Building Websites Using the Model-View-Controller Pattern

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

This chapter 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
- Improving performance and scalability using caching
- Querying a database and using display templates

Setting up an ASP.NET Core MVC website

ASP.NET Core Razor Pages are good 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 like Razor syntax, 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 on the website.
- **Views**: Razor View files, that is, `.cshtml` files, that render data in view models into HTML web pages. Razor Views are different from Razor Pages but they share the same file extension, `.cshtml`. When creating a Razor Page, it must have the `@page` directive at the top of its file. When creating a Razor View, you must *not* use the `@page` directive! If you do, the controller will not pass the model and it will be `null`, throwing a `NullReferenceException` when you try to access any of its members.
- **Controllers**: Classes that execute code when an HTTP request arrives at the web server. The controller methods usually instantiate a view model and pass that to a view in order to generate an HTTP response. This is returned to the web browser or other client that made the original request.

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

Creating an ASP.NET Core MVC website

You will use a project template to create an ASP.NET Core MVC website project that has a database for authenticating and authorizing users. Visual Studio 2022 defaults to using SQL Server LocalDB for the accounts database. Visual Studio Code (or more accurately the `dotnet` CLI tool) uses SQLite by default and you can specify a switch to use SQL Server LocalDB instead.

Let's see it in action:

1. Use your preferred code editor to open the `PracticalApps` solution.
2. Add an MVC website project with authentication accounts stored in a database, as defined in the following list:
   - Project template: **ASP.NET Core Web App (Model-View-Controller) [C#]** / `mvc`.
   - Project file and folder: `Northwind.Mvc`.
   - Solution file and folder: `PracticalApps`.
   - **Authentication type: Individual Accounts** / `--auth Individual`.
   - For Visual Studio 2022, leave all other options as their defaults, for example, HTTPS is enabled, and Docker is disabled.
   - For Visual Studio Code, in the `PracticalApps` solution folder, use `dotnet new mvc --auth Individual -o Northwind.Mvc` and `dotnet sln add Northwind.Mvc`.
   - For JetBrains Rider, right-click the `PracticalApps` solution, navigate to **Add | New Project...**, in the **New Project** dialog box, select **ASP.NET Core Web Application**, for **Type**, select **Web App (Model-View-Controller)**, and for **Auth**, select **Individual authentication**, and then click **Create**.
3. Build the `Northwind.Mvc` project.
4. At the command prompt or terminal, use the `help` switch to see other options for this project template, as shown in the following command:

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

5. Note the results, as shown in the following partial output:

```
ASP.NET Core Web App (Model-View-Controller) (C#)
Author: Microsoft
Description: A project template for creating an ASP.NET Core application
with example ASP.NET Core MVC Views and Controllers. This template can
also be used for RESTful HTTP services.
```

There are many options, especially related to authentication, as shown in *Table 1*:

Switches	**Description**
`-au` or `--auth`	The type of authentication to use: `None` (default): This choice also allows you to disable HTTPS. `Individual`: Individual authentication that stores registered users and their passwords in a database (SQLite by default). We will use this in the project we create for this chapter. `IndividualB2C`: Individual authentication with Azure AD B2C. `SingleOrg`: Organizational authentication for a single tenant. `MultiOrg`: Organizational authentication for multiple tenants. `Windows`: Windows authentication. Mostly useful for intranets.
`-uld` or `--use-local-db`	Whether to use SQL Server LocalDB instead of SQLite. This option only applies if `--auth Individual` or `--auth IndividualB2C` is specified. The value is an optional bool with a default of `false`.
`-rrc` or `--razor-runtime-compilation`	Determines if the project is configured to use Razor runtime compilation in Debug builds. This can improve the performance of the startup process during debugging because it can defer the compilation of Razor views. The value is an optional bool with a default of `false`.
`-f` or `--framework`	The target framework for the project. Values can be `net8.0` (default), `net7.0`, or `net6.0`. Older versions are no longer supported.

Table 1: Additional switches for the dotnet new mvc project template

Creating the authentication database for SQL Server LocalDB

If you created the MVC project using Visual Studio 2022, or you used `dotnet new mvc` with the `-uld` or `--use-local-db` switch, then the database for authentication and authorization will be stored in SQL Server LocalDB. But the database does not yet exist.

If you created the MVC project using `dotnet new` or JetBrains Rider, then the database for authentication and authorization will be stored in SQLite and the file has already been created, named `app.db`.

The connection string for the authentication database is named `DefaultConnection` and it is stored in the `appsettings.json` file in the root folder for the MVC website project.

For SQLite, see the following setting:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "DataSource=app.db;Cache=Shared"
  },
```

If you created the MVC project using Visual Studio 2022 only, then let's create its authentication database now by following a few simple steps:

1. In the `Northwind.Mvc` project, in `appsettings.json`, note the database connection string named `DefaultConnection`, as shown highlighted in the following configuration:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
Northwind.Mvc-440bc3c1-f7e7-4463-99d5-896b6a6500e0;Trusted_
Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

> Your database name will use the pattern `aspnet-[ProjectName]-[GUID]` and have a different GUID value from the example above.

2. At a command prompt or terminal, in the `Northwind.Mvc` folder, enter the command to run database migrations so that the database used to store credentials for authentication is created, as shown in the following command:

```
dotnet ef database update
```

3. Note the database is created with tables like `AspNetRoles`, as shown in the following partial output:

```
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 8.0.0 initialized 'ApplicationDbContext'
using provider 'Microsoft.EntityFrameworkCore.SqlServer:8.0.0' with
options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (129ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
```

```
      CREATE DATABASE [aspnet-Northwind.Mvc2-440bc3c1-f7e7-4463-99d5-
896b6a6500e0];
...
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      CREATE TABLE [AspNetRoles] (
          [Id] nvarchar(450) NOT NULL,
          [Name] nvarchar(256) NULL,
          [NormalizedName] nvarchar(256) NULL,
          [ConcurrencyStamp] nvarchar(max) NULL,
          CONSTRAINT [PK_AspNetRoles] PRIMARY KEY ([Id])
      );
...
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (8ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId],
[ProductVersion])
      VALUES (N'00000000000000_CreateIdentitySchema', N'8.0.0');
```

Changing the port numbers and starting the website

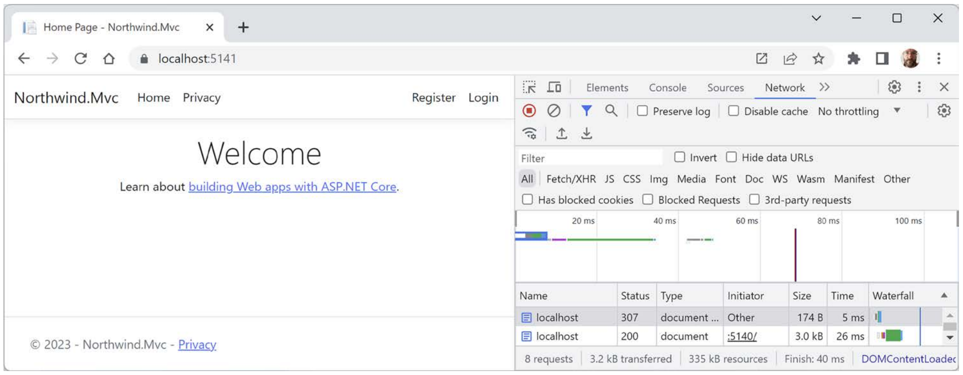
Let's review the behavior of the default ASP.NET Core MVC website project template:

1. In the `Northwind.Mvc` project, expand the `Properties` folder.
2. In `launchSettings.json`, change the configured port numbers for the `https` profile, as shown highlighted in the following configuration:

```
"applicationUrl": "https://localhost:5141;http://localhost:5140",
```

3. Save the changes to the `launchSettings.json` file.
4. Start the `Northwind.Mvc` website using the `https` launch profile:
    - If you are using Visual Studio 2022, then in the toolbar, select the **https** profile, select **Google Chrome** as the **Web Browser**, and then start the project without debugging.
    - If you are using Visual Studio Code, then enter the command to start the project with the `https` launch profile, as shown in the following command: `dotnet run --launch-profile https`, and then start Chrome.
5. In Chrome, open **Developer Tools**.

6. Navigate to `http://localhost:5140/` and note the following, as shown in *Figure 1*:
   - Requests for HTTP on port `5140` are automatically redirected to HTTPS on port `5141`.
   - The top navigation menu 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, **Northwind.Mvc**, shown in the header and footer:



Welcome

Learn about building Web apps with ASP.NET Core.

© 2023 - Northwind.Mvc - Privacy

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	307	document ...	Other	174 B	5 ms	
localhost	200	document	:5140/	3.0 kB	26 ms	

8 requests | 3.2 kB transferred | 335 kB resources | Finish: 40 ms | DOMContentLoadec

Figure 1: The ASP.NET Core MVC project template website home page

7. Leave the browser running.

Exploring visitor registration

By default, passwords must have at least one non-alphanumeric character, at least one digit (0-9), and 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 practices 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:

1. In Chrome, close the **Developer Tools** pane.
2. In the top navigation menu, click **Register**.
3. Enter an email and password, and then click the **Register** button. (I used `test@example.com` and `Pa$$w0rd`.)
4. On the **Register confirmation** page, 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 could customize. By default, the **Confirm email** page just says **Thank you for confirming your email.**
5. In the top navigation menu, click **Login**, enter your email and password (note that there is an optional check box 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.

6. In the top navigation menu, click your email address. This will navigate to an account management page. 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. This last feature is good for compliance with legal regulations like the European GDPR.
7. Close Chrome and shut down the web server.

Reviewing an MVC website project structure

In your code editor, in Visual Studio **Solution Explorer** (toggle on **Show All Files**), Visual Studio Code **EXPLORER**, or JetBrains Rider, hover your mouse in the **Solution** pane and click the eyeball icon, and then review the structure of an MVC website 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 temporary files needed during the build process and 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 also put images and other static file resources like PDF documents in here. The project template includes Bootstrap and jQuery libraries.
- `app.db`: This is the SQLite database that stores registered visitors. (If you used SQL Server LocalDB, then it will not be needed.)

- `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 Core Identity system and logging levels.
- `Northwind.Mvc.csproj`: This file contains project settings like the use of the web .NET SDK, an entry for SQLite 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 EF Core and ASP.NET Core Identity packages.
- `Northwind.Mvc.csproj.user`: This file contains Visual Studio 2022 session settings for remembering options. For example, which launch profile was selected, like `https`. Visual Studio 2022 hides this file, and it should not normally be included in source code control because it is specific to an individual developer.
- `Program.cs`: This file defines a hidden `Program` class that contains the `<Main>$` entry point. It builds a pipeline for processing incoming HTTP requests and hosts the website using default options like configuring the Kestrel web server and loading `appsettings`. It adds and configures services that your website needs, for example, ASP.NET Core Identity for authentication, SQLite or SQL Server for identity data storage, and so on, and routes for your application.

Reviewing the ASP.NET Core Identity database

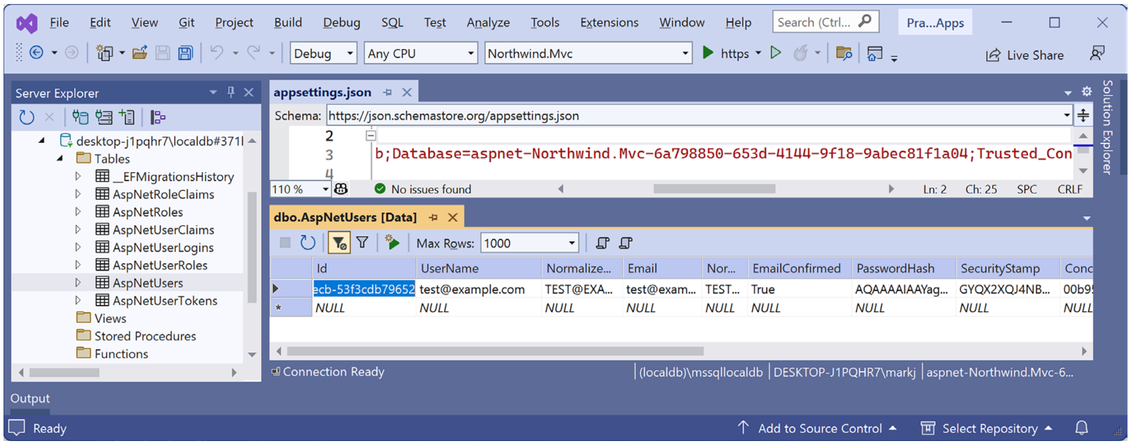
Open `appsettings.json` to find the connection string used for the ASP.NET Core Identity database, as shown highlighted for SQL Server LocalDB in the following markup:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
Northwind.Mvc-2F6A1E12-F9CF-480C-987D-FEFB4827DE22;Trusted_
Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

If you used SQL Server LocalDB for the identity data store, then you can use **Server Explorer** to connect to the database. You can copy and paste the connection string from the `appsettings.json` file. Remember to remove the second backslash between `(localdb)` and `mssqllocaldb`.

If you installed an SQLite tool such as SQLiteStudio, then you can open the SQLite `app.db` database file.

You can then 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 *Figure 2*:



```
b;Database=aspnet-Northwind.Mvc-6a798850-653d-4144-9f18-9abec81f1a04;Trusted_Con
```

Id	UserName	Normalize...	Email	Nor...	EmailConfirmed	PasswordHash	SecurityStamp	Conc
ecb-53f3cdb79652	test@example.com	TEST@EXA...	test@exam...	TEST...	True	AQAAAAIAAYag...	GYQX2XQJ4NB...	00b9
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 2: AspNetUsers table with the registered user

Exploring an ASP.NET Core MVC website

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

.NET 5 and earlier ASP.NET Core project templates used both a `Program` class and a `Startup` class to separate initialization and configuration, but with .NET 6 and later, Microsoft encourages putting everything in a single `Program.cs` file.

ASP.NET Core MVC initialization

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

1. In `Program.cs`, note that it can be divided into four important sections from top to bottom. As you review the sections, you might want to add regions to remind yourself of what each section is used for.
2. The first section imports some namespaces, as shown in the following code:

```
#region Import namespaces
using Microsoft.AspNetCore.Identity; // To use IdentityUser.
using Microsoft.EntityFrameworkCore; // To use UseSqlServer method.
using Northwind.Mvc.Data; // To use ApplicationDbContext.
#endregion
```

Remember that, by default, many other namespaces are imported using the implicit usings feature of .NET 6 and later. Build the project and then the globally imported namespaces can be found in the following file: `obj\Debug\net8.0\Northwind.Mvc.GlobalUsings.g.cs`.

3. The second section creates and configures a web host builder that does the following:

    - Registers an application database context using SQL Server or SQLite. The database connection string is loaded from the `appsettings.json` file.
    - Adds ASP.NET Core Identity for authentication and configures it to use the application database.
    - Adds support for MVC controllers with views, as shown in the following code:

    ```
    #region Configure the host web server including services

    var builder = WebApplication.CreateBuilder(args);

    // Add services to the container.
    var connectionString = builder.Configuration
      .GetConnectionString("DefaultConnection") ??
      throw new InvalidOperationException(
        "Connection string 'DefaultConnection' not found.");

    builder.Services.AddDbContext<ApplicationDbContext>(options =>
      options.UseSqlServer(connectionString)); // Or UseSqlite.

    builder.Services.AddDatabaseDeveloperPageExceptionFilter();

    builder.Services.AddDefaultIdentity<IdentityUser>(options =>
      options.SignIn.RequireConfirmedAccount = true)
      .AddEntityFrameworkStores<ApplicationDbContext>();

    builder.Services.AddControllersWithViews();

    var app = builder.Build();

    #endregion
    ```

    The `builder` object has two commonly used objects, `Configuration` and `Services`:

    - `Configuration` contains merged values from all the places you could set configuration: `appsettings.json`, environment variables, command-line arguments, and so on.
    - `Services` is a collection of registered dependency services.

    The call to `AddDbContext` is an example of registering a dependency service. ASP.NET Core implements the **dependency injection** (**DI**) design pattern so that other components like controllers can request needed services through their constructors. Developers register those services in this section of `Program.cs` (or if using a `Startup` class, then in its `ConfigureServices` method).

4. The third section configures the HTTP pipeline through which requests and responses flow in and out. It configures a relative URL path to run database migrations 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:

```
#region Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())
{
  app.UseMigrationsEndPoint();
}
else
{
  app.UseExceptionHandler("/Home/Error");
  // The default HSTS value is 30 days. You may want to change this for 
production scenarios, see https://aka.ms/aspnetcore-hsts.
  app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.MapRazorPages();

#endregion
```

We learned about most of these methods and features in *Chapter 13*, *Building Websites Using ASP.NET Core Razor Pages*.

Good Practice: What does the extension method `UseMigrationsEndPoint` do? You could read the official documentation, but it does not help much. For example, it does not tell us what relative URL path it defines by default: `https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.migrationsendpointextensions.usemigrationsendpoint`. Luckily, ASP.NET Core is open source, so we can read the source code and discover what it does, at the following link: `https://github.com/dotnet/aspnetcore/blob/main/src/Middleware/Diagnostics.EntityFrameworkCore/src/MigrationsEndPointOptions.cs#L18`. Get into the habit of exploring the source code for ASP.NET Core to understand how it works.

Apart from the `UseAuthorization` method, the most important new method in this section of `Program.cs` 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 topic.

Warning! Although we will not create any Razor Pages in this chapter, 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.

5. The fourth and final section has a thread-blocking method call that runs the website and waits for incoming HTTP requests to respond to, as shown in the following code:

```
#region Start the host web server listening for HTTP requests.
app.Run(); // This is a blocking call.
#endregion
```

The default MVC route

The responsibility of a route is to discover the name of a controller class to instantiate and 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 the defaults of `Home` for the controller and `Index` for the action (the = assignment sets a default for a named segment).

Table 2 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

Table 2: Example URLs mapped via the default route

Controllers and actions

In MVC, the C stands for *controller*. From the route and an incoming URL, ASP.NET Core 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
  {
...
```

The ControllerBase class

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 *Chapter 15*, *Building and Consuming Web Services*.

`ControllerBase` has many useful properties for working with the current HTTP context, as shown in *Table 3*:

Property	Description
`Request`	Just the HTTP request, for example, headers, query string parameters, the body of the request as a stream that you can read from, the content type and length, and cookies.
`Response`	Just the HTTP response, for example, headers, the body of the response as a stream that you can write to, the content type and length, status code, and cookies. There are also delegates like `OnStarting` and `OnCompleted` that you can hook a method up to.
`HttpContext`	Everything about the current HTTP context, including the request and response, information about the connection, a collection of features that have been enabled on the server with middleware, and a `User` object for authentication and authorization.

Table 3: Useful properties for working with the current HTTP context

The Controller class

Microsoft provides another class named `Controller` that your classes can inherit from if they need view support, as shown in the following code:

```
namespace Microsoft.AspNetCore.Mvc
{
  //
  // Summary:
  // A base class for an MVC controller with view support.
  public abstract class Controller : ControllerBase,
    IActionFilter, IFilterMetadata, IAsyncActionFilter, IDisposable
  {
...
```

`Controller` has many useful properties for working with views, as shown in *Table 4*:

Property	Description
`ViewData`	A dictionary in which the controller can store key/value pairs that is accessible in a view. The dictionary's lifetime is only for the current request/response.
`ViewBag`	A dynamic object that wraps the `ViewData` to provide a friendlier syntax for setting and getting dictionary values.
`TempData`	A dictionary in which the controller can store key/value pairs that is accessible in a view. The dictionary's lifetime is for the current request/response and the next request/response for the same visitor session. This is useful for storing a value during an initial request, responding with a redirect, and then reading the stored value in the subsequent request.

Table 4: Useful properties for working with views

`Controller` has many useful methods for working with views, as shown in *Table 5*:

Method	Description
`View`	Returns a `ViewResult` after executing a view that renders a full response, for example, a dynamically generated web page. The view can be selected using a convention or be specified with a string name. A model can be passed to the view.
`PartialView`	Returns a `PartialViewResult` after executing a view that is part of a full response, for example, a dynamically generated chunk of HTML. The view can be selected using a convention or be specified with a string name. A model can be passed to the view.
`ViewComponent`	Returns a `ViewComponentResult` after executing a component that dynamically generates HTML. The component must be selected by specifying its type or its name. An object can be passed as an argument.
`Json`	Returns a `JsonResult` containing a JSON-serialized object. This can be useful for implementing a simple Web API as part of an MVC controller that primarily returns HTML for a human to view.

Table 5: Useful methods for working with views

The responsibilities of a controller

The responsibilities of a controller are as follows:

- Identify the services that the controller needs 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.

Good Practice: Controllers should be *thin*, meaning they only perform the above-listed activities but do not implement any business logic. All business logic should be implemented in services that the controller calls when needed.

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:

    - Extra namespaces are imported, which I have added comments for to show which types they are needed for.
    - A private read-only 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 `IActionResult` 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:

```
using Microsoft.AspNetCore.Mvc; // To use Controller,
IActionResult.
using Northwind.Mvc.Models; // To use ErrorViewModel.
using System.Diagnostics; // To use Activity.

namespace Northwind.Mvc.Controllers;

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 });
    }
  }
```

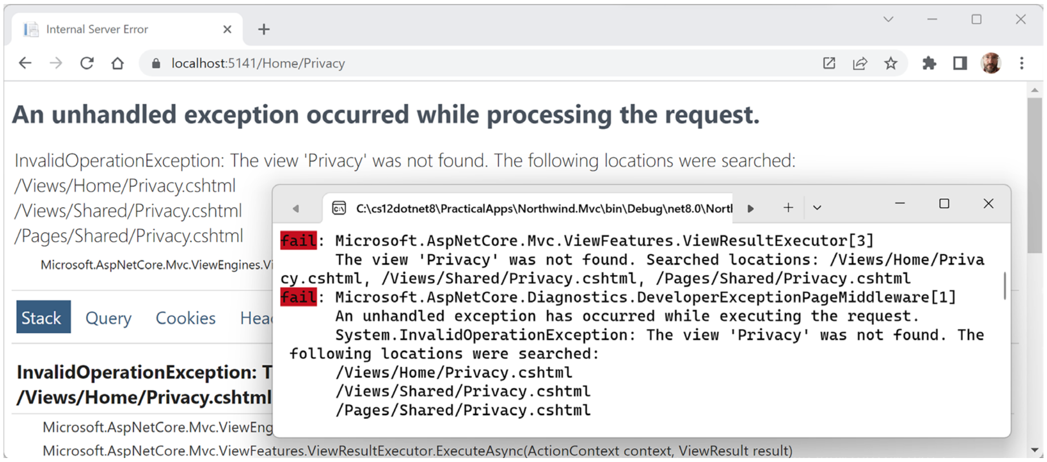
If the visitor navigates to a path of `/` or `/Home`, then it is the equivalent of `/Home/Index` because those were the default names for the controller and action in the default route.

The view search path convention

The `Index` and `Privacy` methods are identical in 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.

Let's deliberately break one of the page names so that we can see the paths searched by default:

1. In the `Northwind.Mvc` project, expand the `Views` folder and then the `Home` folder.
2. Rename the `Privacy.cshtml` file to `Privacy2.cshtml`.
3. Start the `Northwind.Mvc` website project using the `https` launch profile.
4. Start Chrome, navigate to `https://localhost:5141/`, click **Privacy**, and note the paths that are searched for a view to render the web page (including in `Shared` folders for MVC views and Razor Pages) in the exception in both the browser and the command prompt or terminal output, as shown in *Figure 3*:


Internal Server Error
localhost:5141/Home/Privacy
Stack
Query
Cookies
Hea
C:\cs12dotnet8\PracticalApps\Northwind.Mvc\bin\Debug\net8.0\Nortl

An unhandled exception occurred while processing the request.

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

Microsoft.AspNetCore.Mvc.ViewEngines.Vi

InvalidOperationException: T
/Views/Home/Privacy.cshtml

Microsoft.AspNetCore.Mvc.ViewEng
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewResultExecutor.ExecuteAsync(ActionContext context, ViewResult result)

```
fail: Microsoft.AspNetCore.Mvc.ViewFeatures.ViewResultExecutor[3]
      The view 'Privacy' was not found. Searched locations: /Views/Home/Priva
cy.cshtml, /Views/Shared/Privacy.cshtml, /Pages/Shared/Privacy.cshtml
fail: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware[1]
      An unhandled exception has occurred while executing the request.
      System.InvalidOperationException: The view 'Privacy' was not found. The
 following locations were searched:
      /Views/Home/Privacy.cshtml
      /Views/Shared/Privacy.cshtml
      /Pages/Shared/Privacy.cshtml
```

Figure 3: An exception showing the default search path for views

5. Close Chrome and shut down the web server.
6. Rename the `Privacy2.cshtml` file back to `Privacy.cshtml`.

You have now seen the view search path convention, as 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`

Logging using the dependency service

You have just seen that some errors are caught and written to the console. You can write your own messages to the console in the same way by using the logger.

1. In the `Controllers` folder, in `HomeController.cs`, in the `Index` method, add statements before the `return` statement to use the logger to write some messages of various levels to the console, as shown highlighted in the following code:

```
public IActionResult Index()
{
  _logger.LogError("This is a serious error (not really!)");
  _logger.LogWarning("This is your first warning!");
  _logger.LogWarning("Second warning!");
  _logger.LogInformation("I am in the Index method of the
HomeController.");

  return View();
}
```

2. Start the `Northwind.Mvc` website project using the `https` launch profile.
3. Start Chrome and navigate to the home page for the website.
4. At the command prompt or terminal, note the messages, as shown in the following output:

```
fail: Northwind.Mvc.Controllers.HomeController[0]
      This is a serious error (not really!)
warn: Northwind.Mvc.Controllers.HomeController[0]
      This is your first warning!
warn: Northwind.Mvc.Controllers.HomeController[0]
      Second warning!
info: Northwind.Mvc.Controllers.HomeController[0]
      I am in the Index method of the HomeController.
```

5. Close Chrome and shut down the web server.

More Information: You can learn a lot more about ASP.NET Core logging at the following link: `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/`.

Using entity and view models

In MVC, the M stands for *model*. Models represent the data required to respond to a request. There are two types of models commonly used: entity models and view models.

Entity models represent entities in a database like SQL Server or SQLite. Based on the request, one or more entities might need to be retrieved from data storage. Entity models are defined using classes, since they might need to change and then be used to update the underlying data store.

All the data that we want to show in response to a request is the **MVC model**, sometimes called a **view model**, because it is a model that is passed into a view for rendering into a response format like HTML or JSON. View models should be immutable, so they are commonly defined using records.

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 route 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 a 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 *Chapter 12*, *Introducing Web Development Using ASP.NET Core*:

1. In the `Northwind.Mvc` project, add a project reference to `Northwind.DataContext` for either SQLite or SQL Server, as shown in the following markup:

```
<ItemGroup>
  <!-- change Sqlite to SqlServer if you prefer -->
  <ProjectReference Include=
"..\Northwind.DataContext.Sqlite\Northwind.DataContext.Sqlite.csproj" />
</ItemGroup>
```

2. If you are using SQL Server, then add a package reference for ADO.NET for SQL Server, as shown in the following markup:

```
<PackageReference
  Include="Microsoft.Data.SqlClient" Version="5.1.1" />
```

3. Build the `Northwind.Mvc` project to compile its dependencies.
4. If you are using SQL Server, or if you might want to switch to SQL Server from SQLite, then in `appsettings.json`, add a connection string for the Northwind database using SQL Server, as shown highlighted in the following markup:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
Northwind.Mvc-DC9C4FAF-DD84-4FC9-B925-69A61240EDA7;Trusted_
Connection=True;MultipleActiveResultSets=true",
    "NorthwindConnection": "Server=.;Database=Northwind;Integrated
Security=True;MultipleActiveResultSets=true;TrustServerCertificate=true;"
  },
```

Modify the connection string to match wherever your Northwind database is, for example, in Azure SQL Database in the cloud or Azure SQL Edge in Docker, for example. If you have to use SQL Server authentication, do not store the user and password in this file! You will set them from environment variables in code.

5. In `Program.cs`, import the namespace to work with your entity model types, as shown in the following code:

```
using Northwind.EntityModels; // To use AddNorthwindContext method.
```

6. Before the `builder.Build` method call, add statements to load the appropriate connection string and then to register the `Northwind` database context, as shown in the following code:

```
// If you are using SQLite, default is "..\Northwind.db".
builder.Services.AddNorthwindContext();

/*
// If you are using SQL Server.
string? sqlServerConnection = builder.Configuration
  .GetConnectionString("NorthwindConnection");

if (sqlServerConnection is null)
{
  Console.WriteLine("SQL Server database connection string is missing!");
}
else
{
  // If you are using SQL Server authentication then disable
  // Windows Integrated authentication and set user and password.
  Microsoft.Data.SqlClient.SqlConnectionStringBuilder sql =
    new(sqlServerConnection);

  sql.IntegratedSecurity = false;
  sql.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
  sql.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

  builder.Services.AddNorthwindContext(sql.ConnectionString);
}
*/
```

7. In the `Models` folder, add a class file named `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.

8. In `HomeIndexViewModel.cs`, add statements to define a record that has three properties for a count of the number of visitors, and lists of categories and products, as shown in the following code:

```
using Northwind.EntityModels; // To use Category, Product.

namespace Northwind.Mvc.Models;

public record HomeIndexViewModel(int VisitorCount,
  IList<Category> Categories, IList<Product> Products);
```

9. In `HomeController.cs`, import the `Northwind.EntityModels` namespace, as shown in the following code:

```
using Northwind.EntityModels; // To use NorthwindContext.
```

10. 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 readonly NorthwindContext _db;

  public HomeController(ILogger<HomeController> logger,
    NorthwindContext db)
  {
    _logger = logger;
    _db = db;
  }
```

ASP.NET Core will use constructor parameter injection to pass an instance of the `NorthwindContext` database context using the connection string you specified in `Program.cs`.

11. In the `Index` action method, 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 1,000, and using the `Northwind` database to get lists of categories and products, and then pass the model to the view, as shown highlighted in the following code:

```
  HomeIndexViewModel model = new
  (
    VisitorCount: Random.Shared.Next(1, 1001),
    Categories: _db.Categories.ToList(),
    Products: _db.Products.ToList()
  );

  return View(model); // Pass the model to the view.
}
```

Remember the view search convention: 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 views that match the action method name in the `Shared` folder and for Razor Pages in the `Pages` folder.

Implementing views

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

```
@using Northwind.Mvc
@using Northwind.Mvc.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"] – Northwind.Mvc</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" />
```

In *Chapter 13*, *Building Websites Using ASP.NET Core Razor Pages*, we used the public link to the Bootstrap website to reference version 5.3. The MVC project template uses a local copy of Bootstrap version 5.1. This might have been updated to a later version by the time you read this.

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 justify-content-between">
```

```
    <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>
    <partial name="_LoginPartial" />
  </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, like the `employees` component that you created in the previous chapter, 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 injects the contents of a specific Razor view for a page like the `Index.cshtml` file at that point in the shared layout.

11. Note the rendering of `<script>` elements at the bottom of the page so that it does not 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.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js">
</script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("Scripts", required: false)
```

12. In `_LoginPartial.cshtml`, note the login functionality is implemented using the ASP.NET Core Identity system as Razor Pages using an `asp-area` named `Identity`, as shown in the following markup:

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
```

```
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
  @if (SignInManager.IsSignedIn(User))
  {
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="Identity"
         asp-page="/Account/Manage/Index" title="Manage">
      Hello @User.Identity?.Name!</a>
    </li>
    <li class="nav-item">
      <form class="form-inline" asp-area="Identity"
            asp-page="/Account/Logout"
            asp-route-returnUrl="@Url.Action("Index", "Home",
            new { area = "" })">
        <button type="submit" class="nav-link btn
                btn-link text-dark">Logout</button>
      </form>
    </li>
  }
  else
  {
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="Identity"
         asp-page="/Account/Register">Register</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="Identity"
         asp-page="/Account/Login">Login</a>
    </li>
  }
</ul>
```

13. In `_ValidationScriptsPartial.cshtml`, note this partial view has references to a pair of jQuery scripts for performing validation, as shown in the following markup:

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></
script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.
unobtrusive.min.js"></script>
```

Good Practice: If you create a Razor View that uses a model with validation attributes like `[Required]` and `[StringLength]`, then add this partial view to the `Scripts` block to enable validation on the client side by the browser, as shown in the following markup:

```
@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

How cache busting with Tag Helpers works

When `asp-append-version` is specified with a `true` value in a `<link>`, `<img>`, or `<script>` element, the Tag Helper for that tag type is invoked.

They work by automatically appending a query string value named `v` that is generated from a SHA256 hash of the referenced source file, as shown in the following example generated output:

```
<script src="~/js/site.js? v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM"></
script>
```

You can see this for yourself in the current project because the `_Layout.cshtml` file has the `<script src="~/js/site.js" asp-append-version="true"></script>` element.

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.

The `src` attribute must be set to a static file stored on the local web server, usually in the `wwwroot` folder, but you can configure additional locations. Remote references are not supported.

Customizing an ASP.NET Core MVC website

Now that you've reviewed the structure of a basic MVC website, you will customize and extend it. You have already registered an EF Core model for the `Northwind` database, so the next task is to output some of that data on the home page.

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 `product-columns` ID, as shown in the following code:

```
#product-columns
{
  column-count: 3;
}
```

Setting up the category images

The Northwind database includes a table of eight 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 named `category1.jpeg`, `category2.jpeg`, and so on, up to `category8.jpeg`.

You can download images from the GitHub repository for this book at the following link: `https://github.com/markjprice/cs12dotnet8/tree/main/code/images/Categories`

Razor syntax and expressions

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

```
@{
  Order order = new()
  {
    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 statements to import the namespace for Northwind entities and set the model type to use the `HomeIndexViewModel`, as shown in the following code:

   ```
   @using Northwind.EntityModels
   @model HomeIndexViewModel
   ```

   Now, whenever we type `Model` in this view, your code editor will know the correct type for the model and will provide IntelliSense for it.

   While entering code in a view, remember the following:

   - Declare the type for the model using `@model` (with a lowercase m).
   - Interact with the instance of the model using `@Model` (with an 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, as shown highlighted in the following markup:

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

4. Under the existing `<div>` element, after its closing `</div>`, add new markup to output categories in a carousel and products as an unordered list, as shown in the following markup:

   ```
   @if (Model is not null)
   {
   <div id="categories" class="carousel slide" data-bs-ride="carousel"
        data-bs-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-bs-target="#categories" data-bs-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="/home/categorydetail/
  @Model.Categories[c].CategoryId">View</a>
          </p>
        </div>
      </div>
    }
    </div>
    <a class="carousel-control-prev" href="#categories"
      role="button" data-bs-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-bs-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>
      @if (Model is not null)
      {
      <h2>Products</h2>
      <div id="product-columns">
        <ul class="list-group">
        @foreach (Product p in @Model.Products)
        {
          <li class="list-group-item d-flex justify-content-between align-
  items-start">
            <a asp-controller="Home" asp-action="ProductDetail"
               asp-route-id="@p.ProductId" class="btn btn-outline-primary">
              <div class="ms-2 me-auto">@p.ProductName</div>
              <span class="badge bg-primary rounded-pill">
                @(p.UnitPrice is null ? "zero" : p.UnitPrice.Value.
  ToString("C"))
              </span>
            </a>
          </li>
        }
        </ul>
      </div>
      }
    </div>
  </div>
```

While reviewing the preceding Razor markup, note the following:

- JetBrains Rider might tell you that `Model` is never `null`, so you do not need to check for `null`. Visual Studio 2022 will warn you the opposite, which is why I put in the `null` check. Unfortunately, it is a common programmer error to pass an object for the model that is `null`.
- It is easy to mix static HTML elements such as `<ul>` and `<li>` with C# code to output the carousel of categories and the list of product names.
- The `<div>` element with the `id` attribute of `product-columns` will use the custom style that we defined earlier, so all the content in that element will display in three columns.
- The `<img>` 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 `HomeController` and its `ProductDetail` action method. This action method does not exist yet, but you will add it later in this chapter. 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:5141/Home/ProductDetail/43`.

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

1. Start the `Northwind.Mvc` website project using the `https` launch profile.
2. 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 *Figure 4*:

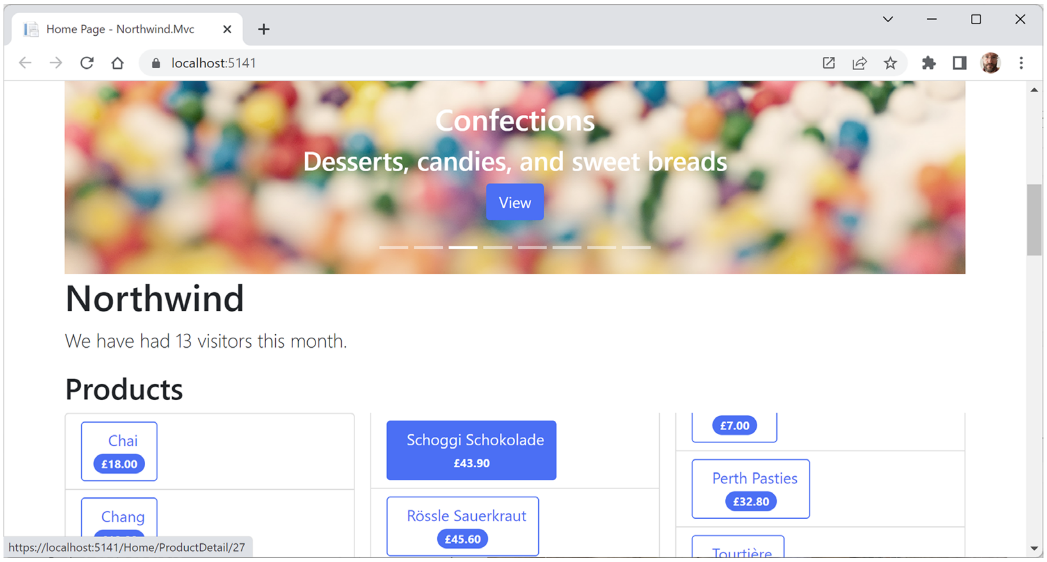


Figure 4: The updated Northwind MVC website home page

For now, clicking on any of the categories or product links gives **404 Not Found** errors, so let's see how we can implement pages that use the parameters passed to them to see the details of a product or category.

3. Close Chrome and shut down the web server.

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 `HomeController`, import the namespace for calling the `Include` extension method so that we can get related entities, as shown in the following code:

```
using Microsoft.EntityFrameworkCore; // To use Include method.
```

2. Add an action method named `ProductDetail`, as shown in the following code:

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

  Product? model = _db.Products.Include(p => p.Category)
    .SingleOrDefault(p => p.ProductId == id);

  if (model is null)
  {
    return NotFound($"ProductId {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 `BadRequest` method to return a `400` 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` value and include the related category information so we can see its name.
- 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.

3. In the `Views/Home` folder, add a new Razor View file named `ProductDetail.cshtml`. (In Visual Studio, the item template is named **Razor View - Empty**. In Visual Studio Code, at the command prompt or terminal, enter `dotnet new view -n ProductDetail`.)

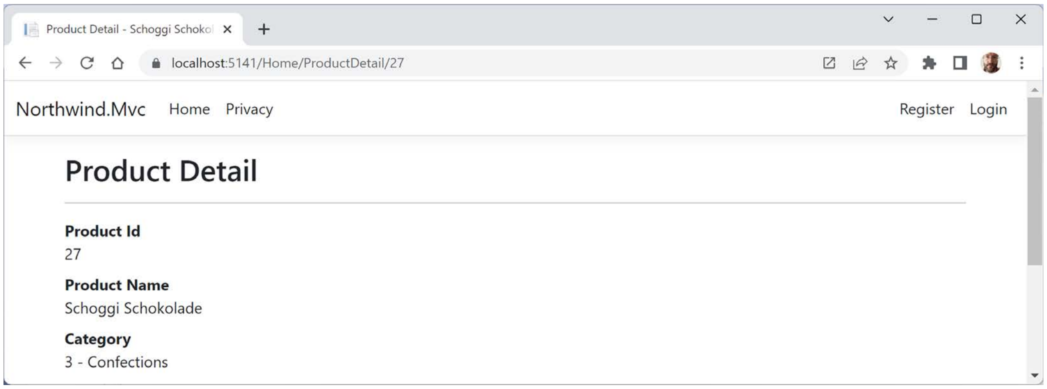
   Warning! Be careful not to add a Razor Page. If you do, then the file will have an `@page` directive at the top, which will prevent the model from being passed from the controller to the view and you will get a `NullReferenceException`!

4. Modify the contents, as shown in the following markup:

```
@model Northwind.EntityModels.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</dt>
    <dd>@Model.CategoryId - @Model.Category?.CategoryName</dd>
    <dt>Unit Price</dt>
    <dd>@(Model.UnitPrice is null ? "zero" :
          Model.UnitPrice.Value.ToString("C"))</dd>
    <dt>Units In Stock</dt>
    <dd>@Model.UnitsInStock</dd>
  </dl>
</div>
```

5. Start the `Northwind.Mvc` project using the `https` launch profile.
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 *Figure 5*:


Product Detail - Schoggi Schoko
localhost:5141/Home/ProductDetail/27
Northwind.Mvc
Home
Privacy
Register
Login

Product Detail

Product Id
27

Product Name
Schoggi Schokolade

Category
3 - Confections

Figure 5: The product details page for Schoggi Schokolade

8. View **Developer Tools**.
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.
10. Close Chrome and shut down the web server.

Disambiguating action methods

Although the C# compiler can differentiate between the two methods by noting that the signatures are different, from the point of view of routing an HTTP request, 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`. You do this by decorating the action method with one of the following attributes: `[HttpPost]`, `[HttpPut]`, and so on.

Model binders in detail

Model binders are a powerful yet easy way to set parameters of action methods based on values passed in an HTTP request, 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 used 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" value="2" />
  <input type="submit" />
</form>
```

Model binders can populate almost any type:

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

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 create a somewhat artificial example to illustrate what can be achieved using the default model binder and what we can do with the model state by applying some validation rules to the bound model and showing invalid data messages in the view:

1. In the `Models` folder, add a new file named `Thing.cs`.
2. Modify the contents to define a record with three properties; a nullable integer named `Id`, a `string` named `Color`, and a `string` named `Email`, each with appropriate validation attributes, as shown in the following code:

```
// To use [Range], [Required], [EmailAddress].
using System.ComponentModel.DataAnnotations;

namespace Northwind.Mvc.Models;

public record Thing(
  [Range(1, 10)] int? Id,
  [Required] string? Color,
  [EmailAddress] string? Email
);
```

3. In the `Models` folder, add a new class file named `HomeModelBindingViewModel.cs`.
4. Modify its contents to define a record with properties to store the bound model, a flag to indicate that there are errors, and a sequence of error messages, as shown in the following code:

```
namespace Northwind.Mvc.Models;

public record HomeModelBindingViewModel(Thing Thing, bool HasErrors,
  IEnumerable<string> ValidationErrors);
```

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

```
// This action method will handle GET and other requests except POST.
public IActionResult ModelBinding()
{
  return View(); // The page with a form to submit.
}

[HttpPost] // This action method will handle POST requests.
public IActionResult ModelBinding(Thing thing)
{
  HomeModelBindingViewModel model = new(
    Thing: thing, HasErrors: !ModelState.IsValid,
    ValidationErrors: ModelState.Values
      .SelectMany(state => state.Errors)
      .Select(error => error.ErrorMessage)
  );

  return View(model); // Show the model bound thing.
}
```

The first `ModelBinding` action method will implicitly be used for all other types of HTTP requests, like `GET`, `PUT`, `DELETE`, and so on, because the second `ModelBinding` action method is decorated with `[HttpPost]`.

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

```
@model 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?id=3">
  <input name="color" value="Red" />
  <input name="email" value="test@example.com" />
  <input type="submit" />
```

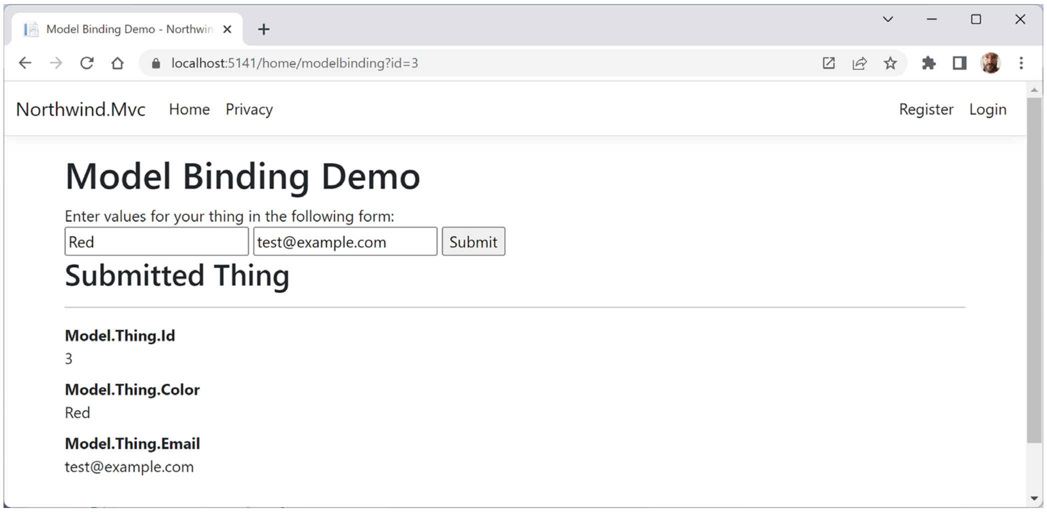
```
</form>
@if (Model is not null)
{
<h2>Submitted Thing</h2>
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Model.Thing.Id</dt>
    <dd>@Model.Thing.Id</dd>
    <dt>Model.Thing.Color</dt>
    <dd>@Model.Thing.Color</dd>
    <dt>Model.Thing.Email</dt>
    <dd>@Model.Thing.Email</dd>
  </dl>
</div>
  @if (Model.HasErrors)
  {
  <div>
    @foreach(string errorMessage in Model.ValidationErrors)
    {
      <div class="alert alert-danger" role="alert">@errorMessage</div>
    }
  </div>
  }
}
@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

8. In `Views/Home`, in `Index.cshtml`, in the first `<div>`, after rendering the heading, add a new paragraph with a link to the model binding page, as shown in the following markup:

```
<p><a asp-action="ModelBinding" asp-controller="Home">Binding</a></p>
```

9. Start the `Northwind.Mvc` website project using the `https` launch profile.
10. On the home page, click **Binding**.

11. Click the **Submit** button and note the value for the `Id` property is set from the query string parameter in the `action` of the form, and the values for the color and email properties are set from the form elements, as shown in *Figure 6*:


Model Binding Demo - Northwin
localhost:5141/home/modelbinding?id=3
Northwind.Mvc
Home
Privacy
Register
Login

Model Binding Demo

Enter values for your thing in the following form:

Red | test@example.com | Submit

Submitted Thing

Model.Thing.Id
3

Model.Thing.Color
Red

Model.Thing.Email
test@example.com

Figure 6: The Model Binding Demo page

12. Close Chrome and shut down the web server.

Passing a route parameter

Now we will set the property using a route parameter:

1. In the `Views\Home` folder, in `ModelBinding.cshtml`, modify the action for the form to pass the value `2` as an MVC route parameter, as shown highlighted in the following markup:

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

2. Start the `Northwind.Mvc` website project using the `https` launch profile.
3. On the home page, click **Binding**.
4. Click the **Submit** button and note the value for the `Id` property is set from the route parameter.
5. Close Chrome and shut down the web server.

Passing a form parameter

Now we will set the property using a form parameter:

1. In the `Views\Home` folder, in `ModelBinding.cshtml`, modify the action for the form to pass the value 1 as a form element parameter, as shown highlighted in the following markup:

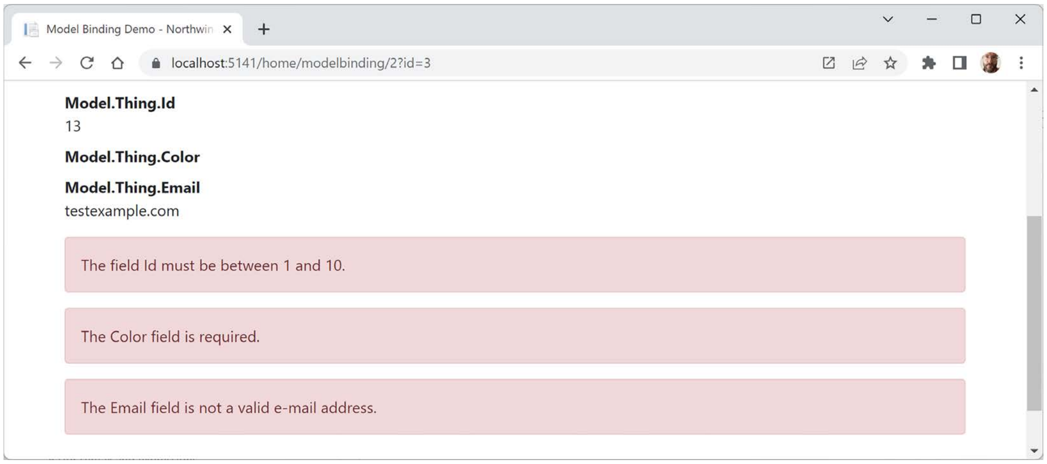
```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
```

```
    <input type="submit" />
  </form>
```

2. Start the `Northwind.Mvc` website project using the `https` launch profile.
3. On the home page, click **Binding**.
4. Click the **Submit** button and note the values for all the properties are both set from the form element parameters.

Good Practice: If you have multiple parameters with the same name, then remember that form parameters have the highest priority and query string parameters have the lowest priority for automatic model binding.

5. Enter an `Id` of `13`, clear the color textbox, delete the `@` from the email address, click the **Submit** button, and note the error messages, as shown in *Figure 7*:


Model Binding Demo - Northwin
localhost:5141/home/modelbinding/2?id=3

Model.Thing.Id
13
Model.Thing.Color
Model.Thing.Email
testexample.com

The field Id must be between 1 and 10.

The Color field is required.

The Email field is not a valid e-mail address.

Figure 7: The Model Binding Demo page with field validations

6. Close Chrome and shut down the web server.

Good Practice: What regular expression does Microsoft use for the implementation of the `EmailAddress` validation attribute? Find out at the following link: `https://github.com/microsoft/referencesource/blob/5697c29004a34d80acdaf5742d7e699022c64ecd/System.ComponentModel.DataAnnotations/DataAnnotations/EmailAddressAttribute.cs#L54`

Defining views with HTML Helper methods

While creating a view for ASP.NET Core MVC, you can use the `Html` object and its methods to generate markup. When Microsoft first introduced ASP.NET MVC in 2009, these HTML Helper methods were the way to programmatically render HTML.

Modern ASP.NET Core retains these HTML Helper methods for backward compatibility and provides Tag Helpers, which are usually easier to read and write in most scenarios. But there are notable situations where Tag Helpers cannot be used, like in Razor components.

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. For example, `Html.ActionLink(linkText: "Binding", actionName: "ModelBinding", controllerName: "Home")` would generate `<a href="/home/modelbinding">Binding</a>`. You can achieve the same result using the anchor tag helper `<a asp-action="ModelBinding" asp-controller="Home">Binding</a>`.
- `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.
- `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 `"&lt;script&gt;"`. This is not normally necessary since the Razor `@` symbol encodes string values by default.
- `Raw`: Use this to render a string value *without* encoding it as HTML.
- `PartialAsync` and `RenderPartialAsync`: Use these to generate HTML markup for a partial view. You can optionally pass a model and view data.

Defining views with Tag Helpers

Tag Helpers make it easier to make static HTML elements dynamic. The markup is cleaner and easier to read, edit, and maintain than if you use HTML Helpers.

However, Tag Helpers do not replace HTML Helpers because there are some things that can only be achieved with HTML Helpers, like rendering output that contains multiple nested tags. Tag Helpers also cannot be used in Razor components. So, you must learn how to use HTML Helpers and treat Tag Helpers as an optional choice that is better in some scenarios.

Tag Helpers are especially useful for **Front End** (**FE**) developers who primarily work with HTML, CSS, and JavaScript because the FE developer does not have to learn C# syntax. Tag Helpers just use what look like normal HTML attributes on elements.

The attribute names and values can also be selected from IntelliSense if your code editor supports that; both Visual Studio 2022 and Visual Studio Code do.

For example, to render a linkable hyperlink to a controller action, you could use an HTML Helper method, as shown in the following markup:

```
@Html.ActionLink("View our privacy policy.", "Privacy", "Index")
```

To make it clearer how it works, you could use named parameters:

```
@Html.ActionLink(linkText: "View our privacy policy.",
  action: "Privacy", controller: "Index")
```

But using a Tag Helper would be even clearer and cleaner for someone who works with HTML a lot:

```
<a asp-action="Privacy" asp-controller="Home">View our privacy policy.</a>
```

All three examples above generate the following rendered HTML element:

```
<a href="/home/privacy">View our privacy policy.</a>
```

Cross-functional filters

When you need to add some functionality to multiple controllers and actions, 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 an action method with the attribute. This will only affect that one action method.
- At the controller level, by decorating the controller class with the attribute. This will affect all methods of the controller.
- At the global level, by adding the attribute type to the `Filters` collection of the `MvcOptions` instance, which can be used to configure MVC when calling the `AddControllersWithViews` method, as shown in the following code:

  ```
  builder.Services.AddControllersWithViews(options =>
    {
      options.Filters.Add(typeof(MyCustomFilter));
    });
  ```

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, showing the privacy page currently requires the following URL path, which specifies both the controller and action:

```
https://localhost:5141/home/privacy
```

We could make the route simpler, as shown in the following link:

```
https://localhost:5141/private
```

Let's see how to do that:

1. In `HomeController.cs`, add an attribute to the `Privacy` method to define a simplified route, as shown highlighted in the following code:

   ```
   [Route("private")]
   public IActionResult Privacy()
   ```

2. Start the `Northwind.Mvc` website project using the `https` launch profile.
3. In the address bar, enter the following URL path, and note that the simplified path shows the **Privacy** page:

   ```
   https://localhost:5141/private
   ```

4. Close Chrome and shut down the web server.

Improving performance and scalability using caching

Figure 8 is a summary diagram of types and locations of caching:

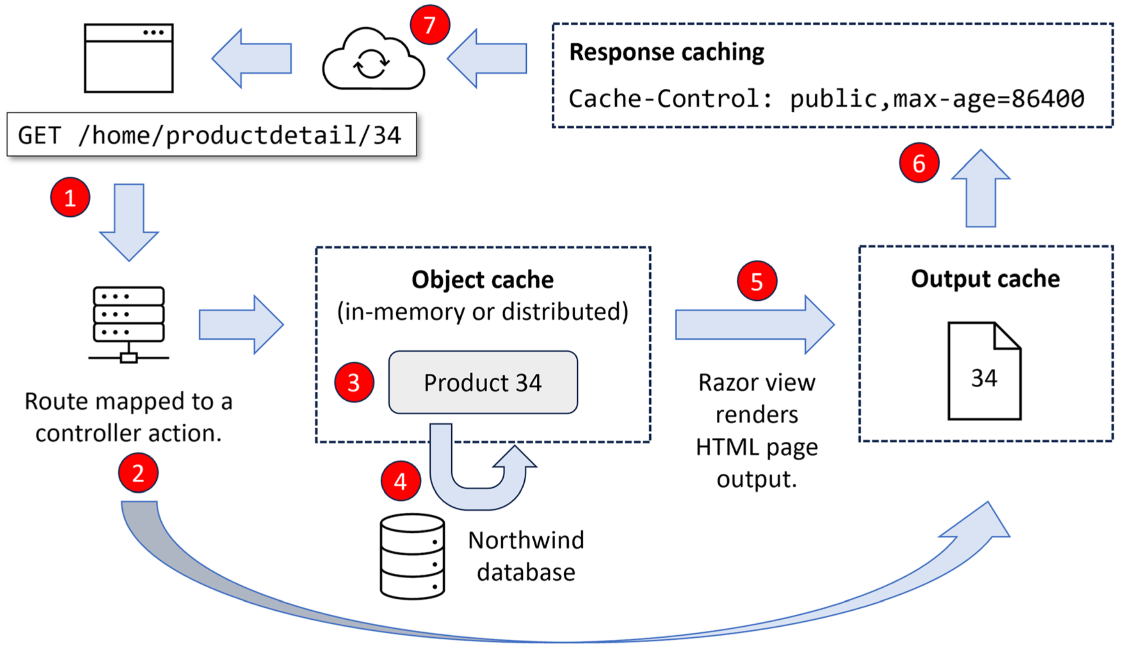

7
Response caching
Cache-Control: public,max-age=86400
GET /home/productdetail/34
6
1
Object cache
(in-memory or distributed)
5
Output cache
3
Product 34
34
Razor view renders HTML page output.
Route mapped to a controller action.
4
2
Northwind database

Figure 8: Types and locations for caching

Let's review a scenario. The numbered bullets match the numbered labels in *Figure 8*:

1. A web browser requests the product detail page for the product with an ID of 34.

2. On the web server, the output caching system looks in the output cache to see if that web page already exists with a match on its route. If not, the request is mapped to the appropriate controller action method and the `id` value `34` is passed in.
3. The action method implementation looks in the object cache to see if that entity already exists.
4. If not, it is retrieved from the database and stored in the object cache. Using an object cache is covered in *Chapter 15*, *Building and Consuming Web Services*.
5. The action method passes the product entity to the appropriate Razor view, which renders it into an HTML page and stores it in the output cache.
6. The action method adds response caching headers to the response before returning it.
7. Any intermediaries as well as the web browser can read the HTTP response headers and cache them if allowed.

On the next request, caching at various levels could improve matters:

1. The web browser might have the page in its local cache due to response caching. No further action is required. If not, a request is made to the web server. If intermediaries like a content delivery network (CDN) were allowed to cache the response, then they could return it to the browser without needing to forward the request to the web server.
2. If a request arrives at the web server, and the page is in the output cache, the action method processing can be bypassed, and the page is immediately returned with new response headers set.

Caching HTTP responses

To improve response times and scalability, you might want to try to cache the HTTP response that is generated by an action method by decorating the method with the `[ResponseCache]` attribute. This tells intermediaries like CDNs and the web browser itself how long they should cache the response for by adding HTTP headers to the response.

Good Practice: Response caching is only advisory. You cannot force other systems to cache if you do not control them. Keep in mind that any response caching you configure could be ignored.

You indicate where the response should be cached and for how long by setting parameters in the `[ResponseCache]` attribute, as shown in the following list:

- `Duration`: In seconds. This sets the `max-age` HTTP response header measured in seconds. Common choices are one hour (3,600 seconds) and one day (86,400 seconds).
- `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`.

Let's see an example:

1. In `HomeController.cs`, add an attribute to the `Index` method to cache the response for 10 seconds on the browser or any proxies between the server and browser, as shown highlighted in the following code:

```
[ResponseCache(Duration = 10 /* seconds */,
  Location = ResponseCacheLocation.Any)]
public IActionResult Index()
```

2. In `Views`, in `Home`, open `Index.cshtml`, and after the **Welcome** heading, add a paragraph to output the current time in long format to include seconds, as shown in the following markup:

```
<p class="alert alert-primary">@DateTime.Now.ToLongTimeString()</p>
```

3. Start the `Northwind.Mvc` website project using the `https` launch profile.
4. Note the time on the home page.
5. View **Developer Tools**, select the **Network** tab, select the **localhost** request, refresh the page, and note the `Cache-Control` response header, as shown in the following output:

```
Cache-Control: public,max-age=10
```

6. Click **Register** so that you leave the home page.

Warning! Do *not* click the **Reload the page** button as this will override the cache and refresh the page with a new request to the web server.

7. Click **Home** and note the time on the home page is the same because a cached version of the page is used.
8. Click **Register**. Wait at least 10 seconds.
9. Click **Home** and note the time has now updated.
10. Click **Log in**, enter your email and password, and then click **Log in**.
11. Note the time on the home page.
12. Click **Privacy**.
13. Click **Home** and note the page is not cached.
14. View the console and note the warning message explaining that your caching has been overridden because the visitor is logged in and, in this scenario, ASP.NET Core uses anti-forgery tokens and they should not be cached, as shown in the following output:

```
warn: Microsoft.AspNetCore.Antiforgery.DefaultAntiforgery[8]
      The 'Cache-Control' and 'Pragma' headers have been overridden
and set to 'no-cache, no-store' and 'no-cache' respectively to prevent
caching of this response. Any response that uses antiforgery should not
be cached.
```

15. Close Chrome and shut down the web server.

Output caching endpoints

Output caching stores dynamically generated responses on the server so that they do not have to be regenerated again for another request. This can improve performance.

Let's see it in action with examples of applying output caching to some endpoints:

1. In the `Northwind.Mvc` project, in `Program.cs`, add statements after the call to `AddNorthwindContext` to add the output cache middleware and override the default expiration timespan to make it only 10 seconds, as shown highlighted in the following code:

```
builder.Services.AddNorthwindContext();

builder.Services.AddOutputCache(options =>
{
  options.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(10);
});
```

Good Practice: The default expiration timespan is one minute. Think carefully about what the duration should be based on your website's typical visitor behavior.

2. In `Program.cs`, add statements before the call to map controllers to use the output cache, as shown highlighted in the following code:

```
app.UseOutputCache();

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

3. In `Program.cs`, add statements after the call to map Razor Pages to create two endpoints that respond with plain text, one that is not cached and one that uses the output cache, as shown highlighted in the following code:

```
app.MapRazorPages();

app.MapGet("/notcached", () => DateTime.Now.ToString());
app.MapGet("/cached", () => DateTime.Now.ToString()).CacheOutput();
```

4. In `appsettings.Development.json`, add a log level of `Information` for the output caching middleware, as shown highlighted in the following configuration:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.OutputCaching": "Information"
    }
  }
}
```

5. Start the `Northwind.Mvc` website project using the `https` launch profile.
6. Arrange the browser and command prompt or terminal window so that you can see both.
7. In the browser, make sure that you are not logged in to the MVC website because that would disable caching, then navigate to `https://localhost:5141/notcached`, and note nothing is written to the command prompt or terminal.
8. In the browser, click the **Refresh** button several times and note that the time is always updated because it is not served from the output cache.
9. In the browser, navigate to `https://localhost:5141/cached`, and note that messages are written to the console or terminal to tell you that you have made a request for a cached resource, but it does not have anything in the output cache, so it has now cached the output, as shown in the following output:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[7]
      No cached response available for this request.
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[9]
      The response has been cached.
```

10. In the browser, click the **Refresh** button several times and note that the time is not updated, and an output caching message tells you that the value was served from the cache, as shown in the following output:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[5]
      Serving response from cache.
```

11. Continue refreshing until 10 seconds have passed and note that messages are written to the command prompt or terminal to tell you that the cached output has been updated.
12. Close the browser and shut down the web server.

Output caching MVC views

Now let's see how we can output cache an MVC view:

1. In the `Views\Home` folder, in `ProductDetail.cshtml`, add a paragraph `<p>` to show the current time, as shown highlighted in the following markup:

```
<h2>Product Detail</h2>
<p class="alert alert-success">@DateTime.Now.ToLongTimeString()</p>
```

2. Start the `Northwind.Mvc` website project using the `https` launch profile.
3. Arrange the browser and command prompt or terminal window so that you can see both.
4. On the home page, scroll down and then select one of the products.
5. On the product detail page, note the current time, and then refresh the page and note that the time updates every second.
6. Close the browser and shut down the web server.
7. In `Program.cs`, at the end of the call to map controllers, add a call to the `CacheOutput` method, as shown highlighted in the following code:

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

8. Start the `Northwind.Mvc` website project using the `https` launch profile and arrange the browser window and command prompt or terminal window so that you can see both.
9. On the home page, scroll down, select one of the products, and note that the product detail page is not in the output cache, so SQL commands are executed to get the data. Then, once the Razor view generates the page, it is stored in the cache, as shown in the following output:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[7]
      No cached response available for this request.
dbug: 20/09/2022 17:23:02.402 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)

      Executing DbCommand [Parameters=[@__id_0='?' (DbType = Int32)],
CommandType='Text', CommandTimeout='30']
      SELECT "p"."ProductId", "p"."CategoryId", "p"."Discontinued",
"p"."ProductName", "p"."QuantityPerUnit", "p"."ReorderLevel",
"p"."SupplierId", "p"."UnitPrice", "p"."UnitsInStock",
"p"."UnitsOnOrder", "c"."CategoryId", "c"."CategoryName",
"c"."Description", "c"."Picture"
      FROM "Products" AS "p"
      LEFT JOIN "Categories" AS "c" ON "p"."CategoryId" =
"c"."CategoryId"
      WHERE "p"."ProductId" = @__id_0
      LIMIT 2
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
```

```
      Executed DbCommand (7ms) [Parameters=[@__id_0='?' (DbType =
Int32)], CommandType='Text', CommandTimeout='30']
      SELECT "p"."ProductId", "p"."CategoryId", "p"."Discontinued",
"p"."ProductName", "p"."QuantityPerUnit", "p"."ReorderLevel",
"p"."SupplierId", "p"."UnitPrice", "p"."UnitsInStock",
"p"."UnitsOnOrder", "c"."CategoryId", "c"."CategoryName",
"c"."Description", "c"."Picture"
      FROM "Products" AS "p"
      LEFT JOIN "Categories" AS "c" ON "p"."CategoryId" =
"c"."CategoryId"
      WHERE "p"."ProductId" = @__id_0
      LIMIT 2
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[9]
      The response has been cached.
```

10. On the product detail page, note the current time, and then refresh the page and note that the whole page, including the time and product detail data, is served from the output cache, as shown in the following output:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[5]
      Serving response from cache.
```

11. Keep refreshing until 10 seconds have passed and note that the page is then regenerated from the database and the current time is shown.
12. In the browser address bar, change the product ID number to a value between 1 and 77 to request a different product, and note that the time is current, and therefore a new cached version has been created for that product ID, because the ID is part of the relative path.
13. Refresh the browser and note the time is cached (and therefore the whole page is).
14. In the browser address bar, change the product ID number to a value between 1 and 77 to request a different product, and note that the time is current, and therefore a new cached version has been created for that product ID, because the ID is part of the relative path.
15. In the browser address bar, change the product ID number back to the previous ID and note the page is still cached with the time that the previous page was first added to the output cache.
16. Close the browser and shut down the web server.

Varying cached data by query string

If a value is different in the relative path, then output caching automatically treats the request as a different resource and so caches different copies for each, including differences in any query string parameters. Consider the following URLs:

- `https://localhost:5141/Home/ProductDetail/12`
- `https://localhost:5141/Home/ProductDetail/29`
- `https://localhost:5141/Home/ProductDetail/12?color=red`
- `https://localhost:5141/Home/ProductDetail/12?color=blue`

All four requests will have their own cached copy of their own page. If the query string parameters have no effect on the generated page, then that is a waste.

Let's see how we can fix this problem. We will start by disabling varying the cache by query string parameter values, and then implement some page functionality that uses a query string parameter:

1. In `Program.cs`, in the call to `AddOutputCache`, increase the default expiration to 20 seconds (or set a larger value to give yourself more time later) and add a statement to define a named policy to disable varying by query string parameters, as shown highlighted in the following code:

   ```
   builder.Services.AddOutputCache(options =>
   {
     options.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(20);
     options.AddPolicy("views", p => p.SetVaryByQuery(""));
   });
   ```

2. In `Program.cs`, in the call to `CacheOutput` for MVC, specify the named policy, as shown in the following code:

   ```
   app.MapControllerRoute(
       name: "default",
       pattern: "{controller=Home}/{action=Index}/{id?}")
     .CacheOutput(policyName: "views");
   ```

3. In `ProductDetail.cshtml`, modify the `<p>` that outputs the current time to set its alert style based on a value stored in the `ViewData` dictionary, as shown highlighted in the following markup:

   ```
   <p class="alert alert-@ViewData["alertstyle"]">
     @DateTime.Now.ToLongTimeString()</p>
   ```

4. In the `Controllers` folder, in `HomeController.cs`, in the `ProductDetail` action method, store a query string value in the `ViewData` dictionary, as shown in the following code:

   ```
   public IActionResult ProductDetail(int? id,
     string alertstyle = "success")
   {
     ViewData["alertstyle"] = alertstyle;
   ```

5. Start the `Northwind.Mvc` website project using the `https` launch profile.
6. Arrange the browser and command prompt or terminal window so that you can see both.
7. On the home page, scroll down, select one of the products, and note the color of the alert is green because the `alertstyle` defaults to `success`.
8. In the browser address bar, append a query string parameter, `?alertstyle=warning`, and note that it is ignored, because the same cached page is returned.

9. In the browser address bar, change the product ID number to a value between 1 and 77 to request a different product and append a query string parameter, `?alertstyle=warning`. Note that the alert is yellow because it is treated as a new request.
10. In the browser address bar, append a query string parameter, `?alertstyle=info`, and note that it is ignored, because the same cached page is returned.
11. Close the browser and shut down the web server.
12. In `Program.cs`, in the call to `AddOutputCache`, in the call to `AddPolicy`, set `alertstyle` as the only named parameter to vary by for query string parameters, as shown highlighted in the following code:

```
builder.Services.AddOutputCache(options =>
{
  options.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(20);
  options.AddPolicy("views", p => p.SetVaryByQuery("alertstyle"));
});
```

13. Start the `Northwind.Mvc` website project using the `https` launch profile.
14. Repeat the steps above to confirm that requests for different `alertstyle` values do have their own cached copies but any other query string parameter would be ignored.
15. Close the browser and shut down the web server.

Disabling caching to avoid confusion

Before we continue, let's disable view output caching, otherwise you are likely to be confused by the website behavior if or when you forget caching is enabled!

1. In `Program.cs`, disable the output caching by commenting out the call to `CacheOutput`, as shown highlighted in the following code:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
//  .CacheOutput(policyName: "views");
```

2. Save the changes.

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.

In previous examples, we defined a view model that contained properties for every value that needed to be rendered in the view. In this example, there will be two values: a list of products and the price the visitor entered. To avoid having to define a class or record for the view model, we will pass the list of products as the model and store the maximum price in the `ViewData` collection.

Let's implement this feature:

1. In `HomeController.cs`, add a new action method, as shown in the following code:

```
public IActionResult ProductsThatCostMoreThan(decimal? price)
{
  if (!price.HasValue)
  {
    return BadRequest("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.Any())
  {
    return NotFound(
      $"No products cost more than {price:C}.");
  }

  ViewData["MaxPrice"] = price.Value.ToString("C");

  return View(model);
}
```

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

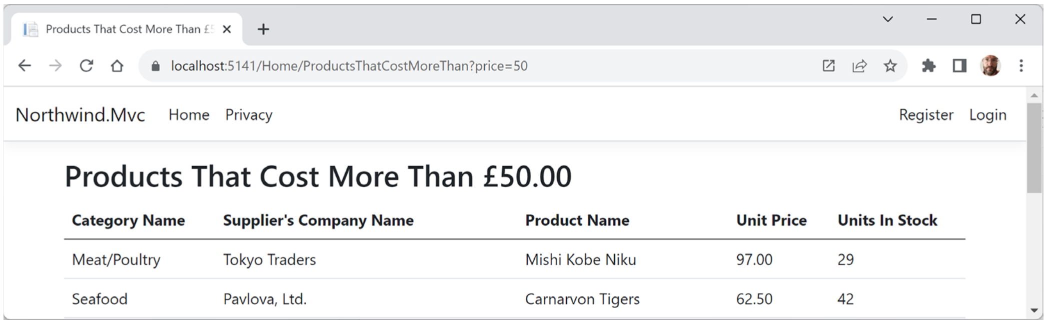
```
@using Northwind.EntityModels
@model IEnumerable<Product>
@{
  string title =
    $"Products That Cost More Than {ViewData["MaxPrice"]}";
  ViewData["Title"] = title;
}
<h2>@title</h2>
@if (Model is null)
{
  <div>No products found.</div>
}
```

```
else
{
  <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 (Product p in Model)
    {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => p.Category.CategoryName)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.Supplier.CompanyName)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.ProductName)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.UnitPrice)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.UnitsInStock)
        </td>
      </tr>
    }
    <tbody>
  </table>
}
```

4. In the `Views/Home` folder, open `Index.cshtml`.
5. 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 **Submit** 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>
```

6. Start the `Northwind.Mvc` website project using the `https` launch profile.
7. On the home page, enter a price in the form, for example, `50`, and then click on **Submit**.
8. Note the table of the products that cost more than the price that you entered, as shown in *Figure 9*:


Products That Cost More Than £
localhost:5141/Home/ProductsThatCostMoreThan?price=50
Northwind.Mvc
Home
Privacy
Register
Login

Products That Cost More Than £50.00

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

Figure 9: A filtered list of products that cost more than £50

9. Close Chrome and shut down the web server.

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 for 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 in the pool 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.

Good Practice: Make your controller action methods asynchronous.

Localization and globalization for .NET and ASP.NET Core projects are covered in multiple chapters in the companion book, *Apps and Services with .NET 8*.

Making controller action methods asynchronous

In an earlier task, you imported the `Microsoft.EntityFrameworkCore` namespace so that you could use the `Include` extension method. You are about to use another extension method that requires that namespace to be imported.

It is easy to make an existing action method asynchronous:

1. At the top of `HomeController.cs`, make sure the namespace for adding EF Core extension methods is imported, as shown in the following code:

```
// To use the Include and ToListAsync extension methods.
using Microsoft.EntityFrameworkCore;
```

2. In `HomeController.cs`, modify the `Index` action method to be asynchronous and await the calls to asynchronous methods to get the categories and products, as shown highlighted in the following code:

```
[ResponseCache(Duration = 10 /* seconds */,
  Location = ResponseCacheLocation.Any)]
public async Task<IActionResult> Index()
{
  _logger.LogError("This is a serious error (not really!)");
  _logger.LogWarning("This is your first warning!");
  _logger.LogWarning("Second warning!");
  _logger.LogInformation("I am in the Index method of the
HomeController.");

  HomeIndexViewModel model = new
```

```
    (
      VisitorCount: Random.Shared.Next(1, 1001),
      Categories: await _db.Categories.ToListAsync(),
      Products: await _db.Products.ToListAsync()
    );

    return View(model); // Pass the model to the 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,
```

4. In the `ProductDetail` action method, await the calls to asynchronous methods to get the product, as shown highlighted in the following code:

```
Product? model = await _db.Products.Include(p => p.Category)
  .SingleOrDefaultAsync(p => p.ProductId == id);
```

5. Start the `Northwind.Mvc` website project using the `https` launch profile.
6. Note that the functionality of the website is the same, but trust that it will now scale better.
7. Close Chrome and shut down the web server.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 1 – Test your knowledge

Answer the following questions:

1. What do the files with the special names `_ViewStart` and `_ViewImports` do when created in the `Views` folder?
2. What are the names of the three segments defined in the default ASP.NET Core MVC route, what do they represent, and which are optional?
3. What does the default model binder do, and what data types can it handle?
4. In a shared layout file like `_Layout.cshtml`, how do you output the content of the current view?
5. In a shared layout file like `_Layout.cshtml`, how do you output a section that the current view can supply content for, and how does the view supply the contents for that section?
6. When calling the `View` method inside a controller's action method, what paths are searched for the view by convention?
7. How can you instruct the visitor's browser to cache the response for 24 hours?
8. Why might you enable Razor Pages even if you are not creating any yourself?

9. How does ASP.NET Core MVC identify classes that can act as controllers?
10. In what ways does ASP.NET Core MVC make it easier to test a website?

Exercise 2 – Practice implementing MVC by implementing a category detail page

The `Northwind.Mvc` project has a home page that shows categories, but when the `View` button is clicked, the website returns a `404 Not Found` error, for example, for the following URL:

```
https://localhost:5141/home/categorydetail/1
```

Extend the `Northwind.Mvc` project by adding the ability to show a detail page for a category.

For example, add an action method to the `HomeController` class, as shown in the following code:

```
public async Task<IActionResult> CategoryDetail(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest("You must pass a category ID in the route, for example, /
Home/CategoryDetail/6");
  }

  Category? model = await db.Categories.Include(p => p.Products)
    .SingleOrDefaultAsync(p => p.CategoryId == id);

  if (model is null)
  {
    return NotFound($"CategoryId {id} not found.");
  }

  return View(model); // Pass model to view and then return result.
}
```

And create a view that matches the name `CategoryDetail.cshtml`, as shown in the following markup:

```
@model Northwind.EntityModels.Category
@{
  ViewData["Title"] = "Category Detail - " + Model.CategoryName;
}
<h2>Category Detail</h2>
<div>
  <dl class="dl-horizontal">
    <dt>Category Id</dt>
    <dd>@Model.CategoryId</dd>
```

```
      <dt>Product Name</dt>
      <dd>@Model.CategoryName</dd>
      <dt>Products</dt>
      <dd>@Model.Products.Count</dd>
      <dt>Description</dt>
      <dd>@Model.Description</dd>
    </dl>
  </div>
```

Exercise 3 – Practice improving scalability by understanding and implementing async action methods

Almost a decade ago, Stephen Cleary wrote an excellent article for MSDN Magazine explaining the scalability benefits of implementing async action methods for ASP.NET. The same principles apply to ASP.NET Core, but even more so, because unlike the old ASP.NET as described in the article, ASP.NET Core supports asynchronous filters and other components.

Read the article at the following link:

`https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/october/async-programming-introduction-to-async-await-on-asp-net`

Exercise 4 – Practice unit testing MVC controllers

Controllers are where the business logic of your website runs, so it is important to test the correctness of that logic using unit tests, as you learned in *Chapter 4, Writing, Debugging, and Testing Functions*.

Write some unit tests for `HomeController`.

Good Practice: You can read more about how to unit test controllers at the following link: `https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/testing`.

Exercise 5 – Using a filter to control authorization

In this chapter, you learned about cross-functional filters like `[Route]` and `[ResponseCache]`. To learn how to secure parts of a website using the `[Authorize]` attribute, you can read an optional online-only section at the following link:

`https://github.com/markjprice/cs12dotnet8/blob/main/docs/ch13-authorization.md`

Summary

In this chapter, you learned how to build large, complex websites in a way that is easy to unit test by registering and injecting dependency services like database contexts and loggers. You learned about:

- Configuration
- Routes
- Models
- Views
- Controllers
- Caching

In the next chapter, you will learn how to build and consume services that use HTTP as the communication layer, aka web services.