code snippet first checks to see if service workers are supported, and if they are, registers your worker. You can now fetch resources and add them to the cache. An interesting performance enhancing use case for this is prefetching resources (that the user may need) ahead of time and putting them in the cache. Scope is an optional parameter and isn't strictly necessary in this case, as the file is in the root of the domain. We've shown it only to demonstrate usage, but it may be useful to specify this if the file was in a subfolder.

Before going any further, you should check that your worker has been installed correctly. In Chrome, you can open the special URL `chrome://inspect/#service-workers` to see any active service workers. For example, after opening instabail.uk in one tab, you can open the service worker inspector in another; you should see something like the following screenshot:
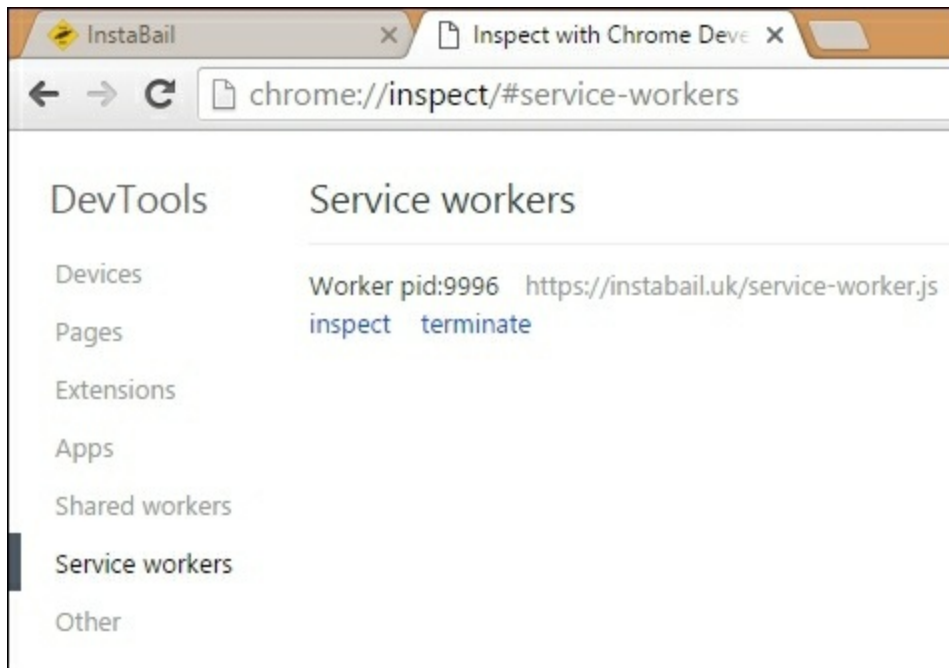
You can also visit `c hrome://serviceworker-internals` in Chrome to see the status of all service workers that have been registered, even if the sites aren't still open. For example, even after closing instabail.uk you should continue to see something like the following screenshot:
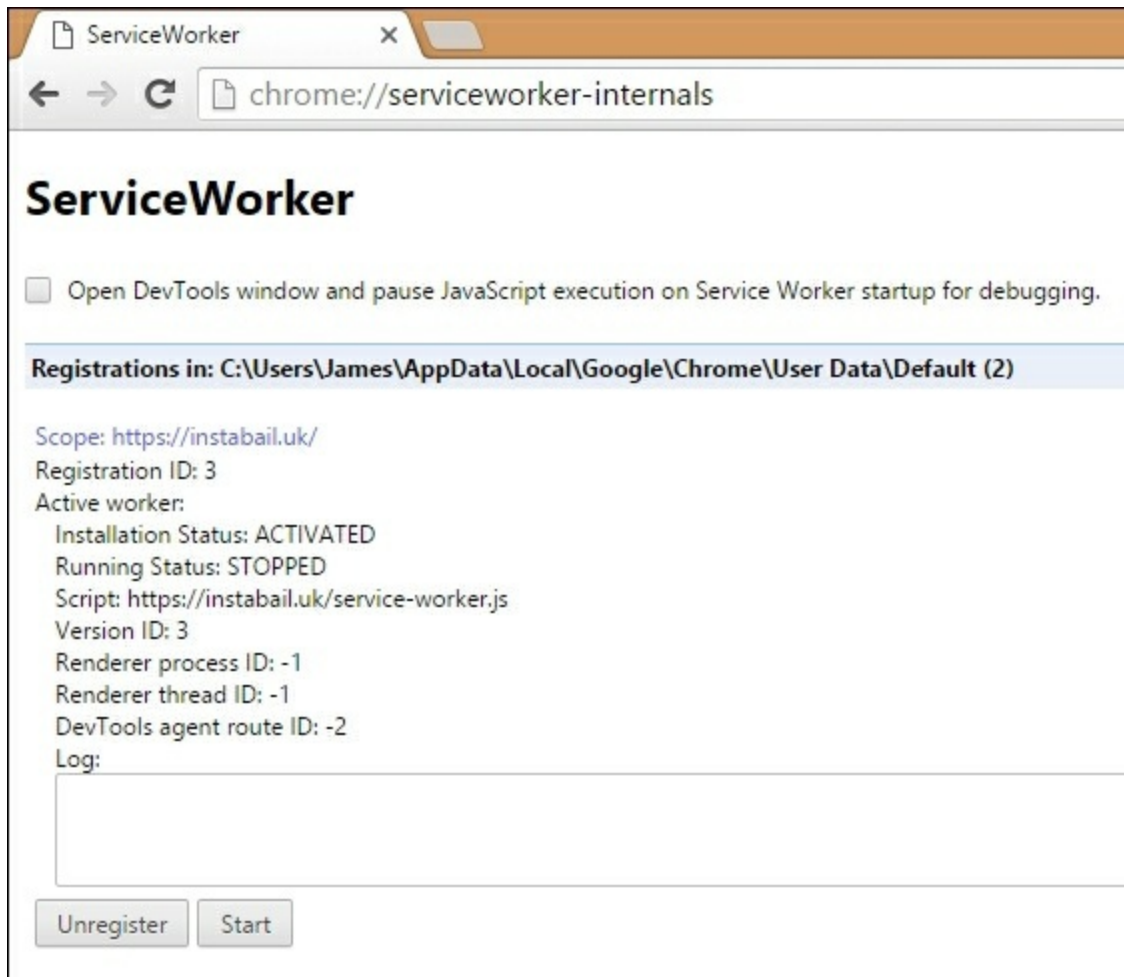
You can remove service workers by clicking the **Unregister** button. If the service is running, you will have **Stop** and **Inspect** buttons in place of **Start**. This page may be removed or merged into the inspector in a future version of Chrome.

Tip

If you are using an older version of Chrome (earlier than 50), you may see an error (`net::ERR_FILE_EXISTS`) against your service worker file in the console; but this is fine, so don't worry. It's simply a bug in Chrome as it tries to update your service worker, but finds that there aren't any changes.

Now you can start adding content to your service worker JavaScript file. We first need to install the worker and cache some files, which is done with an event listener, as shown in the following code:

```
self.addEventListener('install', function (event) {
    event.waitUntil(
        caches.open('cache-v01').then(function (cache) {
            return cache.addAll([
                '/',
```

```
            '/Content/bootstrap.min.css'
        ]);
    })
);
});
```

We have named our cache `cache-v01`, and provided an array of resources to cache. You would probably have more entries here and define the array outside of the function, but we have kept things simple here for clarity.

Tip

Don't cache your homepage if it dynamically renders live content. You may also want to use cache busting parameters for resources, as mentioned previously.

We can then add a `fetch` event listener to perform the magic of caching and fetching resources.

```
self.addEventListener('fetch', function (event) {
    event.respondWith(
        caches.match(event.request)
            .then(function (response) {
                if (response) return response;
                var myReq = event.request.clone();
                return fetch(myReq).then(
                    function (response) {
                        var myResp = response.clone();
                        caches.open('cache-v01')
                            .then(function (cache) {
                                cache.put(event.request, myResp);
                            });
                        return response;
                    }
                );
            }
        )
    );
});
```

First we check if the requested resource is in the cache and if it is, we return this. With promises, you can chain the `then` functions together and fall through them. If there is a cache miss due to the resource not being in the cache, we perform a `fetch` to our server to get the resource and return this. We then add the resource to the others by putting it in the same cache. We clone the request and response, because they are streams and can only be consumed once.

Note

The `fetch` function is the modern version of an `XMLHttpRequest` (XHR) and is used to retrieve data over the network. You can't use a synchronous XHR inside of a service worker, as they're designed to be asynchronous.

You can inspect your service worker and the caches in more detail by using the browser developer tools (**F12**). On the **Resources** tab, select **Service Workers** and you will see something like the following screenshot:
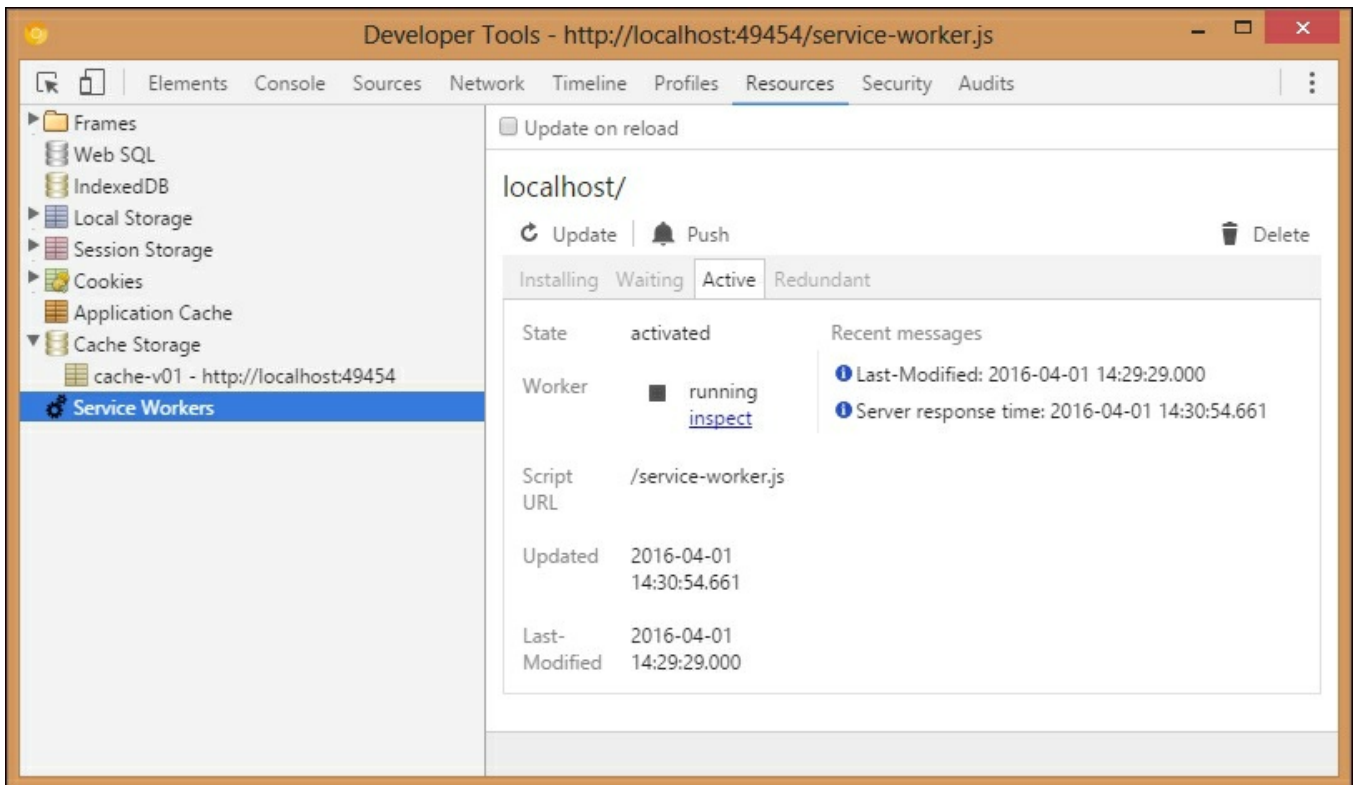
If you select **Cache Storage,** you will see the contents of the cache, which will look something like the following:
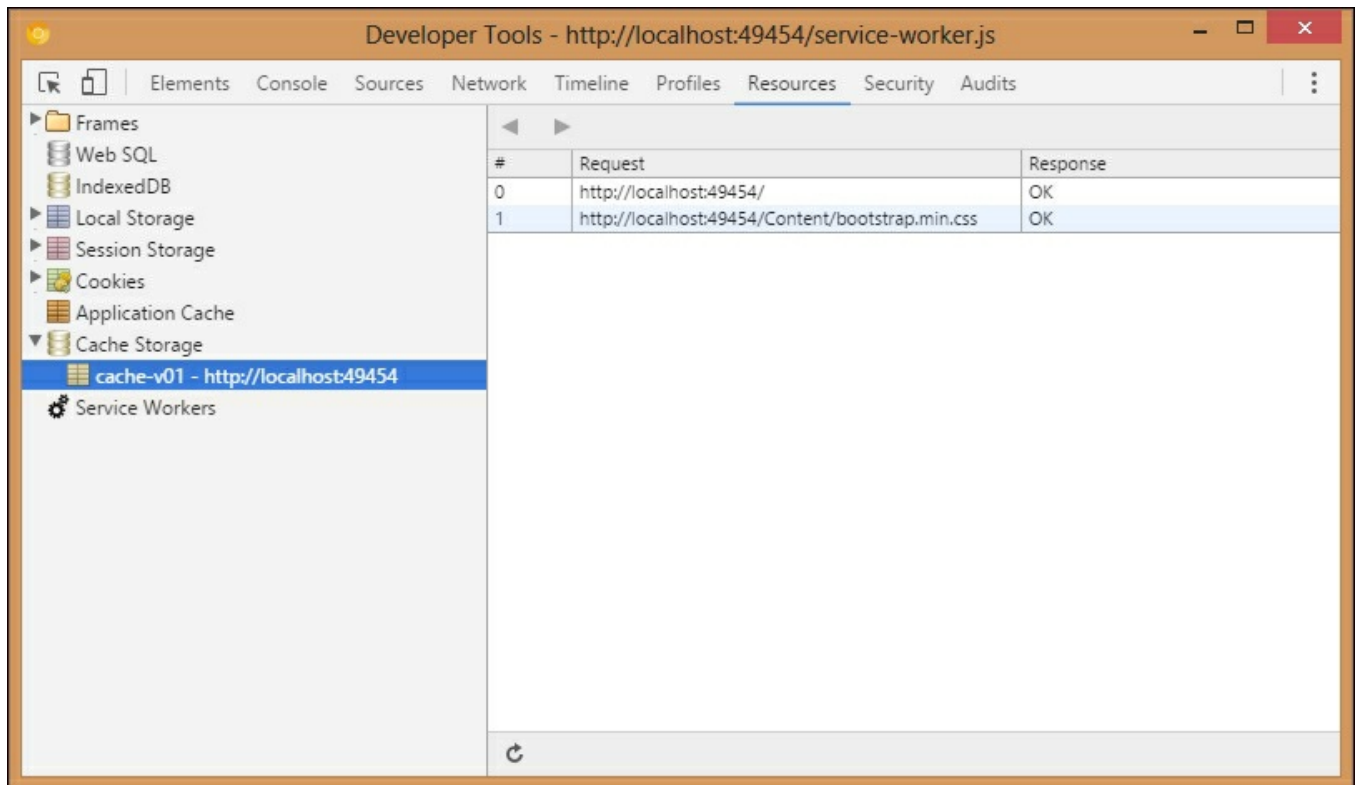
You can refresh the cache and delete items by right clicking. The **Application Cache**, above **Cache Storage**, would show the deprecated AppCache resources. As you navigate around, your site pages will be added to the cache (these pages should be suitable for caching, as they won't be requested from the server after this if using our demo code). After this, once you refresh the cache view, you should see more entries listed, which may look something like the following screenshot:
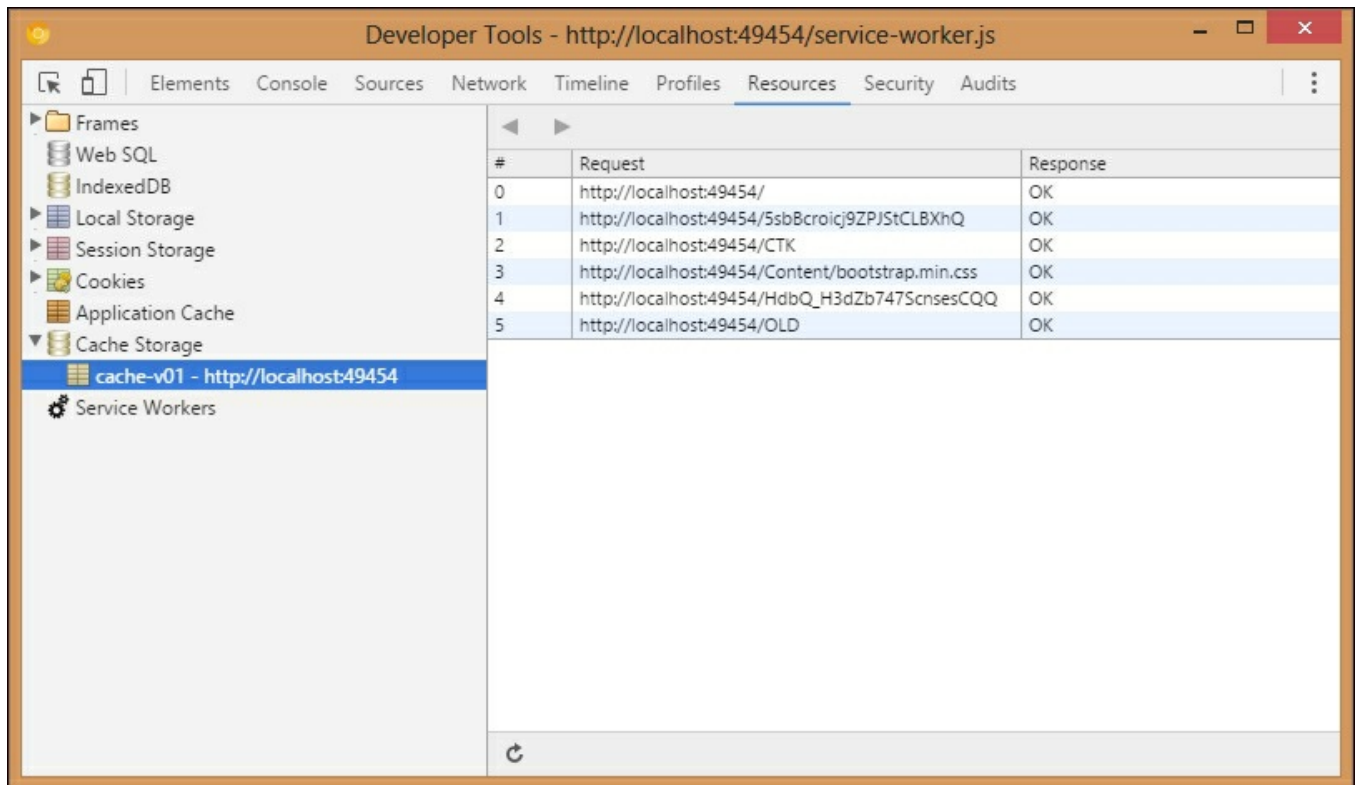
You can see that the cache entries are listed alphabetically and not in the order in which they were added to the cache. These pages will now be a snapshot, fixed at the point in time that they were retrieved. This may not be the functionality that you want!

For simplicity, the service worker that we've built here is a trivial example and you would likely want to expand it to at least handle the case where the network `fetch` fails by adding a `catch` statement. For example, you could serve a previously cached offline fallback page in its place. You should also check that you're not caching error pages from the server, so test the response status code.

You also need to carefully consider your cache invalidation strategy. Service workers give you the tools to build this, as they don't make as many assumptions as the HTML5 AppCache did. For example, you can now programmatically delete entries from the cache.

We'll leave it here for client-side script controlled caching, but you may want to look into this in more detail, especially once the specification has stabilized and browser support is more widespread. There are many other new features now available in JavaScript, which make async programing like this easier than it used to be. For example, arrow functions, which are similar to LINQ lambda expressions in C#.

Web and proxy servers

Caching from a server's point of view is intimately linked to client-side caching in the browser. In addition to storing resources on the server, the headers that you set will be used to control caches everywhere.

The HTTP headers that you set are used by both proxy servers and browsers, including not only standard browsing, but also fetching from a service worker. For example, if the `Cache-Control` header specifies `no-store`, then you won't be able to add the resource to a cache from your worker.

IIS

Internet Information Services (**IIS**) is Microsoft's web server. It can be used to serve content from your ASP.NET application or as a proxy server along with many other things such as FTP. Although IIS does support output caching, the `OutputCache` action attribute is not available in ASP.NET Core. Yet, you can use `ResponseCache` to set the correct headers instead, as covered in [Chapter 4](#), *Addressing Network Performance* .

IIS can also be used as a proxy, for example, in front of the Kestrel web server on a single machine. However, when caching for multiple web servers, you may be better off using dedicated proxy server software such as Varnish.

Varnish

Varnish is a free reverse proxy server that runs on Unix-like operating systems such as **Linux** and **FreeBSD**. You can install it with your package manager (for example, **apt** or **yum**) or provision a proxy server with DevOps software such as **Chef** or **Puppet**. To configure Varnish, you use a **domain-specific language** (**DSL**) called **Varnish Configuration Language** (**VCL**).

Note

You can read more about Varnish at varnish-cache.org .

You shouldn't need to configure Varnish too much if you are using HTTP caching headers correctly. You can also use the custom HTTP `PURGE` method to remove entries from the cache, which works with the Squid proxy software too. You may occasionally see a cryptic **guru meditation** error if Varnish is not properly configured, but you should be able to track down the issue in the Varnish logs. It could indicate that no healthy web servers are available.

Varnish configuration is beyond the scope of this book, but it's very well documented on the Varnish website. If you don't want to run your own proxy server, then you could use a CDN. You may still want your own proxy in addition to using a CDN, as

large CDNs, with many **points of presence (PoP)**, might request the same resource via each PoP, and not share assets across them. This can be an issue if you pay a lot for bandwidth, although some CDNs have a feature (often called origin shielding) that can help with this.

Working with a content delivery network

A content delivery network is commonly used in two ways–as a proxy for offloading your content or as a hosting provider for common third party libraries and frameworks. You can use a dynamic CDN service, such as CloudFlare or Akamai, for the first use case, but the second situation (using a static CDN from Google or Microsoft) is more common and that's what we'll cover here.

Although using a CDN for your libraries, such as jQuery and Twitter bootstrap, is becoming less useful with the adoption of HTTP/2, it can still be helpful for reducing your hosting costs. If you use a popular CDN and library, then the user may also already have a copy. For example, if the user has been to another site that uses jQuery from Google's CDN, then it will already be in their browser cache.

It is essential to have a fallback copy of whatever files you require from a CDN. This is easier than ever with the Razor view engine support built into ASP.NET Core.

The following code shows how jQuery is included in the default MVC Razor layout, for non-development environments. Both the CDN and local versions are specified along with a test.

```
<script
  src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
  asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
  asp-fallback-test="window.jQuery">
</script>
```

The preceding code snippet not only renders the standard `<script>` tag for the Microsoft CDN, but also adds the following inline JavaScript afterwards, which includes the local version if the CDN load fails:

```
(window.jQuery || document.write(
"<script src="\/lib\/jquery\/dist\/jquery.min.js"><\/script>"));
```

Previously, you would have to do this manually, usually in a hurry when your CDN went down. This new helper also works for other scripts and CSS files. For more examples, take a look at `_Layout.cshtml` in the default template.

Tip

It's important to use a secure HTTPS connection to CDN resources in order to avoid mixed content warnings or script loading errors in browsers; most popular CDNs now support HTTPS. For additional information on CDNs, see Chapter 4, *Addressing Network Performance* .

When not to cache

There are certain situations when you shouldn't cache pages or at least you need to be very careful in how you go about it. As a general rule of thumb, caching the rendered output of an authorized page is a bad idea. In other words, if a user has logged into your site, and you are serving them customized content (which could easily be sensitive), then you need to consider caching very carefully.

If you accidentally serve one user's cached content to another, then at best, it will be annoying, as the personalization will be incorrect. At worst, you could expose private information to the wrong person and potentially get into legal trouble.

Note

This is similar to the general rule of not universally enabling CORS if you serve authenticated content. It can be done successfully, but you need to understand the mechanisms in order to configure it to work safely.

For caching, you would need a unique identifier in the URL that can't be guessed. Some dynamic cache control systems, used by network appliances and CDNs, can make use of cookies for this, but it's beyond normal HTTP-based cache control. It is similar to how you might need sticky sessions on a load balancer, because your application was not designed to be stateless.

For authenticated caching, it may be better to not cache at the web level and instead cache at the application level and below. This allows you to cache smaller discrete chunks rather than a whole page, which can enhance reusability.

Application layer caching

Application level (or layer) caching means storing reusable data temporarily inside your infrastructure, but not in the main database. This can be in the memory of your application or web servers, but with multiple servers, this tends to be in a distributed in-memory store such as Memcached or Redis.

You can, of course, use both in-memory stores on your web servers and a centralized cache. However, if you have multiple web servers, then you will need a way to synchronize the caches. You can use **publish-subscribe (pub/sub)** messaging for this, which we will cover later in this chapter.

The following diagram shows a simple centralized caching setup. In a real situation, you would probably have multiple clustered cache servers.
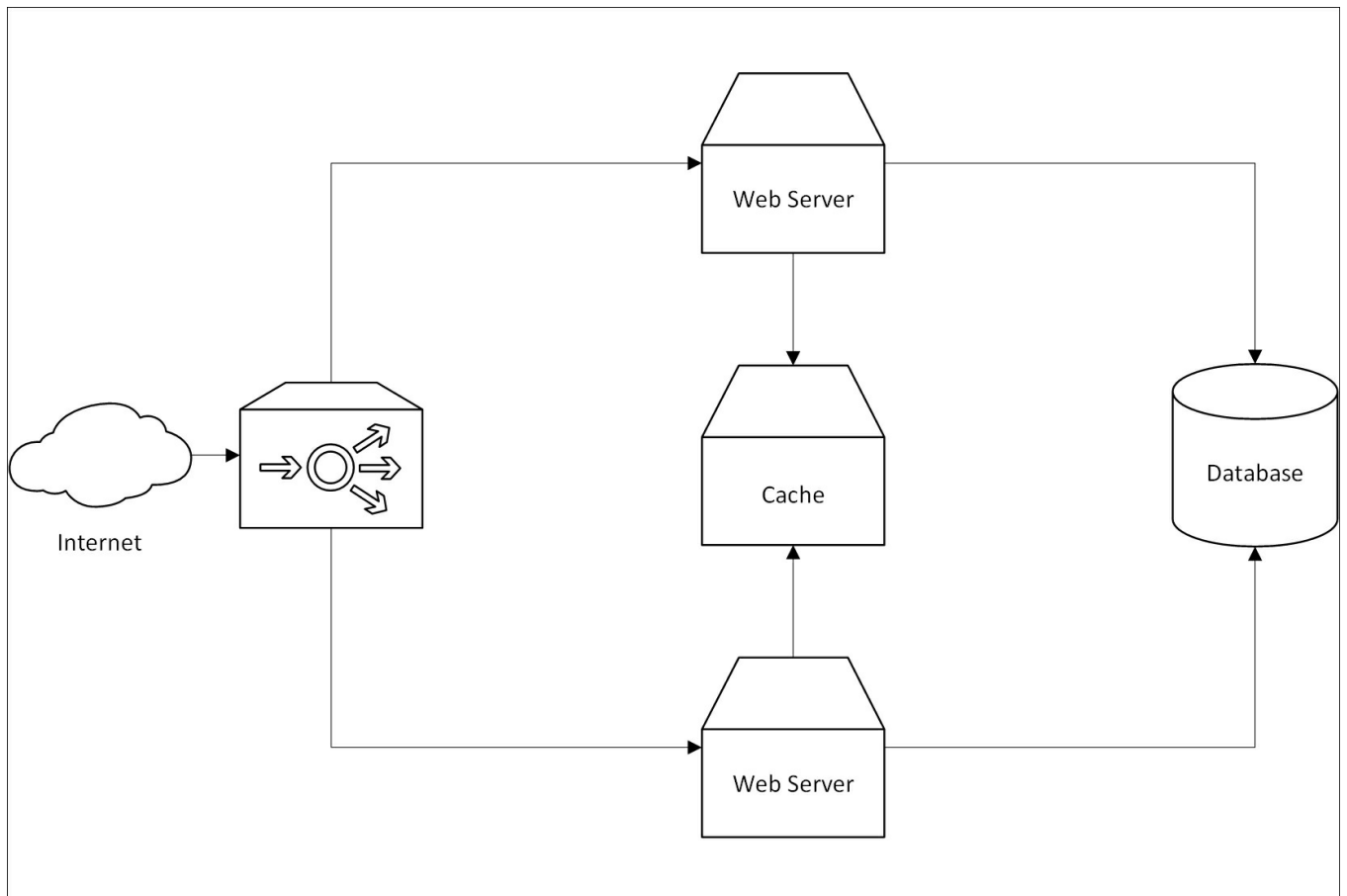
The web servers can now ask the cache if the data they need is in there, before going to the database. This reduces load on the DB and is often quicker, as the data will be in memory if present. If there is a cache miss and the database does need to be queried, then the result can be written to the cache for other servers to use.

Redis

Redis is a popular in-memory store that can also persist data to disk for permanent storage. It runs best on Linux, but a version is available on Windows for development purposes. Redis additionally supports pub/sub messaging, which can be useful for cache invalidation. You can read more about Redis at redis.io .

You may wish to use the Windows version of Redis for local development work, but still deploy to the supported version on Linux. You can get the Windows version at github.com/MSOpenTech/redis or you could run the Linux version in a virtual machine, perhaps using **Docker** or **Vagrant**.

Redis cache is provided as a service on both Azure and AWS (**ElastiCache** offers both Memcached and Redis). You don't need to manage your own server, but because the technology is not cloud-specific, you won't get locked in if you want to migrate in

the future.

As Redis keeps the entire dataset in memory, but, is also able to persist to disk, it can be suitable as a primary data store unlike Memcached. However, it is more commonly used only as a cache, especially if a cloud service is used and it's paired with a cloud database such as **Azure SQL Database** or AWS **Relational Database Service** (**RDS**).

There are two recommended .NET C# clients for Redisa–ServiceStack.Redis and StackExchange.Redis. The Stack Exchange client is used heavily on sites such as Stack Overflow and is easier to use correctly than the the ServiceStack one. You can read more about it at github.com/StackExchange/StackExchange.Redis and install it via NuGet.

If using caching at the application layer, then you will probably need to write a significant amount of custom code. You will also need to work out what format to serialize your data into for storage in the cache. If serving directly to browsers, then JSON could be useful. But if is to be used internally, then you may prefer a binary format such as MS Bond or Protocol Buffers.

Tip

See Chapter 6, *Understanding Code Execution and Asynchronous Operations*, for more on serialization formats and libraries.

Database result set caching

Caching at the database level is similar to application level caching, and it uses similar infrastructure, but requires less custom code. You can use the caching features built into an O/RM, which may make it easier to retrofit.

Note

When we talk about database caching here, we are not referring to caching within the database engine itself. DBs use extensive performance enhancing techniques, such as query caching, and hold lots of their data in memory. However, this is abstracted away from the developer, and the caching we mention here refers to storing the output of a query in an application cache. This is similar to, but subtly different from, the previous section, where you would be storing custom objects.

In the context of O/RMs (such as NHibernate and Entity Framework), this is known as second-level caching. First level caching generally already happens per session, by default, and is used to help avoid things like Select N+1 problems. Second-level caching operates at a level higher than individual transactions and allows you to

share cached data across multiple database sessions over your entire application.

Message queuing

A **message queue** (**MQ**) is an asynchronous and reliable way of moving data around your system. It is useful for offloading work from your web application to a background service, but can also be used to update multiple parts of your system concurrently. For example, distributing cache invalidation data to all of your web servers.

MQs add complexity and we will cover managing this in Chapter 8, *The Downsides of Performance-Enhancing Tools* . However, they can also assist in implementing a **microservices architecture** where you break up your monolith into smaller parts, interfaced against contracts. This can make things easier to reason about within large organizations, where different teams manage the various parts of the application. We will discuss this in more detail in the next chapter, as queues aren't the only way of implementing this style of architecture. For example, HTTP APIs can also be used to do this.

Coffee shop analogy

If using MQs, then you may need to implement extra reconciliation logic for errors occurring in the background. This is best explained with a coffee shop analogy.

If you purchase a takeaway coffee, perhaps in a branch of a popular multinational chain of franchised caffeinated beverage outlets (that dislikes paying tax), then your drink is prepared asynchronously to the payment processing. Typically, you place your order and a barista will start to prepare your coffee, before you have paid for it. Additionally, you will normally pay before receiving your drink. There are many things that could go wrong here, but they are rare enough for the extra cost to be worth it, as it speeds up the ordinary workflow.

For example, you may find that you are unable to pay after placing your order, but the coffee creation process has already begun. This would result in wasted stock, unless there is another customer waiting whom it could be used for. Or perhaps, after you have paid, the barista discovers that a key ingredient for your order is missing. They could either offer you a refund, or negotiate a different drink.

Although more complex, this process is clearly superior to performing the actions in series. If you had to demonstrate that you had the means to pay, your drink was made, and then you completed paying, only one customer could be served at a time. Assuming there are enough staff, payment processing and drink preparation can be performed in parallel, which avoids a long queue of customers.

Running a coffee shop in this way makes intuitive sense and yet, in a web application, it is common to have a relatively long running transaction complete before informing the user of the result. In some situations, it may be better to assume the action will succeed, inform the user of this immediately and have a process in place in case it goes wrong.

For example, payment processing gateways can be slow and unreliable, so it may be better to charge a user's credit card after accepting an order. However, this means that you can no longer handle a failure by showing the user an error message. You will have to use other methods of communication.

When you order items on Amazon, they take payment details immediately, but they process the payment in the background and send you e-mails with the results. If the payment failed, they would need to cancel the order fulfilment and notify you. This requires extra logic, but is quicker than processing the payment transaction and checking stock before confirming the order.

Message queuing styles

There are, broadly, two styles of message queuing–with and without a central broker. With a broker, all messages go through a hub, which manages the communication. Examples of this style include **RabbitMQ**, **ActiveMQ,** and MS **BizTalk**.

There are also brokerless styles (that don't use a broker), where communication between nodes is direct. An example of this style includes ZeroMQ (**Ã˜MQ**), which has a native C# port called **NetMQ**.

Cloud queuing services, including **Azure Service Bus**, Azure Queue storage, and AWS **Simple Queue Service** (**SQS**), are also available. However, as with all non-generic cloud services, you should be wary of getting locked in. There are cloud providers of standard RabbitMQ hosting, which makes migration to your own infrastructure easier down the line if you don't initially want to run your own server. For example, CloudAMQP offers RabbitMQ hosting on multiple cloud platforms.

RabbitMQ implements the **Advanced Message Queuing Protocol** (**AMQP**), which helps to ensure interoperability between different MQ brokers, for example, to allow communication with the **Java Message Service** (**JMS**). Azure Service Bus also supports AMQP, but a big benefit of RabbitMQ is that you can install it on your development machine for local use, without an internet connection.

There is also **Microsoft Message Queuing** (**MSMQ**), which is built into Windows. While this is useful for communication between processes on a single machine, it can be tricky to get it working reliably between multiple servers.

Common messaging patterns

There are two types of common messaging patterns: point-to-point unicast and publish-subscribe. These send messages to a single recipient and many recipients respectively.

Unicast

Unicast is the standard message queuing approach. A message is sent from one service process or software agent to another. The queuing framework will ensure that this happens reliably and will provide certain guarantees about delivery.

This approach is dependable, but doesn't scale well as a system grows, because each node would need to know about all its recipients. It would be better to loosely couple system components together so that they don't need to have knowledge about any of the others.

This is often achieved by using a broker, which has three main advantages:

- By using a broker, you can decouple processes from each other so that they aren't required to know about the system architecture or be alive at the same time. They only care about the message types and the broker takes care of routing the message to the correct destination.
- Broker queues enable an easy distribution of work pattern, especially when combining multiple producers. You can have multiple processes consuming the same queue and the broker will allocate messages to them in a round-robin fashion. This is a simple way of building a parallel system, without having to do any asynchronous programming or worrying about threads. You can just run multiple copies of your code, perhaps on separate machines if constrained by hardware and they will run simultaneously.
- You can easily broadcast or multicast a particular type of message, perhaps to indicate that an event has occurred. Other processes that care about this event can listen to the messages without the publisher knowing about them. This is known as the pub/sub pattern.

Pub/sub

Pub/sub, as the name suggests, is where a software agent publishes a message onto a queue, and other agents can subscribe to that type of message to receive it. When a message is published, all subscribers receive it, but crucially the publisher does not require any knowledge of the subscribers or even need to know how many there are (or even if there are any at all).

Pub/sub is best done with the broker style of message queuing architecture. It can be done without a broker, but it is not particularly reliable. If your use case can tolerate message loss, then you may be able to get away without a broker. But, if you require guaranteed delivery, then you should use one. Using the RabbitMQ broker also allows you to take advantage of exchanges which can perform complex routing of messages.

If you don't want to lose messages, then you need to carefully design your pub/sub system (even if using a broker). A published message that has no subscribers may simply disappear into the ether without a trace and this might not be what you want.

The following diagram shows the differences between simple message forwarding, work distribution, and pub/sub:
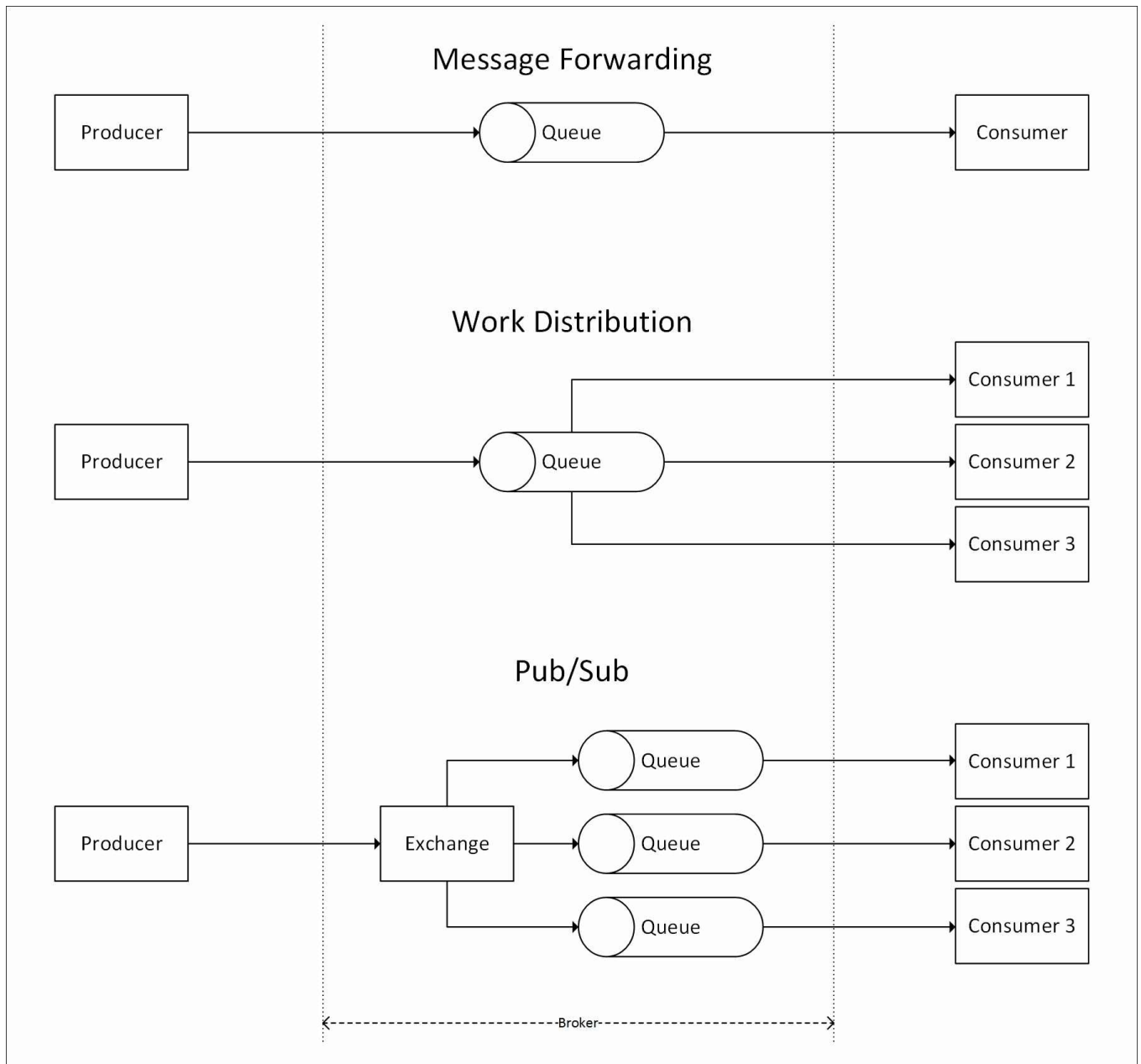
Clearly, if you require a reliable broker, then it needs to be highly available. Typically, you would cluster multiple brokers together to provide redundancy. Using a broker also allows you to write custom rules to define which subscribers receive what messages. For example, your payment system may only care about orders, but your logging server may want to get all messages from all systems.

You will want to monitor not only your broker servers but also the length of the queues. In other words, the number of messages in each queue should always be steady and close to zero. If the number of messages in a queue is steadily growing, this might indicate a problem which your operations team will need to resolve. It may be that your consumers can't process messages faster than your producers are sending them and you need to add more consumers. This could be automated and your

monitoring software could Spin-up an extra instance to scale your system, meeting a temporary spike in demand.

RabbitMQ

RabbitMQ is a free and open source message queuing server. It's written in Erlang, which is the same robust language that WhatsApp uses for its messaging backend.

RabbitMQ is currently maintained by Pivotal (whose labs also make the Pivotal Tracker agile project management tool), but it was originally made by LShift. It was then acquired by VMware before being spun out as a joint venture. It's distributed under the **Mozilla Public License** (**MPL**) v1.1, an older version of the license that the Firefox web browser uses.

The messaging server can be used from many different languages and frameworks such as Java, Ruby, and .NET. This can make it helpful for linking diverse applications together, for example, a Rails app that you want to interface with an ASP.NET Core app or C# service.

Note

You can read more about RabbitMQ and download builds from rabbitmq.com .

RabbitMQ is more modern than systems such as MSMQ and includes features such as an HTTP API and a web admin management interface. Along with HTTP and AMQP, it also supports **Simple Text Orientated Messaging Protocol** (**STOMP**) and **MQTT**, which is useful for lightweight **Internet of Things** (**IoT**) hardware applications. All of these protocols improve interoperability with other messaging systems and they can normally be secured using standard TLS.

The web management interface shows you how many messages are flowing through your queues, and how they are configured. It also allows you to administer your queues (tasks such as purging or deleting messages) and looks something like the following screenshot:

RabbitMQ™

| Overview | Connections | Channels | Exchanges | Queues | Admin |

Overview

▼ **Totals**

Queued messages (chart: last minute) (?)

Ready	■ 0
Unacked	■ 0
Total	■ 0

Message rates (chart: last minute) (?)

Currently idle

Global counts (?)

| Connections: 0 | Channels: 0 | Exchanges: 7 | Queues: 0 |

HTTP API | Command Line

Update [every 5 seconds ⌄]

Last update: 2016-04-06 14:32:06

Queuing frameworks and libraries

You will typically want to use a prebuilt client library or framework to interact with your message queues. There are official libraries in many different languages for the various queuing systems, which offer low-level access. For example, RabbitMQ has an official .NET/C# client library.

However, there are also other opinionated clients and frameworks, which offer a higher level of abstraction for common messaging tasks. For example, **NServiceBus** (**NSB**), which supports RabbitMQ, MSMQ, SQL Server, and Azure, is a commercial offering.

A free alternative to NSB is **MassTransit** (masstransit-project.com), which is a lightweight service bus and distributed application framework. It has also spun out the super convenient **Topshelf** framework (topshelf-project.com), which makes creating Windows services really easy. Neither yet runs on .NET Core, but support for both of these projects is in progress.

One interesting feature of MassTransit (and NSB) is support for sagas. A saga is a complex state machine that allows you to model the story of an entire workflow. Rather than defining individual messages and documenting how they all fit together, you can implicitly capture the golden path and error flows within a saga.

There is also the excellent open source library **EasyNetQ**, which makes implementing pub/sub on RabbitMQ trivial. You can read about it at EasyNetQ.com . Unfortunately, neither the official RabbitMQ client nor EasyNetQ support .NET Core at the time of writing. However, work is in progress for the official client and EasyNetQ has the issue logged.

The RabbitMQ team is working on a new, asynchronous official .NET client, which will only target .NET Core and the **Task Parallel Library** (**TPL**). We covered the TPL, which is part of the parallel extensions that also include PLINQ, in Chapter 6, *Understanding Code Execution and Asynchronous Operations* .

Hopefully, by the time you read this, things will have stabilized and you will be able to use RabbitMQ with .NET Core. Otherwise, you can use the traditional .NET Framework, which may currently be more suitable for the enterprise style applications that normally require message queuing.

However, you could look into using **RestBus**, which is a RabbitMQ library that supports ASP.NET Core. You can read more about it at restbus.org ; it also supports both Web API and ServiceStack.

Note

Library and framework support for .NET Core and ASP.NET Core can change rapidly, so check ANCLAFS.com for the latest information. Feel free to help out and contribute to this list or to any of the open source projects that need porting.

Summary

In this chapter, we have investigated the various tools and methods used for caching and message queuing. These two techniques offer different ways of improving the performance of your system by moving data to other locations and not having one massive monolith do everything.

These are both advanced topics and difficult to cover in such a small space. Hopefully, you have been introduced to some fresh ideas that can help you with solving problems in original ways. If you have discovered a new technology that you think will assist, you're encouraged to read the documentation and specifications for all of the implementation details.

However, before you dive in, you should understand that advanced techniques are complex and have downsides, which can reduce your development speed. In the next chapter, we'll learn about these downsides and discover approaches for managing complexity such as microservices.

Chapter 8. The Downsides of Performance-Enhancing Tools

A lot of the topics that we covered in this book improve performance at a cost. Your application will become more complicated and harder to understand or reason about. This chapter discusses these trade-offs and how to mitigate their impact.

You should implement many of the approaches that you learned so far in this book only if you require them and not just because they are interesting or challenging. It's often preferable to keep things simple if the existing performance is good enough.

You will learn how to make pragmatic choices about what technologies and techniques you should use. You'll also see how to manage the complexities if you choose to use advanced methods.

Topics covered in this chapter include the following:

- Managing complexity with frameworks and architecture
- Building a healthy culture to deliver high performance
- Distributed debugging and performance logging
- Understanding statistics and stale data

Many books and guides only focus on the positives of new tools and frameworks. However, nothing comes for free, and there is always a penalty, which may not be immediately obvious.

You may not feel the effects of the choices that you make, particularly in technical architecture, for a long time. You might not discover that a decision was bad until you try to build on it, perhaps years later.

Managing complexity

One of the main problems with performance-enhancing techniques is that they typically make a system more complicated. This can make a system harder to modify and it may also reduce your productivity. Therefore, although your system runs faster, your development is now slower.

We commonly find this complexity problem in enterprise software, although usually for different reasons. Typically, many unnecessary layers of abstraction are used, supposedly to keep the software flexible. Ironically, this actually makes it slower to add new features. This may seem counterintuitive, until you realize that simplicity makes change easier.

Note

There's a satirical enterprise edition of the popular programmer interview coding test *FizzBuzz*, which is available at fizzbuzz.enterprises . It's good inspiration for how not to do things.

If you don't need a feature yet, then it's often best to leave it out rather than building it just in case you might need it in the future. The more code you write, the more bugs it will have, and the harder it will be to understand. Over-engineering is a common negative psychological trait that is easy to fall victim to if you aren't aware of it, and marketers often exploit this.

For a non-software example, four-wheel drive SUVs are sold to people who will never need their off-road capabilities on the false premise that it may potentially come in useful someday. Yet the financial, safety, environmental, and parking convenience costs outweigh this supposed benefit because it's never used.

We often term this development advice from the **extreme programming** (**XP**) philosophy, **You Aren't Going to Need It** (**YAGNI**). Although we sometimes use slightly different words, the meaning is the same. YAGNI advocates keeping things simple and only building what you immediately need.

This doesn't mean that you should make your software hard to modify. It's still important to stay flexible, just don't add features before you need them. For example, adding an abstraction interface when there is only a single implementation may be overkill. You could easily add it along with the second implementation if and when you build it.

It's difficult to move fast yet not break things when doing so. How you achieve high

reliability in addition to a consistently good development speed will depend on many things that are specific to your situation, such as your team size, organizational structure, and company culture.

One method is to embrace change and develop a system where you can refactor your code in confidence. Using a statically-compiled language, such as C#, is a good start, but you should also have a comprehensive test suite to avoid regressions.

You should design a system so that it is loosely coupled, which means that you can change parts in isolation without a lot of knock-on effects. This also makes it easier to unit test and unit tests are invaluable to refactor in confidence and prevent functional regressions.

We will cover testing and automation with a **Continuous Integration (CI)** workflow in the next chapter. In this chapter, we will talk more about various architectural styles that can help you maintain your application.

Understanding complexity

When learning about new ways of doing things, you should avoid doing them without understanding the reasons. You should know the benefits and downsides and then measure the changes to prove that they are what you expect. Don't just blindly implement something and assume it improves the situation. Try to avoid **cargo cult programming** and always objectively evaluate a new approach.

Note

Cargo cult programming is the practice of emulating something successful but failing to understand the reasons why it works. Its name comes from the cargo cults of the Pacific who built false airstrips after the Second World War to encourage cargo delivery. We use it to describe many things where correlation has been confused with causation.

One example is a company encouraging long hours to deliver a project because they have heard of successful projects where employees worked long hours. However, they fail to understand that the successful project and long hours are both independent byproducts of a highly motivated and competent workforce, and they are not directly related.

It's important to keep code readable, not just for others on your team or new members but also for yourself in the future (when you will forget how something works and why you wrote it in this way). This doesn't simply mean writing helpful explanatory comments in the code, although this is a very good practice. It also applies to source control comments and keeping documentation up to date.

Readability also involves keeping things simple by only making them as complex as they need to be and not hiding functionality in unnecessary layers of abstraction. For example, not using clever programming techniques to reduce the file line-count when a standard structure (for example, a loop or `if` statement) would be more readable and only slightly longer.

It helps to have a standard way of doing things in your team to avoid surprises. Using the same method everywhere can be more valuable than finding a better way of doing it, and then having lots of different ways. If there is consensus, then you can go back and retrofit the better method everywhere where you need it.

Complexity reduction

There are various solutions to manage the complexity that performance-enhancing techniques can add. These usually work by reducing the amount of logic that you need to think about at any one time by hiding the complications.

One option is to use frameworks that standardize how you write your application, which can make it easier to reason about. Another approach is to use an architecture that allows you to only think about small parts of your code base in isolation. By breaking up a complex app into manageable chunks, it becomes easier to work with.

Note

This idea of modularity is related to the **Single Responsibility Principle** (**SRP**), which is the first of the **SOLID** principles (the others are Open/Closed, Liskov substitution, Interface segregation, and Dependency inversion). It is also similar to the higher level **Separation of Concerns** (**SoC**) and to the simplicity of the Unix philosophy. It is better to have many tools that each do one thing well, rather than one tool that does many things badly.

Frameworks

Frontend frameworks, such as **React** (created at Facebook), are designed to reliably build web application views in JavaScript. These help large teams work on a project by simplifying the data flow and standardizing the approach.

Tip

React can be integrated with ASP.NET Core using the **ReactJS.NET** project (reactjs.net). We can use **React Native** to build cross-platform apps that share code across iOS, Android, and the **Universal Windows Platform** (**UWP**), targeting phone, desktop, and Xbox (github.com/ReactWindows). There's also **CodePush** to let you live update your JavaScript apps (including Cordova), without going through an app store (microsoft.github.io/code-push). If you prefer coding in C#, then you can build your cross-platform mobile apps with **Xamarin** (which is now free after Microsoft acquired it). However, we won't go further into any of these technologies in this book.

On the backend, we have the server-side frameworks of .NET Core and ASP.NET Core. Along with C# features, these provide convenient ways of simplifying historically-complicated features. For example, the `async` and `await` keywords hide a lot of the complicated logic associated with asynchronous programming, and lambda functions concisely express intent.

We covered many of these features earlier in this book, so we won't go over them again here. We also highlighted libraries that can make your life easier by hiding boilerplate code for complex operations, for example, `EasyNetQ` and `RestBus`.

Note

Hiding a complex process is never perfect, and you will occasionally come across abstractions that leak some of their implementation detail. For example, when handling exceptions, you may find that the issue you're interested in is now wrapped in an aggregate exception. If you're not careful, then your error logs may no longer contain the detail that you desire.

What we have yet to talk about in detail is the architecture of a web application. Splitting a monolithic system up into discrete parts can not only improve performance, but if done right, it can also make it easier to maintain.

Architecture

In the previous chapter, when discussing message queuing, we briefly covered the microservices architecture. This style is a more modern reimagining of the traditional **Service Oriented Architecture** (**SOA**), and although using reliable MQ communication is preferred, we can also perform this with **representational state transfer** (**REST**ful) HTTP APIs.

Typically, we build a traditional web app as a single application or monolith. This is common if the app has grown organically over an extended period of time, and this is a perfectly acceptable practice. It's a poor decision to over-engineer too early before there is any need, which may never materialize.

Excessive popularity is a nice problem to have, but don't optimize for this prematurely. This isn't an excuse to make things unnecessarily slow, so be sure to understand the tradeoffs involved.

Tip

Using a monolithic architecture is not an excuse to build something badly, and you should plan for expansion, even if you do not implement it immediately. You can keep things simple while still allowing for future growth.

Although the application is a single unit, you should split the code base into well-organized modules, which are linked together in a simple, logical, and easy-to-understand way. Refer to the SOLID principles, mentioned previously.

If a monolithic application can't easily scale to meet user demand and is performing

poorly as a result, then you can split it up into smaller separate services. You may also wish to split an app up if it has become too cumbersome to iterate on quickly and development speed has slowed.

Monolith versus microservices

The following diagram shows some differences between a typical monolith and a microservices architecture:
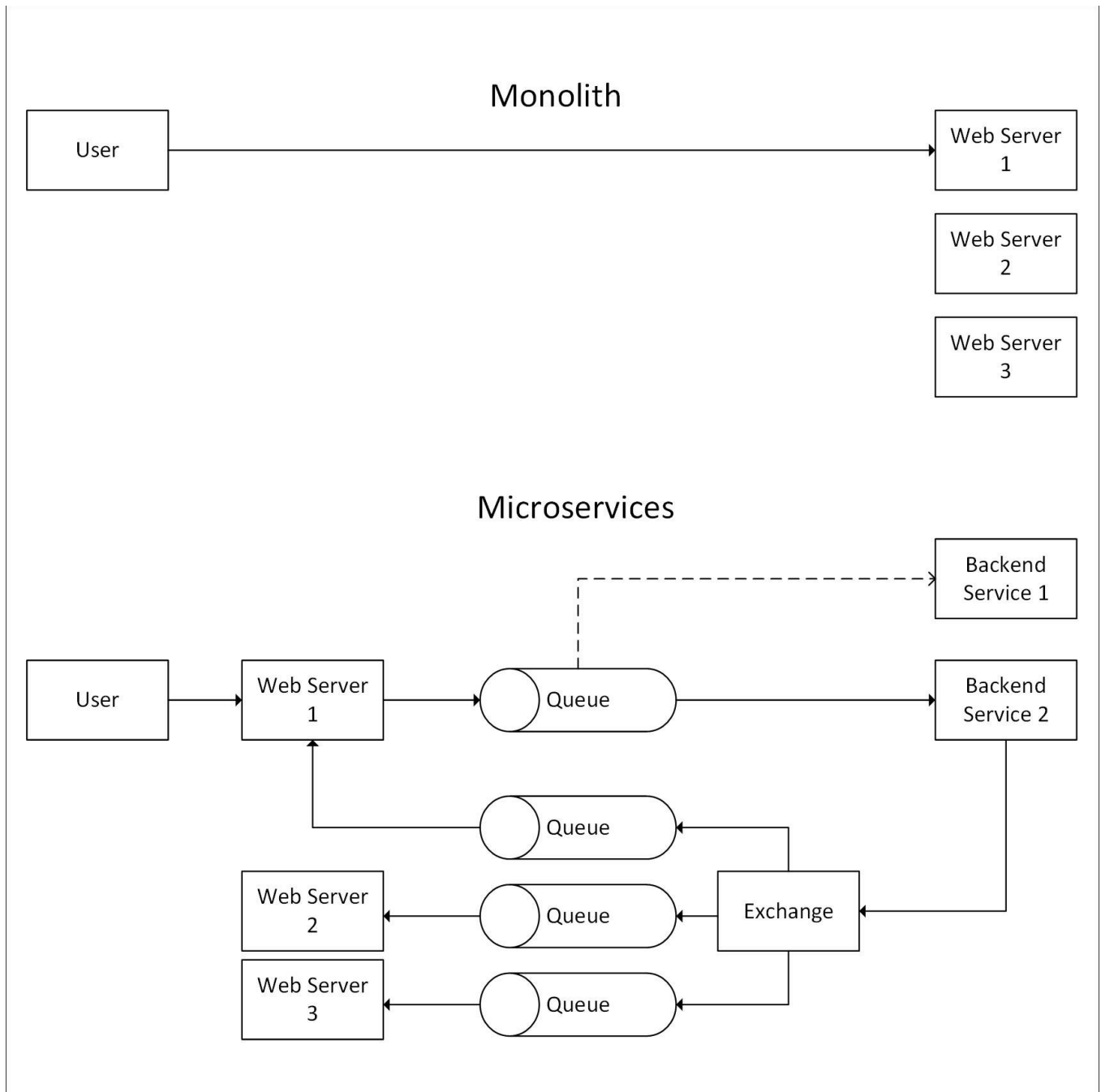
Here, the user makes a request to an application running on a web farm. We have omitted firewalls, load balancers, and databases for clarity, but multiple web servers are shown to illustrate that the same codebase runs on multiple machines.

In the initial monolith architecture, the user communicates directly with a single web server. This is ideally per request/response pair. However, if the application was poorly designed and holds state in memory, then **sticky sessions** may cause the load to pool on certain servers.

The second example in the diagram of a microservices architecture is obviously

more complicated but also more flexible. The user again sends a request to a web server, but instead of doing all of the work, the server puts a message into a queue.

The work in this queue is distributed between multiple backend services of which the first one is busy, so a second service picks up the message. When the service completes, it sends a message to an exchange on the message broker, which uses a pub/sub broadcast to inform all of the web servers.

One added piece of complexity is that the architecture should have already sent the response to the user's original web request, so you need to consider the **user experience** (**UX**) more carefully. For example, you can display a progress indicator and update the status of this with an asynchronous WebSocket connection.

Architecture comparison

The monolith approach is simple, and you can just add more web servers to handle additional users. However, this approach can become cumbersome as an application (and development team) grows larger because the slightest change requires a full redeployment to each web server.

In addition, a monolith is easy to scale vertically (up) but hard to scale horizontally (out), which we covered previously. However, a monolith is easier to debug, so you need to be careful and have good monitoring and logging. You don't want any random outage investigation to turn into a murder mystery hunt because of unnecessarily implemented microservices.

Note

Historically, Facebook had a deployment process that consisted of slowly compiling their PHP code base to a gigantic gigabyte-scale binary (for runtime performance reasons). They then needed to develop a modified version of BitTorrent to efficiently distribute this huge executable to all of their web servers. Although this was impressive infrastructure engineering, it didn't address the root cause of their **technical debt** problem, and they have since moved on to better solutions, such as the **HipHop Virtual Machine** (**HHVM**), which is similar to the .NET CLR.

If you wish to practice continuous delivery and deploy multiple times a week (or even many times a day), then it's advantageous to break your web application up. You can then maintain and deploy each part separately, communicating with each other using messages against an agreed API.

This separation can also help you use agile development methodologies–for example, using many smaller teams—€"rather than big teams because smaller teams perform

better.

Your instrument of control to scale a monolith is very crude, as all work is done in one place. You can't scale parts of your app independently to the other components. You can only scale the whole thing even if the high load is concentrated in a small part. This is analogous to a central bank only having control of a single interest rate as a lever, which affects many things at once. If your app is distributed, then you only need to scale the part that requires it, avoiding over provisioning and reducing costs.

A well-used (and often rephrased) quote from Abraham Maslow goes:

> *"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."*

This is known as the law of the instrument and is related to the confirmation bias. If you only have one tool, then you are likely to use this tool for everything and only see things that support your existing ideas. This commonly applies to tools and frameworks, but it can also apply to scaling techniques.

The first step towards modularity may be to split a big web application into many smaller web apps so that you can deploy them separately. This strategy can have some benefits, but it also has many limitations. Some logic may be unsuitable to host in a web app, for example, long-running processes, such as those monitoring a filesystem or used to manipulate media.

You may also find that you need to duplicate code that was shared in the original monolith. If you adhere to the **Don't Repeat Yourself** (**DRY**) doctrine, then you might extract this functionality to a library. Yet, now you have dependencies and versioning to manage. You'll also need processes to build, package and host your library in a private repository, all of which can slow down development and reduce your agility.

Note

Sometimes, we also refer to DRY as **Duplication Is Evil** (**DIE**), and this is the sensible idea that an implementation should only occur in a unique location. If other code requires this functionality, then it should call the original function and not have copied code pasted in. This means that you only need to make a change in a single place to apply it everywhere.

A more advanced approach is to extract functionality into a separate service, which can handle many different web applications. If designed correctly, then this service won't need to have any knowledge of these apps and will simply respond to messages. This allows you to add a new web app without making changes to the code

of other applications or services.

Refactoring

Refactoring a monolith into services can be tricky if the application has become a tightly-coupled big-ball-of-mud, but if it's been well-built (with plenty of test coverage and loosely-coupled code), then it shouldn't be too taxing. There's a big difference between a well-built monolith and a big-ball-of-mud made with little thought.

It's worth expanding on test coverage, as unit tests are essential for successful refactoring without introducing regressions or new bugs. Unit tests allow you to refactor and tidy up in confidence, and they prevent the creation of code that is fragile, which developers are afraid to touch. We will cover testing in greater detail, including automation, in Chapter 9, *Monitoring Performance Regressions* .

Both of these design patterns may appear superficially similar, but internally, it's a different story. The following diagram illustrates the differences between a well-architected monolith and a messy big-ball-of-mud:
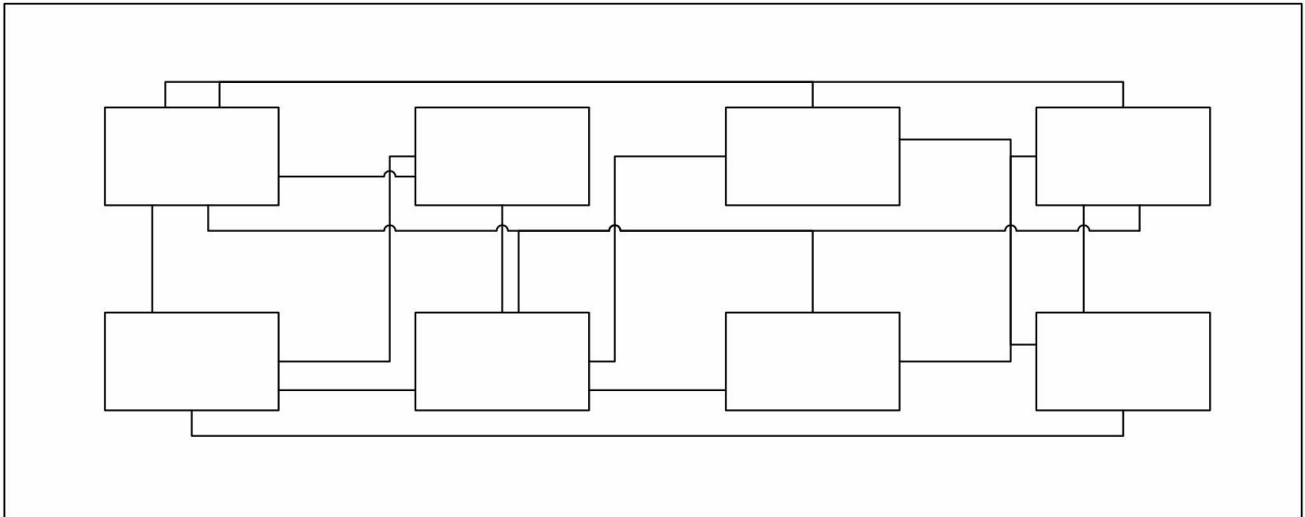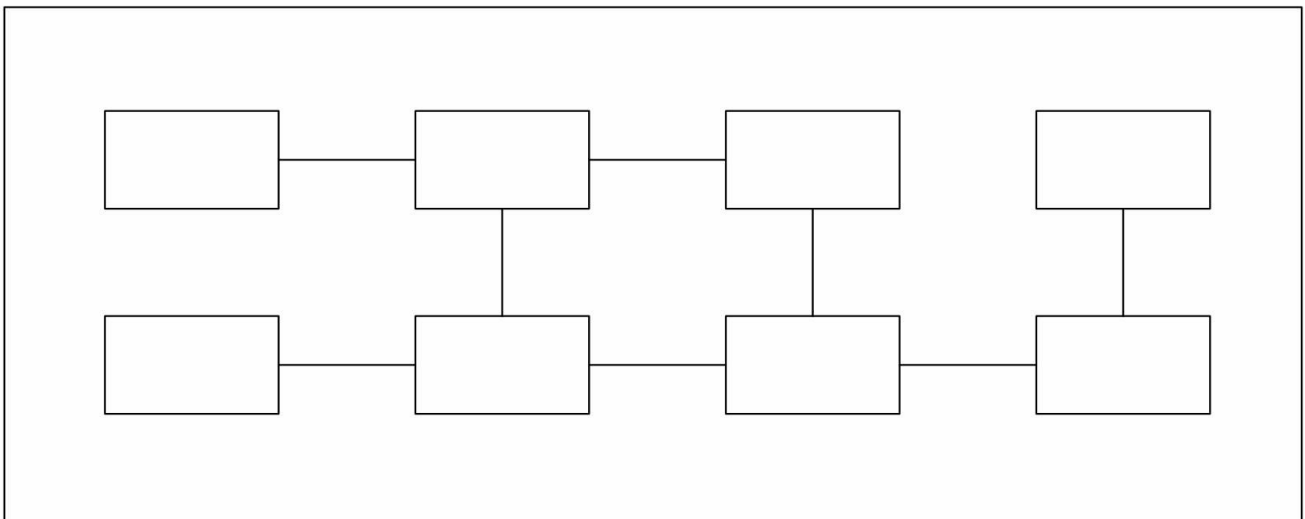
The large boxes represent the code running on each web server (the **Web Server N** boxes from the monolith example in the earlier diagram). From the outside, they look the same, but the difference is in the internal coupling of the code.

The big-ball-of-mud is a tangled mess with code referencing other functions throughout the application. The monolith is well-structured with a clear separation of concerns between different modules.

Changes to code in the big-ball-of-mud can have unexpected side effects because other parts of it may rely on the implementation details of what you are modifying. This makes it brittle, difficult to alter, and developers may be afraid of touching it.

The well-built monolith is easy to refactor and split out into separate services because the code is neatly-organized. It uses abstract interfaces to communicate between modules, and code doesn't reach into another class's concrete implementation details. It also has excellent unit test coverage, which runs automatically.

Although both are a single code base, the quality of the monolith is much higher because it is well-distributed internally. Good design is important for future-proofing, as we build in and allow for expansion. The monolith hasn't been split up too early (before scaling was required), but the design makes this easy to do later. In contrast, the big-ball-of-mud has accumulated a large amount of technical debt, which it needs to pay off before we can make any further progress.

Tip

Technical debt (or tech debt for short) is the concept of not finishing a job or cutting corners, which could lead to more difficulties until it is paid back. For example, failing to properly document a system will make altering it later more difficult.

Tech debt is not necessarily a bad thing if it is deliberately taken on in full knowledge, logged, and paid back later. For example, waiting to write documentation until after a release can speed up delivery. However, tech debt that is not paid back or is accumulated without knowledge (simply due to sloppy coding), will get worse over time and cause bigger issues later.

The best way to deliver a high-quality and flexible application (that's easy to refactor) is to have a competent and conscientious development team. However, these attributes can often be more about culture, communication, and motivation than simply skill or raw talent. Although this isn't a book about team management, having a healthy culture is very important, so we'll cover a little of this here. Everyone in an operations and software development department can help create a positive and friendly culture, even if you usually need buy-in from higher up as well.

Tip

Tools are also very useful in refactoring and testing. We covered the Visual Studio IDE and ReSharper plugin previously, but there are many more tools for testing. We will cover more testing tools, including automation, in Chapter 9, *Monitoring Performance Regressions* .

A culture of high performance

If you want to achieve high performance, then it's important to foster a company culture that encourages this and recognizes performance as vital. Culture can't just come from the bottom up only involving engineers, it also needs to come from the top-down and management must buy in to the performance prerogative.

Note

This section is not very technical, so feel free to skip it if you don't care about management or the human side of software development.

A blameless culture

The most important attributes of a high-performance culture are that it should be open and blameless. Everyone needs to be focused on achieving the best possible outcomes through measuring and learning. Attributing fault to individuals is toxic to delivering great software, and this is not only the case when it comes to performance.

If something goes wrong, then it is a process problem and the focus should be on improving it and preventing repeat mistakes in the future, for example, by automating it. This is similar to how safety-critical industries, such as air travel, behave because they recognize that blaming people discourages them from raising issues early before a disaster occurs.

A related philosophy is the Japanese process of Kaizen, which encourages continuous improvement by everyone. The car manufacturer Toyota pioneered Kaizen practices to improve the efficiency of their production line, and most automotive companies and many other different industries have since adopted them.

Some industries also have processes to encourage whistle-blowing that protect the individuals raising concerns. However, if this is required in web application development, then it's a sure sign that the culture needs work. Developers should feel that they are able to directly raise concerns bypassing their line manager without consequence. If everyone's opinion is respected, then this shouldn't even be necessary.

Intellectual dishonesty

If team members get defensive when ideas are challenged, then this is a sign that things may not be working well. Everybody makes mistakes and has gaps in their knowledge, a truth the best engineers embrace. You should strive for a culture where everyone is open to new ideas and is always asking questions.

If people are unable to accept constructive criticism and have their ideas challenged, then they may lack confidence and be covering up a lack of competence. Experienced developers know that you never stop learning, they admit their ignorance and are always open to offers of improvements.

Being closed to suggestions alters the behavior of others, and they will stop raising small issues early. This results in open secrets about poor quality, and the first that is known about a problem is at release time, at which point everything is much worse (or on fire).

This is not an excuse to be nasty to people, so always try to be nice and gently explain the reasons behind your criticism. It's easy to find fault in anything, so always propose an alternative approach. If you are patient, then a reasonable person will be grateful for the learning opportunity and appreciate gaining experience.

A good rule to follow is "don't be a jerk" and treat others as you would like to be treated, so be kind and think about how you would feel if the situation was reversed. Just remember that being nice is not always compatible with doing the right thing.

A little self-deprecation can go a long way to making you more approachable, rather than simply dictating the one-true-way. However, you should make it clear when something is a joke or tongue in cheek, especially when dealing with cultures that are more direct or when communicating textually.

For example, North Americans are often less subtle than and not as sarcastic as the British (who also spell some word differently, and some would say more correctly). Obviously, use your own judgment because this may be terrible advice and could cause offense or, even worse, a full-on diplomatic incident. Hopefully, it is self-evident that this whole paragraph is tongue in cheek.

People who have integrity and confidence in their ideas can afford to be modest and self-deprecating, but internal company culture can influence this too. A particularly bad practice is to conduct performance reviews by *laddering* (also known as stack ranking), which involves putting everyone in order, relative to everybody else. This is toxic because it rewards people who focus more on marketing themselves than

those who recognize the deficiencies in their technical skills and try to improve them. In the pathological case, all of the best people are forced out, and you end up with a company full of sociopathic sharp suits, who are technically illiterate or even morally bankrupt.

Slow down to go faster

Sometimes, the company must allow the development team to slow down on feature delivery in order to focus on performance and resolving technical debt. They should be given time to be thoughtful about design decisions so that they can cogitate avoiding premature generalization or careless optimization.

Performance is a significant selling point of software, and it is much easier to build quality throughout the development process than in a polishing phase at the end. There is a significant body of evidence that suggests that good performance improves **Return on Investment** (**RoI**) and creates customers. This is especially true on the web, where poor performance decreases conversion rate and search engine ranking.

Having a healthy culture is not only important for the runtime performance of your software but also for the speed at which you can develop it. The team should be encouraged to behave rigorously and write precise, but also concise, code.

You get what you measure, and if you only measure the rate of feature delivery, or even worse simply **Lines of Code** (**LoC**) written, then quality will suffer. This will hurt you in the long run and is a false economy.

False economies are when you make short term cost saving measures that actually lose you more money in the long term. Examples of this are skimping on hardware for developers or interrupting someone in the middle of coding with something trivial, which could easily wait until later, and forcing them to switch contexts.

Another nonsoftware example is a shortsighted government making cuts to investment in research by sacrificing long-term growth for short-term gain, possibly due to the lack of any long-term economic plan.

Note

Hardware is significantly cheaper than developer time. Therefore, if everyone on your team doesn't have a beefy box and multiple massive monitors, then productivity is being needlessly diminished. Even over a decade ago, Facebook's software developer job adverts listed dual 24-inch widescreen monitors as a perk.

From the ground up

In a healthy and progressive culture, it can be tempting to rewrite poor quality software from scratch, perhaps in the latest trendy framework, but this is usually a mistake. Released software is battle-hardened (no matter how badly it was built), and if you rewrite it, then you will probably make the same mistakes again, for example, re-implementing bugs that you already patched. This is especially true if the software was not built with good unit test coverage, which can help prevent regressions.

The only case where you could reasonably rewrite an application from the ground up is if you had deliberately made a prototype to explore the problem space with the sole intention of throwing it away. However, you should be very careful because if you've actually built a **Minimum Viable Product** (**MVP**) instead, then these have a habit of sticking around for a long time and forming the foundations of larger applications.

A better approach to a full rewrite is to add tests to the application (if it doesn't already have them) and gradually refactor it to improve the quality and performance. If you built an application in such a way as to make it difficult to unit test, then you can start with **User Interface** (**UI**) tests, perhaps using a headless web browser. We will cover testing more, including performance testing, in the next chapter.

Shared values

Culture is really just a set of shared values–things such as openness, sustainability, inclusivity, diversity, and ethical behavior. It can help having these values formally documented so that everyone knows what you stand for.

You may have a progressive open salary policy so that others can't use secret earning information as a tool to pay people less. However, this would need to apply universally, as it's unhealthy to have multiple conflicting cultures because this can precipitate an us-versus-them attitude.

There's plenty more to say about culture, but, as you can see, there are many competing concerns to balance. The most important idea is to make intentional and thoughtful tradeoffs. There isn't one correct choice, but you should always be conscious of the consequences and appreciate how tiny actions can alter team performance.

The price of performance

Developers should have an idea of the available budget for performance and understand the cost of the code that they write, not just in execution throughput but in readability, maintainability, and power efficiency. Throwing more cores at a unit of work is not nearly as good as refactoring it to be simpler.

Efficiency has become increasingly important, especially with the rise of mobile devices and cloud computing time-based usage billing. Parallelizing an inefficient algorithm may solve a performance problem in the time domain, but it's a crude brute-force approach and altering the underlying implementation may be better.

Less is often more and sometimes doing nothing is the best approach. Software engineering is not only about knowing what to build but what not to build. Keeping things simple helps others on your team use your work. You should aim to avoid surprising anyone with nonobvious behavior. For example, consider that you build an API and then give it conventional defaults; if this could potentially take a long time, then make the methods asynchronous to indicate this fact.

You should aim to make it easy to succeed and hard to fail when building on your code. Make it difficult to do the wrong thing, for example, if a method is unsafe, name it to document this fact. Being a competent programmer is only a small part of being a good developer, you also need to be helpful and proficient at communicating clearly.

In fact, being an expert programmer can be a downside if you don't deliberately keep things simple to aid the understanding of others on your team. You should always balance performance improvements against the side effects, and you shouldn't make them at the expense of future development efficiency without good reason.

Distributed debugging

Distributed systems can make it difficult to debug problems, and you need to plan for this in advance by integrating technology that can help with visibility. You should know what metrics you want to measure and what parameters are important to record.

As you run a web application, it isn't a case of deploy and forget as it might be with mobile apps or desktop software. You will need to keep a constant automated eye on your application to ensure that it is always available. If you monitor the correct performance metrics, then you can get early warning signs of problems and can take preventative action. If you only measure uptime or responsiveness, then the first that you may know of a problem is an outage notification, probably at an unsociable hour.

You may outsource your infrastructure to a cloud-hosting company so that you don't have to worry about hardware or platform failures. However, this doesn't completely absolve you of responsibility and your software will still need continuous monitoring. You may need to architect your application differently to work in harmony with your hosting platform and scale or self-heal when issues arise.

If you design your system correctly, then your web application will run itself, and you'll rarely get notified of actions that require your attention. If you can successfully automate all of the things, rather than babysitting a live deployment, then that's more time you can use to build the future.

In a distributed architecture, you can't simply attach a debugger to the live web server, not that this is a good idea even when possible. There is no live server anymore, there are now many live servers and even more processes. To get a holistic picture of the system, you will need to simultaneously examine the state of multiple modules.

There are many tools that you can use to help with centralizing your debug information. You can retrofit some of them. However, to get the most out of them, you should decide what to measure upfront and build telemetry capabilities into your software from the start.

Logging

Logging is vital to a high-performance application, so while it is true that logging adds some overhead and can slow down execution, omitting it would be short-sighted and a false economy. Without logging, you won't know what is slow and requires improvement. You will also have other concerns, such as reliability, for which logging is essential.

Error logging

You may have used (or at least be familiar with) the excellent ASP.NET package called **Error Logging Modules and Handlers** (**ELMAH**) to catch unhandled exceptions (elmah.github.io). ELMAH is great for already existing applications, as you can drop it into a live running web app. However, it's preferable to have error logging built into your software from the start.

Unfortunately, ASP.NET Core does not support ELMAH, but there is a similar package called **Error Logging Middleware** (**ELM**). Adding this to your web application is just as simple as installing Glimpse, but it is really just a prototype and doesn't have all the features of ELMAH. First, add the `Microsoft.AspNetCore.Diagnostics.Elm` NuGet package to your project, as shown in the following image:
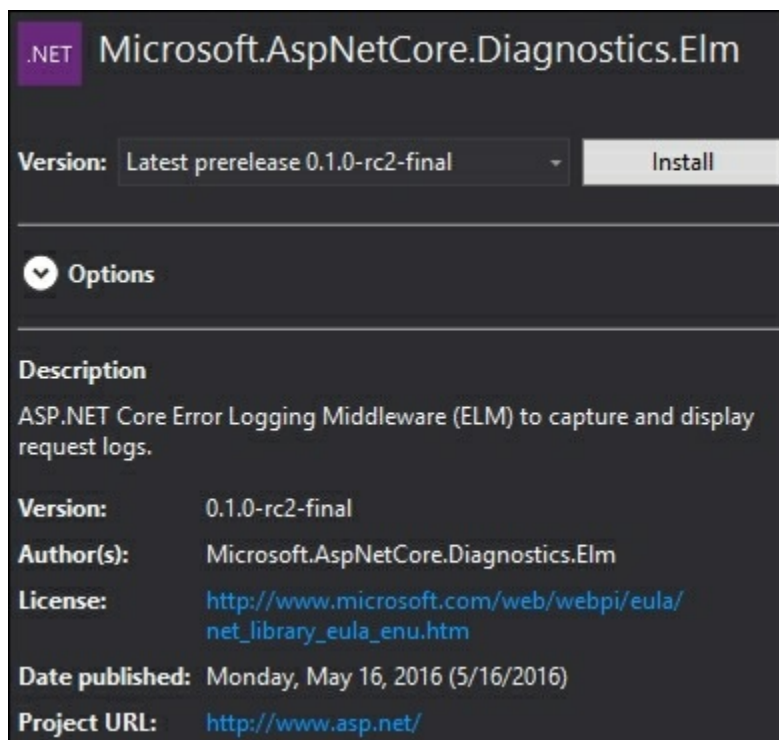
Then, in the `ConfigureServices` method of the `Startup` class, add the following line of code:

```
services.AddElm();
```

You also need to add the following lines to the `Configure` method of the same class:

```
app.UseElmPage();
app.UseElmCapture();
```

Tip

To start with, you may want to put these in the `env.IsDevelopment() if` statement so that they are only active on your workstation. If you use ELM on production, then these logs will definitely need securing.

You can now visit the `/Elm` path of your web application in your browser to see the logs, as shown in the following image. Use the search box to filter results, click on **v** at the end of each line (not shown) to expand the entry's details, and click on ^ to collapse it again:

ASP.NET Core Logs

| Debug ∨ | | filter |

Clear Logs

Path	Method	Host	Status Code
RequestId:0HKS4EMQJE8H4 RequestPath:/Elm			
Non-scope Log			
Non-scope Log			
Non-scope Log			
Non-scope Log			
/images/banner3.svg	GET	localhost:53820	200
Non-scope Log			
/lib/bootstrap/dist/fonts/glyphicons-halflings-regular.woff2	GET	localhost:53820	200
Non-scope Log			
/images/banner4.svg	GET	localhost:53820	200
Non-scope Log			
/images/banner2.svg	GET	localhost:53820	200
/images/banner1.svg	GET	localhost:53820	200
Non-scope Log			
Non-scope Log			
Non-scope Log			
/js/site.js	GET	localhost:53820	200
/lib/bootstrap/dist/js/bootstrap.js	GET	localhost:53820	200

Note

When using ELM, ELMAH, or even the default error pages, you should be careful to not show detailed exceptions or stack traces to real users. This isn't simply because they're unfriendly and unhelpful, they are a genuine security concern. Error logs and stack traces can reveal the internal workings of your application, which malicious actors can exploit.

ELM is fairly basic, but it can still be useful. However, there are better solutions already built into the default templates–things, such as the integrated support for

logging and **Application Insights**.

Application Insights

Application Insights allows you to monitor the performance of your application and view requests or exceptions. You can use it with Azure, but it will also work locally without needing an Azure account. You can easily enable Application Insights by keeping the relevant checkbox ticked, as shown in the following image, when you create a new ASP.NET Core web application project in Visual Studio, and you don't need to sign in to Azure to do this:
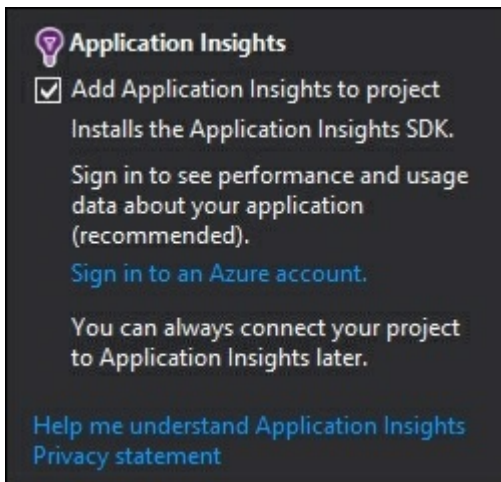
Tip

The Application Insights template option is only available under the **Web** templates not the **.NET Core** templates.

Build and run the new project, then open the **Application Insights Search** window to see the output. Navigate around the web app in your browser, and you will see records start to appear, which should look something like the following:

Tip

If you don't see any records in the output, then you may need to update the NuGet package (`Microsoft.ApplicationInsights.AspNetCore`) or click on the Search icon.

You can filter by event type, for example, requests or exceptions, and there are more detailed filters to refine your search. You can select an individual record to see more details, which will look something like the following:

Although you can access this information locally without Azure, you can choose to
select **Configure Application Insights** and set up logging of data to your Azure
account.
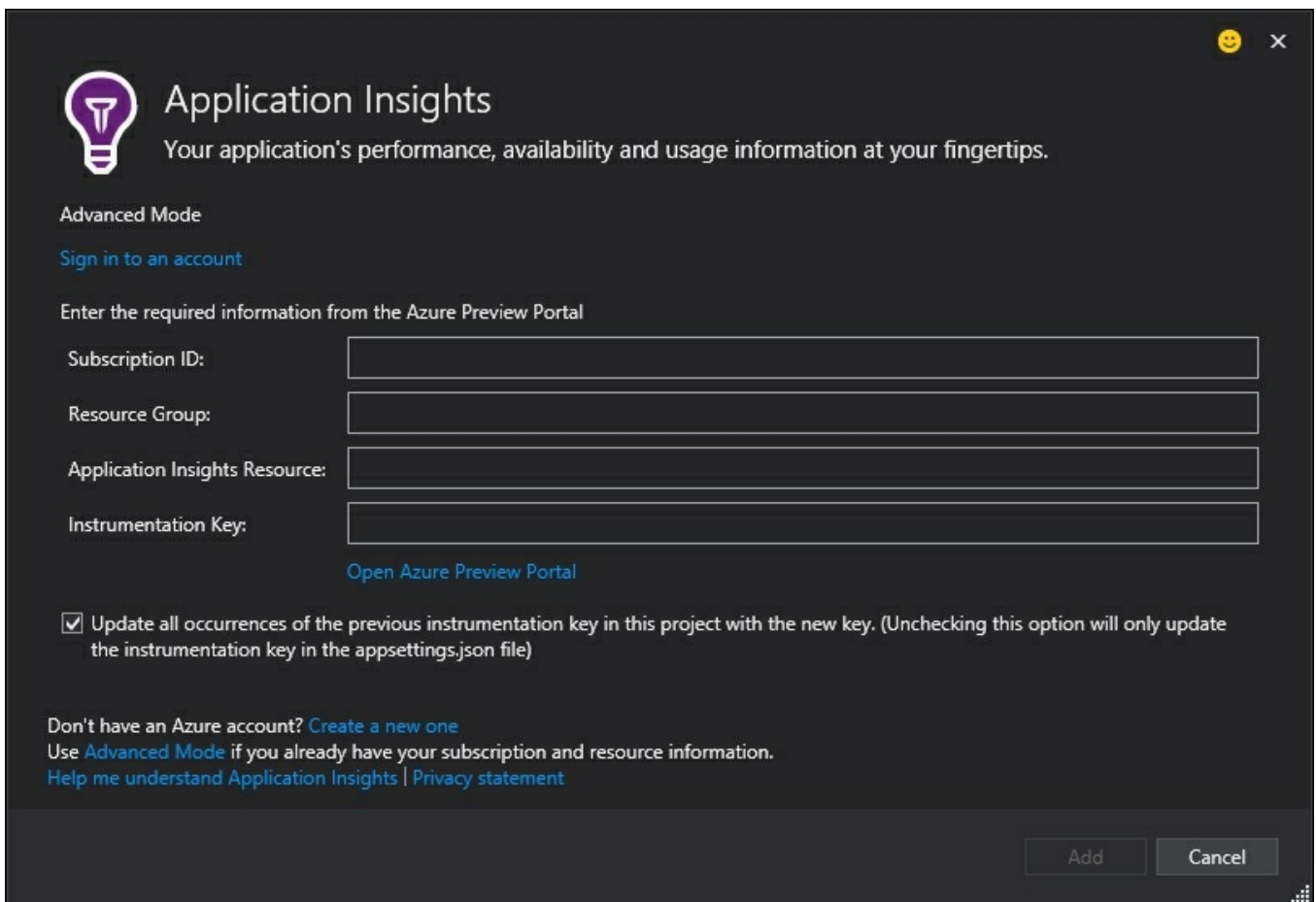
If you know what you're doing, then you can use **Advanced Mode** and enter your **Subscription ID** and **Instrumentation Key** manually:

Note

Secret configuration information, such as this, is now stored outside of the source tree to help prevent it leaking. You can now define secrets as environment variables or in a secrets store held in your user profile.

Once you set up Azure, then you can view the telemetry results online and aggregate information from many diverse systems. We don't have space to cover this in more detail here, but you can find more information online at azure.microsoft.com/en-us/documentation/services/application-insights .

Integrated logging

Logging is now built into ASP.NET Core (in the `Microsoft.Extensions.Logging` package) as well as **Dependency Injection** (**DI**), and both are included in the default templates. This reduces the barrier to entry, and it's now trivial to use these helpful

technologies in even the smallest of projects.

Previously, you would have to add logging and DI libraries before you started, which could put many people off using them. If you didn't already have a standard choice for both, then you would need to wade through the plethora of projects and research the merits of each.

Tip

You can still use your preferred libraries if you are already acquainted with them. It's just that there are now sensible defaults, which you can override.

Logging is configured in the `Startup` class, and if you use the standard web application template, then this will already be included for you. The logger factory reads settings from the `appsettings.json` file. In here, you can configure the logging level, and by default, the relevant section looks like the following:

```
"Logging": {
  "IncludeScopes": false,
  "LogLevel": {
    "Default": "Debug",
    "System": "Information",
    "Microsoft": "Information"
  }
}
```

This sets a very chatty log level, which is useful for development, but you will probably want to only log warnings, and errors when you run this in production.

Note

Along with Application Insights and logging, useful debugging tools included by default in the `Startup` class include a developer exception page, which is like an advanced version of the old **Yellow Screen Of Death** (**YSOD**) ASP.NET error page. Also included is a database error page, which can helpfully apply EF migrations and mitigates a common pitfall of previous EF code-first migration deployments.

To add the logger to your MVC home controller via constructor injection, you can use the following code (after adding `using Microsoft.Extensions.Logging;` to your `using` statements at the top of the file):

```
private readonly ILogger Logger;
public HomeController(ILoggerFactory loggerFactory)
{
    Logger = loggerFactory.CreateLogger<HomeController>();
}
```

After this, you can use `Logger` to log events inside action methods, as follows:

```
Logger.LogDebug("Home page loaded");
```

There are many more logging levels available and overloaded methods so that you can log additional information, for example, exceptions. We won't go into any more detail here, but you don't simply have to log text events, you can also record execution times (perhaps with a stopwatch) or increment counters to see how often certain events occur. Next, we'll see how to view these numbers centrally and read them correctly.

Note

For more examples of how to use logging, you can examine the default account controller (if you included individual user accounts authentication in the template). For more information, you can read the documentation at docs.asp.net/en/latest/fundamentals/logging.html .

Centralized logging

Logging is great. However, in a distributed system, you will want to feed all of your logs and exceptions into a single location, where you can easily analyze them and generate alerts. One potential option for this is Logstash (elastic.co/products/logstash), which we fleetingly mentioned in Chapter 2, *Measuring Performance Bottlenecks* .

If you prefer a more modular approach and want to record performance counters and metrics, then there is **StatsD**, which listens for UDP packets and pushes to Graphite for storage and graphing. You can get it at github.com/etsy/statsd and there are a few .NET clients listed on the wiki, along with example C# code in the main repository.

You may wish to use message queuing for logging so that you can quickly put a logged event into a queue and forget about it, rather than directly hitting a logging server. If you directly call an API (and aren't using UDP), then make sure that it's asynchronous and nonblocking. You don't want to slow down your application by logging inefficiently.

There are also cloud options available, although the usual caveats about lock-in apply. AWS has **CloudWatch**, which you can read more about at aws.amazon.com/cloudwatch . Azure Diagnostics is similar, and you can integrate it with Application Insights, read more at azure.microsoft.com/en-us/documentation/articles/azure-diagnostics .

There are other cross-platform cloud services available, such as New Relic or

Stackify, but these can be quite expensive, and you may wish to keep your logging within your own infrastructure. You could shoehorn the data into analytics software, such as **Google Analytics** or the privacy-focused Piwik (which is open source and can be self-hosted), but these are less suitable because they're designed for a slightly different purpose.

Statistics

When interpreting your collected metrics, it helps to know some basic statistics in order to read them correctly. Taking a simple *mean* average can mislead you and may not be as important as some other characteristics.

When instrumenting your code to collect metadata, you should have a rough idea of how often particular logging statements will be called. This doesn't have to be exact, and a rough order of magnitude approximation (or **Fermi estimate**) will usually suffice.

The question you should try to answer is how much of the data should be collected, all of it or a random sample? If you need to perform sampling, then you should calculate how big the sample size should be. We covered sampling in relation to SQL in Chapter 5 , *Optimizing I/O Performance* , and the idea is similar here. Performance statistics require the same level of rigor as benchmarking does, and you can easily be misled or draw incorrect conclusions.

StatsD includes built-in support for sampling, but there are many other approaches available if you want to investigate them. For example, online streaming algorithms and reservoirs are two options. The important thing to keep in mind for performance is to use a fast **random number generator** (**RNG**). As, for sampling, this doesn't need to be cryptographically secure, a **pseudorandom number generator** (**PRNG**) is fine. In .NET, you can use `new Random()` for a PRNG, rather than the more secure option of `RandomNumberGenerator.Create()`. See Chapter 6, *Understanding Code Execution and Asynchronous Operations* , for more examples of how to use both of these.

When looking at your results, the outliers may be more interesting than the average. Although the median is more valuable than the mean, in this case, you should really look at the percentiles, for example, the 90th, 95th, and 99th percentiles. These data points can represent only a small fraction of your data, but at scale, they can occur frequently. You want to optimize for these worst case scenarios because if your users experience pages loads taking over five seconds ten percent of the time (even though the average looks fast), then they may go elsewhere.

There's much more to say on statistics, but beyond the basics, there are diminishing returns. If your math is rusty, then it is probably wise to have a refresher (Wikipedia is great for this). Then, you can explore some more advanced techniques, for example, the high-performance HyperLogLog (**HLL**) algorithm, which can estimate the size of a large set of elements using very little memory. Redis supports the HLL

data structure (with the `PFADD`, `PFCOUNT`, and `PFMERGE` commands). For more on different data structures, refer to [Chapter 6](#), *Understanding Code Execution and Asynchronous* Operations.

Note

This is only a brief introduction to performance logging, but there is much more for you to explore. For example, if you want to standardize your approach, then you can look into APDEX (apdex.org), which sets a standard method to record the performance of applications and compute scores.

Managing stale caches

It's worth providing a quick reminder to still consider simple issues after all of this complexity. It is far too easy to get lost in the details of a complicated bug or performance tweak and miss the obvious.

Tip

A good technique to help with this is *rubber duck debugging*, which gets its name from the process of explaining your problem to a rubber duck on your desk. Most of us have experienced solving a problem after asking for help, even though the other person hasn't said anything. The process of explaining the problem to someone (or something) else clarifies it, and the solution becomes obvious.

If something appears to not be working after a fix, then check simple things first. See whether the patch has actually been delivered and deployed. You may be seeing stale code from a cache instead of your new version.

When managing caches, versioning is a useful tool to help you identify stale assets. You can alter filenames or add comments to include a unique version string. This can be **Sematic Versioning** (**SemVer**), an ISO date and timestamp, or a hash of the contents. For more on cache busting, refer to [Chapter 7](#), *Learning Caching and Message Queuing* .

Note

SemVer, is a great way to version your code because it implicitly captures information on compatibility and breaking changes. You can read more about SemVer at semver.org .

Summary

In this chapter, we saw how there are always downsides to every decision and every choice has a cost attached because nothing comes for free. There are always tradeoffs involved, and you need to be aware of the consequences of your actions, which may be small and subtle.

The key lesson is to take a thoughtful and rigorous approach to adding any performance-enhancing technique. Measurement is crucial to achieving this, but you also need to know how data can mislead you if you collect or interpret it incorrectly.

In the next chapter, we will continue with the measurement theme and learn how to use tests to monitor for performance regressions. You will see how to use testing (including unit testing), automation, and continuous integration to ensure that once you solve a performance problem, it stays this way.

Chapter 9. Monitoring Performance Regressions

This chapter will cover writing automated tests to monitor performance along with adding these to a **Continuous Integration** (**CI**) and deployment system. By constantly checking for regressions, you'll avoid accidentally building a slow application. We'll also cover how to safely load test a system without forcing it offline and how to ensure that tests mimic real life usage as far as possible.

Topics covered in this chapter include the following:

- Profiling
- Load testing
- Automated testing
- Performance monitoring
- Continuous integration and deployment
- Realistic environments and production-like data
- UI testing with selenium and phantom headless browsers
- A/B testing for conversion optimization
- Cloud services and hosting
- DevOps

You will see how to automate performance monitoring and testing so that you don't need to remember to keep doing it manually. You'll learn how to catch regressions early before they cause trouble and how to safely back them out for rework.

Profiling and measurement

We started this book by highlighting the importance of measurement and profiling by covering some simple techniques in [Chapter 2](#), *Measuring Performance Bottlenecks*. We continued this theme throughout, and we'll end the book on it as well because it's impossible to overstate how important measuring and analyzing reliable evidence is.

Previously, we covered using Glimpse to provide insights into the running of your web application. We also demonstrated the Visual Studio diagnostics tools and the Application Insights **Software Development Kit** (**SDK**). There's another tool that's worth mentioning and that's the Prefix profiler, which you can get at prefix.io .

Prefix is a free web-based ASP.NET profiler that supports ASP.NET Core. However, it doesn't support .NET Core yet (although this is planned), so you'll need to run ASP.NET Core on .NET Framework 4.6 for now. There's a live demo on their website (at demo.prefix.io) if you want to quickly check it out and it looks like the following:
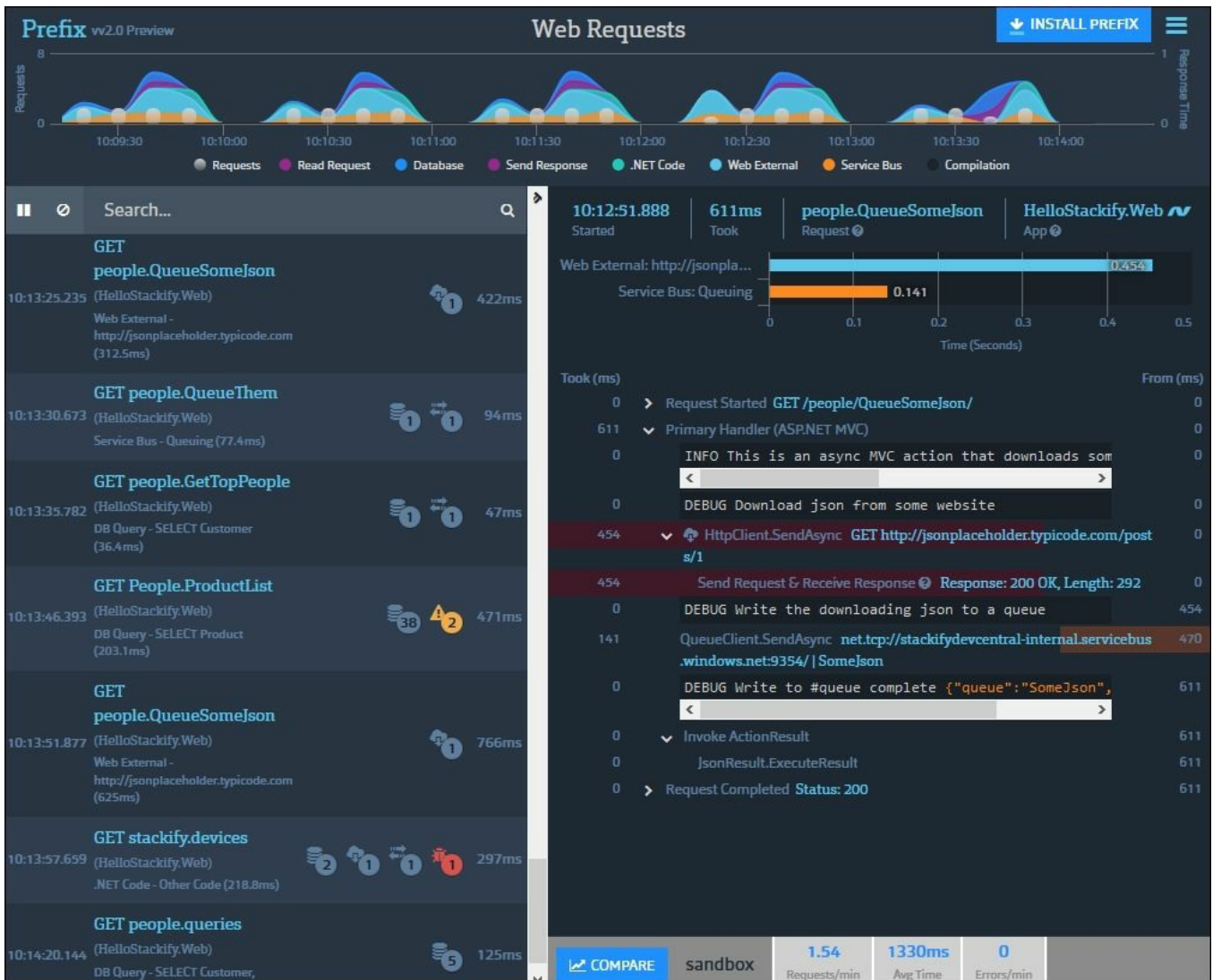
Note

ASP.NET Core can run on top of either .NET Core or the existing stable framework. For a refresher on how everything fits together, refer back to Chapter 1, *Why Performance Is a Feature*, or for more details, refer to Chapter 6, *Understanding Code Execution and Asynchronous Operations* .

You may also want to look at the **PerfView** performance analysis tool from Microsoft, which is used in the development of .NET Core. You can download PerfView from microsoft.com/download/details.aspx?id=28567 as a ZIP file that you can just extract and run. It is useful to analyze the memory of .NET applications among other things, and it looks like this when you launch it:
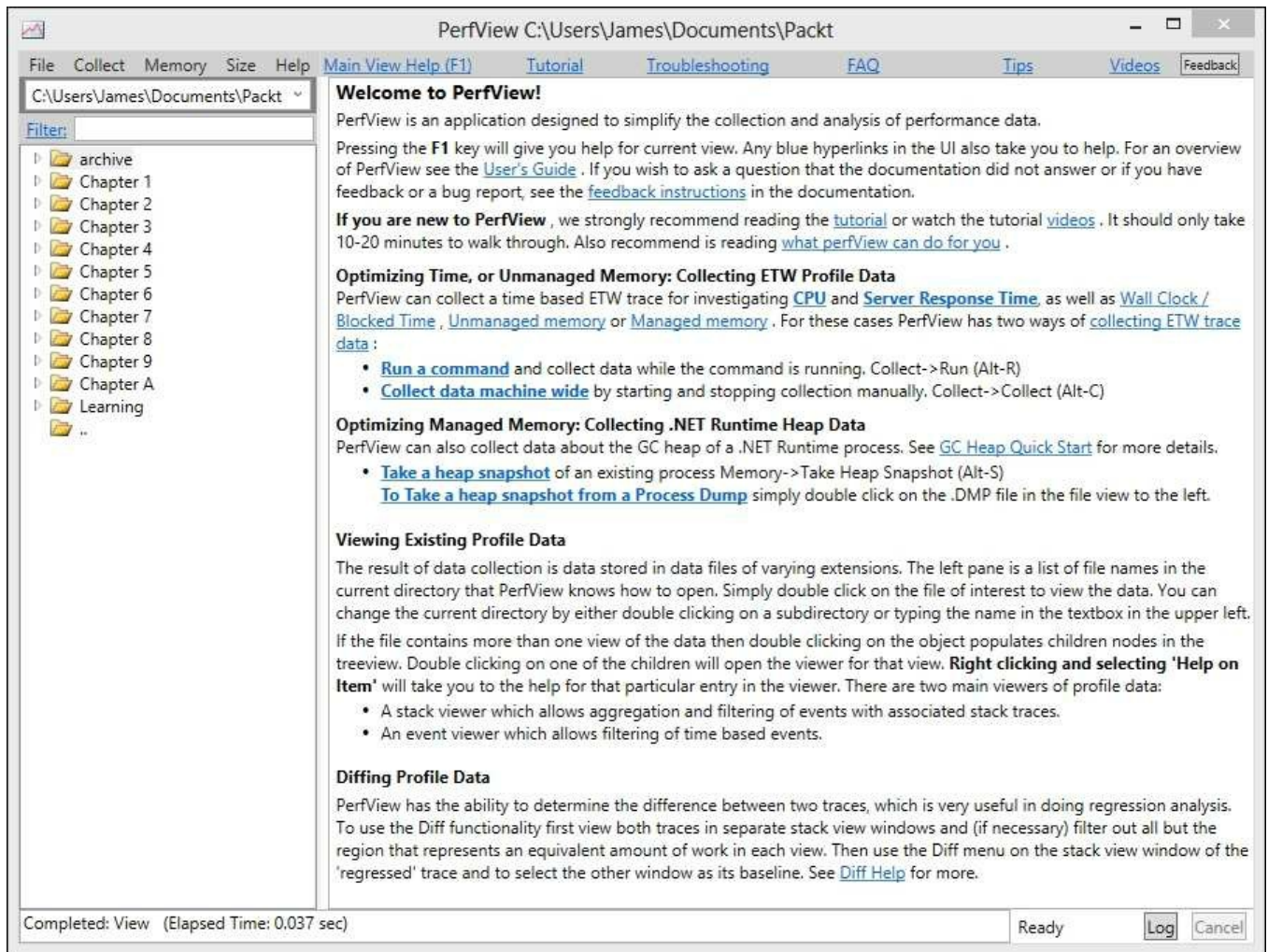
You can use PerfView for many debugging activities, for example, to take a snapshot of the heap or force GC runs. We don't have space for a detailed walkthrough here, but the included instructions are good, and there are blogs on MSDN with guides and many video tutorials on *Channel 9* at channel9.msdn.com/Series/PerfView-Tutorial if you need more information.

Tip

Sysinternals tools (technet.microsoft.com/sysinternals) can also be helpful, but as they do not focus much on .NET, they are less useful in this context.

While tools such as these are great, what would be even better is to build performance monitoring into your development workflow. Automate everything that you can, and this makes performance checks transparent, routine, and run by default.

Manual processes are bad because you can skip steps, and you can easily make errors. You wouldn't dream of developing software by e-mailing files around or editing code directly on a production server, so why not automate your performance

tests too.

Change control processes exist to ensure consistency and reduce errors. This is why using a **Source Control Management (SCM)** system, such as Git or **Team Foundation Server (TFS)** is essential. It's also extremely useful to have a build server and perform CI or even fully-automated deployments.

Note

Source control allows multiple people to work on a file simultaneously and merge the changes later. It's like Word's track changes feature, but actually usable. We assume that we're preaching to the converted and you already use source control. If not, stop reading right now and go install an SCM system.

If the code deployed in production differs from what you have on your local workstation, then you have very little chance of success. This is one of the reasons why SQL **Stored Procedures (SP**s/**sproc**s) are difficult to work with, at least without rigorous version control. It's far too easy to modify an old version of an SP on a development DB, accidentally revert a bug fix, and end up with a regression. If you must use SPs, then you will need a versioning system, such as ReadyRoll (which Redgate has now acquired).

As this isn't a book on **Continuous Delivery (CD)**, we will assume that you are already practicing CI and have a build server, such as JetBrains TeamCity, ThoughtWorks GoCD, CruiseControl.NET, or a cloud service, such as AppVeyor. Perhaps you're even automating your deployments using a tool, such as Octopus Deploy**,** and you have your own internal NuGet feeds, using software such as The Motley Fool's Klondike, or a cloud service, such as MyGet (which also supports npm, bower, and VSIX packages).

Tip

NuGet packages are a great way of managing internal projects. In new versions of Visual Studio, you can see the source code of packages and debug into them. This means no more huge solutions containing a ludicrous number of projects.

Bypassing processes and doing things manually will cause problems, even if you are following a script. If it can be automated then it probably should be, and this includes testing.

Testing

Testing is essential to producing high-quality and well-performing software. The secret to productive testing is to make it easy, reliable, and routine. If testing is difficult or tests regularly fail because of issues unrelated to the software (for example, environmental problems), then tests will not be performed or the results will be ignored. This will cause you to miss genuine problems and ship bugs that you could have easily avoided.

There are many different varieties of testing, and you may be familiar with the more common cases used for functional verification. In this book, we will mainly focus on tests pertaining to performance. However, the advice here is applicable to many types of testing.

Automated testing

As previously mentioned, the key to improving almost everything is automation. Tests that are only run manually on developer workstations add very little value. Of course, it should be possible to run the tests on desktops, but this shouldn't be the official result because there's no guarantee that they will pass on a server (where the correct functioning matters more).

Tip

Although automation usually occurs on servers, it can be useful to automate tests that run on developer workstations too. One way of doing this in Visual Studio is to use a plugin, such as **NCrunch**. This runs your tests as you work, which can be very useful if you practice **Test-Driven Development** (**TDD**) and write your tests before your implementations. You can read more about NCrunch and see the pricing at ncrunch.net , or there's a similar open source project at continuoustests.com .

One way of enforcing testing is to use gated check-ins in TFS, but this can be a little draconian, and if you use an SCM, such as Git, then it's easier to work on branches and simply block merges until all the tests pass. You want to encourage developers to check-in early and often because this makes merges easier. Therefore, it's a bad idea to have features in progress sitting on workstations for a long time (generally no longer than a day).

Continuous integration

CI systems automatically build and test all of your branches and feed this information back to your version control system. For example, using the GitHub API, you can block the merging of pull requests until the build server reports a successfully tested merge result.

Both **Bitbucket** and **GitLab** offer free CI systems, called pipelines, so you may not need any extra systems in addition to one for source control because everything is in one place. GitLab also offers an integrated Docker container registry, and there is an open source version that you can install locally. .NET Core and Visual Studio support Docker well, and we'll cover this again later in the chapter.

You can do something similar with **Visual Studio Team Services** for CI builds and unit testing. Visual Studio also has Git services built into it.

This process works well for unit testing because unit tests must be quick so that you get feedback early. Shortening the iteration cycle is a good way of increasing productivity, and you'll want the lag to be as small as possible.

However, running tests on each build isn't suitable for all types of testing because not all tests can be quick. In this case, you'll need an additional strategy so as not to slow down your feedback loop.

Note

There are many unit testing frameworks available for .NET, for example, NUnit, xUnit, and MSTest (Microsoft's unit test framework) along with multiple graphical ways of running tests locally, such as the Visual Studio Test Explorer and the ReSharper plugin. People have their favorites, but it doesn't really matter what you choose because most CI systems will support all of them.

Slow testing

Some tests are slow, but even if each test is fast, they can easily add up to a lengthy time if you have a lot of them. This is especially true if they can't be parallelized and need to be run in sequence, so you should always aim to have each test stand on its own without any dependencies on others.

It's good practice to divide your tests into rings of importance so that you can at least run a subset of the most crucial on every CI build. However, if you have a large test suite or some tests that are unavoidably slow, then you may choose to only run these once a day (perhaps overnight) or every week (maybe over the weekend).

Some testing is simply slow by nature and performance testing can often fall into this category, for example, load testing or **User Interface** (**UI**) testing. We usually class this as integration testing, rather than unit testing because they require your code to be deployed to an environment for testing and the tests can't simply exercise the binaries.

To make use of such automated testing, you will need to have an automated deployment system in addition to your CI system. If you have enough confidence in your test system, then you can even have live deployments happen automatically. This works well if you also use **feature switching** to control the rollout of new features.

We won't go into the implementation details of Continuous Integration or automated deployments in this book. However, we will cover feature switching, how to apply performance testing to CI processes, and what to do when you discover a regression.

Fixing performance regressions

If you discover a performance issue at the unit testing stage, then you can simply rework this feature, but it's more likely that these problems will surface in a later testing phase. This can make it more challenging to remedy the problem because the work may already have been built upon and have other commits on top of it.

The correct course of action is often to back-out regressions immediately or at least as soon as possible upon discovery. Delays will only make the issue harder to fix later, which is why it's important to get fast feedback and highlight problems quickly.

It's important to be disciplined and always remove regressions, even though it may be painful. If you let the occasional minor regression in, then you can easily become sloppy and let more serious ones in over time because of the precedent it sets.

Load testing

Load testing is the process of discovering how many concurrent users your web app can support. You generally perform it on a test environment with a tool that gradually ramps up a simulated load, for example, JMeter (jmeter.apache.org). Perhaps, you'd prefer using a JMeter compatible cloud service, such as BlazeMeter, or an alternative, such as **Loader.io**, if your test systems are web-facing.

Load testing can take a significant amount of time depending on the scale of your service because it can be configured to continue until the test environment gets unacceptably slow for users or falls over and becomes unresponsive. You need to be extremely careful with load testing and not only from the point of view of accidentally testing your live systems to destruction while they're in use.

You also need to be wary of getting false results, which may mislead you into concluding that your system can handle more load than it actually will. Balancing these two competing concerns of safety and realism can be difficult. It's important to get realistic results, but you need to balance this against not stressing your production environment and impacting the experience of real users.

Realism

Keeping it real is an important principle of performance testing. If you don't use a realistic environment and workload, then your results may be worse than having no data because they could mislead you into bad decisions. When you have no information, you at least know that you're in the dark and just guessing.

We'll cover workloads and feature switching shortly, including an example of how to implement your own simple version from scratch. First, let's look at how to make your test environments representative of production.

Realistic environments

Using a test environment that is as close to production (or as live-like) as possible is a good step toward ensuring reliable results. You can try and use a smaller set of servers and then scale your results up to get an estimate of live performance. However, this assumes that you have an intimate knowledge of how your application scales and what hardware constraints will be the bottlenecks.

A better option is to use your live environment or rather what will become your production stack. You first create a staging environment that is identical to live, then you deploy your code to it, and then you run your full test suite, including a comprehensive performance test, ensuring that it behaves correctly. Once you are happy, then you simply swap staging and production, perhaps using DNS or Azure **staging slots**.

The following diagram shows how you first release to staging and go live simply by making staging become production:

Release

Go Live
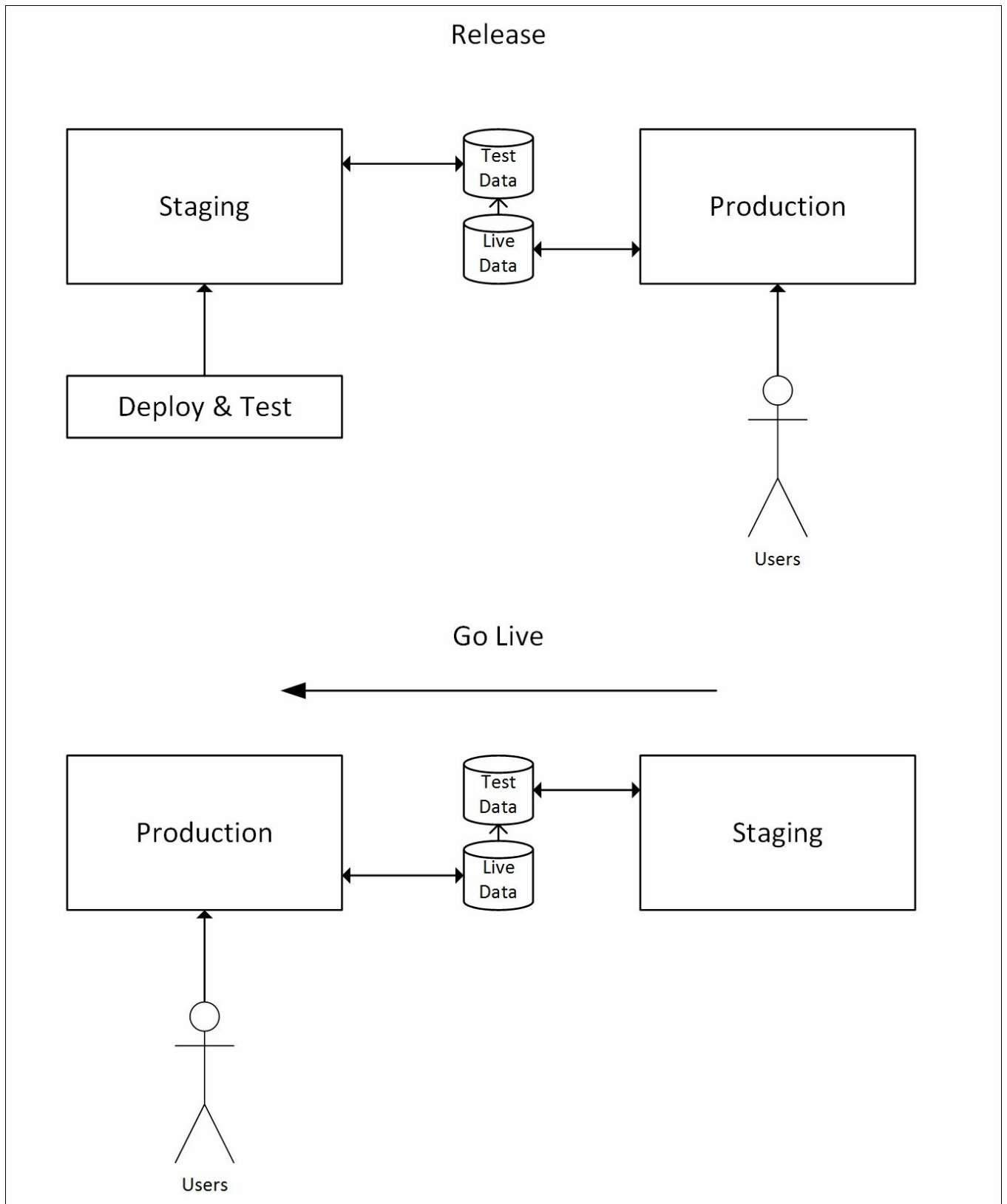
Your old live environment now either becomes your test environment or if you use immutable cloud instances, then you can simply terminate it and Spin-up a new staging system. This concept is known as *blue-green deployment*, but unfortunately specific implementation instructions are beyond the scope of this book.

You don't necessarily have to move all users across at once in a big bang, and you can move a few over first to test whether everything is correct. We'll cover this shortly in the section on feature switching.

Realistic workloads

Another important part of performing realistic testing is to use real data and real workloads. Synthetic data often won't exercise a system fully or find exotic edge cases. You can use **fuzzing** to generate random data, but if you have a production system, then you can simply use the data from this and parse your logs to generate a realistic workload to replay.

Obviously, don't replay actions onto a live system that could modify user data and be wary of data privacy concerns or any action that could generate an external event, for example, sending a mass e-mail or charging a credit card. You can use a dedicated test system, but you still need to be careful to stub out any external APIs that don't have a test version.

Another approach is to use dummy users for your testing if you don't mind your users discovering the fakes and possibly interacting with them. One case of this approach is Netflix's cult *Example Show*, which is a homemade video of an employee running around their office grounds with a laptop.

Feature switching

An alternative to fake users or test environments is to run performance tests on your real live environment using genuine users as they interact with your web application. You can use feature switching to gradually roll out a new feature to a small subset of your user base and monitor the system for excessive load. If everything seems normal, then you can continue the rollout; otherwise, you can rollback.

Note

You may hear similar ideas referred to by other names, such as **Feature Flags**, **Feature Toggles**, **Canary Releases**, or **Branch by Abstraction**. Some of these may have subtly different meanings but the general guiding principle of gradually releasing a change is much the same.

The following diagram shows how you could progressively migrate users to a new feature by deploying it but not initially enabling it. Your first users could be members of staff so that any issues are detected early:
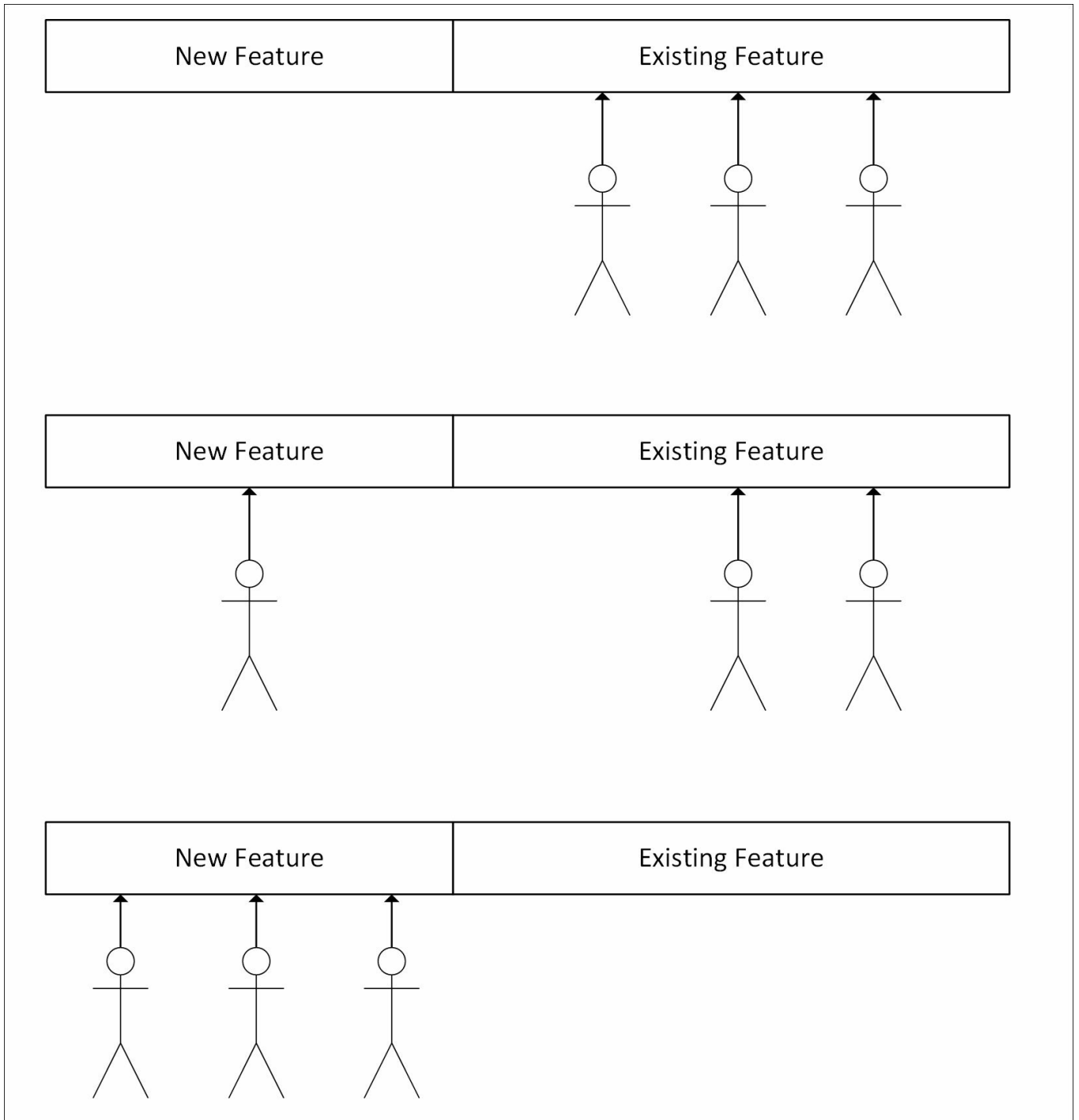
Once all users are consuming the new feature, then you can safely remove the old feature (and the feature switch itself). It's a good idea to regularly tidy up like this to avoid clutter and make later work easier.

If you discover that your system experiences excessive load as you slowly increase the percentage of users on the new feature, then you can halt the rollout and avoid your servers becoming overloaded and unresponsive. You then have time to investigate and either back out of the change or increase available resources.

One variation on this theme is when a new feature requires a data migration and this migration is computationally or networking-intensive, for example, migrating user-submitted videos to a new home and transcoding them in the process. In this case, the excessive load that you are monitoring will only be transient, and you don't need to back out of a feature. You only need to ensure that the rate of migration is low enough to not excessively tax your infrastructure.

Although the new feature is usually branched to in code, you can instead perform switching at the network level if you use blue-green deployment. This is known as a canary release and can be done at the DNS or load balancer level. However, specific implementation details are again beyond the remit of this book.

You can find many open source feature switching libraries online or you could write your own, which we will show you how to do later. A couple of .NET feature-switching libraries are github.com/mexx/FeatureSwitcher and github.com/Jason-roberts/FeatureToggle . Unfortunately, neither of these work on .NET Core yet, although support is planned for **FeatureToggle**. There are also paid cloud services available, which offer easy management interfaces, such as **LaunchDarkly**.

Note

Library and framework support for .NET Core and ASP.NET Core change rapidly, so check ANCLAFS.com for the latest information.

To illustrate feature switching, let's build our own extremely simple implementation for an ASP.NET Core web application. To start with, we take the default existing homepage view (`Index.cshtml`) and make a copy (`IndexOld.cshtml`). We then make our changes to `Index.cshtml`, but these aren't important for the purposes of this demo.

In `HomeController.cs`, we change the logic to return the new view a relatively small percentage of the time, and the old one otherwise. The original code was simply the following:

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

We change this action to return the new view in a quarter of the cases, like the following:

```
public IActionResult Index()
{
    var rand = new Random();
```

```
    if (rand.Next(99) < 25)
    {
        return View();
    }
    return View("IndexOld");
}
```

This code picks a random number out of a hundred, and if it is less than 25, then it loads the new view. Clearly, you wouldn't hardcode values like this, and you would probably use a database to store configuration and control it with a web admin page.

If you load the page in a browser, then three out of four times you should get the original page, which looks like the following:
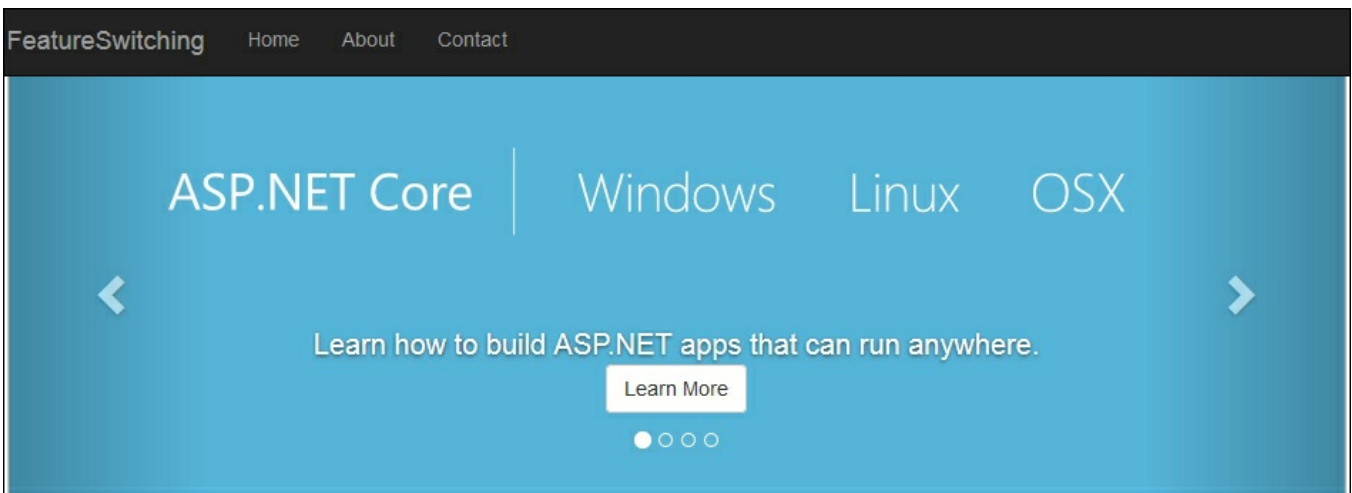
However, roughly one every four page loads, you will get the new page, which looks like the following:
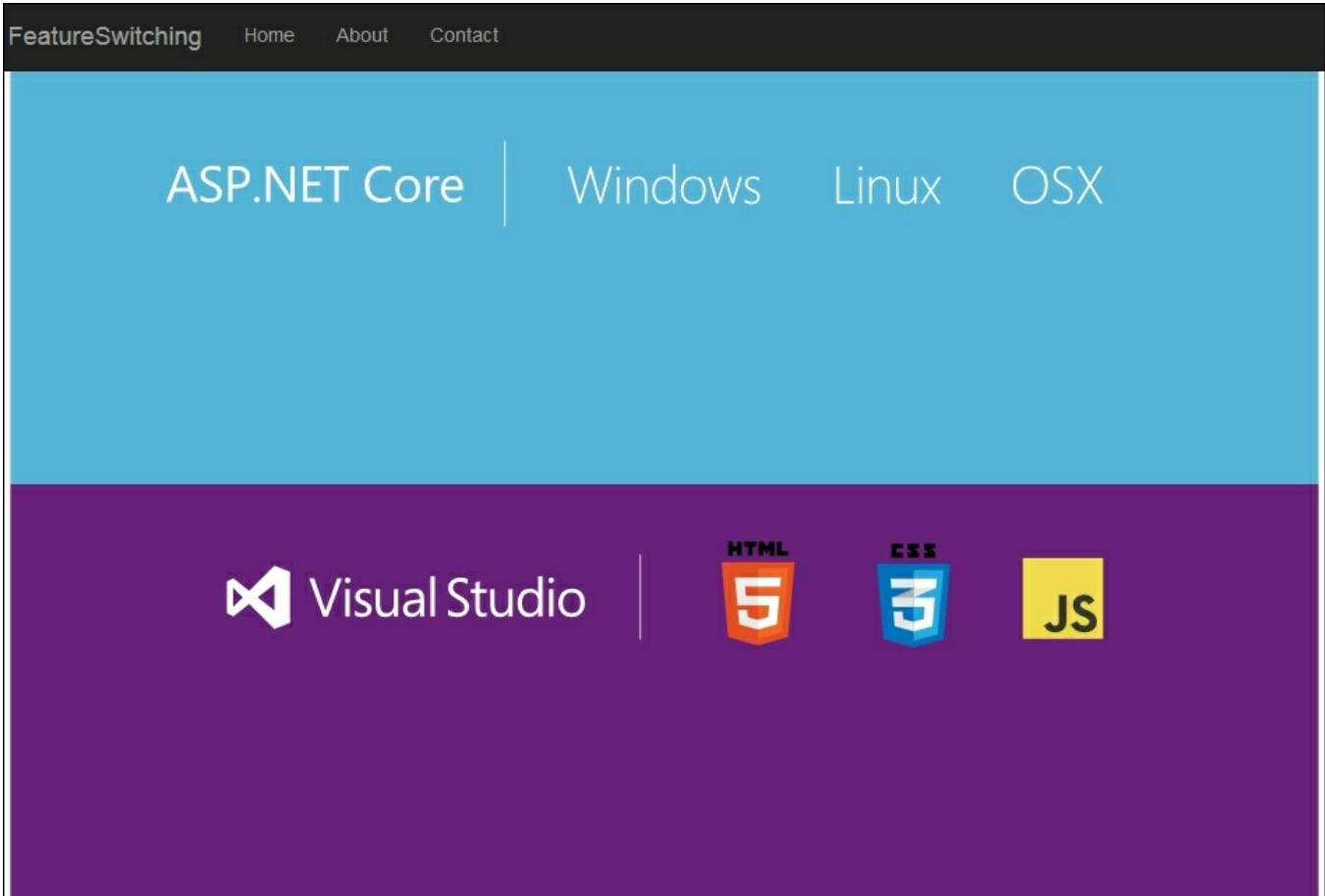
We removed the carousel and cut the number of image loads in half. You can see the difference in the browser developer tools, for example, the Firefox performance analysis for the original looks like the following:

Primed cache

■	4 images	40.27 kB	0.23 s
■	1 html	8.26 kB	0.05 s
■	2 css	0.25 kB	0.02 s
■	3 js	0 kB	0.01 s

Size: 48.77 kB
Time: 0.31 seconds
Cached responses: 3
Total requests: 10

Empty cache

■	3 js	119.59 kB	0.35 s
■	2 css	118.66 kB	0.12 s
■	4 images	40.27 kB	0.23 s
■	1 html	8.26 kB	0.05 s

Size: 286.78 kB
Time: 0.74 seconds
Cached responses: 0
Total requests: 10

Whereas for the new version it looks like this.
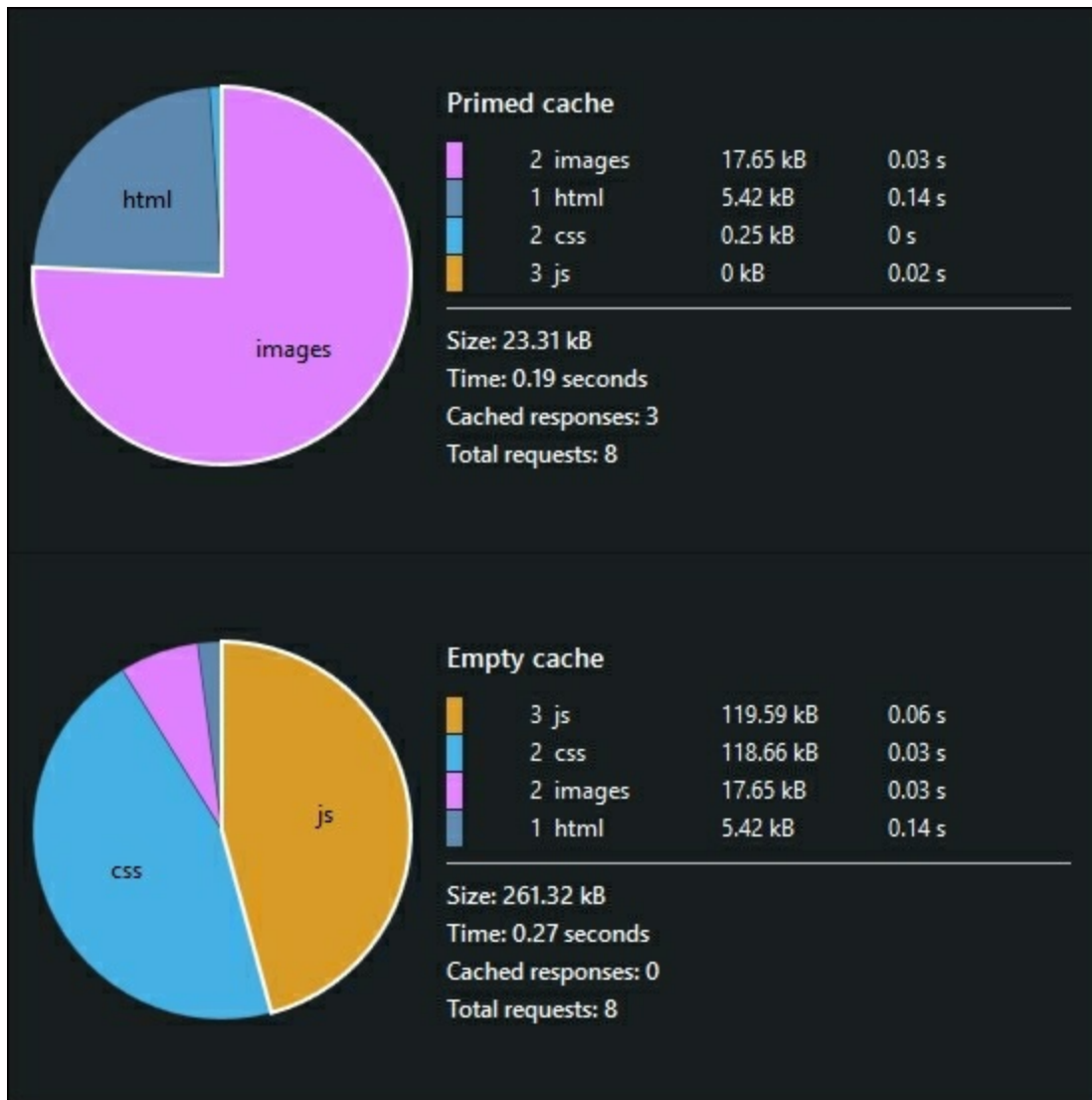
You can see that the number of **images** decreased, which has reduced the page size and the total number of requests required.

Note

These measurements were taken with the hosting environment set to `Production`. If you remain with the default of `Development`, then your results will differ. Refer to the following documentation page for how to change this: docs.asp.net/en/latest/fundamentals/environments.html .

Once the new view has been progressively rolled out to all users (with no performance problems), then the action method code can be reverted to the original state and the old view deleted. Obviously, this is a trivial example. Hopefully, you can now see how feature switching can be performed.

Using feature switching to rollout in this way works well for performance testing, but

you need to be confident that your new code is functionally correct or at least safe to deploy. However, in certain circumstances you can also test functionality by extending this approach and performing experiments.

Experimenting for science

If you take the feature switching rollout to its logical conclusion, then you can switch on a new refactored version for a small percentage of users, but not actually expose the output. You will run the existing version in parallel as a control and show this to the user. However, you will collect the results of both versions, including performance data. You can then compare them to ensure that the new (hopefully higher performance) version is correct and consistent with the existing behavior.

GitHub have an open source Ruby library, called `Scientist`, which they used to successfully refactor their permissions and merging code. In the process, they discovered existing legacy bugs, found missing features, optimized database queries, and swapped a search system. Scientist not only displays the correctness of new code but also its performance relative to the old implementation.

Tip

An important concern with `Scientist` (or any method using this experimentation approach) is to not change data. All operations should be read-only and have no side effects; otherwise, you will perform the modification more than once, which could have undesirable consequences.

There's a .NET port of `Scientist` by GitHub employee Phil Haack, called *Scientist.NET*, and you can get it at github.com/Haacked/Scientist.net . You could also install it via NuGet. However, it's currently a prerelease as it's still a work in progress, but it should support .NET Core. Check ANCLAFS.com for the latest information.

This idea of experimenting on users is similar to the marketing concept of **A/B testing**, which is used to discover conversion effectiveness. However, with `Scientist`, you don't typically show the users different outputs or even intend the output to change, you just record the new output for the subset of users that you have it enabled for.

A/B testing

A/B testing is similar to feature switching, but we usually use it to test the effectiveness of different web page designs and how they affect the *conversion funnel* analytics. We normally use it for digital marketing rather than for software development because it's less about what is correct and more about what customers prefer. However, the underlying technical principals are comparable to feature switching and experimenting.

In a simple example, you serve half of your visitors the old version of a page and the other half a new version designed to increase engagement. You then record how many visitors click through or perform some action on each variant. If the new version performs better, then you keep it and roll it out to everyone. However, if it converted worse, you roll back to the old version and try again.

You can see that this is very similar to how feature switching works, and you can use the same tools to do both. The only difference is what you are measuring, user analytics or the health of your infrastructure.

A/B testing is not normally used for backend features and is simply to assess UI alternatives. However, it is different to functionally testing your web interface, so let's cover the basics of UI testing now.

User interface testing

There's one area of an application that is traditionally difficult to test, and that's the UI. This is particularly the case for GUIs, as used in web applications. One reason for this is that users typically have a lot of flexibility in how they display their interface, and this is especially true for web browsers. A naïve pixel-based testing approach is going to be extremely fragile.

You need to design your application to be easily testable, and the UI is no exception. While it is possible to test the UI of a poorly-designed legacy application and this may even be the easiest testing approach, if you consider UI testing up front, then life will be a lot easier. For example, including sensible IDs on your HTML **Document Object Model** (**DOM**) elements makes testing much more straightforward and less fragile.

Checking your application from a web browser perspective can be a useful additional step on top of automated unit and integration testing. You can use it not only to test for functional correctness and regressions but also to measure client side performance, which is increasingly important. There are many UI testing tools available, most of which you can integrate into your CI pipeline to automate testing.

Web UI testing tools

One of the most popular web-testing tools is **Selenium**, which allows you to easily write tests and automate web browsers using **WebDriver**. Selenium is useful for many other tasks apart from testing, and you can read more about it at docs.seleniumhq.org .

Note

WebDriver is a protocol to remote control web browsers, and you can read about it at w3c.github.io/webdriver/webdriver-spec.html .

Selenium uses real browsers, the same versions your users will access your web application with. This makes it excellent to get representative results, but it can cause issues if it runs from the command line in an unattended fashion. For example, you may find your test server's memory full of dead browser processes which timed out.

You may find it easier to use a dedicated headless test browser, which, while not exactly the same as what your users will see, is more suitable for automation. Of course, the best approach is to use a combination of both, perhaps running headless tests first and then running the same tests on real browsers with WebDriver.

One of the most well-known headless test browsers is **PhantomJS**. This is based on the **WebKit** engine, so it should give you similar results to Chrome and Safari. PhantomJS is useful for many things apart from testing, such as capturing screenshots, and you can drive it with many different testing frameworks. As the name suggests, you can control PhantomJS with JavaScript, and you can read more about it at phantomjs.org .

Note

WebKit is an open source engine for web browsers, which was originally part of the **KDE** Linux desktop environment. It is mainly used in Apple's Safari browser, but a fork called **Blink** is used in Google Chrome, Chromium, and Opera. You can read more at webkit.org .

Other automatable testing browsers, based on different engines, are available, but they have some limitations. For example, **SlimerJS** (slimerjs.org) is based on the **Gecko** engine used by Firefox, but is not fully headless.

You probably want to use a higher-level testing utility rather than scripting browser engines directly. One such utility that provides many useful abstractions is **CasperJS** (casperjs.org), which supports running on both PhantomJS and SlimerJS.

Another tool is **Capybara**, which allows you to easily simulate user interactions in Ruby. It supports Selenium, WebKit, **Rack**, and PhantomJS (via Poltergeist), although it's more suitable for Rails apps. You can read more at jnicklas.github.io/capybara.

There is also **TrifleJS** (triflejs.org), which uses the .NET `WebBrowser` class (the Internet Explorer **Trident** engine), but this is a work in progress. Additionally, there's **Watir** (watir.com), which is a set of Ruby libraries targeting Internet Explorer and WebDriver. However, neither have been updated in a while, and IE has changed a lot recently.

Note

Microsoft Edge (codenamed Spartan) is the new version of IE, and the Trident engine was forked to **EdgeHTML**. The JavaScript engine (Chakra) was open sourced as **ChakraCore** (github.com/Microsoft/ChakraCore).

It shouldn't matter too much what browser engine you use, and PhantomJS will work fine as a first pass for automated tests. You can always test with real browsers after using a headless one, perhaps with Selenium or with PhantomJS using WebDriver.

Tip

When we refer to browser engines (WebKit/Blink, Gecko, or Trident/EdgeHTML), we generally mean only the rendering and layout engine, not the JavaScript engine (SFX/Nitro/FTL/B3, V8, SpiderMonkey, or Chakra/ChakraCore).

You'll probably still want to use a utility such as CasperJS to make writing tests easier, and you'll likely need a test framework, such as **Jasmine** (jasmine.github.io) or **QUnit** (qunitjs.com) too. You can also use a test runner that supports both Jasmine and QUnit, such as **Chutzpah** (mmanela.github.io/chutzpah).

You can integrate your automated tests with many different CI systems, for example, Jenkins or JetBrains TeamCity. If you prefer a cloud-hosted option, then there's Travis CI (travis-ci.org) and AppVeyor (appveyor.com), which is also suitable to build .NET apps.

You may prefer to run your integration and UI tests from your deployment system, for example, to verify a successful deployment in Octopus Deploy. There are also dedicated, cloud-based web-application UI testing services available, such as BrowserStack (browserstack.com).

Automating UI performance tests

Automated UI tests are clearly great to check for functional regressions, but they are also useful to test performance. You have programmatic access to the same information provided by the network inspector in the browser developer tools.

You can integrate the **YSlow** (yslow.org) performance analyzer with PhantomJS, enabling your CI system to check for common web-performance mistakes on every commit. YSlow came out of Yahoo!, and it provides rules used to identify bad practices, which can slow down web applications for users. It's a similar idea to Google's PageSpeed Insights service (which you can automate via its API).

However, YSlow is pretty old, and things have moved on in web development recently, for example, HTTP/2. A modern alternative is "the coach" from **sitespeed.io** , and you can read more at github.com/sitespeedio/coach . You should check out their other open source tools too, such as the dashboard at dashboard.sitespeed.io , which uses Graphite and Grafana.

You can also export the network results (in industry standard HAR format) and analyze them however you like, for example, visualizing them graphically in waterfall format, as you might do manually with your browser developer tools.

Note

The **HTTP Archive (HAR)** format is a standard way of representing the content of monitored network data to export it to other software. You can copy or save as HAR in some browser developer tools by right-clicking a network request.

Staying alert

Whenever you perform testing, particularly UI or performance testing, you will get noisy results. Reliability is not perfect, and there will always be failures that are not due to bugs in the code. You shouldn't let these false positives cause you to ignore failing tests, and although the easiest course of action may be disabling them, the correct thing to do to make them more reliable.

Tip

The scientifically minded know that there is no such thing as a perfect filter in binary classification, and always look at the precision and recall of a system. Knowing the rate of false positives and negatives is important to get a good idea of the accuracy and tradeoffs involved.

To avoid testing fatigue, it can be helpful to engage developers and instill a responsibility to fix failing tests. You don't want everyone thinking that it's somebody else's problem. It should be pretty easy to see who broke a test by the commit in version control, and it's then their job to fix the broken test.

You can have a bit of fun with this and create something a bit more interesting than a wall mounted dashboard. Although having **information radiators** is useful if you don't get desensitized to them. There's plenty of cheap **Internet of Things (IoT)** hardware available today, which allows you to turn some interesting things into build or test failure alarms. For example, an **Arduino**-controlled traffic light, an **ESP8266**-operated foam-missile turret, or a Raspberry Pi-powered animatronic figure.

DevOps

When using automation and techniques such as feature switching, it is essential to have a good view of your environments so that you know the utilization of all the hardware. Good tooling is important to perform this monitoring, and you want to easily be able to see the vital statistics of every server. This will consist of at least the CPU, memory, and disk space consumption, but it may include more, and you will want alarms set up to alert you if any of these stray outside allowed bands.

The practice of **DevOps** is the culmination of all of the automation we covered previously with development, operations, and quality-assurance testing teams all collaborating. The only missing pieces left now are to provision and configure infrastructure and then monitor it while in use. Although DevOps is a culture, there is plenty of tooling that can help.

DevOps tooling

One of the primary themes of DevOps tooling is defining infrastructure as code. The idea is that you shouldn't manually perform a task, such as setting up a server, when you can create software to do it for you. You can then reuse these provisioning scripts, which will not only save you time but will also ensure that all of the machines are consistent and free of mistakes or missed steps.

Provisioning

There are many systems available to commission and configure new machines. Some popular configuration-management automation tools are **Ansible** (ansible.com), **Chef** (chef.io), and **Puppet** (puppet.com).

Not all of these tools work great on Windows servers, partly because Linux is easier to automate. However, you can run ASP.NET Core on Linux and still develop on Windows, using Visual Studio while testing in a VM. Developing for a VM is a great idea because it solves the problems in setting up environments and issues where it "works on my machine" but not in production.

Vagrant (vagrantup.com) is a great command line tool to manage developer VMs. It allows you to easily create, Spin-up, and share developer environments. The successor to Vagrant, **Otto** (ottoproject.io) takes this a step further and abstracts deployment too so that you can push to multiple cloud providers without worrying about the intricacies of **CloudFormation**, **OpsWorks**, or anything else.

If you create your infrastructure as code, then your scripts can be versioned and tested, just like your application code. We'll stop before we get too far off topic, but the point is that if you have reliable environments, which you can easily verify, instantiate, and perform testing on, then CI is a lot easier.

Monitoring

Monitoring is essential, especially for web applications, and there are many tools available to help with it. A popular open source infrastructure-monitoring system is **Nagios** (nagios.org). Another more modern open source alerting and metrics tool is **Prometheus** (prometheus.io).

If you use a cloud platform, then there will be monitoring built in, for example, AWS CloudWatch or Azure Diagnostics. There are also cloud services to directly monitor your website, such as **Pingdom** (pingdom.com), **UptimeRobot** (uptimerobot.com), **Datadog** (datadoghq.com), and **PagerDuty** (pagerduty.com).

You probably already have a system in place to measure availability, but you can also use the same systems to monitor performance. This is not only helpful to ensure a responsive user experience, but it can also provide early warning signs that a failure is imminent. If you are proactive and take preventative action, then you can save yourself a lot of trouble reactively fighting fires.

This isn't a book on operations, but it helps to consider application support requirements at design time. Development, testing, and operations aren't competing disciplines, and you will succeed more often if you work as one team rather than simply throwing an application over the fence and saying it "worked in test, ops problem now."

Hosting

It's well worth considering the implications of various hosting options because if developers can't easily access the environments they need, then this will reduce their agility. They may have to work around an availability problem and end up using insufficient or unsuitable hardware, which will hamper progress and cause future maintenance problems. Or their work will simply be blocked and delivery set back.

Unless you are a very large organization, hosting in-house is generally a bad idea for reliability, flexibility, and cost reasons. Unless you have some very sensitive data, then you should probably use a data center.

You can co-locate servers in a data center, but then you need staff to be on call to fix hardware problems. Or you can rent a physical server and run your application on "bare metal", but you may still need remote hands to perform resets or other tasks on the machine.

The most flexible situation is to rent self-service virtual machines, commonly known as cloud hosting. There are many hosting companies that offer this, but the big players are Amazon, Google, and Microsoft.

Microsoft's Azure is the obvious choice for a .NET shop and it has improved immensely since launch compared to its original offering. Its **Platform as a Service (PaaS)** to host .NET applications is the most polished and the easiest to get running on quickly. It also offers lots of extra services that go beyond simple VMs.

However, .NET Core and ASP.NET Core are new types of framework, which are not aimed solely at C# developers, who would usually pick the default option offered by Microsoft. The targeted market is developers who may be used to choosing alternative platforms and open source frameworks. Therefore, it makes sense to cover other options, which people new to .NET may be more familiar with.

Note

The difference between PaaS and **Infrastructure as a Service (IaaS)** is that PaaS is higher level and provides a service to run applications, not computers. IaaS simply provides you with a VM. However, with PaaS, this is abstracted away, and you don't need to worry about setting up and updating an instance.

Amazon Web Services (AWS) is the most established cloud host, and it started by only offering IaaS, although they now offer PaaS options. For example, **Elastic Beanstalk** supports .NET. Even though Windows (and SQL Server) support has

improved, it is still not first class, and Linux is clearly their main platform.

However, now that .NET Core and SQL Server 2016 run on Linux, this may be less of a problem. You will pay a premium for Windows server instances, but you're also charged more for some enterprise Linux distributions (**Redhat** and **SUSE**). However, you can run other Linux distributions, such as **Ubuntu** or **CentOS** without paying a premium.

Google Cloud Platform used to consist of **App Engine** (**GAE**), which was quite a restrictive PaaS, but it is getting more flexible. They now offer a generic IaaS called **Compute Engine** (**GCE**), and there's a new flexible version of GAE, which is more like GCE. These new offerings are useful, as you couldn't traditionally run .NET on GAE, and they also provide other products such as pre-emptible VMs and a cloud CDN (with HTTPS at no extra cost).

Azure may still be the best choice for you, and it integrates well with other Microsoft products, such as Visual Studio. However, it is worth having healthy competition because this keeps everyone innovating. It's definitely a good idea to frequently look at the pricing of all options, (which changes regularly) because you can save a lot of money depending on the workload that you run. You can avoid vendor lock-in by avoiding custom services.

Tip

If you are eligible for Microsoft's BizSpark program, then you can get three years of Azure credits (suitable to host a small application). You also get an MSDN subscription for free software licenses.

Whatever hosting provider you choose to use, it is sensible to avoid vendor lock-in and use services that are generic, which you can easily swap for an alternative. For example, using hosted open source software, such as PostgreSQL, Redis, or RabbitMQ, rather than an equivalent custom cloud provider product. You can also take a resilient multicloud approach to protect yourself from an outage of a single provider.

Docker is a great technology for this purpose because many different cloud services support it. For example, you can run the same container on Azure Container Service, Docker Cloud, AWS EC2 Container Service, and Google's **Kubernetes** Container Engine.

Docker also runs on Windows (using Hyper-V), and in the new version of Visual Studio you can deploy to and debug into a container. This can run ASP.NET Core on Linux, and when you are ready to deploy, you can just push to production and have

confidence that it will work as it did on your machine. You can read more about this at docker.com/Microsoft , and there are some interesting videos on the Docker blog.

Note

When choosing a cloud (or even a region), it's important to not only consider the monetary cost. You should also factor in environmental concerns, such as the main fuel used for the power supply. For example, some regions can be cheaper, but this may be because they use dirty coal power, which contributes to climate change and our future security.

Summary

In this chapter, you saw how you might integrate automated testing into a CI system in order to monitor for performance regressions. You also learned some strategies to roll out changes and ensure that tests accurately reflect real life. We also briefly covered some options for DevOps practices and cloud-hosting providers, which together make continuous performance testing much easier.

In the next chapter, we'll wrap-up everything that we covered throughout this book and suggest some areas for further investigation. We'll reinforce our learnings so far, give you some interesting ideas to think about, contemplate possible futures for .NET, and consider the exciting direction the platform is taking.

Chapter 10. The Way Ahead

This chapter sums up what you learned by reading this book. It refreshes the main tenets of performance and it reminds you that you should always remain pragmatic. We'll recap why you shouldn't optimize just for optimization's sake and why you should always measure the problems and results. This chapter also introduces more advanced and exotic techniques that you may wish to consider learning about if you need more speed or are a serious performance enthusiast.

Topics covered in this chapter include the following:

- A summary of previously-covered topics
- Platform invoke and native code
- Alternative architectures, such as ARM
- Advanced hardware (GPUs, FPGAs, ASICs, SSDs, and RAM SANs)
- Machine learning and AI
- Big data and MapReduce
- The Orleans virtual actor model
- Custom transport layers
- Advanced hashing functions
- Library and framework support
- The future of .NET Core

We'll reinforce how to assess and solve performance issues by refreshing your memory of the lessons in the previous chapters. You'll also gain an awareness of other advanced technology that's available to assist you with delivering high performance, which you may wish to investigate further. Finally, we'll highlight what libraries and frameworks support .NET Core and ASP.NET Core, and try to hypothesize possible future directions for these exciting new platforms.

Reviewing what we learned

Let's briefly recap what we covered earlier in this book to serve as an aide-mémoire.

In Chapter 1, *Why Performance Is a Feature* , we discussed the basic premise of this book and showed you why you need to care about the performance of your software. Then, in Chapter 2, *Measuring Performance Bottlenecks*, we showed you that the only way you can solve performance problems is to carefully measure your application.

In Chapter 3, *Fixing Common Performance Problems*, we looked at some of the most frequent performance mistakes and how to fix them. After this, we went a little deeper in Chapter 4, *Addressing Network Performance* , and dug into the networking layer that underpins all web applications. Then in Chapter 5, *Optimizing I/O Performance*, we focused on input/output and how this can negatively affect performance.

In Chapter 6, *Understanding Code Execution and Asynchronous Operations*, we jumped into the intricacies of C# code and looked at how its execution can alter performance. Then in Chapter 7, *Learning Caching and Message Queuing*, we initially looked at caching, which is widely regarded to be quite hard. Then, we investigated message queuing as a way to build a distributed and reliable system.

In Chapter 8, *The Downsides of Performance Enhancing Tools*, we concentrated on the negatives of the techniques that we previously covered, as nothing comes for free. Then in Chapter 9, *Monitoring Performance Regressions*, we looked at measuring performance again but, in this case, from an automation and **Continuous Integration (CI)** perspective.

Further reading

If you've read this far, then you will probably want some pointers for other things to research and read up on. For the rest of this chapter, we'll highlight some interesting topics that you may want to look into further, but we couldn't cover in more detail in this book.

Going native

One of the problems with the old ASP.NET is that it was really slow, which is why one of the main guiding principles of ASP.NET Core has been performance. Impressive progress has already been made, but there are plenty more opportunities for further enhancements.

One of the most promising areas is the native tool chain, which we briefly mentioned in Chapter 6, *Understanding Code Execution and Asynchronous Operations* . It's still in its early days but it looks like it could be very significant.

Previously, if you wanted to call unmanaged native code from managed .NET code, you needed to use Platform Invoke (**PInvoke**), but this had performance overheads and safety concerns. Even if your native code was faster, the overheads often meant it was not worth bothering about.

The native tool chain should give native levels of performance, but with the safety and convenience of a managed runtime. Ahead-of-time compilation is fascinating and very technical, but it can offer a performance boost along with simplified deployments if we know the architecture.

It's also possible to optimize for different processors that may offer special performance features and instructions. For example, targeting low-energy ARM chips instead of the usual Intel style processors.

Processor architecture

Typically, when writing desktop or server software, you will target an Intel-based architecture, such as x86 or x64. However, ARM-based chips are gaining popularity, and they can offer fantastic power efficiency. If software is specially optimized for them, then they can also offer excellent performance.

For example, the **Scratch** graphical programming language, used to teach computing, has been optimized for a Raspberry Pi 3 and it now runs roughly twice as quick as it does on an Intel Core i5. Other software has also been optimized for the ARM processor, for example, the Kodi open source media player.

ARM Holdings is simply an intellectual property company and they don't make any processors themselves. Other companies, such as Apple and Broadcom, license the designs or architecture and fabricate their **System on a Chip** (**SoC**) products.

This means that there are many different chips available, which run multiple different versions of the ARM architecture and instruction set. This fragmentation can make it harder to support unless you pick a specific platform.

Windows 10 IoT Core runs on the Raspberry Pi (version 2 and 3) and it can now be set up using the standard **New Out Of the Box Software** (**NOOBS**) installer. Windows 10 IoT Core is not a full desktop environment to run normal applications, but it does allow you to make hardware projects and program them with C# and .NET. However, for web applications, you would probably run .NET Core on Linux, such as Raspbian (based on **Debian**).

Hardware is hard

We previously mentioned additional hardware that can be used for computation in Chapter 6, *Understanding Code Execution and Asynchronous Operations* , including **Graphics Processing Units (GPUs)**, **Field Programmable Gate Arrays (FPGAs)**, and **Application Specific Integrated Circuits (ASICs)**.

Not only can these devices be used for specific processing tasks, but their storage can be used as well. For example, you can borrow the RAM from a GPU if main memory is exhausted. However, this technique is not required as much as it used to be when memory was more limited.

You may have heard of RAM SANs, which were SANs using standard RAM for permanent storage (with a battery backup). However, these have become less relevant as SSDs (based on flash memory) improved in speed and increased in capacity to the point of replacing mechanical drives for many common tasks.

You can still purchase high performance SANs, but they will probably be based on flash memory rather than RAM. If you use RAM as your main storage (for example, with Redis), then it is important to use **error-correcting code memory** (**ECC**). ECC RAM is more expensive, but it is better suited for server use. However, some cloud instances don't use it or it's hard to find out whether it's even offered because it isn't listed in the specification.

One application of custom computation hardware is **Machine Learning** (**ML**), specifically the deep learning branch of ML, using multilevel neural networks. This technology saw impressive advances in recent years and this led to products, such as self-driving vehicles. ML applications can make very good use of non-CPU processing, particularly GPUs, and NVIDIA provides many tools and libraries to help with this.

Google also recently announced that they built a custom ASIC, called a **Tensor Processing Unit** (**TPU**), to accelerate their TensorFlow machine learning library and cloud services. You can read more at tensorflow.org and cloud.google.com/ml .

Machine learning

You don't only have to use **Artificial Intelligence** (**AI**) to replace drivers and other jobs, such as call center staff or even doctors. You can use some basic ML in your own web applications to provide product suggestions relevant to your customers or analyze marketing effectiveness, just like Amazon or Netflix do.

You don't even need to build your own ML systems, as you can use cloud services, such as **Azure ML**. This allows you to use a graphical drag and drop interface to build your ML systems, although you can also use Python and R.

You still need to know a little bit about data science, such as the elementary principles of binary classification and training data, but even so, it significantly reduces the barrier to entry. Yet, if you want to fully explore ML and big data possibilities, then you probably need a dedicated data scientist.

You can try out Azure ML at studio.azureml.net and you don't even need to register. The following screenshot shows an example of what it looks like:
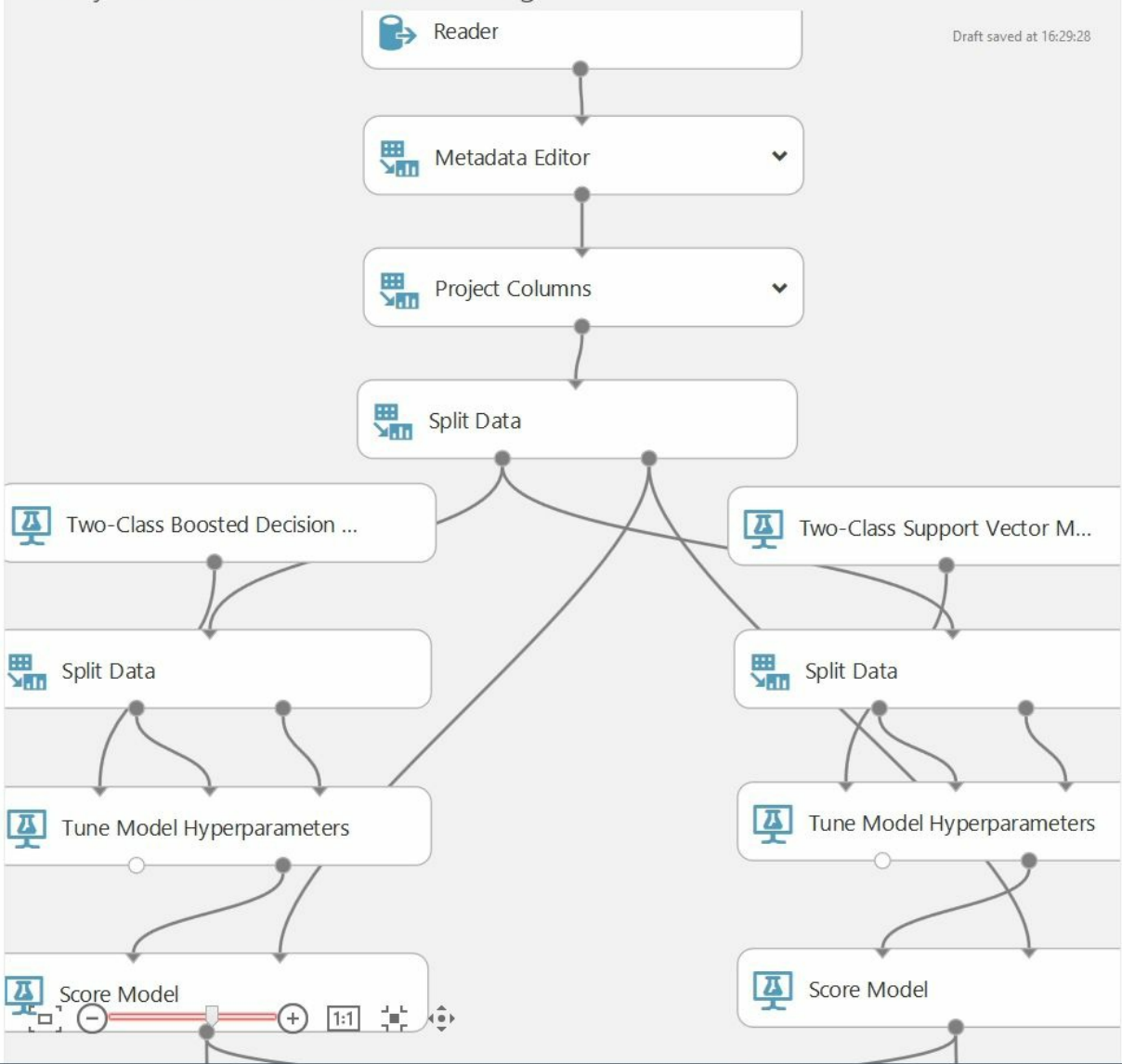
Big data and MapReduce

Big data is probably an overused term these days and what is sometimes described as big is usually more like medium data. Big data is when you have so much information that it is difficult to process, or even store, on a single machine. Traditional approaches often break down with big data, as they are not adequate for the huge automatically-acquired datasets that are common today. For example, the amount of data constantly collected by IoT sensors or by our interactions with online services can be vast.

One caveat of big data and ML is that, although they are good at finding correlations in large sets of data points, you cannot use them to find causations. You also need to be mindful of data privacy concerns and be very careful of not judging someone before they have acted, based only on a predicted predisposition.

Note

Anonymizing data is incredibly difficult and is not nearly as simple as removing personal contact details. There have been many cases of large "anonymized" datasets being released where individuals were later easily identified from the records.

One technology that is useful to analyze big data is **MapReduce**, which is a way of simplifying an operation so that it is suitable to run in parallel on a distributed infrastructure. A popular implementation of MapReduce is Apache Hadoop and you can use this in Azure with **HDInsight**, which also supports related tools, including Apache Spark and Apache **Storm**. Other options to handle large datasets include Google's Cloud Bigtable or **BigQuery**.

You can see the available options for Azure HDInsight in the portal, as shown in the following image. Some features are still in preview or have restrictions, but they may have moved to general availability by the time you read this.

You can see from this image that Spark is in preview and is currently available only on Linux. Hadoop is more established and is also available on Windows, as shown in the following screenshot:

The next image shows that Storm is also available but not on the premium preview (including Microsoft R Server):

Orleans

Another interesting project is an open source framework from Microsoft called **Orleans**, which is a distributed virtual actor model that was used to power the cloud services of Halo Xbox games. What this means is that, if you build your system by separating your logic into separate actors, this allows it to easily scale, based on demand.

In Orleans, actors are known as grains and you can write them in C# by inheriting from an interface. These are then executed by an Orleans server, called a **silo**. Grains can be persisted to storage, such as SQL or **Azure Tables**, to save their state and to reactivate later. Orleans can also make use of the Bond serializer for greater efficiency.

Unfortunately, Orleans does not currently support .NET Core, but the porting work is in progress. Orleans allows you to write simple and scalable systems with low latency and you can read more at dotnet.github.io/orleans .

Custom transports

In [Chapter 4](#), *Addressing Network Performance*, we started with an introduction to TCP/IP and briefly mentioned UDP. We also covered TLS encryption and how to minimize the impact of secure connections while still reaping performance benefits.

UDP is simpler and quicker than TCP, but you either need to not care about reliable delivery (multiplayer games and voice/video chat) or build your own layer to provide this. In Chapter 8, *The Downsides of Performance-Enhancing Tools*, we highlighted **StatsD**, which uses UDP to avoid blocking latency while logging to a remote central server.

There are alternatives to TLS if you aren't constrained to a browser, but if you're developing a web application, this will probably only apply inside of your server infrastructure. For example, the WhatsApp messaging app uses **Noise Pipes** and Curve25519 from the Noise Protocol Framework (noiseprotocol.org) between the smartphone app and their servers in addition to the Signal Protocol for end-to-end encryption.

Using Noise Pipes instead of TLS increases performance because fewer round trips are required to set up a connection. Another option with similar benefits is the **secure pipe daemon** (**spiped**), as used by and created for the secure Linux backup software, **Tarsnap**. However, you do need to preshare the keys, but you can read more about this at www.tarsnap.com/spiped.html.

Advanced hashing

We covered hashing functions a fair amount in this book, especially in Chapter 6, *Understanding Code Execution and Asynchronous Operations* . This area is constantly advancing and it is useful to keep an eye on the future to see what's coming. Although today it is reasonable to use a member of the SHA-2 family for quick hashing and PBKDF2 for slow (password) hashing, this is unlikely to always be the case.

For fast hashing, there is a new family of algorithms called **SHA-3**, which should not be confused with SHA-384 or SHA-512 (which are both in the SHA-2 family). SHA-3 is based on an algorithm called Keccak, which was the winner of a competition to find a suitable algorithm for the new standard. Other finalists included Skein (skein-hash.info) and BLAKE2 (blake2.net), which is faster than MD5, but actually secure.

An algorithm called Argon2 won a similar competition for password hashing (password-hashing.net). To see why this matters, you can visit haveibeenpwned.com (run by .NET security guru Troy Hunt) to see whether your details are in one of the large number of data breaches. For example, LinkedIn was breached and didn't use secure password hashing (only an unsalted SHA-1 hash). Consequently, most of the plain text passwords were cracked. Therefore, if a LinkedIn account password was reused on other sites, then these accounts can be taken over.

It's a very good idea to use a password manager and create strong unique passwords for every site. It is also beneficial to use two-factor authentication (sometimes also called two-step verification) if available. For example, you can do this by entering a code from a smartphone app in addition to a password. This is particularly useful for e-mail accounts, as they can normally be used to recover other accounts.

Library and framework support

There have been some significant changes to .NET Core and ASP.NET Core between RC1 and RC2, more than normal for release candidates. Sensibly, many popular libraries and frameworks were waiting for RC2, or later, before adding support.

Obviously, a book is a bad place to keep up with the changes, so the author has put together a website to display the latest compatibility information. You can find the *ASP.NET Core Library and Framework Support* list at ANCLAFS.com .

If you would like to update anything or add a library or framework, then please send a pull request. The repository is located at github.com/jpsingleton/ANCLAFS and it includes lots of useful tools, libraries, frameworks, and more. We mentioned many of these earlier in this book and the following sample is just a small selection of what is listed because package support will grow over time:

- Scientist.NET
- FeatureSwitcher
- FeatureToggle
- MiniProfiler
- Glimpse
- Prefix
- Dapper
- Simple.Data
- EF Core
- Hangfire
- ImageResizer
- DynamicImage
- ImageProcessorCore
- RestBus
- EasyNetQ
- RabbitMQ Client
- MassTransit
- Topshelf
- Nancy
- SignalR
- Orleans

The future

A quote often attributed to the physicist Niels Bohr goes, as follows:

"Prediction is very difficult, especially about the future."

However, we'll have a go at this anyway, starting with the more straightforward bits. The official ASP.NET Core roadmap lists SignalR, Web Pages, and Visual Basic support shipping after the version 1.0 **Release To Manufacturing/Marketing (RTM)**.

After this, it is fairly safe to assume that features will be added to bring the Core line closer to the existing frameworks. This includes Entity Framework, which is currently missing some of the big features of the full EF, for example, lazy loading.

There is also the move towards the .NET Platform Standard, to enhance portability across .NET Core, the .NET Framework and Mono. For example, .NET Core 1.0 and .NET Framework 4.6.3 both implement .NET Platform Standard 1.6 (`netstandard1.6`). However, you probably don't need to worry about this unless you are writing a library. Refer to the documents at dotnet.github.io/docs/core-concepts/libraries if you are.

Microsoft has said it will listen to user feedback and use it to drive the direction of the platforms, so you have a voice. Even if you don't, then, as the code is all open source, you can help shape the future by adding the features that you want.

Further into the future is harder to predict, but there has been a steady stream of projects being open sourced by Microsoft from the early offerings of ASP.NET MVC to the Xamarin framework for cross-platform app development.

It's an exciting time to be working with C# and .NET, especially if you want to branch out from web development. The Unity game engine is now part of the .NET foundation, and there are some interesting recent developments in **Virtual Reality (VR)** or Augmented Reality (**AR**) hardware. For example, Microsoft Hololens, Oculus Rift, Samsung Gear VR, and HTC Vive are all unbelievably better than the basic VR that came out a couple of decades ago.

It's also a great time to be looking at IoT, which, while it may still be looking for its killer app, has so much more cheap and powerful hardware available. A Raspberry Pi Zero costs only $5 and it now supports an upgraded camera module. With a computer such as the Raspberry Pi 3, which offers almost desktop class performance for $35, anyone can now easily learn to code (perhaps in C# or .NET) and make

things, especially children.

Following the wisdom of Alan Kay:

"The best way to predict the future is to invent it."

So, get out there and make it! And make sure that you share what you've done.

Summary

We hope that you enjoyed this book and learned how to make your web applications high-performing and easy to maintain, particularly when you use C#, ASP.NET Core and .NET Core. We tried to keep as much advice as possible applicable to general web app development, while gently introducing you to the latest open source frameworks from Microsoft and others.

Clearly a topic such as this changes quite rapidly, so keep your eyes open online for updates. There is an announcements repository on GitHub that is worth watching at github.com/aspnet/announcements . Hopefully, a lot of the lessons in this book are generally good ideas and they will still be sensible for a long time to come.

Always keep in mind that optimizing for its own sake is pointless, unless it's just an academic exercise. You need to measure and weigh the benefits against the downsides; otherwise, you may end up making things worse. It is easy to make things more complex, harder to understand, difficult to maintain, or all of these.

It's important to instill these pragmatic performance ideas into team culture, or else teams won't always adhere to them. However, above all, remember to still have fun!

Bibliography

This learning path has been prepared for you to explore ASP.NET and learn to use them in your web applications.It comprises of the following Packt products:

- *Learning ASP.NET Core MVC, Mugilan T. S. Ragupathi*
- *ASP.NET Core and Angular 2, Valerio De Sanctis*
- *ASP.NET Core 1.0 High Performance, James Singleton*