1

What's New in ASP.NET Core MVC

WHAT'S IN THIS CHAPTER?

- History of the .NET web stack
- Explanation of all the pieces of this new .NET Core puzzle
- Introduction to the ASP.NET Core and the new concepts it brings
- Some of the new notable features of ASP.NET Core MVC

The year 2016 is a historical milestone for Microsoft's .NET web stack, as it is the year in which Microsoft released .NET Core, a complete open-source and cross-platform framework for building applications and services. It includes ASP.NET Core and a reworked MVC framework.

This first chapter is a brief introduction to ASP.NET Core. It can be used either as a refresher if you already have experience with this new framework or as a teaser and summary if you haven't seen anything yet.

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com` Search for the book's ISBN (978-1-119-18131-6), and you will find the code in the chapter 1 download and individually named according to the names throughout the chapter.

GETTING THE NAMES RIGHT

Before delving into the new framework, it is important to get all the names and version numbers right, as for the untrained eye it can otherwise seem just a big mess.

ASP.NET Core

ASP.NET Core was released in 2016. It is a full rewrite of ASP.NET, completely open-source, cross-platform, and developed without the burden of backward compatibility. Notable features are a new execution environment, a new project and dependency management system, and a new web framework called ASP.NET Core MVC that unifies the programming model of both ASP.NET MVC and WebAPI. The rest of this chapter is mainly focused on all the features of ASP.NET Core.

.NET Core

ASP.NET Core can run on the standard .NET framework (from version 4.5 onward), but in order to be cross-platform it needed the CLR to be cross-platform as well. That's why .NET Core was released. .NET Core is a small, cloud-optimized, and modular implementation of .NET, consisting of the CoreCLR runtime and .NET Core libraries. The peculiarity is that this runtime is made of many components that can be installed separately depending on the necessary features, can be updated individually, and are bin-deployable so that different applications can run on different versions without affecting each other. And, of course, it can run on OSX and Linux.

.NET Core also provides a command-line interface (referred to as .NET CLI) that is used by both tools and end users to interact with the .NET Core SDK.

Visual Studio Code

Visual Studio Code is the cross-platform text editor developed by Microsoft for building ASP.NET Core applications (and many other frameworks and languages) without the full-fledged Visual Studio. It can also be used on OSX and Linux.

Visual Studio 2017

Visual Studio 2017 introduces a completely renewed installation procedure based on "workloads" to better tailor it to users' needs. One of these workloads, the ASP.NET one, includes integration with the most popular front-end tools and frameworks. This book covers them further in the upcoming chapters.

Versions Covered in this Book

I hope that now the version and naming madness is a bit clearer. This book covers Visual Studio 2017, ASP.NET Core (and ASP.NET Core MVC), and .NET Core, but it will not cover anything that is related to the full framework. At the end of the book, Visual Studio Code is also covered.

Figure 1-1 shows how all these components relate to each other.

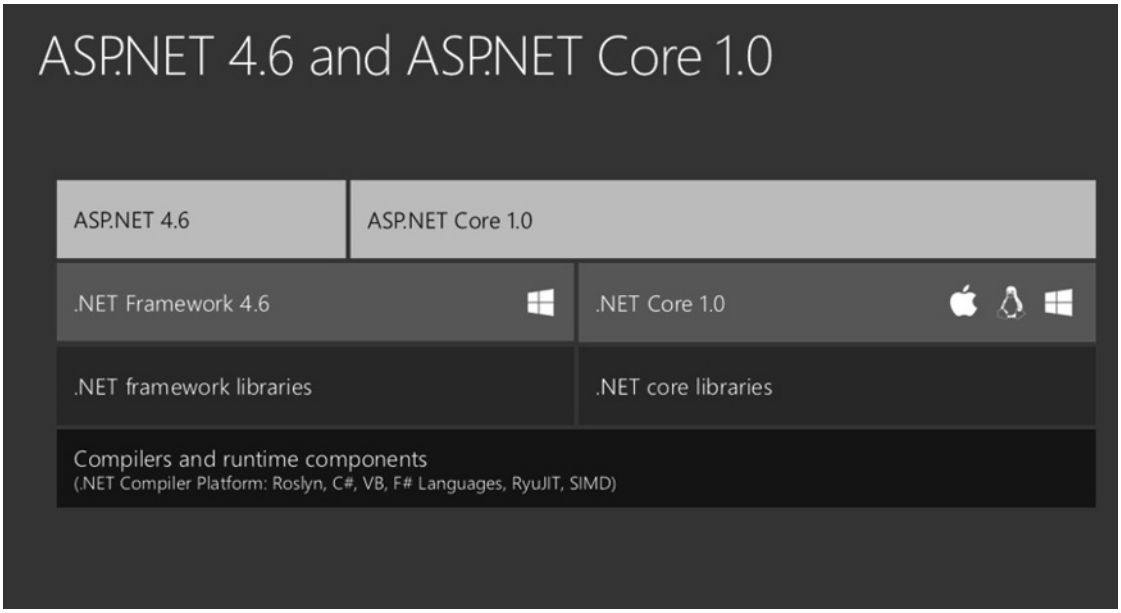

FIGURE 1-1: Diagram of the new .NET stack

A BRIEF HISTORY OF THE MICROSOFT .NET WEB STACK

Before diving into the new features of ASP.NET Core and ASP.NET Core MVC, I think it is important to look back at the evolution of the .NET web stack and the reasons why we arrived at ASP .NET Core and .NET Core.

ASP.NET Web Forms

In 2001, Microsoft released the .NET framework and its first web framework: ASP.NET Web Forms. It was developed for two types of users:

- Developers who had experience with classic ASP and were already building dynamic web sites mixing HTML and server-side code in Jscript. They were also used to interacting with the underlying HTTP connection and web server via abstractions provided by the core objects.
- Developers who were coming from the traditional WinForm application development. They didn't know anything about HTML or the web and were used to building applications by dragging UI components on a design surface.

Web Forms were designed to cater to both types of developers. Web Forms provided the abstractions to deal with HTTP and web server objects and introduced the concept of server-side events to hide the stateless nature of the web, using the ViewState. The result was a very successful, feature-rich web framework with a very approachable programming model.

It had its limitations though:

- All the core web abstractions were delivered within the System.Web library, and all the other web features depended on it.
- Because it was based on a design-time programming model, ASP.NET, the .NET framework and also Visual Studio were intimately tied. For this reason, ASP.NET had to follow the release cycle of the other products, meaning that years passed between major releases.
- ASP.NET only worked with Microsoft's web server, Internet Information Services (IIS).
- Unit testing was almost impossible and only achievable using libraries that changed the way Web Forms worked.

ASP.NET MVC

For a few years these limitations didn't cause any problems, but with other frameworks and languages pushing the evolution of web development, Microsoft started to struggle to follow their faster pace. They were all very small and focused components assembled and updated as needed, while ASP.NET was a huge monolithic framework that was difficult to update.

The problem was not only a matter of release cycles. The development style also was changing. Hiding and abstracting away the complexities of HTTP and HTML markup helped a lot of WinForm developers to become web developers, but after more than five years of experience, developers wanted more control, especially over the markup rendered on pages.

In order to solve these two problems, in 2008 the ASP.NET team developed the ASP.NET MVC framework, based on the Model-View-Controller design pattern, which was also used by many of the popular frameworks at the time. This pattern allowed a cleaner and better separation of business and presentation logic, and, by removing the server-side UI components, it gave complete control of the HTML markup to developers. Furthermore, instead of being included inside the .NET framework, it was released out of band, making faster and more frequent releases possible.

Although the ASP.NET MVC framework solved most of the problems of Web Forms, it still depended on IIS and the web abstracting library System.Web. This means that it was still not possible to have a web framework that was totally independent from the larger .NET framework.

ASP.NET Web API

Fast-forward a few years, and new paradigm for building web applications started to become widespread. These were the so-called single page applications (SPAs). Basically, instead of interconnected, server-generated, data-driven pages, applications were becoming mostly static pages where data was displayed interacting with the server via Ajax calls to web services or Web APIs. Also, many services started releasing APIs for mobile apps or third-party apps to interact with their data.

Another web framework was released to adapt better to these new scenarios: ASP.NET Web API. The ASP.NET team also took this opportunity to build an even more modular component model,

finally ditching System.Web and creating a web framework that could live its own life independently from the rest of ASP.NET and the larger .NET framework. A big role was also played by the introduction of NuGet, Microsoft's package distribution system, making it possible to deliver all these components to developers in a managed and sustainable way. One additional advantage of the break-up from System.Web was the capability to not depend on IIS anymore and to run inside custom hosts and possibly other web servers.

OWIN and Katana

ASP.NET MVC and ASP.NET Web API solved all the shortcomings of the original ASP.NET, but, as often happens, they created new ones. With the availability of lightweight hosts and the proliferation of modular frameworks, there was the real risk that application developers would need separate processes to handle all the aspects of modern applications.

In order to respond to this risk even before it became a real problem, a group of developers, taking inspiration from Rack for Ruby and partially from Node.js, came out with a specification to standardize the way frameworks and other additional components can be managed from a central hosting process. This specification is called OWIN, which stands for Open Web Interface for .NET. OWIN defines the interface that components, be they full-fledged frameworks or just small filters, have to implement in order to be instantiated and called by the hosting process.

Based on this specification, in 2014 Microsoft released Katana, an OWIN-compliant host and server, and implemented lots of connectors to allow developers to use most of its web frameworks inside Katana.

But some problems persisted. First of all, ASP.NET MVC was still tied to System.Web, so it could not run inside Katana. Also, because all the frameworks were developed at different points in time, they had different programming models. For example, both ASP.NET MVC and Web API supported dependency injection, but differently from each other. This meant that developers using both frameworks in the same application had to configure dependency injection twice, in two different ways.

The Emergence of ASP.NET Core and .NET Core

The ASP.NET team realized that there was only one way to solve all the remaining problems and at the same time make web development on .NET possible outside of Visual Studio and on other platforms. They re-wrote ASP.NET from the ground up and created a new cross-platform .NET runtime that later came to be .NET Core.

.NET CORE

Now that it is probably more clear why ASP.NET Core came to be, it is time to take a better look at .NET Core, the new entry point of this whole new stack. .NET Core is a cross-platform and open-source implementation of the .NET Standard Library, and it is made of a few components:

- The .NET Runtime, also known as CoreCLR, which implements the basic functionalities such as JIT Compilation, the base .NET types, garbage collection, and low-level classes
- CoreFX, which contains all the APIs defined in the .NET Standard Library, such as Collections, IO, Xml, async, and so on

- Tools and language compilers that allow developers to build apps
- The dotnet application host, which is used to launch .NET Core applications and the development tools

> **DEFINITION** *The .NET Standard Library is a formal specification of all the .NET APIs that can be used in all .NET runtimes. It basically enhances the CLR specifications (ECMA 335) by also defining all the APIs from the Base Class Library (BCL) that must be implemented by all .NET runtimes. The goal of such a standard is to allow the same application or library to run on different runtimes (from the standard framework to Xamarin and Universal Windows Platform).*

Getting Started with .NET Core

Installing .NET Core on Windows is pretty trivial, as it gets installed by selecting the .NET Core workload when installing Visual Studio 2017. And creating a .NET Core application is just like creating any other application with Visual Studio. Chapter 8 shows how to install .NET Core and develop applications without Visual Studio, also on a Mac, but it is important to understand how .NET Core applications are built because it will it make easier later to do the same without Visual Studio or even on another operating system.

The dotnet Command Line

The most important tool that comes with .NET Core is the `dotnet` host, which is used to launch .NET Core console applications, including the development tools, via the new .NET command-line interface (CLI). This CLI centralizes all the interactions with the framework and acts as the base layer that all other IDEs, like Visual Studio, use to build applications.

In order to try it out, just open the command prompt, create a new folder, move into this folder, and type `dotnet new console`. This command creates the skeleton of a new .NET Core console application (Listing 1-1), made of a `Program.cs` code file and the `.csproj` project definition file, named as the folder in which the command was launched.

LISTING 1-1: Sample Program.cs file

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
```

```
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The `new` command can be executed using other arguments to specify the type of project to build: `console` (the one we used before), `web`, `mvc`, `webapi`, `classlib`, `xunit` (for unit testing), and some others that are discussed further in Chapter 8. This is also the structure of all commands of the .NET CLI: `dotnet` followed by the command name, followed by its arguments.

.NET Core is a modular system, and, unlike the standard .NET framework, these modules have to be included on a one to one basis. These dependencies are defined in the `.csproj` project file and must be downloaded using another command of the .NET Core CLI: `restore`. Executing `dotnet restore` from the command prompt downloads all the dependencies needed by the application. This is needed if you add or remove dependencies while developing, but it's not strictly needed immediately after creating a new application because the `new` command runs it automatically for you.

Now that all the pieces are ready, the application can be executed by simply typing the command `dotnet run`. This first builds the application and then invokes it via the `dotnet` application host.

In fact, this could be done manually as well, first by explicitly using the `build` command and then by launching the result of the build (which is a DLL with the same name of the folder where the application has been created) using the application host: `dotnet bin\Debug\netcoreapp2.0\consoleapplication.dll` (`consoleapplication` is the name of the folder).

In addition to building and running apps, the `dotnet` command can also deploy them and create packages for sharing libraries. It can do even more thanks to its extensibility model. These topics are discussed further in Chapter 8.

INTRODUCING ASP.NET CORE

Now that you are armed with a bit of knowledge of .NET Core tooling, you can safely transition to Visual Studio and explore ASP.NET Core.

Overview of the New ASP.NET Core Web Application Project

As with previous version of the framework, you create a new ASP.NET Core application using the command menu `File` ➪ `New` ➪ `Project` and then choosing ASP.NET Core Web Application from the .NET Core group of projects.

Here you have several additional options, as shown in Figure 1-2:

- **Console App:** This creates a console application like the one in Listing 1-1.
- **Class Library:** This is a .NET Core library that can be reused in other projects.
- **Unit Test Project:** This is a test project running on the Microsoft MSTest framework.
- **xUnit Test Project:** This is another test project, but built using the xUnit OSS test framework.

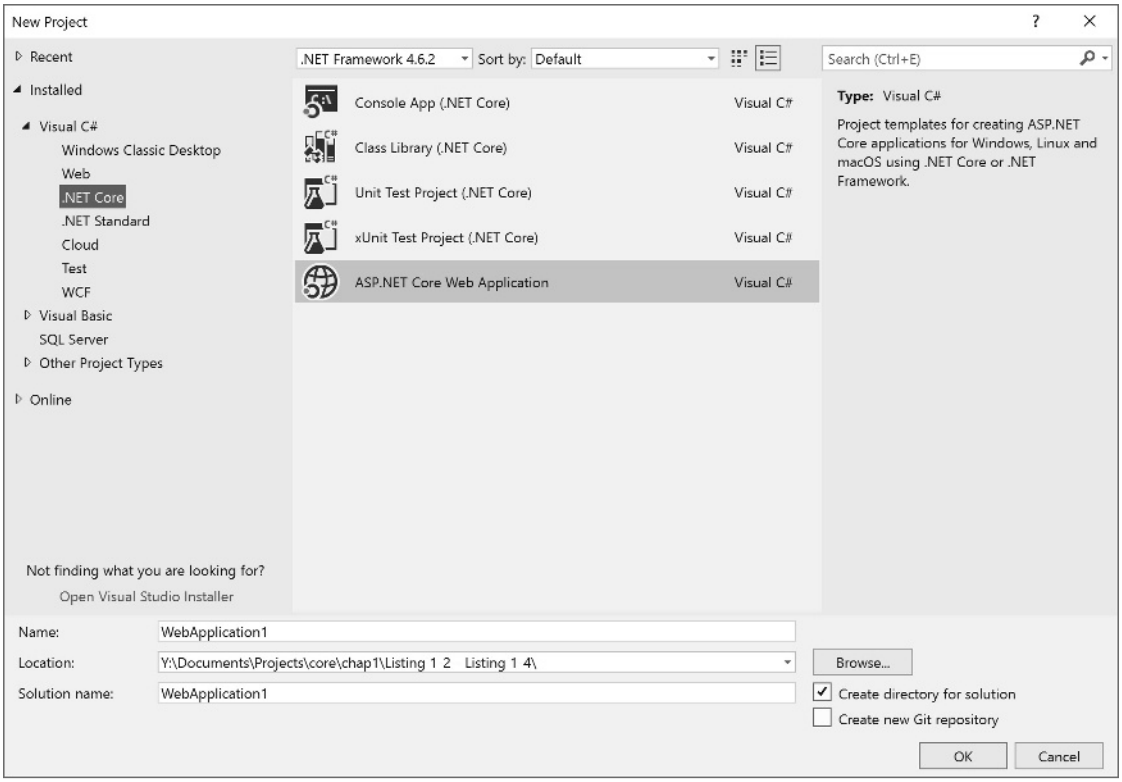

FIGURE 1-2: New Project Window

At this point you get the familiar template selection window (Figure 1-3), which by default gives you three options:

- Empty creates an ASP.NET Core project with the bare minimum to get you started.
- Web API creates an ASP.NET Core project that contains the dependencies and the skeleton on which to build a REST web application.
- Web Application creates a web application built with Razor pages, which is a simpler development paradigm that isn't covered in this book.
- Web Application (Model-View-Controller) creates the full-blown project with everything you might need in a web application.
- Angular, React.js, and React.js and Redux are project templates used to create single-page applications using these frameworks.

In addition to the authentication type, you can also choose with which version of ASP.NET Core to build the application (ASP.NET Core 1.0, 1.1, or 2.0) and whether to enable support for Docker (this last option is covered in Chapter 7).

For this initial overview, you will select the Web Application (Model-View-Controller) template and proceed through all the pieces of the puzzle.

Figure 1-4 shows the all the files and folders added to the project, and you can see that there are already a lot of changes compared to the traditional ASP.NET project. Apart from the `Controllers` and `Views` folders, all the rest is different.

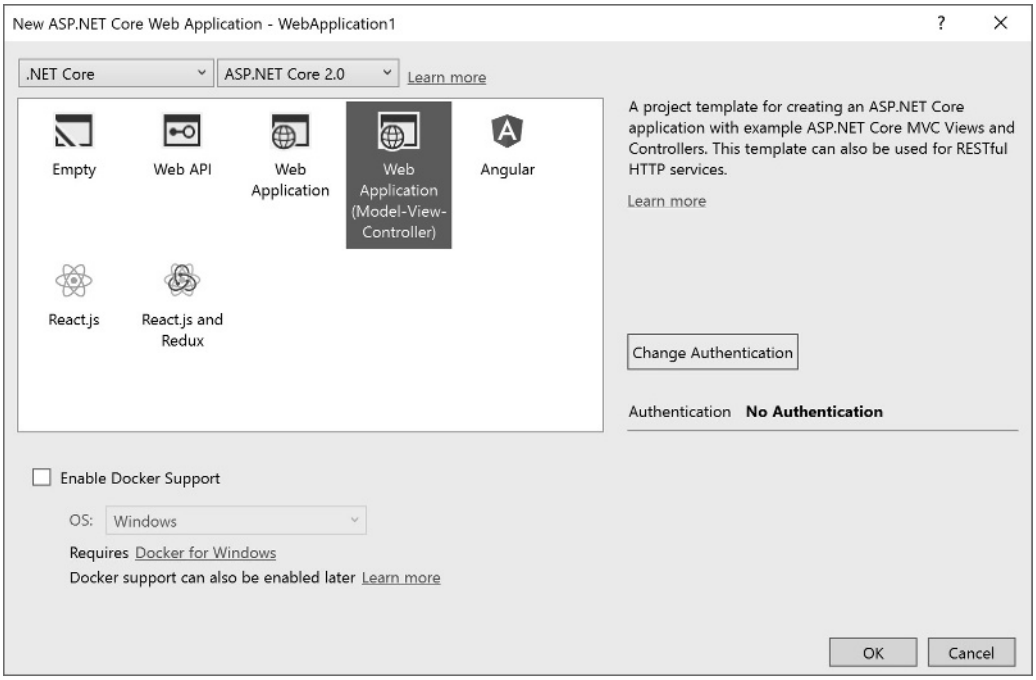

FIGURE 1-3: Web Application templates

Starting from the top, the first new element is the Connected Services node, which contains the list of extensions that connect to a third party remote service.

The next element is a node called Dependencies. This contains all the dependencies the application has, which can be .NET packages (via NuGet), Bower, as shown in Figure 1-4, or NPM if you application needs it.

A reference to Bower appears also later in the tree with the file `bower.json`, which contains the actual configuration of all the dependencies. These dependencies, once downloaded, will be stored in the `lib` folder inside the new `wwwroot` folder.

The next element is the `wwwroot` folder, which is even represented with a different "globe" icon. This is where all the static files of the application, CSS styles, images and JavaScript files, will be.

These files in the root of the project are also new additions:

- `appsettings.json` is the new location for storing application settings instead of storing them in the `appsetting` element in the `web.config`.
- `bower.json` is the configuration file for Bower dependencies.
- `bundleconfig.json` defines the configuration for bundling and minifying JavaScript and CSS files.

- `Program.cs` is where the web application starts. As mentioned earlier, the .NET Core app host can only start console applications, so web projects also need an instance of `Program.cs`.
- `Startup.cs` is the main entry point for ASP.NET Core web applications. It is used to configure how the application behaves. Thus the `Global.asax` file, which was used for this purpose before, has disappeared.
- `web.config` disappeared as it's not needed any more.

FIGURE 1-4: The elements of the new ASP.NET Core Web Application

Of the many changes introduced in the new project template, some are on the .NET side, like the `Startup.cs`, and others are in the broader web development sphere, like the introduction of Bower, the capability to include dependencies to NPM, minification, bundling, and the new approach to publishing applications.

Chapter 5 covers Bower and NPM in more detail, while Chapter 6 describes automated builds and publishing. The rest of this chapter is about all the changes introduced to the .NET side of things, starting with the `Startup.cs` file.

OWIN

In order to understand the new ASP.NET Core execution model and why there is this new `Startup.cs` file, you have to look at OWIN (Open Web Interface for .NET), the application model by which

ASP.NET Core is inspired. OWIN defines a standard way for application components to interact with each other. The specification is very simple as it basically defines only two elements: the layers of which an application is composed and how these elements communicate.

OWIN Layers

The layers are shown in Figure 1-5. They consist of the following:

- **Host:** The host is responsible for starting up the server and managing the process. In ASP .NET Core this role is implemented by the `dotnet` host application or by IIS directly.
- **Server:** This is the actual web server, the one that receives HTTP requests and sends back the responses. In ASP.NET Core there are a few implementations available. These include IIS, IIS Express, and Kestrel or WebListener when the application is run within the dotnet host in self-hosting scenarios.
- **Middleware:** Middleware is composed of pass-through components that handle all requests before delivering them to the final application. These components make up the execution pipeline of an ASP.NET Core application and can implement anything from simple logging to authentication to a full-blown web framework like ASP.NET MVC.
- **Application:** This layer is the code specific to the final application, typically built on top of one of the middleware components, like a web framework.

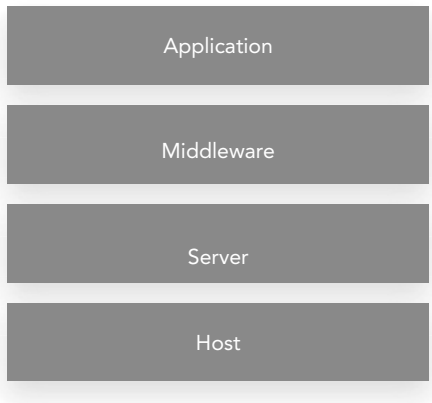

FIGURE 1-5: OWIN Layers

OWIN Communication Interface

In OWIN, all the components that are part of the pipeline communicate with each other by passing a dictionary that contains all information about the request and server state. If you want to make sure all middleware components are compatible, they must implement a delegate function called `AppFunc` (or application delegate):

```
using AppFunc = Func<
  IDictionary<string, object>, // Environment
  Task>; // Done
```

This code basically says that a middleware component must have a method that receives the `Environment` dictionary and returns a `Task` with the async operation to be executed.

> **NOTE** *The signature of* `AppFunc` *is the one defined by the OWIN specifications. While working inside ASP.NET Core, it's rarely used as the .NET Core API provides an easier way to create and register middleware components in the pipeline.*

A Better Look at Middleware

Even if not strictly standardized in the specification yet, OWIN also recommends a way to set up the application and register middleware components in the pipeline by using a builder function. Once registered, middleware components are executed one after the other until the last produces the result of the operation. At this point, middleware is executed in the opposite order until the response is sent back to the user.

An example of a typical application built with middleware might be the one shown in Figure 1-6. The request arrives, is handled by a logging component, is decompressed, passes through authentication, and finally reaches the web framework (for example ASP.NET MVC), which executes the application code. At this point the execution steps back, re-executing any post-processing steps in middleware (for example, recompressing the output or logging the time taken to execute the request) before being sent out to the user.

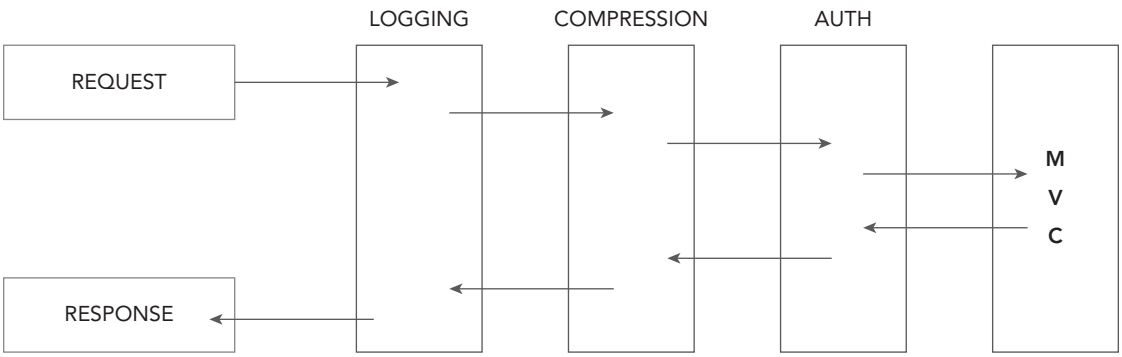

FIGURE 1-6: Execution of middleware

Anatomy of an ASP.NET Core Application

In order to better understand ASP.NET Core and its new approach to web development with .NET, it is worthwhile to create a new ASP.NET Core project. This time you will use the Empty project template to focus just on the minimum sets of files needed to start an ASP.NET Core application.

As shown in Figure 1-7, the project tree in the Solution Explorer is almost empty in comparison to the Web Application project template of Figure 1-4. The only elements needed are the `Program.cs` and `Startup.cs` code files.

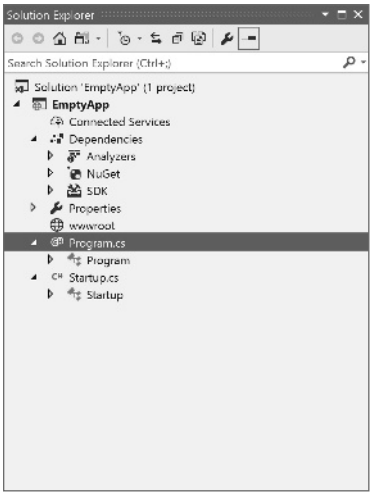

FIGURE 1-7: Empty project template

Host Builder Console Application

An ASP.NET Core application is basically a console application (Listing 1-2) that creates a web server in its `Main` method.

LISTING 1-2: Program.cs

```
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 `BuildWebHost` method is used to create the web application host using the default configuration and by specifying which class to use for the startup (`UseStartup<Startup>`).

The web host created uses Kestrel as the server, sets it up to integrate with IIS when needed, and specifies all the default configurations for logging and configuration sources.

ASP.NET Core Startup Class

The configuration of the execution pipeline of an ASP.NET Core application is done via the `Configure` method of the `Startup` class. At its simplest this method needs a parameter of type `IApplicationBuilder` to receive an instance of the application builder, which is used to assemble together all middleware components.

Listing 1-3 shows the code of the `Startup` class created by the empty project template. It has two methods, `ConfigureServices` and the aforementioned `Configure`. `ConfigureServices` is covered later in this chapter, when talking about dependency injection, so you'll focus on the `Configure` method for the moment.

LISTING 1-3: Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {

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

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

The important part of Listing 1-3 is the call to the `app.Run` method. It tells the application to run the delegate function specified in the lambda expression. In this case, this web application will always return the text string "Hello World!".

The `Run` method is used to configure *terminal* middleware, which doesn't pass the execution to the next component in the pipeline. In Listing 1-3, a specific middleware component is also added, using `app.UseDeveloperExceptionPage()`. As general rule, third-party middleware usually provides a `UseSomething` method for facilitating the registration into the pipeline. Another way of adding

custom middleware is by calling the `app.Use` method, specifying the application delegate function that should treat the request.

As you might have noticed, in Listing 1-3 the `Configure` method has an additional parameter: `IHostingEnvironment`, which provides information on the hosting environment (including the current `EnvironmentName`). You'll see more about them in a while.

NEW FUNDAMENTAL FEATURES OF ASP.NET CORE

Together with a whole new startup model, ASP.NET Core also gained some features that previously needed third-party components or some custom development:

- Easier handling of multiple environments
- Built-in dependency injection
- A built-in logging framework
- A better configuration infrastructure that is both more powerful and easier to set up and use

Environments

One of the basic features available in ASP.NET Core is the structured approach for accessing information about the environment in which the application is running. This deals with understanding whether the environment is development, staging, or production.

This information is available inside the `IHostingEnvironment` parameter passed to the `Configure` method. The current environment can be identified by simply checking its `EnvironmentName` property. For the most common environment names, there are some extension methods that make the process even easier: `IsDevelopment()`, `IsStaging()`, and `IsProduction()`, and you can use `IsEnvironment(envName)` for other more exotic names.

Once you have identified the environment, you can add features to handle different conditions based on the environment. For example, you can enable detailed errors to only be displayed in development and user-friendly messages to only be displayed in production.

If differences between the environments are even more pronounced, ASP.NET Core allows different startup classes or configuration methods per environment. For example, if a class named `StartupDevelopment` exists, this class will be used instead of the standard `Startup` class when the environment is `Development`. Likewise, the `ConfigureDevelopment()` method will be used instead of `Configure()`.

The environment is specified via the environment variable `ASPNETCORE_ENVIRONMENT`, which can be set in many different ways. For example, it can be set via the Windows Control Panel, via batch scripts (especially in servers), or directly from within Visual Studio in the project properties debug section (Figure 1-8).

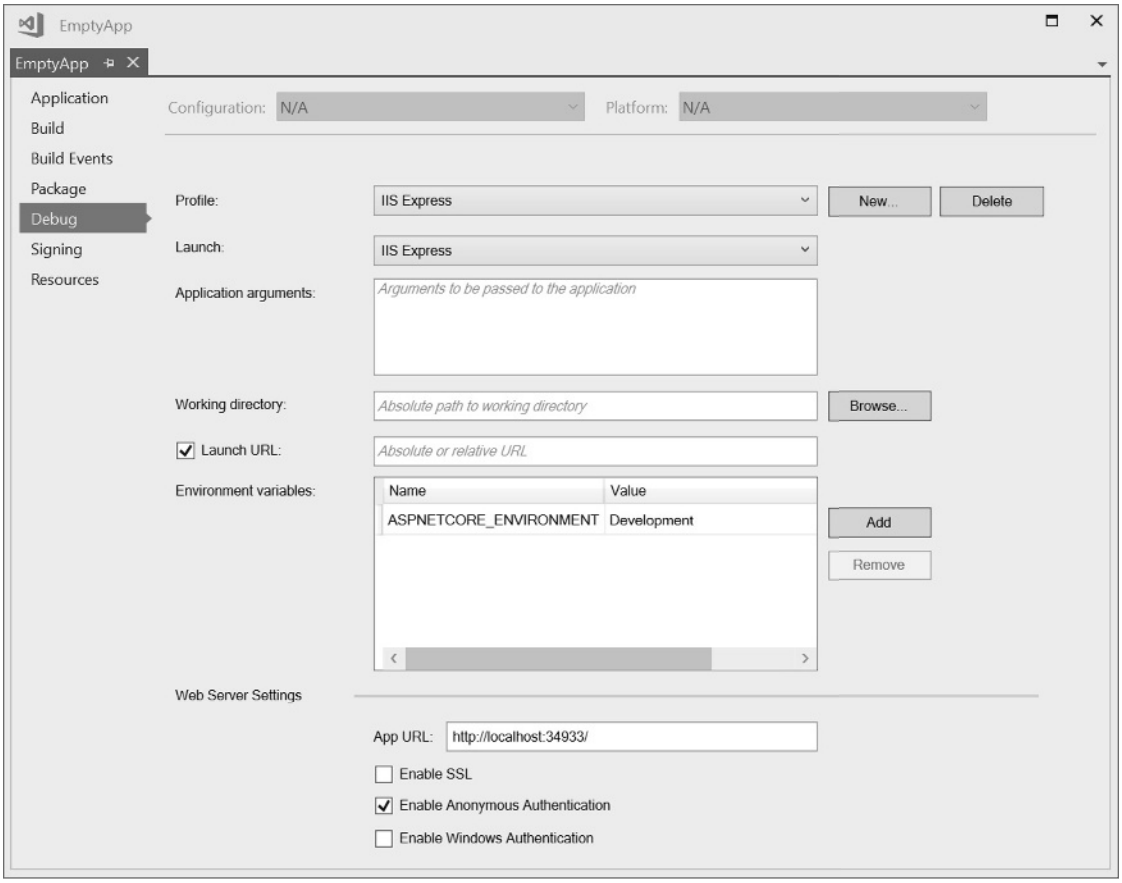

FIGURE 1-8: Project settings

Once set via the GUI, this information is stored in the `launchSettings.json` file, as shown in Listing 1-4.

LISTING 1-4: LaunchSettings.json

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:34933/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
```

```
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "EmptyApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:34934"
    }
  }
}
```

Dependency Injection

In the previous ASP.NET framework, the usage of external dependency injection libraries was left to the goodwill of developers. ASP.NET Core not only has built-in support for it, but actually requires its usage in order for applications to work.

What Is Dependency Injection?

Dependency injection (DI) is a pattern used to build loosely coupled systems. Instead of directly instantiating dependencies or accessing static instances, classes get the objects they need somehow from the outside. Typically these classes declare which objects they need by specifying them as parameters of their constructor.

Classes designed following this approach adhere to the Dependency Inversion Principle. It states that:

> *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *B. Abstractions should not depend on details. Details should depend on abstractions.*
>
> Robert C. "Uncle Bob" Martin

This also means that these classes should not require concrete objects but just their abstractions, in the form of interfaces.

The problem with systems built in this way is that, at a certain point, the number of objects to create and to "inject" into classes can become unmanageable. To handle this, you have a factory method that can take care of creating all these classes and their associated dependencies. Such a class is called a *container*. Typically containers work by keeping a list of which concrete class they have to instantiate for a given interface. Later when they are asked to create the instance of a class, they

look at all its dependencies and create them based on that list. In this way very complex graphs can also be created with just one line of code.

In addition to instantiating classes, these containers, called *Inversion of Control* or *Dependency Injection Containers* (IoC/DI containers), can also manage the lifetime of dependencies, which means that they also know whether they can reuse the same object or they must create another instance every time.

> **NOTE** *This was a very brief introduction to a very wide and complicated topic. There are numerous books on the topic, as well as lots of articles available on the Internet. In particular I suggest the articles from Robert C. "Uncle Bob" Martin or from Martin Fowler.*

Using Dependency Injection in ASP.NET Core

Despite the relative complexity of the concept, using dependency injection in ASP.NET Core is very easy. The configuration of the container is done inside the `ConfigureServices` method of the `Startup` class. The actual container is the `IServiceCollection` variable that is passed to the method as a parameter named `services`. It is to this collection that all dependencies must be added.

There are two types of dependencies: the ones needed for the framework to work and those needed by the application to work. The first type of dependencies is usually configured via extension methods like `AddService`. For example, you can add the services needed to run ASP.NET MVC by calling `services.AddMvc()`, or you can add the database context needed by the Entity Framework using `services.AddDbContext<MyDbContext>(...)`. The second type of dependencies is added by specifying an interface and one concrete type. The concrete type will be instantiated every time the container receives a request for the interface.

The syntax for adding the services depends on the kind of lifetime the service needs:

- **Transient** services are created every time they are requested and are typically used for stateless lightweight services. Such services are added using `services.AddTransient<IEmailSender,EmailSender>()`.
- **Scoped** services are created once per web request and are usually used to hold references to repositories, data access classes, or any service that keeps some state that is used for the whole duration of the request. They are registered using `services.AddScoped<IBlogRepository, BlogRepository>()`.
- **Singleton** services are created once, the first time they are requested, and later the same instance is reused for all following requests. Singletons are usually used to keep the status of an application throughout all its life. Singletons are registered using `services.AddSingleton<IApplicationCache, ApplicationCache>()`.

A typical `ConfigureServices` method for an ASP.NET Core application can look like the following snippet taken from the default project template when choosing individual user accounts:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString
        ("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

Additionally, a specific instance can be given (and in this case this is what will always be created by the container). For more complex scenarios a factory method can be configured to help the container create instances of specific services.

The usage of the dependencies is even easier. In the constructor of the class, controller, or service, just add a parameter with the type of the dependencies required. A better example is shown later in this chapter when covering the MVC framework.

Logging

ASP.NET Core comes with an integrated logging library with basic providers that write to console and to the debug output already configured as part of the setup of the default web host via the `WebHost.CreateDefaultBuilder` as seen in Listing 1-2.

Logger Instantiation

The logger is injected directly using dependency injection by specifying a parameter of type `ILogger<T>` in the constructor of the controllers or services. The dependency injection framework will provide you with a logger whose category is the full type name (for example `Wrox.FrontendDev.MvcSample.HomeController`).

Writing Log Messages

Writing messages is easily done with the extension methods provided by the built-in logging library.

```
_logger.LogInformation("Reached bottom of pipeline for request {path}", context.
Request.Path)
_logger.LogWarning("File not found")
_logger.LogError("Cannot connect to database")
```

Additional Logging Configuration

The logger is already configured by default with the console and debug providers, but additional providers and configuration can be specified.

All additional configuration must be specified in the `Program.cs` file, when setting up the web host, using the `ConfigureLogging` method.

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging((hostingContext, logging)=>
    {
       //Here goes all configuration
    })
    .Build();
```

ASP.NET Core comes with built-in providers to write to the console, the debug window, Trace, Azure App logging, and the Event Log (only on the standard framework), but if needed third-party logging providers like NLog or Serilog can be added as well.

For example, to add another provider like the one that writes to the Windows Event Log, `logging .AddEventLog()` must be called inside the `ConfigureLogging` method.

Another important configuration that must be specified is the log level that you want to write to the log files. This can be done for the whole application using the method `logging .SetMinimumLevel(LogLevel.Warning)`. In this example, only warnings, errors, or critical errors will be logged.

Configuration of the logging level can be more granular, taking into account the logger provider and the category (most of the time the name of the class from which the message originates).

Say for example that you want to send all log messages to the debug provider and in the console logger you are interested in all the messages from your own code but only in warnings or above originating from the ASP.NET Core libraries.

This is configured using *filters*. They can be specified via configuration files, with code, or even with custom functions.

```
The easiest approach is using JSON inside the standard appsettings.json
configuration file:{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Microsoft.AspNet.Core": "Warning",
        "MyCode": "Information"
      }
    },
    "LogLevel": {
      "Default": "Warning",
    }
  }
}
```

Something similar can be done by calling the `AddFilter` method in the `ConfigureLogging` method when building the web host:

```
logging.AddFilter<ConsoleLoggerProvider>("Microsoft.AspNet",LogLevel.Warning);
logging.AddFilter<DebugLoggerProvider>("Default",LogLevel.Information);
```

Both methods can be used together, and multiple filters can potentially apply to one single log message. The logging framework applies the following rules to decide which filter to apply:

1. First it selects all the filters that apply to the provider and all the ones that are specified without the provider.
2. Then the categories are evaluated and the most specific is applied. For example `Microsoft.AspNet.Core.Mvc` is more specific than `Microsoft.AspNet.Core`.
3. Finally, if multiple filters are still left, the one specified as last is taken.

Configuration

If you worked with Configuration Settings in the standard ASP.NET framework, you know that it could be very complicated to set up, apart from the simple scenarios.

The new configuration framework supports different sources of settings (XML, JSON, INI, environment variables, command-line arguments, and in-memory collections). It also automatically manages different environments and makes it very easy to create strongly-typed configuration options.

The recommended approach for using the new configuration system is to set it up when building the web host and then read it within your application, either directly or via the new strongly-typed option.

Setting Up the Configuration Sources

Because the `Configuration` class, in its simplest form, is just a key/value collection, the setup process consists of adding the sources from which all these key/value pairs must be read from. The default web host builder already sets it up for you, so you just need to know where the configuration is read from:

- The first source of the configuration is the `appsettings.json` file in the root of the project.
- Then the configuration is read from a file named `appsettings.{env.EnvironmentName}.json`.
- The configuration can be read from environment variables.
- Finally, there are also the arguments used when launching the application using the `dotnet run` command.

This setup allows the default settings defined in the first `appsettings.json` file to be overwritten in another JSON file whose name depends on the current environment and finally by a possible environment variable set on the server or argument passed to the command-line tool that runs the

application. For example, paths to folders or database connection strings can be different in different environments.

Other configuration sources are the in-memory collection source, typically used as the first source, to provide default values for settings, and the Users Secrets source, used to store sensitive information that you don't want committed to a source code repository, like password or authorization tokens.

Reading Values from Configuration

Reading the collection is also easy. Settings are read just by using their key, for example `Configuration["username"]`. If the values are from a source that allows trees of settings, like JSON files, the key is to use a concatenation of all the property names, separated by `:`, starting from the root of the hierarchy.

For example, to read the connection string defined in the setting file of Listing 1-5, the following key should be used: `ConnectionStrings:DefaultConnection`. Sections of the settings can be accessed in a similar way, but instead of using the simple dictionary key approach, the `GetSection` method must be used. For example, `Configuration.GetSection("Logging")` gets the whole subsection related to logging (which can then be passed to the logger providers instead of configuring them by code).

LISTING 1-5: Appsettings.json file of the default project template

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=aspnet-ConfigSample-c18648e9-6f7a-40e6-
b3f2-12a82e4e92eb;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Unfortunately this naive approach works only if you have access directly to the instance of the configuration class (for example, as with the `Startup` class). There are two options to share the configuration with other components. The first is to create a custom service that centralizes the access to configuration, as was done with the standard ASP.NET framework. And second, which is the new and recommended approach, is easier to configure and requires no custom coding. This approach uses `Options`.

Using Strongly-Typed Configuration

Creating strongly-typed configuration options doesn't require much more than creating the actual classes to hold the settings.

For example, say you want to access the following configuration settings.

```
"MySimpleConfiguration": "option from json file",
"MyComplexConfiguration": {
    "Username": "simonech",
    "Age": 42,
    "IsMvp": true
}
```

All you need to do is to create two classes that map the properties in the JSON file one to one, as shown in Listing 1-6.

LISTING 1-6: Options' classes (Configuration\MyOptions.cs)

```
public class MyOptions
{
    public string MySimpleConfiguration { get; set; }
    public MySubOptions MyComplexConfiguration  { get; set;}
}

public class MySubOptions
{
    public string Username { get; set; }
    public int Age { get; set; }
    public bool IsMvp { get; set; }

}
```

All that is left now is to create the binding between the configuration and the classes. This is done via the `ConfigureService` method as shown in following snippet.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddOptions();
  services.Configure<MyOptions>(Configuration);
}
```

The `AddOptions` method just adds support for injecting options into a controller or service, while the `Configure<TOption>` extension method scans the `Configuration` collection and maps its keys to the properties of the Options classes. If the collection contains keys that do not map, they are simply ignored.

If an option class is just interested in the values of a sub-section, for example `MyComplexConfiguration`, the `Configure<TOption>` extension method can be called by specifying the section to use as the configuration root, similar to what is done when configuring Logging:

```
services.Configure<MySubOptions>(Configuration.GetSection("MyComplexConfiguration"))
```

Now options are ready to be injected into any controller or service that requests them, via its constructor.

Listing 1-7 shows a controller that accesses the option class `MySubOptions` by simply adding a parameter of type `IOptions<MySubOptions>` to the constructor. Notice that it is not the actual option class to be injected but an accessor for it, so when using it the `Value` property needs to be used.

LISTING 1-7: HomeController using options

```
public class HomeController : Controller
{
    private readonly MySubOptions _options;

    public HomeController(IOptions<MySubOptions> optionsAccessor)
    {
        _optionsAccessor = optionsAccessor.Value;
    }

    public IActionResult Index()
    {
        var model = _options;
        return View(model);
    }
}
```

ALTERNATIVES TO `IOptions`

Using `IOptions` is the approach recommended by the ASP.NET Core team because it opens the door to other scenarios such as automatic reload of the configuration on change, but some people find this approach a bit too complicated.

Luckily, there are a few other alternatives, one of which is simply passing the configuration to the controllers by registering it directly in the IoC container. Most of the code is similar to what you use with `IOptions` with the exception of the `ConfigureServices` method and the `Controller`.

Instead of enabling the Options framework by calling the `AddOptions` method, you can directly bind the `Configuration` object to the strongly-typed class and then register it inside the IoC container.

```
var config = new MySubOptions();
Configuration.GetSection("MyComplexConfiguration").Bind(config);
services.AddSingleton(config);
```

This way the configuration can be used directly by the controller without going through the `IOptions` interface.

AN OVERVIEW OF SOME ASP.NET CORE MIDDLEWARE

So far the application you built doesn't do a lot of work. It just always renders a text string. But you can add more functionality simply by adding some of the middleware that has been released as part of ASP.NET Core.

Diagnostics

The first additional component you might want to add is available in the package `Microsoft.AspNetCore.Diagnostics`. There is no need to manually add the package because in ASP.NET Core 2.0 all the packages are already included as part of the `Microsoft.AspNetCore.All` metapackage.

It contains a few different components that help with handling errors. The first is the developer exception page, added to the pipeline using `UseDeveloperExceptionPage`, which is a more powerful replacement of the Yellow Page of Death, as it also shows some information on the status of the request, cookies, and headers (Figure 1-9).

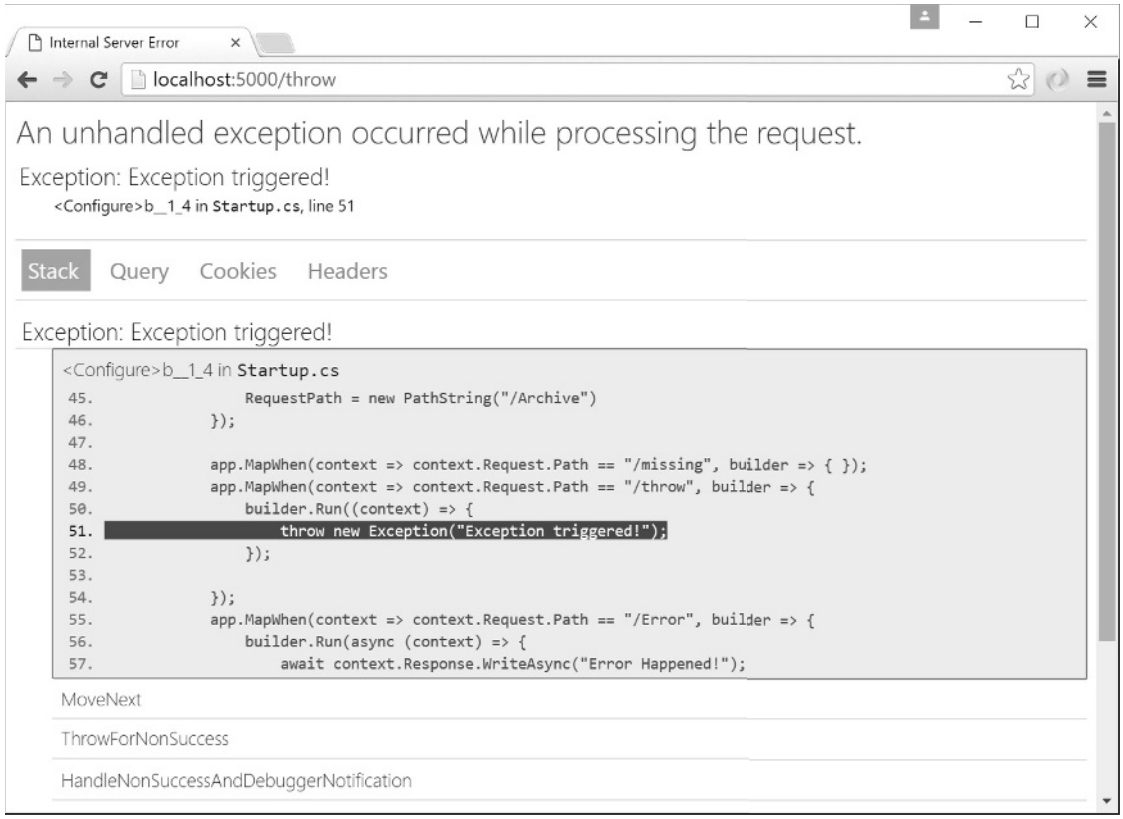

FIGURE 1-9: Developer Exception Page

This page is useful while in development, but such detailed information should never be exposed to the public. The Exception Handler middleware can be used to send users to a different page when an error happens by specifying the path to which the application has to redirect:

```
app.UseExceptionHandler("/Error")
```

If a page doesn't exist, normally the application should return an HTTP 404 status code and a page not found warning, but ASP.NET Core won't do so unless instructed. Luckily, it can be easily done as you just need to add it to the pipeline using `app.UseStatusCodePages()`.

Serving Static Files

HTML, CSS, JavaScript, and images can be served by an ASP.NET Core application by using functionalities of the `Microsoft.AspNetCore.StaticFiles` package and by registering the middleware using `app.UseStaticFiles()`.

This middleware component serves all files under the `wwwroot` folder as if they were in the root path of the application. So the `/wwwroot/index.html` file will be returned when a request for `http://example.com/index.html` arrives. Optionally, other paths can be defined for serving folders outside of `wwwroot`. The following snippet shows how to create another instance of the StaticFile middleware that serves all files under `MyArchive` when requests for the path `http://example.com/archive` arrive.

```
app.UseStaticFiles(new StaticFileOptions()
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(Directory.GetCurrentDirectory(), @"MyArchive")),
    RequestPath = new PathString("/Archive")
});
```

If you want to have `index.html` files served automatically without specifying their name, another middleware component, `UseDefaultFiles`, must be added before any `UseStaticFiles`.

Other components from this package are `UseDirectoryBrowser`, which allows browsing of files and folders, and `UseFileServer`, which adds all the functionality of the three other components (but for security reasons directory browsing is disabled by default).

> **WARNING** *There are some security considerations. The* `UseStaticFiles` *middleware doesn't perform any check on authorization rules, so all files stored under* `wwwroot` *are publicly accessible. Also, enabling directory browsing is a security risk and should not be done in a production site. If either protection of static assets or directory browsing are needed, it is better to store the files in a folder not accessible from the web and return the results via a controller action using ASP.NET Core MVC.*

Application Frameworks

The most important middleware components are the ones that completely take over the execution and host the code of the application. With ASP.NET Core there are two application frameworks available:

- **MVC** is used for building web applications that render HTML and handle user interactions.
- **Web API** is used for building RESTful web services that can be consumed by either single-page applications or by native applications on mobile or IoT devices.

These two frameworks share many concepts, and, unlike with the previous versions, the programming model has been unified so that there is almost no difference between the two.

ASP.NET CORE MVC

It might seem strange that a chapter titled "What's New in ASP.NET Core MVC" of a book that's about frontend development with ASP.NET Core MVC doesn't mention MVC almost till the end of the chapter. The reason is that almost all the new features in the updated MVC framework are related to the move from the standard ASP.NET framework to ASP.NET Core. The new startup process, the new OWIN-based execution pipeline, the new hosting model, the built-in configuration, logging, and dependency injection libraries already have been covered.

This last section of the chapter covers the new features that are specific to the MVC framework, starting from the new way of setting it up inside an ASP.NET Core application and how to define the routing table. Later it covers how to use dependency injection in controllers and ends with interesting new features related to views: view components and tag helpers.

Using the MVC Framework inside ASP.NET Core

The easiest way to start an MVC project on ASP.NET Core is to create a new project using the Web Application template. This will set up everything so that you can start right away with writing the code of the application. Most of the wiring up is done inside the `Startup` class (Listing 1-8).

LISTING 1-8: Startup class for the Web Application template

```
public class Startup
{
    public Startup(IConfigurationRoot configuration)
    {
        Configuration = configuration;
    }

    public IConfigurationRoot Configuration { get; }
```

```
    // This method gets called by the runtime. Use this method to add services to
the container.
    public void ConfigureServices(IServiceCollection services)
    {
        // Add framework services.
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }

        app.UseStaticFiles();

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

In addition to what has already been described in previous sections of this chapter (diagnostics, error handling, and serving of static files), the default template adds the `Mvc` middleware to the pipeline and the `Mvc` services to the built-in IoC container.

While adding the `Mvc` middleware, routing is configured as well. In this case the default route is specified. It matches the first segment of an URL to the controller name, the second to the action name, and the third to the argument named `id` of action method. And if they are not specified, the request will be handled by the action named `Index` and by the controller named `Home`.

It doesn't differ from the previous ASP.NET MVC framework. It's just a different way of defining the routing table. Instead of doing it inside the `global.asax` file, it is done inside the configuration of the middleware.

Using Dependency Injection in Controllers

Dependency injection was covered earlier in this chapter, together with how to add custom services into the built-in container. Now you will take a look at how to use these services inside controllers and action methods.

One of the many reasons for using an abstraction is to make it easy to test the behavior of the application. For example, if an online shop has to display a special message on the first day of spring, you probably don't want to wait till March 21st to make sure that application works correctly. So, in this case, instead of depending directly on the `System.DateTime.Today` property, it would be wiser to wrap it inside an external service so that it can later be replaced with a fake implementation that, for the purpose of testing, always returns March 21st.

This is done by defining the interface, which in this case is very simple, and by implementing it in a concrete class, as is done in Listing 1-9.

LISTING 1-9: IDateService interface and its implementations

```
public interface IDateService
{
    DateTime Today { get; }
}

public class DateService: IDateService
{
    public DateTime Today
    {
        get {
            return DateTime.Today;
        }
    }
}

public class TestDateService : IDateService
{
    public DateTime Today
    {
        get
        {
            return new DateTime(2017, 3, 21);
        }
    }
}
```

Once the interface and the concrete class are ready, the controller must be modified to allow injection into its constructor. Listing 1-10 shows how this is done.

LISTING 1-10: HomeController with constructor injection

```
public class HomeController : Controller
{
    private readonly IDateService _dateService;

    public HomeController(IDateService dateService)
    {
        _dateService = dateService;
    }
```

```
    public IActionResult Index()
    {
        var today = _dateService.Today;
        if(today.Month==3 && today.Day==21)
            ViewData["Message"] = "Spring has started, enjoy our spring sales!";
        return View();
    }
}
```

The last piece needed to tie the service and controller together is the registration of the service into the built-in IoC Container. As previously seen, this is done inside the `ConfigureServices` method, using `services.AddTransient<IDateService, DateService>()`.

Another way of using services inside an action method is via the new `[FromServices]` binding attribute. This is particularly useful if the service is used only inside one specific method and not throughout the entire controller. Listing 1-10 could be rewritten using this new attribute as shown in Listing 1-11.

LISTING 1-11: HomeController with action method parameter injection

```
public class HomeController : Controller
{
    public IActionResult Index([FromServices] IDateService dateService)
    {
        var today = dateService.Today;
        if(today.Month==3 && today.Day==21)
            ViewData["Message"] = "Spring has started, enjoy our spring sales!";
        return View();
    }
}
```

View Components

Now that you have seen the setup procedure and some new features of the controller, you will take a look at what's new on the View side, starting with View Components. They are in a way similar to Partial Views, but they are more powerful and are used in different scenarios.

Partial Views are, as the name suggests, views. They are used to split a complex view into many smaller and reusable parts. They are executed in the context of the view, so they have access to the view model, and being just razor files, they cannot have complicated logic.

View Components, on the other hand, do not have access to the view model but just to the arguments that are passed to it. They are reusable components that encapsulate both backend logic and a razor view. They are therefore made of two parts: the view component class and a razor view. They are used in the same scenarios as Child Actions, which have been removed from the MVC framework in ASP.NET Core, and are reusable portions of pages that also need some logic that might involve querying a database or web services, like sidebars, menus, and so on.

The component class inherits from `ViewComponent` and must implement the method `Invoke` or `InvokeAsync`, which returns `IViewComponentResult`. By convention view component classes are located in a `ViewComponents` folder in the root of the project, and its name must end with `ViewComponent`. Listing 1-12 shows a view component class named `SideBarViewComponent` that shows a list of links that need to appear in all the pages of the site.

LISTING 1-12: ViewComponents\SideBarViewComponent.cs file

```
namespace MvcSample.ViewComponents
{
    public class SideBarViewComponent : ViewComponent
    {
        private readonly ILinkRepository db;
        public SideBarViewComponent(ILinkRepository repository)
        {
            db = repository;
        }

        public IViewComponentResult Invoke (int max = 10)
        {
            var items = db.GetLinks().Take(max);
            return View(items);
        }
    }
}
```

As shown in the example, the view component class can make use of the dependency injection framework just like controllers do (in this case it uses a repository class that returns a list of links).

The view rendered by the view component is just like any other view, so it receives the view model specified in the `View` method, and it is accessible via the `@Model` variable. The only detail to remember is its name. By convention, the view must be `Views\Shared\Components\<ComponentName>\Default.cshtml` (so, in this example it should be `Views\Shared\Components\SideBar\Default.cshtml`), as shown in Listing 1-13.

LISTING 1-13: Views\Shared\Components\SideBar\Default.cshtml

```
@model IEnumerable<MvcSample.Model.Link>

<h2>Blog Roll</h2>
<ul>
    @foreach (var link in Model)
    {
        <li><a href="@link.Url">@link.Title</a></li>
    }
</ul>
```

Finally, to include the view component into a view, the `@Component.InvokeAsync` method must be called, providing an anonymous class with the parameters for the view component's `Invoke` method.

```
@await Component.InvokeAsync("SideBar", new { max = 5})
```

If you used Child Actions from the previous version you will immediately notice the main difference. The parameters are provided directly by the calling method and are not extrapolated by the route via model binding. This is because view components are not action methods, but are a whole new element that doesn't reuse the standard MVC execution pipeline. An added benefit is that you cannot expose these components by mistake to the web, like you could do with Child Actions if you forgot to specify the `[ChildOnly]` attribute.

Tag Helpers

Tag helpers are a new concept introduced in ASP.NET Core MVC. They are a mash-up of standard HTML tags and Razor HTML helpers, and they take the best part of both of them. Tag helpers look like standard HTML tags, so there is no more switching between writing HTML and C# code. They also have some of the server-side logic of HTML helpers, so, for example, they can read the value of the view model and conditionally add CSS classes.

Using Tag Helpers from ASP.NET Core

For example, take a look at how to write an input textbox for a form. With HTML helpers you would write `@Html.TextBoxFor(m => m.Email)`, while using tag helpers the code is `<input asp-for="Email" />`.The first case is C# code that returns HTML, while the second case is just HTML that is enhanced with some special attribute (`asp-for` in this case).

The advantage becomes more obvious when the HTML tag needs additional attributes (for example if you want to add a specific class or some `data-*` or `aria-*` attributes). With HTML helpers you would need to provide an anonymous object with all the additional attributes, while with tag helpers you write as if you were writing standard static HTML and just add the special attribute.

The differences become apparent by comparing the two syntaxes for a textbox that needs an additional class and for which you want to disable autocomplete. With HTML helpers it is:

```
@Html.TextBoxFor(m=>m.Email, new { @class = "form-control", autocomplete="off" })
```

The same textbox using a tag helper is:

```
<input asp-for="Email" class="form-control" autocomplete="off" />
```

Another added value of using tag helpers is the support inside Visual Studio. Tag helpers get IntelliSense and have a different syntax highlighting.

Figures 1-10 through 1-12 show what happens when you start typing in Visual Studio a tag that could be a tag helper. In the IntelliSense list, you can identify which tags could be tag helpers because of the new icon (the `@` sign with `< >` angular brackets). Once you select the tag, IntelliSense shows all the possible attributes, again identifying the tag helpers with the new icon. Finally, when the attribute is typed as well, Visual Studio recognizes it as a tag helper; it colorizes it differently and also provides IntelliSense for the value of the property `asp-for`.

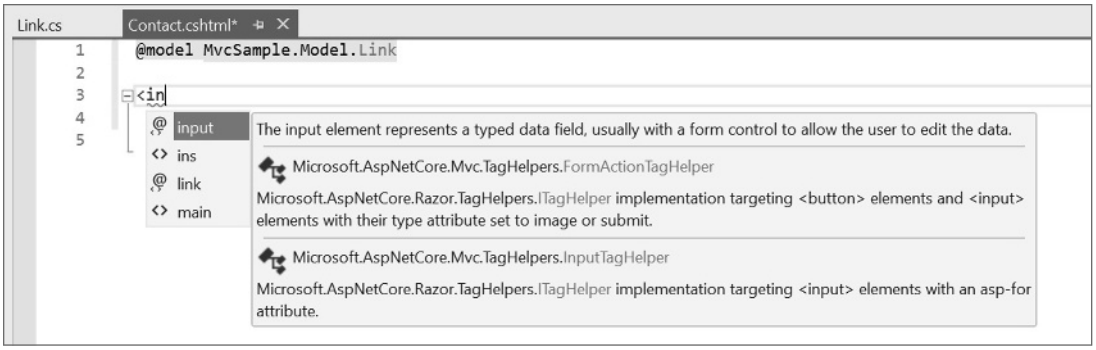

FIGURE 1-10: Identifying which tags can be tag helpers

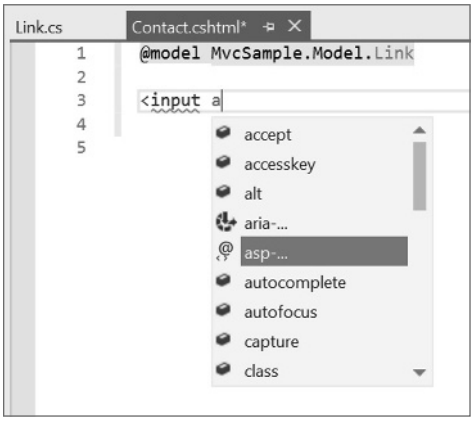

FIGURE 1-11: Attributes for a tag

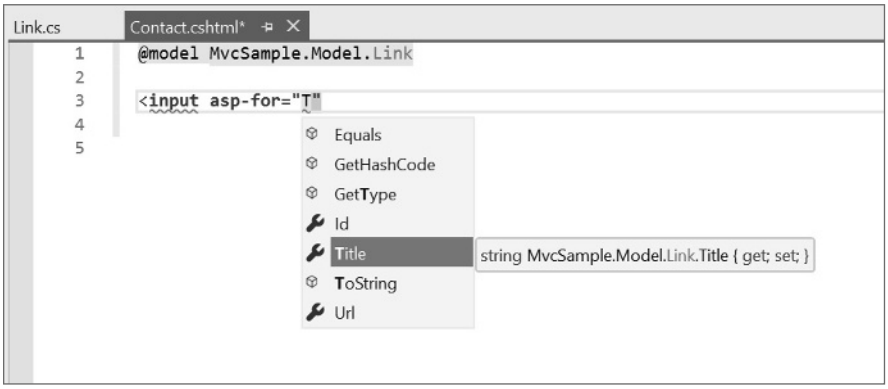

FIGURE 1-12: A well-typed attribute

ASP.NET Core MVC comes with many tag helpers for rendering forms but also for other tasks, such as an `Image` tag helper that can also add a version number to the URL to make sure it is not cached or an `Environment` tag helper for conditionally rendering different HTML fragments depending on which environment it is.

Writing Custom Tag Helpers

In addition to the one available built-in, custom tag helpers can be easily created. They are very useful when you need to output a long and repetitive piece of HTML code that changes very little from one instance to another.

To see how to build a custom tag helper, let's make one that automatically creates an email link by specifying the email address. We'll create something that converts `<email>info@wrox.com</email>` to `<a href="mailto:info@wrox.com">info@wrox.com</a>`.

The tag helper is a class, named `<Helper>TagHelper`, that inherits from `TagHelper` and implements the `Process` or `ProcessAsync` methods.

Those two methods have the two following parameters:

- `context` contains the information on the current execution context.
- `output` contains a model of the original HTML and has to be modified by the tag helper.

Listing 1-14 shows the full code for the tag helper.

LISTING 1-14: EmailTagHelper.cs

```
public class EmailTagHelper: TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
    {
        output.TagName = "a";
        var content = await output.GetChildContentAsync();
        output.Attributes.SetAttribute("href", "mailto:"+content.GetContent());
    }
}
```

Let's see what the code does.

The first line replaces the tag name (which in our case is `email`) with the one needed in the HTML code. Since we are generating a link, it must be an `a` tag.

The second line gets the content of the element. This is done using the `GetChildContentAsync` method, which also takes care of executing any Razor expression present.

Finally, the `href` attribute is set to the previously retrieved string.

Before using the newly created tag helper, we must instruct the framework where to look for tag helpers. This is done in the `_ViewImports.cshtml` file. See Listing 1-15.

LISTING 1-15: _ViewImports.cshtml

```
@using MvcSample
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "*, MvcSample"
```

The first line is added by the default project and is needed to be able to use the built-in tag helpers, while the second instructs the framework to look for new tag helpers in any class of the project.

Finally we can use the tag helper by typing the following:

```
<email>info@wrox.com</email>
```

In addition to this sample, Chapter 4 shows the code for a tag helper that renders a Bootstrap component.

View Components as Tag Helpers

We've seen how to add a View Component in a razor view by using the `InvokeAsync` method. But starting with ASP.NET Core 1.1, View Components can also be included using the same syntax as tag helpers (and IntelliSense) by appending the prefix `vc`.

With this syntax, the View Component of Listings 1-12 and 1-13 can also be instantiated using `<vc:sidebar max="5"></sidebar>`, and also gets IntelliSense as shown in Figure 1-13.

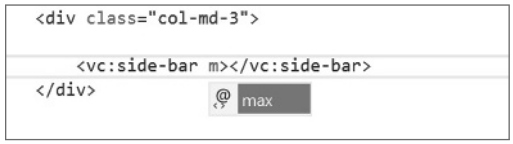

FIGURE 1-13: IntelliSense on View Components

Web API

Unlike previous versions of Web API, with ASP.NET Core, Web API applications reuse all the same features and configurations of MVC ones.

For example, to write an API that returns the list of links used in the side bar of Listing 1-12, you just need to create a controller that adheres to the Web API routing conventions and that specifies the HTTP verbs to which each action responds. See Listing 1-16.

LISTING 1-16: LinksController.cs

```
[Route("api/[controller]")]
public class LinksController : Controller
{
    private readonly ILinkRepository db;
    public LinksController(ILinkRepository repository)
    {
```

```
            db = repository;
        }

        [HttpGet]
        public IEnumerable<Link> Get()
        {
            return db.GetLinks();
        }

        [HttpGet("{id}")]
        public Link Get(int id)
        {
            return db.GetLinks().SingleOrDefault(l=>l.Id==id);
        }
    }
```

This controller will respond to HTTP GET requests to the URL `/api/Links` by returning the list of all links in JSON format, and to `/api/Links/4` by returning the link whose id is 4. This behavior is specified by the `Route` attribute, which configures the name of the API method, and by the `HttpGet` method, which specifies which action to execute when the API is called with GET.

SUMMARY

ASP.NET Core introduces a new, more modern framework that encourages writing code of good quality thanks to the built-in support for dependency injection and the easy-to-use component model. Together with this better framework, the whole development experience got an overhaul. New command-line-based developer tools make it possible to develop with a lighter IDE, and the introduction of elements typical of the front-end development world like Bower, NPM, and Gulp make the new .NET stack more appealing for developers coming from different backgrounds.

But all these changes also bring new challenges. .NET developers have to evolve and start learning new technologies and get fluent in other languages. The rest of the book covers in detail all these new technologies and languages that are now required in order to be a skilled .NET web developer.