CHAPTER 14

Configuring Applications

The topic of configuration may not seem interesting, but it reveals a lot about how MVC applications work and how HTTP requests are handled. Resist the temptation to skip this chapter, and take the time to understand the way that the configuration system shapes MVC web applications. It will give you a solid foundation for understanding the chapters that follow.

In this chapter, I explain how these are used to configure MVC applications and show how MVC builds on features provided by the ASP.NET Core platform. Table 14-1 puts configuring applications in context.

Table 14-1. *Putting Configuration in Context*

Question	Answer
What is it?	The `Program` and `Startup` classes and the JSON files are used to configure how an application works and what packages it depends on.
Why is it useful?	The configuration system allows applications to be tailored to their environments and to manage their package dependencies.
How is it used?	The most important component is the `Startup` class, which is used to create services (which are objects that provide common functionality throughout an application) and middleware components (which are used to handle HTTP requests).
Are there any pitfalls or limitations?	In complex applications, the configuration can become difficult to manage. See the "Dealing with Complex Configurations" section for ASP.NET features intended to manage this problem.
Are there any alternatives?	No. The configuration system is an integral part of ASP.NET and the means by which MVC applications are set up.

Table 14-2 summarizes the chapter.

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_14

Table 14-2. *Chapter Summary*

Problem	Solution	Listing
Add functionality to the application	Add NuGet packages to the `csproj` file	5–8
Manage the initialization of the ASP.NET application	Use the `Program` class	9–11
Configure the application	Use the `ConfigureServices` and `Configure` methods of the `Startup` class	12–13
Create common functionality	Use the `ConfigureServices` method to create services	14–16
Generate content responses	Create content-generating middleware	17–19
Prevent requests from traversing the request pipeline	Create short-circuiting middleware	20–21
Edit a request before it is processed by other middleware components	Create request-editing middleware	22–24
Edit a response that has been processed by other middleware components	Create response-editing middleware	25–26
Set up MVC functionality	Use the `UseMvc` or `UseMvcWithDefaultRoute` method	27
Change the application configuration for different environments	Use the hosting environment service	28
Handle application errors	Use the developer or production error-handling middleware	29–30
Manage multiple browsers during development	Use Browser Link	31
Enable images, JavaScript files, and CSS files	Enable the static content middleware	32
Separate configuration data from C# code	Create external configuration sources, such as JSON files	33–35
Log application data	Use the logging middleware	36–38
Prepare dependency injection for use with Entity Framework Core	Disable scope validation	39
Configure MVC services	Use the options features	40
Configure complex applications	Use multiple external files or classes	41–45

Preparing the Example Project

For this chapter, I created a new project called ConfiguringApps using the Empty template. I am going to configure the application later in the chapter, but there are some basics that I need to put in place in preparation for the changes I make.

I am going to use Bootstrap to style the HTML content in this chapter, so I created the `bower.json` file using the Bower Configuration File item template and added the package shown in Listing 14-1.

Listing 14-1. Adding Bootstrap in the bower.json File in the ConfiguringApps Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Next, I created the `Controllers` folder and added a class file called `HomeController.cs`, which I used to define the controller shown in Listing 14-2.

Listing 14-2. The Contents of the HomeController.cs File in the Controllers Folder

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() => View(new Dictionary<string, string> {
                ["Message"] = "This is the Index action"
            });
    }
}
```

I created the `Views/Home` folder and added a view file called `Index.cshtml` with the content shown in Listing 14-3.

Listing 14-3. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model Dictionary<string, string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
    <title>Result</title>
</head>
<body class="p-1">
    <table class="table table-condensed table-bordered table-striped">
        @foreach (var kvp in Model) {
            <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
        }
    </table>
</body>
</html>
```

The link element in the view relies on a built-in tag helper to select the Bootstrap CSS files. To enable the built-in tag helpers, I used the MVC View Imports Page item template to create the _ViewImports.cshtml file in the Views folder and added the expression shown in Listing 14-4.

Listing 14-4. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Start the application and you will see the message shown in Figure 14-1.

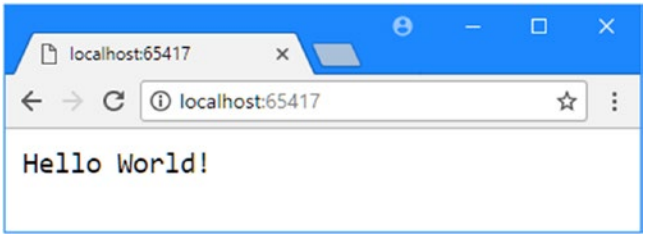

Figure 14-1. *Running the example application*

Configuring the Project

The most important configuration file is <projectname>.csproj, which replaces the project.json file used in earlier versions of ASP.NET Core. This file, which is called ConfiguringApps.csproj in the example project, is hidden by Visual Studio and must be accessed by right-clicking the project item in the Solution Explorer window and selecting Edit ConfiguringApps.csproj from the pop-up menu. Listing 14-5 shows the initial content of the ConfiguringApps.csproj file, which was created by Visual Studio as part of the Empty project template.

Listing 14-5. The Contents of the ConfiguringApps.csproj File in the ConfiguringApps Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

The csproj file is used to configure the MSBuild tool, which is used to build .NET projects. Configuration is performed using XML elements, and Table 14-3 describes the elements that are in the default configuration file. I use other configuration elements in later examples, but the elements in the table are enough to start development on an ASP.NET Core MVC project.

Table 14-3. *The XML Configuration Elements in the Default csproj File*

Element	Description
Project	This is the root element, which denotes that this is an MSBuild configuration file. The Sdk attribute is set to Microsoft.NET.Sdk.Web to provide the set of implicit package imports that are required to build the project.
PropertyGroup	This element groups related configuration properties to add structure to the file.
TargetFramework	This element specifies the .NET Framework that is targeted by the build process and must be defined within a PropertyGroup element. The default value is netcoreapp2.0, which targets .NET Core 2.0.
ItemGroup	This element groups related configuration items to add structure to the file.
Folder	This element tells MSBuild how to deal with a folder in the project. The element in the listing tells MSBuild to include the wwwroot folder when the application is published.
PackageReference	This element is used to specify a dependency on the NuGet package, which is identified through the Include and Version attributes. The Microsoft.AspNetCore.All package is used to provide access to all of the individual packages that provide ASP.NET Core and MVC Framework functionality.

Adding Packages to the Project

The most important role for the csproj file is to list the packages that the project depends on. When Visual Studio detects a change to the csproj file, it inspects the list of packages, downloads any new additions, and removes any packages that are no longer required.

With the release of ASP.NET Core 2, all of the basic functionality required for ASP.NET Core MVC, the MVC Framework, and Entity Framework Core is included in the Microsoft.AspNetCore.All meta-package, which is a convenience feature that avoids the need to start a new development effort by adding a long list of NuGet packages to the project.

Even so, you will still need to add NuGet packages for third-party or advanced features. There are three ways to add packages to a project. The first is to select Tools ➤ NuGet Package Manager ➤ Manage NuGet Packages for Solution, which allows the management of NuGet packages through an easy-to-use interface. If you are new to .NET development, then this is the best approach because it reduces the chance of making a mistake when selecting a package.

To add the System.Net.Http package, for example, which provides support for making (rather than receiving) HTTP requests, you can go to the Browse section of the package manager, search by name, and see a complete list of the versions available, including any prerelease versions, as shown in Figure 14-2.

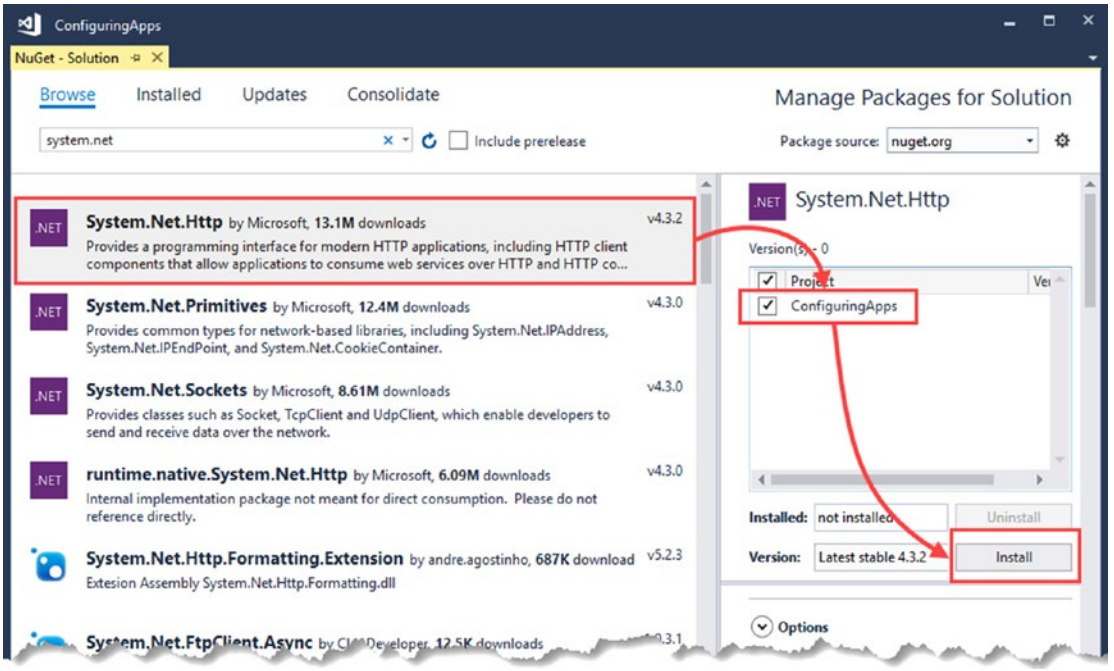

Figure 14-2. *Adding a package using the NuGet package manager*

Select the package and version you need, select the projects that the package is required for, and click the Install button. Visual Studio will download the package and update the `csproj` file.

You can also add packages using the command line, although this requires you to know the name of the package you require (and, ideally, the version). Listing 14-6 shows the command you would run in the `ConfiguringApps` folder to add the `System.Net.Http` package to the project.

Listing 14-6. Adding a Package to the Project

```
dotnet add package System.Net.Http --version 4.3.2
```

The NuGet package manager and the `dotnet add package` command both add `PackageReference` elements to the `csproj` file. If you prefer, you can add packages by editing the configuration file to add the `PackageReference` element manually. This is the most direct approach but requires care to avoid mistyping the name of the package or specifying a version number that doesn't exist. In Listing 14-7, you can see the addition to the `csproj` file for the `System.Net.Http` package.

Listing 14-7. Adding a Package in the ConfiguringApps.csproj File in the ConfiguringApps Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
```

```
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
  </ItemGroup>
```

The PackageReference element has an Include attribute that specifies the package name and a Version attribute that specifies the version number.

Adding Tools Packages to the Project

Although you can add regular packages in different ways, some packages extend the range of tasks that can be performed with the dotnet command-line tool, and these packages require a different kind of element in the csproj file, which is a DotNetCliToolReference element instead of the PackageReference element used by packages that contain features used directly by the application. These packages can be added to projects only by editing the csproj file directly.

Listing 14-8 shows the addition of the package that allows database migrations to be created and applied using the dotnet ef commands used in Part 1 of this book.

Listing 14-8. Adding Tools Packages to the ConfiguringApps.csprog File in the ConfiguringApps Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="2.0.0 " />
  </ItemGroup>

</Project>
```

When you add tools packages to a project, you can either include the DotNetCliToolReference elements in the same ItemGroup as the regular PackageReference elements, as I have done in Listing 14-8, or create a separate ItemGroup element. When you save the changes to the csproj file, Visual Studio will download and install the packages and use them to configure the dotnet command-line tool.

Understanding the Program Class

The Program class is defined in a file called Program.cs and provides the entry point for running the application, providing .NET with a main method that can be executed to configure the hosting environment and select the class that completes the configuration for the ASP.NET Core application. The default contents of the Program class are enough to get most projects up and running, and Listing 14-9 shows the default code added to projects by Visual Studio.

Listing 14-9. The Default Contents of the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The Main method provides the entry point that all .NET applications must provide so they can be executed by the runtime. The Main method in the Program class calls the BuildWebHost method, which is responsible for configuring ASP.NET Core.

The BuildWebHost method uses static methods defined by the WebHost class to configure ASP.NET Core. With the release of ASP.NET Core 2, the configuration is simplified by the use of the CreateDefaultBuilder method, which configures ASP.NET Core using settings that are likely to suit most projects. The UseStartup method is called to identify the class that will provide application-specific configuration; the convention is to use a class called Startup, which I describe later in this chapter. The Build method processes all the configuration settings and creates an object that implements the IWebHost interface, which is returned to the Main method, which calls Run to start handling HTTP requests.

Digging into the Configuration Detail

The CreateDefaultBuilder method is a convenient way to jump-start the configuration of ASP.NET Core, but it does hide a lot of important detail, which can be a problem if you need to change the way that your application is configured. Listing 14-10 replaces the CreateDefaultBuilder method with individual statements that are called to create the default configuration.

Listing 14-10. Detailed Configuration Statements in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {
            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    var env = hostingContext.HostingEnvironment;
                    config.AddJsonFile("appsettings.json", optional: true,
                            reloadOnChange: true)
                        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                            optional: true, reloadOnChange: true);

                    if (env.IsDevelopment()) {
                        var appAssembly =
                            Assembly.Load(new AssemblyName(env.ApplicationName));
                        if (appAssembly != null) {
                            config.AddUserSecrets(appAssembly, optional: true);
                        }
                    }

                    config.AddEnvironmentVariables();

                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logging) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.GetSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
```

```
                .UseIISIntegration()
                .UseDefaultServiceProvider((context, options) => {
                    options.ValidateScopes =
                        context.HostingEnvironment.IsDevelopment();
                })
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

Table 14-4 lists each of the configuration methods added to the `BuildWebHost` method and provides a brief description of what they do.

Table 14-4. *The Default ASP.NET Core Configuration Methods*

Name	Description
`UseKestrel`	This method configures the Kestrel web server, as described in the "Using Kestrel Directly" sidebar.
`UseContentRoot`	This method configures the root directory for the application, which is used for loading configuration files and delivering static content such as images, JavaScript, and CSS.
`ConfigureAppConfiguration`	This method is used to prepare the configuration data for the application, as described in the "Configuring the Application" section later in this chapter.
`AddUserSecrets`	This method is used to store sensitive data outside of code files, as described at https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets. This is a somewhat awkward feature, which I do not use in this book.
`ConfigureLogging`	This method is used to configure logging for the application, as described in the "Configuring Logging" section later in this chapter.
`UseIISIntegration`	This method enables integration with IIS and IIS Express.
`UseDefaultServiceProvider`	This method is used to configure dependency injection, as described in the "Configuring Dependency Injection" section.
`UseStartup`	This method specifies the class that will be used to configure ASP.NET, as described in the "Understanding the Startup Class" section.

I explain some of the more complex statements shown in Listing 14-10 later in the chapter. For now, I am going to remove some of the configuration statements so that only the basic configuration remains, as shown in Listing 14-11.

Listing 14-11. Simplifying the Configuration in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

These are the statements that provide a basic configuration that will work for most ASP.NET Core MVC applications. I'll add back in the other statements as I explain the features they relate to.

USING KESTREL DIRECTLY

Kestrel is a cross-platform web server designed to run ASP.NET Core applications. It is used automatically when you run an ASP.NET Core application using IIS Express (which is the server provided by Visual Studio for use during development) or the full version of IIS, which has been the traditional web platform for .NET applications.

You can also run Kestrel directly if you want, which means you can run your ASP.NET Core MVC applications on any of the supported platforms, bypassing the Windows-only restriction of IIS. There are two ways to run an application using Kestrel. The first is to click the arrow at the right edge of the IIS Express button on the Visual Studio toolbar and select the entry that matches the name of the project. This will open a new command prompt and run the application using Kestrel.

You can achieve the same effect by opening your own command prompt, navigating to the folder that contains the application's configuration files (the one that contains the `csproj` file), and running the following command:

```
dotnet run
```

By default, the Kestrel server starts listening for HTTP requests on port 5000. If there is a `Properties/launchSettings.json` file in the project, the HTTP port and environment for the application will be read from this file.

Understanding the Startup Class

The Program class is responsible for jump-starting the application, but the most important configuration work is delegated through the UseStartup method, like this:

```
...
.UseStartup<Startup>()
...
```

The UseStartup method relies on a type parameter to identify the class that will configure ASP.NET Core. The conventional name for this class is Startup, which is the name used by the ASP.NET Core MVC project templates, including the Empty template used to create the example project for this chapter.

Examining how the Startup class works provides insights into the way that HTTP requests are processed and how MVC integrates into the rest of the ASP.NET Core platform.

In this section, I start with the simplest possible Startup class and add features to demonstrate the effect of different configuration options, ending up with a configuration that is suitable for most MVC projects. As the starting point, Listing 14-12 shows the Startup class that Visual Studio adds to Empty projects, which sets up just enough functionality to get ASP.NET Core to handle HTTP requests.

Listing 14-12. The Initial Contents of the Startup.cs File

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

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

The Startup class defines two methods, ConfigureServices and Configure, that set up the shared features required by an application and tell ASP.NET Core how they should be used.

When the application starts, ASP.NET Core creates a new instance of the `Startup` class and calls its `ConfigureServices` method so that the application can create its *services*. As I explain in the "Understanding ASP.NET Services" section, services are objects that provide functionality to other parts of the application. This is a vague description, but that's because services can be used to provide just about any functionality.

Once the services have been created, ASP.NET calls the `Configure` method. The purpose of the `Configure` method is to set up the *request pipeline*, which is the set of components—known as *middleware*—that are used to handle incoming HTTP requests and produce responses for them. I explain how the request pipeline works and demonstrate how to create middleware components in the "Understanding ASP.NET Middleware" section. Figure 14-3 shows the way that ASP.NET uses the `Startup` class.

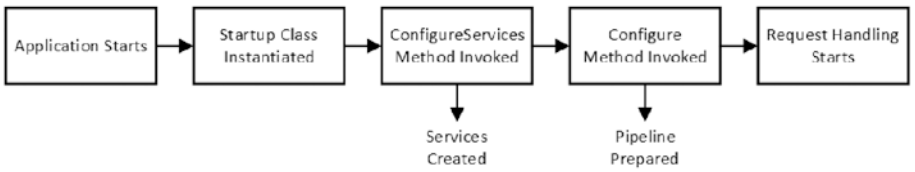

Figure 14-3. *How ASP.NET uses the Startup class to configure an application*

It isn't especially useful to have a `Startup` class that just returns the same "Hello, World" message for all requests, so before I explain what the methods in the class do in detail, I need to jump ahead a little and enable MVC, as shown in Listing 14-13.

Listing 14-13. Enabling MVC in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

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

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

With these additions—which I explain in the sections that follow—there is enough infrastructure in place to process HTTP requests and generate responses using controllers and views. If you run the application, you will see the output shown in Figure 14-4.

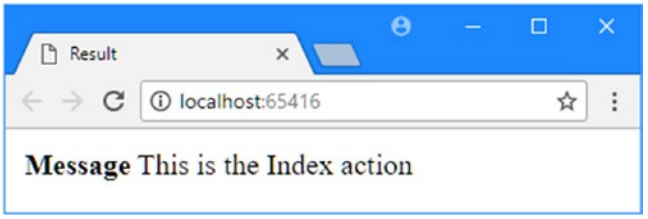

Figure 14-4. *The effect of enabling MVC*

Notice that the content is not styled. The minimal configuration in Listing 14-13 doesn't provide any support for serving up static content, such as CSS stylesheets and JavaScript files, so the `link` element in the HTML rendered by the `Index.cshtml` view produces a request for the Bootstrap CSS style sheet that the application can't process, which prevents the browser from getting the style information it required. I fix this problem in the "Adding the Remaining Middleware Components" section.

Understanding ASP.NET Services

ASP.NET Core calls the `Startup.ConfigureServices` method so that the application can set up the *services* it requires. The term *service* refers to any object that provides functionality to other parts of the application. As noted, this is a vague description because services can do *anything* that your application requires. As an example, I added an `Infrastructure` folder to the project and added to it a class file called `UptimeService.cs`, which I used to define the class shown in Listing 14-14.

Listing 14-14. The Contents of the UptimeService.cs File in the Infrastructure Folder

```
using System.Diagnostics;

namespace ConfiguringApps.Infrastructure {

    public class UptimeService {
        private Stopwatch timer;

        public UptimeService() {
            timer = Stopwatch.StartNew();
        }

        public long Uptime => timer.ElapsedMilliseconds;
    }
}
```

When this class is created, its constructor starts a timer that keeps track of how long the application has been running. This is a nice example of a service because it provides functionality that can be used in the rest of the application and it benefits from being created when the application is started.

ASP.NET services are registered using the `ConfigureServices` method of the `Startup` class, and in Listing 14-15, you can see how I have registered the `UptimeService` class.

Listing 14-15. Registering a Custom Service in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

As its argument, the `ConfigureServices` method receives an object that implements the `IServiceCollection` interface. Services are registered using extension methods called on the `IServiceCollection` interface that specify different configuration options. I describe the options available for creating services in Chapter 18, but for the moment, I have used the `AddSingleton` method, which means that a single `UptimeService` object will be shared throughout the application.

Services are closely related to a feature called *dependency injection*, which allows components such as controllers to easily obtain services and which I describe in depth in Chapter 18. Services registered in `Startup.ConfigureServices` can be accessed by creating a constructor that accepts an argument of the service type you require. Listing 14-16 shows the constructor I added to the `Home` controller to receive the shared `UptimeService` object that I created in Listing 14-15. I have also updated the controller's `Index` action method so that it includes the value of the service's `Update` property in the view data it produces.

Listing 14-16. Accessing a Service in the HomeController.cs File in the Controllers Folder

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;

        public HomeController(UptimeService up) => uptime = up;
```

```
        public ViewResult Index()
            => View(new Dictionary<string, string> {
                ["Message"] = "This is the Index action",
                ["Uptime"] = $"{uptime.Uptime}ms"
            });
    }
}
```

When MVC needs an instance of the Home controller class to handle an HTTP request, it inspects the HomeController constructor and finds that it requires an UptimeService object. MVC then inspects the set of services that have been configured in the Startup class, finds that UptimeService has been configured so that a single UptimeService object is used for all requests, and passes that object as the constructor argument when the HomeController is created.

Services can be registered and consumed in more complex ways, but this example demonstrates the central idea behind services and shows how defining a service in the Startup class allows you to define functionality or data that be used throughout an application.

If you run the application and request the default URL, you will see a response that includes the number of milliseconds since the application has started, which is obtained from the UptimeService object that was created in the Startup class, as illustrated in Figure 14-5. (Strictly speaking, it is the time since the UptimeService service object was created but this is close enough to the application startup to make no difference for the purpose of this chapter).

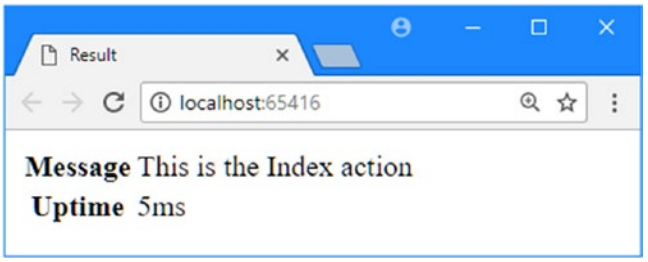

Figure 14-5. *Using a simple service*

Each time a request for the default URL is received, MVC creates a new HomeController object and provides it with the shared UptimeService object as a constructor argument. This allows the Home controller access to the application's uptime without being concerned about how this information is provided or implemented.

Understanding the Built-In MVC Services

A package as complex as MVC uses many services; some are for its internal use, and others offer functionality to developers. Packages define extension methods that set up all the services they require in a single method call. For MVC, this method is called AddMvc, and it is one of the two methods I added to the Startup class to get MVC working.

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<UptimeService>();
    services.AddMvc();
}
...
```

This method sets up every service that MVC needs without filling up the `ConfigureServices` method with an enormous list of individual services.

■ **Note** The Visual Studio IntelliSense feature will show you a long list of other extension methods that you can call on the `IServiceCollection` object in the `ConfigureServices` method. Some of these methods, such as `AddSingleton` and `AddScoped`, are used to register services in different ways. The other methods, such as `AddRouting` or `AddCors`, add individual services that are already applied by the `AddMvc` method. The result is that for most applications, the `ConfigureServices` method contains a small number of custom services, the call to the `AddMvc` method, and, optionally, some statements to configure the built-in services, which I describe in the "Configuring MVC Services" section.

Understanding ASP.NET Middleware

In ASP.NET Core, *middleware* is the term used for the components that are combined to form the *request pipeline*. The request pipeline is arranged like a chain, and when a new request arrives, it is passed to the first middleware component in the chain. This component inspects the request and decides whether to handle it and generate a response or to pass it to the next component in the chain. Once a request has been handled, the response that will be returned to the client is passed back along the chain, which allows all of the earlier components to inspect or modify it.

The way that middleware components work may seem a little odd, but it allows for a lot of flexibility in the way that applications are put together. Understanding how the use of middleware shapes an application can be important, especially if you are not getting the responses you expect. To explain how the middleware system works, I am going to create some custom components that demonstrate each of the four types of middleware that you will encounter.

Creating Content-Generating Middleware

The most important type of middleware generates content for clients, and it is this category to which MVC belongs. To create a content-generating middleware component without the complexity of MVC, I added a class called `ContentMiddleware.cs` to the `Infrastructure` folder and used it to define the class shown in Listing 14-17.

Listing 14-17. The Contents of the ContentMiddleware.cs File in the Infrastructure Folder

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ContentMiddleware {
        private RequestDelegate nextDelegate;

        public ContentMiddleware(RequestDelegate next) => nextDelegate = next;
```

```
        public async Task Invoke(HttpContext httpContext) {
            if (httpContext.Request.Path.ToString().ToLower() == "/middleware") {
                await httpContext.Response.WriteAsync(
                "This is from the content middleware", Encoding.UTF8);
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

Middleware components don't implement an interface or derive from a common base class. Instead, they define a constructor that takes a `RequestDelegate` object and define an `Invoke` method. The `RequestDelegate` object represents the next middleware component in the chain, and the `Invoke` method is called when ASP.NET receives an HTTP request.

Information about the HTTP request and the response that will be returned to the client is provided through the `HttpContext` argument to the `Invoke` method. I describe the `HttpContext` class and its properties in Chapter 17, but for this chapter, it is enough to know that the `Invoke` method in Listing 14-17 inspects the HTTP request and checks to see whether the request has been sent to the `/middleware` URL. If it has, then a simple text response is sent to the client; if a different URL has been used, then the request is forwarded to the next component in the chain.

The request pipeline is set up inside the `Configure` method of the `Startup` class. In Listing 14-18, I have removed MVC methods from the example application and used the `ContentMiddleware` class as the sole component in the pipeline.

Listing 14-18. Using a Custom Middleware in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

Custom middleware components are registered with the `UseMiddleware` extension method within the `Configure` method. The `UseMiddleware` method uses a type parameter to specify the middleware class. This so that ASP.NET Core can build up a list of all the middleware components that are going to be used and then instantiate them to create the chain. If you run the application and request the `/middleware` URL, you will see the result shown in Figure 14-6.

Figure 14-6. *Generating content from a custom middleware component*

Figure 14-7 illustrates the middleware pipeline that I created using the `ContentMiddleware` class. When ASP.NET Core receives an HTTP request, it passes it to the only middleware component registered in the `Startup` class. If the URL is `/middleware`, then the component generates a result, which is returned to ASP.NET Core and sent to the client.

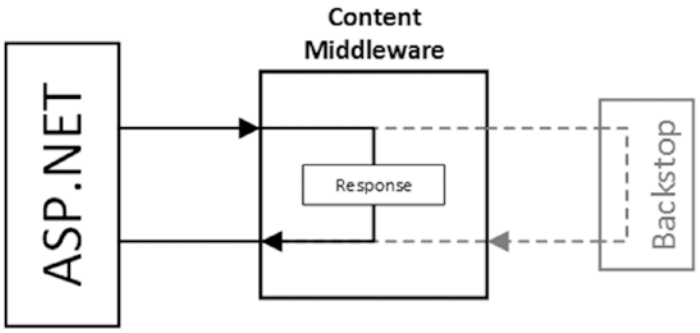

Figure 14-7. *The example middleware pipeline*

If the URL isn't `/middleware`, then the `ContentMiddleware` class passes on the request to the next component in the chain. Since there is no other component, the request reaches a backstop handler provided by ASP.NET Core when it creates the pipeline, which sends the request back along the pipeline in the other direction (a process that will make more sense once you see how the other types of middleware work).

Using Services in Middleware

It isn't just controllers that can use services that have been set up in the `ConfigureServices` method. ASP.NET Core inspects the constructors of middleware classes and uses services to provide values for any arguments that have been defined. In Listing 14-19, I have added an argument to the constructor of the `ContentMiddleware` class, which tells ASP.NET Core that it requires an `UptimeService` object.

Listing 14-19. Using a Service in the ContentMiddleware.cs File in the Infrastructure Folder

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ContentMiddleware {
        private RequestDelegate nextDelegate;
        private UptimeService uptime;

        public ContentMiddleware(RequestDelegate next, UptimeService up) {
            nextDelegate = next;
            uptime = up;
        }

        public async Task Invoke(HttpContext httpContext) {
            if (httpContext.Request.Path.ToString().ToLower() == "/middleware") {
                await httpContext.Response.WriteAsync(
                    "This is from the content middleware "+
                        $"(uptime: {uptime.Uptime}ms)",  Encoding.UTF8);
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

Being able to use services means that middleware components can share common functionality and avoid code duplication. Run the application and request the `/middleware` URL and you will see the output shown in Figure 14-8.

Figure 14-8. *Using a service in custom middleware*

Creating Short-Circuiting Middleware

The next type of middleware intercepts requests before they reach the content generation components in order to *short-circuit* the pipeline process, often for performance purposes. Listing 14-20 shows the contents of a class file called `ShortCircuitMiddleware.cs` that I added to the `Infrastructure` folder.

Listing 14-20. The Contents of the ShortCircuitMiddleware.cs File in the Infrastructure Folder

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ShortCircuitMiddleware {
        private RequestDelegate nextDelegate;

        public ShortCircuitMiddleware(RequestDelegate next) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContext) {
            if (httpContext.Request.Headers["User-Agent"]
                    .Any(h => h.ToLower().Contains("edge"))) {
                httpContext.Response.StatusCode = 403;
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

This middleware component inspects the request's `User-Agent` header, which is used by browsers to identify themselves. Using the `User-Agent` header to identify specific browsers isn't reliable enough to use in a real application, but it is sufficient for this example.

The term *short-circuiting* is used because this type of middleware doesn't always forward requests to the next component in the chain. In this case, if the `User-Agent` header contains the term `edge`, the component sets the status code to `403 - Forbidden` and doesn't forward the request to the next component. Since the request is being rejected, there is no point in allowing the request to be handled by other components, which would needlessly consume system resources. Instead, the request handling is terminated early, and the 403 response is sent to the client.

Middleware components receive requests in the order in which they are set up in the `Startup` class, which means that short-circuiting middleware must be set up before content-generating middleware, as shown in Listing 14-21.

Listing 14-21. Registering Short-Circuiting Middleware in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMiddleware<ShortCircuitMiddleware>();
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

If you run the application and request any URL using the Microsoft Edge browser, then you will see the 403 error. Requests from other browsers are ignored by the `ShortCircuitMiddleware` component and are passed on to the next component in the chain, which means that a response will be generated when the requested URL is `/middleware`. Figure 14-9 shows the addition of the short-circuiting component to the middleware pipeline.

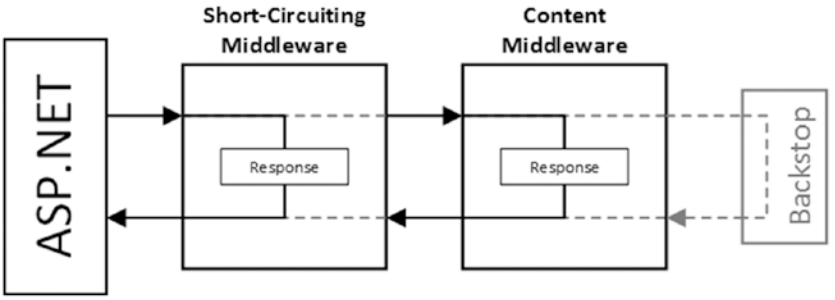

Figure 14-9. *Adding a short-circuiting component to the middleware pipeline*

Creating Request-Editing Middleware

The next type of middleware component examined doesn't generate a response. Instead, it changes requests before they reach other components later in the chain. This kind of middleware is mainly used for platform integration to enrich the ASP.NET Core representation of an HTTP request with platform-specific features. It can also be used to prepare requests so that they are easier to process by subsequent components. As a demonstration, I added the `BrowserTypeMiddleware.cs` file to the `Infrastructure` folder and used it to define the middleware component shown in Listing 14-22.

Listing 14-22. The Contents of the BrowserTypeMiddleware.cs File in the Infrastructure Folder

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class BrowserTypeMiddleware {
        private RequestDelegate nextDelegate;
```

```
        public BrowserTypeMiddleware(RequestDelegate next) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContext) {
            httpContext.Items["EdgeBrowser"]
                = httpContext.Request.Headers["User-Agent"]
                    .Any(v => v.ToLower().Contains("edge"));
            await nextDelegate.Invoke(httpContext);
        }
    }
}
```

This component inspects the `User-Agent` header of the request and looks for the term `edge`, which suggests that the request may have been made using the Edge browser. The `HttpContext` object provides a dictionary through the `Items` property that is used to pass data between components, and the outcome of the header search is stored with the key `EdgeBrowser`.

To demonstrate how middleware components can cooperate, Listing 14-23 shows the `ShortCircuitMiddleware` class, which rejects requests when they are from Edge, making its decision based on the data produced by the `BrowserTypeMiddleware` component.

Listing 14-23. Cooperating with Another Component in the ShortCircuitMiddleware.cs File

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

    public class ShortCircuitMiddleware {
        private RequestDelegate nextDelegate;

        public ShortCircuitMiddleware(RequestDelegate next) => nextDelegate = next;

        public async Task Invoke(HttpContext httpContext) {
            if (httpContext.Items["EdgeBrowser"] as bool? == true) {
                httpContext.Response.StatusCode = 403;
            } else {
                await nextDelegate.Invoke(httpContext);
            }
        }
    }
}
```

By their nature, middleware components that edit requests need to be placed before those components that they cooperate with or that rely on the changes they make. In Listing 14-24, I have registered the `BrowserTypeMiddleware` class as the first component in the pipeline.

Listing 14-24. Registering a Middleware Component in the Startup.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>();
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

Placing the component at the start of the pipeline ensures that the request has already been modified before it is received by the other components, as shown in Figure 14-10.

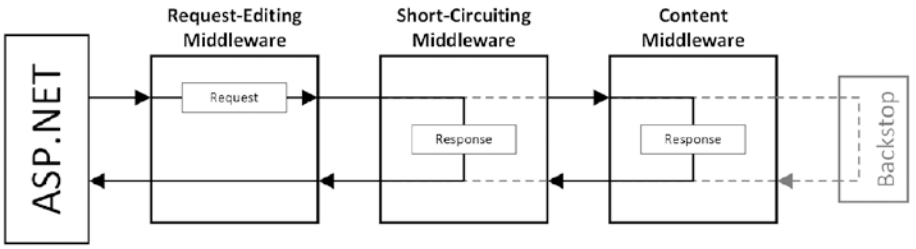

Figure 14-10. *Adding a response-editing component to the middleware pipeline*

Creating Response-Editing Middleware

The final type of middleware operates on the responses generated by other components in the pipeline. This is useful for logging details of requests and their responses or for dealing with errors. Listing 14-25 shows the contents of the `ErrorMiddleware.cs` file, which I added to the `Infrastructure` folder to demonstrate this type of middleware component.

Listing 14-25. The Contents of the ErrorMiddleware.cs File in the Infrastructure Folder

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
```

```
namespace ConfiguringApps.Infrastructure {

    public class ErrorMiddleware {
        private RequestDelegate nextDelegate;

        public ErrorMiddleware(RequestDelegate next) {
            nextDelegate = next;
        }

        public async Task Invoke(HttpContext httpContext) {
            await nextDelegate.Invoke(httpContext);

            if (httpContext.Response.StatusCode == 403) {
                await httpContext.Response
                    .WriteAsync("Edge not supported", Encoding.UTF8);
            } else if (httpContext.Response.StatusCode == 404) {
                await httpContext.Response
                    .WriteAsync("No content middleware response", Encoding.UTF8);
            }
        }
    }
}
```

The component isn't interested in a request until it has made its way through the middleware pipeline and a response has been generated. If the response status code is 403 or 404, then the component adds a descriptive message to the response. All other responses are ignored. Listing 14-26 shows the registration of the component class in the `Startup` class.

■ **Tip** You may be wondering where the `404 - Not Found` status code comes from since it isn't set by any of the three middleware components I have created. The answer is that this is how the response is configured by ASP.NET when the request enters the pipeline and is the result returned to the client if no middleware component changes the response.

Listing 14-26. Registering a Response-Editing Middleware Component in the Startup.cs File

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

namespace ConfiguringApps {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMiddleware<ErrorMiddleware>();
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>();
            app.UseMiddleware<ContentMiddleware>();
        }
    }
}
```

I registered the `ErrorMiddleware` class so that it occupies the first position in the pipeline. This may seem odd for a component that is interested only in responses, but registering the component at the start of the chain ensures that it is able to inspect the responses generated by any other component, as illustrated in Figure 14-11. If this component is placed later in the pipeline, then it will only be able to inspect responses generated by some of the other components.

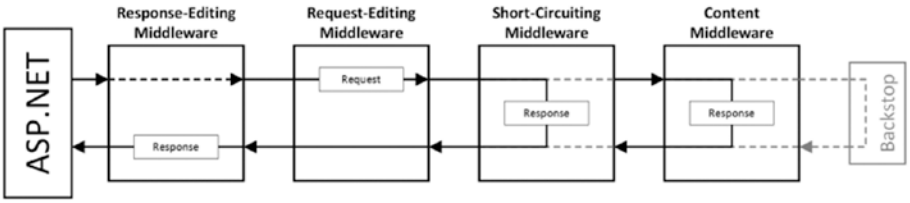

Figure 14-11. *Adding a response-editing component to the middleware pipeline*

You can see the effect of the new middleware by starting the application and requesting any URL except `/middleware`. The result will be the error message shown in Figure 14-12.

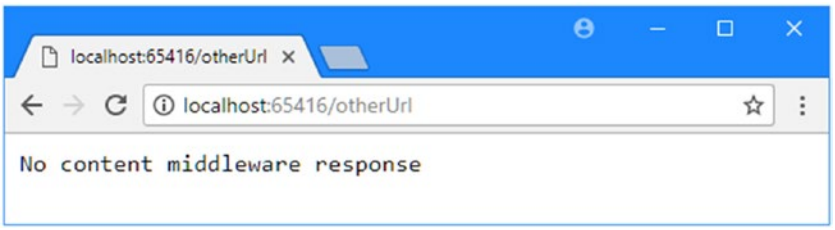

Figure 14-12. *Editing the responses of other middleware components*

Understanding How the Configure Method Is Invoked

The ASP.NET Core platform inspects the `Configure` method before it is invoked and gets a list of its arguments, which it provides using the services set up in the `ConfigureServices` method or using the special services shown in Table 14-5.

Table 14-5. *The Special Services Available As Configure Method Arguments*

Type	Description
`IApplicationBuilder`	This interface defines the functionality required to set up an application's middleware pipeline.
`IHostingEnvironment`	This interface defines the functionality required to differentiate between different types of environment, such as development and production.

Using the Application Builder

Although you don't have to define any arguments at all for the `Configure` method, most `Startup` classes will use at least the `IApplicationBuilder` interface because it allows the middleware pipeline to be created, as demonstrated earlier in the chapter. For custom middleware components, the `UseMiddleware` extension method is used to register classes. Complex content-generating middleware packages provide a single method that sets up all of their middleware components in a single step, just like they provide a single method for defining the services they use. In the case of MVC, two extension methods are available, as described in Table 14-6.

Table 14-6. *The MVC IApplicationBuilder Extension Methods*

Name	Description
`UseMvcWithDefaultRoute`	This method sets up the MVC middleware components with the default route.
`UseMvc`	This method sets up the MVC middleware components using a custom routing configuration specified using a lambda expression.

Routing is the process by which request URLs are mapped to controllers and actions are defined by the application; I describe routing in detail in Chapters 15 and 16. The `UseMvcWithDefaultRoute` method is useful for getting started with MVC development, but most applications call the `UseMvc` method, even if the result is to explicitly define the same routing configuration that would have been created by the `UseMvcWithDefaultRoute` method, as shown in Listing 14-27. This makes the routing configuration used by the application obvious to other developers and makes it easy to add new routes later (which almost all applications require at some point).

Listing 14-27. Setting Up the MVC Middleware in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {
```

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseMiddleware<ErrorMiddleware>();
            app.UseMiddleware<BrowserTypeMiddleware>();
            app.UseMiddleware<ShortCircuitMiddleware>();
            app.UseMiddleware<ContentMiddleware>();

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

Since MVC sets up content-generating middleware components, the `UseMvc` method is called after all the other middleware components have been registered. To prepare the services that MVC depends on, the `AddMvc` method must be called in the `ConfigureServices` method.

Using the Hosting Environment

The `IHostingEnvironment` interface provides some basic—but important—information about the hosting environment in which the application is running using the properties described in Table 14-7.

Table 14-7. *The IHostingEnvironment Properties*

Name	Description
`ApplicationName`	This property returns the name of the application, which is set by the hosting platform.
`EnvironmentName`	This property returns a `string` that describes the current environment, as described after this table.
`ContentRootPath`	This property returns the path that contains the application's content files and configuration files.
`WebRootPath`	This property returns a string that specifies the directory that contains the static content for the application. This is usually the `wwwroot` folder.
`ContentRootFileProvider`	This property returns an object that implements the `IFileProvider` interface and that can be used to read files from the folder specified by the `ContentRootPath` property.
`WebRootFileProvider`	This property returns an object that implements the `IFileProvider` interface and that can be used to read files from the folder specified by the `WebRootPath` property.

The `ContentRootPath` and `WebRootPath` properties are interesting but not needed in most applications because there is a built-in middleware component that can be used to deliver static content, as described in the "Enabling Static Content" section later in this chapter.

The important property is `EnvironmentName`, which allows the configuration of the application to be modified based on the environment in which it is running. There are three conventional environments (*development*, *staging*, and *production*), and each represents a commonly used environment.

The current hosting environment is set using an environment variable called `ASPNETCORE_ENVIRONMENT`. To set the environment variable, select ConfiguringApps Properties from the Visual Studio Project menu and switch to the Debug tab. Double-click the Value field for the environment variable, which is set to Development by default, and change it to Staging, as shown in Figure 14-13. Save your changes to have the new environment name take effect.

■ **Tip** Environment names are not case-sensitive, so *Staging* and *staging* are treated as the same environment. Although *development*, *staging*, and *production* are the conventional environments, you can use any name you like. This can be useful if there are multiple developers on a project and each requires different configuration settings, for example. See the "Dealing with Complex Configurations" section later in the chapter for details on how to deal with complex differences between environment configurations.

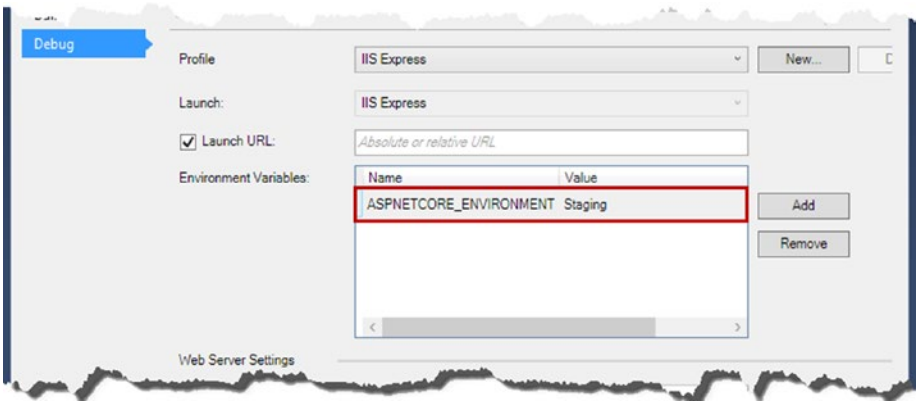

Figure 14-13. *Setting the name of the hosting environment*

Within the `Configure` method, you can determine which hosting environment is being used by reading the `IHostingEnvironment.EnvironmentName` property or using one of the extension methods that operate on `IHostingEnvironment` objects, as described in Table 14-8.

Table 14-8. *IHostingEnvironment Extension Methods*

Name	Description
`IsDevelopment()`	This method returns `true` if the hosting environment name is `Development`.
`IsStaging()`	This method returns `true` if the hosting environment name is `Staging`.
`IsProduction()`	This method returns `true` if the hosting environment name is `Production`.
`IsEnvironment(env)`	This method returns `true` if the hosting environment name matches the `env` argument.

The extension methods are used to alter the set of middleware components in the pipeline to tailor the behavior of the application to different hosting environments. In Listing 14-28, I use one of the extension methods to ensure that the custom middleware components created earlier in the chapter are only present in the pipeline in the Development hosting environment.

Listing 14-28. Using the Hosting Environment in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseMiddleware<ErrorMiddleware>();
                app.UseMiddleware<BrowserTypeMiddleware>();
                app.UseMiddleware<ShortCircuitMiddleware>();
                app.UseMiddleware<ContentMiddleware>();
            }

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

The three custom middleware components won't be added to the pipeline with the current configuration, which has set the hosting environment to Staging. If you run the application and request the /middleware URL, you will receive a 404 - Not Found error because the only middleware components available are the ones set up by the UseMvc method, which have no controllers available that can process this URL.

■ **Note** Once you have tested the effect of changing the hosting environment, be sure to change it back to Development; otherwise, the examples in the rest of the chapter won't work properly.

Adding the Remaining Middleware Components

There are a set of commonly used middleware components that are useful in most MVC projects and that I use in the examples in this book. In the sections that follow, I add these components to the request pipeline and explain how they work.

Enabling Exception Handling

Even the most carefully written application will encounter exceptions, and it is important to handle them appropriately. In Listing 14-29, I have added middleware components that deal with exceptions to the request pipeline to the `Startup` class. I have also removed the custom middleware components so that I can focus on MVC.

Listing 14-29. Adding Exception-Handling Middleware in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }

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

The UseStatusCodePages method adds descriptive messages to responses that contain no content, such as 404 - Not Found responses, which can be useful since not all browsers show their own messages to the user.

The UseDeveloperExceptionPage method sets up an error-handling middleware component that displays details of the exception in the response, including the exception trace. This isn't information that should be displayed to users, so the call to UseDeveloperExceptionPage is made only in the development hosting environment, which is detected using the IHostingEnvironmment object.

For the staging or production environment, the UseExceptionHandler method is used instead. This method sets up an error handling that allows a custom error message to be displayed that won't reveal the inner workings of the application. The argument to the UseExceptionHandler method is the URL that the client should be redirected to in order to receive the error message. This can be any URL provided by the application, but the convention is to use /Home/Error.

In Listing 14-30, I have added the ability to generate exceptions on demand to the Index action of the Home controller and have added an Error action so requests generated by the UseExceptionHandler component can be processed.

Listing 14-30. Generating and Handling Exceptions in the HomeController.cs File in the Controllers Folder

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;

        public HomeController(UptimeService up) => uptime = up;

        public ViewResult Index(bool throwException = false) {
            if (throwException) {
                throw new System.NullReferenceException();
            }
            return View(new Dictionary<string, string> {
                ["Message"] = "This is the Index action",
                ["Uptime"] = $"{uptime.Uptime}ms"
            });
        }

        public ViewResult Error() => View(nameof(Index),
            new Dictionary<string, string> {
                ["Message"] = "This is the Error action"});
            }
}
```

The changes to the Index action rely on the model binding feature, which I describe in Chapter 26, to obtain a throwException value from the request. The action throws a NullReferenceException if throwException is true and executes normally if it is false.

The Error action uses the Index view to display a simple message. You can see the effect of the different exception-handling middleware components by running the application and requesting the /Home/Index?throwException=true URL. The query string provides the value for the Index action argument, and the response that you see will depend on the hosting environment name. Figure 14-14 shows the output produced by the UseDeveloperExceptionPage (for the Development hosting environment) and UseExceptionHandler middleware (for all other hosting environments).

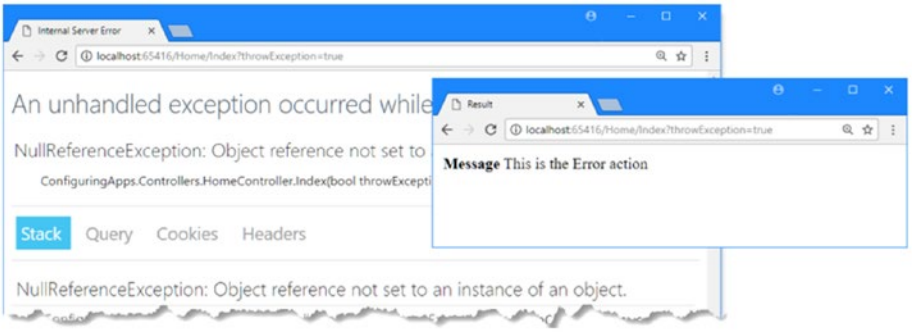

Figure 14-14. *Handling exceptions in development and staging/production*

The developer exception page provides details of the exception and options to explore its stack trace and the request that caused it. By contrast, the user exception page should be used simply to indicate that something has gone wrong.

Enabling Browser Link

I described the Browser Link feature in Chapter 6 and demonstrated how it can be used to manage browsers during development. The server-side part of Browser Link is implemented as a middleware component that must be added to the `Startup` class as part of the application configuration, without which the Visual Studio integration won't work. Browser Link is useful only during development and should not be used in staging or production because it edits the responses generated by other middleware components to insert JavaScript code that opens HTTP connections back to the server side so that it can receive reload notifications. In Listing 14-31, you can see how the `UseBrowserLink` method, which registers the middleware component, is called only for the `Development` hosting environment.

Listing 14-31. Enabling Browser Link in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }

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

Enabling Static Content

The final middleware component that is useful for most projects provides access to the files in the `wwwroot` folder so that applications can include images, JavaScript files, and CSS stylesheets. The `UseStaticFiles` method adds a component that short-circuits the request pipeline for static files, as shown in Listing 14-32.

Listing 14-32. Enabling Static Content in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
```

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

Static content is typically required regardless of the hosting environment, which is why I call the `UseStaticFiles` section for all environments. This addition means that the `link` element in the `Index` view will work properly and allow the browser to load the Bootstrap CSS style sheet. You can see the effect by starting the application, as shown in Figure 14-15.

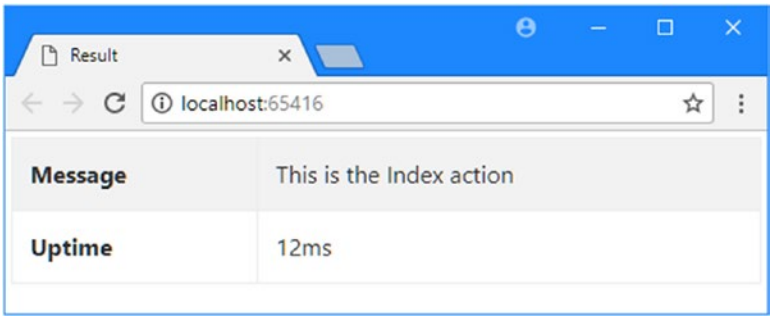

Figure 14-15. *Enabling static content*

Configuring the Application

Some configuration data changes often, such as when the application moves from the development to the production environment and different details are required for database servers. Rather than hard-code this information in the `Startup` class, ASP.NET Core allows configuration data to be provided from a range of more easily changed sources, such as environment variables, command-line arguments, and files written in the JavaScript Object Notation (JSON) format.

Configuration data is usually handled automatically, but since I have replaced the default settings in the `Program` class, I need to explicitly add the code that will get the data and make it available for use in the rest of the application, as shown in Listing 14-33.

Listing 14-33. Loading Configuration Data in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    config.AddJsonFile("appsettings.json",
                        optional: true, reloadOnChange: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

The `ConfigureAppConfiguration` method is used to handle the configuration data and its arguments are a `WebHostBuilderContext` object and an object that implements the `IConfigurationBuilder` interface. The `WebBostBuilderContext` class defines the properties described in Table 14-9.

Table 14-9. *The Properties Defined by the WebBostBuilderContext Class*

Name	Description
`HostingEnvironment`	This property returns an object that implements the `IHostingEnvironment` interface and provides information about the hosting environment in which the application is running. See the "Using the Hosting Environment" section earlier in the chapter for details.
`Configuration`	This property returns an object that implements the `IConfiguration` interface, which provides read-only access to the configuration data in the application.

The `IConfigurationBuilder` interface is used to prepare the configuration data for the rest of the application, which is typically done using extension methods. The three methods used in Listing 14-33 to add configuration data are described in Table 14-10.

Table 14-10. *The IConfigurationBuilder Extension Methods for Adding Configuration Data*

Name	Description
`AddJsonFile`	This method is used to load configuration data from a JSON file, such as `appsettings.json`.
`AddEnvironmentVariables`	This method is used to load configuration data from environment variables.
`AddCommandLine`	This method is used to load configuration data from the command-line arguments used to start the application.

Of the three methods that are used to load configuration data in Listing 14-33, it is the `AddJsonFile` method that is the most interesting. The arguments to the method specify the file name, whether the file is optional, and whether the configuration data should be reloaded if the file changes:

```
...
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
...
```

The values used for the arguments specify a file called `appsettings.json` file, which is the conventional name for the JSON configuration file. This file is optional, meaning that an exception will not be thrown if the file doesn't exist, and will be monitored for changes so that the configuration data can be refreshed automatically.

RELOADING CONFIGURATION DATA

The ASP.NET Core configuration system supports reloading data when configuration files change. Some of the built-in middleware components, such as the logging system, support this feature, which means logging levels can be changed at runtime without restarting the application. You can incorporate similar capabilities in custom middleware components as well.

But just because a feature makes something possible doesn't mean it's sensible. Making changes to configuration files on production systems is a recipe for downtime. It is all too easy to mistype the changes you want and create a malfunctioning configuration. There can be unforeseen consequences even if you make the change successfully, such as logging data filling up disks or crippling performance.

My advice is to avoid live edits and make sure all changes are pushed through your standard testing process before being deployed into production. It can be tempting to poke around a live system to diagnose a problem, but it rarely ends well. If you find yourself editing production configuration files, then you should ask yourself whether you are about to make a small problem into a much larger one.

Creating the JSON Configuration File

The most common uses for the `appsettings.json` file are to store database connection strings and logging settings, but you can store any data that your application requires.

To see how the configuration system works, add a new JSON file called `appsettings.json` to the root folder of the project with the content shown in Listing 14-34.

Listing 14-34. The Contents of the appsettings.json File in the ConfiguringApps Folder

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
}
```

The JSON format allows a structure to be defined for configuration settings. The JSON content in the listing defines a configuration category called `ShortCircuitMiddleware` that contains a configuration property called `EnableBrowserShortCircuit`, which is set to `true`.

JSON: QUOTING AND COMMAS

If you are new to working with JSON, then it is worth taking some time to read the specification at www.json.org. The format is simple to work with, and there is good support for generating and parsing JSON data on most platforms, including within MVC applications (see Chapters 20 and 21 for examples) and at the client using a simple JavaScript API. In fact, most MVC developers won't deal directly with JSON at all, and it is only in the configuration files that hand-crafting JSON is required.

There are two pitfalls that many developers new to JSON fall into, and while you should still take the time to read the specification, knowing the most common problems will give you somewhere to start when Visual Studio or ASP.NET Core can't parse your JSON files. Here is an addition to the `appsettings.json` file to show the two most common problems:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  mysetting : [ fast, slow ]
}
```

First, almost everything in JSON is quoted. It is easy to forget that you are writing C# code and expect property names and values to be accepted without quotes. In JSON, anything other than Boolean values and numbers has to be quoted, like this:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  "mysetting" : [ "fast", "slow"]
}
```

Second, when you add a new property to the JSON description of an object, you must remember to add a comma to the previous brace character, like this:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
  "mysetting" : [ "fast", "slow"]
}
```

It can be hard to see the difference even when it is highlighted—which is why it is such a common error—but I have added a comma following the } character that closes the `ShortCircuitMiddleware` section. Be careful, though, because a trailing comma that has no following section is also illegal. If your JSON changes are causing problems, there are the two errors to check for first.

Using Configuration Data

The `Startup` class can access the configuration data by defining a constructor with an `IConfiguration` argument. When the `UseStartup` method is called in the `Program` class, the configuration data prepared by the `ConfigureAppConfiguration` is used to create the `Startup` object. Listing 14-35 shows the addition of the constructor to the `Startup` class and shows how the configuration data can be accessed.

Listing 14-35. Receiving and Using Configuration Data in the Startup.cs File in the ConfiguringApps Folder

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

namespace ConfiguringApps {
    public class Startup {

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }
```

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

            if ((Configuration.GetSection("ShortCircuitMiddleware")?
                    .GetValue<bool>("EnableBrowserShortCircuit")).Value) {
                    app.UseMiddleware<BrowserTypeMiddleware>();
                    app.UseMiddleware<ShortCircuitMiddleware>();
            }

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
                app.UseBrowserLink();
            } else {
                app.UseExceptionHandler("/Home/Error");
            }
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

The `IConfiguration` object is received by the constructor and assigned to a property called `Configuration`, which can then be used to access the configuration data that has been loaded from environment variables, the command line, and the `appsettings.json` file.

To obtain a value, you navigate through the structure of the data to the configuration section you require, which is represented by another object that implements the `IConfiguration` interface, which provides a subset of members available for `IConfigurationRoot`, as shown in Table 14-11.

Table 14-11. *The Members Defined by the IConfiguration Interface*

Name	Description
`[key]`	The indexer is used to obtain a `string` value for a specific key.
`GetSection(name)`	This method returns an `IConfiguration` object that represents a section of the configuration data.
`GetChildren()`	This method returns an enumeration of the `IConfiguration` objects that represent the subsections of the current configuration object.

There are also extension methods that can be used to operate on `IConfiguration` objects to get values and convert them from strings into other types, as described in Table 14-12.

Table 14-12. *Extension Methods for the IConfiguration Interface*

Name	Description
`GetValue<T>(keyName)`	This method gets the value associated with the specified key and attempts to convert it to the type T.
`GetValue<T>(keyName, defaultValue)`	This method gets the value associated with the specified key and attempts to convert it to the type T. The default value will be used if there is no value for the key in the configuration data.

It is important not to assume that a configuration value will be specified. In the listing, I use the null conditional operator to ensure that I have received the `ShortCircuitMiddleware` section before trying to get the `EnableBrowserShortCircuit` value. The result is that the custom middleware will be added to the request pipeline only if the `ShortCircuitMiddleware/EnableBrowserShortCircuit` value has been defined and set to `true`.

Configuring Logging

ASP.NET Core includes support for capturing and handling logging data, and many of the built-in middleware components have been written to generate logging messages. Logging is set up automatically in most projects, but since I am using individual configuration statements in the `Program` class, I need to add the statements shown in Listing 14-36 to set up the logging feature.

Listing 14-36. Configuring Logging in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    config.AddJsonFile("appsettings.json",
                        optional: true, reloadOnChange: true);
```

```
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logging) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.GetSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

The `ConfigureLogging` method sets up the logging system using a lambda function that receives a `WebHostBuilderContext` object (described earlier in the chapter) and an object that implements the `ILoggingBuilder` interface. A set of extension methods operate on the `ILoggingBuilder` interface to configure the logging system, as described in Table 14-13.

Table 14-13. *Extension Methods for the ILoggingBuilder Interface*

Name	Description
`AddConfiguration`	This method is used to configure the logging system using the configuration data that has been loaded from the `appsettings.json` file, from the command line, or from environment variables.
`AddConsole`	This method sends logging messages to the console, which is useful when starting the application using the `dotnet run` command.
`AddDebug`	This method sends logging messages to the debug output window when the Visual Studio debugger is running.
`AddEventLog`	This method sends logging messages to the Windows Event Log, which is useful if you deploy to Windows Server and want the log messages from the ASP.NET Core MVC application to be incorporated with those from other types of application.

Understanding the Logging Configuration Data

The `AddConfiguration` method is used to configure the logging system using configuration data that is typically defined in the `appsettings.json` file. Listing 14-37 adds a configuration section called `Logging` to the `appsettings.json` file, which corresponds to the name used for the `AddConfiguration` method in Listing 14-36.

Listing 14-37. Adding a Configuration Section to the appsettings.json File in the ConfiguringApps Folder

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
```

```
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

The logging configuration specified the level of message that should be displayed from different sources of logging data. The logging system supports six levels of debugging information, as described in Table 14-14 in order of importance.

Table 14-14. *The ASP.NET Debugging Levels*

Level	Description
`Trace`	This level is used for messages that are useful during development but that are not required in production.
`Debug`	This level is used for detailed messages required by developers to debug problems.
`Information`	This level is used for messages that describe the general operation of the application.
`Warning`	This level is used for messages that describe events that are unexpected but that do not interrupt the application.
`Error`	This level is used for messages that describe errors that interrupt the application.
`Critical`	This level is used for messages that describe catastrophic failures.
`None`	This level is used to disable logging messages.

The `Default` entry in Listing 14-37 sets the threshold for displaying logging messages to `Debug`, which means that only messages of the `Debug` level or greater will be displayed. The remaining entries override the default for logging messages from specific namespaces so that logging messages that originate from the `System` or `Microsoft` namespaces will be displayed only if they are of the `Information` level or greater.

To see the effect of enabling logging, start the application using the Visual Studio debugger by selecting Debug ➤ Start Debugging. Look at the Output window and you will see logging messages that show how each HTTP request is handled, like this:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:65417/
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method ConfiguringApps.Controllers.HomeController.Index
      (ConfiguringApps) with arguments (False) - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
      Executing ViewResult, running view at path /Views/Home/Index.cshtml.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action ConfiguringApps.Controllers.HomeController.Index
      (ConfiguringApps) in 1597.3535ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 1695.6314ms 200 text/html; charset=utf-8
```

Creating Custom Log Messages

The logging messages in the previous section were generated by the ASP.NET Core and MVC components that handled the HTTP request and generated the response. This kind of message can provide useful information, but you can also generate custom log messages that are specific to your application, as shown in Listing 14-38.

Listing 14-38. Custom Logging in the HomeController.cs File in the Controllers Folder

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Logging;

namespace ConfiguringApps.Controllers {

    public class HomeController : Controller {
        private UptimeService uptime;
        private ILogger<HomeController> logger;

        public HomeController(UptimeService up, ILogger<HomeController> log) {
            uptime = up;
            logger = log;
        }

        public ViewResult Index(bool throwException = false) {
            logger.LogDebug($"Handled {Request.Path} at uptime {uptime.Uptime}");

            if (throwException) {
                throw new System.NullReferenceException();
            }
            return View(new Dictionary<string, string> {
                ["Message"] = "This is the Index action",
                ["Uptime"] = $"{uptime.Uptime}ms"
            });
        }

        public ViewResult Error() => View(nameof(Index),
            new Dictionary<string, string> {
                ["Message"] = "This is the Error action"});
        }
}
```

The `ILogger` interface defines the functionality required to create log entries and to obtain an object that implements this interface, and the `HomeController` class has a constructor argument whose type is `ILogger<HomeController>`. The type parameter allows the logging system to use the name of the class in the log messages, and the value for the constructor argument is provided automatically through the dependency injection feature that I describe in Chapter 18.

Once you have an ILogger, you can create log messages using extension methods defined in the Microsoft.Extensions.Logging namespace. There are methods for each of the logging levels described in Table 14-14. The HomeController class uses the LogDebug method to create a message at the Debug level. To see the effect, run the application using the Visual Studio debugger and examine the Output window for the log message, like this:

```
dbug: ConfiguringApps.Controllers.HomeController[0]
  Handled / at uptime 12
```

There are a lot of messages displayed when the application starts up, which can make it hard to pick out individual messages. It is easier to see single messages if you click the Clear All button at the top of the Output window and then reload the browser—this will ensure that only the log messages that relate to a single request are displayed.

Configuring Dependency Injection

The default configuration for ASP.NET Core applications includes preparing the service provider, which is used by the dependency injection feature that I describe in detail in Chapter 18. Listing 14-39 shows the addition of the configuration statements to the Program class.

Listing 14-39. Configuring Services in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    config.AddJsonFile("appsettings.json",
                        optional: true, reloadOnChange: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
```

```
                    config.AddCommandLine(args);
                }
            })
            .ConfigureLogging((hostingContext, logging) => {
                logging.AddConfiguration(
                    hostingContext.Configuration.GetSection("Logging"));
                logging.AddConsole();
                logging.AddDebug();
            })
            .UseIISIntegration()
            .UseDefaultServiceProvider((context, options) => {
                options.ValidateScopes =
                    context.HostingEnvironment.IsDevelopment();
            })
            .UseStartup<Startup>()
            .Build();
    }
}
}
```

The `UseDefaultServiceProvider` method uses the built-in ASP.NET Core service provider. There are alternative service providers available, but the built-in features are acceptable for most projects, and I recommend that you use a third-party component only if you have a specific problem to solve and you have a good understanding of dependency injection, which I describe in Chapter 18.

The `UseDefaultServiceProvider` accepts a lambda function that receives a `WebHostBuilderContext` object and a `ServiceProviderOptions` object, which is used to configure the built-in service provider. The only configuration property is called `ValidateScopes`, and disabling the feature is required when working with Entity Framework Core, as explained in Chapter 8.

Configuring MVC Services

When you call `AddMvc` in the `ConfigureServices` method of the `Startup` class, it sets up all the services that are required for MVC applications. This has the advantage of convenience because it registers all the MVC services in a single step but does mean that some additional work is required to reconfigure the services to change the default behavior.

The `AddMvc` method returns an object that implements the `IMvcBuilder` interface, and MVC provides a set of extension methods that can be used for advanced configuration, the most useful of which are described in Table 14-15. Many of these configuration options relate to features that I describe in detail in later chapters.

Table 14-15. *Useful IMvcBuilder Extension Methods*

Name	Description
`AddMvcOptions`	This method configures the services used by MVC, as described after the table.
`AddFormatterMappings`	This method is used to configure a feature that allows clients to specify the data format they receive, as described in Chapter 20.
`AddJsonOptions`	This method is used to configure the way that JSON data is created, as described in Chapter 20.
`AddRazorOptions`	This method is used to configure the Razor view engine, as described in Chapter 21.
`AddViewOptions`	This method is used to configure how MVC handles views, including which view engines are used. See Chapter 21 for details.

The `AddMvcOptions` method configures the most important MVC services. It accepts a function that receives an `MvcOptions` object, which provides a set of configuration properties, the most useful of which are described in Table 14-16.

Table 14-16. *Selected MvcOptions Properties*

Name	Description
`Conventions`	This property returns a list of the model conventions that are used to customize how MVC creates controllers and actions, as described in Chapter 31.
`Filters`	This property returns a list of the global filters, as described in Chapter 19.
`FormatterMappings`	This property returns the mappings used to allow clients to specify the data format they receive, as described in Chapter 20.
`InputFormatters`	This property returns a list of the objects used to parse request data, as described in Chapter 20.
`ModelValidatorProviders`	This property returns a list of the objects used to validate data, as described in Chapter 27.
`OutputFormatters`	This property returns a list of the classes that format data sent from API controllers, as described in Chapter 20.
`RespectBrowserAcceptHeader`	This property specifies whether the `Accept` header is taken into account when deciding what data format to use for a response, as described in Chapter 20.

These configuration options are used to fine-tune the way that MVC operates, and you will find detailed descriptions of the features they relate to in the chapters specified in the table. As a quick demonstration, however, Listing 14-40 shows how the `AddMvcOptions` method can be used to change a configuration option.

Listing 14-40. Changing a Configuration Option in the Startup.cs File in the ConfiguringApps Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<UptimeService>();
    services.AddMvc().AddMvcOptions(options => {
        options.RespectBrowserAcceptHeader = true;
    });
}
...
```

The lambda expression passed to the `AddMvcOptions` method receives an `MvcOptions` object, which I use to set the `RespectBrowserAcceptHeader` property to `true`. This change allows clients to have more influence over the data format selected by the content negotiation process, as described in Chapter 20.

Dealing with Complex Configurations

If you need to support a large number of hosting environments or if there are a lot of differences between your hosting environments, then using `if` statements to branch configurations in the `Startup` class can result in a configuration that is hard to read and hard to edit without causing unexpected changes. In the sections that follow, I describe different ways that the `Startup` class can be used for complex configurations.

Creating Different External Configuration Files

The default configuration for the application performed by the `Program` class looks for JSON configuration files that are specific to the hosting environment being used to run the application, so a file called `appsettings.production.json` can be used to store settings that are specific to the production platform. Listing 14-41 restores the statement that loads the JSON file to the `Program` class, which I removed at the start of the chapter.

Listing 14-41. Loading Environment Files in the Program.cs File in the ConfiguringApps Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }
```

```
        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    var env = hostingContext.HostingEnvironment;
                    config.AddJsonFile("appsettings.json",
                            optional: true, reloadOnChange: true)
                        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                            optional: true, reloadOnChange: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
                        config.AddCommandLine(args);
                    }
                })
                .ConfigureLogging((hostingContext, logging) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.GetSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
                })
                .UseIISIntegration()
                .UseDefaultServiceProvider((context, options) => {
                    options.ValidateScopes =
                        context.HostingEnvironment.IsDevelopment();
                })
                .UseStartup<Startup>()
                .Build();
        }
    }
}
```

When you load configuration data from a platform-specific file, the configuration settings it contains override any existing data with the same names. As an example, I used the ASP.NET Configuration File item template to create a file called `appsettings.development.json` with the configuration data shown in Listing 14-42. The configuration data in this file sets the `EnableBrowserShortCircuit` value to `false`.

■ **Tip** The `appsettings.development.json` file might seem to disappear after you create it. If you extend the arrow to the left of the `appsettings.json` entry in the Solution Explorer window, you will see that Visual Studio groups items with similar names together.

Listing 14-42. The Contents of the appsettings.development.json File in the ConfiguringApps Folder

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": false
  }
}
```

The appsettings.json file will be loaded when the application starts, followed by the appsettings.development.json file the application is running in the development environment. The result is that the EnableBrowserShortCircuit value will be false when the application is running in the development environment and true when in staging and production.

Creating Different Configuration Methods

Selecting different configuration data files can be useful but doesn't provide a complete solution for complex configurations because data files don't contain C# statements. If you want to vary the configuration statements used to create services or register middleware components, then you can use different methods, where the name of the method includes the hosting environment, as shown in Listing 14-43.

Listing 14-43. Using Different Method Names in the Startup.cs File

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

namespace ConfiguringApps {
    public class Startup {

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc().AddMvcOptions(options => {
                options.RespectBrowserAcceptHeader = true;
            });
        }

        public void ConfigureDevelopmentServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseExceptionHandler("/Home/Error");
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
```

```
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }

        public void ConfigureDevelopment(IApplicationBuilder app,
                IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseBrowserLink();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

When ASP.NET Core looks for the `ConfigureServices` and `Configure` methods in the `Startup` class, it first checks to see whether there are methods that include the name of the hosting environment. In the listing, I added a `ConfigureDevelopmentServices` method, which will be used instead of the `ConfigureServices` method in the `Development` environment, and a `ConfigureDevelopment` method, which will be used instead of the `Configure` method. You can define separate methods for each of the environments that you need to support and rely on the default methods being called if there are no environment-specific methods available. In the example, this means the `ConfigureServices` and `Configure` methods will be used for the staging and production environments.

■ **Caution** The default methods are not called if there are environment-specific methods defined. In Listing 14-43, for example, ASP.NET Core will not call the `Configure` method in the `Development` environment because there is a `ConfigureDevelopment` method. This means each method is responsible for the complete configuration required for its environment.

Creating Different Configuration Classes

Using different methods means you don't have to use `if` statements to check the hosting environment name, but it can result in large classes, which is a problem in itself. For especially complex configurations, the final progression is to create a different configuration class for each hosting environment. When ASP.NET looks for the `Startup` class, it first checks to see whether there is a class whose name includes the current hosting environment. To this end, I added a class file called `StartupDevelopment.cs` to the project and used it to define the class shown in Listing 14-44.

Listing 14-44. The Contents of the StartupDevelopment.cs File in the ConfiguringApps Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
    public class StartupDevelopment {
```

```
        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<UptimeService>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseBrowserLink();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

This class contains `ConfigureServices` and `Configure` methods that are specific to the development hosting environment. To enable ASP.NET to find the environment-specific `Startup` class, a change is required to the `Program` class, as shown in Listing 14-45.

Listing 14-45. Enabling Environment-Specific Startup in the Program.cs File

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
    public class Program {

        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) {

            return new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) => {
                    var env = hostingContext.HostingEnvironment;
                    config.AddJsonFile("appsettings.json",
                        optional: true, reloadOnChange: true)
                    .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                            optional: true, reloadOnChange: true);
                    config.AddEnvironmentVariables();
                    if (args != null) {
```

```
                config.AddCommandLine(args);
            }
        })
        .ConfigureLogging((hostingContext, logging) => {
            logging.AddConfiguration(
                hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
        })
        .UseIISIntegration()
        .UseDefaultServiceProvider((context, options) => {
            options.ValidateScopes =
                context.HostingEnvironment.IsDevelopment();
        })
        .UseStartup(nameof(ConfiguringApps))
        .Build();
    }
  }
}
```

Rather than specifying a class, the `UseStartup` method is given the name of the assembly that it should use. When the application starts, ASP.NET will look for a class whose name includes the hosting environment, such as `StartupDevelopment` or `StartupProduction`, and fall back to using the regular `Startup` class if one does not exist.

Summary

In this chapter, I explained how MVC applications are configured. I described the role of the `Program` and `Startup` classes and the default configuration options they provide. I showed you how requests are processed using a pipeline and how different types of middleware are used to control the flow of requests and the responses they elicit. In the next chapter, I introduce the routing system, which is how MVC deals with mapping request URLs to controllers and actions.