CHAPTER 5

Working with Razor

In an ASP.NET Core MVC application, a component called the *view engine* is used to produce the content sent to clients. The default view engine is called Razor, and it processes annotated HTML files for instructions that insert dynamic content into the output sent to the browser.

In this chapter, I give you a quick tour of the Razor syntax so you can recognize Razor expressions when you see them. I am not going to supply an exhaustive Razor reference in this chapter; think of this more as a crash course in the syntax. I explore Razor in depth as I continue through the book, within the context of other MVC features. Table 5-1 puts Razor in context.

Table 5-1. *Putting Razor in Context*

Question	Answer
What is it?	Razor is the view engine responsible for incorporating data into HTML documents.
Why is it useful?	The ability to dynamically generate content is essential to being able to write a web application. Razor provides features that make it easy to work with the rest of the ASP.NET Core MVC using C# statements.
How is it used?	Razor expressions are added to static HTML in view files. The expressions are evaluated to generate responses to client requests.
Are there any pitfalls or limitations?	Razor expressions can contain almost any C# statement, and it can be hard to decide whether logic should belong in the view or in the controller, which can erode the separation of concerns that is central to the MVC pattern.
Are there any alternatives?	You can write your own view engine, as I explain in Chapter 21. There are some third-party view engines available, but they tend to be useful for niche situations and don't attract long-term support.

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_5

Table 5-2 summarizes the chapter.

Table 5-2. *Chapter Summary*

Problem	Solution	Listing
Access the view model	Use an `@model` expression to define the model type and `@Model` expressions to access the model object	5, 14, 17
Use type names without qualifying them with namespaces	Create a view imports file	6, 7
Define content that will be used by multiple views	Use a layout	8–10
Specify a default layout	Use a view start file	11–13
Pass data from the controller to the view outside of the view model	Use the view bag	15–16
Generate content selectively	Use Razor conditional expressions	18, 19
Generate content for each item in an array or collection	Use a Razor `foreach` expression	20–21

Preparing the Example Project

To demonstrate how Razor works, I created an ASP.NET Core Web Application (.NET Core) project called Razor using the Empty template, just as in the previous chapter. I enabled the MVC framework by make the changes shown in Listing 5-1 to the Startup class.

Listing 5-1. Enabling MVC in the Startup.cs File in the Razor Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Razor {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }
```

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

Defining the Model

Next, I created a `Models` folder and added to it a class file called `Product.cs`, which I used to define the simple model class shown in Listing 5-2.

Listing 5-2. The Contents of the Product.cs File in the Models Folder

```
namespace Razor.Models {

    public class Product {

        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Creating the Controller

The default configuration that I set up in the `Startup.cs` file follows the MVC convention of sending requests to a controller called `Home` by default. I created a `Controllers` folder and added to it a class file called `HomeController.cs`, which I used to define the simple controller shown in Listing 5-3.

Listing 5-3. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            Product myProduct = new Product {
                ProductID = 1,
                Name = "Kayak",
                Description = "A boat for one person",
                Category = "Watersports",
                Price = 275M
            };

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

The controller defines an action method called Index, in which I create and populate the properties of a Product object. I pass the Product to the View method so that it is used as the model when the view is rendered. I do not specify the name of a view file when I call the View method, so the default view for the action method will be used.

Creating the View

To create the default view for the Index action method, I created a Views/Home folder and added to it an MVC View Page file called Index.cshtml, to which I added the content shown in Listing 5-4.

Listing 5-4. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    Content will go here
</body>
</html>
```

In the sections that follow, I go through the different parts of a Razor view and demonstrate some of the different things you can do with one. When learning about Razor, it is helpful to bear in mind that views exist to express one or more parts of the model to the user—and that means generating HTML that displays data that is retrieved from one or more objects. If you remember that I am always trying to build an HTML page that can be sent to the client, then everything that Razor does begins to make sense. If you run the application, you will see the simple output shown in Figure 5-1.

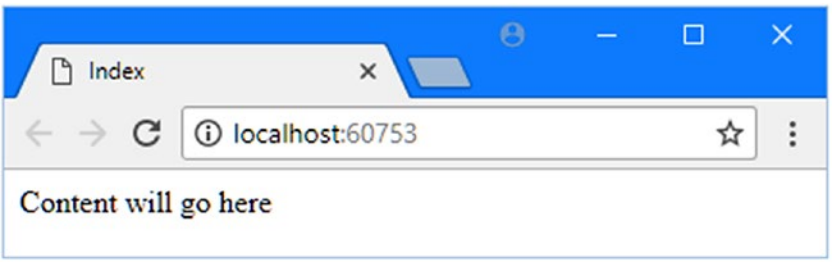

Figure 5-1. *Running the example application*

Working with the Model Object

Let's start with the first line in the `Index.cshtml` view file:

```
...
@model Razor.Models.Product
...
```

Razor expressions start with the `@` character. In this case, the `@model` expression declares the type of the model object that I will pass to the view from the action method. This allows me to refer to the methods, fields, and properties of the view model object through `@Model`, as shown in Listing 5-5, which displays a simple addition to the `Index` view.

Listing 5-5. Referring to a View Model Object Property in the Index.cshtml File in the Views/Home Folder

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    @Model.Name
</body>
</html>
```

■ **Note** Notice that I declare the view model object type using `@model` (a lowercase `m`) and access the `Name` property using `@Model` (an uppercase `M`). This is slightly confusing when you start working with Razor, but it quickly becomes second nature.

If you run the application, you will see the output shown in Figure 5-2.

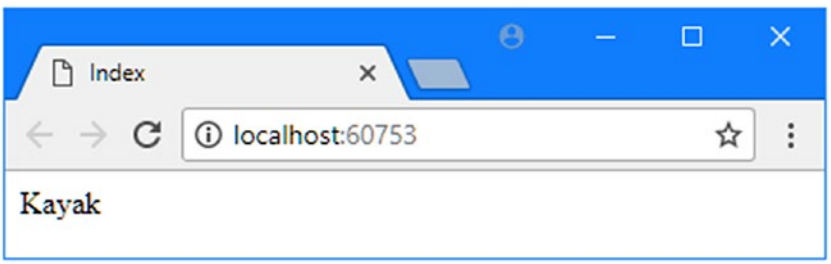

Figure 5-2. *The effect of reading a property value in the view*

A view that uses the `@model` expression to specify a type is known as a *strongly typed view*. Visual Studio is able to use the `@model` expression to pop up suggestions of member names when you type `@Model` followed by a period, as shown in Figure 5-3.

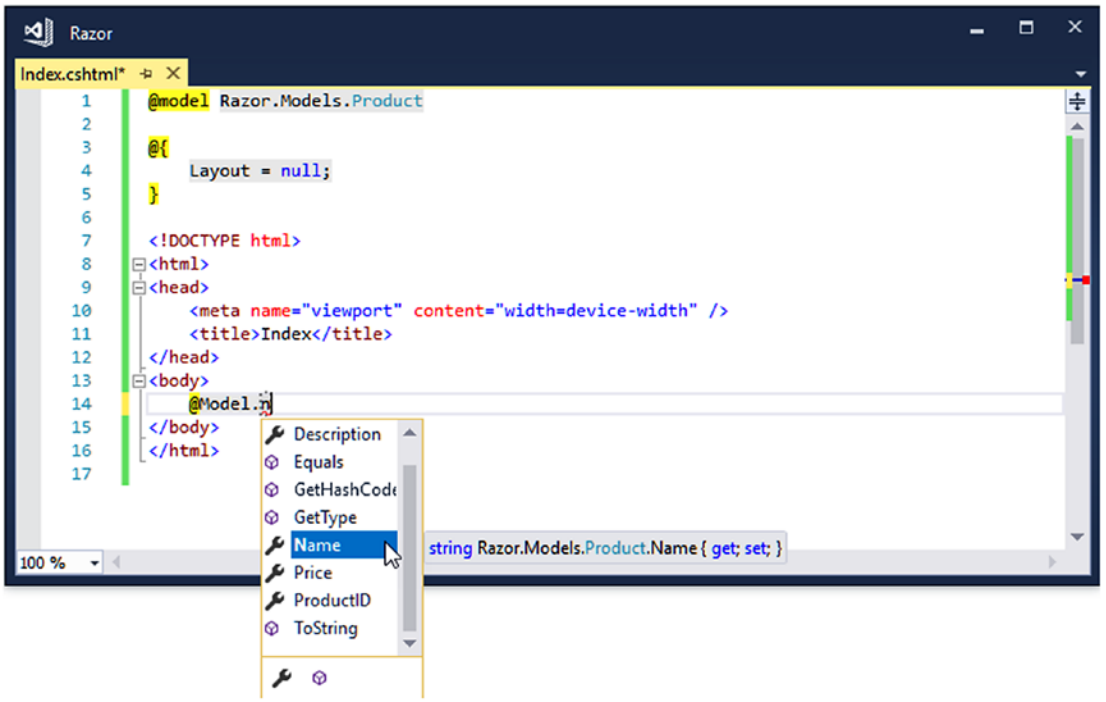

Figure 5-3. *Visual Studio offering suggestions for member names based on the @Model expression*

The Visual Studio suggestions for member names help avoid errors in Razor views. You can ignore the suggestions if you prefer, and Visual Studio will highlight problems with member names so that you make corrections, just as it does with regular C# class files. You can see an example of problem highlighting in Figure 5-4, where I have tried to reference `@Model.NotARealProperty`. Visual Studio has realized that the `Product` class I specified at the model type does not have such a property and has highlighted an error in the editor.

Figure 5-4. *Visual Studio reporting a problem with an @Model expression*

Using View Imports

When I defined the model object at the start of the `Index.cshtml` file, I had to include the namespace that contains the model class, like this:

```
...
@model Razor.Models.Product
...
```

By default, all types that are referenced in a strongly typed Razor view must be qualified with a namespace. This isn't a big deal when the only type reference is for the model object, but it can make a view more difficult to read when writing more complex Razor expressions such as the ones I describe later in this chapter.

You can specify a set of namespaces that should be searched for types by adding a *view imports* file to the project. The view imports file is placed in the `Views` folder and is named `_ViewImports.cshtml`.

■ **Note** Files in the `Views` folder whose names begin with an underscore (the _ character) are not returned to the user, which allows the file name to differentiate between views that you want to render and the files that support them. View imports files and layouts (which I describe shortly) are prefixed with an underscore.

To create the view imports file, right-click the `Views` folder in the Solution Explorer, select Add ➤ New Item from the pop-up menu, and select the MVC View Imports Page template from the ASP.NET Core ➤ Web category, as shown in Figure 5-5.

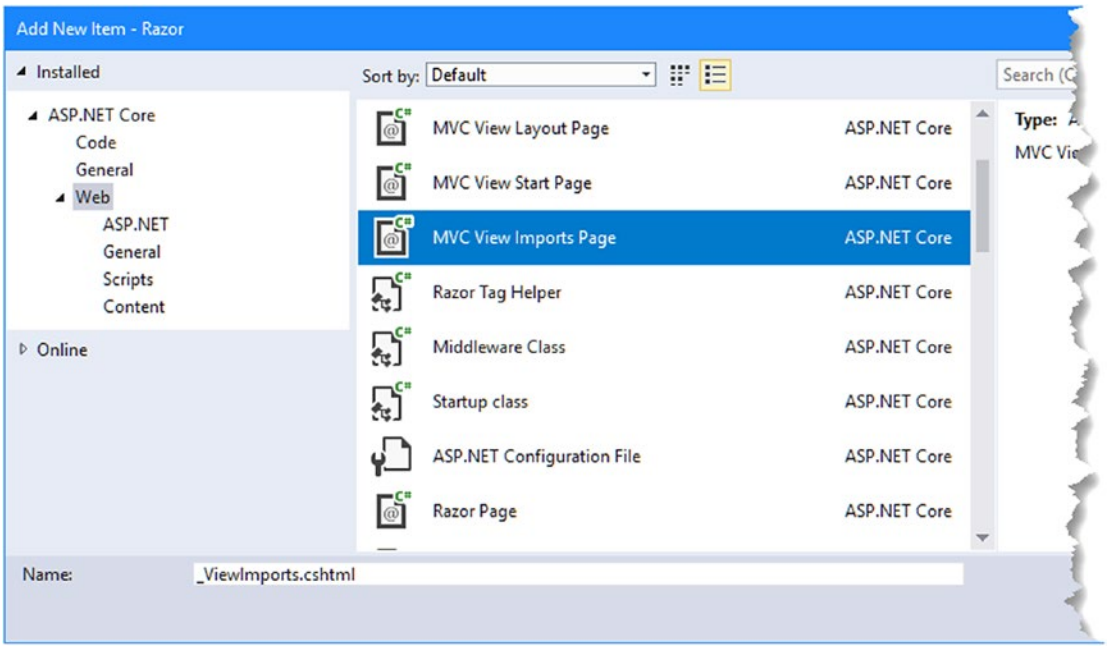

Figure 5-5. *Creating a view imports file*

Visual Studio will automatically set the name of the file to _ViewImports.cshtml, and clicking the Add button will create the file, which will be empty. Add the expression shown in Listing 5-6.

Listing 5-6. The Content of the _ViewImports.cshtml File in the Views Folder

```
@using Razor.Models
```

The namespaces that should be searched for classes used in Razor views are specified using the @using expression, followed by the namespace. In Listing 5-6, I have added an entry for the Razor.Models namespace that contains the model class in the example application.

Now that the Razor.Models namespace is included in the view imports file, I can remove the namespace from the Index.cshtml file, as shown in Listing 5-7.

Listing 5-7. Omitting the Model Namespace in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    @Model.Name
</body>
</html>
```

■ **Tip** You can also add an @using expression to individual view files, which allows types to be used without namespaces in a single view.

Working with Layouts

There is another important Razor expression in the Index.cshtml view file:

```
...
@{
    Layout = null;
}
...
```

This is an example of a Razor *code block*, which allows me to include C# statements in a view. The code block is opened with @{ and closed with }, and the statements it contains are evaluated when the view is rendered.

This code block sets the value of the `Layout` property to `null`. Razor views are compiled into C# classes in an MVC application, and the base class that is used defines the `Layout` property. I'll show you how this all works in Chapter 21, but the effect of setting the `Layout` property to `null` is to tell MVC that the view is self-contained and will render all of the content required for the client.

Self-contained views are fine for simple example apps, but a real project can have dozens of views, and some views will have shared content. Duplicating shared content in views becomes hard to manage, especially when you need to make a change and have to track down all of the views that need to be altered.

A better approach is to use a Razor layout, which is a template that contains common content and that can be applied to one or more views. When you make a change to a layout, the change will automatically affect all the views that use it.

Creating the Layout

Layouts are typically shared by views used by multiple controllers and are stored in a folder called `Views/Shared`, which is one of the locations that Razor looks in when it tries to find a file. To create a layout, create the `Views/Shared` folder, right-click it, and select Add ➤ New Item from the pop-up menu. Select the MVC View Layout Page template from the ASP.NET category and set the file name to `_BasicLayout.cshtml`, as shown in Figure 5-6. Click the Add button to create the file. (Like view import files, the names of layout files begin with an underscore.)

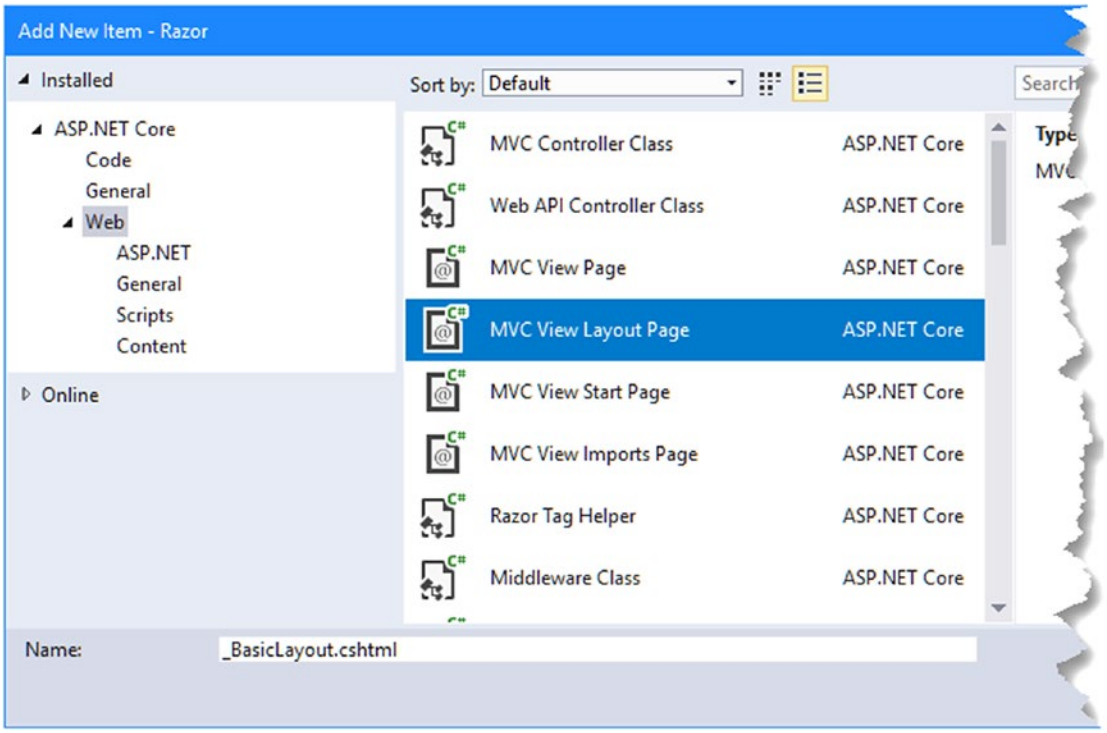

Figure 5-6. *Creating a layout*

Listing 5-8 shows the initial contents of the `_BasicLayout.cshtml` file, added by Visual Studio when it creates the file.

Listing 5-8. The Initial Contents of the _BasicLayout.cshtml File in the Views/Shared Folder

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

Layouts are a specialized form of view, and there are two `@` expressions in the listing. The call to the `@RenderBody` method inserts the contents of the view specified by the action method into the layout markup, like this:

```
...
<div>
    @RenderBody()
</div>
...
```

The other Razor expression in the layout looks for a property called `ViewBag.Title` in order to set the contents of the `title` element, like this:

```
...
<title>@ViewBag.Title</title>
...
```

The `ViewBag` is a handy feature that allows data values to be passed around an application and, in this case, between a view and its layout. You will see how this works when I apply the layout to a view.

The HTML elements in a layout will be applied to any view that uses it, providing a template for defining common content. In Listing 5-9, I have added some simple markup to the layout so that its template effect will be obvious.

Listing 5-9. Adding Content to the _BasicLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        #mainDiv {
            padding: 20px;
            border: solid medium black;
            font-size: 20pt
        }
    </style>
```

```
</head>
<body>
    <h1>Product Information</h1>
    <div id="mainDiv">
        @RenderBody()
    </div>
</body>
</html>
```

I have added a header element as well as some CSS to style the contents of the `div` element that contains the `@RenderBody` expression, just to make it clear what content comes from the layout and what comes from the view.

Applying a Layout

To apply the layout to the view, I need to set the value of the `Layout` property and remove the HTML that will now be provided by the layout, such as the `html`, `head`, and `body` elements, as shown in Listing 5-10.

Listing 5-10. Applying a Layout in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    Layout = "_BasicLayout";
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

The `Layout` property specifies the name of the layout file that will be used for the view, without the `cshtml` file extension. Razor will look for the specified layout file in the `/Views/Home` and `Views/Shared` folders.

I also set the `ViewBag.Title` property in the listing. This will be used by the layout to set the contents of the `title` element when the view is rendered.

The transformation of the view is dramatic, even for such a simple application. The layout contains all the structure required for any HTML response, which leaves the view to focus on just the dynamic content that presents the data to the user. When MVC processes the `Index.cshtml` file, it applies the layout to create a unified HTML response, as shown in Figure 5-7.

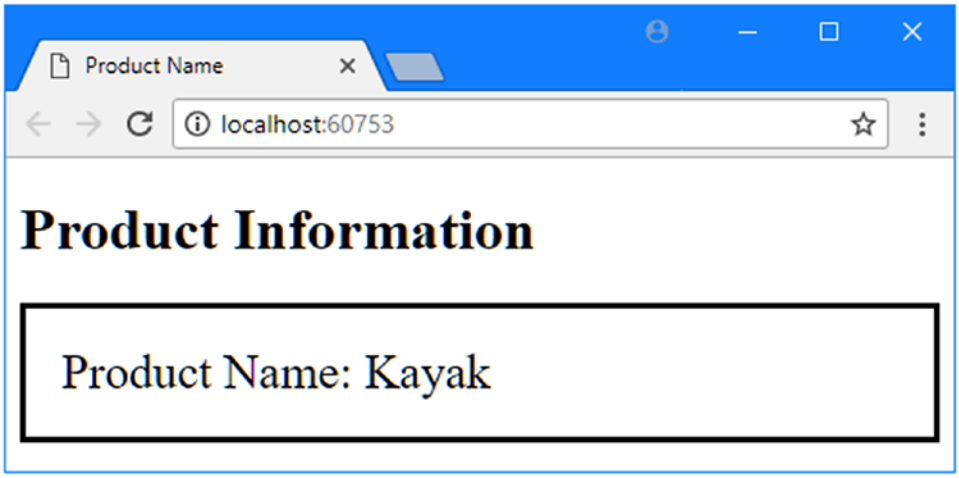

Figure 5-7. *The effect of applying a layout to a view*

Using a View Start File

I still have a tiny wrinkle to sort out, which is that I have to specify the layout file I want in every view. If I need to rename the layout file, I am going to have to find every view that refers to it and make a change, which will be an error-prone process and counter to the general theme of easy maintenance that runs through MVC development.

I can resolve this by using a *view start file*. When it renders a view, MVC will look for a file called `_ViewStart.cshtml`. The contents of this file will be treated as though they were contained in the view file itself, and I can use this feature to automatically set a value for the `Layout` property.

To create a view start file, right-click the `Views` folder, select Add ➤ New Item from the pop-up menu, and choose the MVC View Start Page template from the ASP.NET category, as shown in Figure 5-8.

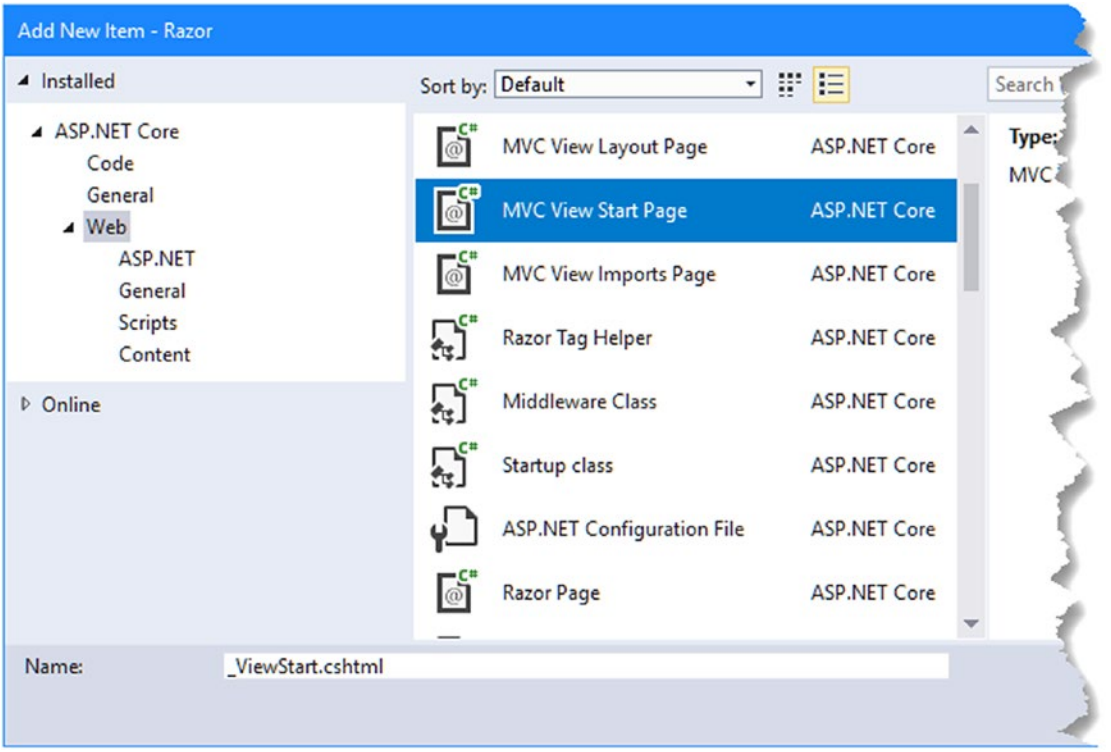

Figure 5-8. *Creating a view start file*

Visual Studio will set the name of the file to `_ViewStart.cshtml`, and clicking the Add button will create the file with the initial content shown in Listing 5-11.

Listing 5-11. The Initial Contents of the _ViewStart.cshtml File in the Views Folder

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

To apply my layout to all the views in the application, I changed the value assigned to the `Layout` property, as shown in Listing 5-12.

Listing 5-12. Applying a Default View in the _ViewStart.cshtml File in the Views Folder

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

Since the view start file contains a value for the `Layout` property, I can remove the corresponding expression from the `Index.cshtml` file, as shown in Listing 5-13.

Listing 5-13. Applying a View Start File in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

I do not have to specify that I want to use the view start file. MVC will locate the file and use its contents automatically. The values defined in the view file take precedence, which makes it easy to override the view start file.

You can also use multiple view start files to set defaults for different parts of the application. Razor looks for the closest view start file to the view that it being processed, which means you can override the default setting by adding a view start file to the `Views/Home` or `Views/Shared` folders, for example.

■ **Caution** It is important to understand the difference between omitting the `Layout` property from the view file and setting it to `null`. If your view is self-contained and you do not want to use a layout, then set the `Layout` property to `null`. If you omit the `Layout` property, then MVC will assume that you *do* want a layout and that it should use the value it finds in the view start file.

Using Razor Expressions

Now that I have shown you the basics of views and layouts, I am going to turn to the different kinds of expressions that Razor supports and how you can use them to create view content. In a good MVC application, there is a clear separation between the roles that the action method and view perform. The rules are simple; I have summarized them in Table 5-3.

Table 5-3. *The Roles Played by the Action Method and the View*

Component	Does Do	Doesn't Do
Action method	Passes a view model object to the view	Passes formatted data to the view
View	Uses the view model object to present content to the user	Changes any aspect of the view model object

I come back to this theme throughout this book. To get the best from MVC, you need to respect and enforce the separation between the different parts of the app. As you will see, you can do quite a lot with Razor, including using C# statements—but you should not use Razor to perform business logic or manipulate your domain model objects in any way. Listing 5-14 shows the addition of a new expression to the Index view.

Listing 5-14. Adding an Expression to the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
```

I could have formatted the value of the `Price` property in the action method and passed it to the view. It would have worked, but taking this approach undermines the benefit of the MVC pattern and reduces my ability to respond to changes in the future. As I said, I will return to this theme again, but you should remember that ASP.NET Core MVC does not enforce proper use of the MVC pattern and that you must remain aware of the effect of the design and coding decisions you make.

PROCESSING VERSUS FORMATTING DATA

It is important to differentiate between *processing* data and *formatting* it. Views *format* data, which is why I passed the `Product` object in the previous section to the view, rather than formatting the object's properties into a display string. Processing data—including selecting the data objects to display—is the responsibility of the controller, which will call on the model to get and modify the data it requires. It can sometimes be hard to figure out where the line between processing and formatting is, but as a rule of thumb, I recommend erring on the side of caution and pushing anything but the simplest of expressions out of the view and into the controller.

Inserting Data Values

The simplest thing you can do with a Razor expression is to insert a data value into the markup. The most common way to do this is with the `@Model` expression. The `Index` view already includes examples of this approach, like this:

```
...
<p>Product Name: @Model.Name</p>
...
```

You can also insert values using the `ViewBag` feature, which is the feature I used in the layout to set the content of the `title` element. The `ViewBag` can be used to pass data from the controller to the view, supplementing the model, as shown in Listing 5-15.

Listing 5-15. Using the View Bag in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {
```

```
        public ViewResult Index() {
            Product myProduct = new Product {
                ProductID = 1,
                Name = "Kayak",
                Description = "A boat for one person",
                Category = "Watersports",
                Price = 275M
            };

            ViewBag.StockLevel = 2;

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

The ViewBag property returns a dynamic object that can be used to define arbitrary properties. In the listing, I have defined a property called StockLevel and assigned a value of 2 to it. Since the ViewBag property is dynamic, I don't have to declare the property names in advance, but it does mean that Visual Studio is unable to provide autocomplete suggestions for view bag properties.

Knowing when to use the view bag and when the model should be extended is a matter of experience and personal preference. My personal style is to use the view bag only to give a view hints about how to render data and not to use it for data values that are displayed to the user. But that's just what works for me. If you do use the view bag for data you want to display to the user, then you access values using the @ViewBag expression, as shown in Listing 5-16.

Listing 5-16. Displaying a View Bag Value in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
<p>Stock Level: @ViewBag.StockLevel</p>
```

Figure 5-9 shows the result of the new data value.

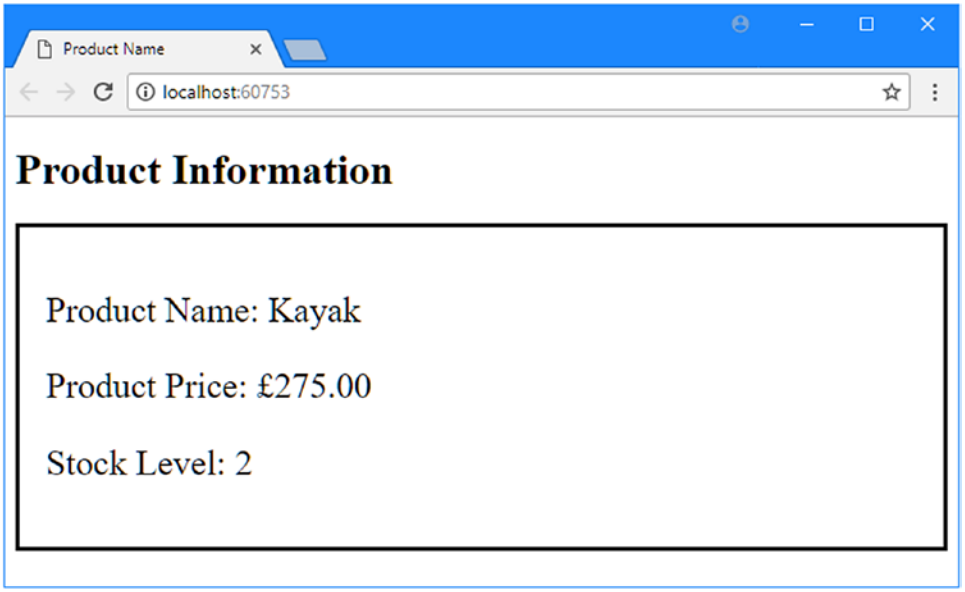

Figure 5-9. *Using Razor expressions to insert data values*

Setting Attribute Values

All the examples so far have set the content of elements, but you can also use Razor expressions to set the value of element *attributes*. Listing 5-17 shows the user of the `@Model` and `@ViewBag` expressions to set the contents of attributes on elements in the `Index` view.

Listing 5-17. Set Attribute Values in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel="@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level: @ViewBag.StockLevel</p>
</div>
```

I used the Razor expressions to set the value for some `data` attributes on a `div` element.

■ **Tip** Data attributes, which are attributes whose names are prefixed by `data-`, have been an informal way of creating custom attributes for many years and have been made part of the formal standard as part of HTML5. They are most often applied so that JavaScript code can locate specific elements or so that CSS styles can be more narrowly applied.

If you run the example application and look at the HTML source that is sent to the browser, you will see that Razor has set the values of the attributes, like this:

```
<div data-productid="1" data-stocklevel="2">
    <p>Product Name: Kayak</p>
    <p>Product Price: £275.00</p>
    <p>Stock Level: 2</p>
</div>
```

Using Conditional Statements

Razor is able to process conditional statements, which means that I can tailor the output from a view based on values in the view data. This kind of technique is at the heart of Razor and allows you to create complex and fluid layouts that are still reasonably simple to read and maintain. In Listing 5-18, I have updated the Index view so that it includes a conditional statement.

Listing 5-18. Using a Conditional Razor Statement in the Index.cshtml File in the Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel="@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level:
        @switch (ViewBag.StockLevel) {
            case 0:
                @:Out of Stock
                break;
            case 1:
            case 2:
            case 3:
                <b>Low Stock (@ViewBag.StockLevel)</b>
                break;
            default:
                @: @ViewBag.StockLevel in Stock
                break;
        }
    </p>
</div>
```

To start a conditional statement, you place an @ character in front of the C# conditional keyword, which is `switch` in this example. You terminate the code block with a close brace character (}) just as you would with a regular C# code block.

Inside the Razor code block, you can include HTML elements and data values into the view output just by defining the HTML and Razor expressions, like this:

```
...
<b>Low Stock (@ViewBag.StockLevel)</b>
...
```

I do not have to put the elements or expressions in quotes or denote them in any special way—the Razor engine will interpret these as output to be processed. However, if you want to insert literal text into the view when it is not contained in an HTML element, then you need to give Razor a helping hand and prefix the line like this:

```
...
@: Out of Stock
...
```

The `@:` characters prevent Razor from interpreting this as a C# statement, which is the default behavior when it encounters text. You can see the result of the conditional statement in Figure 5-10.

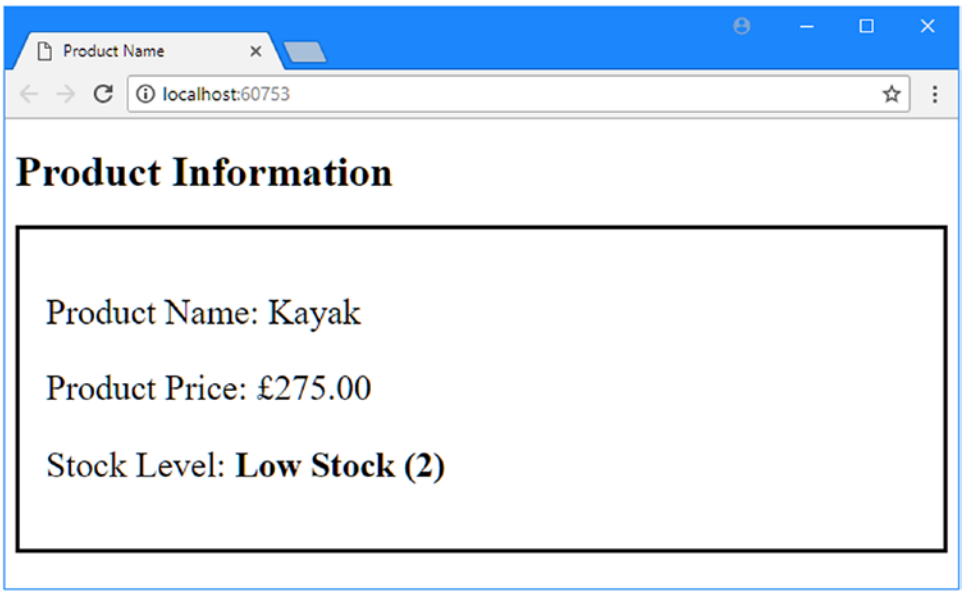

Figure 5-10. *Using a switch statement in a Razor view*

Conditional statements are important in Razor views because they allow content to be varied based on the data values that the view receives from the action method. As an additional demonstration, Listing 5-19 shows the addition of an `if` statement to the `Index.cshtml` view. As you might imagine, this is a commonly used conditional statement.

Listing 5-19. Using an if Statement in a Razor View in the Index.cshtml File in Views/Home Folder

```
@model Product

@{
    ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel="@ViewBag.StockLevel">
    <p>Product Name: @Model.Name</p>
    <p>Product Price: @($"{Model.Price:C2}")</p>
    <p>Stock Level:
        @if (ViewBag.StockLevel == 0) {
            @:Out of Stock
        } else if (ViewBag.StockLevel > 0 && ViewBag.StockLevel <= 3) {
            <b>Low Stock (@ViewBag.StockLevel)</b>
        } else {
            @: @ViewBag.StockLevel in Stock
        }
    </p>
</div>
```

This conditional statement produces the same results as the `switch` statement, but I wanted to demonstrate how you can mesh C# conditional statements with Razor views. I explain how this works in Chapter 21, when I describe views in depth.

Enumerating Arrays and Collections

When writing an MVC application, you will often want to enumerate the contents of an array or some other kind of collection of objects and generate content that details each one. To demonstrate how this is done, in Listing 5-20 I have revised the `Index` action in the `Home` controller to pass an array of `Product` objects to the view.

Listing 5-20. Using an Array in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            Product[] array = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };
            return View(array);
        }
    }
}
```

This action method creates a `Product[]` object that contains simple data values and passes them to the `View` method so that the data is rendered using the default view. In Listing 5-21, I have changed the model type for the `Index` view and used a `foreach` loop to enumerate the objects in the array.

■ **Tip** The Model term in Listing 5-21 doesn't need to be prefixed with an @ character because it is part of a larger C# expression. It can be difficult to figure out when an @ character is required and when it is not, but the Visual Studio IntelliSense for Razor files will tell you when you get it wrong by underlining errors.

Listing 5-21. Enumerating an Array in the Index.cshtml File in the Views/Home Folder

```
@model Product[]

@{
    ViewBag.Title = "Product Name";
}

<table>
    <thead>
        <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody>
        @foreach (Product p in Model) {
            <tr>
                <td>@p.Name</td>
                <td>@($"{p.Price:C2}")</td>
            </tr>
        }
    </tbody>
</table>
```

The `@foreach` statement enumerates the contents of the model array and generates a row in a table for each of them. You can see how I created a local variable called `p` in the `foreach` loop and then referred to its properties using the Razor expressions `@p.Name` and `@p.Price`. You can see the result in Figure 5-11.

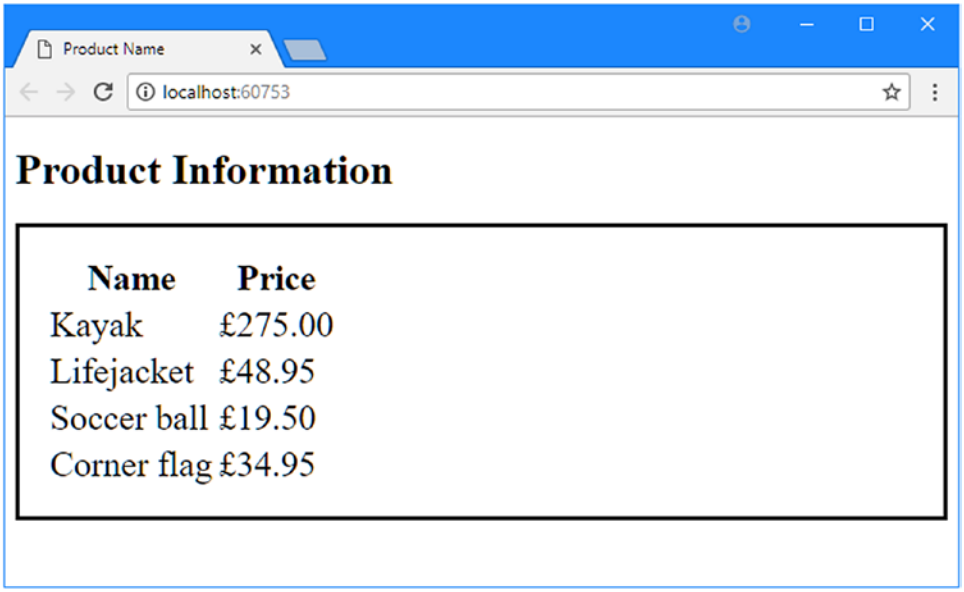

Figure 5-11. *Using Razor to enumerate an array*

Summary

In this chapter, I gave you an overview of the Razor view engine and how it can be used to generate HTML. I showed you how to refer to data passed from the controller via the view model object and the view bag and how Razor expressions can be used to tailor responses to the user based on data values. You will see many different examples of how Razor can be used in the rest of the book, and I describe how the MVC view mechanism works in detail in Chapter 21. In the next chapter, I introduce some of the features provided by Visual Studio for working with ASP.NET Core MVC projects.