

Building Websites Using the Model-View-Controller Pattern

This section is about building websites with a modern HTTP architecture on the server side using Microsoft ASP.NET Core MVC, including the startup configuration, authentication, authorization, routes, request and response pipeline, models, views, and controllers that make up an ASP.NET Core MVC project.

This section will cover the following topics:

- Setting up an ASP.NET Core MVC website
- Exploring an ASP.NET Core MVC website
- Customizing an ASP.NET Core MVC website
- Using other project templates

Setting up an ASP.NET Core MVC website

ASP.NET Core Razor Pages are great for simple websites. For more complex websites, it would be better to have a more formal structure to manage that complexity.

This is where the **Model-View-Controller (MVC)** design pattern is useful. It uses technologies similar to Razor Pages, but allows a cleaner separation between technical concerns, as shown in the following list:

- **Models:** Classes that represent the data entities and view models used in the website.
- **Views:** Razor files, that is, `.cshtml` files, that render data in view models into HTML web pages. Blazor uses the `.razor` file extension, but do not confuse them with Razor files!
- **Controllers:** Classes that execute code when an HTTP request arrives at the web server.
- The code usually creates a view model that may contain entity models and passes it to a view to generate an HTTP response to send back to the web browser or other client.

The best way to understand using the MVC design pattern for web development is to see a working example.

Creating and exploring an ASP.NET Core MVC website

You will use the `mvc` project template to create an ASP.NET Core MVC application with a database for authenticating and authorizing users:

1. In the folder named `PracticalApps`, create a folder named `NorthwindMvc`.
2. In Visual Studio Code, open the `PracticalApps` workspace and then add the `NorthwindMvc` folder to the workspace.
3. Navigate to **Terminal | New Terminal** and select `NorthwindMvc`.
4. In **TERMINAL**, create a new MVC website project with authentication stored in a SQLite database, as shown in the following command:

```
dotnet new mvc --auth Individual
```

You can enter the following command to see other options for this template:

```
dotnet new mvc --help
```

5. In **TERMINAL**, enter the command `dotnet run` to start the website.
6. Start Chrome and open **Developer tools**.
7. Navigate to `http://localhost:5000/` and note the following, as shown in the following screenshot:
 - Requests for HTTP are automatically redirected to HTTPS on port 5001.
 - The navigation menu at the top with links to **Home**, **Privacy**, **Register**, and **Login**. If the viewport width is 575 pixels or less, then the navigation collapses into a hamburger menu.
 - The title of the website, **NorthwindMvc**, shown in the header and footer.

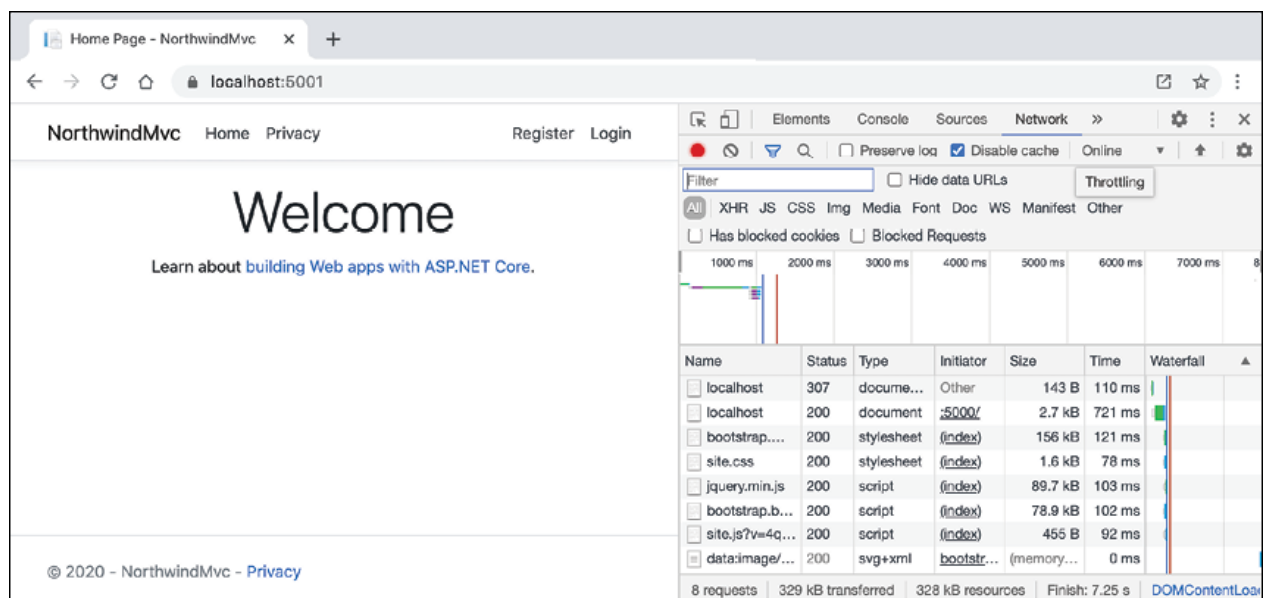


Figure 16.1: The Northwind MVC website homepage

8. Click **Register**, enter an email and password, and click the **Register** button.

By default, passwords must have at least one non-alphanumeric character, they must have at least one digit (0-9), and they must have at least one uppercase letter (A-Z). I use `Pa$$w0rd` in scenarios like this when I am just exploring.

The MVC project template follows best practice for **double-opt-in (DOI)**, meaning that after filling in an email and password to register, an email is sent to the email address, and the visitor must click a link in that email to confirm that they want to register.

We have not yet configured an email provider to send that email, so we must simulate that step.

9. Click the link with the text **Click here to confirm your account** and note that you are redirected to a **Confirm email** web page that you can customize.
10. In the top navigation menu, click **Login**, enter your email and password (note that there is an optional checkbox to remember you, and there are links if the visitor has forgotten their password or they want to register as a new visitor), and then click the **Log in** button.
11. Click your email in the top navigation menu to navigate to an account management page, and note that you can set a phone number, change your email address, change your password, enable two-factor authentication (if you add an authenticator app), and download and delete your personal data, as shown in the following screenshot:

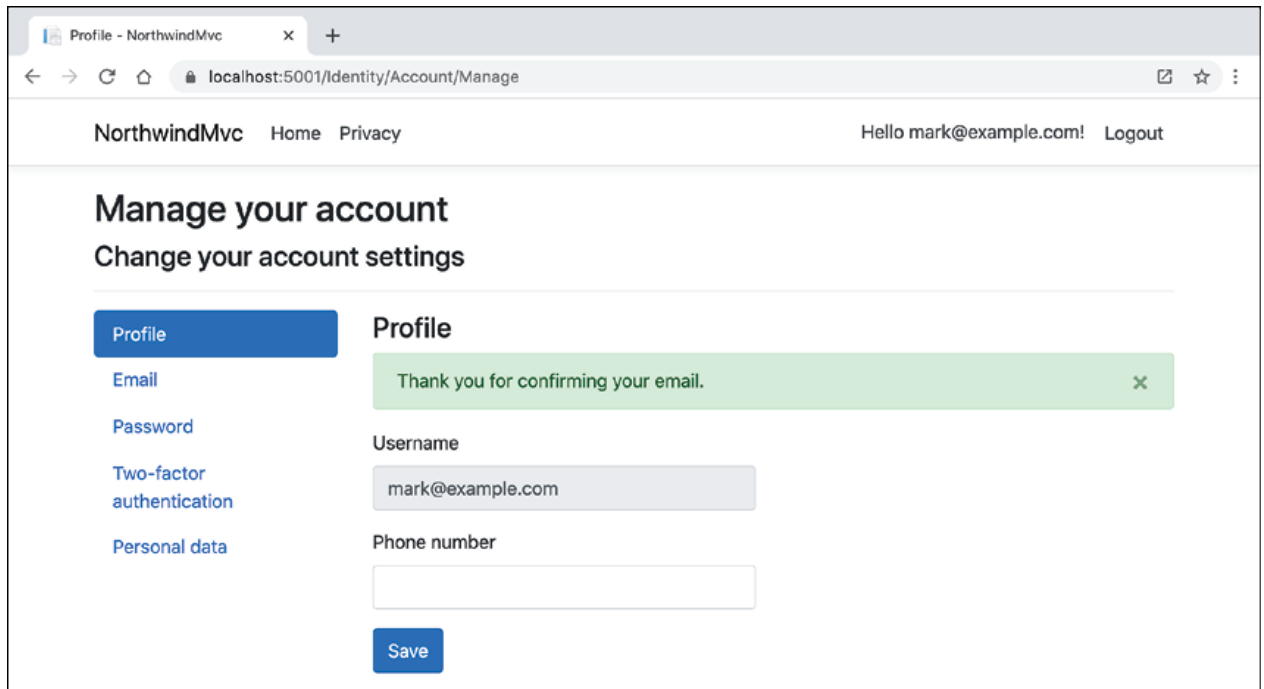


Figure 16.2: Website visitor Profile settings

More Information: Some of these built-in features of the basic MVC project template make it easier for your website to be compliant with modern privacy requirements, like the European Union's **General Data Protection Regulation (GDPR)** that became active in May 2018. You can read more at the following link: <https://docs.microsoft.com/en-us/aspnet/core/security/gdpr>

12. Close the browser.
13. In **TERMINAL**, press `ctrl + c` to stop the console application and shut down the Kestrel web server that is hosting your ASP.NET Core website.

More Information: You can read more about ASP.NET Core's support for authenticator apps at the following link: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity-enable-qrcodes>

Reviewing the ASP.NET Core MVC website

In Visual Studio Code, look at the **EXPLORER** pane, as shown in the following screenshot:

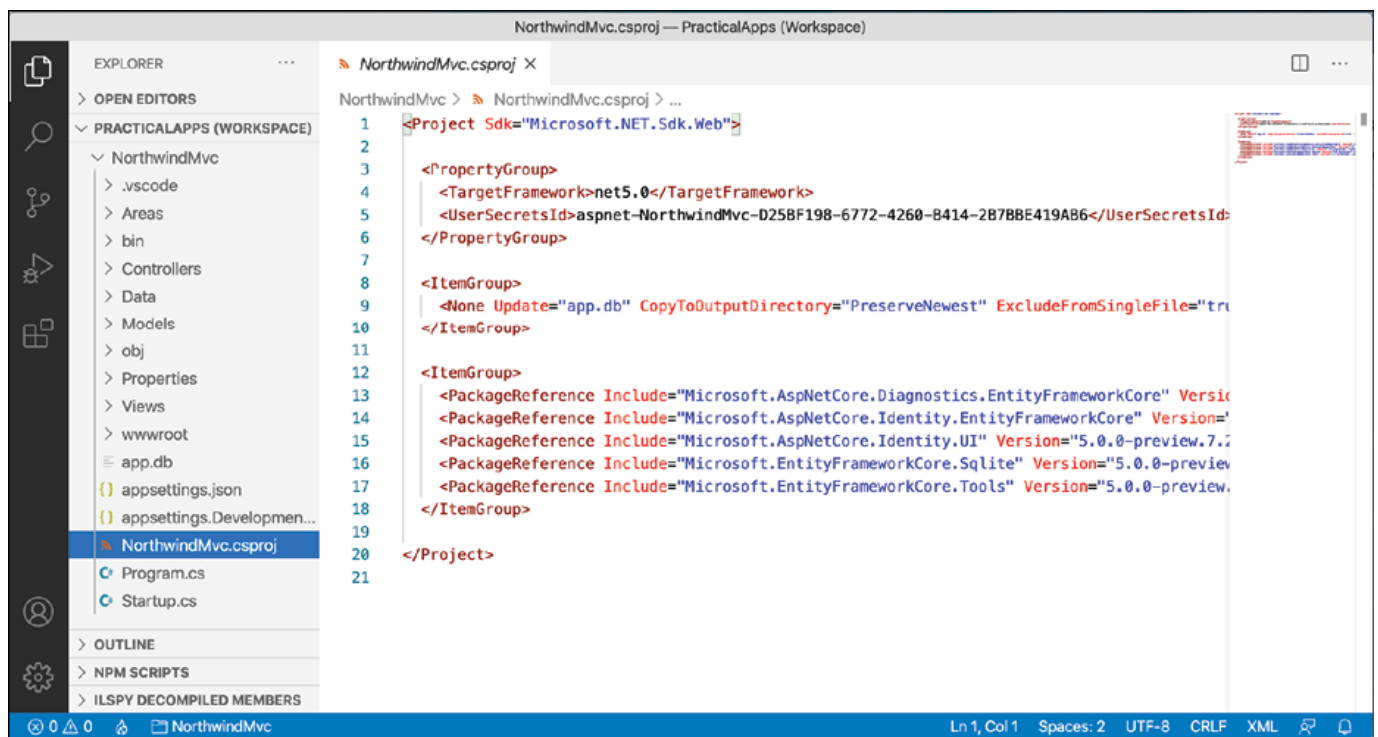


Figure 16.3: The EXPLORER pane showing the initial structure of an MVC project

We will look in more detail at some of these parts later, but for now, note the following:

- **Areas:** This folder contains nested folders and a file needed to integrate your website project with **ASP.NET Core Identity**, which is used for authentication.
- **bin, obj:** These folders contain the compiled assemblies for the project.
- **Controllers:** This folder contains C# classes that have methods (known as actions) that fetch a *model* and pass it to a *view*, for example, HomeController.cs.
- **Data:** This folder contains Entity Framework Core migration classes used by the **ASP.NET Core Identity** system to provide data storage for authentication and authorization, for example, ApplicationDbContext.cs.
- **Models:** This folder contains C# classes that represent all of the data gathered together by a controller and passed to a view, for example, ErrorViewModel.cs.
- **Properties:** This folder contains a configuration file for IIS or IIS Express on Windows and for launching the website during development named launchSettings.json. This file is only used on the local development machine and is not deployed to your production website.
- **Views:** This folder contains the .cshtml Razor files that combine HTML and C# code to dynamically generate HTML responses. The _ViewStart file sets the default layout and _ViewImports imports common namespaces used in all views like tag helpers:
 - **Home:** This subfolder contains Razor files for the home and privacy pages.
 - **Shared:** This subfolder contains Razor files for the shared layout, an error page, and two partial views for logging in and validation scripts.
- **wwwroot:** This folder contains static content used by the website, such as CSS for styling, libraries of JavaScript, JavaScript for this website project, and a favicon.ico file. You would also put images and other static file resources like documents in here.
- **app.db:** This is the SQLite database that stores registered visitors.
- **appsettings.json** and **appsettings.Development.json:** These files contain settings that your website can load at runtime, for example, the database connection string for the ASP.NET Identity system and logging levels.
- **NorthwindMvc.csproj:** This file contains project settings like use of the Web .NET SDK, an entry to ensure that the app.db file is copied to the website's output folder, and a list of NuGet packages that your project requires, including:
 - Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
 - Microsoft.AspNetCore.Identity.EntityFrameworkCore
 - Microsoft.AspNetCore.Identity.UI
 - Microsoft.EntityFrameworkCore.Sqlite
 - Microsoft.EntityFrameworkCore.Tools
- **Program.cs:** This file defines a class that contains the Main entry point that builds a pipeline for processing incoming HTTP requests and hosts the website using default options like configuring the Kestrel web server and loading appsettings. While building the host, it calls the UseStartup<T>() method to specify another class that performs additional configuration.
- **Startup.cs:** This file adds and configures services that your website needs, for example, ASP.NET Core Identity for authentication, SQLite for data storage, and so on, and routes for your application.

More Information: You can read more about default configuration of web hosts at the following link:
<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host>

Reviewing the ASP.NET Core Identity database

If you installed an SQLite tool such as **SQLiteStudio**, then you can open the database and see the tables that the ASP.NET Core Identity system uses to register users and roles, including the `AspNetUsers` table used to store the registered visitor, as shown in the following screenshot:

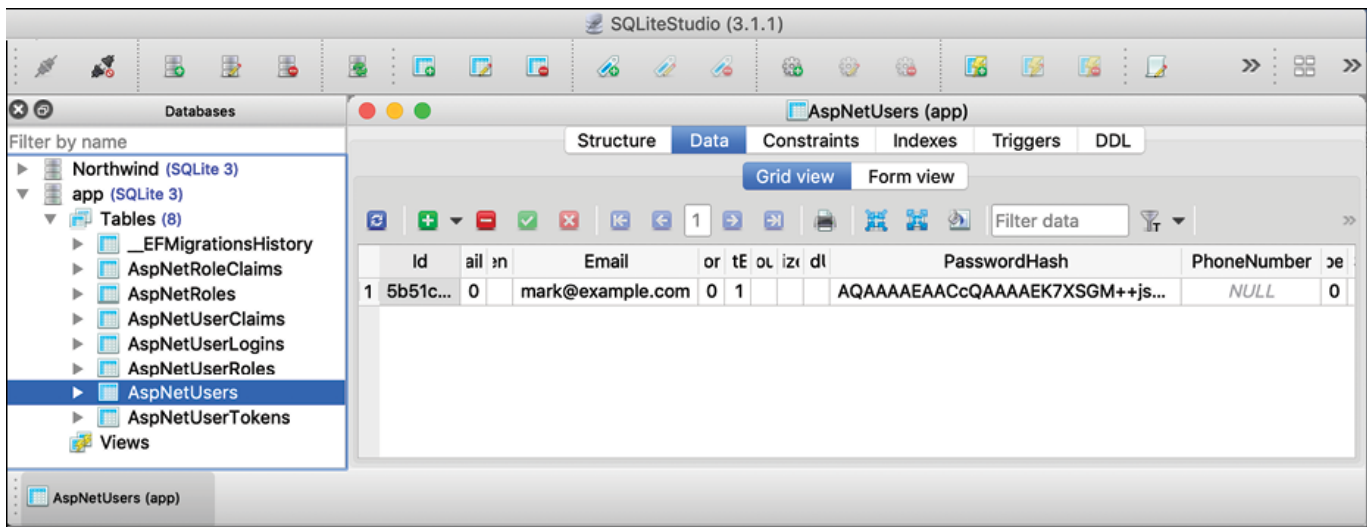


Figure 16.4: Viewing registered visitors in the app database

Customizing an ASP.NET Core MVC website

Now that you've reviewed the structure of a basic MVC website, you will customize it. You have already added code to retrieve entities from the `Northwind` database, so the next task is to output that information on the home page.

More Information: To find suitable images for the eight categories, I searched on a site that has free stock photos for commercial use with no attribution at the following link: <https://www.pexels.com/>

Defining a custom style

The home page will show a list of the 77 products in the `Northwind` database. To make efficient use of space, we want to show the list in three columns. To do this, we need to customize the stylesheet for the website:

1. In the `wwwroot\css` folder, open the `site.css` file.
2. At the bottom of the file, add a new style that will apply to an element with the newspaper ID, as shown in the following code:

```
#newspaper
{
    column-count: 3;
}
```

Setting up the category images

The `Northwind` database includes a table of categories, but they do not have images, and websites look better with some colorful pictures:

1. In the `wwwroot` folder, create a folder named `images`.
2. In the `images` folder, add eight image files from the downloads named `category1.jpeg`, `category2.jpeg`, and so on, up to `category8.jpeg`.

Understanding Razor syntax

Before we customize the home page view, let's review an example Razor file that has an initial Razor code block that instantiates an order with price and quantity and then outputs information about the order on the web page, as shown in the following markup:

```
@{
    var order = new Order
    {
        OrderID = 123,
        Product = "Sushi",
        Price = 8.49M,
        Quantity = 3
    };
}
<div>Your order for @order.Quantity of @order.Product has a total cost of $@order.Price * @order.Quantity</div>
```

The preceding Razor file would result in the following incorrect output:

```
Your order for 3 of Sushi has a total cost of $8.49 * 3
```

Although Razor markup can include the value of any single property using the `@object.property` syntax, you should wrap expressions in parentheses, as shown in the following markup:

```
<div>Your order for @order.Quantity of @order.Product has a total cost of $@(order.Price * @order.Quantity)</div>
```

The preceding Razor expression results in the following correct output:

```
Your order for 3 of Sushi has a total cost of $25.47
```

Defining a typed view

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

1. In the `Views\Home` folder, open `Index.cshtml`.
2. At the top of the file, add a statement to set the model type to use the `HomeIndexViewModel`, as shown in the following code:

```
@model NorthwindMvc.Models.HomeIndexViewModel
```

Now, whenever we type `Model` in this view, the Visual Studio Code C# extension will know the correct type for the model and will provide IntelliSense for it.

While entering code in a view, remember the following:

- To declare the type for the model, use `@model` (with lowercase `m`).
- To interact with the model instance, use `@Model` (with uppercase `M`).

Let's continue customizing the view for the home page.

3. In the initial Razor code block, add a statement to declare a `string` variable for the current item and replace the existing `<div>` element with the new markup to output categories in a carousel and products as an unordered list, as shown in the following markup:

```
@model NorthwindMvc.Models.HomeIndexViewModel
@{
    ViewData["Title"] = "Home Page";
    string currentItem = "";
}
<div class="carousel slide" data-ride="carousel"
    data-interval="3000" data-keyboard="true">
    <ol class="carousel-indicators">
        @for (int c = 0; c < Model.Categories.Count; c++)
        {
            if (c == 0)
            {
                currentItem = "active";
            }
            else
            {
                currentItem = "";
            }
            <li data-target="#categories" data-slide-to="@c"
                class="@currentItem"></li>
        }
    </ol>
    <div class="carousel-inner">
        @for (int c = 0; c < Model.Categories.Count; c++)
        {
            if (c == 0)
            {
                currentItem = "active";
            }
            else
            {
                currentItem = "";
            }
            <div class="carousel-item @currentItem">
                <img class="d-block w-100" src=
                    "~/images/category@(Model.Categories[c].CategoryID).jpeg"
                    alt="@Model.Categories[c].CategoryName" />
                <div class="carousel-caption d-none d-md-block">
                    <h2>@Model.Categories[c].CategoryName</h2>
                    <h3>@Model.Categories[c].Description</h3>
                    <p>
                        <a class="btn btn-primary"
                            href="/category/@Model.Categories[c].CategoryID">View</a>
                    </p>
                </div>
            </div>
        }
    </div>
    <a class="carousel-control-prev" href="#categories"
        role="button" data-slide="prev">
        <span class="carousel-control-prev-icon"
            aria-hidden="true"></span>
        <span class="sr-only">Previous</span>
    </a>
    <a class="carousel-control-next" href="#categories"
        role="button" data-slide="next">
        <span class="carousel-control-next-icon"
            aria-hidden="true"></span>
        <span class="sr-only">Next</span>
    </a>
</div>
<div class="row">
    <div class="col-md-12">
        <h1>Northwind</h1>
        <p class="lead">
            We have had @Model.VisitorCount visitors this month.
        </p>
        <h2>Products</h2>
        <div>
            <ul>
                @foreach (var item in @Model.Products)
                {
                    <li>
                        <a asp-controller="Home"
                            asp-action="ProductDetail"
                            asp-route-
                                @item.ProductName costs
                                @item.UnitPrice.ToString("C")
                            >
                        </a>
                    </li>
                }
            </ul>
        </div>
    </div>
</div>
```



```

    }
  </ul>
</div>
</div>
</div>

```

While reviewing the preceding Razor markup, note the following:

- It is easy to mix static HTML elements such as `` and `` with C# code to output the carousel of categories and the list of product names.
- The `<div>` element with the `id` attribute of `newspaper` will use the custom style that we defined earlier, so all of the content in that element will display in three columns.
- The `` element for each category uses parentheses around a Razor expression to ensure that the compiler does not include the `.jpeg` as part of the expression, as shown in the following markup:
`"~/images/category@(Model.Categories[c].CategoryID).jpeg"`
- The `<a>` elements for the product links use tag helpers to generate URL paths. Clicks on these hyperlinks will be handled by the `Home` controller and its `ProductDetail` action method. This action method does not exist yet, but you will add it later in this section. The ID of the product is passed as a route segment named `id`, as shown in the following URL path for Ipoh Coffee:

`https://localhost:5001/Home/ProductDetail/43`

Reviewing the customized home page

Let's see the result of our customized home page:

1. Start the website by entering the following command: `dotnet run`.
2. Start Chrome and navigate to `http://localhost:5000`.
3. Note the home page has a rotating carousel showing categories, a random number of visitors, and a list of products in three columns, as shown in the following screenshot:

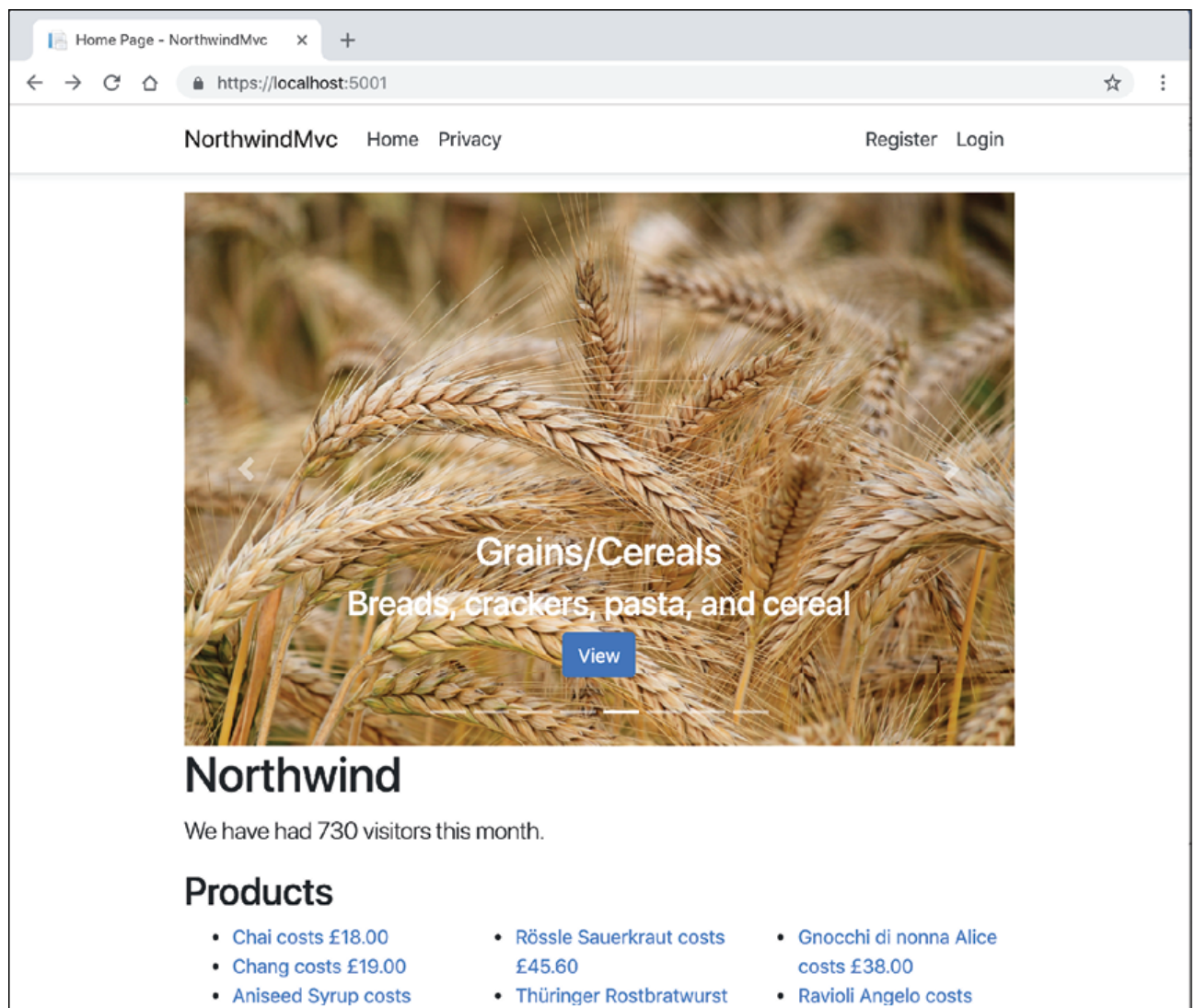


Figure 16.5: The updated Northwind MVC website homepage

At the moment, clicking on any of the categories or product links gives 404 Not Found errors, so let's see how we can pass parameters so that we can see the details of a product or category.

4. Close Chrome.
5. Stop the website by pressing `Ctrl + C` in **TERMINAL**.

Passing parameters using a route value

One way to pass a simple parameter is to use the `id` segment defined in the default route:

1. In the `HomeController` class, add an action method named `ProductDetail`, as shown in the following code:

```
public IActionResult ProductDetail(int? id)
{
    if (!id.HasValue)
    {
        return NotFound("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 NotFound($"Product with ID of {id} not found.");
    }
    return View(model); // pass model to view and then return result
}
```

Note the following:

- This method uses a feature of ASP.NET Core called **model binding** to automatically match the `id` passed in the route to the parameter named `id` in the method.
- Inside the method, we check to see whether `id` does not have a value, and if so, we call the `NotFound` method to return a 404 status code with a custom message explaining the correct URL path format.
- Otherwise, we can connect to the database and try to retrieve a product using the `id` variable.
- If we find a product, we pass it to a view; otherwise, we call the `NotFound` method to return a 404 status code and a custom message explaining that a product with that ID was not found in the database.

Remember that if the view is named to match the action method and is placed in a folder that matches the controller name or a shared folder, then ASP.NET Core MVC's conventions will find it automatically.

2. Inside the `Views/Home` folder, add a new file named `ProductDetail.cshtml`.
3. Modify the contents, as shown in the following markup:

```
@model Packt.Shared.Product
@{
    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.ToString("C")</dd>
        <dt>Units In Stock</dt>
        <dd>@Model.UnitsInStock</dd>
    </dl>
</div>
```

4. Start the website by entering the following command: `dotnet run`.
5. Start Chrome and navigate to `http://localhost:5000`.
6. When the home page appears with the list of products, click on one of them, for example, the second product, **Chang**.
7. Note the URL path in the browser's address bar, the page title shown in the browser tab, and the product details page, as shown in the following screenshot:

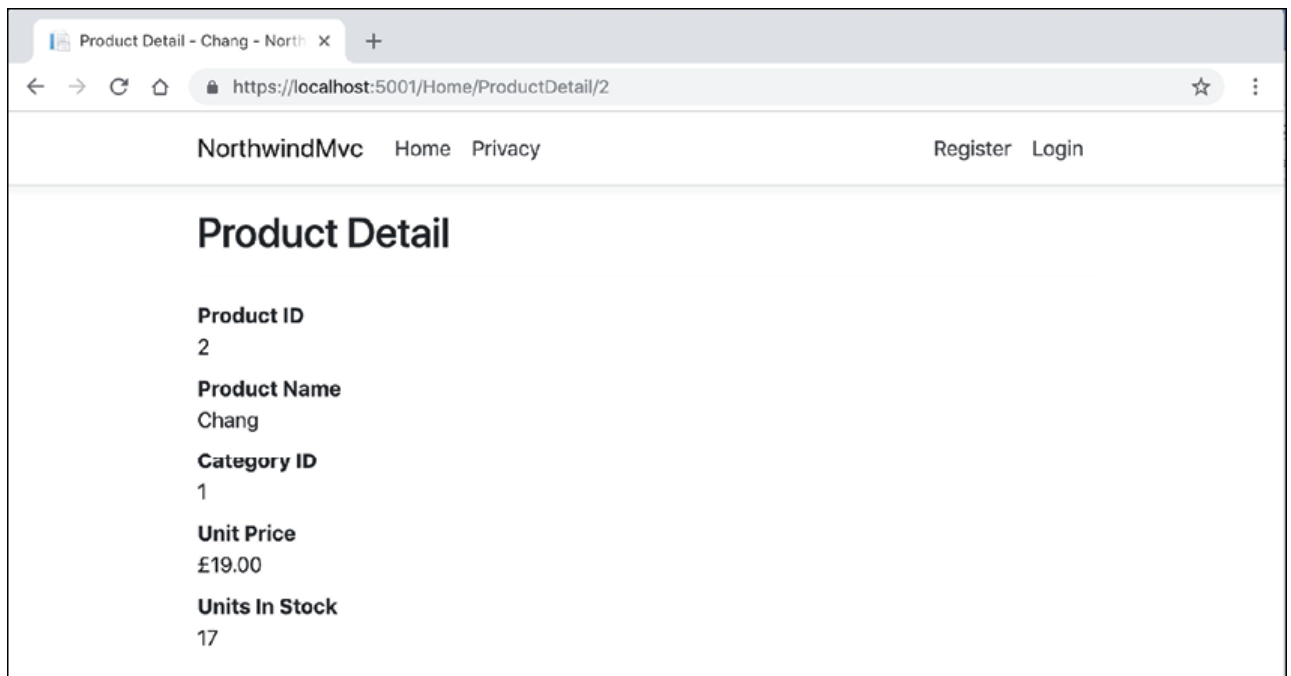


Figure 16.6: A product detail page

8. Toggle on the developer tools pane.
9. Edit the URL in the address box of Chrome to request a product ID that does not exist, like 99, and note the 404 Not Found status code and custom error response, as shown in the following screenshot:

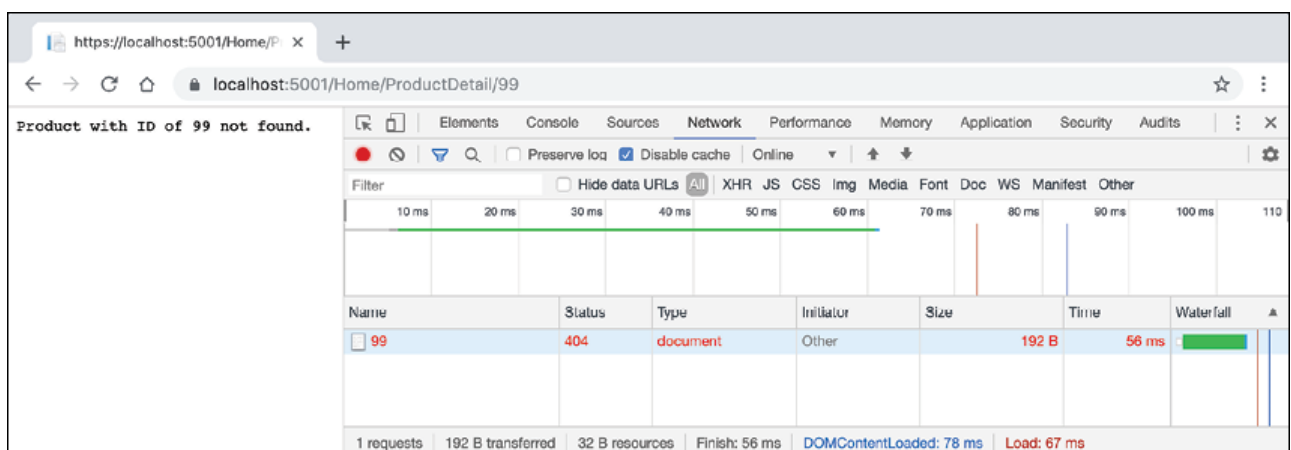


Figure 16.7: Requesting a product ID that doesn't exist

Understanding model binders

Model binders are very powerful, and the default one does a lot for you. After the default route identifies a controller class to instantiate and an action method to call, if that method has parameters, then those parameters need to have values set.

Model binders do this by looking for parameter values passed in the HTTP request as any of the following types of parameters:

- **Route parameter**, like `id` as we did in the previous section, as shown in the following URL path: `/Home/ProductDetail/2`
- **Query string parameter**, as shown in the following URL path: `/Home/ProductDetail?id=2`
- **Form parameter**, as shown in the following markup:

```
<form action="post" action="/Home/ProductDetail">
  <input type="text" name="id" />
  <input type="submit" />
</form>
```

Model binders can populate almost any type:

- Simple types, like `int`, `string`, `DateTime`, and `bool`.
- Complex types defined by `class` or `struct`.
- Collections types, like arrays and lists.

Let's create a somewhat artificial example to illustrate what can be achieved using the default model binder:

1. In the `Models` folder, add a new file named `Thing.cs`.
2. Modify the contents to define a class with two properties for a nullable number named `ID` and a string named `color`, as shown in the following code:

```
namespace NorthwindMvc.Models
{
    public class Thing
    {
        public int? ID { get; set; }
        public string Color { get; set; }
    }
}
```

3. Open `HomeController.cs` and add two new action methods, one to show a page with a form and one to display it with a parameter using your new model type, as shown in the following code:

```
public IActionResult ModelBinding()
{
    return View(); // the page with a form to submit
}
public IActionResult ModelBinding(Thing thing)
{
    return View(thing); // show the model bound thing
}
```

4. In the `Views\Home` folder, add a new file named `ModelBinding.cshtml`.
5. Modify its contents, as shown in the following markup:

```
@model NorthwindMvc.Models.Thing
@{
    ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
    Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding?id=3">
    <input name="color" value="Red" />
    <input type="submit" />
</form>
@if (Model != null)
{
    <h2>Submitted Thing</h2>
    <hr />
    <div>
        <dl class="dl-horizontal">
            <dt>Model.ID</dt>
            <dd>@Model.ID</dd>
            <dt>Model.Color</dt>
            <dd>@Model.Color</dd>
        </dl>
    </div>
}
```

6. Start the website, start Chrome, and navigate to `https://localhost:5001/home/modelbinding`.
7. Note the unhandled exception about an ambiguous match, as shown in the following screenshot:

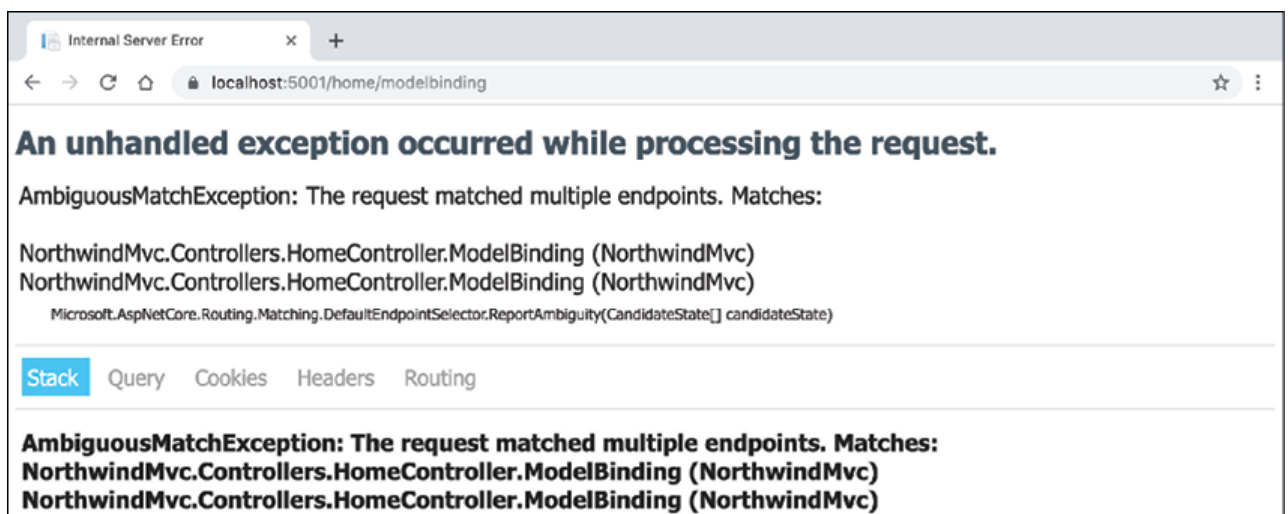


Figure 16.8: An unhandled exception error

Although the C# compiler can differentiate between the two methods by noting that the signatures are different, from HTTP's point of view, both methods are potential matches. We need an HTTP-specific way to disambiguate the action methods. We could do this by creating different names for the actions, or by specifying that one method should be used for a specific HTTP verb, like `GET`, `POST`, or `DELETE`.

8. Stop the website by pressing `Ctrl + C` in **TERMINAL**.

9. In `HomeController.cs`, decorate the second `ModelBinding` action method to indicate that it should be used for processing HTTP POST requests, that is, when a form is submitted, as shown highlighted in the following code:

```
[HttpPost]
public IActionResult ModelBinding(Thing thing)
```

10. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
11. Click the **Submit** button and note the value for the `ID` property is set from the query string parameter and the value for the `color` property is set from the form parameter, as shown in the following screenshot:

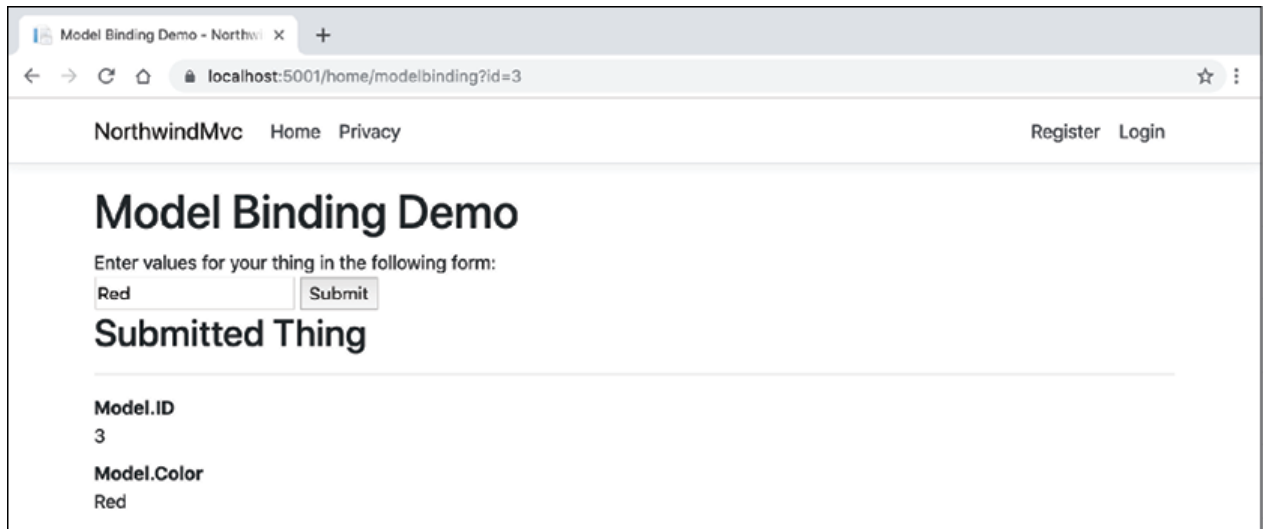


Figure 16.9: The Model Binding Demo page

12. Stop the website.
13. Modify the action for the form to pass the value 2 as a route parameter, as shown highlighted in the following markup:

```
<form method="POST" action="/home/modelbinding/2?id=3">
```

14. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
15. Click the **Submit** button and note the value for the `ID` property is set from the route parameter and the value for the `color` property is set from the form parameter.
16. Stop the website.
17. Modify the action for the form to pass the value 1 as a form parameter, as shown highlighted in the following markup:

```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input type="submit" />
</form>
```

18. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
19. Click the **Submit** button and note the values for the `ID` and `color` properties are both set from the form parameters.
20. If you have multiple parameters with the same name, then form parameters have the highest priority and query string parameters have the lowest priority for automatic model binding.

More Information: For advanced scenarios, you can create your own model binders by implementing the `IModelBinder` interface: <https://docs.microsoft.com/en-us/aspnet/core/mvc/advanced/custom-model-binding>

Validating the model

The process of model binding can cause errors, for example, data type conversions or validation errors if the model has been decorated with validation rules. What data has been bound and any binding or validation errors are stored in `ControllerBase.ModelState`.

Let's explore what we can do with model state by applying some validation rules to the bound model and then showing invalid data messages in the view:

1. In the `Models` folder, open `Thing.cs`.
2. Import the `System.ComponentModel.DataAnnotations` namespace.
3. Decorate the `ID` property with a validation attribute to limit the range of allowed numbers to 1 to 10, and one to ensure that the visitor supplies a color, as shown in the following highlighted code:

```
public class Thing
{
    [Range(1, 10)]
    public int? ID { get; set; }
    [Required]
```

```

    public string Color { get; set; }
}

```

4. In the `Models` folder, add a new file named `HomeModelBindingViewModel.cs`.
5. Modify its contents to define a class with two properties for the bound model and for any errors, as shown in the following code:

```

using System.Collections.Generic;
namespace NorthwindMvc.Models
{
    public class HomeModelBindingViewModel
    {
        public Thing Thing { get; set; }
        public bool HasErrors { get; set; }
        public IEnumerable<string> ValidationErrors { get; set; }
    }
}

```

6. In the `Controllers` folder, open `HomeController.cs`.
7. In the second `ModelBinding` method, comment out the previous statement that passed the thing to the view, and instead add statements to create an instance of the view model. Validate the model and store an array of error messages, and then pass the view model to the view, as shown in the following code:

```

public IActionResult ModelBinding(Thing thing)
{
    // return View(thing); // show the model bound thing
    var model = new HomeModelBindingViewModel
    {
        Thing = thing,
        HasErrors = !ModelState.IsValid,
        ValidationErrors = ModelState.Values
            .SelectMany(state => state.Errors)
            .Select(error => error.ErrorMessage)
    };
    return View(model);
}

```

8. In `Views\Home`, open `ModelBinding.cshtml`.
9. Modify the model type declaration to use the view model class, add a `<div>` to show any model validation errors, and change the output of the thing's properties because the view model has changed, as shown highlighted in the following markup:

```

@model NorthwindMvc.Models.HomeModelBindingViewModel
@{
    ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
    Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding/2?id=3">
    <input name="id" value="1" />
    <input name="color" value="Red" />
    <input type="submit" />
</form>
@if (Model != null)
{
    <h2>Submitted Thing</h2>
    <hr />
    <div>
        <dl class="dl-horizontal">
            <dt>Model.Thing.ID</dt>
            <dd>@Model.Thing.ID</dd>
            <dt>Model.Thing.Color</dt>
            <dd>@Model.Thing.Color</dd>
        </dl>
    </div>
    @if (Model.HasErrors)
    {
        <div>
            @foreach(string errorMessage in Model.ValidationErrors)
            {
                <div class="alert alert-danger" role="alert">@errorMessage</div>
            }
        </div>
    }
}

```

10. Start the website, start Chrome, and navigate to `https://localhost:5001/home/modelbinding`.
11. Click the **Submit** button and note that 1 and Red are valid values.
12. Enter an ID of 99, clear the color textbox, click the **Submit** button, and note the error messages, as shown in the following screenshot:

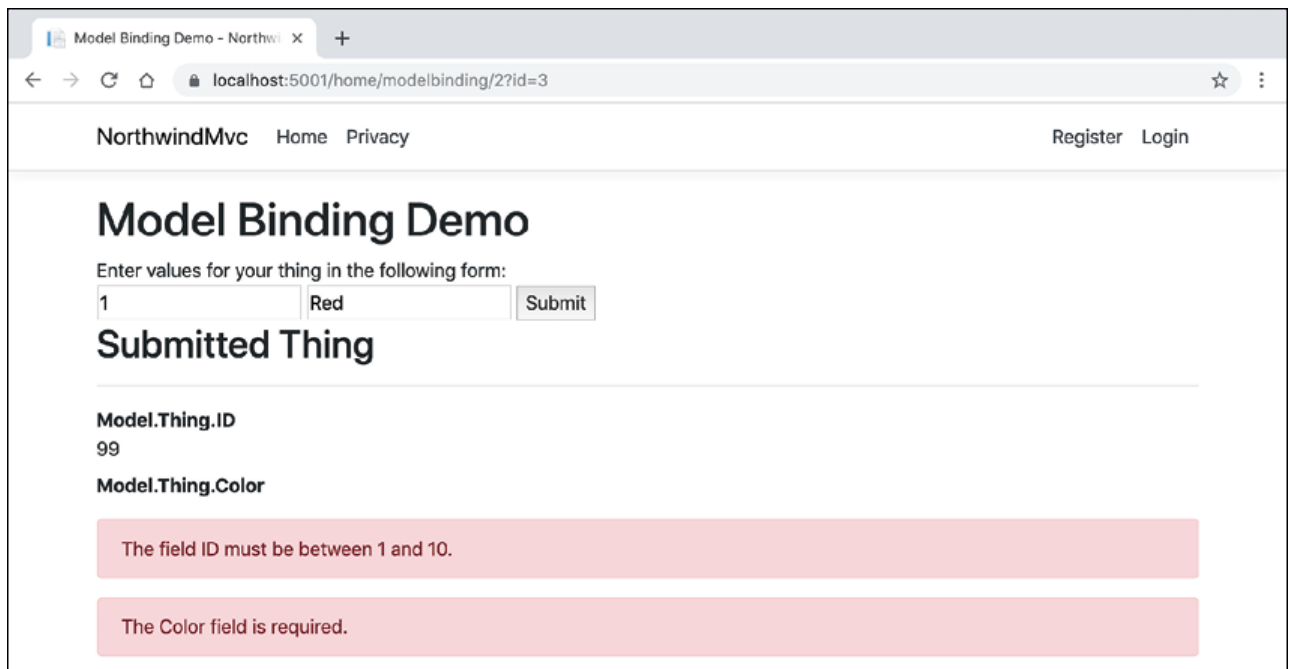


Figure 16.10: The Model Binding Demo page with field validations

13. Close the browser and stop the website.

More Information: You can read more about model validation at the following link: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>

Understanding view helper methods

While creating a view for ASP.NET Core MVC, you can use the `Html` object and its methods to generate markup.

Some useful methods include the following:

- **ActionLink:** Use this to generate an anchor `<a>` element that contains a URL path to the specified controller and action.
- **AntiForgeryToken:** Use this inside a `<form>` to insert a `<hidden>` element containing an anti-forgery token that will be validated when the form is submitted.

More Information: You can read more about anti-forgery tokens at the following link: <https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery>

- **Display and DisplayFor:** Use this to generate HTML markup for the expression relative to the current model using a display template. There are built-in display templates for .NET types and custom templates can be created in the `DisplayTemplates` folder. The folder name is case-sensitive on case-sensitive filesystems.
- **DisplayForModel:** Use this to generate HTML markup for an entire model instead of a single expression.
- **Editor and EditorFor:** Use this to generate HTML markup for the expression relative to the current model using an editor template. There are built-in editor templates for .NET types that use `<label>` and `<input>` elements, and custom templates can be created in the `EditorTemplates` folder. The folder name is case-sensitive on case-sensitive filesystems.
- **EditorForModel:** Use this to generate HTML markup for an entire model instead of a single expression.
- **Encode:** Use this to safely encode an object or string into HTML. For example, the string value `"<script>"` would be encoded as `"<script>"`. This is not normally necessary since the Razor `@` symbol encodes string values by default.
- **Raw:** Use this to render a string value *without* encoding as HTML.
- **PartialAsync and RenderPartialAsync:** Use these to generate HTML markup for a partial view. You can optionally pass a model and view data.

More Information: You can read more about the `HtmlHelper` class at the following link: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.htmlhelper>

Querying a database and using display templates

Let's create a new action method that can have a query string parameter passed to it and use that to query the Northwind database for products that cost more than a specified price:

1. In the `HomeController` class, import the `Microsoft.EntityFrameworkCore` namespace. We need this to add the `Include` extension method so that we can include related entities, as you learned in *Section 04, Working with Databases Using Entity Framework Core*.
2. Add a new action method, as shown in the following code:

```

public IActionResult ProductsThatCostMoreThan(decimal? price)
{
    if (!price.HasValue)
    {
        return NotFound("You must pass a product price in the query string, for example, /Home/ProductsThatCostMoreThan?price=50");
    }
    IEnumerable<Product> model = db.Products
        .Include(p => p.Category)
        .Include(p => p.Supplier)
        .Where(p => p.UnitPrice > price);
    if (model.Count() == 0)
    {
        return NotFound(
            $"No products cost more than {price:C}.");
    }
    ViewData["MaxPrice"] = price.Value.ToString("C");
    return View(model); // pass model to view
}

```

3. Inside the Views/Home folder, add a new file named ProductsThatCostMoreThan.cshtml.
4. Modify the contents, as shown in the following code:

```

@model IEnumerable<Packt.Shared.Product>
@{
    string title =
        "Products That Cost More Than " + ViewData["MaxPrice"];
    ViewData["Title"] = title;
}
<h2>@title</h2>
<table class="table">
    <thead>
        <tr>
            <th>Category Name</th>
            <th>Supplier's Company Name</th>
            <th>Product Name</th>
            <th>Unit Price</th>
            <th>Units In Stock</th>
        </tr>
    </thead>
    <tbody>
        @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>
        }
    </tbody>
</table>

```

5. In the Views/Home folder, open Index.cshtml.
6. Add the following form element below the visitor count and above the Products heading and its listing of products. This will provide a form for the user to enter a price. The user can then click on the **Submit** button to call the action method that shows only products that cost more than the entered price:

```

<h3>Query products by price</h3>
<form asp-action="ProductsThatCostMoreThan" method="get">
    <input name="price" placeholder="Enter a product price" />
    <input type="submit" />
</form>

```

7. Start the website, use Chrome to navigate to the website, and on the home page, enter a price in the form, for example, 50, and then click on **Submit**. You will see a table of the products that cost more than the price that you entered, as shown in the following screenshot:

Category Name	Supplier's Company Name	Product Name	Unit Price	Units In Stock
Meat/Poultry	Tokyo Traders	Mishi Kobe Niku	97.00	29
Seafood	Pavlova, Ltd.	Carnarvon Tigers	62.50	42
Confections	Specialty Biscuits, Ltd.	Sir Rodney's Marmalade	81.00	40

Figure 16.11: A filtered list of products that cost more than £50

8. Close the browser and stop the website.

Improving scalability using asynchronous tasks

When building a desktop or mobile app, multiple tasks (and their underlying threads) can be used to improve responsiveness, because while one thread is busy with the task, another can handle interactions with the user.

Tasks and their threads can be useful on the server side too, especially with websites that work with files, or request data from a store or a web service that could take a while to respond. But they are detrimental to complex calculations that are CPU-bound, so leave these to be processed synchronously as normal.

When an HTTP request arrives at the web server, a thread from its pool is allocated to handle the request. But if that thread must wait for a resource, then it is blocked from handling any more incoming requests. If a website receives more simultaneous requests than it has threads in its pool, then some of those requests will respond with a server timeout error, 503 Service Unavailable.

The threads that are locked are not doing useful work. They *could* handle one of those other requests but only if we implement asynchronous code in our websites.

Whenever a thread is waiting for a resource it needs, it can return to the thread pool and handle a different incoming request, improving the scalability of the website, that is, increasing the number of simultaneous requests it can handle.

Why not just have a larger thread pool? In modern operating systems, every thread pool thread has a 1 MB stack. An asynchronous method uses a smaller amount of memory. It also removes the need to create new threads in the pool, which takes time. The rate at which new threads are added to the pool is typically one every two seconds, which is a loooooong time compared to switching between asynchronous threads.

Making controller action methods asynchronous

It is easy to make an existing action method asynchronous:

1. In the HomeController class, make sure that the System.Threading.Tasks namespace has been imported.
2. Modify the Index action method to be asynchronous, to return a Task<T>, and to await the calls to asynchronous methods to get the categories and products, as shown highlighted in the following code:

```
public async Task<IActionResult> Index()
{
    var model = new HomeIndexViewModel
    {
        VisitorCount = (new Random()).Next(1, 1001),
        Categories = await db.Categories.ToListAsync(),
        Products = await db.Products.ToListAsync()
    };
    return View(model); // pass model to view
}
```

3. Modify the ProductDetail action method in a similar way, as shown highlighted in the following code:

```
public async Task<IActionResult> ProductDetail(int? id)
{
    if (!id.HasValue)
    {
        return NotFound("You must pass a product ID in the route, for example, /Home/ProductDetail/21");
    }
}
```

```
var model = await db.Products
    .SingleOrDefaultAsync(p => p.ProductID == id);
if (model == null)
{
    return NotFound($"Product with ID of {id} not found.");
}
return View(model); // pass model to view and then return result
}
```

4. Start the website, use Chrome to navigate to the website, and note that the functionality of the website is the same, but trust that it will now scale better.
5. Close the browser and stop the website.

Exploring an ASP.NET Core MVC website

Let's walk through the parts that make up a modern ASP.NET Core MVC website.

Understanding ASP.NET Core MVC startup

Appropriately enough, we will start by exploring the MVC website's default startup configuration:

1. Open the `Startup.cs` file.
2. Note the read-only `Configuration` property that can be passed in and set in the class constructor, as shown in the following code:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}
public IConfiguration Configuration { get; }
```

3. Note that the `ConfigureServices` method adds an application database context using SQLite with its database connection string loaded from the `appsettings.json` file for its data storage, adds ASP.NET Core Identity for authentication and configures it to use the application database, and adds support for MVC controllers with views, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();
}
```

More Information: You can learn more about the Identity UI library that is distributed as a Razor class library and can be overridden by a website at the following link: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?tabs=netcore-cli>

The call to `AddDbContext` is an example of registering a dependency service. ASP.NET Core implements the **dependency injection (DI)** design pattern so that controllers can request needed services through their constructors. Developers register those services in the `ConfigureServices` method.

More Information: You can read more about dependency injection at the following link: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

4. Next, we have the `Configure` method, which configures a detailed exception and database error page if the website runs in development, or a friendlier error page and HSTS for production. HTTPS redirection, static files, routing, and ASP.NET Identity are enabled, and an MVC default route and Razor Pages are configured, as shown in the following code:

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

```

else
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days.
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
}

```

We learned about most of these methods in *Section 07, Building Websites Using ASP.NET Core Razor Pages*. Apart from the `UseAuthentication` and `UseAuthorization` methods, the most important new method in the `Configure` method is `MapControllerRoute`, which maps a default route for use by MVC. This route is very flexible because it will map to almost any incoming URL, as you will see in the next section.

Although we will not create any Razor Pages in this section, we need to leave the method call that maps Razor Page support because our MVC website uses ASP.NET Core Identity for authentication and authorization, and it uses a Razor Class Library for its user interface components, like visitor registration and login.

More Information: You can read more about configuring middleware at the following link: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/>

Understanding the default MVC route

The responsibility of a route is to discover the name of a controller class to instantiate an action method to execute with an optional `id` parameter to pass into the method that will generate an HTTP response.

A default route is configured for MVC, as shown in the following code:

```

endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

```

The route pattern has parts in curly brackets `{ }` called **segments**, and they are like named parameters of a method. The value of these segments can be any string. Segments in URLs are not case-sensitive.

The route pattern looks at any URL path requested by the browser 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).

The following table contains example URLs and how the default route would work out the names of a controller and action:

URL	Controller	Action	ID
/	Home	Index	
/Muppet	Muppet	Index	

/Muppet/Kermit	Muppet	Kermit	
/Muppet/Kermit/Green	Muppet	Kermit	Green
/Products	Products	Index	
/Products/Detail	Products	Detail	
/Products/Detail/3	Products	Detail	3

Understanding controllers and actions

In ASP.NET Core MVC, the *C* stands for *controller*. From the route and an incoming URL, ASP.NET Core MVC knows the name of the controller, so it will then look for a class that is decorated with the `[Controller]` attribute or derives from a class decorated with that attribute, for example, the Microsoft provided class named `ControllerBase`, as shown in the following code:

```
namespace Microsoft.AspNetCore.Mvc
{
    //
    // Summary:
    // A base class for an MVC controller without view support.
    [Controller]
    public abstract class ControllerBase
    ...
}
```

As you can see in the XML comment, `ControllerBase` does not support views. It is used for creating web services, as you will see in *Section 9, Building and Consuming Web Services*.

Microsoft provides a class named `Controller` that your classes can inherit from if they do need view support.

The responsibilities of a controller are as follows:

- Identify the services that the controller needs in order to be in a valid state and to function properly in their class constructor(s).
- Use the *action* name to identify a method to execute.
- Extract parameters from the HTTP request.
- Use the parameters to fetch any additional data needed to construct a view model and pass it to the appropriate view for the client. For example, if the client is a web browser, then a view that renders HTML would be most appropriate. Other clients might prefer alternative renderings, like document formats such as a PDF file or an Excel file, or data formats, like JSON or XML.
- Return the results from the view to the client as an HTTP response with an appropriate status code.

Let's review the controller used to generate the home, privacy, and error pages:

1. Expand the `Controllers` folder.
2. Open the file named `HomeController.cs`.
3. Note, as shown in the following code, that:
 - A private field is declared to store a reference to a logger for the `HomeController` that is set in a constructor.
 - All three action methods call a method named `View()` and return the results as an `ActionResult` interface to the client.
 - The `Error` action method passes a view model into its view with a request ID used for tracing. The error response will not be cached:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }
}
```

```

}
public IActionResult Index()
{
    return View();
}
public IActionResult Privacy()
{
    return View();
}
[ResponseCache(Duration = 0,
    Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View(new ErrorViewModel { RequestId =
        Activity.Current?.Id ?? HttpContext.TraceIdentifier });
}
}

```

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

Understanding the view search path convention

Both the `Index` and `Privacy` methods have identical implementation, yet they return different web pages. This is because of conventions. The call to the `View` method looks in different paths for the Razor file to generate the web page:

1. In the `NorthwindMvc` project, expand the `Views` and then `Home` folders.
2. Rename the `Privacy.cshtml` file to `Privacy2.cshtml`.
3. In **TERMINAL**, enter the command `dotnet run` to start the website.
4. Start Chrome, navigate to `http://localhost:5000/`, click **Privacy**, and note the paths that are searched for a view to render the web page including in `Shared` folders, as shown in the following output:

```

InvalidOperationException: The view 'Privacy' was not found. The following locations were searched:
/Views/Home/Privacy.cshtml
/Views/Shared/Privacy.cshtml
/Pages/Shared/Privacy.cshtml

```

5. Close Chrome.
6. Rename the `Privacy2.cshtml` file back to `Privacy.cshtml`.

The view search path convention is shown in the following list:

- Specific Razor view: `/Views/{controller}/{action}.cshtml`
- Shared Razor view: `/Views/Shared/{action}.cshtml`
- Shared Razor page: `/Pages/Shared/{action}.cshtml`

Unit testing MVC

Controllers are where the business logic of your website runs, so it is important to test the correctness of that logic using unit tests.

More Information: You can read more about how to unit test controllers at the following link: <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing>

Understanding filters

When you need to add some functionality to multiple controllers and actions, then you can use or define your own filters that are implemented as an attribute class.

Filters can be applied at the following levels:

- At the action-level by decorating the method with the attribute. This will only affect the one method.
- At the controller-level by decorating the class with the attribute. This will affect all methods of this controller.
- At the global level by adding an instance of the attribute to the `Filters` collection of the `IServiceCollection` in the `ConfigureServices` method of the `Startup` class. This will affect all methods of all controllers in the project.

More Information: You can read more about filters at the following link:

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters>

Using a filter to secure an action method

For example, you might want to ensure that one particular method of a controller can only be called by members of certain security roles. You do this by decorating the method with the `[Authorize]` attribute, as shown in the following code:

```
[Authorize(Roles = "Sales,Marketing")]
public IActionResult SalesAndMarketingEmployeesOnly()
{
    return View();
}
```

More Information: You can read more about authorization at the following link:

<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction>

Using a filter to cache a response

You might want to cache the HTTP response that is generated by an action method by decorating the method with the `[ResponseCache]` attribute, as shown in the following code:

```
[ResponseCache(Duration = 3600, // in seconds therefore 1 hour
    Location = ResponseCacheLocation.Any)]
public IActionResult AboutUs()
{
    return View();
}
```

You control where the response is cached and for how long by setting parameters, as shown in the following list:

- **Duration:** In seconds. This sets the `max-age` HTTP response header.
- **Location:** One of the `ResponseCacheLocation` values, `Any`, `Client`, or `None`. This sets the `cache-control` HTTP response header.
- **NoStore:** If `true`, this ignores `Duration` and `Location` and sets the `cache-control` HTTP response header to `no-store`.

More Information: You can read more about response caching at the following link:

<https://docs.microsoft.com/en-us/aspnet/core/performance/caching/response>

Using a filter to define a custom route

You might want to define a simplified route for an action method instead of using the default route.

For example, to show the privacy page currently requires the following URL path, which specifies both the controller and action:

<https://localhost:5001/home/privacy>

We could decorate the action method to make the route simpler, as shown in the following code:

```
[Route("private")]
public IActionResult Privacy()
{
    return View();
}
```

Now, we can use the following URL path, which uses the attribute-defined route:

```
https://localhost:5001/private
```

Understanding entity and view models

In ASP.NET Core MVC, the *M* stands for *model*. Models represent the data required to respond to a request. **Entity models** represent entities in a data store like SQLite. Based on the request, one or more entities might need to be retrieved from data storage. All of 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 into a *view* for rendering into a response format like HTML or JSON.

For example, the following HTTP GET request might mean that the browser is asking for the product details page for product number 3:

```
http://www.example.com/products/details/3
```

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

Imagine that when a user comes to our website, we want to show them a carousel of categories, a list of products, and a count of the number of visitors we have had this month.

We will reference the Entity Framework Core entity data model for the Northwind database that you created in *Section 06, Introducing Practical Applications of C# and .NET*:

1. In the NorthwindMvc project, open NorthwindMvc.csproj.
2. Add a project reference to NorthwindContextLib, as shown in the following markup:

```
<ItemGroup>
  <ProjectReference Include=
    "..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

3. In **TERMINAL**, enter the following command to rebuild the project:

```
dotnet build
```

4. Modify Startup.cs to import the System.IO and Packt.Shared namespaces and add a statement to the ConfigureServices method to configure the Northwind database context, as shown in the following code:

```
string databasePath = Path.Combine("...", "Northwind.db");
services.AddDbContext<Northwind>(options =>
    options.UseSqlite($"Data Source={databasePath}"));
```

5. Add a class file to the Models folder and name it HomeIndexViewModel.cs.

Good Practice: Although the ErrorViewModel class created by the MVC project template does not follow this convention, I recommend that you use the naming convention {Controller}{Action}ViewModel for your view model classes.

6. Modify the class definition to have three properties for a count of the number of visitors, and lists of categories and products, as shown in the following code:

```
using System.Collections.Generic;
using Packt.Shared;
namespace NorthwindMvc.Models
{
    public class HomeIndexViewModel
    {
        public int VisitorCount;
        public IList<Category> Categories { get; set; }
        public IList<Product> Products { get; set; }
    }
}
```

7. Open the HomeController class.
8. Import the Packt.Shared namespace.
9. Add a field to store a reference to a Northwind instance, and initialize it in the constructor, as shown highlighted in the following code:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private Northwind db;
    public HomeController(ILogger<HomeController> logger,
        Northwind injectedContext)
    {
        _logger = logger;
        db = injectedContext;
    }
}
```

ASP.NET Core will use constructor parameter injection to pass an instance of the Northwind database context using the database path you specified in the Startup class.

10. Modify the contents of the Index action method to create an instance of the view model for this method, simulating a visitor count using the Random class to generate a number between 1 and 1000, and using the Northwind database to get lists of categories and products, and then pass the model to the view, as shown in the following code:

```
var model = new HomeIndexViewModel
{
    VisitorCount = (new Random()).Next(1, 1001),
    Categories = db.Categories.ToList(),
    Products = db.Products.ToList()
};
return View(model); // pass model to view
```

When the View() method is called in a controller's action method, ASP.NET Core 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.cshtml. It will also search for shared views.

Understanding views

In ASP.NET Core MVC, the V stands for *view*. The responsibility of a view is to transform a model into HTML or other formats.

There are multiple **view engines** that could be used to do this. The default view engine is called **Razor**, and it uses the @ symbol to indicate server-side code execution.

The Razor Pages feature introduced with ASP.NET Core 2.0 uses the same view engine and so can use the same Razor syntax.

Let's modify the home page view to render the lists of categories and products:

1. Expand the Views folder, and then expand the Home folder.
2. Open the Index.cshtml file and note the block of C# code wrapped in @{ }. This will execute first and can be used to store data that needs to be passed into a shared layout file like

the title of the web page, as shown in the following code:

```
@{
    ViewData["Title"] = "Home Page";
}
```

3. Note the static HTML content in the `<div>` element that uses Bootstrap for styling.

Good Practice: As well as defining your own styles, base your styles on a common library, such as Bootstrap, that implements responsive design.

Just as with Razor Pages, 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, as shown in the following markup:

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

4. In the `Views` folder, open the `_ViewImports.cshtml` file and note that it imports some namespaces and then adds the ASP.NET Core tag helpers, as shown in the following code:

```
@using NorthwindMvc
@using NorthwindMvc.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

5. In the `Shared` folder, open the `_Layout.cshtml` file.
6. Note that the title is being read from the `ViewData` dictionary that was set earlier in the `Index.cshtml` view, as shown in the following markup:

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

7. Note the rendering of links to support Bootstrap and a site stylesheet, where `~` means the `wwwroot` folder, as shown in the following markup:

```
<link rel="stylesheet"
      href="~/lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="~/css/site.css" />
```

8. Note the rendering of a navigation bar in the header, as shown in the following markup:

```
<body>
  <header>
    <nav class="navbar ...">
```

9. Note the rendering of a collapsible `<div>` containing a partial view for logging in and hyperlinks to allow users to navigate between pages using ASP.NET Core tag helpers with attributes like `asp-controller` and `asp-action`, as shown in the following markup:

```
<div class=
  "navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
  <partial name="_LoginPartial" />
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area=""
        asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark"
        asp-area="" asp-controller="Home"
        asp-action="Privacy">Privacy</a>
    </li>
  </ul>
</div>
```

The `<a>` elements use tag helper attributes named `asp-controller` and `asp-action` to specify the controller name and action name that will execute when the link is clicked on. If you

want to navigate to a feature in a Razor Class Library, then you use `asp-area` to specify the feature name.

10. Note the rendering of the body inside the `<main>` element, as shown in the following markup:

```
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>
```

The `@RenderBody()` method call injects the contents of a specific Razor view for a page like the `Index.cshtml` file at that point in the shared layout.

More Information: You can read about why it is good to put `<script>` elements at the bottom of the `<body>` at the following link:

<https://stackoverflow.com/questions/436411/where-should-i-put-script-tags-in-html-markup>

11. Note the rendering of `<script>` elements at the bottom of the page so that it doesn't slow down the display of the page and that you can add your own script blocks into an optional defined section named `scripts`, as shown in the following markup:

```
<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js">
</script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("scripts", required: false)
```

When `asp-append-version` is specified with a `true` value in any element along with an `src` attribute, the Image Tag Helper is invoked (it does not only affect images!).

It works by automatically appending a query string value named `v` that is generated from a hash of the referenced source file, as shown in the following example generated output:

```
<script src="~/js/site.js?v=K1_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM"></script>
```

If even a single byte within the `site.js` file changes, then its hash value will be different, and therefore if a browser or CDN is caching the script file, then it will bust the cached copy and replace it with the new version.

More Information: You can read how cache busting using query strings works at the following link: <https://stackoverflow.com/questions/9692665/cache-busting-via-params>

Customizing an ASP.NET Core MVC website

Now that you've reviewed the structure of a basic MVC website, you will customize it. You have already added code to retrieve entities from the `Northwind` database, so the next task is to output that information on the home page.

More Information: To find suitable images for the eight categories, I searched on a site that has free stock photos for commercial use with no attribution at the following link: <https://www.pexels.com/>

Defining a custom style

The home page will show a list of the 77 products in the `Northwind` database. To make efficient use of space, we want to show the list in three columns. To do this, we need to customize the stylesheet for the website:

1. In the `wwwroot\css` folder, open the `site.css` file.
2. At the bottom of the file, add a new style that will apply to an element with the newspaper ID, as shown in the following code:

```
#newspaper
{
    column-count: 3;
}
```

Setting up the category images

The `Northwind` database includes a table of categories, but they do not have images, and websites look better with some colorful pictures:

1. In the `wwwroot` folder, create a folder named `images`.
2. In the `images` folder, add eight image files from the downloads named `category1.jpeg`, `category2.jpeg`, and so on, up to `category8.jpeg`.

Understanding Razor syntax

Before we customize the home page view, let's review an example Razor file that has an initial Razor code block that instantiates an order with price and quantity and then outputs information about the order on the web page, as shown in the following markup:

```
@{
    var order = new Order
    {
        OrderID = 123,
        Product = "Sushi",
        Price = 8.49M,
        Quantity = 3
    };
}
<div>Your order for @order.Quantity of @order.Product has a total cost of $@order.Price * @order.Quantity</div>
```

The preceding Razor file would result in the following incorrect output:

```
Your order for 3 of Sushi has a total cost of $8.49 * 3
```

Although Razor markup can include the value of any single property using the `@object.property` syntax, you should wrap expressions in parentheses, as shown in the following markup:

```
<div>Your order for @order.Quantity of @order.Product has a total cost of $@(order.Price * @order.Quantity)</div>
```

The preceding Razor expression results in the following correct output:

```
Your order for 3 of Sushi has a total cost of $25.47
```

Defining a typed view

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

1. In the `Views\Home` folder, open `Index.cshtml`.
2. At the top of the file, add a statement to set the model type to use the `HomeIndexViewModel`, as shown in the following code:

```
@model NorthwindMvc.Models.HomeIndexViewModel
```

Now, whenever we type `Model` in this view, the Visual Studio Code C# extension will know the correct type for the model and will provide IntelliSense for it.

While entering code in a view, remember the following:

- To declare the type for the model, use `@model` (with lowercase `m`).
- To interact with the model instance, use `@Model` (with uppercase `M`).

Let's continue customizing the view for the home page.

3. In the initial Razor code block, add a statement to declare a `string` variable for the current item and replace the existing `<div>` element with the new markup to output categories in a carousel and products as an unordered list, as shown in the following markup:

```
@model NorthwindMvc.Models.HomeIndexViewModel
@{
    ViewData["Title"] = "Home Page";
    string currentItem = "";
}
<div class="carousel slide" data-ride="carousel"
    data-interval="3000" data-keyboard="true">
    <ol class="carousel-indicators">
        @for (int c = 0; c < Model.Categories.Count; c++)
        {
            if (c == 0)
            {
                currentItem = "active";
            }
            else
            {
                currentItem = "";
            }
            <li data-target="#categories" data-slide-to="@c"
                class="@currentItem"></li>
        }
    </ol>
    <div class="carousel-inner">
        @for (int c = 0; c < Model.Categories.Count; c++)
        {
            if (c == 0)
            {
                currentItem = "active";
            }
            else
            {
                currentItem = "";
            }
            <div class="carousel-item @currentItem">
                <img class="d-block w-100" src=
                    "~/images/category@(Model.Categories[c].CategoryID).jpeg"
                    alt="@Model.Categories[c].CategoryName" />
                <div class="carousel-caption d-none d-md-block">
                    <h2>@Model.Categories[c].CategoryName</h2>
                    <h3>@Model.Categories[c].Description</h3>
                    <p>
                        <a class="btn btn-primary"
                            href="/category/@Model.Categories[c].CategoryID">View</a>
                    </p>
                </div>
            </div>
        }
    </div>
    <a class="carousel-control-prev" href="#categories"
        role="button" data-slide="prev">
        <span class="carousel-control-prev-icon"
            aria-hidden="true"></span>
        <span class="sr-only">Previous</span>
    </a>
    <a class="carousel-control-next" href="#categories"
        role="button" data-slide="next">
        <span class="carousel-control-next-icon"
            aria-hidden="true"></span>
        <span class="sr-only">Next</span>
    </a>
</div>
<div class="row">
    <div class="col-md-12">
        <h1>Northwind</h1>
        <p class="lead">
            We have had @Model.VisitorCount visitors this month.
        </p>
        <h2>Products</h2>
        <div>
            <ul>
                @foreach (var item in @Model.Products)
                {
                    <li>
                        <a asp-controller="Home"
                            asp-action="ProductDetail"
                            asp-route-@
                                @item.ProductName costs
                                @item.UnitPrice.ToString("C")
                            >
                        </a>
                    </li>
                }
            </ul>
        </div>
    </div>
</div>
```

```

    }
  </ul>
</div>
</div>
</div>

```

While reviewing the preceding Razor markup, note the following:

- It is easy to mix static HTML elements such as `` and `` with C# code to output the carousel of categories and the list of product names.
- The `<div>` element with the `id` attribute of `newspaper` will use the custom style that we defined earlier, so all of the content in that element will display in three columns.
- The `` element for each category uses parentheses around a Razor expression to ensure that the compiler does not include the `.jpeg` as part of the expression, as shown in the following markup:
`"~/images/category@(Model.Categories[c].CategoryID).jpeg"`
- The `<a>` elements for the product links use tag helpers to generate URL paths. Clicks on these hyperlinks will be handled by the `Home` controller and its `ProductDetail` action method. This action method does not exist yet, but you will add it later in this section. The ID of the product is passed as a route segment named `id`, as shown in the following URL path for Ipoh Coffee:

`https://localhost:5001/Home/ProductDetail/43`

Reviewing the customized home page

Let's see the result of our customized home page:

1. Start the website by entering the following command: `dotnet run`.
2. Start Chrome and navigate to `http://localhost:5000`.
3. Note the home page has a rotating carousel showing categories, a random number of visitors, and a list of products in three columns, as shown in the following screenshot:

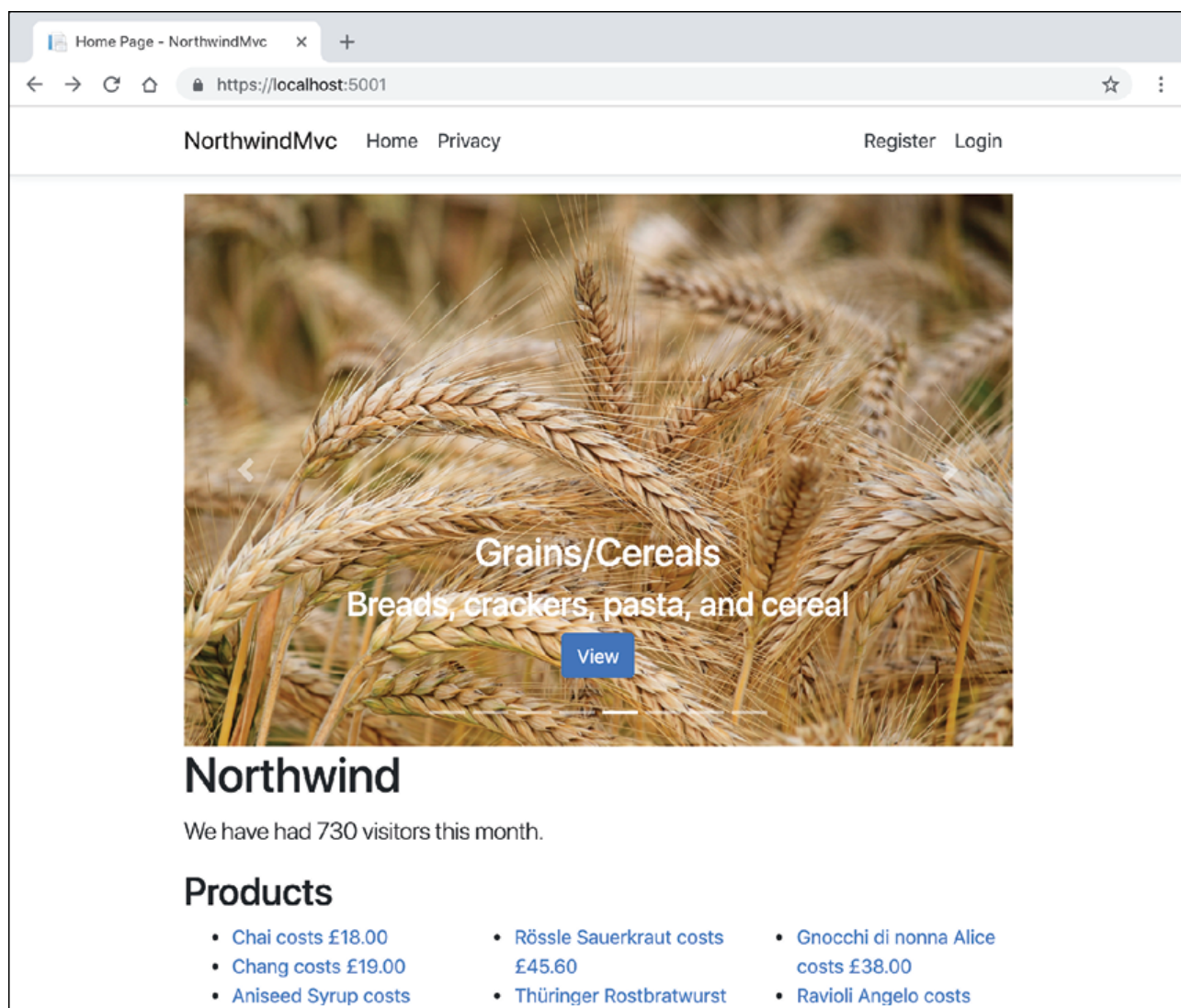


Figure 16.5: The updated Northwind MVC website homepage

At the moment, clicking on any of the categories or product links gives 404 Not Found errors, so let's see how we can pass parameters so that we can see the details of a product or category.

4. Close Chrome.
5. Stop the website by pressing `Ctrl + C` in **TERMINAL**.

Passing parameters using a route value

One way to pass a simple parameter is to use the `id` segment defined in the default route:

1. In the `HomeController` class, add an action method named `ProductDetail`, as shown in the following code:

```
public IActionResult ProductDetail(int? id)
{
    if (!id.HasValue)
    {
        return NotFound("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 NotFound($"Product with ID of {id} not found.");
    }
    return View(model); // pass model to view and then return result
}
```

Note the following:

- This method uses a feature of ASP.NET Core called **model binding** to automatically match the `id` passed in the route to the parameter named `id` in the method.
- Inside the method, we check to see whether `id` does not have a value, and if so, we call the `NotFound` method to return a 404 status code with a custom message explaining the correct URL path format.
- Otherwise, we can connect to the database and try to retrieve a product using the `id` variable.
- If we find a product, we pass it to a view; otherwise, we call the `NotFound` method to return a 404 status code and a custom message explaining that a product with that ID was not found in the database.

Remember that if the view is named to match the action method and is placed in a folder that matches the controller name or a shared folder, then ASP.NET Core MVC's conventions will find it automatically.

2. Inside the `Views/Home` folder, add a new file named `ProductDetail.cshtml`.
3. Modify the contents, as shown in the following markup:

```
@model Packt.Shared.Product
@{
    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.ToString("C")</dd>
        <dt>Units In Stock</dt>
        <dd>@Model.UnitsInStock</dd>
    </dl>
</div>
```

4. Start the website by entering the following command: `dotnet run`.
5. Start Chrome and navigate to `http://localhost:5000`.
6. When the home page appears with the list of products, click on one of them, for example, the second product, **Chang**.
7. Note the URL path in the browser's address bar, the page title shown in the browser tab, and the product details page, as shown in the following screenshot:

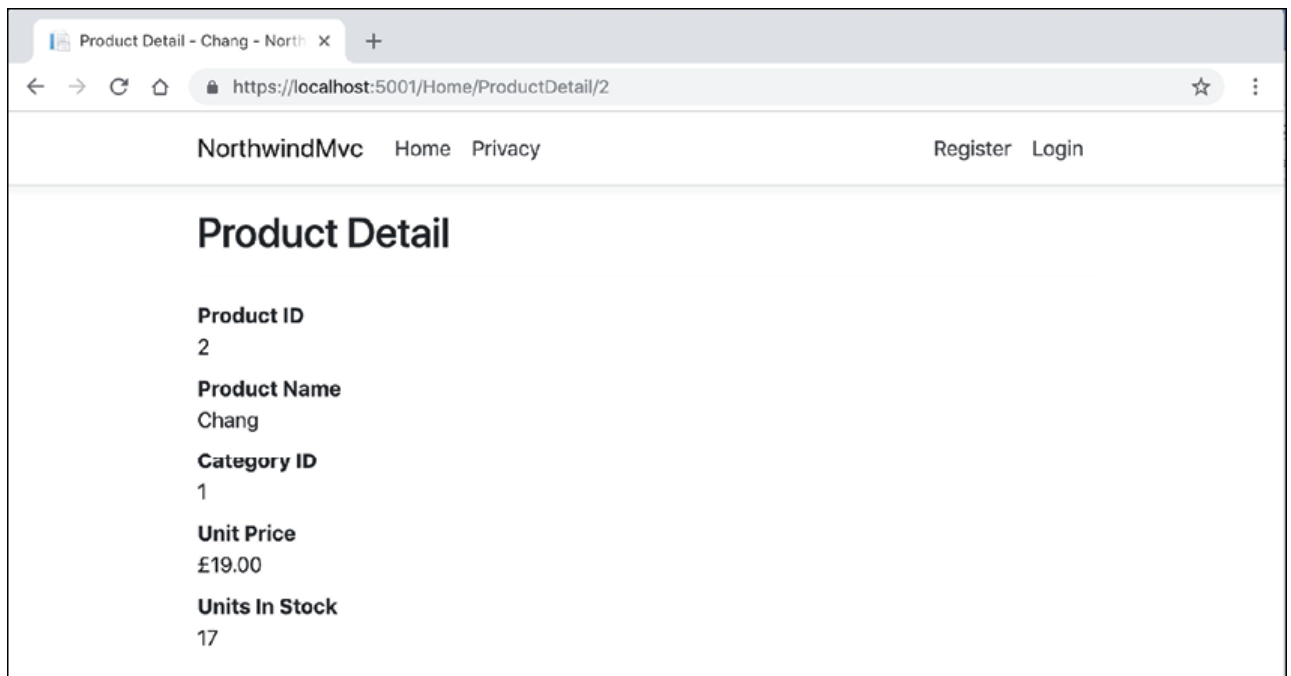


Figure 16.6: A product detail page

8. Toggle on the developer tools pane.
9. Edit the URL in the address box of Chrome to request a product ID that does not exist, like 99, and note the 404 Not Found status code and custom error response, as shown in the following screenshot:

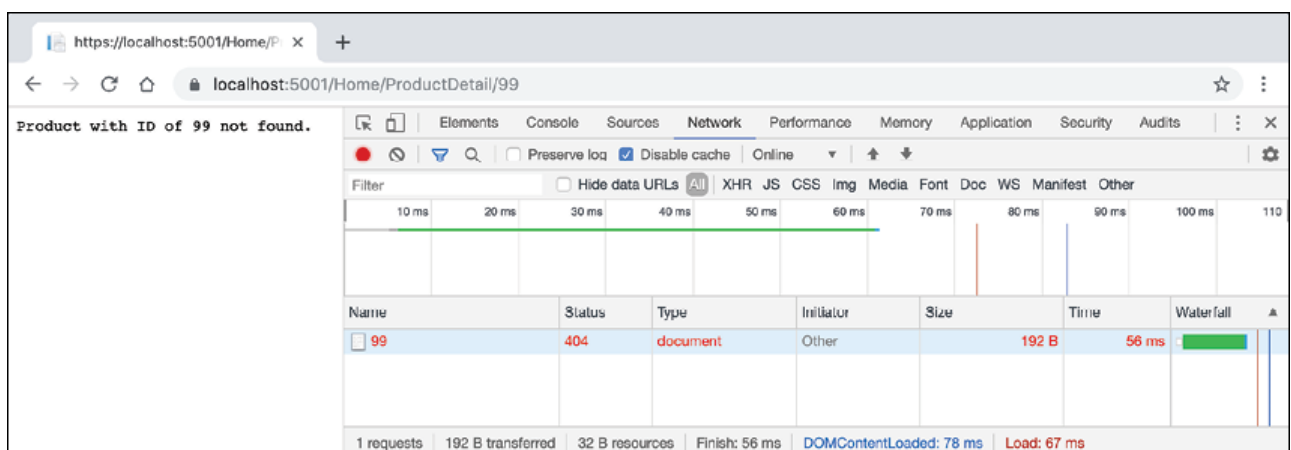


Figure 16.7: Requesting a product ID that doesn't exist

Understanding model binders

Model binders are very powerful, and the default one does a lot for you. After the default route identifies a controller class to instantiate and an action method to call, if that method has parameters, then those parameters need to have values set.

Model binders do this by looking for parameter values passed in the HTTP request as any of the following types of parameters:

- **Route parameter**, like `id` as we did in the previous section, as shown in the following URL path: `/Home/ProductDetail/2`
- **Query string parameter**, as shown in the following URL path: `/Home/ProductDetail?id=2`
- **Form parameter**, as shown in the following markup:

```
<form action="post" action="/Home/ProductDetail">
  <input type="text" name="id" />
  <input type="submit" />
</form>
```

Model binders can populate almost any type:

- Simple types, like `int`, `string`, `DateTime`, and `bool`.
- Complex types defined by `class` or `struct`.
- Collections types, like arrays and lists.

Let's create a somewhat artificial example to illustrate what can be achieved using the default model binder:

1. In the `Models` folder, add a new file named `Thing.cs`.
2. Modify the contents to define a class with two properties for a nullable number named `ID` and a string named `color`, as shown in the following code:

```
namespace NorthwindMvc.Models
{
    public class Thing
    {
        public int? ID { get; set; }
        public string Color { get; set; }
    }
}
```

3. Open `HomeController.cs` and add two new action methods, one to show a page with a form and one to display it with a parameter using your new model type, as shown in the following code:

```
public IActionResult ModelBinding()
{
    return View(); // the page with a form to submit
}
public IActionResult ModelBinding(Thing thing)
{
    return View(thing); // show the model bound thing
}
```

4. In the `Views\Home` folder, add a new file named `ModelBinding.cshtml`.
5. Modify its contents, as shown in the following markup:

```
@model NorthwindMvc.Models.Thing
@{
    ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
    Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding?id=3">
    <input name="color" value="Red" />
    <input type="submit" />
</form>
@if (Model != null)
{
    <h2>Submitted Thing</h2>
    <hr />
    <div>
        <dl class="dl-horizontal">
            <dt>Model.ID</dt>
            <dd>@Model.ID</dd>
            <dt>Model.Color</dt>
            <dd>@Model.Color</dd>
        </dl>
    </div>
</div>
}
```

6. Start the website, start Chrome, and navigate to `https://localhost:5001/home/modelbinding`.
7. Note the unhandled exception about an ambiguous match, as shown in the following screenshot:

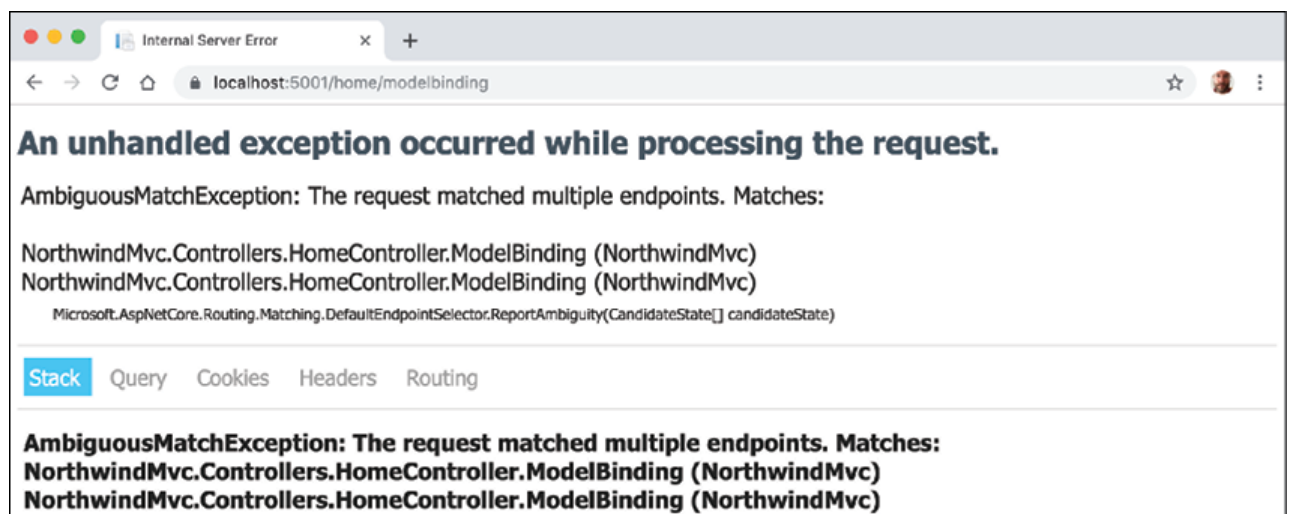


Figure 16.8: An unhandled exception error

Although the C# compiler can differentiate between the two methods by noting that the signatures are different, from HTTP's point of view, both methods are potential matches. We need an HTTP-specific way to disambiguate the action methods. We could do this by creating different names for the actions, or by specifying that one method should be used for a specific HTTP verb, like `GET`, `POST`, or `DELETE`.

8. Stop the website by pressing `Ctrl + C` in **TERMINAL**.

9. In `HomeController.cs`, decorate the second `ModelBinding` action method to indicate that it should be used for processing HTTP POST requests, that is, when a form is submitted, as shown highlighted in the following code:

```
[HttpPost]
public IActionResult ModelBinding(Thing thing)
```

10. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
11. Click the **Submit** button and note the value for the `ID` property is set from the query string parameter and the value for the `color` property is set from the form parameter, as shown in the following screenshot:

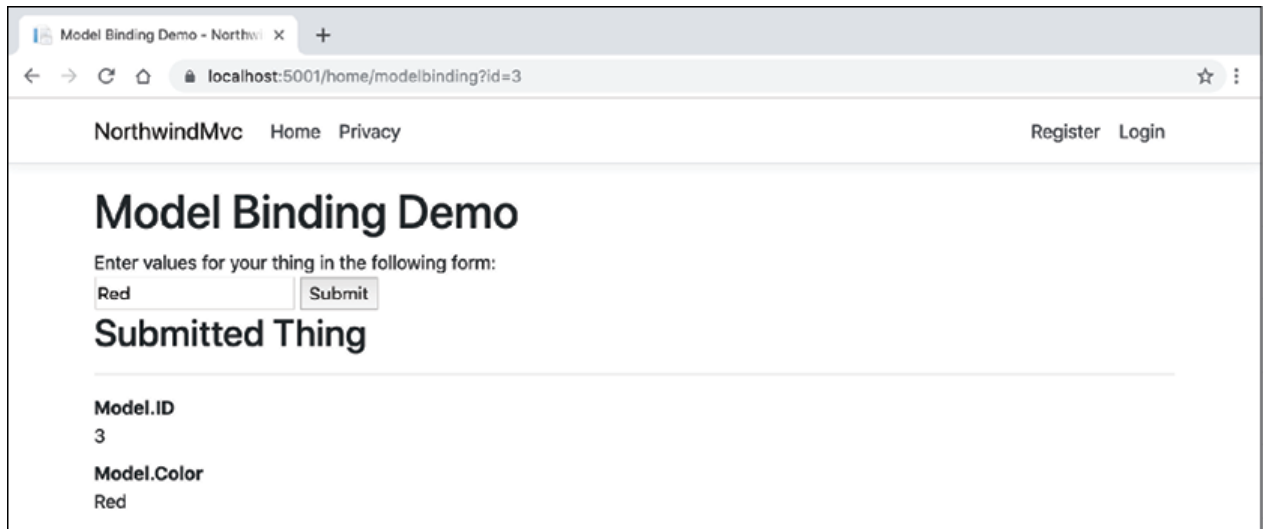


Figure 16.9: The Model Binding Demo page

12. Stop the website.
13. Modify the action for the form to pass the value 2 as a route parameter, as shown highlighted in the following markup:

```
<form method="POST" action="/home/modelbinding/2?id=3">
```

14. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
15. Click the **Submit** button and note the value for the `ID` property is set from the route parameter and the value for the `color` property is set from the form parameter.
16. Stop the website.
17. Modify the action for the form to pass the value 1 as a form parameter, as shown highlighted in the following markup:

```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input type="submit" />
</form>
```

18. Start the website, start Chrome, and navigate to <https://localhost:5001/home/modelbinding>.
19. Click the **Submit** button and note the values for the `ID` and `color` properties are both set from the form parameters.
20. If you have multiple parameters with the same name, then form parameters have the highest priority and query string parameters have the lowest priority for automatic model binding.

More Information: For advanced scenarios, you can create your own model binders by implementing the `IModelBinder` interface: <https://docs.microsoft.com/en-us/aspnet/core/mvc/advanced/custom-model-binding>

Validating the model

The process of model binding can cause errors, for example, data type conversions or validation errors if the model has been decorated with validation rules. What data has been bound and any binding or validation errors are stored in `ControllerBase.ModelState`.

Let's explore what we can do with model state by applying some validation rules to the bound model and then showing invalid data messages in the view:

1. In the `Models` folder, open `Thing.cs`.
2. Import the `System.ComponentModel.DataAnnotations` namespace.
3. Decorate the `ID` property with a validation attribute to limit the range of allowed numbers to 1 to 10, and one to ensure that the visitor supplies a color, as shown in the following highlighted code:

```
public class Thing
{
    [Range(1, 10)]
    public int? ID { get; set; }
    [Required]
```

```

    public string Color { get; set; }
}

```

4. In the `Models` folder, add a new file named `HomeModelBindingViewModel.cs`.
5. Modify its contents to define a class with two properties for the bound model and for any errors, as shown in the following code:

```

using System.Collections.Generic;
namespace NorthwindMvc.Models
{
    public class HomeModelBindingViewModel
    {
        public Thing Thing { get; set; }
        public bool HasErrors { get; set; }
        public IEnumerable<string> ValidationErrors { get; set; }
    }
}

```

6. In the `Controllers` folder, open `HomeController.cs`.
7. In the second `ModelBinding` method, comment out the previous statement that passed the thing to the view, and instead add statements to create an instance of the view model. Validate the model and store an array of error messages, and then pass the view model to the view, as shown in the following code:

```

public IActionResult ModelBinding(Thing thing)
{
    // return View(thing); // show the model bound thing
    var model = new HomeModelBindingViewModel
    {
        Thing = thing,
        HasErrors = !ModelState.IsValid,
        ValidationErrors = ModelState.Values
            .SelectMany(state => state.Errors)
            .Select(error => error.ErrorMessage)
    };
    return View(model);
}

```

8. In `Views\Home`, open `ModelBinding.cshtml`.
9. Modify the model type declaration to use the view model class, add a `<div>` to show any model validation errors, and change the output of the thing's properties because the view model has changed, as shown highlighted in the following markup:

```

@model NorthwindMvc.Models.HomeModelBindingViewModel
@{
    ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
    Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding/2?id=3">
    <input name="id" value="1" />
    <input name="color" value="Red" />
    <input type="submit" />
</form>
@if (Model != null)
{
    <h2>Submitted Thing</h2>
    <hr />
    <div>
        <dl class="dl-horizontal">
            <dt>Model.Thing.ID</dt>
            <dd>@Model.Thing.ID</dd>
            <dt>Model.Thing.Color</dt>
            <dd>@Model.Thing.Color</dd>
        </dl>
    </div>
    @if (Model.HasErrors)
    {
        <div>
            @foreach(string errorMessage in Model.ValidationErrors)
            {
                <div class="alert alert-danger" role="alert">@errorMessage</div>
            }
        </div>
    }
}

```

10. Start the website, start Chrome, and navigate to `https://localhost:5001/home/modelbinding`.
11. Click the **Submit** button and note that 1 and Red are valid values.
12. Enter an ID of 99, clear the color textbox, click the **Submit** button, and note the error messages, as shown in the following screenshot:

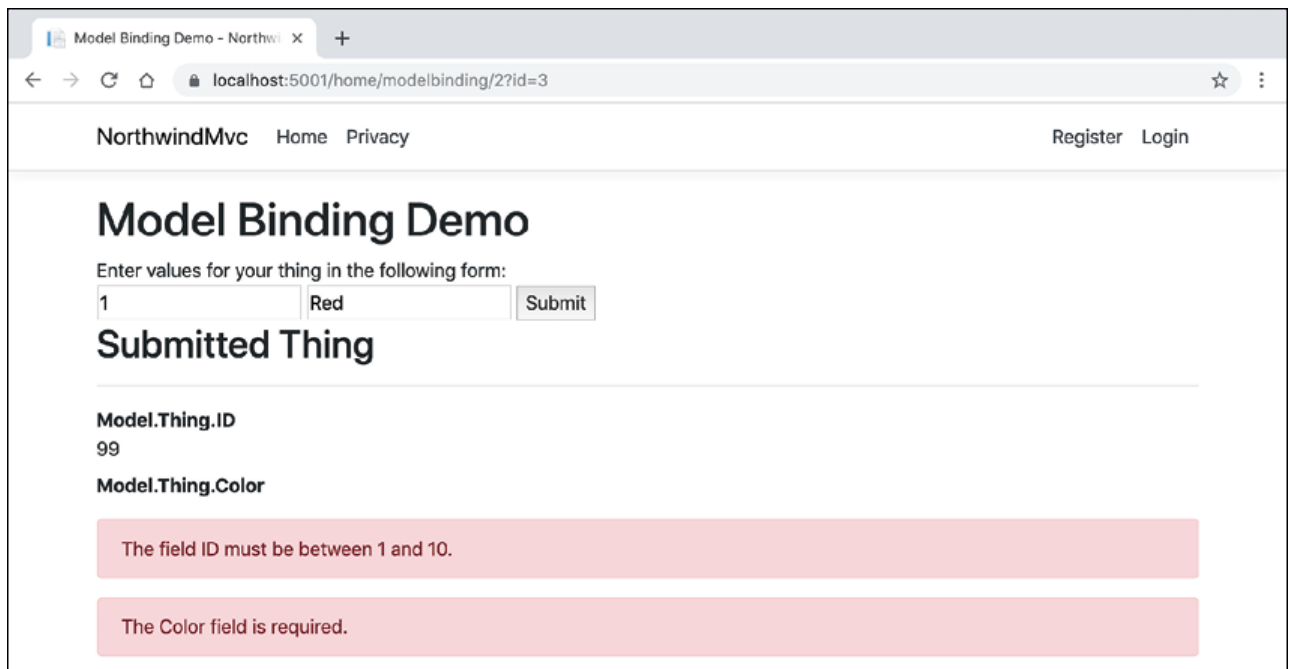


Figure 16.10: The Model Binding Demo page with field validations

13. Close the browser and stop the website.

More Information: You can read more about model validation at the following link: <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>

Understanding view helper methods

While creating a view for ASP.NET Core MVC, you can use the `Html` object and its methods to generate markup.

Some useful methods include the following:

- **ActionLink:** Use this to generate an anchor `<a>` element that contains a URL path to the specified controller and action.
- **AntiForgeryToken:** Use this inside a `<form>` to insert a `<hidden>` element containing an anti-forgery token that will be validated when the form is submitted.

More Information: You can read more about anti-forgery tokens at the following link: <https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery>

- **Display and DisplayFor:** Use this to generate HTML markup for the expression relative to the current model using a display template. There are built-in display templates for .NET types and custom templates can be created in the `DisplayTemplates` folder. The folder name is case-sensitive on case-sensitive filesystems.
- **DisplayForModel:** Use this to generate HTML markup for an entire model instead of a single expression.
- **Editor and EditorFor:** Use this to generate HTML markup for the expression relative to the current model using an editor template. There are built-in editor templates for .NET types that use `<label>` and `<input>` elements, and custom templates can be created in the `EditorTemplates` folder. The folder name is case-sensitive on case-sensitive filesystems.
- **EditorForModel:** Use this to generate HTML markup for an entire model instead of a single expression.
- **Encode:** Use this to safely encode an object or string into HTML. For example, the string value `"<script>"` would be encoded as `"<script>"`. This is not normally necessary since the Razor `@` symbol encodes string values by default.
- **Raw:** Use this to render a string value *without* encoding as HTML.
- **PartialAsync and RenderPartialAsync:** Use these to generate HTML markup for a partial view. You can optionally pass a model and view data.

More Information: You can read more about the `HtmlHelper` class at the following link: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.htmlhelper>

Querying a database and using display templates

Let's create a new action method that can have a query string parameter passed to it and use that to query the Northwind database for products that cost more than a specified price:

1. In the `HomeController` class, import the `Microsoft.EntityFrameworkCore` namespace. We need this to add the `Include` extension method so that we can include related entities, as you learned in *Section 04, Working with Databases Using Entity Framework Core*.
2. Add a new action method, as shown in the following code:

```

public IActionResult ProductsThatCostMoreThan(decimal? price)
{
    if (!price.HasValue)
    {
        return NotFound("You must pass a product price in the query string, for example, /Home/ProductsThatCostMoreThan?price=50");
    }
    IEnumerable<Product> model = db.Products
        .Include(p => p.Category)
        .Include(p => p.Supplier)
        .Where(p => p.UnitPrice > price);
    if (model.Count() == 0)
    {
        return NotFound(
            $"No products cost more than {price:C}.");
    }
    ViewData["MaxPrice"] = price.Value.ToString("C");
    return View(model); // pass model to view
}

```

3. Inside the Views/Home folder, add a new file named ProductsThatCostMoreThan.cshtml.
4. Modify the contents, as shown in the following code:

```

@model IEnumerable<Packt.Shared.Product>
@{
    string title =
        "Products That Cost More Than " + ViewData["MaxPrice"];
    ViewData["Title"] = title;
}
<h2>@title</h2>
<table class="table">
    <thead>
        <tr>
            <th>Category Name</th>
            <th>Supplier's Company Name</th>
            <th>Product Name</th>
            <th>Unit Price</th>
            <th>Units In Stock</th>
        </tr>
    </thead>
    <tbody>
        @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>
        }
    </tbody>
</table>

```

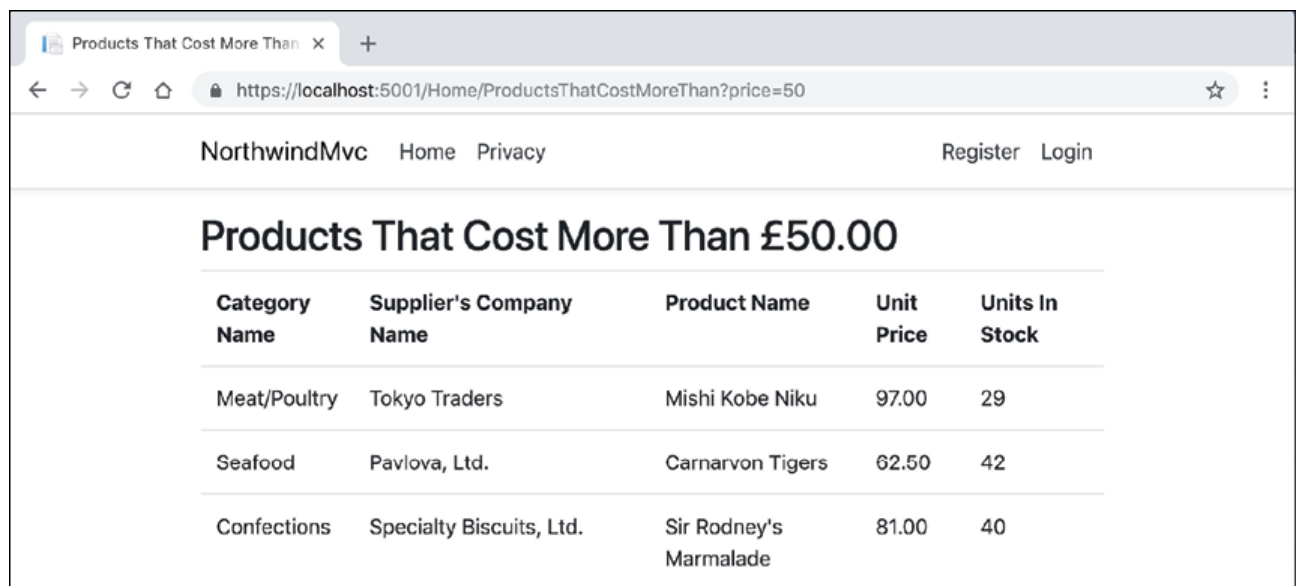
5. In the Views/Home folder, open Index.cshtml.
6. Add the following form element below the visitor count and above the Products heading and its listing of products. This will provide a form for the user to enter a price. The user can then click on the **Submit** button to call the action method that shows only products that cost more than the entered price:

```

<h3>Query products by price</h3>
<form asp-action="ProductsThatCostMoreThan" method="get">
    <input name="price" placeholder="Enter a product price" />
    <input type="submit" />
</form>

```

7. Start the website, use Chrome to navigate to the website, and on the home page, enter a price in the form, for example, 50, and then click on **Submit**. You will see a table of the products that cost more than the price that you entered, as shown in the following screenshot:



Category Name	Supplier's Company Name	Product Name	Unit Price	Units In Stock
Meat/Poultry	Tokyo Traders	Mishi Kobe Niku	97.00	29
Seafood	Pavlova, Ltd.	Carnarvon Tigers	62.50	42
Confections	Specialty Biscuits, Ltd.	Sir Rodney's Marmalade	81.00	40

Figure 16.11: A filtered list of products that cost more than £50

8. Close the browser and stop the website.

Improving scalability using asynchronous tasks

When building a desktop or mobile app, multiple tasks (and their underlying threads) can be used to improve responsiveness, because while one thread is busy with the task, another can handle interactions with the user.

Tasks and their threads can be useful on the server side too, especially with websites that work with files, or request data from a store or a web service that could take a while to respond. But they are detrimental to complex calculations that are CPU-bound, so leave these to be processed synchronously as normal.

When an HTTP request arrives at the web server, a thread from its pool is allocated to handle the request. But if that thread must wait for a resource, then it is blocked from handling any more incoming requests. If a website receives more simultaneous requests than it has threads in its pool, then some of those requests will respond with a server timeout error, 503 Service Unavailable.

The threads that are locked are not doing useful work. They *could* handle one of those other requests but only if we implement asynchronous code in our websites.

Whenever a thread is waiting for a resource it needs, it can return to the thread pool and handle a different incoming request, improving the scalability of the website, that is, increasing the number of simultaneous requests it can handle.

Why not just have a larger thread pool? In modern operating systems, every thread pool thread has a 1 MB stack. An asynchronous method uses a smaller amount of memory. It also removes the need to create new threads in the pool, which takes time. The rate at which new threads are added to the pool is typically one every two seconds, which is a loooooong time compared to switching between asynchronous threads.

Making controller action methods asynchronous

It is easy to make an existing action method asynchronous:

1. In the HomeController class, make sure that the System.Threading.Tasks namespace has been imported.
2. Modify the Index action method to be asynchronous, to return a Task<T>, and to await the calls to asynchronous methods to get the categories and products, as shown highlighted in the following code:

```
public async Task<IActionResult> Index()
{
    var model = new HomeIndexViewModel
    {
        VisitorCount = (new Random()).Next(1, 1001),
        Categories = await db.Categories.ToListAsync(),
        Products = await db.Products.ToListAsync()
    };
    return View(model); // pass model to view
}
```

3. Modify the ProductDetail action method in a similar way, as shown highlighted in the following code:

```
public async Task<IActionResult> ProductDetail(int? id)
{
    if (!id.HasValue)
    {
        return NotFound("You must pass a product ID in the route, for example, /Home/ProductDetail/21");
    }
}
```

```
var model = await db.Products
    .SingleOrDefaultAsync(p => p.ProductID == id);
if (model == null)
{
    return NotFound($"Product with ID of {id} not found.");
}
return View(model); // pass model to view and then return result
}
```

4. Start the website, use Chrome to navigate to the website, and note that the functionality of the website is the same, but trust that it will now scale better.
5. Close the browser and stop the website.

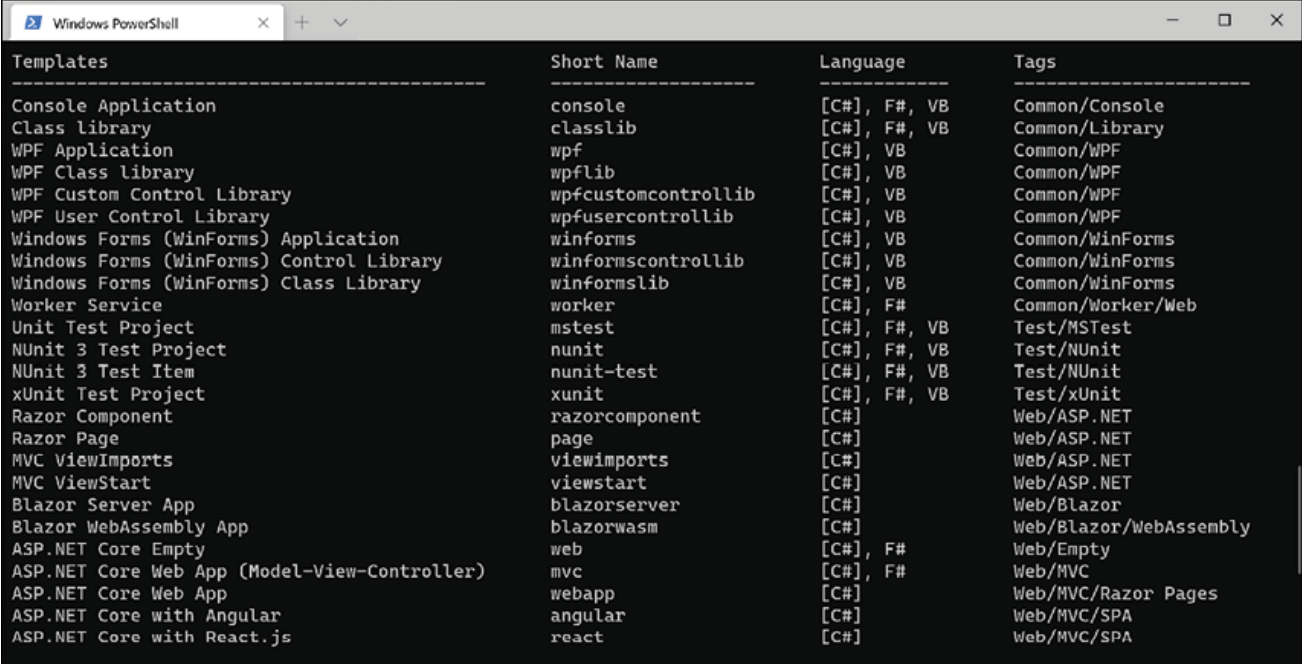
Using other project templates

When you install the .NET Core SDK, there are many project templates included:

1. In **TERMINAL**, enter the following command:

```
dotnet new --help
```

2. You will see a list of currently installed templates, including templates for Windows desktop development if you are running on Windows, as shown in the following screenshot:



Templates	Short Name	Language	Tags
Console Application	console	[C#], F#, VB	Common/Console
Class Library	classlib	[C#], F#, VB	Common/Library
WPF Application	wpf	[C#], VB	Common/WPF
WPF Class Library	wpflib	[C#], VB	Common/WPF
WPF Custom Control Library	wpfcustomcontrollib	[C#], VB	Common/WPF
WPF User Control Library	wpfusercontrollib	[C#], VB	Common/WPF
Windows Forms (WinForms) Application	winforms	[C#], VB	Common/WinForms
Windows Forms (WinForms) Control Library	winformscontrollib	[C#], VB	Common/WinForms
Windows Forms (WinForms) Class Library	winformslib	[C#], VB	Common/WinForms
Worker Service	worker	[C#], F#	Common/Worker/Web
Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
Blazor WebAssembly App	blazorwasm	[C#]	Web/Blazor/WebAssembly
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA

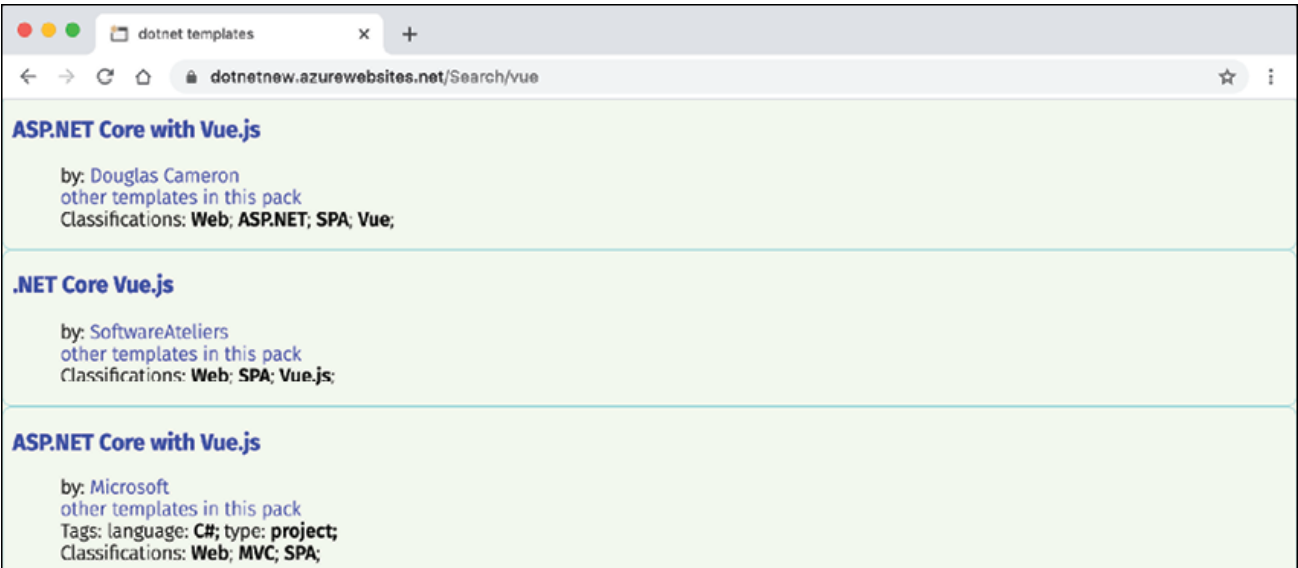
Figure 16.12: A list of project templates

3. Note the web-related project templates, including ones for creating SPAs using Blazor.

Installing additional template packs

Developers can install lots of additional template packs:

1. Start a browser and navigate to <http://dotnetnew.azurewebsites.net/>.
2. Enter `vue` in the textbox, click the **Search templates** button, and note the list of available templates for Vue.js, including one published by Microsoft, as shown in the following screenshot:



ASP.NET Core with Vue.js by: Douglas Cameron other templates in this pack Classifications: Web; ASP.NET; SPA; Vue;
.NET Core Vue.js by: SoftwareAteliers other templates in this pack Classifications: Web; SPA; Vue.js;
ASP.NET Core with Vue.js by: Microsoft other templates in this pack Tags: language: C#; type: project; Classifications: Web; MVC; SPA;

Figure 16.13: A list of Vue.js templates

3. Click on **ASP.NET Core with Vue.js** by Microsoft, and note the instructions for installing and using this template, as shown in the following command:

```
dotnet new --install "Microsoft.AspNetCore.SpaTemplates"
```

More Information: You can see more templates at the following link:

<https://github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new>

Understanding other communication technologies

The ASP.NET Core Web API is not the only Microsoft technology for implementing services or communicating between components of a distributed application. Although we will not cover these technologies in detail, you should be aware of what they can do and when they should be used.

Understanding Windows Communication Foundation (WCF)

In 2006, Microsoft released .NET Framework 3.0 with some major frameworks, one of which was **Windows Communication Foundation (WCF)**. It abstracted the business logic implementation of a service from the technology used to communicate with it. It heavily used XML configuration to declaratively define endpoints, including their address, binding, and contract (known as the ABCs of endpoints). Once you have understood how to do this, it is a powerful yet flexible technology.

Microsoft has decided not to officially port WCF to .NET Core, but there is a community-owned OSS project named **Core WCF** managed by the .NET Foundation. If you need to migrate an existing service from .NET Framework to .NET Core, or build a client to a WCF service, then you could use Core WCF. Be aware that it can never be a full port since parts of WCF are Windows-specific.

More Information: You can read more and download the Core WCF repository from the following link: <https://github.com/CoreWCF/CoreWCF>

Technologies like WCF allow for the building of distributed applications. A client application can make **remote procedure calls (RPC)** to a server application. Instead of using a port of WCF to do this, we could use an alternative RPC technology.

Understanding gRPC

gRPC is a modern open source high-performance RPC framework that can run in any environment.

More Information: You can read about gRPC at the following link: <https://grpc.io>

Like WCF, gRPC uses contract-first API development that supports language-agnostic implementations. You write the contracts using .proto files with their own language syntax and tools to convert them into various languages like C#. It minimizes network usage by using Protobuf binary serialization.

More Information: Microsoft officially supports gRPC with ASP.NET Core. You can learn how to use gRPC with ASP.NET Core at the following link: <https://docs.microsoft.com/en-us/aspnet/core/grpc/aspnetcore>

Exercises

Exercise 8.1.

The carousel on the index page shows *Categories*. When the 'View' button is clicked, the website returns a 404 Not Found error.

Instead of returning an error message, add an MVC Page to your NorthwindMvc application that displays:

- the category selected
- a description of that category
- a list of products that fall under that category
 - link those listed products to the *ProductDetail* page)

