MANNING

SAMPLE CHAPTER

ASP.NET Core IN ACTION

Andrew Lock

ASP.NET Core in Action

by Andrew Lock

Chapter 1

brief contents

Part 1

Getting started with MVC

Web applications are everywhere these days, from social media web apps and news sites, to the apps on your phone. Behind the scenes, there is almost always a server running a web application or web API. Web applications are expected to be infinitely scalable, deployed to the cloud, and highly performant. Getting started can be overwhelming at the best of times and doing so with such high expectations can be even more of a challenge.

The good news for you as readers is that ASP.NET Core was designed to meet those requirements. Whether you need a simple website, a complex e-commerce web app, or a distributed web of microservices, you can use your knowledge of ASP.NET Core to build lean web apps that fit your needs. ASP.NET Core lets you build and run web apps on Windows, Linux, or macOS. It's highly modular, so you only use the components you need, keeping your app as compact and performant as possible.

In part 1, you'll go from a standing start all the way to building your first web applications and APIs. Chapter 1 gives a high-level overview of ASP.NET Core, which you'll find especially useful if you're new to web development in general. You'll get your first glimpse of a full ASP.NET Core application in chapter 2, in which we look at each component of the app in turn and see how they work together to generate a response.

Chapter 3 looks in detail at the middleware pipeline, which defines how incoming web requests are processed and how a response is generated. We take a detailed look at one specific piece of middleware, the MVC middleware, in chapters 4 through 6. This is the main component used to generate responses in ASP.NET Core apps, so we examine the behavior of the middleware itself, routing,

and model binding. In Chapters 7 and 8, we look at how to build a UI for your application using the Razor syntax and Tag Helpers, so that users can navigate and interact with your app. Finally, in chapter 9, we explore specific features of ASP.NET Core that let you build web APIs, and how that differs from building UI-based applications.

There's a lot of content in part 1, but by the end, you'll be well on your way to building simple applications with ASP.NET Core. Inevitably, I gloss over some of the more complex configuration aspects of the framework, but you should get a good understanding of the MVC middleware and how you can use it to build dynamic web apps. In later parts of this book, we'll dive deeper into the framework, where you'll learn how to configure your application and add extra features, such as user profiles.

1 Getting started with ASP.NET Core

This chapter covers

- What is ASP.NET Core?
- How ASP.NET Core works
- Choosing between .NET Core and .NET Framework
- Preparing your development environment

Choosing to learn and develop with a new framework is a big investment, so it's important to establish early on whether it's right for you. In this chapter, I'll provide some background about ASP.NET Core, what it is, how it works, and why you should consider it for building your web applications.

If you're new to .NET development, this chapter will help you to choose a development platform for your future apps. For existing .NET developers, I'll also provide guidance on whether now is the right time to consider moving your focus to .NET Core, and the advantages ASP.NET Core can bring over previous versions of ASP.NET.

By the end of this chapter, you should have a good idea of the model you intend to follow, and the tools you'll need to get started—so without further ado, let's dive in!

1.1 An introduction to ASP.NET Core

ASP.NET Core is the latest evolution of Microsoft's popular ASP.NET web framework, released in June 2016. Recent versions of ASP.NET have seen many incremental updates, focusing on high developer productivity and prioritizing backwards compatibility. ASP.NET Core bucks that trend by making significant architectural changes that rethink the way the web framework is designed and built.

ASP.NET Core owes a lot to its ASP.NET heritage and many features have been carried forward from before, but ASP.NET Core is a new framework. The whole technology stack has been rewritten, including both the web framework and the underlying platform.

At the heart of the changes is the philosophy that ASP.NET should be able to hold its head high when measured against other modern frameworks, but that existing .NET developers should continue to be left with a sense of familiarity.

1.1.1 Using a web framework

If you're new to web development, it can be daunting moving into an area with so many buzzwords and a plethora of ever-changing products. You may be wondering if they're all necessary—how hard can it be to return a file from a server?

Well, it's perfectly possible to build a static web application without the use of a web framework, but its capabilities will be limited. As soon as you want to provide any kind of security or dynamism, you'll likely run into difficulties, and the original simplicity that enticed you will fade before your eyes!

Just as you may have used desktop or mobile development frameworks for building native applications, ASP.NET Core makes writing web applications faster, easier, and more secure. It contains libraries for common things like

- Creating dynamically changing web pages
- Letting users log in to your web app
- Letting users use their Facebook account to log in to your web app using OAuth
- Providing a common structure to build maintainable applications
- Reading configuration files
- Serving image files
- Logging calls to your web app

The key to any modern web application is the ability to generate dynamic web pages. A *dynamic web page* displays different data depending on the current logged-in user, for example, or it could display content submitted by users. Without a dynamic framework, it wouldn't be possible to log in to websites or to have any sort of personalized data displayed on a page. In short, websites like Amazon, eBay, and Stack Overflow (seen in figure 1.1) wouldn't be possible.

Hopefully, it's clear that using a web framework is a sensible idea for building high-quality web applications. But why choose ASP.NET Core? If you're a C# developer, or even if you're new to the platform, you've likely heard of, if not used, the previous version of ASP.NET—so why not use that instead?

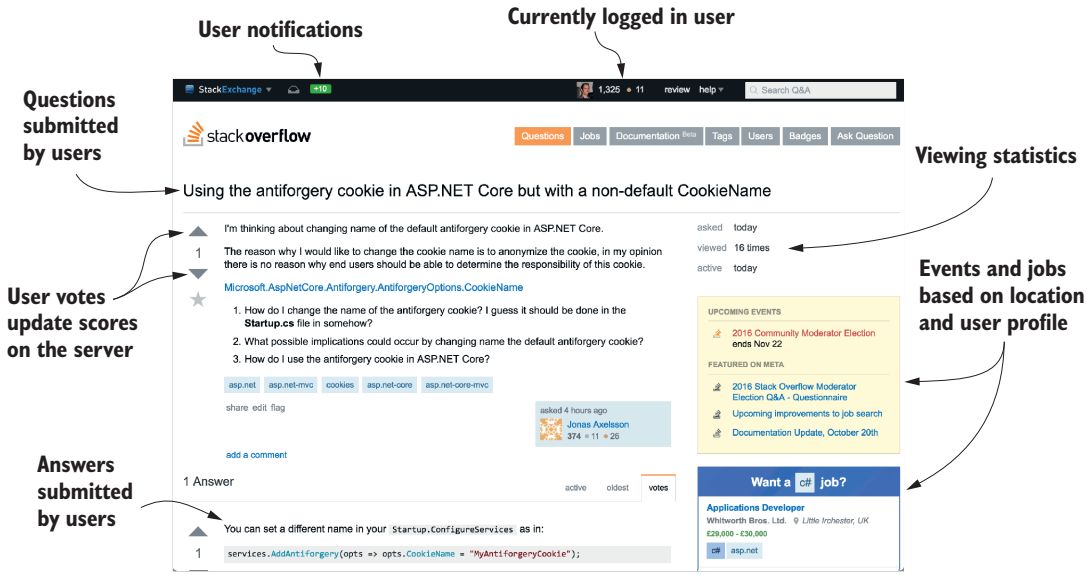

Figure 1.1 The Stack Overflow website (http://stackoverflow.com) is built using ASP.NET and is almost entirely dynamic content.

1.1.2 The benefits and limitations of ASP.NET

To understand *why* Microsoft decided to build a new framework, it's important to understand the benefits and limitations of the existing ASP.NET web framework.

The first version of ASP.NET was released in 2002 as part of .NET Framework 1.0, in response to the then conventional scripting environments of classic ASP and PHP. ASP.NET Web Forms allowed developers to rapidly create web applications using a graphical designer and a simple event model that mirrored desktop application-building techniques.

The ASP.NET framework allowed developers to quickly create new applications, but over time, the web development ecosystem changed. It became apparent that ASP.NET Web Forms suffered from many issues, especially when building larger applications. In particular, a lack of testability, a complex stateful model, and limited influence over the generated HTML (making client-side development difficult) led developers to evaluate other options.

In response, Microsoft released the first version of ASP.NET MVC in 2009, based on the Model-View-Controller pattern, a common web design pattern used in other frameworks such as Ruby on Rails, Django, and Java Spring. This framework allowed you to separate UI elements from application logic, made testing easier, and provided tighter control over the HTML-generation process.

ASP.NET MVC has been through four more iterations since its first release, but they have all been built on the same underlying framework provided by the System .Web.dll file. This library is part of .NET Framework, so it comes pre-installed with all

versions of Windows. It contains all the core code that ASP.NET uses when you build a web application.

This dependency brings both advantages and disadvantages. On the one hand, the ASP.NET framework is a reliable, battle-tested platform that's a great choice for building modern applications on Windows. It provides a wide range of features, which have seen many years in production, and is well known by virtually all Windows web developers.

On the other hand, this reliance is limiting—changes to the underlying System.Web.dll are far-reaching and, consequently, slow to roll out. This limits the extent to which ASP.NET is free to evolve and results in release cycles only happening every few years. There's also an explicit coupling with the Windows web host, Internet Information Service (IIS), which precludes its use on non-Windows platforms.

In recent years, many web developers have started looking at cross-platform web frameworks that can run on Windows, as well as Linux and macOS. Microsoft felt the time had come to create a framework that was no longer tied to its Windows legacy, thus ASP.NET Core was born.

1.1.3 What is ASP.NET Core?

The development of ASP.NET Core was motivated by the desire to create a web framework with four main goals:

- To be run and developed cross-platform
- To have a modular architecture for easier maintenance
- To be developed completely as open source software
- To be applicable to current trends in web development, such as client-side applications and deploying to cloud environments

In order to achieve all these goals, Microsoft needed a platform that could provide underlying libraries for creating basic objects such as lists and dictionaries, and performing, for example, simple file operations. Up to this point, ASP.NET development had always been focused, and dependent, on the Windows-only .NET Framework. For ASP.NET Core, Microsoft created a lightweight platform that runs on Windows, Linux, and macOS called .NET Core, as shown in figure 1.2.

.NET Core shares many of the same APIs as .NET Framework, but it's smaller and currently only implements a subset of the features .NET Framework provides, with the goal of providing a simpler implementation and programming model. It's a completely new platform, rather than a fork of .NET Framework, though it uses similar code for many of its APIs.

With .NET Core alone, it's possible to build console applications that run cross-platform. Microsoft created ASP.NET Core to be an additional layer on top of console applications, such that converting to a web application involves adding and composing libraries, as shown in figure 1.3.

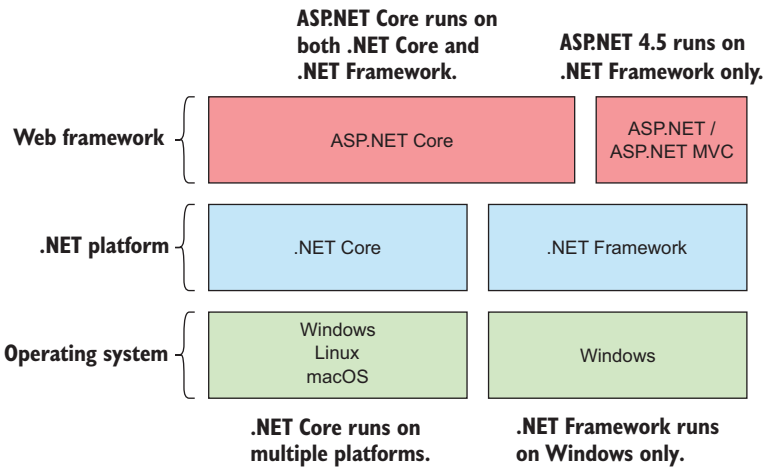

Figure 1.2 The relationship between ASP.NET Core, ASP.NET, .NET Core, and .NET Framework. ASP.NET Core runs on both .NET Framework and .NET Core, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so is tied to the Windows OS.

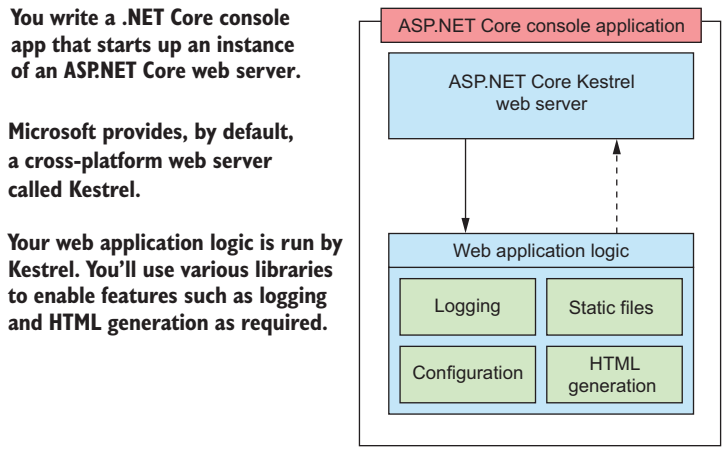

Figure 1.3 The ASP.NET Core application model. The .NET Core platform provides a base console application model for running command-line apps. Adding a web server library converts this into an ASP.NET Core web app. Additional features, such as configuration and logging, are added by way of additional libraries.

By adding an ASP.NET Core web server to your .NET Core app, your application can run as a web application. ASP.NET Core is composed of many small libraries that you can choose from to provide your application with different features. You'll rarely need all the libraries available to you and you only add what you need. Some of the libraries are common and will appear in virtually every application you create, such as the ones for reading configuration files or performing logging. Other libraries build on top of these base capabilities to provide application-specific functionality, such as third-party logging-in via Facebook or Google.

Most of the libraries you'll use in ASP.NET Core can be found on GitHub, in the Microsoft ASP.NET Core organization repositories at https://github.com/aspnet. You can find the core libraries here, such as the Kestrel web server and logging libraries, as well as many more peripheral libraries, such as the third-party authentication libraries.

All ASP.NET Core applications will follow a similar design for basic configuration, as suggested by the common libraries, but in general the framework is flexible, leaving you free to create your own code conventions. These common libraries, the extension libraries that build on them, and the design conventions they promote make up the somewhat nebulous term ASP.NET Core.

1.2 When to choose ASP.NET Core

Hopefully, you now have a general grasp of what ASP.NET Core is and how it was designed. But the question remains: should you use it? Microsoft will be heavily promoting ASP.NET Core as its web framework of choice for the foreseeable future, but switching to or learning a new web stack is a big ask for any developer or company. This section describes some of the highlights of ASP.NET Core and gives advice on the sort of applications you should build with it, as well as the sort of applications you should avoid.

1.2.1 What type of applications can you build?

ASP.NET Core provides a generalized web framework that can be used on a variety of applications. It can most obviously be used for building rich, dynamic websites, whether they're e-commerce sites, content-based sites, or large n-tier applications—much the same as the previous version of ASP.NET.

Currently, there's a comparatively limited number of third-party libraries available for building these types of complex applications, but there are many under active development. Many developers are working to port their libraries to work with ASP.NET Core—it will take time for more to become available. For example, the open source content management system (CMS), Orchard[1] (figure 1.4), is currently available as a beta version of Orchard Core, running on ASP.NET Core and .NET Core.

Traditional, server-side-rendered web applications are the bread and butter of ASP.NET development, both with the previous version of ASP.NET and ASP.NET

[1] The Orchard project (www.orchardproject.net/). Source code at https://github.com/OrchardCMS/.

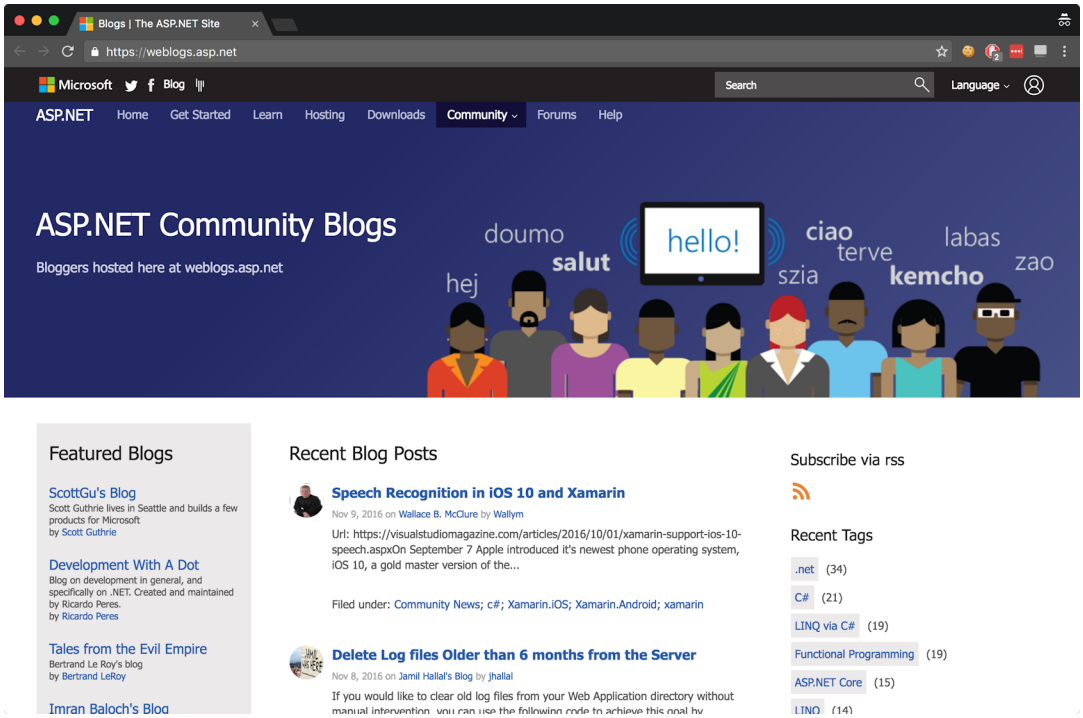

Figure 1.4 The ASP.NET Community Blogs website (https://weblogs.asp.net) is built using the Orchard CMS. Orchard 2 is available as a beta version for ASP.NET Core development.

Core. Additionally, single-page applications (SPAs), which use a client-side framework that commonly talks to a REST server, are easy to create with ASP.NET Core. Whether you're using Angular, Ember, React, or some other client-side framework, it's easy to create an ASP.NET Core application to act as the server-side API.

> **DEFINITION** *REST* stands for REpresentational State Transfer. RESTful applications typically use lightweight and stateless HTTP calls to read, post (create/update), and delete data.

ASP.NET Core isn't restricted to creating RESTful services. It's easy to create a web service or remote procedure call (RPC)-style service for your application, depending on your requirements, as shown in figure 1.5. In the simplest case, your application might expose only a single endpoint, narrowing its scope to become a microservice. ASP.NET Core is perfectly designed for building simple services thanks to its cross-platform support and lightweight design.

You should consider multiple factors when choosing a platform, not all of which are technical. One example is the level of support you can expect to receive from its creators. For some organizations, this can be one of the main obstacles to adopting open source software. Luckily, Microsoft has pledged to provide full support for each

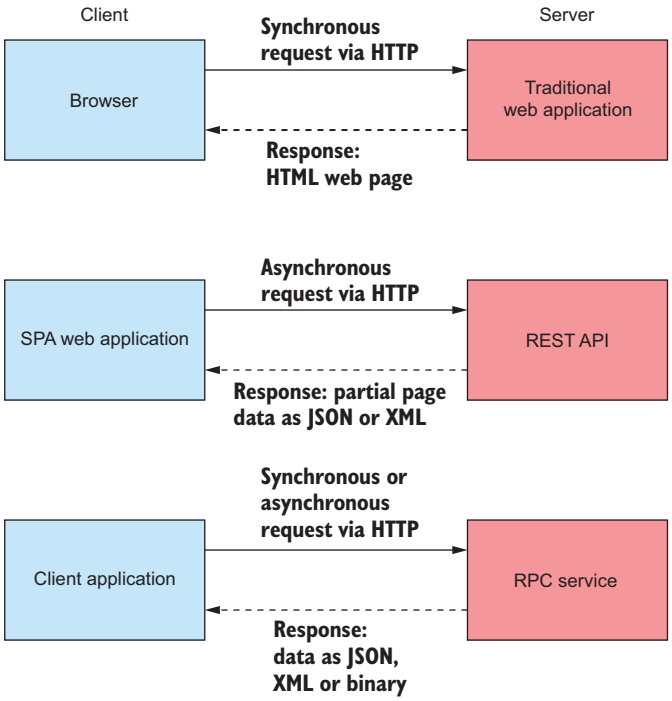

Figure 1.5 ASP.NET Core can act as the server-side application for a variety of different clients: it can serve HTML pages for traditional web applications, it can act as a REST API for client-side SPA applications, or it can act as an ad-hoc RPC service for client applications.

major and minor point release of the ASP.NET Core framework for three years[2]. And as all development takes place in the open, you can sometimes get answers to your questions from the general community, as well as Microsoft directly.

When deciding whether to use ASP.NET Core, you have two primary dimensions to consider: whether you're already a .NET developer, and whether you're creating a new application or looking to convert an existing one.

1.2.2 If you're new to .NET development

If you're new to .NET development and are considering ASP.NET Core, then welcome! Microsoft is pushing ASP.NET Core as an attractive option for web development beginners, but taking .NET cross-platform means it's competing with many other frameworks on their own turf. ASP.NET Core has many selling points when compared to other cross-platform web frameworks:

- It's a modern, high-performance, open source web framework.
- It uses familiar design patterns and paradigms.

[2] View the support policy at www.microsoft.com/net/core/support.

- C# is a great language (or you can use VB.NET or F# if you prefer).
- You can build and run on any platform.

ASP.NET Core is a re-imagining of the ASP.NET framework, built with modern software design principles on top of the new .NET Core platform. Although new in one sense, .NET Core has drawn significantly from the mature, stable, and reliable .NET Framework, which has been used for well over a decade. You can rest easy knowing that by choosing ASP.NET Core and .NET Core, you'll be getting a dependable platform as well as a fully-featured web framework.

Many of the web frameworks available today use similar, well-established design patterns, and ASP.NET Core is no different. For example, Ruby on Rails is known for its use of the Model-View-Controller (MVC) pattern; Node.js is known for the way it processes requests using small discrete modules (called a pipeline); and dependency injection is found in a wide variety of frameworks. If these techniques are familiar to you, you should find it easy to transfer them across to ASP.NET Core; if they're new to you, then you can look forward to using industry best practices!

> **NOTE** You'll encounter MVC in chapter 4, a pipeline in chapter 3, and dependency injection in chapter 10.

The primary language of .NET development, and ASP.NET Core in particular, is C#. This language has a huge following, and for good reason! As an object-oriented C-based language, it provides a sense of familiarity to those used to C, Java, and many other languages. In addition, it has many powerful features, such as Language Integrated Query (LINQ), closures, and asynchronous programming constructs. The C# language is also designed in the open on GitHub, as is Microsoft's C# compiler, code-named Roslyn.[3]

> **NOTE** I will use C# throughout this book and will highlight some of the newer features it provides, but I won't be teaching the language from scratch. If you want to learn C#, I recommend *C# in Depth* by Jon Skeet (Manning, 2008).

One of the major selling points of ASP.NET Core and .NET Core is the ability to develop and run on any platform. Whether you're using a Mac, Windows, or Linux, you can run the same ASP.NET Core apps and develop across multiple environments. As a Linux user, a wide range of distributions are supported (RHEL, Ubuntu, Debian, CentOS, Fedora, and openSUSE, to name a few), so you can be confident your operating system of choice will be a viable option. Work is even underway to enable ASP.NET Core to run on the tiny Alpine distribution, for truly compact deployments to containers.

[3] The C# language and .NET Compiler Platform GitHub source code repository can be found at https://github.com/dotnet/roslyn.

Built with containers in mind

Traditionally, web applications were deployed directly to a server, or more recently, to a virtual machine. Virtual machines allow operating systems to be installed in a layer of virtual hardware, abstracting away the underlying hardware. This has several advantages over direct installation, such as easy maintenance, deployment, and recovery. Unfortunately, they're also heavy both in terms of file size and resource use.

This is where containers come in. Containers are far more lightweight and don't have the overhead of virtual machines. They're built in a series of layers and don't require you to boot a new operating system when starting a new one. That means they're quick to start and are great for quick provisioning. Containers, and Docker in particular, are quickly becoming the go-to platform for building large, scalable systems.

Containers have never been a particularly attractive option for ASP.NET applications, but with ASP.NET Core, .NET Core, and Docker for Windows, that's all changing. A lightweight ASP.NET Core application running on the cross-platform .NET Core framework is perfect for thin container deployments. You can learn more about your deployment options in chapter 16.

As well as running on each platform, one of the selling points of .NET is the ability to write and compile only once. Your application is compiled to Intermediate Language (IL) code, which is a platform-independent format. If a target system has the .NET Core platform installed, then you can run compiled IL from any platform. That means you can, for example, develop on a Mac or a Windows machine and deploy *the exact same files* to your production Linux machines. This compile-once, run-anywhere promise has finally been realized with ASP.NET Core and .NET Core.

1.2.3 If you're a .NET Framework developer creating a new application

If you're currently a .NET developer, then the choice of whether to invest in ASP.NET Core for new applications is a question of timing. Microsoft has pledged to provide continued support for the older ASP.NET framework, but it's clear their focus is primarily on the newer ASP.NET Core framework. In the long term then, if you want to take advantage of new features and capabilities, it's likely that ASP.NET Core will be the route to take.

Whether ASP.NET Core is right for you largely depends on your requirements and your comfort with using products that are early in their lifecycle. The main benefits over the previous ASP.NET framework are

- Cross-platform development and deployment
- A focus on performance as a feature
- A simplified hosting model
- Regular releases with a shorter release cycle
- Open source
- Modular features

As a .NET developer, if you aren't using any Windows-specific constructs, such as the Registry, then the ability to build and deploy applications cross-platform opens the door to a whole new avenue of applications: take advantage of cheaper Linux VM hosting in the cloud, use Docker containers for repeatable continuous integration, or write .NET code on your Mac without needing to run a Windows virtual machine. ASP.NET Core, in combination with .NET Core, makes all of this possible.

It's important to be aware of the limitations of cross-platform applications—not all the .NET Framework APIs are available in .NET Core. It's likely that most of the APIs you need will make their way to .NET Core over time, but it's an important point to be aware of. See the "Choosing a platform for ASP.NET Core" section later in this chapter to determine if cross-platform is a viable option for your application.

NOTE With the release of .NET Core 2.0 in August 2017, the number of APIs available dramatically increased, more than doubling the API surface area.

The hosting model for the previous ASP.NET framework was a relatively complex one, relying on Windows IIS to provide the web server hosting. In a cross-platform environment, this kind of symbiotic relationship isn't possible, so an alternative hosting model has been adopted, one which separates web applications from the underlying host. This opportunity has led to the development of Kestrel: a fast, cross-platform HTTP server on which ASP.NET Core can run.

Instead of the previous design, whereby IIS calls into specific points of your application, ASP.NET Core applications are console applications that self-host a web server and handle requests directly, as shown in figure 1.6. This hosting model is conceptually much simpler and allows you to test and debug your applications from the command line, though it doesn't remove the need to run IIS (or equivalent) in production, as you'll see in section 1.3.

Changing the hosting model to use a built-in HTTP web server has created another opportunity. Performance has been somewhat of a sore point for ASP.NET applications in the past. It's certainly possible to build high-performing applications—Stack Overflow (http://stackoverflow.com) is testament to that—but the web framework itself isn't designed with performance as a priority, so it can end up being somewhat of an obstacle.

To be competitive cross-platform, the ASP.NET team have focused on making the Kestrel HTTP server as fast as possible. TechEmpower (www.techempower.com/benchmarks) has been running benchmarks on a whole range of web frameworks from various languages for several years now. In Round 13 of the plain text benchmarks, TechEmpower announced that ASP.NET Core with Kestrel was the fastest mainstream full-stack web framework, in the top ten of all frameworks![4]

[4] As always in web development, technology is in a constant state of flux, so these benchmarks will evolve over time. Although ASP.NET Core may not maintain its top ten slot, you can be sure that performance is one of the key focal points of the ASP.NET Core team.

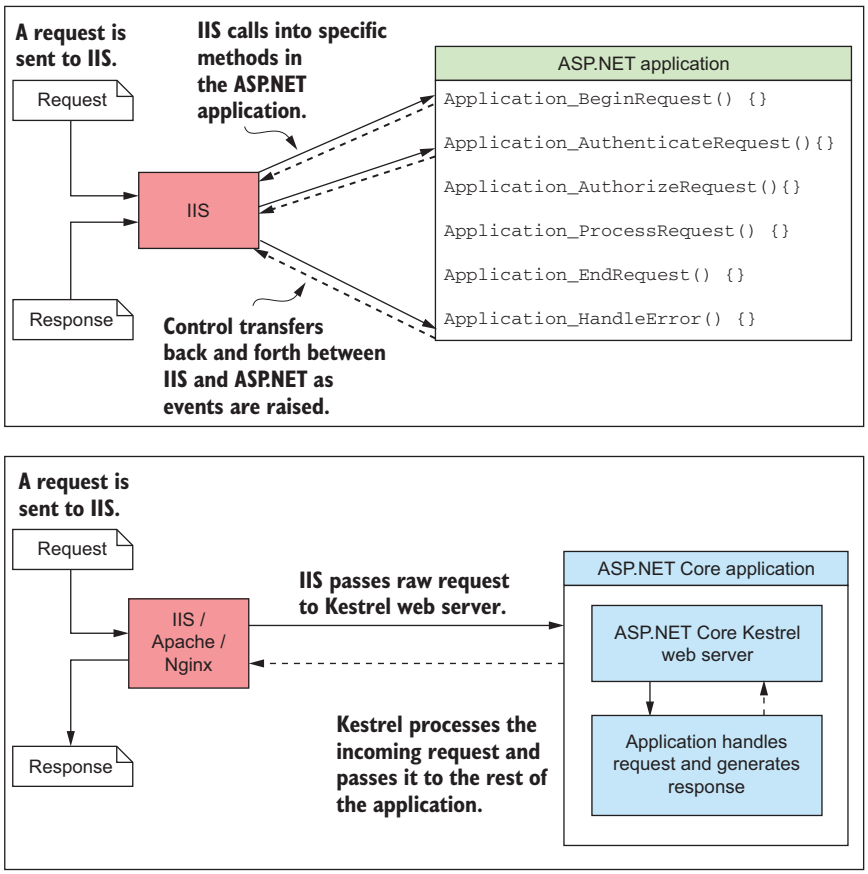

Figure 1.6 The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). With the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response, but has no deeper knowledge of the application.

Web servers: naming things is hard

One of the difficult aspects of programing for the web is the confusing array of often conflicting terminology. For example, if you've used IIS in the past, you may have described it as a web server, or possibly a web host. Conversely, if you've ever built an application using Node.js, you may have also referred to that application as a web server. Alternatively, you may have called the physical machine on which your application runs a web server!

Similarly, you may have built an application for the internet and called it a website or a web application, probably somewhat arbitrarily based on the level of dynamism it displayed.

In this book, when I say "web server" in the context of ASP.NET Core, I am referring to the HTTP server that runs as part of your ASP.NET Core application. By default, this is the Kestrel web server, but that's not a requirement. It would be possible to write a replacement web server and substitute if for Kestrel if you desired.

The web server is responsible for receiving HTTP requests and generating responses. In the previous version of ASP.NET, IIS took this role, but in ASP.NET Core, Kestrel is the web server.

I will only use the term web application to describe ASP.NET Core applications in this book, regardless of whether they contain only static content or are completely dynamic. Either way, they're applications that are accessed via the web, so that name seems the most appropriate!

Many of the performance improvements made to Kestrel did not come from the ASP.NET team themselves, but from contributors to the open source project on GitHub.[5] Developing in the open means you typically see fixes and features make their way to production faster than you would for the previous version of ASP.NET, which was dependent on .NET Framework and, as such, had long release cycles.

In contrast, ASP.NET Core is completely decoupled from the underlying .NET platform. The entire web framework is implemented as NuGet packages, independent of the underlying platform on which it builds.

NOTE NuGet is a package manager for .NET that enables importing libraries into your projects. It's equivalent to Ruby Gems, npm for JavaScript, or Maven for Java.

To enable this, ASP.NET Core was designed to be highly modular, with as little coupling to other features as possible. This modularity lends itself to a pay-for-play approach to dependencies, where you start from a bare-bones application and only add the additional libraries you require, as opposed to the kitchen-sink approach of previous ASP.NET applications. Even MVC is an optional package! But don't worry, this approach doesn't mean that ASP.NET Core is lacking in features; it means you need to opt in to them. Some of the key infrastructure improvements include

- Middleware "pipeline" for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (Web API) infrastructure
- Highly extensible configuration system
- Scalable for cloud platforms by default using asynchronous programming

[5] The Kestrel HTTP server GitHub project can be found at https://github.com/aspnet/KestrelHttpServer.

Each of these features was possible in the previous version of ASP.NET but required a fair amount of additional work to set up. With ASP.NET Core, they're all there, ready, and waiting to be connected!

Microsoft fully supports ASP.NET Core, so if you have a new system you want to build, then there's no significant reason not to. The largest obstacle you're likely to come across is a third-party library holding you back, either because they only support older ASP.NET features, or they haven't been converted to work with .NET Core yet.

Hopefully, this section has whetted your appetite with some of the many reasons to use ASP.NET Core for building new applications. But if you're an existing ASP.NET developer considering whether to convert an existing ASP.NET application to ASP.NET Core, that's another question entirely.

1.2.4 Converting an existing ASP.NET application to ASP.NET Core

In contrast with new applications, an existing application is presumably already providing value, so there should always be a tangible benefit to performing what may amount to a significant rewrite in converting from ASP.NET to ASP.NET Core. The advantages of adopting ASP.NET Core are much the same as for new applications: cross-platform deployment, modular features, and a focus on performance. Determining whether or not the benefits are sufficient will depend largely on the particulars of your application, but there are some characteristics that are clear indicators *against* conversion:

- Your application uses ASP.NET Web Forms
- Your application is built using WCF or SignalR
- Your application is large, with many "advanced" MVC features

If you have an ASP.NET Web Forms application, then attempting to convert it to ASP.NET Core isn't advisable. Web Forms is inextricably tied to System.Web.dll, and as such will likely never be available in ASP.NET Core. Converting an application to ASP.NET Core would effectively involve rewriting the application from scratch, not only shifting frameworks but also shifting design paradigms. A better approach would be to slowly introduce Web API concepts and try to reduce the reliance on legacy Web Forms constructs such as ViewData. You can find many resources online to help you with this approach, in particular, the www.asp.net/web-api website.

Similarly, if your application makes heavy use of SignalR, then now may not be the time to consider an upgrade. ASP.NET Core SignalR is under active development but has only been released in alpha form at the time of writing. It also has some significant architectural changes compared to the previous version, which you should take into account.

Windows Communication Foundation (WCF) is currently not supported either, but it's possible to consume WCF services by jumping through some slightly obscure hoops. Currently, there's no way to host a WCF service from an ASP.NET Core application, so if you need the features WCF provides and can't use a more conventional REST service, then ASP.NET Core is probably best avoided.

If your application was complex and made use of the previous MVC or Web API extensibility points or message handlers, then porting your application to ASP.NET Core could be complex. ASP.NET Core is built with many similar features to the previous version of ASP.NET MVC, but the underlying architecture is different. Several of the previous features don't have direct replacements, and so will require rethinking.

The larger the application, the greater the difficulty you're likely to have converting your application to ASP.NET Core. Microsoft itself suggests that porting an application from ASP.NET MVC to ASP.NET Core is at least as big a rewrite as porting from ASP.NET Web Forms to ASP.NET MVC. If that doesn't scare you, then nothing will!

So, when *should* you port an application to ASP.NET Core? As I've already mentioned, the best opportunity for getting started is on small, green-field, new projects instead of existing applications. That said, if the application in question is small, with little custom behavior, then porting *might* be a viable option. Small implies reduced risk and probably reduced complexity. If your application consists primarily of MVC or Web API controllers and associated Razor views, then moving to ASP.NET Core may be feasible.

1.3 How does ASP.NET Core work?

By now, you should have a good idea of what ASP.NET Core is and the sort of applications you should use it for. In this section, you'll see how an application built with ASP.NET Core works, from the user requesting a URL, to a page being displayed on the browser. To get there, first you'll see how an HTTP request works for any web server, and then you'll see how ASP.NET Core extends the process to create dynamic web pages.

1.3.1 How does an HTTP web request work?

As you know, ASP.NET Core is a framework for building web applications that serve data from a server. One of the most common scenarios for web developers is building a web app that you can view in a web browser. The high-level process you can expect from any web server is shown in figure 1.7.

The process begins when a user navigates to a website or types a URL in their browser. The URL or web address consists of a *hostname* and a *path* to some resource on the web app. Navigating to the address in your browser sends a request from the user's computer to the server on which the web app is hosted, using the HTTP protocol.

> **DEFINITION** The *hostname* of a website uniquely identifies its location on the internet by mapping via the Domain Name Service (DNS) to an IP Address. Examples include microsoft.com, www.google.co.uk, and facebook.com.

The request passes through the internet, potentially to the other side of the world, until it finally makes its way to the server associated with the given hostname on which the web app is running. The request is potentially received and rebroadcast at multiple routers along the way, but it's only when it reaches the server associated with the hostname that the request is processed.

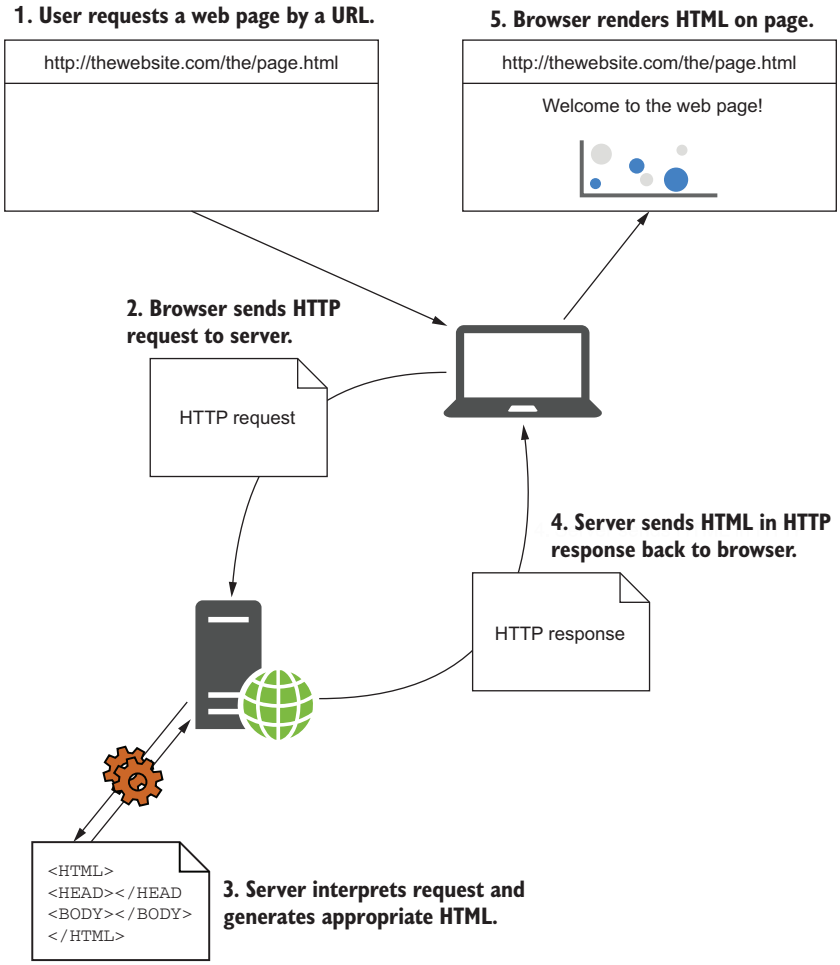

Figure 1.7 Requesting a web page. The user starts by requesting a web page, which causes an HTTP request to be sent to the server. The server interprets the request, generates the necessary HTML, and sends it back in an HTTP response. The browser can then display the web page.

Once the server receives the request, it will check that it makes sense, and if it does, will generate an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, or a simple acknowledgment. For this example, I'll assume the user has reached the homepage of a web app, and so the server responds with some HTML. The HTML is added to the HTTP response, which is then sent back across the internet to the browser that made the request.

As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server. To display the complete web page, instead of a static, colorless,

raw HTML file, the browser must repeat the request process, fetching every referenced file. HTML, images, CSS for styling, and JavaScript files for extra behavior are all fetched using the exact same HTTP request process.

Pretty much all interactions that take place on the internet are a facade over this same basic process. A basic web page may only require a few simple requests to fully render, whereas a modern, large web page may take hundreds. The Amazon.com homepage (www.amazon.com), for example, currently makes 298 requests, including 6 CSS files, 14 JavaScript files, and 245 image files!

Now you have a feel for the process, let's see how ASP.NET Core dynamically generates the response on the server.

1.3.2 How does ASP.NET Core process a request?

When you build a web application with ASP.NET Core, browsers will still be using the same HTTP protocol as before to communicate with your application. ASP.NET Core itself encompasses everything that takes place on the server to handle a request, including verifying the request is valid, handling login details, and generating HTML.

Just as with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure 1.8. A reverse-proxy server captures the request, before passing it to your application. In Windows, the reverse-proxy server will typically be IIS, and on Linux or macOS it might be NGINX or Apache.

> **DEFINITION** A *reverse proxy* is software responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy. This setup has several benefits, primarily security and performance for the web servers.

The request is forwarded from the reverse proxy to your ASP.NET Core application. Every ASP.NET Core application has a built-in web server, Kestrel by default, which is responsible for receiving raw requests and constructing an internal representation of the data, an `HttpContext` object, which can be used by the rest of the application.

From this representation, your application should have all the details it needs to create an appropriate response to the request. It can use the details stored in `HttpContext` to generate an appropriate response, which may be to generate some HTML, to return an "access denied" message, or to send an email, all depending on your application's requirements.

Once the application has finished processing the request, it will return the response to the web server. The ASP.NET Core web server will convert the representation into a raw HTTP response and send it back to the reverse proxy, which will forward it to the user's browser.

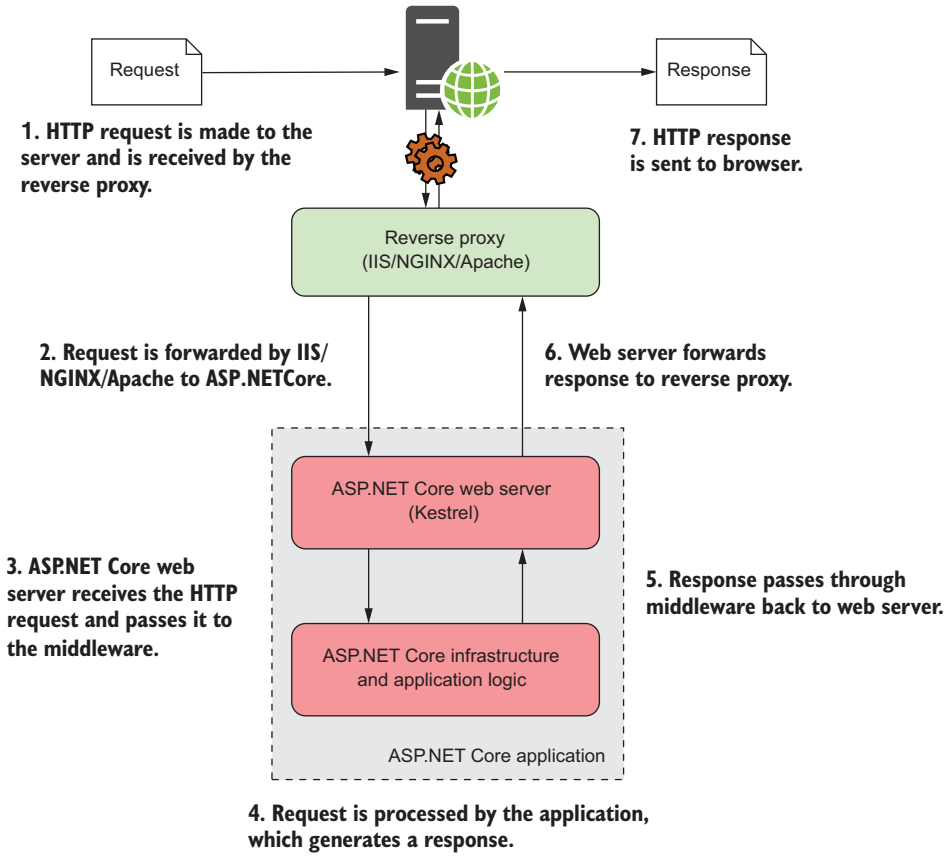

Figure 1.8 How an ASP.NET Core application processes a request. A request is received from a browser at the reverse proxy, which passes the request to the ASP.NET Core application, which runs a self-hosted web server. The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server relays this to the reverse proxy, which sends the response to the browser.

To the user, this process appears to be the same as for the generic HTTP request shown in figure 1.7—the user sent an HTTP request and received an HTTP response. All the differences are server-side, within our application.

You may be thinking that having a reverse proxy *and* a web server is somewhat redundant. Why not have one or the other? Well, one of the benefits is the decoupling of your application from the underlying operating system. The same ASP.NET Core web server, Kestrel, can be cross-platform and used behind a variety of proxies without putting any constraints on a particular implementation. Alternatively, if you wrote a new ASP.NET Core web server, you could use that in place of Kestrel without needing to change anything else about your application.

Another benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. They're often responsible for additional aspects,

such as restarting a process that has crashed. Kestrel can stay as a simple HTTP server without having to worry about these extra features when it's used behind a reverse proxy. Think of it as a simple separation of concerns: Kestrel is concerned with generating HTTP responses; a reverse proxy is concerned with handling the connection to the internet.

You've seen how requests and responses find their way to and from an ASP.NET Core application, but I haven't yet touched on how the response is generated. In part 1 of this book, we'll look at the components that make up a typical ASP.NET Core application and how they all fit together. A lot goes into generating a response in ASP.NET Core, typically all within a fraction of a second, but over the course of the book we'll step through an application slowly, covering each of the components in detail.

Before we dive in, you need to choose an underlying platform for your first ASP.NET Core application and set up a development environment in which to build it.

1.4 Choosing a platform for ASP.NET Core

ASP.NET Core was developed along with .NET Core and is often mentioned in the same breath, so it can be easy to forget that ASP.NET Core is platform-agnostic. You can build and run an ASP.NET Core application on both .NET Core or .NET Framework. The same features will be available in both cases, so why would you choose one over the over? Which route is right for you depends on both your history and the application you're looking to build, so in this section I've highlighted some advantages and disadvantages to consider.

1.4.1 Advantages of using .NET Framework

One of the most significant advantages of the full .NET framework is its maturity—it has been developed for 16 years, has been battle-hardened, and extensively deployed. For some, this maturity will be a significant deciding factor. It will already be installed on your servers and building an ASP.NET Core on top involves (relatively) little risk to your existing environment.

For others, particularly existing ASP.NET developers, the cross-platform and container-friendly .NET Core won't hold any appeal. These developers will, by necessity, be used to deploying to Windows servers, and it's perfectly reasonable to want to continue to do so, while still taking advantage of all ASP.NET Core has to offer.

The biggest reason to stick with the full .NET Framework when .NET Core was first released was because you needed to make use of Windows-specific features, such as the Registry or Directory Services. Microsoft have since released a compatibility pack[6] that makes these APIs available in .NET Core, but they're only available when running .NET Core on Windows, not on Linux or macOS. If you know your app relies on many Windows-only features, then .NET Framework may be the easiest option.

[6] The Windows Compatibility Pack is designed to help port code from .NET Framework to .NET Core. See http://mng.bz/50hu.

> **WARNING** If you choose to run on .NET Framework only, you won't be able to easily run your application cross-platform.

One advantage of using .NET Framework is that it has the greatest library support, in the form of NuGet packages. Library authors are being encouraged to make their libraries work identically on both .NET Framework and .NET Core by targeting .NET Standard, but that transition is a slow process.

.NET Standard[7] defines the APIs that are available on a given .NET platform. It's made up of multiple versions (for example, 1.1 and 1.2), each of which adds additional APIs compared to previous versions. Think of it as an "interface" for various .NET frameworks; the frameworks (such as .NET Core, .NET Framework, and Mono) all "implement" a version of .NET Standard.

> **TIP** You can create a new type of library that targets .NET Standard instead of targeting a specific framework. That allows you to use your library across multiple platforms, including .NET Core and .NET Framework. See appendix A for further details.

.NET Standard 2.0 vastly increases the number of APIs available to libraries that target it, covering almost the same area as .NET Framework 4.6.1. At the time of writing, 56% of packages on NuGet.org target the full framework rather than .NET Standard, so if your application currently relies on one of those packages, you'll have to choose .NET Framework for your ASP.NET Core application.

> **TIP** .NET Standard 2.0 contains a compatibility shim that allows you to reference .NET Framework 4.6.1 libraries from a .NET Standard library. For details, see http://mng.bz/jH8Y and appendix A.

1.4.2 Advantages of using .NET Core

If you're considering ASP.NET Core for a project, the chances are you're also interested in the associated features .NET Core brings, such as the cross-platform capabilities. If that's the case, then those features are obvious reasons to choose .NET Core as the underlying platform to use with ASP.NET Core.

The open source nature of .NET Core development can be a big deciding factor for some people. Open source development means you can clearly see how features and bugs are being addressed. If there's a particular feature you feel strongly about or a bug that's plaguing you, you can always submit a pull-request and see your code in the .NET Core platform!

Related to this, and the highly modular design of .NET Core, it's likely that the platform will see a faster release cycle than other platforms. Updates to .NET Framework require a massive amount of regression testing to ensure there are no subtle interactions that could break old applications. In contrast, installations of .NET Core are

[7] The .NET Standard GitHub repository can be found at https://github.com/dotnet/standard/blob/master/docs/faq.md.

independent of one another, so you can install multiple versions of .NET Core side-by-side. .NET Core also follows semantic versioning (SemVer), so you can be sure that your old applications won't be affected by installing a new version of the framework.

> **WARNING** Be aware that the faster release cycle generally means larger changes between .NET Core versions when you update your apps. For example, upgrading from .NET Core 1.0 to 2.0 is a significant and potentially breaking change.

Which platform you choose will depend on your use case. The full .NET Framework is still supported, and is being actively developed, but it's clear the focus of Microsoft is with .NET Core right now. If you're starting a new application from scratch and the libraries you require have been updated to use .NET Standard, then .NET Core seems to make the most logical choice for the future.

One final option is to multitarget your application, allowing it to run on both .NET Core and .NET Framework. This requires a little more effort to set up and maintain in terms of dependency wrangling, but it's a viable option if you're going to need to run in both environments. In this book, I'm going to be targeting the .NET Core platform, but all the examples should work equally with .NET Framework without any modification.

Once you've selected a platform for your ASP.NET Core applications, it's time to prepare your development environment—the last step before you build your first ASP.NET Core application!

1.5 Preparing your development environment

For .NET developers in a Windows-centric world, Visual Studio was pretty much a developer requirement in the past. But with .NET Core and ASP.NET Core going cross-platform, that's no longer the case.

All of ASP.NET Core (creating new projects, building, testing, and publishing) can be run from the command line for any supported operating system. All you need is the .NET Core SDK and tooling, which provides the .NET Command Line Interface (CLI). Alternatively, if you're on Windows, and not comfortable with the command line, you can still use File > New Project in Visual Studio to dive straight in. With ASP.NET Core, it's all about choice!

In a similar vein, you can now get a great editing experience outside of Visual Studio thanks to the OmniSharp project.[8] This is a set of libraries and editor plugins that provide code suggestions and autocomplete (IntelliSense) across a wide range of editors and operating systems. How you setup your environment will likely depend on which operating system you're using and what you're used to.

Remember that, if you're using .NET Core, the operating system you choose for development has no bearing on the final systems you can run on—whether you

[8] Information about the OmniSharp project can be found at www.omnisharp.net. Source code can be found at https://github.com/omnisharp.

choose Windows, macOS, or Linux for development, you can deploy to any supported system.

1.5.1 If you're a Windows user

For a long time, Windows has been the best system for building .NET applications, and with the availability of Visual Studio that's still the case.

Visual Studio (figure 1.9) is a full-featured integrated development environment (IDE), which provides one of the best all-around experiences for developing ASP.NET Core applications. Luckily, the Visual Studio Community edition is now free for open source, students, and small teams of developers! Visual studio comes loaded with a whole host of templates for building new projects, debugging, and publishing, without ever needing to touch a command prompt.

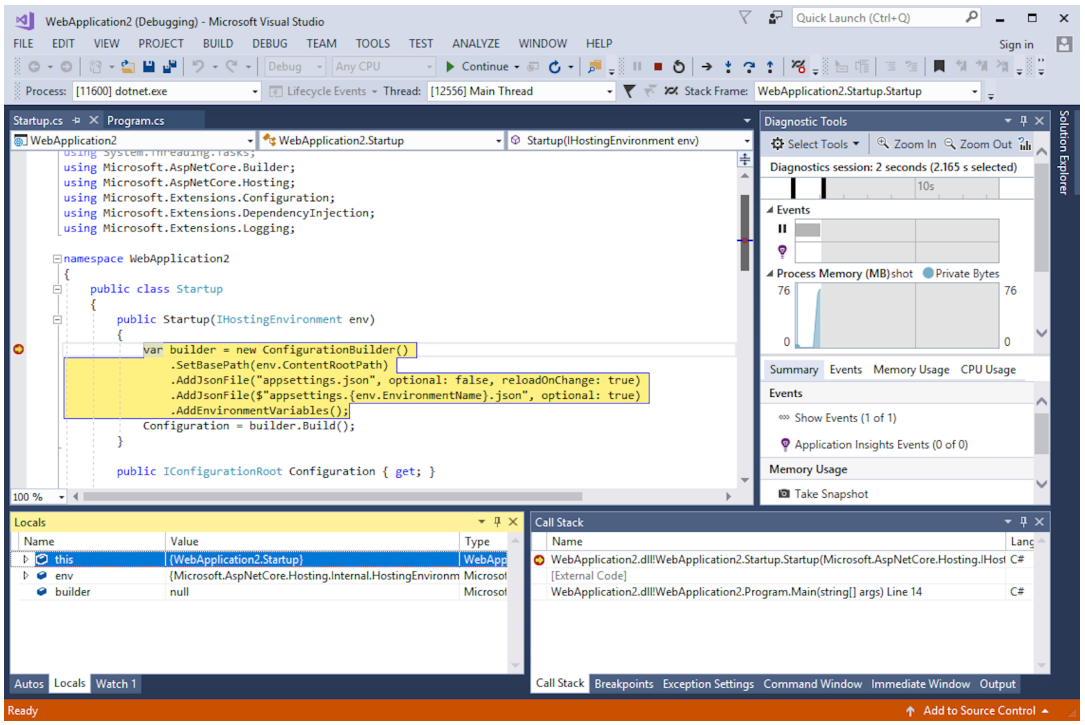

Figure 1.9 Visual Studio provides the most complete ASP.NET Core development environment for Windows users.

Sometimes, though, you don't want a full-fledged IDE. Maybe you want to quickly view or edit a file, or you don't like the sometimes unpredictable performance of Visual Studio. In those cases, a simple editor may be all you want or need, and Visual Studio Code is a great choice. Visual Studio Code (figure 1.10) is an open source, lightweight editor that provides editing, IntelliSense, and debugging for a wide range of languages, including C# and ASP.NET Core.

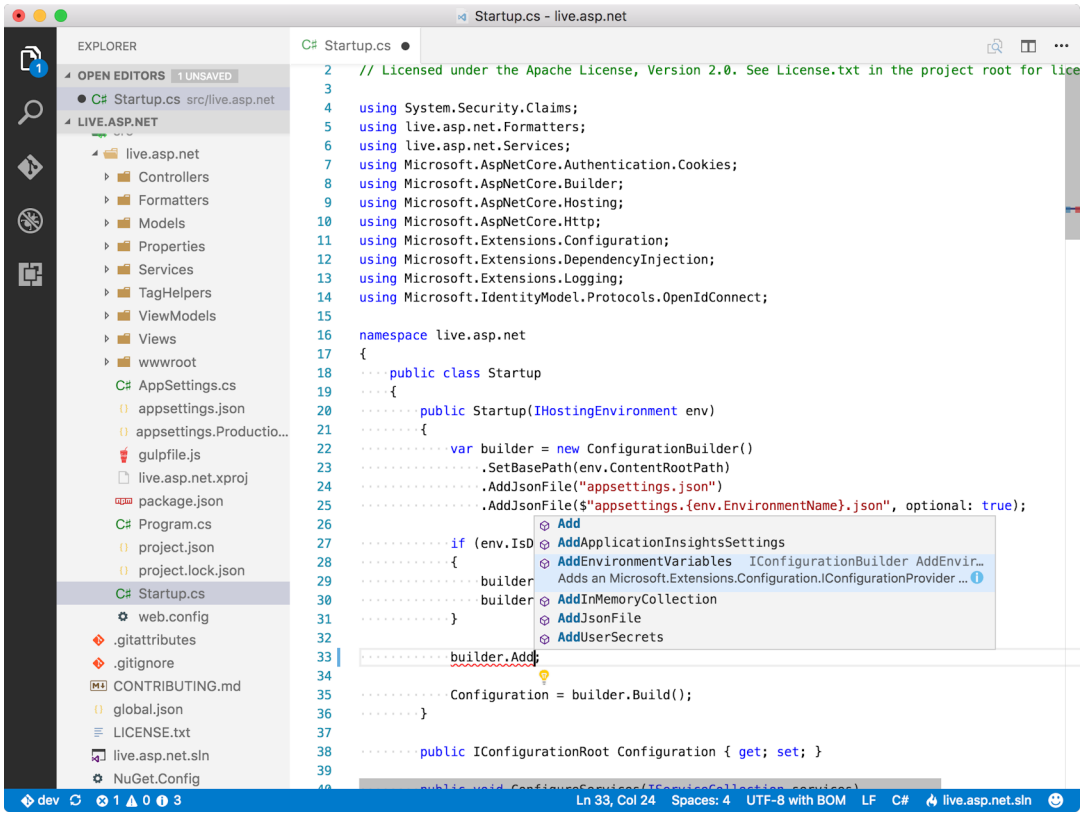

Figure 1.10 Visual Studio Code provides cross-platform IntelliSense and debugging.

Whether you install Visual Studio or another editor, such as Visual Studio Code, you'll need to install the .NET Core tooling to start building ASP.NET Core apps. You can either download it from the ASP.NET website (https://get.asp.net) or select the .NET Core cross-platform development workload during Visual Studio 2017 installation.

1.5.2 If you're a Linux or macOS user

As a Linux or macOS user, you have a whole host of choices. OmniSharp has plugins for most popular editors, such as Vim, Emacs, Sublime, Atom, and Brackets, not to mention the cross-platform Visual Studio Code. Install the appropriate plugin to your favorite and you'll be writing C# in no time.

Again, you'll need to install the .NET Core SDK from the ASP.NET website (https://get.asp.net) to begin .NET Core and ASP.NET Core development. This will give you the .NET Core runtime and the .NET CLI to start building ASP.NET Core applications.

The .NET CLI contains everything you need to get started, including several project templates. You don't get a huge number to choose from by default, but you can

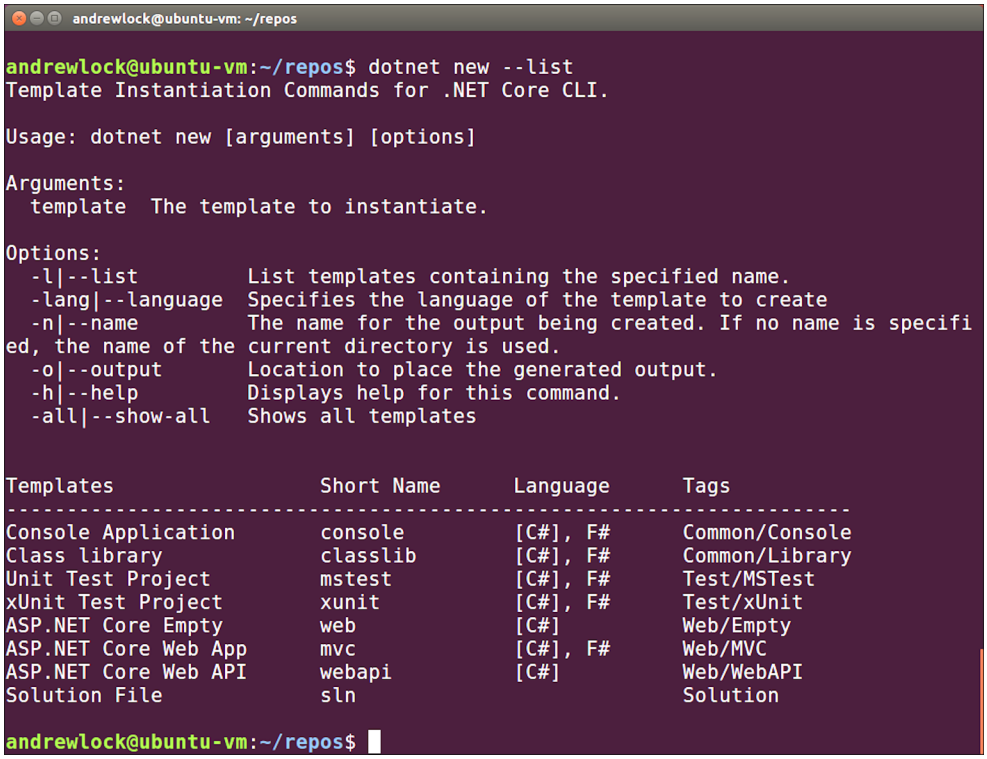

Figure 1.11 The .NET CLI includes several templates by default, as shown here. You can also install additional templates or create your own.

install new ones from GitHub or NuGet if you want more variety. You can easily create applications from the predefined templates to quick-start your development, as shown in figure 1.11.

In addition, in May 2017, Microsoft released Visual Studio for Mac. With VS for Mac you can build cross ASP.NET Core apps, using a similar editor experience to Visual Studio, but on an app designed natively for macOS. VS for Mac is still young, but if you're a macOS user, then it's a great choice and will no doubt see many updates.

In this book, I'll be using Visual Studio for most of the examples, but you'll be able to follow along using any of the tools I've mentioned. The rest of the book assumes you've successfully installed .NET Core and an editor on your computer.

You've reached the end of this chapter; whether you're new to .NET or an existing .NET developer, there's a lot to take in—frameworks, platforms, .NET Framework (which is a platform!). But take heart: you now have all the background you need and, hopefully, a development environment to start building applications using ASP.NET Core.

In the next chapter, you'll create your first application from a template and run it. We'll walk through each of the main components that make up your application and see how they all work together to render a web page.

Summary

- ASP.NET Core is a new web framework built with modern software architecture practices and modularization as its focus.
- It's best used for new, "green-field" projects with few external dependencies.
- Existing technologies such as WCF and SignalR can't currently be used with ASP.NET Core, but work is underway to integrate them.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows dynamically building responses to a given request.
- An ASP.NET Core application contains a web server, which serves as the entry-point for a request.
- ASP.NET Core apps are protected from the internet by a reverse-proxy server, which forwards requests to the application.
- ASP.NET Core can run on both .NET Framework and .NET Core. If you need Windows-specific features such as the Windows Registry, you should use .NET Framework, but you won't be able to run cross-platform. Otherwise, choose .NET Core for the greatest reach and hosting options.
- The OmniSharp project provides C# editing plugins for many popular editors, including the cross-platform Visual Studio Code editor.
- On Windows, Visual Studio provides the most complete all-in-one ASP.NET Core development experience, but development using the command line and an editor is as easy as on other platforms.

.NET DEVELOPMENT

ASP.NET Core IN ACTION

Andrew Lock

The dev world has permanently embraced open platforms with flexible tooling, and ASP.NET Core has changed with it. This free, open source web framework delivers choice without compromise. You can enjoy the benefits of a mature, well-supported stack and the freedom to develop and deploy from and onto any cloud or on-prem platform.

ASP.NET Core in Action opens up the world of cross-platform web development with .NET. You'll start with a crash course in .NET Core, immediately cutting the cord between ASP.NET and Windows. Then, you'll begin to build amazing web applications step by step, systematically adding essential features like logins, configuration, dependency injection, and custom components. Along the way, you'll mix in important process steps like testing, multiplatform deployment, and security.

What's Inside

- Covers ASP.NET Core 2.0
- Dynamic page generation with the Razor templating engine
- Developing ASP.NET Core apps for non-Windows servers
- Clear, annotated examples in C#

Readers need intermediate experience with C# or a similar language.

Andrew Lock has been developing professionally with ASP.NET for the last seven years. His focus is currently on the ASP.NET Core framework.

"Comprehensive coverage of the latest and greatest .NET technology."
—Jason Pike
Atlas RFID Solutions

"A thorough and easy-to-read training guide to the future of Microsoft cross-platform web development."
—Mark Harris, Microsoft

"An outstanding presentation of the concepts and best practices. Explains not only what to do, but why to do it."
—Mark Elston, Advantest America

"Superb starting point for .NET Core 2.0 with valid and relevant real-world examples."
—George Onofrei, Devex

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/asp-net-core-in-action

ISBN-13: 978-1-61729-461-7
ISBN-10: 1-61729-461-6
54999
9 781617 294617

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]