C# 6 and .NET Core 1.0

C# 6 and .NET Core 1.0 has been divided into three high-impact sections to help start putting these new features to work. First, we’ll run you through the basics of C#, as well as object-oriented programming, before taking a quick tour through the latest features of C# 6, such as string interpolation for easier variable value output, exception filtering, and how to perform static class imports.

After this, we’ll dive into the internals of the .NET class libraries, covering topics such as performance, monitoring, debugging, internationalization, serialization, and encryption.

The final section will demonstrate the major types of application that you can build and deploy cross-device and cross-platform. Lastly, we’ll help you build a complete application that can be hosted on all of today’s most popular platforms, including Linux and Docker.

By the end of the book, you’ll be armed with all the knowledge you need to build modern, cross-platform applications using C# and NET Core.

Who this book is written for

Are you struggling to get started with C#? Or maybe you’re interested in the potential of the new cross-platform features that .NET Core can offer? If so, C# 6 and .NET Core 1.0 is the book for you. Whilst you don’t need to know any of the latest features of C# or .NET to get started, it would be beneficial if you have some programming experience.

What you will learn from this book

- Build cross-platform applications using C# 6 and .NET Core 1.0
- Explore ASP.NET Core 1.0 and learn how to create professional web applications
- Improve your application’s performance using multithreading
- Use Entity Framework Core 1.0 and learn how to build Code-First data models
- Master object-oriented programming with C# to increase code reuse and efficiency
- Familiarize yourself with cross-device app development using the Universal Windows Platform and XAML
- Query and manipulate data using LINQ
- Protect your data by using encryption and hashing

Mark J. Price

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 14 'Building Web Applications and Services Using ASP.NET Core'
- A synopsis of the book’s content
- More information on **C# 6 and .NET Core 1.0**
About the Author

Mark J. Price is a Microsoft Certified Trainer (MCT) and Microsoft Specialist, Programming in C# and Architecting Microsoft Azure Solutions, with more than 20 years of educational and programming experience.

Since 1993, Mark has passed more than 90 Microsoft programming exams and specializes in preparing others to pass them too. His students range from professionals with decades of experience to 16-year-old apprentices with no experience at all. He successfully guides all of them by combining educational skills with real-world experience, consulting on and developing systems for enterprises worldwide.

Between 2001 and 2003, Mark was employed full-time to write official courseware for Microsoft in Redmond, USA. His team wrote the first training courses for C# and .NET while they were still an early alpha version. While with Microsoft, he delivered "train-the-trainer" classes to get other MCTs up to speed in C# and .NET.

In 2010, Mark took a postgraduate certificate in education (PGCE). He taught GCSE and A-level mathematics in two secondary schools in London. Mark holds a Computer Science BSc (Hons) degree from the University of Bristol, UK.
Preface

There are many C# books, some, more than a thousand pages long, that aim to be comprehensive references to the C# programming language and the .NET Framework.

This book is different—it is concise and aims to be a fast-paced read that is packed with hands-on walkthroughs. I wrote this book to be the best step-by-step guide to modern cross-platform C# and .NET proven practices.

I will point out the cool corners and gotchas of C# so you can impress colleagues and employers and quickly get productive. Rather than slowing down and boring some of you by explaining every little thing, I assume that if a term I use is new to you, then you know how to Google it with a search engine such as DuckDuckGo.

At the end of each chapter, there is a section entitled Practicing and Exploring, which contains questions to test your knowledge, and usually a hands-on practical exercise, and you will explore topics in depth on your own with a little nudge in the right direction from me.

You can download solutions for the exercises from the GitHub repository at https://github.com/markjprice/cs6dotnetcore. I will provide instructions on how to do this using Visual Studio 2015 at the end of Chapter 1, Hello C#!, Welcome .NET Core!.

What this book covers

Chapter 1, Hello C#!, Welcome .NET Core!, is about setting up your development environment and using various tools to create the simplest application possible with C#. You will learn how to compile C# code at the Command Prompt and how to write and compile code using Visual Studio. You will also learn about the different .NET platforms: .NET Framework, .NET Core, and .NET Native.
Chapter 2, *Speaking C#*, is about the C# language, the grammar and vocabulary that you will use every day to write the source code for your applications. In particular, you will learn how to declare and work with variables of different types.

Chapter 3, *Controlling the Flow, Converting Types, and Handling Exceptions*, is about writing code that makes decisions, repeats blocks of statements, converts between types, and handles errors. You will also learn the best places to look for help.

Chapter 4, *Using Common .NET Types*, is about how .NET types are related to C#. You will learn about .NET Framework, .NET Core, and their class library assemblies of types that allow your applications to connect together existing components to perform common practical tasks.

Chapter 5, *Using Specialized .NET Types*, is about .NET types used to diagnose problems, support multiple languages and cultures, and access features and applications outside of .NET.

Chapter 6, *Building Your Own Types with Object-Oriented Programming*, is about all the different categories of members that a type can have, including fields for storing data and methods for performing actions. You will use OOP concepts such as aggregation and encapsulation.

Chapter 7, *Implementing Interfaces and Inheriting Classes*, is about deriving new types from existing ones using OOP. You will learn how to implement interfaces, about base and derived classes, how to override a type member, how to use polymorphism, and how to cast between classes in an inheritance hierarchy.

Chapter 8, *Working with Relational Data Using the Entity Framework*, is about reading and writing to Microsoft SQL Server (and other databases) using classic ADO.NET and the object-relational mapping technology known as Entity Framework.

Chapter 9, *Querying and Manipulating Data with LINQ*, is about Language Integrated Queries (LINQ)—language extensions that add the ability to work with sequences of items, and filter, sort, and project them into different outputs.

Chapter 10, *Working with Files, Streams, and Serialization*, is about reading and writing to files and streams, text encoding, and serialization.

Chapter 11, *Protecting Your Data and Applications*, is about protecting your data using encryption and hashing, and checking who is running your application and what they are allowed to do.

Chapter 12, *Improving Performance and Scalability with Multitasking*, is about allowing multiple actions to be executed at the same time to improve performance, scalability, and user productivity.
Chapter 13, Building Universal Windows Platform Apps Using XAML, is about learning the basics of XAML, which can be used to define the user interface for a graphical app for the Universal Windows Platform (UWP). This app can then run on Windows 10, Windows 10 Mobile, Xbox One, and even HoloLens.

Chapter 14, Building Web Applications and Services Using ASP.NET Core, is about building web applications and services using a modern HTTP architecture on the server side using Microsoft ASP.NET Core 1.0. You will learn about the models, views, and controllers that make up MVC and the Web API.

Chapter 15, Taking C# Cross-Platform, is about introducing you to how you can take C# cross-platform using .NET Core, ASP.NET Core 1.0, Entity Framework Core 1.0, and Visual Studio Code.

Chapter 16, Building a Quiz, is about designing and building a quiz application that helps students learn C#, .NET Core, and related topics.

Appendix A, Answers to the Test Your Knowledge Questions, has the answers to the test questions at the end of each chapter.

Appendix B, Creating a Virtual Machine for Your Development Environment, shows how to set up a virtual machine in Microsoft Azure for use as a development environment.
This chapter is about building web applications and services using a modern HTTP architecture on the server-side using Microsoft ASP.NET Core 1.0. You will learn about the startup configuration, routes, models, views, and controllers that make up ASP.NET Core.

This chapter will cover the following topics:

- Understanding ASP.NET Core
- ASP.NET Core startup
- ASP.NET Core controllers
- ASP.NET Core models
- ASP.NET Core views
- Taking ASP.NET Core further
- ASP.NET Core Web API
Understanding ASP.NET Core

Microsoft ASP.NET Core 1.0 is part of a suite of Microsoft technologies to build web applications and services that have evolved over the years as shown in the following bullet points:

- ASP.NET Web Forms was released in 2002 and is designed to enable non-web developers, such as those familiar with Visual Basic, to quickly create web applications. Web Forms can only be hosted on Windows, but are used in products such as Microsoft SharePoint. It should be avoided for new projects in favor of ASP.NET Core.

- ASP.NET XML Web Services was released in 2002 and enables developers to build SOAP services. It should be avoided for new projects in favor of WCF or ASP.NET Web API.

- Windows Communication Foundation (WCF) was released in 2006 and enables developers to build SOAP and REST services. SOAP is powerful but complex, so it should be avoided unless you need advanced features, such as distributed transactions and complex messaging topologies.

- ASP.NET MVC was released in 2009 and is designed to cleanly separate the concerns of web developers between the models that represent the data, the views that present that data, and the controllers that fetch the model and pass it to a view. This separation enables improved scalability and unit testing.

- ASP.NET Web API was released in 2012 and enables developers to create HTTP services that are simpler and more scalable than SOAP services.

- ASP.NET Core was released in 2016, and combines MVC and Web API running on the .NET Core. Therefore, it is cross-platform.

**Best Practice**

Choose ASP.NET Core to develop web applications and services, because it includes MVC and Web API, which are modern and cross-platform.
Classic ASP.NET versus modern ASP.NET Core

ASP.NET is almost 15 years old. Until now, it has been built on top of a major part of the .NET Framework, the `System.Web.dll` assembly. Over the years, it has accumulated a lot of features, many of which are not suitable for modern, cross-platform development.

ASP.NET Core is a major redesign of ASP.NET. It removes the dependency on the `System.Web.dll` assembly, and is composed of modular lightweight components.

You can develop and run ASP.NET Core applications cross-platform on Windows, Mac OS X, and Linux. The entire stack is open source, and it is designed to integrate with a variety of client-side frameworks, including AngularJS, KnockoutJS, and Bootstrap.

Client-side web development

When building web applications, a developer needs to know more than just C# and .NET. On the client (that is, in the web browser), you will use a combination of the following components of a web page:

- **HTML5**: This is used for the content and semantics of elements
- **CSS3**: This is used for the format and layout applied to elements
- **JavaScript**: This is used for the behavior of elements

This book is about C#, so we will cover some of the basics. For more information on web browser technologies, I recommend that you refer to the book *HTML5 Web Application Development By Example* by Packt Publishing at https://www.packtpub.com/web-development/html5-web-application-development-example-beginners-guide.

To make it easier to work with HTML5, CSS3, and JavaScript, I recommend that developers install the free Web Essentials extension for Visual Studio.
Install Web Essentials 2015

Start Microsoft Visual Studio 2015. On the Tools menu, choose Extensions and Updates. In the left-hand list, select Online. In the search box, enter web essentials, select Web Essentials 2015.1, click on the Download button, and then follow the instructions:

The HyperText Transfer Protocol (HTTP)

To communicate with a server, the client makes calls over the network using a protocol known as HTTP. HTTP is the technical underpinning of the "web". So when we talk about "web" applications or "web" services, we mean they use HTTP to communicate between a client (often a web browser) and a server.

A client makes an HTTP request for a resource, such as a page identified by a URL (Uniform Resource Locator), and the server sends back an HTTP response. You can use Microsoft Edge and other browsers to record requests and responses.

Start Microsoft Edge. Press F12 to show developer tools. Click on the Network tab. If the Start profiling session button has not been pressed, click on it to start recording, or press Ctrl + E:
In Microsoft Edge's address box, enter http://www.asp.net/mvc.

In the F12 Developer Tools window, in the list of recorded requests, click on the first entry:

<table>
<thead>
<tr>
<th>mvc</th>
<th>HTTP</th>
<th>GET</th>
<th>200</th>
<th>text/html</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.asp.net/">http://www.asp.net/</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On the right-hand side, you will see details about the request and the response:

<table>
<thead>
<tr>
<th>Headers</th>
<th>Body</th>
<th>Parameters</th>
<th>Cookies</th>
<th>Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request URL: <a href="http://www.asp.net/mvc">http://www.asp.net/mvc</a></td>
<td>Request Method: GET</td>
<td>Status Code: 200 / OK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Request Headers**
- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/jxr,*/*
- Accept-Encoding: gzip, deflate
- Accept-Language: en-US, en-GB; q=0.7, en; q=0.3
- Connection: Keep-Alive
- Cookie: omnilD=2e92247_b4ba_4d81_b19b_1739f9a94c2a...
- Host: www.asp.net
- User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/...

**Response Headers**
- Cache-Control: private
- Content-Encoding: gzip
- Content-Length: 17191
- Content-Type: text/html; charset=utf-8
- Date: Thu, 05 Nov 2015 07:53:29 GMT
- Server: Microsoft-IIS/8.0
- Vary: Accept-Encoding
- X-Frame-Options: SAMEORIGIN
- X-Powered-By: ARR/2.5

Note the following aspects:

- The request method is **GET**. Other methods that HTTP defines include **POST**, **PUT**, **DELETE**, **HEAD**, and **PATCH**.
- The response status code is **200 OK**. This means the server found the resource the browser requested. Other status codes include **404 Missing**.
• The request headers include what formats the browser will accept. In this case, the browser is saying it understands HTML, XHTML, and others.
• The browser has told the server that it understands the GZIP and DEFLATE compressions algorithms.
• The browser has told the server which human languages it would prefer: US English, British English, and then any other dialect of English.
• I have been to this site before, so a cookie that was defined by the site is being sent to the server so that it can track me. Microsoft has named it omniID. Does that sound ominous to you?
• The server has sent back the response, compressed using the GZIP algorithm, because it knows that the client can decompress that format.
• The server is running Internet Information Services (IIS) 8.0.

Create a web application project
In Visual Studio, press Ctrl + Shift + N or choose File | New | Project....

In the New Project dialog, in the Installed Templates list, select Visual C#. In the center list, select ASP.NET Web Application, type the name as Ch14/WebApp, type the solution name as Chapter14, and then click on OK:

In the New ASP.NET Project dialog box, in the ASP.NET Core 1.0 Templates section, select the Web Application template.
At the time of writing this book, the final version of ASP.NET Core 1.0 and its Visual Studio project templates have not been released. So, the screenshots in this book show the old branding, ASP.NET 5!

Click on the Change Authentication button and choose No Authentication. Then, click on OK. This tells Visual Studio to allow all users access to the application. We can change this later to secure our web application.

Uncheck the Host in the cloud box, because we want to run this locally. We can always deploy to the Microsoft Azure cloud later.

Click on OK:
After a few seconds, your Solution Explorer window will look like this:

Run the application by pressing F5. The web application is hosted in the free version of IIS Express using a random port number for local testing.

Notice that the home page has a black navigation bar with links to an **About** page and a **Contact** page. The home page has a carousel with four rotating items and four columns of links underneath:
Click on the links to the **About** page and the **Contact** page, and notice the URLs that are used. To get to the **About** page, the user could enter `/Home/About` in the browser address box:

![About page screenshot](image)

Resize the width of the browser window and notice that the navigation bar becomes a clickable "hamburger" menu (three horizontal lines):

![Resized browser window](image)
This is an example of responsive web design. The page looks good on both desktop and mobile devices, by automatically adapting based on the current width of the window:

Explain the parts of an ASP.NET Core web application
Let's walk through the parts that make up a modern ASP.NET Core application.

In Visual Studio, look at Solution Explorer for the Ch14_WebApp project and note the following points:

- **wwwroot**: This folder contains static content, such as stylesheets, images, scripts, and common libraries, that combine resources such as jQuery and Bootstrap
- **Dependencies**: This folder contains Bower and npm for modern package management
- **Controllers**: This folder contains C# classes that have methods (known as actions) that fetch a model and pass it to a view
- **Models**: This (optional) folder contains C# classes that represent all the data required for a request
- **Views**: This folder contains .cshtml files that combine HTML and C# code to enable the dynamic generation of an HTML response
- **project.json**: This file contains a list of NuGet packages (such as the Entity Framework Core) that your project requires and other project configuration
• **Startup.cs**: This file contains the Main entry point for your application and configures the services, pipeline, and routes for your application.

The following screenshot shows the parts of a typical ASP.NET Core project:

![Solution Explorer](image)

**ASP.NET Core startup**

In the **Solution Explorer** window, double-click on the **Startup.cs** file.

Notice the `ConfigureServices` method that adds support for MVC. Later, we will add statements here to add support for the Entity Framework Core:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
}
```
Next, we have the `Configure` method.

The most important statement here is the one that calls `UseMvc` and maps a default route. This route is very flexible, because it would match almost any incoming URL:

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
    if (env.IsDevelopment())
    {
        app.UseBrowserLink();
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    app.UseIISPlatformHandler();
    app.UseStaticFiles();
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}"
        );
    });
}
```

### Understanding the default route

The default route looks at any URL entered by the user in the address bar and matches it to extract the name of a controller, the name of an action, and an optional id value (the ? symbol makes it optional). If the user hasn't entered these names, it uses defaults of Home for the controller, and Index for the action (the = assignment sets a default for a named segment).

Contents in curly brackets {} are called **segments**, and they are like a named parameter of a method. The value of these segments can be any string.

The responsibility of a route is to discover the name of a controller and an action.
Here's a table of example URLs and how MVC would work out the names. Notice that if the user does not supply a name, then the defaults Home and Index are used, as specified when the route was registered. You could change these defaults if you wanted:

<table>
<thead>
<tr>
<th>URL</th>
<th>Controller</th>
<th>Action</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Home</td>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>/Muppet</td>
<td>Muppet</td>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>/Muppet/Kermit</td>
<td>Muppet</td>
<td>Kermit</td>
<td></td>
</tr>
<tr>
<td>/Muppet/Kermit/Green</td>
<td>Muppet</td>
<td>Kermit</td>
<td>Green</td>
</tr>
<tr>
<td>/Products</td>
<td>Products</td>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>/Products/Detail</td>
<td>Products</td>
<td>Detail</td>
<td></td>
</tr>
<tr>
<td>/Products/Detail/3</td>
<td>Products</td>
<td>Detail</td>
<td>3</td>
</tr>
</tbody>
</table>

**ASP.NET Core controllers**

Now that MVC knows the names of the controller and action, it will look for a class that implements an interface named IController. To simplify the requirements, Microsoft supplies a class named Controller that your classes can inherit from.

The responsibilities of a controller are as follows:

- To extract parameters from the HTTP request
- To use the parameters to fetch the correct model and pass it to the correct view
- To return the results from the view to the client as an HTTP response

**Defining the Home controller's actions**

In the Solution Explorer window, expand the Controllers folder and double-click on the file named HomeController.cs:

```csharp
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    public IActionResult About()
    {
        ViewData["Message"] = "Your application description page.";
    }
}
```
```csharp
return View();
}
public IActionResult Contact()
{
    ViewData['Message'] = "Your contact page.";
    return View();
}
public IActionResult Error()
{
    return View();
}
}
```

If the user enters `/`, or `/Home`, then it is the equivalent of `/Home/Index` because those were the defaults.

Note the following aspects:

- None of the action methods currently use a model
- Two of the action methods use a dictionary named `ViewData` to store a string message that can then be read inside a view
- All action methods execute a method named `View` and return the results to the client

### ASP.NET Core models

In MVC, the model represents the data required for a request. For example, an HTTP GET request for `http://www.example.com/products/details/3` would mean you are asking for the details of product number 3.

The controller would need to use the ID value 3 to retrieve the record for that product and pass it to a view that can then turn the model into HTML for display in the browser.

For this initial example, we will create an Entity Data Model (EDM) to access data in the Northwind database.

**Best Practice**

Use a data repository (typically implemented as a service) to manage your data.
Create Entity models for Northwind

If you have not already installed the Northwind database, then follow the instructions at the beginning of Chapter 8, Working with Relational Data Using the Entity Framework.

On the Tools menu, choose NuGet Package Manager and then Package Manager Console.

In the PM> prompt, enter the following commands:

Install-Package EntityFramework.MicrosoftSqlServer
Install-Package EntityFramework.Commands
Install-Package EntityFramework.MicrosoftSqlServer.Design:

If the preceding commands give errors, then add -pre to the end of each command to install the pre-release version as shown in the following screenshot.

In the Solution Explorer window, double-click on project.json. Notice that three dependencies have been added for the Entity Framework Core:

```json
{
  "version": "1.0.0-*",
  "compilationOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
    "EntityFramework.Commands": "7.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer.Design": "7.0.0-rc1-final",
    "Microsoft.AspNet.Diagnostics": "1.0.0-rc1-final",
```

[385]
In the `commands` section, add a new command for `ef`, as follows:

```json
"commands": {
    "ef": "EntityFramework.Commands"
},
```

Save `project.json` file.

In the **Solution Explorer** window, right-click on the project and choose **Add, New Folder...**, and then enter the name of the model.

Right-click on the **Ch14_WebApp** project and choose **Open Folder in File Explorer**. Click in the address box and copy the path to the clipboard by pressing `Ctrl + C`.

Click the Windows Start button and start Command Prompt. In Command Prompt window, enter `cd`, and then right-click to paste the path to your project. Press `Enter` to change to that directory:

```
cd C:\Code\Chapter14\src\Ch14_WebApp
```

Enter the following command to tell the .NET Version Manager to use the latest version of .NET Core:

```
dnvm use 1.0.0-rc1-update1
```

At the time of writing this book, .NET Core was a release candidate. By the time you follow these instructions, the final version should be available, and use the new .NET CLI so the commands `dnvm` and `dnx` will be replaced by the command `dotnet`.

Enter the following command to generate classes that represent entities for all the tables in the Northwind database in the **Models** subfolder:

```
dnx ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;" EntityFramework.MicrosoftSqlServer --outputDir Models
```

In the **Solution Explorer** window, expand the **Models** folder. You will see that a class has been created for each entity, and a class named **NorthwindContext** has been created to represent the whole database:
Best Practice

Create a separate class library project for your entity models. This allows easier sharing between servers and clients. We have not done this for this example to keep it simple for now.

Configure Entity Framework Core as a service

Dependency injection is central to ASP.NET Core. Services, such as the Entity Framework Core that are needed by MVC controllers, must be registered as a service during startup.

In the Solution Explorer window, open the Startup.cs file. Import the following namespaces:

```csharp
using Ch14_WebApp.Models;
using Microsoft.Data.Entity;
```

Add the following statements to the ConfigureServices method:

```csharp
var connection = @"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;";
services.AddDbContext<NorthwindContext>(options => options.UseSqlServer(connection));
```

Since we are setting the database connection string in the ASP.NET Core startup, it does not need to be done in the NorthwindContext class.
In the Solution Explorer window, in the Models folder, open the NorthwindContext.cs file and delete the following method:

```csharp
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Nort
    hwind;Trusted_Connection=True;");
}
```

### Create view models for requests

Imagine that when a user comes to our website, we want to show them a list of products and a count of the number of visitors we have had this month. All the data that we want to show in response to a request is the MVC model, sometimes called a view model because it is a model that is passed to a view.

In the Solution Explorer window, select the Models folder. On the Project menu, choose Add Class..., and in the dialog box, choose Class and name it HomeIndexViewModel.

Modify the class definition to make it look like this:

```csharp
public class HomeIndexViewModel
{
    public int VisitorCount;
    public ICollection<Products> Products { get; set; }
}
```

### Fetch the model in the controller

Open the HomeController class. Import the Ch14_WebApp.Models namespace.

Add a field to store a reference to a NorthwindContext instance and initialize it in a constructor:

```csharp
private NorthwindContext db;

public HomeController(NorthwindContext injectedContext)
{
    db = injectedContext;
}
```

Modify the contents of the Index action method to make it look like this:

```csharp
var model = new HomeIndexViewModel
```
Note that we will simulate a visitor count using the Random class to generate a number between 1 and 1000.

If you build the project now, you might get "CS0833 An anonymous type cannot have multiple properties with the same name" errors due to bugs in the scaffolding feature. To fix the errors, simply delete the duplicate members.

ASP.NET Core views

The responsibility of a view is to transform a model into HTML or other formats. There are multiple view engines that can be used to do this. The default view engine for ASP.NET MVC 3 and later is called Razor, and it uses the @ symbol to indicate server-side code execution.

Rendering the Home controller's views

In the Solution Explorer window, expand the Views folder. Expand the Home folder. Note the three files with the .cshtml file extension.

When the view method is executed in a controller action, MVC looks in the Views folder for a subfolder with the same name as the current controller, that is, Home. It then looks for a file with the same name as the current action, that is, Index, About, or Contact.

In the Index.cshtml file, notice the block of code wrapped in @{  }. This will execute first and can be used to store data that needs to be passed into a shared layout file:

```
@{
    ViewData["Title"] = "Home Page";
}
```
Note the static HTML content in several DIV elements that uses Bootstrap for styling.

**Best Practice**
Instead of defining your own styles, use a library, such as Bootstrap, that implements responsive design using standard CSS3 principles.

## Sharing layouts between views

There is a file, named `_ViewStart.cshtml`, that gets executed by the `View` method. It is used to set defaults that apply to all views.

For example, it sets the `Layout` property of all views to a shared layout file:

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

In the `Shared` folder, open the `_Layout.cshtml` file. Notice that the title is being read from the `ViewData` dictionary that was set earlier in the `Index.cshtml` view:

```
<title>@ViewData["Title"] – Ch14_WebApp</title>
```

Note the rendering of common styles to support Bootstrap and the two sections. During development, the fully-commented and nicely formatted versions of CSS files will be used. For staging and production, the minified versions will be used:

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Note the rendering of hyperlinks to allow users to click between pages using the navigation bar at the top of every page. The `<a>` elements use "tag helper" attributes to specify the controller name and action name that will execute when the link is clicked:
Note the rendering of the body:

@RenderBody()

Note the rendering of script blocks at the bottom of the page (so that it doesn't slow down the display of the page):

<environment names="Development">
  <script src="/lib/jquery/dist/jquery.js"></script>
  <script src="/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <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 src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/
             bootstrap.min.js"
        asp-fallback-src="/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn &&
                          window.jQuery.fn.modal">
  </script>
  <script src="/js/site.min.js" asp-append-version="true"></script>
</environment>

You can add your own script blocks into an optional defined section named scripts:

@RenderSection("scripts", required: false)

### Defining custom styles

In the wwwroot\css folder, open the site.css file.

Add a new style that will apply to an element with the newspaper ID, like this:

```css
#newspaper {
    column-count: 3;
}
```
Notice that the `column-count` property has a purple squiggle under it. When you hover over it, you will see that only Internet Explorer 10 (or later) and Opera 11.6 (or later) currently implement this standard. Luckily, Web Essentials can fix this for us.

Click on the `column-count` property, hover your mouse over the small blue bar, click on the button to pop open a menu, and then click on **Add missing vendor specifics**.

You will see that Web Essentials has added two extra statements. They are faded out to indicate that they will automatically change when you change the value for the original `column-count` property:

```css
#newspaper {
    -moz-column-count: 3;
    -webkit-column-count: 3;
    column-count: 3;
}
```


### Defining a typed view

To improve the IntelliSense when writing a view, you can define the type the view can expect using a `@model` directive at the top.

Back in the `Index.cshtml` view, enter the following code as the first line of the file:

```csharp
@model Ch14_WebApp.Models.HomeIndexViewModel
```

Now whenever we enter `@Model`, Visual Studio's code editor will know the correct type and will provide IntelliSense.

To declare the type of the model, use `@model` (with lowercase `m`). To read the model, use `@Model` (with uppercase `M`).

In `Index.cshtml`, delete all the `<div>` elements and replace them with this code:

```html
<div class="row">
    <div class="col-md-12">
        <h1>Northwind</h1>
    </div>
</div>
```
We have had @Model.VisitorCount visitors this month.

<h2>Products</h2>

<div id="newspaper">
<ul>
@foreach (var item in @Model.Products)
{
<li><a asp-controller="Home" asp-action="ProductDetail" asp-route-id="@item.ProductID">@item.ProductName costs @item.UnitPrice.Value.ToString("C")</a></li>
}
</ul>
</div>

Note how easy it is to mix static HTML elements, such as <ul> and <li>, with C# code to output the list of product names.

Note the <div> element with the id attribute of newspaper. This will use the custom style that we defined earlier, so all the content in that element will display in three columns.

To test the web application, press F5. The results in Microsoft Edge will look something like this:
Taking ASP.NET Core further

Now that you've seen the basics of how models, views, and controllers work together to provide a web application, let's look at some common scenarios, such as passing parameters and annotating models.

Passing parameters using a route value

Back in the HomeController class, add the following action method. It uses something called the default model binder to automatically match the id passed in the route to the parameter named id in the method.

```
public IActionResult ProductDetail(int? id)
{
    if (!id.HasValue)
    {
        return HttpNotFound("You must pass a product ID in the route, for example, /Home/ProductDetail/21");
    }
    var model = db.Products.SingleOrDefault(p => p.ProductID == id);
    if (model == null)
    {
        return HttpNotFound($"A product with the ID of {id} was not found.");
    }
    return View(model); // pass model to view
}
```

Now we need to create a view for this request.

Inside the Views folder, right-click on Home and choose Add | New Item....
Choose MVC View Page and name it ProductDetail.cshtml:

Modify the contents as shown here:

```csharp
@model Ch14_WebApp.Models.Products
@

ViewData["Title"] = "Product Detail - " + Model.ProductName;

<h2>Product Detail</h2>
<hr />
<div>
<dl class="dl-horizontal">
  <dt>Product ID</dt>
  <dd>@Model.ProductID</dd>
  <dt>Product Name</dt>
  <dd>@Model.ProductName</dd>
  <dt>Category ID</dt>
  <dd>@Model.CategoryID</dd>
  <dt>Unit Price</dt>
  <dd>@Model.UnitPrice.Value.ToString("C")</dd>
  <dt>Units In Stock</dt>
  <dd>@Model.UnitsInStock</dd>
</dl>
</div>
```

Test the new action and view by pressing F5.
When the home page appears with the list of products, click one of them, for example, product 26. The result should look something like this:

![Product Detail](image)

**Passing parameters using a query string**

In the `HomeController` class, import the `Microsoft.Data.Entity` namespace.

Add a new action method like this:

```csharp
public IActionResult ProductsThatCostMoreThan(decimal? price)
{
    if (!price.HasValue)
    {
        return HttpNotFound("You must pass a product price in the query string, for example, /Home/ProductsThatCostMoreThan?price=50");
    }
    if (model.Count() == 0)
    {
        return HttpNotFound($"No products cost more than {price:C}." undertaken); // pass model to view
    }
    ViewData["MaxPrice"] = price.Value.ToString("C");
    return View(model); // pass model to view
}
```

Inside the Views folder, right-click on `Home` and choose Add | New Item....

Choose MVC View Page and name it `ProductsThatCostMoreThan.cshtml`. 

---

[396]
Chapter 14

Modify the contents like this:

```csharp
@model IEnumerable<Ch14_WebApp.Models.Products>
@
    ViewData["Title"] = "Products That Cost More Than " + ViewData["MaxPrice"];
}
<h2>Products That Cost More Than @ViewData["MaxPrice"]</h2>
<table class="table">
    <tr>
        <th>@Html.DisplayNameFor(model => model.Category.CategoryName)</th>
        <th>@Html.DisplayNameFor(model => model.Supplier.CompanyName)</th>
        <th>@Html.DisplayNameFor(model => model.ProductName)</th>
        <th>@Html.DisplayNameFor(model => model.UnitPrice)</th>
        <th>@Html.DisplayNameFor(model => model.UnitsInStock)</th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Category.CategoryName)</td>
            <td>@Html.DisplayFor(modelItem => item.Supplier.CompanyName)</td>
            <td>@Html.DisplayFor(modelItem => item.ProductName)</td>
            <td>@Html.DisplayFor(modelItem => item.UnitPrice)</td>
            <td>@Html.DisplayFor(modelItem => item.UnitsInStock)</td>
        </tr>
    }
```
In the **Views** folder, in the **Home** folder, open **Index.cshtml** file and add the following `div` element at the bottom of the file. This will provide a form for the user to enter a price. The user can then click on a submit button to call the action method that shows only products that cost more than the entered price:

```html
<div class="row">
  <form asp-action="ProductsThatCostMoreThan" method="get">
    <input name="price" placeholder="Enter a product price" />
    <input type="submit" />
  </form>
</div>
```

Run the web application by pressing **F5**. On the home page, scroll down and enter a price in the form. Then, click on **Submit Query**.
You will see a table of the products that cost more than the price you entered:

<table>
<thead>
<tr>
<th>CategoryName</th>
<th>CompanyName</th>
<th>ProductName</th>
<th>UnitPrice</th>
<th>UnitsInStock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meat/Poultry</td>
<td>Tokyo Traders</td>
<td>Mishu Kobe Niku</td>
<td>97.00</td>
<td>29</td>
</tr>
<tr>
<td>Seafood</td>
<td>Pavlova, Ltd.</td>
<td>Carnarvon Tigers</td>
<td>62.50</td>
<td>42</td>
</tr>
<tr>
<td>Confections</td>
<td>Specialty Biscuits, Ltd.</td>
<td>Sir Rodney's Marmalade</td>
<td>81.00</td>
<td>40</td>
</tr>
<tr>
<td>Meat/Poultry</td>
<td>Plutzer Lebensmittelgroßmärkte AG</td>
<td>Thüringer Rosbratwurst</td>
<td>123.79</td>
<td>0</td>
</tr>
<tr>
<td>Beverages</td>
<td>Aux joyeux ecclésiastiques</td>
<td>Côte de Blaye</td>
<td>263.50</td>
<td>17</td>
</tr>
<tr>
<td>Produce</td>
<td>G'day, Mate</td>
<td>Marajimup Dried Apples</td>
<td>53.00</td>
<td>20</td>
</tr>
</tbody>
</table>

### Annotating models

You might have noticed that the column headings in the table used the names of the properties by default. This means that if the property is multiple words, it won't have spaces. We can use data annotations to improve this.

In the **Models** folder, open the **Products** class. Import the `System.ComponentModel.DataAnnotations` namespace.

Add `[Display]` attributes before each property if you want to have a different label, for example, Product Name, Unit Price, Units In Stock, and so on:

```csharp
[Display(Name = "Product Name")]
public string ProductName { get; set; }
```

Apply the `[Display]` attribute to the properties of three other classes: Category's Category Name, Customer's Company Name, and Supplier's Company Name.

Start the web application by pressing **F5**.
Enter a product price and click on **Submit Query**. Notice that the column headings now reflect the display attributes and not the actual property names:

![Table of Products](image)

**ASP.NET Core Web API**

Although HTTP was originally designed to request and respond with HTML and other resources for us to look at, it is also good for building services. Roy Fielding stated, in his doctoral dissertation describing the **Representational State Transfer (REST)** architectural style, that the HTTP standard defines:

- URLs to uniquely identify resources
- Methods to perform common tasks, such as **GET** and **DELETE**
- The ability to negotiate media formats such as XML and JSON

To allow the easy creation of services, ASP.NET Core has combined what used to be two types of controller.

In earlier versions of ASP.NET, you would derive from `ApiController` to create a Web API service, and then register API routes in the same route table that MVC uses.

With ASP.NET Core, you use exactly the same `Controller` base class used with MVC, except the routes are usually configured on the controller itself, using attributes, rather than in the route table.
Scaffolding an API controller

In the Solution Explorer window, right-click on the Controllers folder and choose Add | New Item.... Choose Web API Controller Class, enter the name as ShippersController, and then click on Add:

API controllers do not have views. Instead, they use automatic content negotiation with the client to return XML, JSON, or X-WWW-FORMURLENCODED data formats.

In ShippersController class, add the following code:

```csharp
[Route("api/[controller]" )]
public class ShippersController : Controller
{

    private NorthwindContext db;
    public ShippersController(NorthwindContext injectedContext)
    {
        db = injectedContext;
    }

    // GET: api/shippers
    [HttpGet]
    public IEnumerable<Shippers> Get()
    {
        return db.Shippers.ToArray();
    }
}
```
If you have used older versions of ASP.NET Web API, then you know that in that technology, you could create C# methods that begin with any HTTP method (GET, POST, PUT, and so on), and the controller will automatically execute the correct one. In ASP.NET Core, this doesn’t happen anymore, because we are not inheriting from ApiController. So you must apply an attribute such as [HttpGet] to explicitly map HTTP methods to C# methods.

Press F5 to run the application.

In the address bar, enter api/shippers at the end of the URL and press Enter. You will see this JSON response:

![JSON response](image)

If you test with Chrome rather than Microsoft Edge, then you will get an XML response because Chrome prefers XML over JSON.

### Calling a Web API service from a UWP app

Now that we have a service that allows HTTP requests to be used to manage the Shippers table, we can create a client application to call it. The client will often be an HTML page that uses JavaScript to make the calls. However, since this book is about modern C# and .NET, we will build a UWP app.

In Visual Studio, click on File | Add | New Project.... In the Add New Project dialog, in the Installed Templates list, select Visual C#. In the center list, select Blank App (Universal Windows), type the name as Ch14_WinApp, and then click on OK.
In **Solution Explorer** window, right-click on the new project and choose **Manage NuGet Packages**. In the search box, enter **web api client** and press **Enter**. Click on **Install**:

![NuGet Package Manager](image)

On the **Project** menu, choose **Add Class** and name it **Shipper**:

```csharp
public class Shipper
{
    public int ShipperID { get; set; }
    public string CompanyName { get; set; }
    public string Phone { get; set; }
}
```

Open **MainPage.xaml** file and add the following XAML inside the existing Grid element:

```xml
<StackPanel Padding="6">
    <TextBlock FontSize="24">Shippers</TextBlock>
    <GridView ItemsSource="{Binding}"
        ItemTemplate="{Binding ShipperID}"
    <Border Margin="6" Padding="10" CornerRadius="10"
        Background="LightCyan">
        <Grid/>
```

![MainPage.xaml](image)

Column="1" />
This defines a GridView control bound to the DataContext class that we will set when the page loads to the response coming back from the service.

In the Page element, add an event handler for Loading function:

```xml
<Page x:Class="Ch14_Page">
    <Property x:Name="Loading"/>
</Page>
```

In the MainPage.xaml.cs file, import the following namespace:

```xml
using System.Net.Http;
```

Add the following statements to the Page_Loaded method, and add the async keyword to the method's declaration:

```xml
var client = new HttpClient();
client.BaseAddress = new Uri("http://localhost:59468/");
HttpResponseMessage response = await client.GetAsync("api/shippers");
DataContext = await response.Content.ReadAsAsync<Shipper[]>();
```

Make sure you use the same random port number that Visual Studio allocated to your ASP.NET Core application. It is unlikely to be 59468!

On the Build menu, choose Deploy Ch14_WinApp. In the Solution Explorer window, right-click on the solution and choose Properties.

Select Multiple startup projects. Set the action for Ch14_WebApp to Start without debugging. Set the action for Ch14_WinApp to Start.
On the Debug menu, choose Start Debugging or press F5. You will see that the UWP app called the service, deserialized the JSON data, and bound it to the list box:
Practicing and exploring
Test your knowledge and understanding by answering some questions, get some hands-on practice, and explore this chapter's topics with deeper research.

Exercise 14.1 – test your knowledge
Answer the following questions:

1. What is the difference between a web browser and a web server?
2. What is the difference between a URI, a URL, and a URN?
3. What are the four most common HTTP methods?
4. What does it mean when a web server responds with status code 302?
5. What are the responsibilities of a route?
6. What are the responsibilities of a controller?
7. What are the responsibilities of a model?
8. What are the responsibilities of a view?
9. How does ASP.NET distinguish a request for MVC from a request for Web API?
10. What data formats does Web API support by default?

Exercise 14.2 – practice building a data-driven web application
Create an ASP.NET Core web application that connects to the Northwind sample database and enables the user to see a list of customers grouped by country. When the user clicks on a customer record, they then see a page showing the full contact details of that customer and a list of their orders.

Exercise 14.3 – explore topics
Use the following links to read more details about this chapter's topics:

- Learn about ASP.NET Web Forms: http://www.asp.net/web-forms
- Learn about ASP.NET MVC: http://www.asp.net/mvc
- Learn about ASP.NET Web API: http://www.asp.net/web-api
Summary
In this chapter, you learned how to build an ASP.NET Core MVC web application, and an ASP.NET Core Web API service that manages data using Entity Framework Core. You also learned how to consume a REST/HTTP service using HttpClient.

In the next chapter, you will learn how to build an ASP.NET Core web application that can be hosted cross-platform on Windows, Linux, or Mac OS X.
Where to buy this book

You can buy C# 6 and .NET Core 1.0 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.