ASP.NET Core

Succinctly®

by Simone Chiaretta and
Ugo Lattanzi

Syncfusion | Technology Resource Portal

ASP.NET Core Succinctly

By

Simone Chiaretta and Ugo Lattanzi

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: Tres Watkins, content development manager, Syncfusion, Inc.
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Authors

Ugo Lattanzi

Ugo Lattanzi is a solution architect who specializes in enterprise applications with a focus on web applications, service-oriented applications, and all environments where scalability is a top priority.

Due to his experience in recent years, he now focuses on technologies like ASP.NET, Node.js, cloud computing (Azure and AWS), enterprise service bus, AngularJS, and JavaScript. Thanks to his passion for web development, Microsoft has recognized him as a Microsoft MVP for eight years in web technologies.

Ugo is a speaker for many important technology communities; he is a co-organizer of the widely appreciated Web European Conference in Milan; and he has authored several books and articles, including co-authoring *OWIN Succinctly*, released by Syncfusion.

When not working, he loves to spend time with his son Tommaso and his wife Eleonora, and he enjoys sports like snowboarding and cycling.

Simone Chiaretta

Simone Chiaretta is a web architect and developer who enjoys sharing his development experiences—including almost 20 years' worth of knowledge on web development—concerning ASP.NET and other web technologies.

He has been an ASP.NET Microsoft MVP for eight years, authoring several books about ASP.NET MVC, including *Beginning ASP.NET MVC 1.0* and *What's New in ASP.NET MVC 2*, both published by Wrox, as well as coauthoring *OWIN Succinctly*, published by Syncfusion.

Simone has spoken at many international developer conferences and contributed to online developer portals like SimpleTalk. He also co-founded the Italian ALT.NET user group ugialt.NET, and he is the co-organizer of many conferences in Milan, including the Web European Conference.

When not writing code, blog posts, or taking part in the worldwide .NET community, he is either training for or taking part in Ironman triathlons.

He is currently one of many expats living and working in the capital of Europe, Brussels.

About ASP.NET Core Succinctly

There are only a few years one can consider as landmarks of a profession, and 2016 is one. That is when Microsoft released .NET Core, its third major development library and SDK. But this time, it's not just for Windows, and it's not a closed-source product. It's an open-source SDK that allows developers to build and run .NET applications on Windows, Mac, and Linux.

This book specifically covers the web development part of the framework, ASP.NET Core, by first going through the foundations of the library, then covering its basic features, and finally covering the new version of the web application frameworks ASP.NET MVC and Web API. At its end, this book will show you how to deploy applications to the cloud, specifically on Azure.

The book is based on the most recent version of the framework at the time of writing: .NET Core 1.1. When showing how to develop with IDEs, it uses Visual Studio 2017 and the new text-based cross-platform IDE, Visual Studio Code.

Introduction to ASP.NET Core

ASP.NET Core is the web development framework that comes together with the new .NET Core and, besides all the new features, also adopts a significantly new approach to web development.

The first chapter starts by going through the history of Microsoft's web stack to show the motivations that led to this framework. Later, it moves to more practical matters, like showing you how to get started with .NET Core and describing the foundations of the framework.

Chapter 1 What are .NET Core and ASP.NET Core?

Before trying to understand the reason for its existence, let's first try to define what .NET Core and ASP.NET Core are.

.NET Core

The framework .NET Core 1.1 a is modular, cross-platform, cloud-optimized version of the .NET Framework, consisting of the CoreCLR and the implementation of the .NET Standard Library 1.6. One of the main features of this library is the ability to install only the features that are needed for the application you are building, reducing its footprint and the possibility of installing the library itself within the application. This makes it possible for applications built with different versions to co-exist on the same machine without the compatibility problems typical of the full .NET Framework.

ASP.NET Core

ASP.NET Core is a complete rewrite of ASP.NET, built with the goal of being cross-platform, completely open-source, and without the limitations of backward compatibility. Like .NET Core, ASP.NET Core is also built with a modular approach. This means the application you build can include only the needed features without taking on additional burdens. This is made possible by the new startup and execution environment, based on the Open Web Interface for .NET (OWIN) standard. In addition, ASP.NET Core comes with many interesting features that we are going to see throughout the book, like an integrated dependency injection system and a new application framework that unifies the programming models of ASP.NET MVC and Web API.

Chapter 2 A Brief History of the Microsoft Web Stack

ASP.NET Core is very different from anything that came before it. In order to understand why .NET Core came to be, it's important to see what it was before and how the web development world has changed during the last 15 years.

ASP.NET Web Forms

Web Forms was released with the first version of the .NET Framework back in 2002. It was meant to appeal to two different kinds of developers.

The first group included web developers coming from *Classic ASP*, also called Active Server Pages, who were already building websites and web applications with a mix of static HTML and dynamic server-side scripting (usually VBScript), which were used by the framework to hide the complexity of interacting with the underlying HTTP connection and the web server. The framework also provided other services to implement some kind of state management, caching, and other lower-level concerns.

The second group was comprised of a large number of *WinForms* application developers who were forced by changes in the market to start approaching the world of web development. Those developers didn't know how to write HTML. They were used to developing the UI of their applications by dragging elements onto the designer surface of an IDE.

Web Forms was engineered by combining elements from both technologies. It turned out to be a framework where everything was abstracted: the elements of the HTTP connection (request and response), the HTML markup of UI elements, and the stateless nature of the web—this last one was via the infamous *ViewState*.

This mix was very successful, as it allowed a lot of new developers to build web apps thanks to the feature-rich and approachable event-based programming model.

Over the years, many things changed. These new developers became more skilled and wanted to have more control over the HTML markup produced. And the web programming model changed as well, with the more extensive usage of JavaScript libraries, with Ajax, and with the introduction of mobile devices. But it was difficult for Web Forms to evolve:

- It was a huge, monolithic framework that was heavily coupled with the main abstraction layer *System.Web*.

- Given its programming model, which relied on Visual Studio, the release cycle of Web Forms was tied to its IDE and the full framework, so years passed between updates.

- The last limiting factor was the fact that Web Forms was coupled with Internet Information Services (IIS), which was even more difficult to upgrade because it was part of the operating system itself.

ASP.NET MVC

To try and solve the aforementioned issues, in 2009, Microsoft released a new web framework called ASP.NET MVC. It was based on the Model-View-Controller (MVC) pattern to keep a clear separation between business logic and presentation logic, allowing complete control over the HTML markup. In addition to this, it was released as a separate library, one not included in the framework. Thanks to this release model, and not relying 100% on the IDE for the design of the UI, it was possible to easily update it, keeping it more in line with the fast-paced world of web development.

But ASP.NET MVC, although solving the problem of the slow release cycle and removing the HTML markup abstraction, still suffered from a dependency on the .NET Framework and System.Web, which still made it strictly coupled with IIS and Windows.

ASP.NET Web API

Over time, a new web programming model appeared: instead of processing data server-side and sending the fully rendered page to the browser, this new paradigm, later called Single-Page Applications (SPA), used mostly static web pages that fetched data via Ajax from the server and rendered the UI directly on the client via JavaScript. Microsoft needed to implement a lighter, web-aware version of the library it already had for remote communication, Windows Communication Foundation (WCF), so it released ASP.NET Web API.

This library was even more modular than the other libraries, and, probably because it was originally developed by the WCF team and not the ASP.NET team, it didn't rely on System.Web and IIS. This made Web API completely independent from the rest of ASP.NET and IIS, opening possibilities such as running it inside a custom host or potentially under different web servers.

OWIN and Katana

Now, with all these modular frameworks spreading around, there was the concrete risk that developers had to manage separate hosts for different aspects of modern applications. To avoid this becoming a real problem, a group of developers from the .NET community created a new standard, called *OWIN*, which stands for Open Web Interface for .NET, that defines the components of a modern web application and how those components interact.

Microsoft released an OWIN-compliant web server, called *Katana*, and made sure its web frameworks worked inside it. Some problems still remained, however.

ASP.NET MVC, being tied to System.Web, couldn't run inside Katana. Also, all these frameworks, being developed by different teams and at different times, had different programming models. For example, both ASP.NET MVC and Web API supported dependency injection, but in different ways. If developers wanted to combine MVC and Web API inside the same application, they had to implement them twice.

What it brought to .NET Core

With the latest iteration of libraries, it was clear that the maximum potential of the full .NET framework had been reached. Still, many issues remained open:

- One couldn't run .NET applications on a non-Windows system.
- The dependency on the full framework made the .NET apps less suitable for high-density scenarios, like the cloud, where hundreds of applications run on a single machine and must scale up very fast.
- The complexity of the .NET project system prevented the development of .NET apps outside of Visual Studio.

It became clear that the only possible solution was to start from scratch, redesigning the ASP.NET framework to be fully modular with clearly implemented, generic foundations. It also needed to be cross-platform and based on an underlying .NET framework adhering to the same principles.

For these reasons, Microsoft completely rebuilt ASP.NET Core based on a new cross-platform .NET runtime, which later became .NET Core.

Chapter 3 Getting Started with .NET Core

Now that it is clear what ASP.NET Core and .NET Core are, and why they were created, it's time to look at how to install them and how to build a simple application with them.

Installing .NET Core on Windows

Installing on Windows is pretty easy. With Visual Studio 2017, chances are you already installed it. If not, go back to the Visual Studio Installer and make sure you have the .NET Core workload selected.

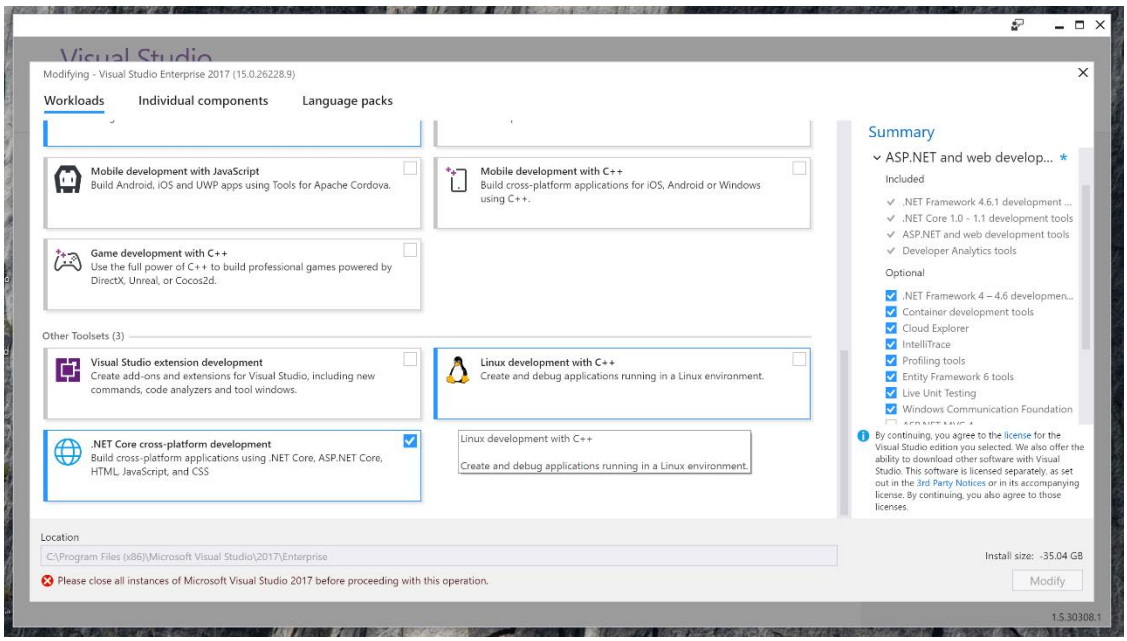

Figure 3-1: Visual Studio Installer

Installing .NET Core on a Mac (or Linux)

The beauty of .NET Core is that it can also be installed on a Mac (or Linux, for that matter) without relying on third-party frameworks, as was needed before with Mono.

Each distribution of Linux has its own individual way of installing, but in the end, the process boils down to the same principles:

- Install prerequisites and configure the package manager of your distribution.
- Invoke the package manager to download and install .NET Core and its tools.

You can read instructions specific to your distribution on the official .NET Core website. As an example, we'll show you how to install on a Mac.

Code Listing 3-1

```
>brew update
>brew install openssl
>ln -s /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib /usr/local/lib/
>ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/
```

Once all these prerequisites have been installed, you can download and install the official SDK for macOS by downloading it from the official .NET Core website.

On Linux and Mac, you do not have Visual Studio to develop apps, but you can use the .NET Core SDK or Visual Studio Code, which is a lightweight, extensible, cross-platform text editor built by Microsoft and the community. The last chapter of this book covers in detail each of the tools with which you can build .NET Core apps.

Building your first .NET Core application

With so many operating systems and tools available, there are many ways to create a .NET Core application, but all the visual tools rely on the SDK to get things done. Here you are going to build your first .NET Core application using the *dotnet* command line interface (CLI).

The main entry point for the SDK is the **`dotnet`** command. This command, depending on the verb used, can do lots of things, from acting as host and runner for the application to creating a new project, managing dependencies, and building applications.

Command-line tools

As an example, let's create a simple "Hello World" command-line application using the `dotnet` CLI. Since the command-line tools are cross-platform, the following steps can be performed on any of the supported systems: Windows, Mac, or Linux.

Open the Command Prompt (or terminal) and create an empty folder, calling it **`HelloWorldApp`**. Next, move inside this newly created folder and launch the **`dotnet new console`** command.

Now, launch the **`dotnet restore`** command to download and install all the packages, and finally, launch the **`dotnet run`** command to build and run the project.

Figure 3-2 shows the output of this sequence of commands.

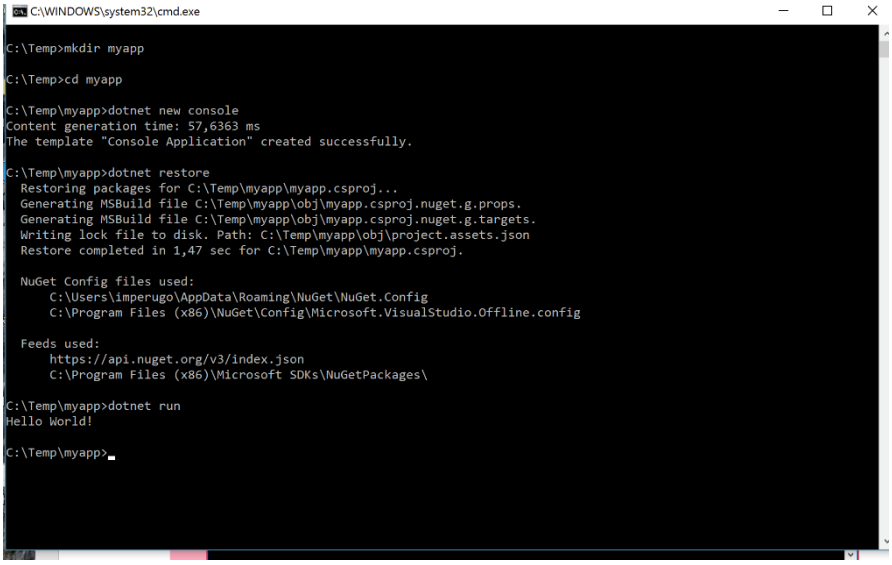

Figure 3-2: dotnet CLI

Let's dive into what happened.

The **`dotnet new console`** command added two files to the folder:

- **`HelloWorldApp.csproj`**, an XML file containing the configuration of the project.
- **`Program.cs`**, the actual code file.

The `Program.cs` file is pretty simple, as it just prints the `Hello World!` string to the console.

Code Listing 3-2

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

What is more interesting to look at is the **`HelloWorldApp.csproj`** file. If you've ever looked at the project files used by Visual Studio with the full .NET Framework, you might wonder why we even discuss it. You rarely looked inside them, as they were black boxes written in XML. However, with .NET Core, they become easier to modify manually or via other commands of the **`dotnet`** tool.

Code Listing 3-3

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" />

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="**\*.cs" />
    <EmbeddedResource Include="**\*.resx" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NETCore.App">
      <Version>1.0.1</Version>
    </PackageReference>
    <PackageReference Include="Microsoft.NET.Sdk">
      <Version>1.0.0-alpha-20161104-2</Version>
      <PrivateAssets>All</PrivateAssets>
    </PackageReference>
  </ItemGroup>

  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
</Project>
```

Apart from the MSBuild imports, the file is basically split into three areas:

- The first area, **<PropertyGroup>**, contains the properties used to configure the build—in this case, the framework it targets and the type of output.
- The second group, **<ItemGroup>**, contains all the files that will be compiled or included in the build. Notice that it uses a globbing syntax (a file wildcard expansion similar to the one used in gitignore files) instead of listing every file individually, like it does with the full .NET Framework.
- The last group, **<ItemGroup>**, still lists all the packages or projects that the current project depends on.

Visual Studio

Another way you can build the same console application is by using Visual Studio. First, go to the new project dialog and select **Console Application (.NET Core)**, as shown in Figure 3-3.

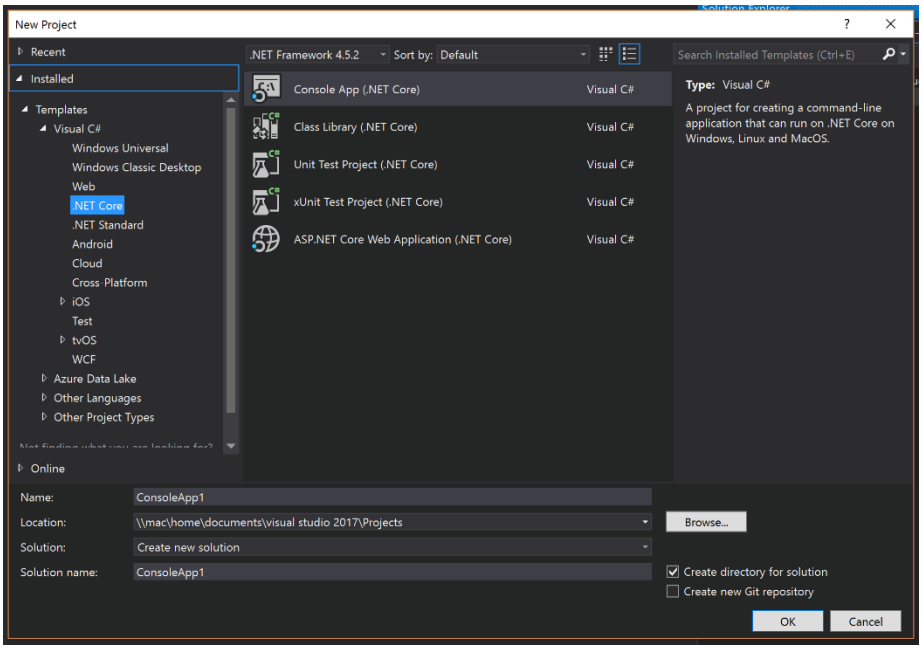

Figure 3-3: New Console Application Using Visual Studio 2017

Once the project has been created, add text to the console application and launch the project. Figure 3-4 shows Visual Studio with the console application loaded.

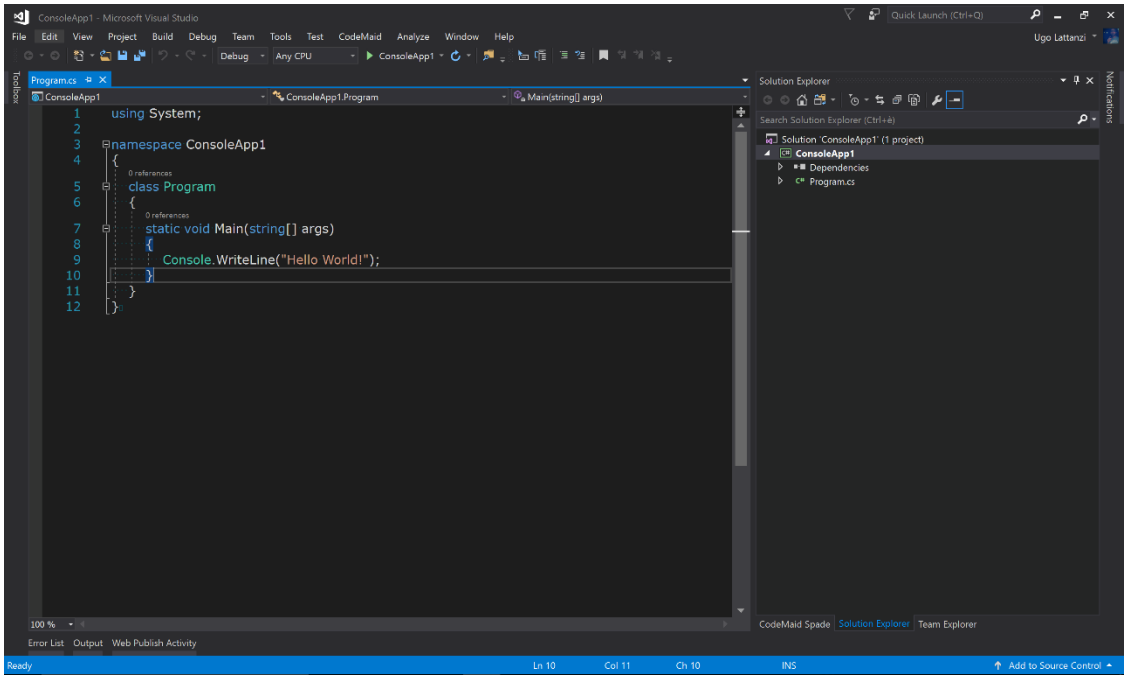

Figure 3-4: Hello World Application

Conclusion

In this chapter, we've shown why 2016 is one of the few landmarks in our industry, together with 1996, when Microsoft released ASP Classic, and 2002, when the .NET Framework and ASP.NET were released.

.NET Core is the next evolution of the Microsoft web development stack. While not as different from the .NET Framework as the .NET Framework was from ASP Classic, it still changes the foundations on which ASP.NET applications are built.

You've also seen how to install .NET Core on Windows, Mac, and Linux, and how to build a simple application using the base layer of .NET tools: the dotnet CLI. With this knowledge, you are ready to explore the new ASP.NET Core application framework.

Chapter 4 ASP.NET Core Basics

In this chapter, you are going to learn about the main innovations of the latest version of ASP.NET that have dramatically changed from the previous version. In particular, you are going to see why ASP.NET Core is defined as a lean framework and how to manage static files, different hosting environments, exceptions, dependency injection, and all the other significant features of the latest release.

Web app startup

You can better understand how to use .NET Core by first creating a new empty application with Visual Studio 2017. If you prefer the superlight VS Code instead of Visual Studio, you must use the command line to create an empty template.

When creating a new ASP.NET Core template from Visual Studio, you have different options. To better understand the flow and all the components you need for a web application, the empty template is the best choice.

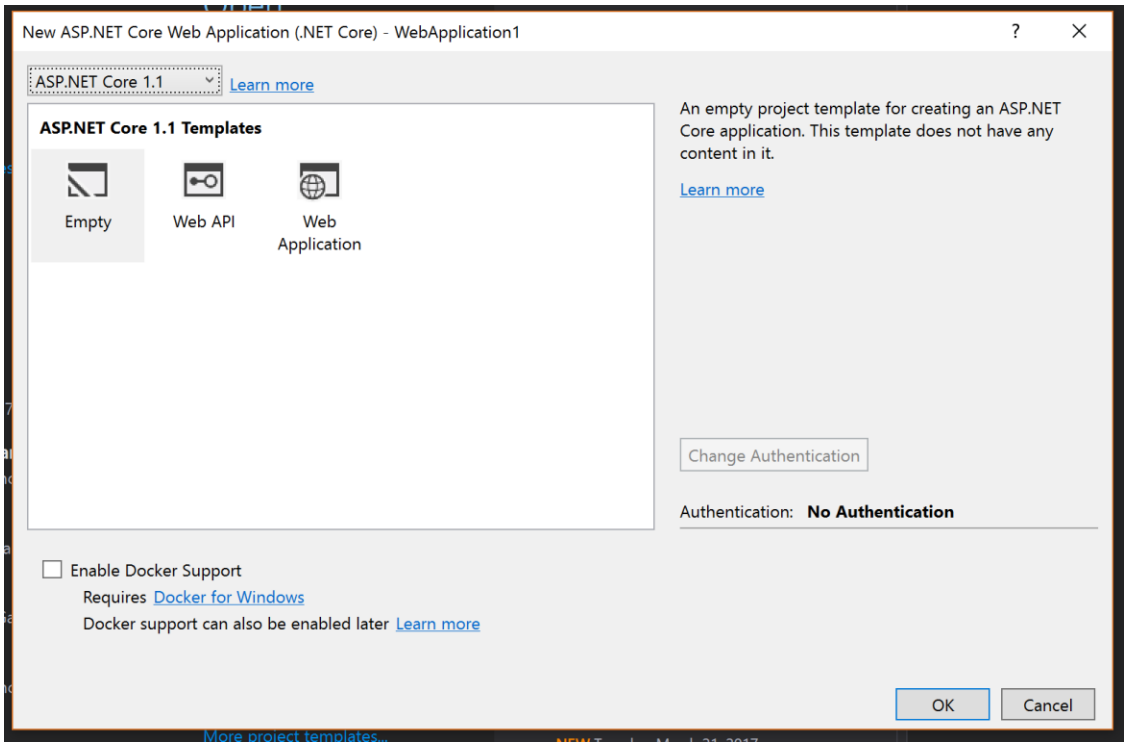

Figure 4-1: New Web Application Using Visual Studio 2017

Looking at the Solution Explorer, notice that the folder structure and files are very different from the previous version of ASP.NET. First of all, there is a `wwwroot` folder that contains all the static files.

In a different section, we will explain how to use the wwwroot folder and the reason why you need it.

All files in the root of the project are either new additions or they changed their role:

- **Program.cs** is the entry point for the web application; everything starts from here. As we mentioned in the chapters before, the .NET Core host can only run console applications. So, the web app is a console application too.

- **project.csproj** is an XML-based project configuration file. It contains all the package references and the build configuration.

- **Startup.cs** is not exactly new. If you already used OWIN, you probably know the role of this class, but we can definitely say if the **Program.cs** is the entry point of the .NET Core app, **Startup.cs** is the entry point of ASP.NET Core application (previously we were using the **global.asax** file).

- **web.config** is still here, but it's almost empty and only used to tell Internet Information Service (IIS) to process all requests using the ASP.NET Core handler.

Program.cs

As mentioned before, this class is the entry point of the .NET Core application, and its role is to create the host for the web application. Since we can host our web application on different web servers with .NET Core, this is the right place to configure everything.

Code Listing 4-1

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

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();
```

```
            host.Run();
        }
    }
}
```

As you can see, this class is simple, as the only two important methods are **`UseKestrel`** and **`UseIISIntegration`**, respectively used to host the application on Kestrel or IIS.

Startup.cs

The ASP.NET Core pipeline starts here, and, as you can see from the following code, almost nothing comes with the template.

Code Listing 4-2

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

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
    public class Startup
    {
        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironmen
t env, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();

            if (env.IsDevelopment())
```

```
            {
                app.UseDeveloperExceptionPage();
            }

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

What's important here are the methods **`ConfigureServices`**, where dependency injection is configured (we'll talk about this later in the chapter), and **`Configure`**, where all needed middleware components are registered and configured.

Tip: We already wrote a book about OWIN, so if you don't know what a middleware component is or how to use it, we suggest you read the free e-book OWIN Succinctly, available from Syncfusion.

The **`app.Run`** method is a perfect example of an inline middleware component. In this case, it is useless because the web application will always return the text string "Hello World!", but more complex logic (i.e. authentication, monitoring, exception handling, and so on) can be added.

Dependency injection

One of the biggest new features of ASP.NET Core is the inclusion of a way to handle dependencies directly inside the base library. This has three major benefits:

- First, it means developers no longer have an excuse not to use it; whereas before it was basically left to their conscience.
- Second, you don't need to use third-party libraries.
- Finally, all the application frameworks and middleware components rely on this central configuration, so there is no need to configure dependency injection in different places and different ways, as was needed before.

What is dependency injection?

Before looking at how to use dependency injection inside ASP.NET Core applications, let's see what it is and why it is important to use.

In order to be easy to maintain, systems are usually made of many classes, each of them with very specific responsibilities. For example, if you want to build a system that sends emails, you might have the main entry point of the system and one class that is responsible for formatting text and then one that is responsible for actually sending the email.

The problem with this approach is that, if references to these additional classes are kept directly inside the entry point, it becomes impossible to change the implementation of the helper class without touching the main class.

This is where dependency injection, usually referred to as DI, comes in to play. Instead of directly instantiating the lower-level classes, the high-level modules receive the instances from the outside, typically as parameters of their constructors.

Described more formally by Robert C. Martin, systems built this way adhere to one of the five SOLID principles, the dependency inversion principle:

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

While manually creating and injecting objects can work in small systems, those objects can become unmanageable when systems grow in size and hundreds or thousands of classes are needed. To solve this problem, another class is required: a factory that takes over the creation of all objects in the system, injecting the right dependencies. This class is called *container*.

The container, called an *Inversion of Control* (IoC) container, keeps a list of all the interfaces needed and the concrete class that implements them. When asked for an instance of any class, it looks at the dependencies it needs and passes them based on the list it keeps. This way very complex graphs of objects can be easily created with a few lines of code.

In addition to managing dependencies, these IoC containers also manage the lifetime of the objects they create. They know whether they can reuse an instance or need to create a new one.

This is a very short introduction of a very important and complicated topic related to the quality of software. Countless articles and books have been written about dependency injection and SOLID principles in general. A good starting point are the articles by Robert C. "Uncle Bob" Martin or by Martin Fowler.

Configuring dependency injection in ASP.NET Core

Now that you understand the importance of using DI in applications, you might wonder how to configure it. It is actually easy. It all happens in the **`ConfigureServices`** method.

Code Listing 4-3

```
public void ConfigureServices(IServiceCollection services)
{
  //Here goes the configuration
}
```

The parameter the method accepts is of the type **`IServiceCollection`**. This is the list used by the container to keep track of all the dependencies needed by the application, so it is to this collection that you add your classes.

There are two types of dependencies that can be added to the `services` list.

First, there are the ones needed by the frameworks to work, and they are usually configured using extension methods like **`AddServiceName`**. For example, if you want to use ASP.NET Core MVC, you need to write **`services.AddMvc()`** so that all the controllers and filters are automatically added to the list. Also, if you want to use Entity Framework, you need to add **`DBContext`** with **`services.AddDbContext<ExampleDbContext>(...)`**.

Then there are the dependencies specific to your application; they must be added individually by specifying the concrete class and the interface it implements. Since you are adding them yourself, you can also specify the lifetime of the service. Three kinds of lifecycles are available in the ASP.NET Core IoC container, and each of them has to be added using a different method.

The first is **Transient**. This lifecycle is used for light-weight services that do not hold any state and are fast to instantiate. They are added using the method **`services.AddTransient<IClock,Clock>()`** and a new instance of the class is created every time it is needed.

The second lifecycle is **Scoped**. This is typically used for services that contain a state that is valid only for the current request, like repositories and data access classes. Services registered as scoped will be created at the beginning of the request and the same instance will be reused every time the class is needed within the same request. They are added using the method **`services.AddScoped<IRepository, Repository>()`**.

The last lifecycle is called **Singleton**, and as the name implies, services registered this way will act like singletons. They are created the first time they are needed and reused throughout the rest of the application. Such services typically hold an application state like an in-memory cache or similar concerns. They are added via the method **`services.AddSingleton<IApplicationCache, ApplicationCache>()`**.

Using dependency injection

Let's now see an example of how DI-IoC is used in an ASP.NET MVC application. ASP.NET MVC is covered in detail in a later chapter, so don't worry if something looks unfamiliar; just focus on how dependencies are configured and reused.

To use dependency injection, you need four classes:

- The class that needs to use an external service, also called the consumer class. In our example, it's an ASP.NET MVC controller.
- The interface that defines what the external service does, which in our example is just giving the time of the day.
- The class that actually implements the interface.
- The Startup.cs file where the configuration will be saved.

The interface

First, you have to define the interface of the service, the only thing the consumer depends on.

Code Listing 4-4

```
public interface IClock
{
    DateTime GetTime();
}
```

Concrete implementation

Once the interface is defined, you need to implement the concrete class that does the actual work.

Code Listing 4-5

```
public class Clock: IClock
{
    public DateTime GetTime()
    {
        return DateTime.Now;
    }
}
```

Consumer controllers

For the sake of this example, you are going to slightly modify the **`HomeController`** that comes with the default project template. The most important change is the addition of a constructor and a private variable to hold the reference of the external dependency.

Code Listing 4-6

```
private readonly IClock _clock;

public HomeController(IClock clock)
{
    _clock = clock;
}
```

Obviously, you also have to use the dependency somehow. For this example, just write the current time in the About page by modifying the **`About`** action method.

Code Listing 4-7

```
public IActionResult About()
{
    ViewData["Message"] = $"It is {_clock.GetTime().ToString("T")}";

    return View();
}
```

The following listing is the complete file.

Code Listing 4-8

```
public class HomeController : Controller
{
    private IClock _clock;

    public HomeController(IClock clock)
    {
        _clock = clock;
    }

    public IActionResult Index()
    {
        return View();
    }

    public IActionResult About()
    {
        ViewData["Message"] = $"It is {_clock.GetTime().ToString("T")}";

        return View();
    }
```

```
    ...
}
```

Tying it all together via the configuration

Now that all elements are in place, you just need to configure the framework so that it injects the right class (**`Clock`**) in all objects that declare a dependency of type **`IClock`** as a parameter of the constructor. You've already seen how this has to be done via the **`ConfigureServices`** method.

Code Listing 4-9

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    // Add application services.
    services.AddTransient<IClock, Clock>();
}
```

The first line configures the application to use ASP.NET Core MVC, and the second one adds our simple clock service. We've shown how to use dependency injection in an ASP.NET application using a very simple external service that just gives the time, but the main elements needed are all there:

- A consumer class that declares its dependencies via parameters in the constructor (by referencing their interfaces).
- The interface of the service.
- The concrete implementation.
- The configuration that binds interface and implementation together and informs the container of their existence.

Environments

Every application we deploy needs to handle at least two or more environments.
For example, in a small application, we have the development environment (also known as dev), the production environment, and in some cases the staging environment.

More complex projects need to manage several environments, like quality assurance (QA), user acceptance test (UAT), preproduction, and so on. In this book, we show only what comes out of the box with ASP.NET Core; however, you can easily understand how to add a new environment.

One of my favorite features of ASP.NET Core, which is included with the framework, is what is called **Hosting Environment Management**. It allows you to work with multiple environments with no friction. But before diving deeper into this feature, you have to understand what the developer needs are.

Old approach

A good developer should never work on a production database, production storage, a production machine, and so on. Usually, in a .NET application, a developer manages this problem using the **`applicationSettings`** section in the `web.config` file combined with **Config Transformation Syntaxt** (more info at MSDN) and Preprocessor Directives.

This approach is tricky and requires you to build the application differently for each environment because the config transformation and the preprocessor directive are applied at compile time. Last but not least, this approach makes your code hard to read and maintain.

As you may have noticed in the previous chapters, the `web.config` file is used only to configure the `AspNetCoreModule` in case our application must be hosted on Internet Information Services (IIS); otherwise, it is useless.

For this reason, don't use the Config Transformation approach—use something cooler.

New approach

ASP.NET Core offers an interface named `IHostingEnvironment` that has been available since the first run of our application. This means we can easily use it in our `Startup.cs` file if we need it.

To know that, the implementation of IHostingEnvironment reads a specific environment variable called ASPNETCORE_ENVIRONMENT and checks its value. If it is **`Development`**, it means you are running the application in Dev mode. If it is **`Staging`**, you are running the application in a staging mode. And so it goes for all the environments you need to manage.

Because this approach is based on an environment variable, the switch between the configuration files happens at runtime and not at compile time like the old ASP.NET.

Visual Studio

Visual Studio has a *run* button, which is pretty awesome for developers because it runs the application attaching the debugger. But what environment will be used by Visual Studio when you push the run button?

By default, Visual Studio uses development mode, but if you want to change it or configure a new environment, you can do so easily by looking at the file launchSettings.json, available in the Properties folder of your application.

If you open it, you should have something like this:

Code Listing 4-10

```
{
    "iisSettings": {
        "windowsAuthentication": false,
        "anonymousAuthentication": true,
        "iisExpress": {
            "applicationUrl": "http://localhost:34081/",
            "sslPort": 0
        }
    },
    "profiles": {
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "Syncfusion.Asp.Net.Core.Succinctly.Environments": {
            "commandName": "Project",
            "launchBrowser": true,
            "launchUrl": "http://localhost:5000",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    }
}
```

The **iisSettings** section contains all the settings related to IIS Express, while the profile section contains the Kestrel configurations. If you are familiar with the JSON format, you can edit all these values in Visual Studio by following these depicted steps.

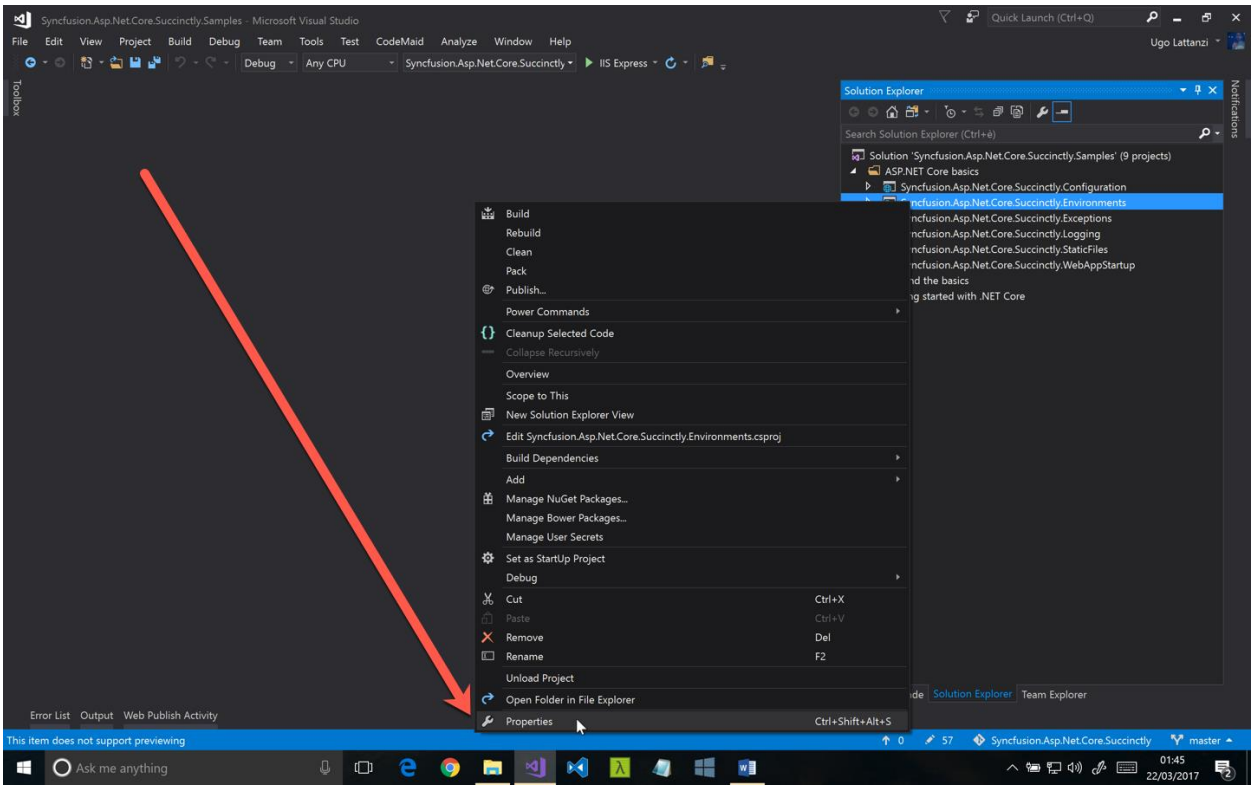

Figure 4-2: Change IIS Express Settings 1

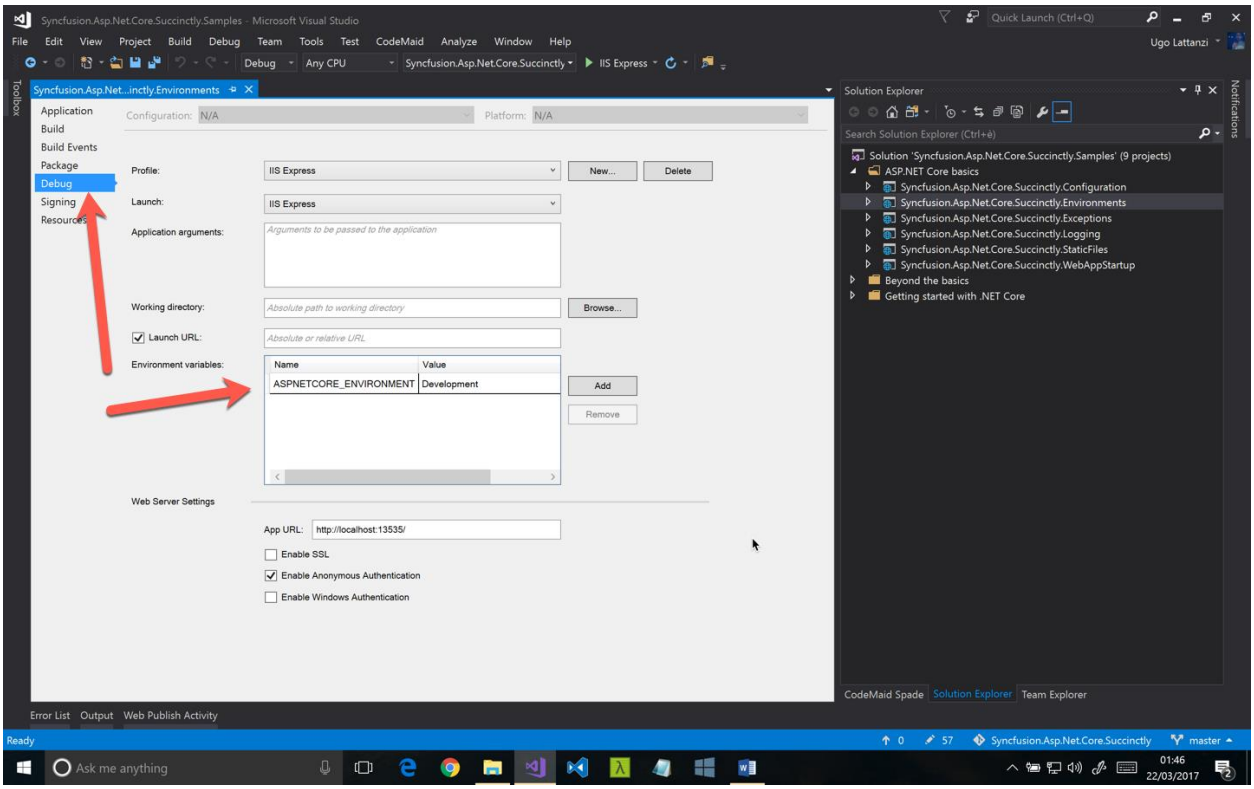

Figure 4-3: Change IIS Express Settings 2

From here, you can also change the settings in the case of Kestrel by using the drop-down menu at the top. If you prefer to work directly with JSON and want to change the environment, change the value for **`ASPNETCORE_ENVIRONMENT`**, and then save the file or add a new item in profiles section with our settings.

IHostingEnvironment

Sometimes a web application needs something more than switching the connection string from a developer database to a production database. For example, you may need see the stack trace of an error in case you are running the app on a local machine, or you may need to show an error page to the end user in the production version.

There are several ways to do that. The most common is undoubtedly to inject the **`IHostingEnvironment`** interface into your constructor and use it to change the behavior of your app. The following simple code is a possible startup class.

Code Listing 4-11

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

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.Environments
{
    public class Startup
    {
        private IHostingEnvironment _env;

        public Startup(IHostingEnvironment env)
        {
            _env = env
        }

        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            if (_env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            // ...
        }
    }
}
```

In this example, the **DeveloperExceptionPage Middleware** is used only if your application is running in a dev mode, which is exactly what we want.

What you did in this class can be repeated in any part of your code as a controller, a service, or however it needs to differ among the environments.

Startup class

The **Startup** class is absolutely the most important class in your application because it defines the pipeline of your web application, and it registers all the needed middleware components. Because of this, it might be very complex with lots of lines of code. If you add checking for the environment, everything could be more complicated and difficult to read and maintain.

For this reason, ASP.NET Core allows you to use different startup classes: one for each environment you want to manage or one for different "configure" methods.

Let's look at the Program.cs file:

Code Listing 4-12

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseIISIntegration()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

The method **.UseStartup<Startup>()** is very clever. It can switch between different classes automatically if you are following the right convention (method name + environment name).

For example, if you duplicate the **Startup** class and rename it **StartupDevelopment**, the extension method will automatically use the new one in the development environment.

You can use the same convention for the **Startup** class methods. So, duplicate the method **Configure** of the Startup.cs file, call it **ConfigureDevelopment**, and it will be called instead of the original one only in the development environment.

Create your own environment

We already mentioned environments like user acceptance test (UAT) or quality assurance (QA), but the **IHostingEnvironments** interface doesn't offer the method **IsUAT()** or **IsQualityAssurance**, so how can you create one?

If you think on it, the answer is pretty easy. It is enough to assign a new value to the **ASPNETCORE_ENVIRONMENT** variable using a set command in a command shell (i.e. **QualityAssurance**) and create an extension method like this:

Code Listing 4-13

```
using Microsoft.AspNetCore.Hosting;

namespace Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions
{
    public static class HostingEnvironmentExtensions
    {
        public static bool IsQualityAssurance(this IHostingEnvironment
hostingEnvironment)
        {
            return hostingEnvironment.EnvironmentName ==
"QualityAssurance";
        }
    }
}
```

Now, remaining on the previous example, we can use the extension method like this:

Code Listing 4-14

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions;

namespace Syncfusion.Asp.Net.Core.Succinctly.Environments
{
    public class Startup
    {
        private IHostingEnvironment _env;

        public Startup(IHostingEnvironment env)
        {
            _env = env
        }

        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
```

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

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            if (_env.IsDevelopment() || _env. IsQualityAssurance())
            {
                app.UseDeveloperExceptionPage();
            }

            // ...
        }
    }
}
```

In subsequent chapters, you will see how to use **`IHostingEnvironments`** in views or configuration files.

Static files

One of the main features of ASP.NET Core is that it can be as lean as you like. This means you are responsible for what you're going to put into the application, but it also means you can make the application very simple and fast.

In fact, if you start your project from an empty ASP.NET Core web application template, the application will not be able to serve static files. If you want to do it, you have to add and configure a specific package.

A common question is "Why doesn't it support static files by default if all websites need static files?"

The truth is that not all websites need to serve static files, especially on high-traffic applications. In this case, the static files should be hosted by a content delivery network (CDN).

Moreover, your web application could be an API application that usually serves data using JSON or XML format instead of images, stylesheets, and JavaScript.

Configure static files

As you'll see by reading this book, most of the configurations are managed by middleware components available on NuGet. That's also true in this case; you have to install a specific package and configure its middleware.

The package to install is **Microsoft.AspNetCore.StaticFiles**. You can get it using the Package Manager.

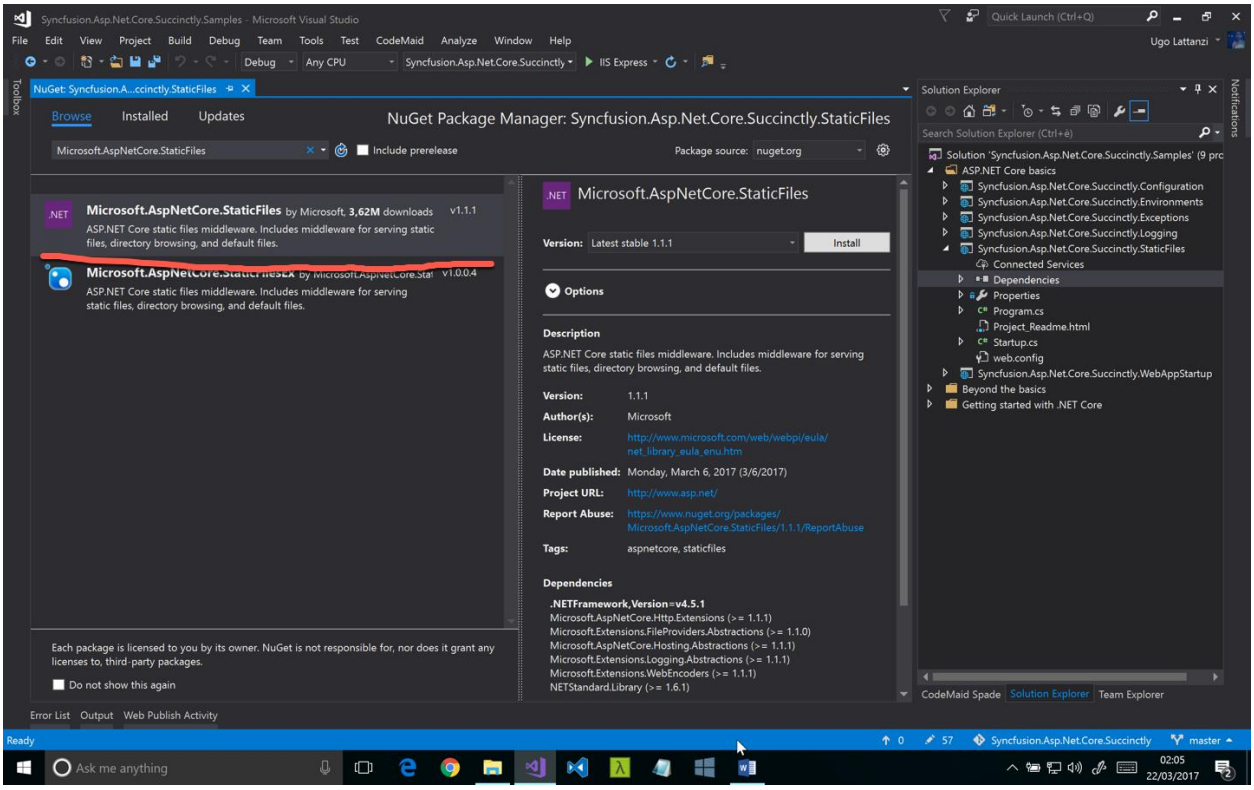

Figure 4-4: Install Static Files Package Using NuGet

Code Listing 4-15

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Syncfusion.Asp.Net.Core.Succinctly.StaticFiles
{
    public class Startup
    {
        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
```

```
        public void Configure(IApplicationBuilder app)
        {
            app.UseStaticFiles();

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

You are almost ready. The last, but still important, thing to know is that the **wwwroot** folder present in the root of your project will contain all the static files, so if you want to serve a file called image1.jpg for the request `http://localhost:5000/image1.jpg`, you have to put it into the root of the `wwwroot` folder, not in the root of the project folder like you were doing with the previous version of ASP.NET.

Single page application

Another scenario where the static-file middleware component combined with ASP.NET Core application could be very useful is for a web app that is a single page application (SPA).

The best way to describe the meaning of SPA is via Wikipedia's definition:

> A single-page application (SPA) is a web application or website that fits on a single webpage with the goal of providing a user experience similar to that of a desktop application.

Basically, most of the business logic is present on the client. The server doesn't need to render different views; it just exposes the data to the client. This is available thanks to JavaScript (combined with modern frameworks like Angular, React, Aurelia) and a set of APIs (in our case, developed with ASP.NET MVC Core).

If there is no server-side rendering, the web server must return a static file when the root domain is called by the browser (http://www.mysite.com). To do that, you have to configure the default documents into the **`Configure`** method of your **`Startup`** class.

If you're okay with the default documents being preconfigured (**default.htm**, **default.html**, **index.htm**, and **index.html**) it is enough to add **`UseFileServer`** like this:

Code Listing 4-16

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Syncfusion.Asp.Net.Core.Succinctly.StaticFiles
{
```

```
    public class Startup
    {
        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            // app.UseStaticFiles();
            app.UseFileServer();

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

Otherwise, if you need to use a specific file with a different name, you can override the default configuration and specify your favorite files as default documents:

Code Listing 4-17

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Syncfusion.Asp.Net.Core.Succinctly.StaticFiles
{
    public class Startup
    {
        // This method is called by the runtime. Use this method to add
services to the container.
        // For more information on how to configure your application,
visit http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to
configure the HTTP request pipeline.
```

```
        public void Configure(IApplicationBuilder app)
        {
            // app.UseStaticFiles();

            var options = new DefaultFilesOptions();
            options.DefaultFileNames.Clear();
            options.DefaultFileNames.Add("mydefault.html");
            app.UseDefaultFiles(options);
            app.UseFileServer();


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

Error handling and exception pages

You can write the best code in the world, but you must accept the fact that errors exist and are part of life. From a certain point of view, you could say that a good application is one that can identify an error in the shortest possible time and return the best possible feedback to the user.

To achieve this goal, you need to deal with different actors, like logging frameworks, exception handling, and custom error pages.

We have dedicated a section later in this chapter to logging frameworks. That’s why we are not showing how to configure the logging output here. For now, it is enough to know that out-of-the-box ASP.NET Core logs all the exceptions, so you don't need to create an exception middleware component of a specific code to log unhandled exceptions.

Of course, if you don't like what comes with the framework, you still have the opportunity to write your own exception handler. The first thing to do is throw an exception into your empty application.

Code Listing 4-18

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Extensions;
using Microsoft.Extensions.DependencyInjection;

namespace Syncfusion.Asp.Net.Core.Succinctly.Exceptions
{
    public class Startup
```

```
    {
        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app)
        {
            app.Run(async context =>
            {
                if (context.Request.Query.ContainsKey("throw"))
                {
                    throw new Exception("Exception triggered!");
                }

                await context.Response.WriteAsync("Hello World!");
            });
        }
    }
}
```

If you run the application and add the variable called **`throw`** to the query string like this—`http://localhost:5000/?throw` (in this case, the application is running using Kestrel with the default configuration)—you should receive an output as follows (although the output page could be different for each browser):

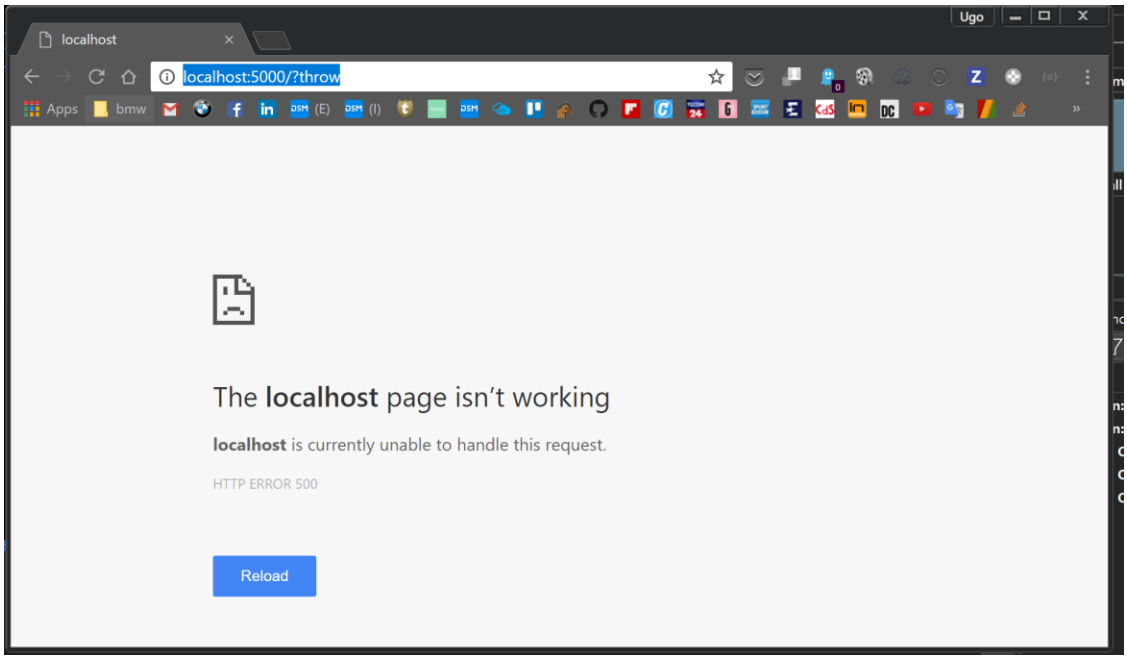

Figure 4-5: Default 500 Error

Developer exception page

As you can see, there isn't useful information, and the feedback isn't user friendly. This is because ASP.NET, for security reasons, doesn't show the stack trace of the exception by default; the end user should never see this error from the server. This rule is almost always valid except when the user is a developer creating the application. In that case, it is important to show the error.

As demonstrated in the section on environment, it's easy to add this only for a development environment.

Fortunately, ASP.NET Core has a better error page than the old YPOD (yellow page of death) generated by the previous version of the ASP.NET. To use the new, fancy error page, you must be sure to install the **Microsoft.AspNetCore.Diagnostics** package from NuGet.

Now, at the beginning of you **`Configure`** method, add this line of code:

Code Listing 4-19

```
// This method is called by the runtime. Use this method to configure the
HTTP request pipeline.
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();
```

```
    //..... other code here
}
```

Restart the web server again from Visual Studio, refresh the page, and the output should contain more useful information.

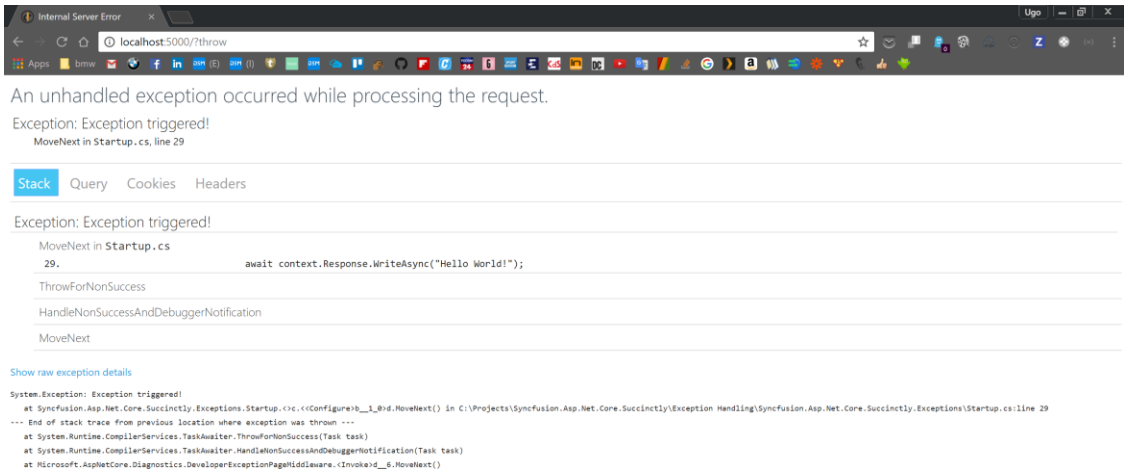

Figure 4-6: Developer Exception Page

On this page, there are four important tabs:

- **Stack** contains the stack information (line of code and call stack).
- **Query** contains all the variables coming from the query string of the request (in this case, there is only one).
- **Cookies** contains all the application cookies with their values.
- **Headers** contains the HTTP headers of the current request.

User-friendly error page

For reasons already explained, you can't show the error information on a production environment, so you have to choose between two possible ways:

- Redirect the user to a specific error page passing the status code as part of the URL.
- Re-execute the request from a new path.

Let's see the first option:

Code Listing 4-20

```
app.UseStatusCodePagesWithRedirects("~/errors/{0}.html");
```

In this case, you should create one page for each status code you want to manage and put it in a folder called `errors` in the `wwwroot` folder (combined with the **`UseStaticFiles`** middleware component, of course).

If you can't use static files, it is enough to remove **`.html`** from the string passing through the method and add a specific route on MVC. If you prefer the second option, using another middleware component will suffice:

Code Listing 4-21

```
UseStatusCodePagesWithReExecute("~/errors/{0}");
```

Finally, your error management could be this:

Code Listing 4-22

```
if (_env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
} else {
    app.UseStatusCodePagesWithRedirects("~/errors/{0}.html");
}
```

Configuration files

How ASP.NET Core handles configuration files significantly changed with the new version. Previously, you used the **`AppSettings`** section of the **`web.config`** file, but now **`web.config`** is not needed by ASP.NET. It is there only if you have to host an application on Internet Information Services (IIS); otherwise, you can strip it out.

If the **`AppSettings`** section isn't needed anymore, how do you store information?

The answer is simple. You use external files (you can have more than one). Fortunately, there is a set of classes that helps to manage that. And although it may seem more uncomfortable, it isn't.

First, choose the file format you prefer. The most common format is JSON, but if you are more familiar with XML, use it.

JSON format

Let's suppose you have an appsettings.json file like this:

Code Listing 4-23

```
{
    "database": {
        "databaseName": "my-db-name",
        "serverHost": "mySqlHost",
        "port": 1433,
        "username": "username",
        "password": "password"
    },
    "facebook": {
        "appId": "app-id",
        "appSecret": "app-secret"
    },
    "smtp": {
        "host": "mysuperhost.mysuperdomain.com",
        "username": "imperugo@gmail.com",
        "password": "my-super-secret-password",
        "enableSsl": true,
        "port": 587
    }
}
```

The more comfortable way to have all this information in a C# application is by using a class with a set of properties. In a perfect world, it would be a class with the same structure of JSON, like this:

Code Listing 4-24

```
namespace Syncfusion.Asp.Net.Core.Succinctly.Environments
{
    public class Configuration
    {
        public DatabaseConfiguration Database { get; set; }
        public FacebookConfiguration Facebook { get; set; }
        public SmtpConfiguration SmtpConfiguration { get; set; }
    }

    public class DatabaseConfiguration
    {
        public string DatabaseName { get; set; }
        public string ServerHost { get; set; }
        public int Port { get; set; }
```

```
        public string Username { get; set; }
        public string Password { get; set; }

        public string ConnectionString => $"Server=tcp:{ServerHost},{Port
};Database={DatabaseName};User ID={Username};Password={Password};Encrypt=
True;TrustServerCertificate=False;Connection Timeout=30;";
    }

    public class FacebookConfiguration
    {
        public string AppId { get; set; }
        public string AppSecret { get; set; }
    }

    public class SmtpConfiguration
    {
        public string Host { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
        public bool EnableSsl { get; set; }
        public int Port { get; set; }
    }
}
```

At this point, it injects the C# classes with the values from the JSON file. Because the configuration file is usually needed at the beginning of the application lifecycle, you have to add the lines of code to the `Startup` class—or more accurately, into the constructor. Before doing this, however, you need to add some packages:

- Microsoft.Extensions.Configuration.EnvironmentVariables
- Microsoft.Extensions.Configuration.FileExtensions
- Microsoft.Extensions.Configuration.Json
- Microsoft.Extensions.Configuration.Binder

There’s no need to say more about the purpose of the packages; their names speak for themselves. So, let's see how the code looks:

Code Listing 4-25

```
private readonly IHostingEnvironment hostingEnvironment;
private readonly Configuration configuration;

public Startup(IHostingEnvironment hostingEnvironment)
{
    this.hostingEnvironment = hostingEnvironment;
```

```
    var builder = new ConfigurationBuilder()
        .SetBasePath(hostingEnvironment.ContentRootPath)
        .AddJsonFile("appsettings.json", false, true)
        .Build();

    var config = new Configuration();
    builder.Bind(config);

    // here the variable configuration contains all the info
    // from the JSON file
}
```

Manage different environments

We already explained the importance of having different environments and how to manage them via C#. The same applies to the configuration.

More than ever, you now need different configuration files—one for each environment. Thanks to ASP.NET Core, this is easy to manage.

To take advantage of what the framework offers, you have to follow a few rules: the first one is related to configuring file names.

For different files, having one for each environment allows you to add the environment name into the file name. For example, **`appsettings.json`** for the development environment must be called **`appsettings.Development.json`**, and **`appsettings.Production.json`** would be used for a production environment.

The second rule is related to the differences among the files. You don't need to have the complete JSON copied in each configuration file because ASP.NET Core will merge the files, overriding only what is specified in the environment configuration file.

To understand what this means, imagine you have the same database instance but a different database name. You can log into the database server using the same credentials, but the same server hosts both from production to development; it just switches the database.

To cover this scenario and keep using the appsettings.json file you used before, look at the appsettings.Development.json file.

Code Listing 4-26

```
{
    "database": {
        "databaseName": "my-development-db-name",
```

```
    },
}
```

And look at **appsettings.Production.json**.

Code Listing 4-27

```
{
    "database": {
        "databaseName": "my-production-db-name",
    },
}
```

This is awesome because you can keep the configuration files very lean, but now you need to educate ASP.NET Core to handle multiple configuration files. To do that, change the code you wrote previously to this:

Code Listing 4-28

```
private readonly IHostingEnvironment hostingEnvironment;
private readonly Configuration configuration;

public Startup(IHostingEnvironment hostingEnvironment)
{
    this.hostingEnvironment = hostingEnvironment;

    var builder = new ConfigurationBuilder()
        .SetBasePath(hostingEnvironment.ContentRootPath)
        .AddJsonFile("appsettings.json", false, true)
        .AddJsonFile($"appsettings.{hostingEnvironment.EnvironmentName}.j
son", true)
        .AddEnvironmentVariables()
        .Build();

    configuration = new Configuration();
    builder.Bind(configuration);
}
```

Now, for a development environment, you will get **my-development-db-name** as the database name; otherwise, it will be **my-production-db-name**. Remember to keep the same JSON structure in your environment configuration files.

Dependency injection

This last part is related to the use of the configuration values across the application, such as the controller, services, and whatever else needs to read the configuration values.

Register the instance of configuration class you created earlier to the ASP.NET Core dependency injection container. As usual, go into the **`Startup.cs`** class and register the instance.

Code Listing 4-29

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton(configuration);
}
```

For configuration scenarios, the Singleton lifecycle is the best option. To get the values into the services, inject the instance on the constructor.

Code Listing 4-30

```
namespace Syncfusion.Asp.Net.Core.Succinctly.Environments
{
    public class MySimpleService
    {
        private readonly Configuration configuration;

        public MySimpleService(Configuration configuration)
        {
            this.configuration = configuration;
        }
    }
}
```

Logging

Logging is very important in a web application, and it is very difficult to implement. This is why so many logging frameworks are available. Search the term "logging" on nuget.org, and you'll see there are more than 1,900 packages.

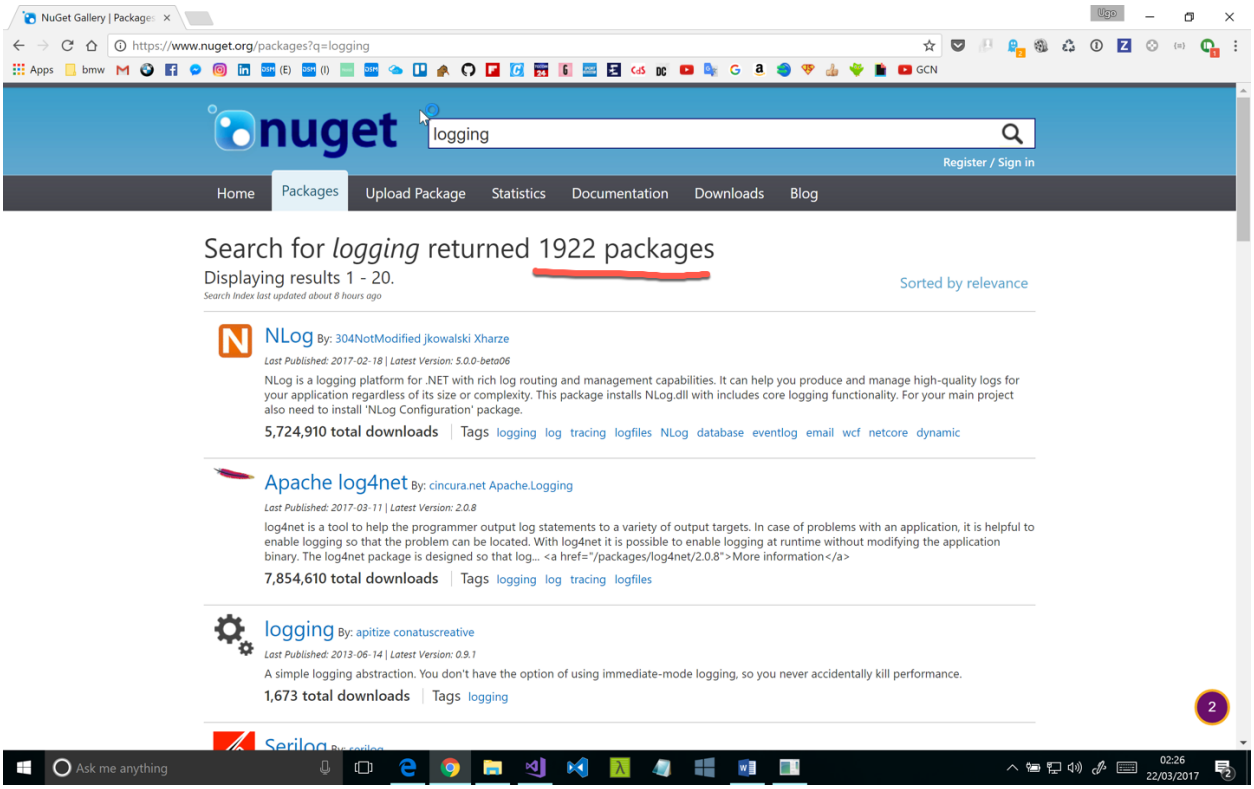

Figure 4-7: NuGet Logging Packages

Of course, not all are logging frameworks, but almost all are related to logging. Logging is very individualized. It is related to the particular environment or application you are working on.

Modern applications are composed of several packages. The combination of both (several logging frameworks and several packages) makes the logging ecosystem very complicated. What happens if each package uses its own logging framework or one that is different from the one used in your application?

It would be a complicated mess to configure each one for each environment. You'd probably spend a lot of time configuring logging instead of writing good code.

To solve this problem, before ASP.NET Core, there was a library called **Common.Logging .NET** (the official repository is on GitHub) that provided a simple logging abstraction to switch between different logging implementations, like Log4net, NLog, Serilog, and so on.

It would be pretty cool if all packages used this, because you could configure the logging once in a single place. Unfortunately, this doesn't allow you to log the code that comes from the .NET Framework because it doesn't have dependencies as opposed to external libraries.

With ASP.NET Core, this problem is completely solved. You don't have to use the Common.Logging .NET library because the framework offers something similar out of the box, and it is integrated with all the packages.

Configure logging

First, you have to choose the output of the log you want—for example, a **console application**, a **trace source**, an **event log,** and so on. For Kestrel, it could be very helpful to use the console output. Add **Microsoft.Extensions.Logging.Console** using Visual Studio package manager.

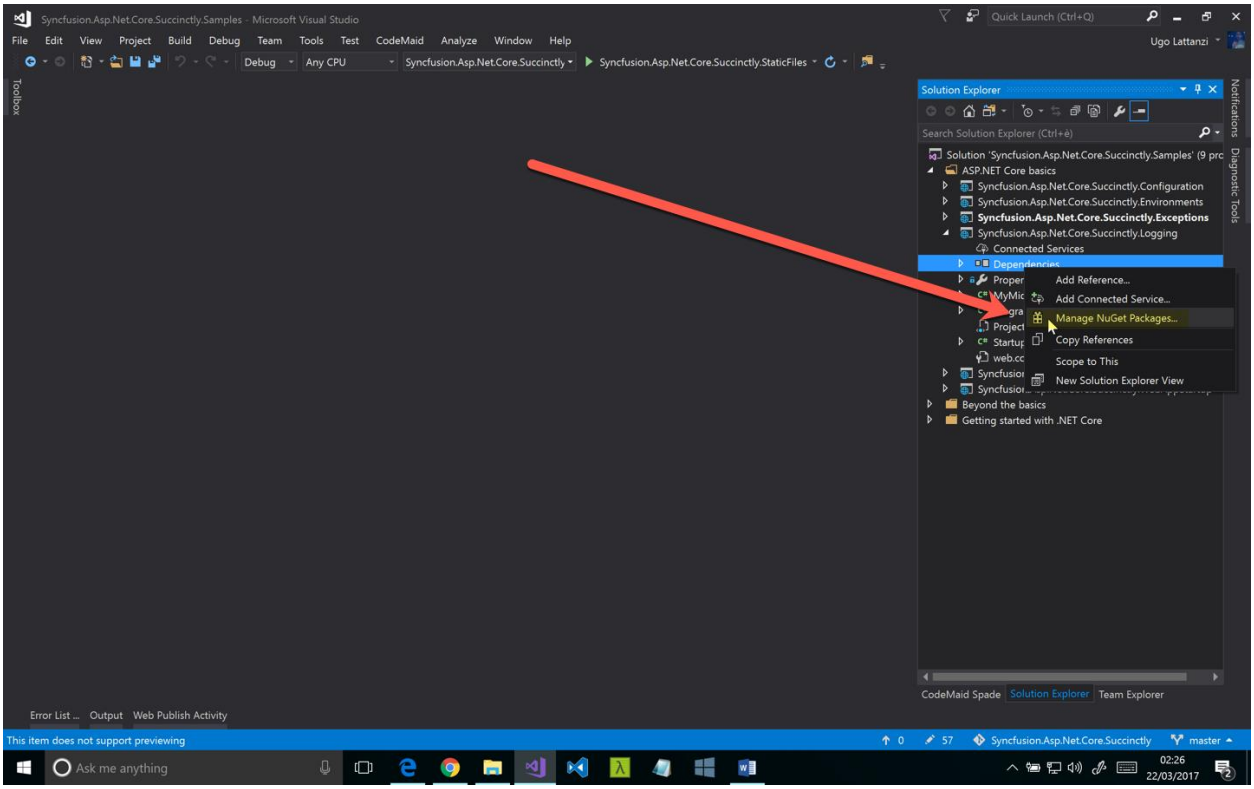

Figure 4-8: Managing NuGet Packages Using Visual Studio

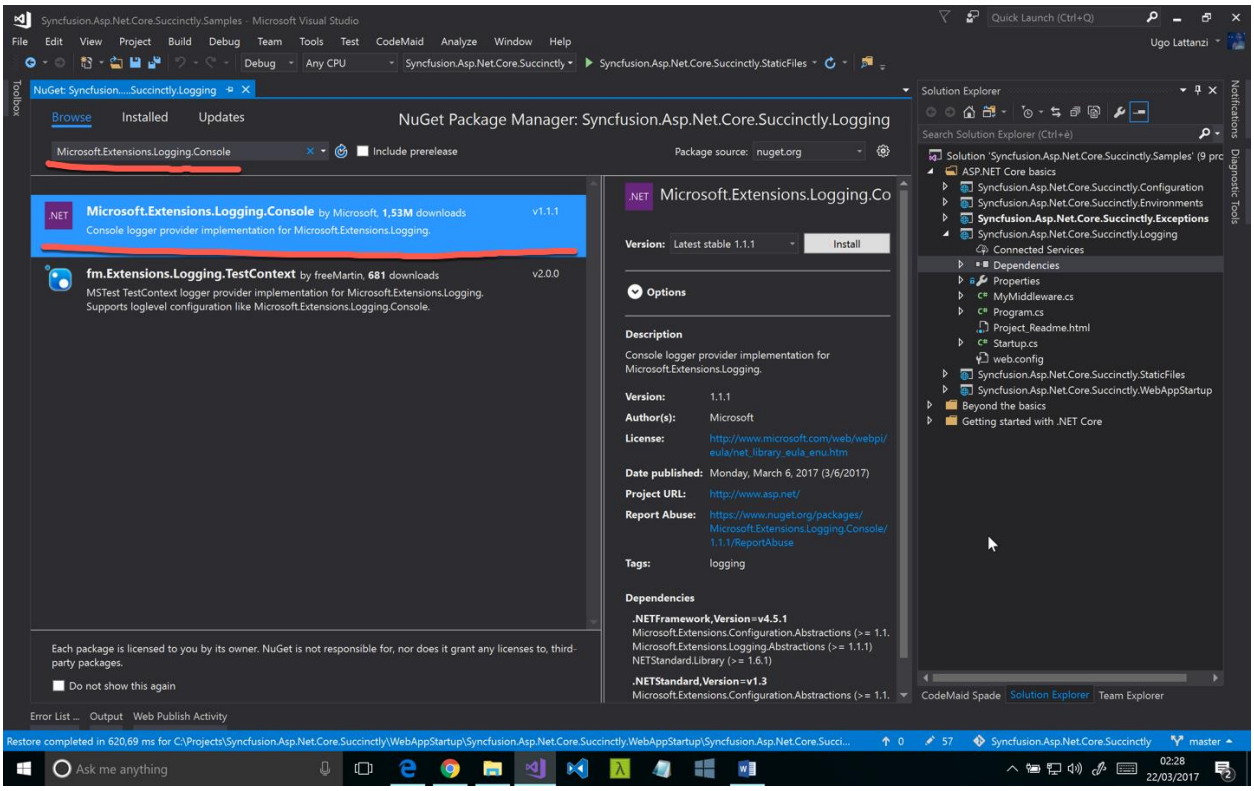

Figure 4-9: Installing Console Logging

The package **Microsoft.Extensions.Logging.Console** could already be installed if you used one of the templates available with Visual Studio 2015.

Now, in the **`Startup`** class, configure the log output. As explained in previous sections, the **`Configure`** method is the best place to do this. **`LoggingFactory`** is the class responsible for generating the log instance.

Code Listing 4-31

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.Logging
{
    public class Startup
    {
        // This method is called by the runtime. Use this method to add s
ervices to the container.
        // For more information on how to configure your application, vis
it http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
```

```
        {
        }

        // This method is called by the runtime. Use this method to confi
gure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, ILoggerFactory log
gerFactory)
        {
            loggerFactory.AddConsole();

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

Visual Studio and VS Code offer an incredible feature called IntelliSense, so when typing **`loggerFactory.`**, it suggests all available options. If you did everything correctly, you should get the following.

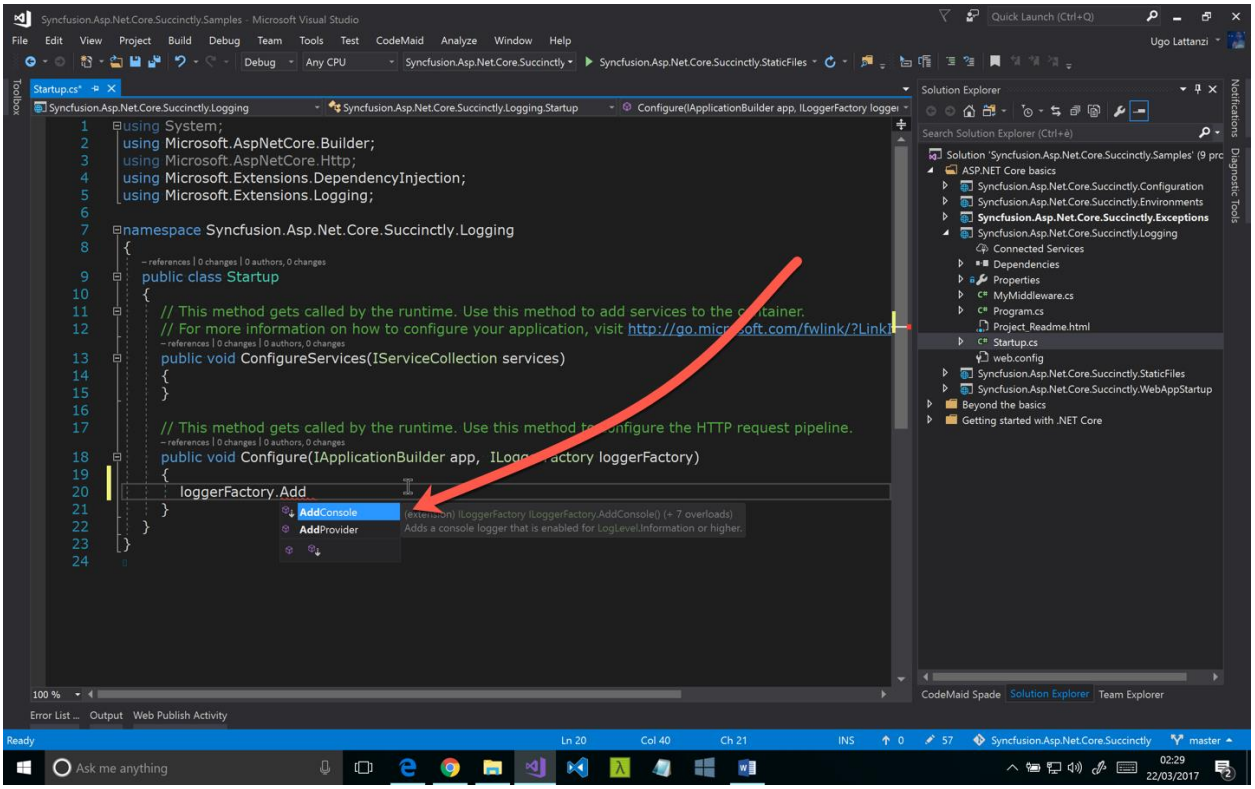

Figure 4-10: Adding Console Provider

Testing logging

The best part of this logging system is that it is natively used by the ASP.NET Core Framework. When running the application, you should see the output depicted in Figure 4-12.

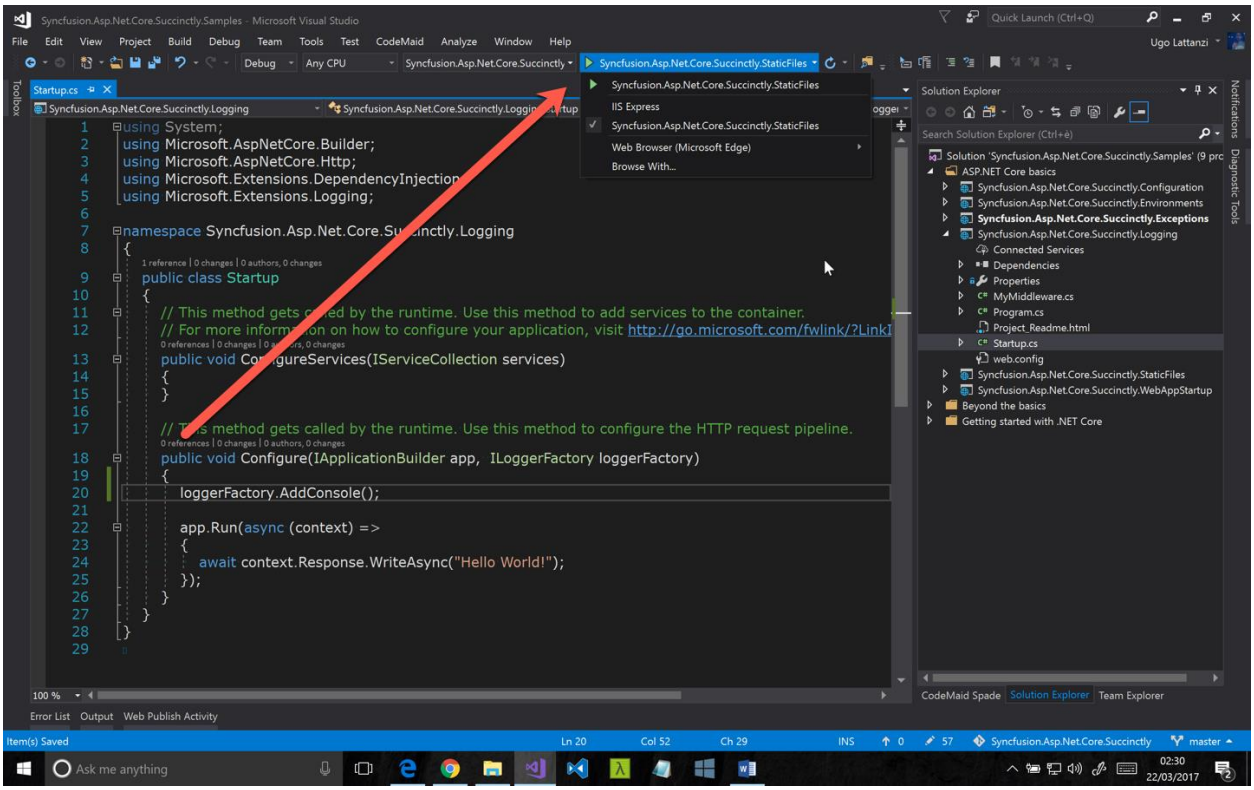

Figure 4-11: Running a Web Application Using Kestrel

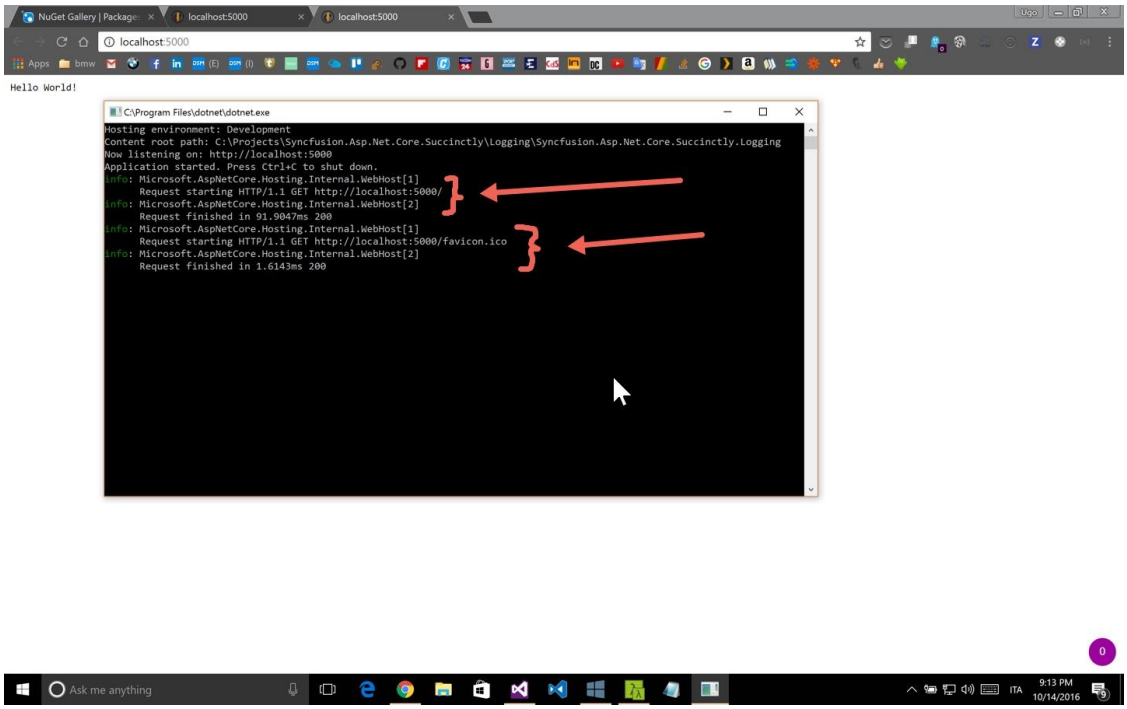

Figure 4-12: Web Application Console Output

Each request is logged twice: the first when the server receives the request and the second when the server completes the request. In Figure 4-12, there are four logs in two groups: the first when the browser requests the page and the second when it requests the favicon.

Change log verbosity

Without tuning the configuration, the log implementations write all information into the output. To restrict large amounts of data in the log output, you can configure it to log only information starting from a specific level. ASP.NET Core logging has six levels:

- Trace = 0
- Debug = 1
- Information = 2
- Warning = 3
- Error = 4
- Critical = 5

In a production environment, you probably want to log starting from the Warning or Error level. To do this, specify the right minimum log level in the **`AddConsole`** method.

Notice that each logger implementation has a different way to specify the minimum level of logging, so this implementation could not work with **TraceSource** or **EventLog**.

Code Listing 4-32

```
public void Configure(IApplicationBuilder app,  ILoggerFactory loggerFact
ory)
{
    Func<string, LogLevel, bool> filter = (name, level) => level >= LogLe
vel.Error;

    loggerFactory.AddConsole(filter);

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

Add a log to your application

You've seen how to configure the log and its output with the .NET Core Framework, but you didn't see how to use it in classes. Thanks to dependency injection, it is very easy—just inject the logger instance into the constructor and use it.

The class to inject is **`ILogger<T>`**, where **`T`** is the class that needs to be logged.

Code Listing 4-33

```
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.Logging
{
    public class MyService
    {
        private readonly ILogger<MyService> _logger;

        public MyService(ILogger<MyService> logger)
        {
            _logger = logger;
        }

        public void DoSomething()
        {
            _logger.LogInformation("Doing something ...");
```

```
            //.... do something
        }
    }
}
```

Create your custom logger

In this book, we are not going to explain how to create your own custom logger, but there are several repositories where you can see how to do that. Here is a short list:

- Serilog at https://github.com/serilog/serilog-extensions-logging.
- Elmah.io at https://github.com/elmahio/Elmah.Io.Extensions.Logging.
- Loggr at https://github.com/imobile3/Loggr.Extensions.Logging.
- NLog at https://github.com/NLog/NLog.Extensions.Logging.
- Slack at https://github.com/imperugo/Microsoft.Extensions.Logging.Slack.
- MongoDb at https://github.com/imperugo/Microsoft.Extensions.Logging.MongoDb.

Conclusion

In this chapter, you learned how to use middleware components to implement features needed by web applications, such as static files, exception handling, dependency injection, hosting, and environment.

Other cool features are available—like data protection and caching. To find out more, go to www.asp.net.

Chapter 5 Beyond the Basics: Application Frameworks

HTTP isn't just for serving up webpages; it's also for serving APIs that expose services and data over HTTP protocol. This chapter will gently introduce you to managing these scenarios using ASP.NET MVC, which hasn't changed much from the previous version.

You'll manage controllers, views, APIs, the newest and coolest tag helper, and you'll play with the view components. But before proceeding, it's important to know that the part related to MVC is just a small introduction. A complex framework like this would easily require an entire book, which is not in the scope of a work targeting ASP.NET, such as this book.

Web API

Web API (application programming interface) is a set of subroutine definitions with the scope of managing data between clients and servers. Over the last few years, the trend has been to build APIs over HTTP protocol, allowing third-party apps to interact with a server thanks to the application protocol.

Probably the most well-known example is Facebook, which allows users to share content, posts, manage pages, and do more from any client—mobile app, desktop, or whatever. This is due to a set of APIs and HTTP, but how does it work?

The good thing about using HTTP protocol is that there are no major differences between the classic web navigation except for the result. In a normal HTTP navigation, when we use a browser to call **`www.tostring.it`**, the server returns HTML. For the APIs, it returns data using JSON format.

The result could also be XML or any other type of structured data. The point is that the result doesn't contain any information about the layout or interaction, like CSS and JavaScript. With the previous version of ASP.NET, a specific library managed APIs. It was called Web API.

With the newest version, this library doesn't exist, and you can handle API and classic requests using the same framework because Web API has merged into the MVC framework. This is the primary difference between the old ASP.NET and the new.

When you think about what we were using in the previous version, it makes absolute sense. Most of the code between the two frameworks was similar. Think about the controller, attributes, and dependency injection—same code, different namespaces.

Installation

ASP.NET MVC Core is not different from what you saw in the previous chapters; you have to install and configure the necessary packages respectively using NuGet and the Startup class.

The package is **Microsoft.AspNetCore.Mvc**, and it has built-in support for building Web APIs. Once you have it installed, register the service and configure it within your web application like this:

Code Listing 5-1

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironmen
t env, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();

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

            app.UseMvcWithDefaultRoute();
        }
    }
}
```

Playing around with URLs and verbs

Because you are managing data and not HTML pages, HTTP verbs are key to understanding what the client needs. There is a kind of convention that almost all API implementations respect and use different HTTP verbs according to the type of action needed.

Suppose you have this request: **api/users**.

Table 1

Resource Sample	Read (GET verb)	Insert (POST verb)	Update (PUT verb)	Partially Update (PATCH verb)	Delete (DELETE verb)
Action	Gets a list of users.	Creates a user.	Updates users with a batch update.	Batch-updates users only with the attributes present in the request.	Errors or deletes all users, depending on what you want.
Response	Lists users.	New user or redirects to the URL to get the single user.	No payload; only HTTP status code.	No payload; only HTTP status code	No payload; only HTTP status code.

In case you want to manage a single user, the request should be **`api/users/1`** where **`1`** is the ID of the user.

Table 2

Resource Sample	Read (GET verb)	Insert (POST verb)	Update (PUT verb)	Partially Update (PATCH verb)	Delete (DELETE verb)
Action	Gets a single user.	Returns an error because the user already exists.	Updates the specified user.	Partially updates the user only with the attributes present in the request.	Deletes the specified user.
Response	Single user.	No payload; only HTTP status code.	Updated user or redirects to URL to get the single user.	Updated user (complete object) or redirects to URL to get the single user.	No payload; only HTTP status code.

To summarize, the URL combined with the HTTP verb allows you to understand what has to be done.

- **`/api`** is the prefix indicating that the request is an API request and not a classic web request.

- **users** is the MVC controller.
- **verb** identifies the action to execute.

Return data from an API

Now that everything is correctly configured, you can create your first MVC controller. Not to deviate too much from what you used in the previous version, let's create a folder called Controllers.

Although not mandatory, the best practice is to create a folder for all the API controllers to separate API from the standard controllers.

Code Listing 5-2

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebApi.Controllers.APIs
{
    [Route("api/[controller]")]
    public class UsersController : Controller
    {
        [HttpGet]
        public User[] Get()
        {
            return new[]
            {
                new User() {Id = 1, Firstname = "Ugo", Lastname = "Lattan
zi", Twitter = "@imperugo"},
                new User() {Id = 2, Firstname = "Simone", Lastname = "Chi
aretta", Twitter = "@simonech"},
            };
        }

        [HttpGet("{id}")]
        public User Get(int id)
        {
            var users = new[]
            {
                new User() {Id = 1, Firstname = "Ugo", Lastname = "Lattan
zi", Twitter = "@imperugo"},
                new User() {Id = 2, Firstname = "Simone", Lastname = "Chi
aretta", Twitter = "@simonech"},
            };

            return users.FirstOrDefault(x => x.Id == id);
```

```
        }
    }

    public class User
    {
        public int Id { get; set; }
        public string Firstname { get; set; }
        public string Lastname { get; set; }
        public string Twitter { get; set; }
    }
}
```

As you can see, the controller is simple and the code is not so different from the previous version. Let's analyze this step by step.

Code Listing 5-3

```
[Route("api/[controller]/[action]")]
```

This says that the controller can manage all the requests with the **`api`** prefix in the URL. Basically, if the URL doesn't start with **`/api`**, the controller will never handle the request.

Code Listing 5-4

```
[HttpGet]
```

This attribute specifies the verb for the action it is decorating on. You could use all the possible verbs, **`HttpGet`**, **`HttpPost`**, **`HttpPut`**, and so on.

Update data using APIs

You just saw how to return data as well as partially read input information from the query string. Unfortunately, this is not so useful when you have to send a lot of data from the client to the server because there is a limit related to the number of characters you can add to the URL. For this reason, you have to move to another verb.

HTTP/1.1 doesn't specify limits for the length of a query string, but limits are imposed by web browser and server software. You can find more information at http://stackoverflow.com/questions/812925/what-is-the-maximum-possible-length-of-a-query-string.

This approach is not so different from what you saw, the only difference is the place where the data comes from: the body instead of the query string.

Code Listing 5-5

```
// Adding user
[HttpPost]
public IActionResult Update([FromBody] User user)
{
    var users = new List<User>();
    users.Add(user);

    return new CreatedResult($"/api/users/{user.Id}", user);
}

//Deleting user
[HttpDelete]
public IActionResult Delete([FromQuery] int id)
{
    var users = new List<User>
    {
        new User() {Id = 1, Firstname = "Ugo", Lastname = "Lattanzi", Twi
tter = "@imperugo"},
        new User() {Id = 2, Firstname = "Simone", Lastname = "Chiaretta",
Twitter = "@simonech"},
    };

    var user = users.SingleOrDefault(x => x.Id == id);

    if (user != null)
    {
        users.Remove(user);

        return new EmptyResult();
    }

    return new NotFoundResult();
}
```

As you can see, the only differences are as follows.

Code Listing 5-6

```
[FromQuery]
```

This specifies that data comes from the payload of the HTTP request.

Code Listing 5-7

```
return new CreatedResult($"/api/users/{user.Id}", user);
```

This returns the created object and the URL where the client can get the user again. The status code is 201 (created).

Code Listing 5-8

```
return new EmptyResult();
```

Basically, this means Status Code 200 (OK). The response body is empty.

Code Listing 5-9

```
return new NotFoundResult();
```

This is the 404 message that is used when the requested client can't be found.

Testing APIs

In order to test the API, the browser can be used because APIs are being implemented over HTTP protocol, but there are other applications that can help with this, too. One popular application is Postman, which can be downloaded for free from getpostman.com.

The following screenshots show how to test the code we used previously.

Retrieve users (/api/users using GET).

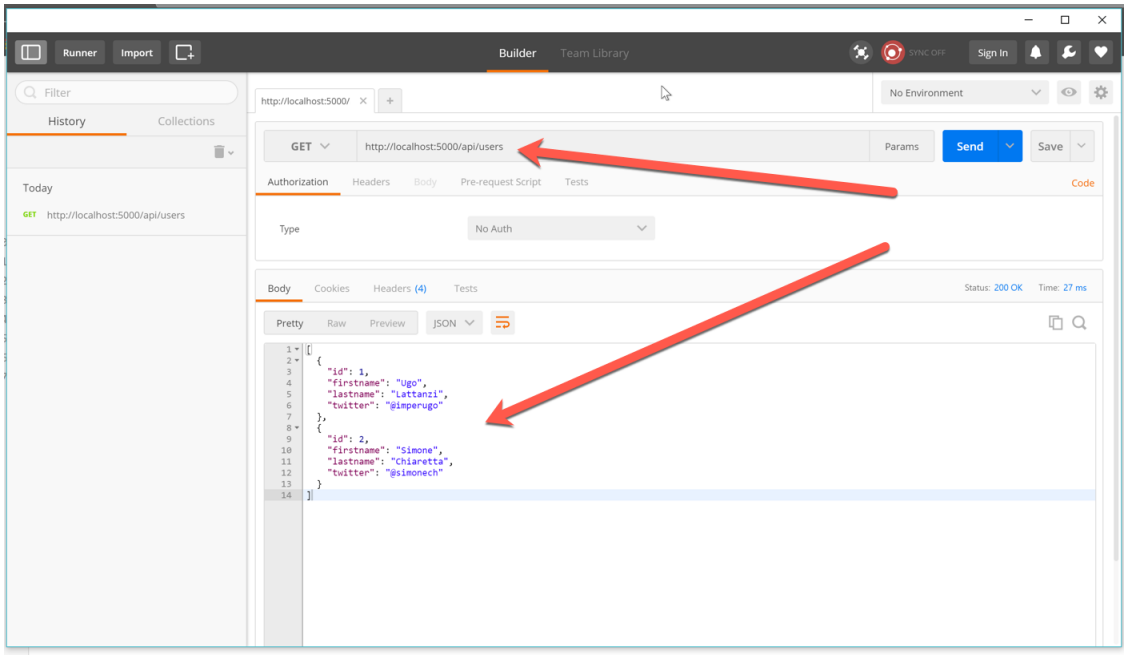

Figure 5-1: Using Postman to Test REST Endpoint-1

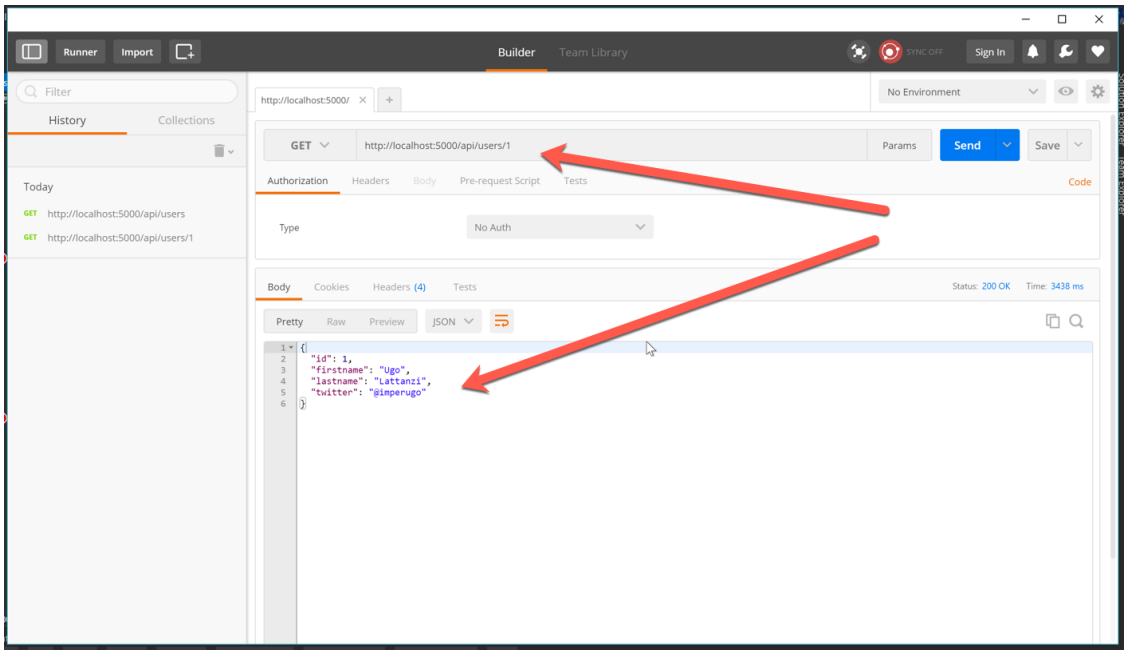

Figure 5-2: Using Postman to Test REST Endpoint-2

Create user (/api/users using POST).

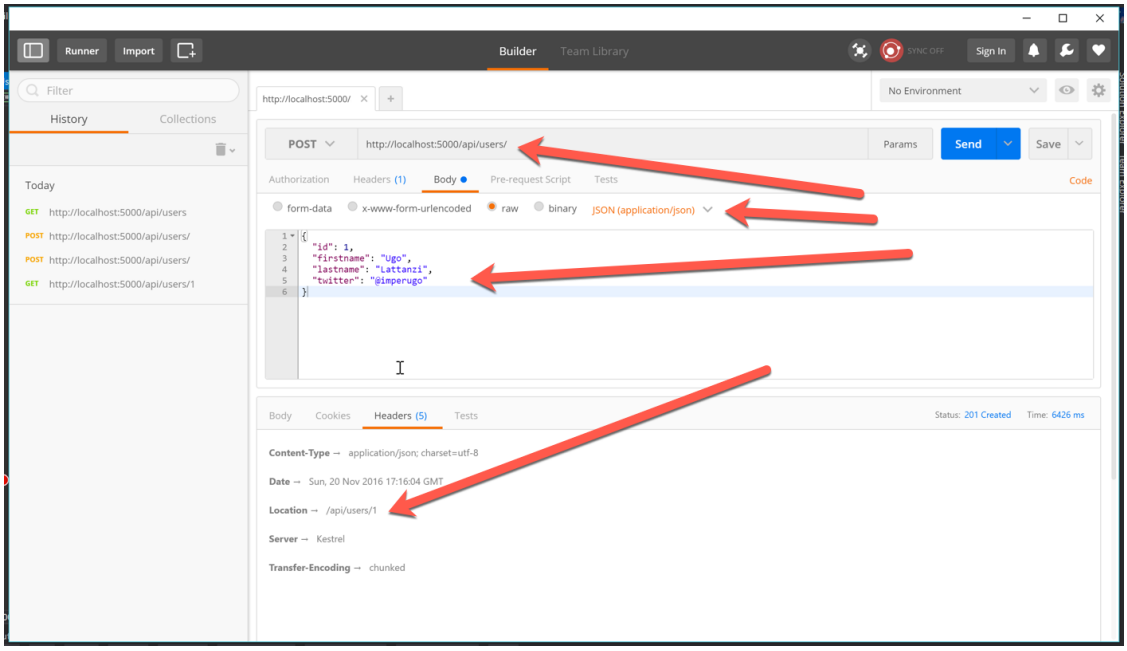

Figure 5-3: Using Postman to Test REST Endpoint-3

Delete user (/api/users?id=2 using DELETE).

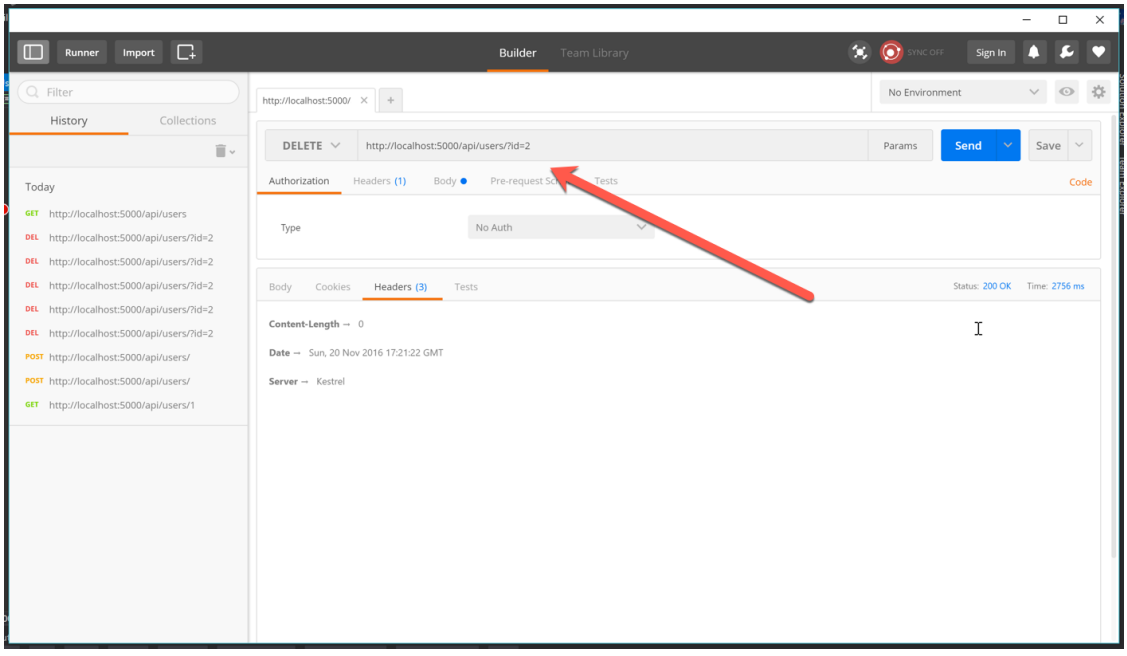

Figure 5-4: Using Postman to Test REST Endpoint-4

ASP.NET MVC Core

As mentioned in the previous section, ASP.NET MVC isn't different than Web API, so the code you are going to see here will be very similar, except for the result.

Because you already installed and registered ASP.NET MVC for APIs, you can jump directly to the code. In the same folder (**`Controller`**) where you created **`UserController`**, you can create another controller with the scope of serving the main page of the website. In this case, call it **`HomeController`**.

Code Listing 5-10

```
using Microsoft.AspNetCore.Mvc;

namespace Syncfusion.Asp.Net.Core.Succinctly.Mvc.Controllers.MVC
{
    public class HomeController : Controller
    {
        [HttpGet]
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

This code is similar to the API controller, the only differences are the missing **`route`** attribute on the controller and the **`View`** method used to return the action. The first one is not needed because you don't have to override the default routing configuration (we will discuss this later in the chapter), and the second one indicates to the controller that it has to render a view and not JSON.

If you try to run this code, you'll get an error like this.

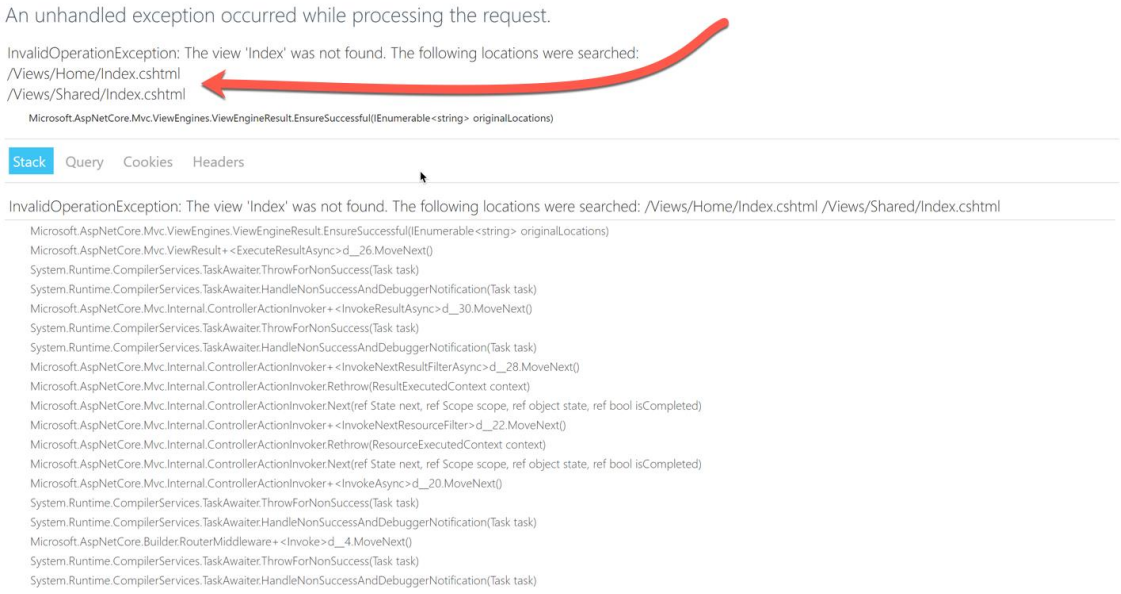

Figure 5-5: Unable to Find the MVC View

This happens because the MVC framework cannot find the view file, but how does it locate the file? The logic behind it is very simple; if you don't specify any information about the view, everything happens using conventions.

First, it is important to know that all the views are placed in the folder `Views` in the root of the project, as you can see in the following screenshot.

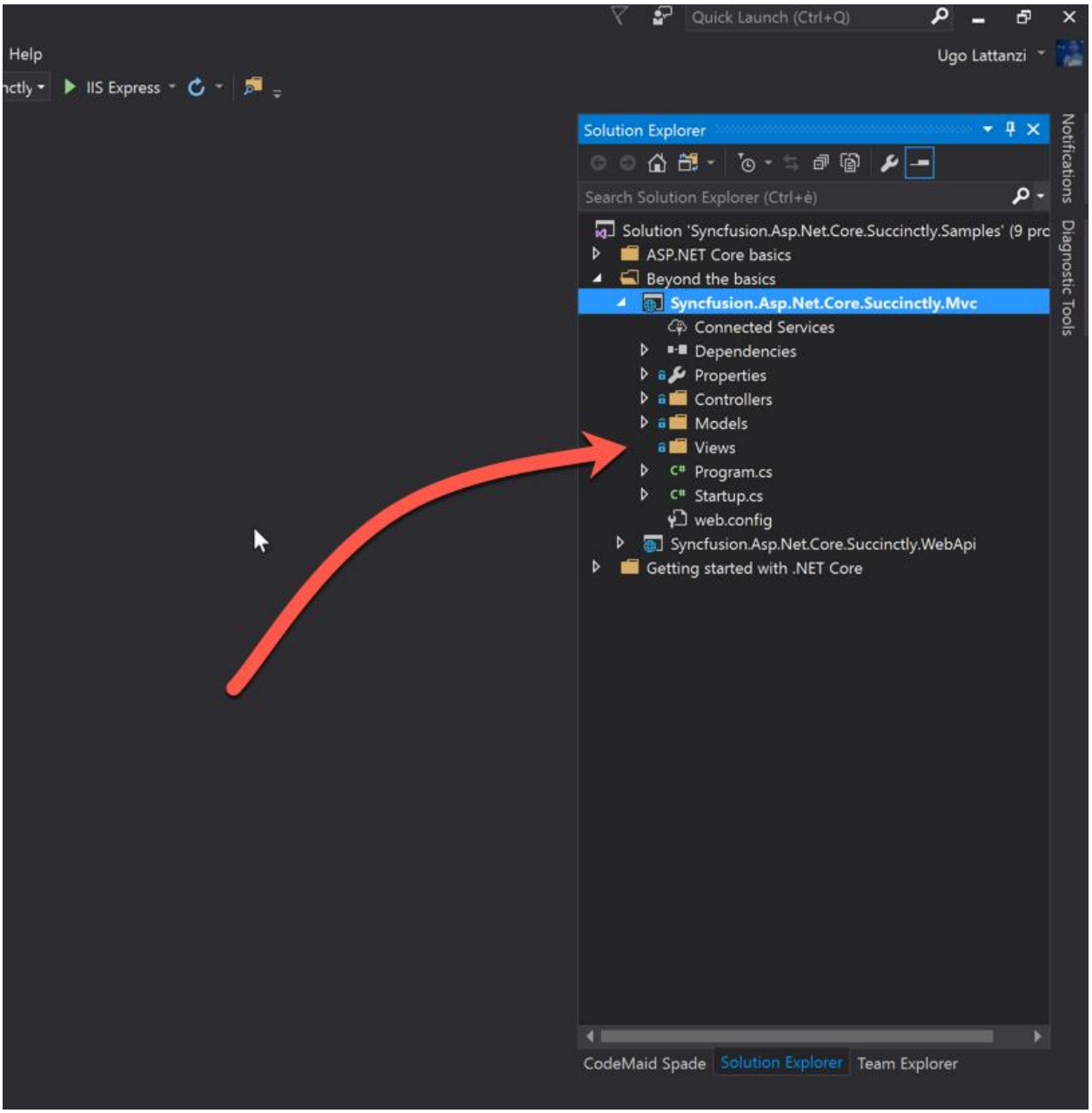

Figure 5-6: The Views Folder

Inside, there must be a folder for each MVC controller you created. In this case, you have just one controller called **`HomeController`**, so you must create the folder Home inside Views.

Finally, it's time to create our views. Using Visual Studio 2017 makes it really simple.

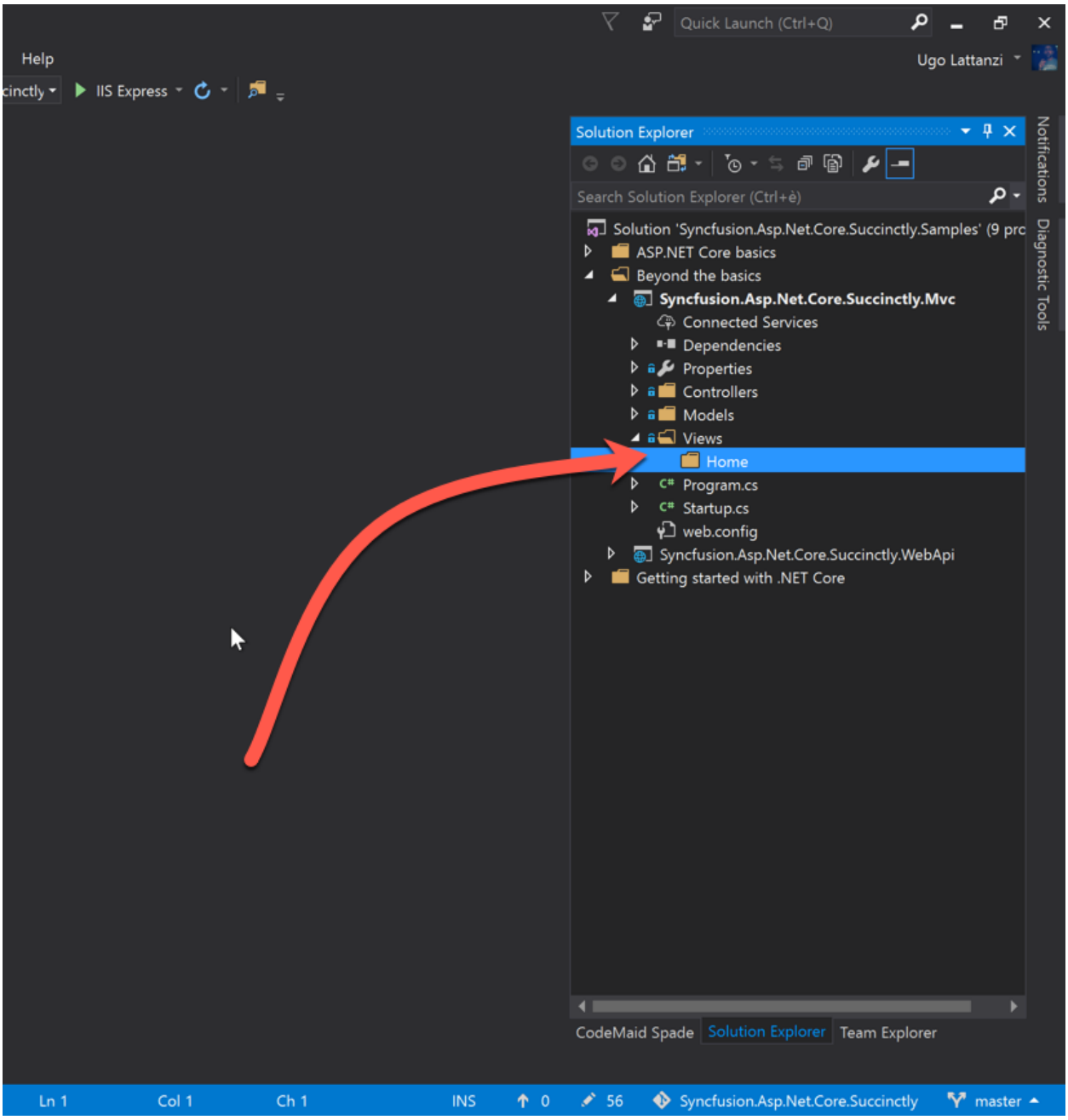

Figure 5-7: The Views Folder for the Home Controller

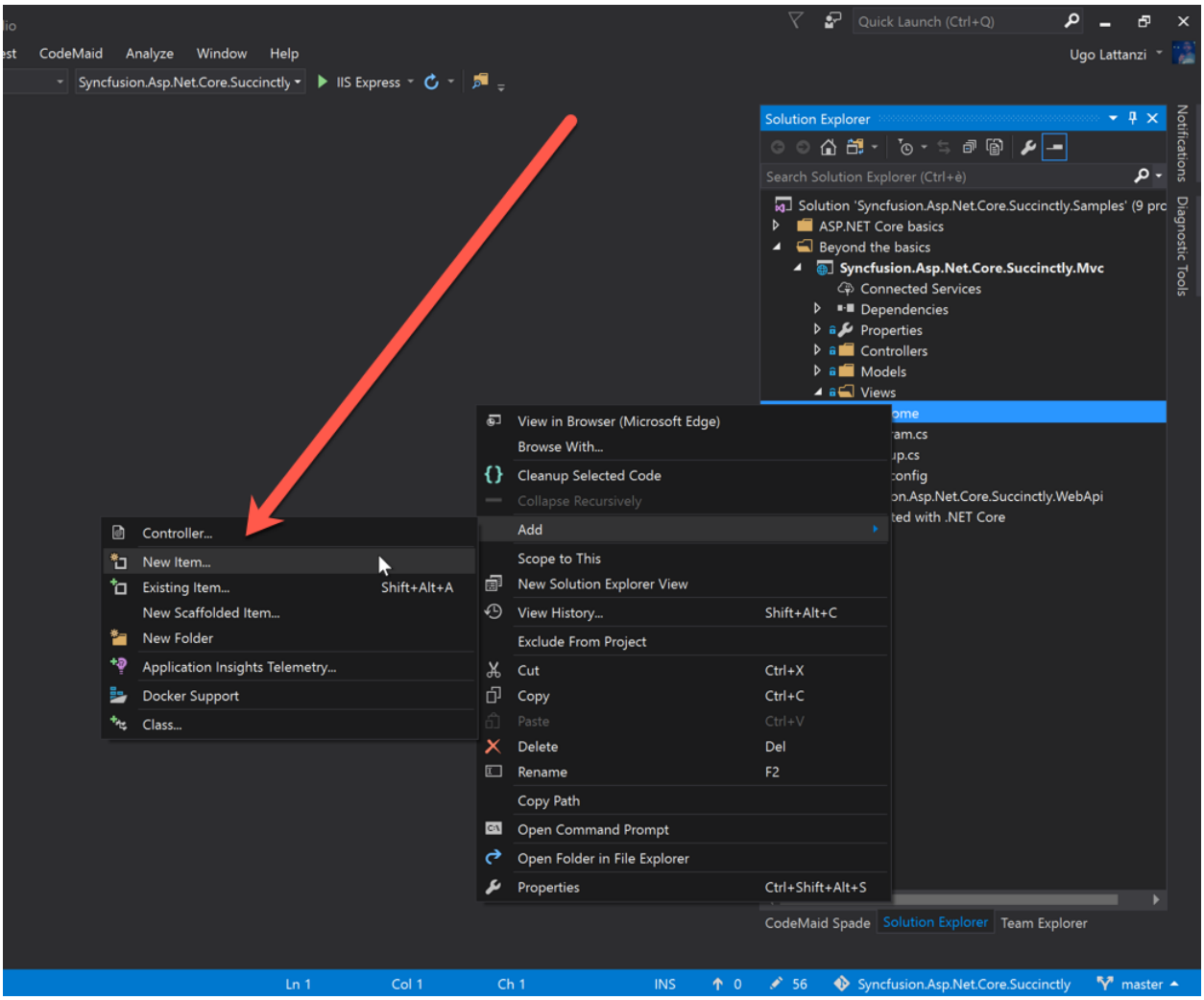

Figure 5-8: Adding the View-1

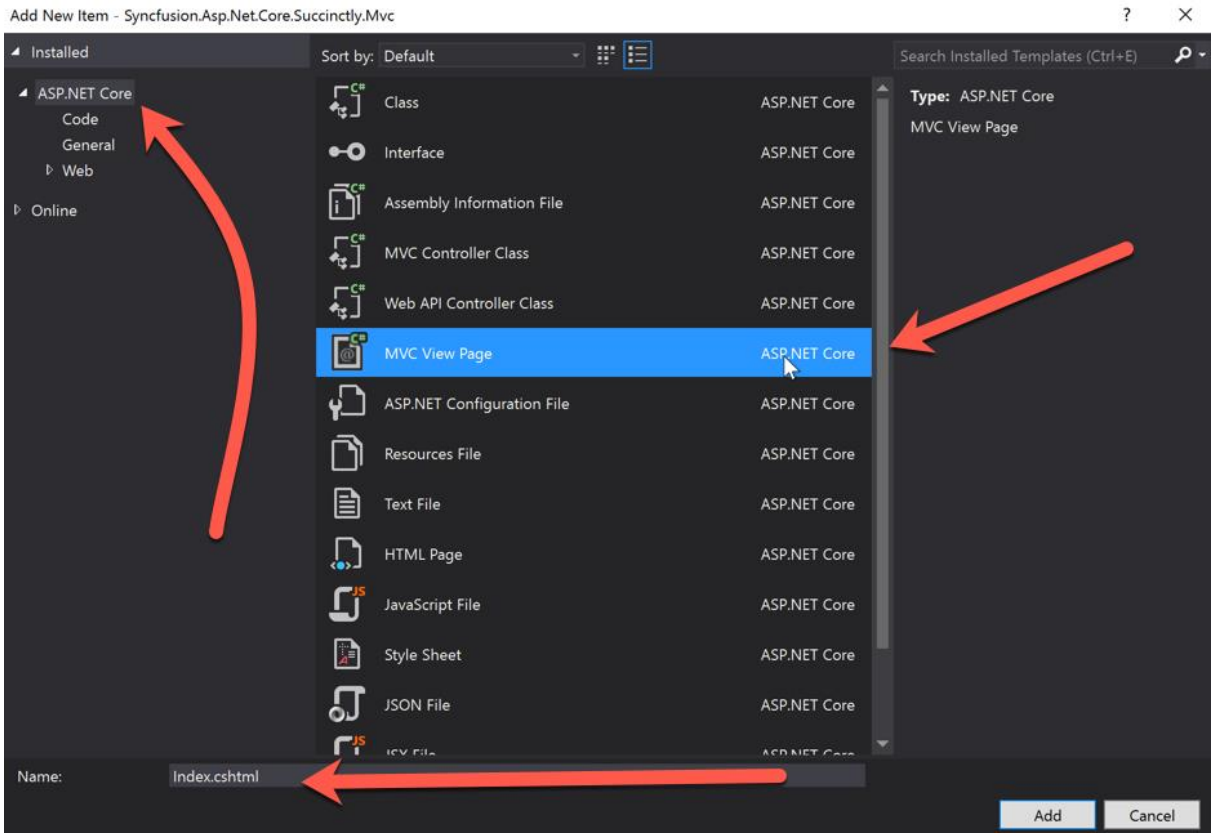

Figure 5-9: Adding the View-2

Now, if you open the newest file, you'll see that it is almost empty. It contains just a few strange sections that you'll recognize because they have different colors and use @ in the beginning.

These parts are managed server-side and contain instructions on how to change the output to render the final HTML. Everything is possible thanks to Razor, the view engine embedded within ASP.NET MVC Core.

A good introduction to Razor syntax is available at asp.net/web-pages/overview/getting-started/introducing-razor-syntax-c. Because you have to render a webpage, you first have to create the right markup, so let's add the following code to our view.

Code Listing 5-11

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Hello World</title>
</head>
<body>
    <h1>Hello World</h1>
```

```
</body>
</html>
```

When running the application again, the result should not be an error, but a simple HTML page.

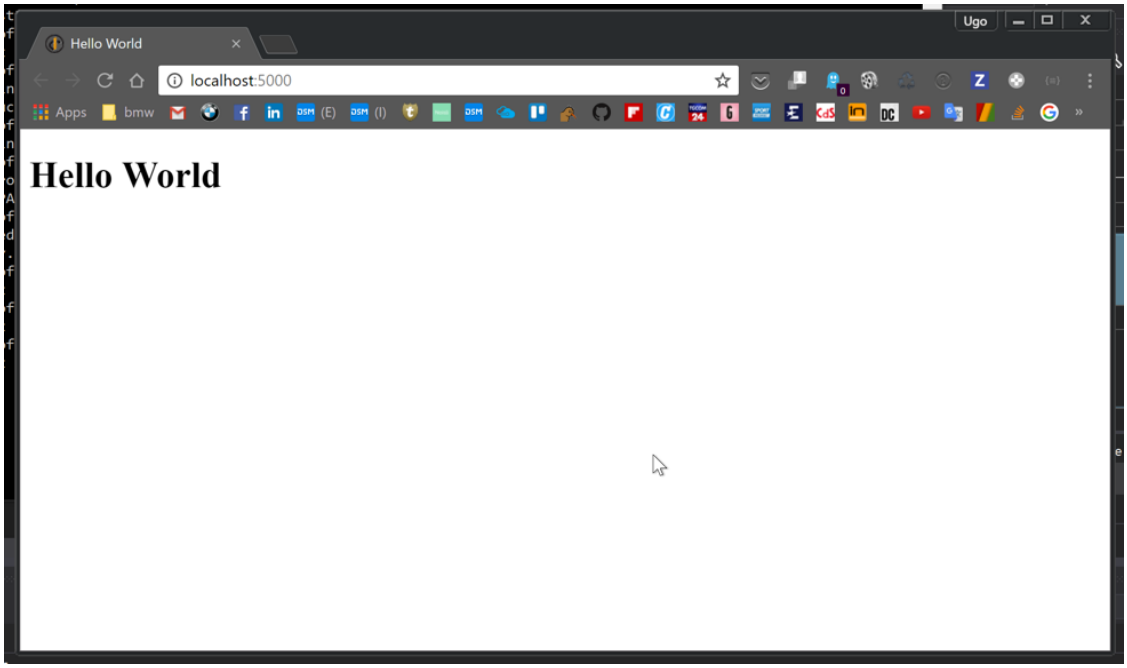

Figure 5-10: Server-Side Page Rendered Using MVC

This is definitely an improvement, but you're returning just a static file; there isn't any kind of transformation. But that isn't needed to use MVC because you already have a specific middleware component to manage static files.

To make the view more elaborated and to use something from the server, you have to go back to the controller and send the data to the view.

Code Listing 5-12

```
using Microsoft.AspNetCore.Mvc;
using Syncfusion.Asp.Net.Core.Succinctly.Mvc.Models;

namespace Syncfusion.Asp.Net.Core.Succinctly.Mvc.Controllers.MVC
{
    public class HomeController : Controller
    {
        [HttpGet]
        public IActionResult Index()
        {
            var users = new[]
```

```
            {
                new User() {Id = 1, Firstname = "Ugo", Lastname = "Lattan
zi", Twitter = "@imperugo"},
                new User() {Id = 2, Firstname = "Simone", Lastname = "Chi
aretta", Twitter = "@simonech"},
            };

            return View(users);
        }
    }
}
```

Notice that we have reused the class **User**, which was previously created for the API response. This code doesn't need explanation; you're just sending an array of users as a model to a **View** method.

In our view, it's time to get users from the controller and print the data to the page. The first thing to do is to specify what kind of model the view is using. In our case, it's **IEnumerable<User>**.

Code Listing 5-13

```
@model IEnumerable<Syncfusion.Asp.Net.Core.Succinctly.Mvc.Models.User>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Hello World</title>
</head>
<body>
    <h1>Hello World</h1>
</body>
</html>
```

Now you have to iterate all the users from the model and show the data.

Code Listing 5-14

```
@model IEnumerable<Syncfusion.Asp.Net.Core.Succinctly.Mvc.Models.User>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Hello World</title>
</head>
```

```
<body>
    <h1>Users</h1>
    <div>
        @foreach(var user in Model) {
            <hr />
            <p>Firstname: @user.Firstname</p>
            <p>Lastname: @user.Lastname</p>
            <p>Twitter: <a href="http://www.twitter.com/@user.Twitter"> @
user.Twitter</a></p>
        }
    </div>
</body>
</html>
```

In this case, the output should be something like the following.

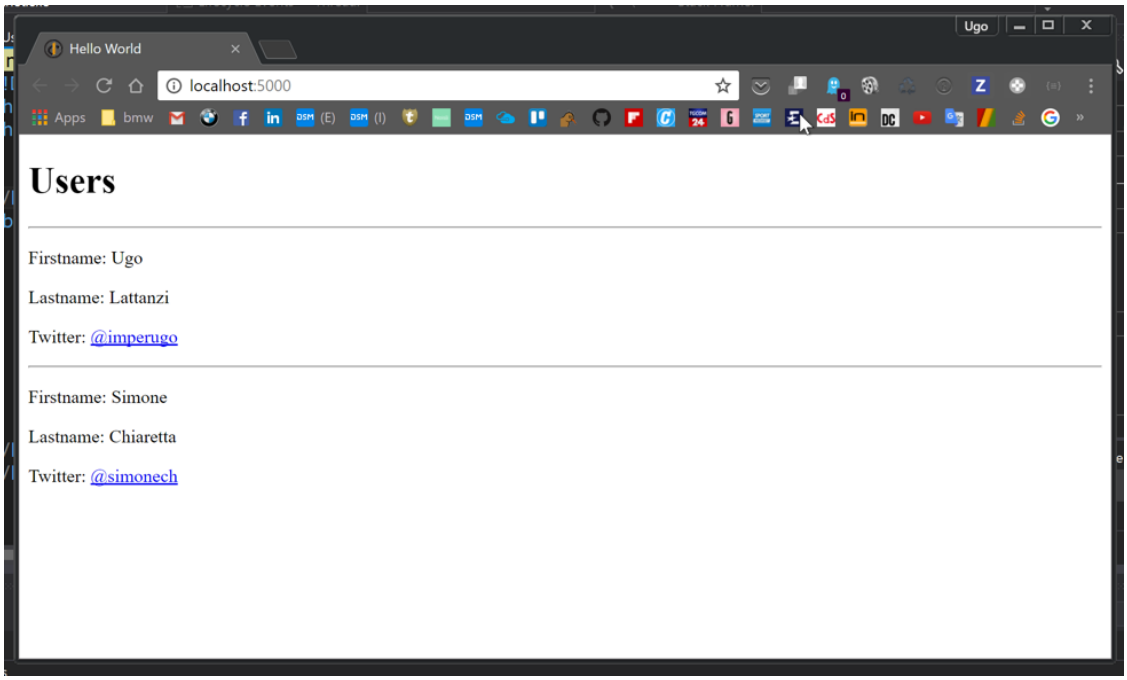

Figure 5-11: Server-Side Page Rendered Using MVC for a List

Routing

You just saw a small introduction to ASP.NET MVC. Notice that **`HomeController`** combined with the **`Index`** action handle the root domain (`http://localhost:5000` in our case). But why?

To get the answer, you have to go back to the MVC configuration where we wrote this code:

Code Listing 5-15

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.Mvc
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironmen
t env, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();

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

            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The method **`app.UseMvcWithDefaultRoute()`** defines the default route; it is equivalent to writing the following:

Code Listing 5-16

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

This part of code says to split the URL into segments, where the first is the controller, the second is the action, and the third (optional) is the parameter. If they are all missing, use the action **`Index`** of **`HomeController`**.

In fact, if you run the application again and write this URL:

`http://localhost:5000/home/index`

The output should be the same.

If you want to manage a URL like `http://localhost:5000/products`, create a **`ProductController`** with an **`Index`** action.

View specific features

After seeing the general features of ASP.NET Core MVC, it is time to see two new features that are specific to the view side of ASP.NET MVC applications:

- Tag Helpers
- View components

Tag Helpers are a new way of exposing server-side code that renders HTML elements. They bring the same features of HTML Razor helpers to the easier-to-use syntax of standard HTML elements.

View components can be seen as either a more powerful version of partial views or a less convoluted way to develop child actions. Let's look in detail at both of these new features.

Tag Helpers

Compared to HTML Razor helpers, Tag Helpers look like standard HTML elements—no more switching context between HTML and Razor syntax.

Let's see an example to make things a bit clearer. If you want to make an editing form with ASP.NET MVC, you have to display a text box that takes the value from the view model, renders validation results, and so on.

Using the previous HTML Razor helpers, you would have written:

```
@Html.TextBoxFor(m=>m.FirstName)
```

But now with Tag Helpers, you can directly write **`<input asp-for="FirstName" />`** without introducing the Razor syntax. Notice that it's just a normal HTML **`<input>`** tag enhanced with the special attribute **`asp-for`**.

It doesn't look like such a big change, but the advantage becomes clear when you add more attributes. One example of this is the **`class`** attribute.

This is how you add the `class` using the old HTML Razor syntax:

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

Using the Tag Helper, it's just this:

`<input asp-for="FirstName" class="form-control" />`.

Basically, you write the tag like you were writing a static HTML tag, with the addition of the special **`asp-for`**.

But this new syntax retains the support of Visual Studio IntelliSense. As soon as you start typing an HTML element that is somehow enhanced via Tag Helpers, you see in the IntelliSense menu that the tag is represented with an icon that is different from normal HTML tags.

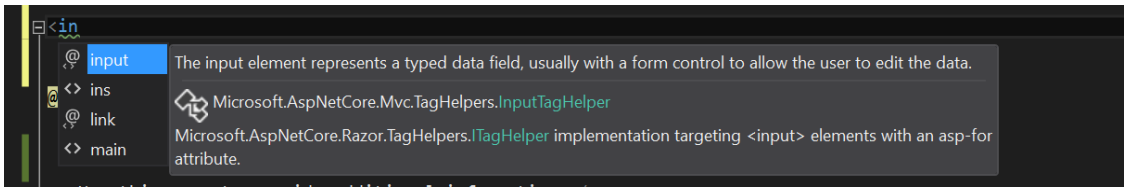

Figure 5-12: Visual Studio IntelliSense for HTML Tags

If you then trigger the IntelliSense menu to see all the available attributes, only one has the same icon again, the **`asp-for`** attribute. Once you add this attribute and open the IntelliSense menu again, you'll see all methods and properties of the page model.

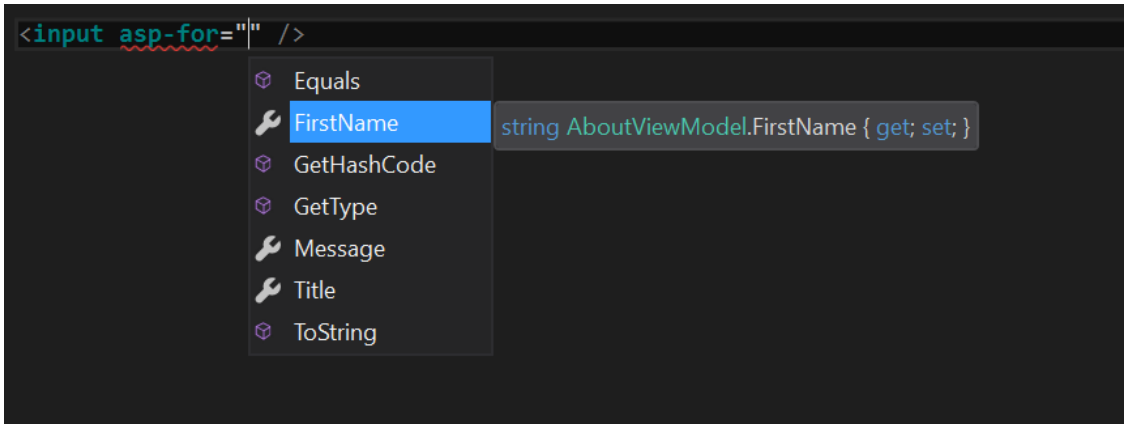

Figure 5-13: Visual Studio IntelliSense for MVC Model

Look at the two previous screenshots carefully. You'll notice that the **`<input>`** tag has changed color. The following picture highlights the difference better.

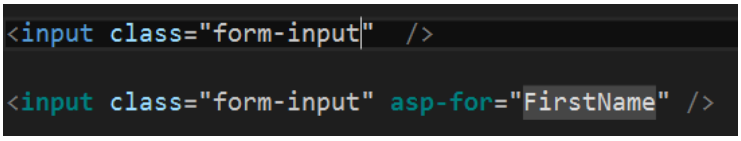

Figure 5-14: Difference Between the Two Tags

In the first line, the element is blue with attributes in blue. While in the second, the element is green with the special attribute also in green (the normal attribute is still blue). Visual Studio recognizes both normal HTML tags and attributes, and Tag Helpers. This avoids confusion when editing a view.

ASP.NET Core MVC comes with a lot of Tag Helpers, most of them are just reimplementing the same HTML Razor helpers used to edit forms, like the **`input`**, **`form`**, **`label`**, and **`select`** elements. But there are other Tag Helpers used to manage cache, to render different HTML elements based on the environment, and to manage script fallback or CSS files. You can see many of these tag helpers used in the `View\Shared\_Layout.cshtml` file in the default project template. A reduced version of the file is available in the following code listing.

Code Listing 5-17

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - MvcSample</title>

    <environment names="Development">
        <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css
"
              asp-fallback-
href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-
property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-
version="true" />
    </environment>
</head>
<body>

// etc.

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
```

```
    <environment names="Staging,Production">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-
2.2.0.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery">
        </script>
        <script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("scripts", required: false)
</body>
</html>
```

Building custom tag helpers

Tag Helpers are easy to create, so it's worth seeing how to create your own. Writing a custom Tag Helper is especially useful when you want to output an HTML snippet that is long and repetitive, but changes very little from one instance to another, or when you want to somehow modify the content of an element.

As an example, you are going to create a Tag Helper that automatically creates a link by just specifying the URL.

Something that takes this:

```
<url>https://www.syncfusion.com/resources/techportal/ebooks</url>
```

And creates a working a tag, like this:

```
 <a
href="https://www.syncfusion.com/resources/techportal/ebooks">https://www.syn
cfusion.com/resources/techportal/ebooks</a>.
```

Start by creating a UrlTagHelper file inside the MVC project. A good convention is to put the file inside a TagHelpers folder.

A Tag Helper is a class that inherits from **`TagHelper`** and defines its behavior by overriding the method **`Process`** or its asynchronous counterpart **`ProcessAsync`**. These methods have two arguments:

- **`context`**, which contains information on the current execution context (even if it is rarely used).

- **output**, which contains a model of the original HTML tag and its content, and is the object that has to be modified by the Tag Helper.

To use a Tag Helper in the views, you have to tell both Visual Studio and the .NET Core framework where to find them. This is done by adding a reference in the _ViewImports.cshtml file.

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

To make sure all the basic steps are done, copy the following code into the UrlTagHelper file.

Code Listing 5-18

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace MvcSample.TagHelpers
{
    public class UrlTagHelper: TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperO
utput output)
        {
            output.TagName = "a";
        }
    }
}
```

This Tag Helper at the moment is useless. It only replaced the tag used when calling the helper from whatever it was (a `URL` in our example) to a tag, but it doesn't create an **`href`** attribute pointing to the specified URL. To do so, you have to read the content of the element and create a new attribute. Since the content could also be a Razor expression, the method to do so is an Async method, **`output.GetChildContentAsync()`**. You also have to change the method you implemented from **`Process`** to **`ProcessAsync`**.

The complete Tag Helper is shown in the following code listing.

Code Listing 5-19

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace MvcSample.TagHelpers
{
    public class UrlTagHelper: TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
```

```
        {
            output.TagName = "a";
            var content = await output.GetChildContentAsync();
            output.Attributes.SetAttribute("href", content.GetContent());
        }
    }
}
```

Tag Helpers can also have attributes. You can extend the URL Tag Helper to specify the target of the link. To add an attribute to a Tag Helper, add a property to the class. In the case of the target, you just need to add **`public string Target { get; set; }`** to the class. But having a string parameter is not a nice experience because IntelliSense doesn't show which values are allowed. You can define the property as an enum whose values are only the ones allowed for the **`target`** HTML attribute. Then you have nice IntelliSense.

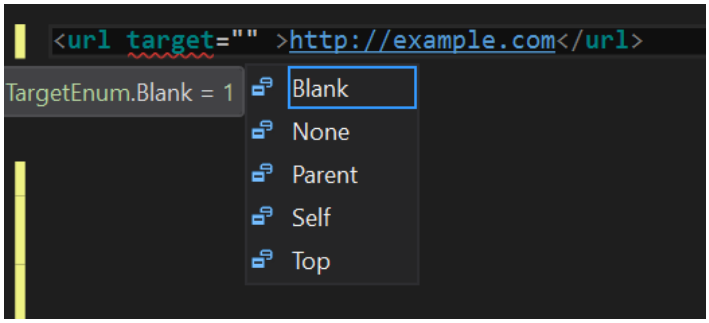

Figure 5-15: HTML Property Suggestion

This is the code of the updated Tag Helper:

Code Listing 5-20

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace MvcSample.TagHelpers
{

    public enum TargetEnum
    {
        None = 0,
        Blank,
        Parent,
        Self,
        Top
    }

    public class UrlTagHelper: TagHelper
    {
```

```
        public TargetEnum Target { get; set; }
        public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
        {
            output.TagName = "a";
            var content = await output.GetChildContentAsync();
            output.Attributes.SetAttribute("href", content.GetContent());
            if (Target!=TargetEnum.None)
            {
                output.Attributes.SetAttribute("target", "_"+Target.ToStr
ing().ToLowerInvariant());
            }

        }
    }
}
```

Now, **<url target="Blank">http://example.com</url>** is rendered as **http://example.com**.

View components

View components are the next new, view-related feature. They are in a way similar to partial views, but they are much more powerful and used to solve different problems.

A partial view is, as the name implies, a view. It is typically used to simplify complex views by splitting them into reusable parts. Partial views have access to the view model of the parent page and don't have complex logic.

On the other hand, view components don't have access to the page model; they only operate on the arguments that are passed to them, and they are composed by both view and class with the logic.

If you used a child action in previous versions of ASP.NET MVC, they more or less solved the same problem in a more elegant way, as their execution didn’t go through the whole ASP.NET MVC execution pipeline starting from the routing.

Typically, view controllers are used to render reusable pieces of pages that also include logic that might involve hitting a database—for example sidebars, menus, conditional login panels, etc.

How to write a view component

As mentioned, a view component is made of two parts. The class containing the logic extends the **`ViewComponent`** class and must implement either the **`Invoke`** or the **`InvokeAsync`** method. This returns **`IViewComponentResult`** with the model that has to be passed to the view. Conventionally, all view components are located in a folder named **`ViewComponents`** in the root of the project.

The view is just like any other view. It receives the model passed by the component class that is accessed via the `@Model` variable. The view for a view component has to be saved in the `Views\Shared\Components\<component-name>\Default.cshtml` file.

As an example, let's build a view component that shows the sidebar of a blog. The component class just calls an external repository to fetch the list of links.

Code Listing 5-21

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using MvcSample.Services;

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

        public async Task<IViewComponentResult> InvokeAsync(int max=10)
        {
            var items = await db.GetLinks().Take(max);
            return View(items);
        }
    }
}
```

Notice that it receives the dependency in the constructor, as shown for controllers in ASP.NET Core MVC. In this case, since the operation of retrieving the links goes to a database, you implement the Async version of the component.

The next step is implementing the view. Nothing special to mention here, just a simple view that renders a list of links.

Code Listing 5-22

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

<h3>Blog Roll</h3>
<ul>
    @foreach (var link in Model)
    {
        <li><a href="@link.Url">@link.Title</a></li>
    }
</ul>
```

The important fact to remember is where this view is located. Following convention, it must be saved as Views\Shared\Components\Sidebar\Default.cshtml.

And now it's time to include the component into a view. This is done by simply calling the view component using the Razor syntax.

Code Listing 5-23

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

This is a bit convoluted, especially the need to create an anonymous class just for passing the arguments. But there is also another way of calling a view component as if it were a tag helper. Starting from ASP.NET Core 1.1, all view components are also registered as Tag Helpers with the prefix **vc**.

Code Listing 5-24

```
<vc:side-bar max="5"></vc:side-bar>
```

Apart from being easier to write, this also implements IntelliSense in the HTML editor.

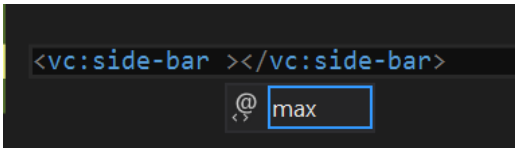

Figure 5-16: IntelliSense for Tag Helpers

Conclusion

ASP.NET Core comes with a rewritten MVC framework, improved from the previous version and aimed at being the unified programming model for any kind of web-based interaction.

You've seen the cool new features introduced for simplifying the development of views. This is the last chapter that explains coding. The next two chapters are more about tooling, showing how to deploy apps and how to develop on Mac without Visual Studio.

Chapter 6 How to Deploy ASP.NET Core Apps

You've built your application with simple ASP.NET Core or by using ASP.NET Core MVC for more complex sites, and the time has come to show it to the world. A few years ago, we were mostly interested in deploying to internal servers, but with the rise of cloud computing, now it's just as likely that we deploy to Azure.

Nevertheless, in this chapter we will discuss both experiences for local servers and for Azure.

Deploying on IIS

ASP.NET Core, being based on OWIN, doesn't depend on the server, but it could in theory run on top of any server that supports it.

Normally, during development, an ASP.NET Core application runs hosted by a local web server, called **Kestrel**, that is configured within the Program.cs file that starts the application. Basically, each ASP.NET Core application is self-contained and could run without external web servers. But while being optimized for performance and being super-fast, Kestrel misses all the management options of a full-fledged web server like IIS.

Being self-contained applications (and not just DLLs) requires a different approach for hosting them inside IIS. Now, IIS is *just* a simple proxy that receives the requests from the client and forwards them to the ASP.NET Core application on Kestrel. It then sits and waits for processing to be completed and finally returns the response back to the originator of the request.

In order to host ASP.NET Core applications in IIS, you need to install a specific module that does the reverse proxy work and makes sure the application is running. This module is called **`AspNetCoreModule`** and can be installed from the ASP.NET Core server hosting bundle.

If you want to try running ASP.NET Core via IIS on a development machine, there is nothing to do because the module is installed as part of the SDK. If, on the other hand, you want to try it on a real server, the module has to be installed. Figure 6-1 shows the module listed inside the IIS manager application.

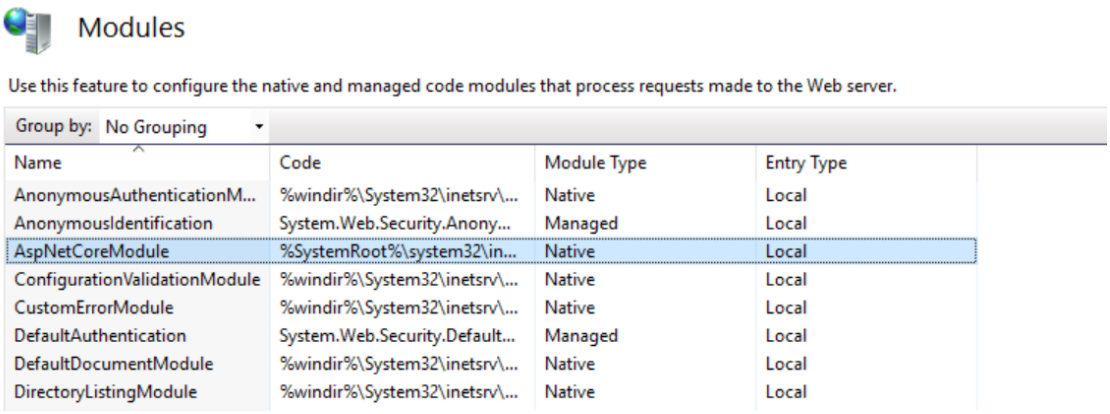

Figure 6-1: IIS Modules

With the module installed, the next step is creating a website. Since the module will just act as a proxy without running any .NET code, the website needs an application pool configured to run without any CLR, so the option **No Managed Code** must be selected, as in Figure 6-2.

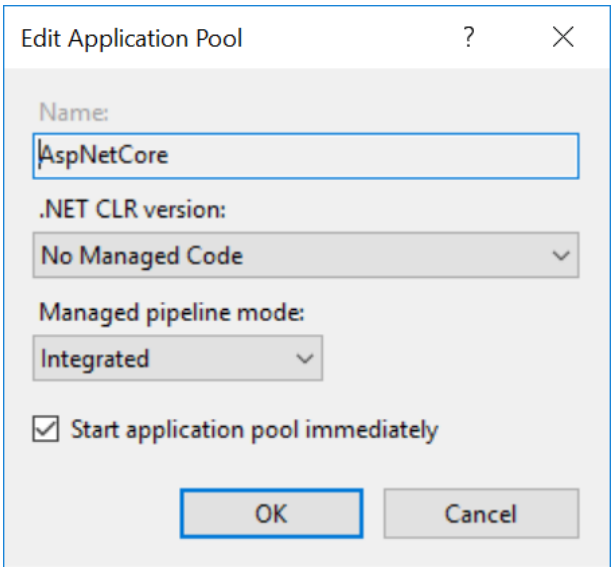

Figure 6-2: IIS Application Pool Basic Configuration

The configuration of the **`AspNetCoreModule`** comes from the `web.config` file in the root of the folder.

Code Listing 6-1

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*"
        modules="AspNetCoreModule"
        resourceType="Unspecified"/>
```

```
    </handlers>
    <aspNetCore
        processPath="dotnet"
        arguments=".\WebApp.dll"
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\stdout"
        forwardWindowsAuthToken="false"/>
  </system.webServer>
</configuration>
```

Publishing the application is easy using the **dotnet** command-line tool. Just use the **publish** command to build and generate a self-contained folder that can be easily copied to the location where the website is. Normally, the published application is saved to the `./bin/[configuration]/[framework]/publish` folder, but the target location can also be changed with a command-line option.

Code Listing 6-2

```
dotnet publish
      --framework netcoreapp1.0
      --output "c:\temp\PublishFolder"
      --configuration Release
```

The publish operation also executes the **postpublish** scripts, so if the project has been created with the default project template, the **publish-iis** command is called, which automatically updates the **web.config** file with the right values for the project.

Now, all that's left is copying the folder to where the website is configured. The ASP.NET Core site is running on IIS.

One thing that is missing in the **dotnet publish** command is the possibility to publish to a remote server and to do incremental updates. Another publish method is the **Publish Dialog** in Visual Studio.

Typically, a system administrator will provide a publishing profile that can be imported into the Publish Dialog. When publishing, the application will be built and then pushed to the remote server via Web Deploy.

Deploying on Azure

Instead of deploying on premises, you could use the cloud. And when we talk about the cloud, we cannot forget to mention Microsoft Azure, which is absolutely one of the most important cloud computing platforms available right now. It offers tons of services supporting almost all possible user needs.

Whether our application needs to scale or not, Microsoft Azure could be a good solution to reduce the friction of server administration, configuration, hardware problems, backup, and so on.

If you prefer to use another cloud hosting service, like Amazon Web Services (AWS) or Google Cloud, you can, of course, but usually they offer virtual machines. Deploying there is similar to doing it on a remote IIS that you manage.

Azure is different because it offers a service called App Service, which breaks down the barriers of configurations.

Deploy to Azure from Visual Studio with Web Deploy

Deploying to Azure from Visual Studio is really easy, especially if all resources have already been created on Azure. If that is the case, follow these steps:

Click **Publish**.

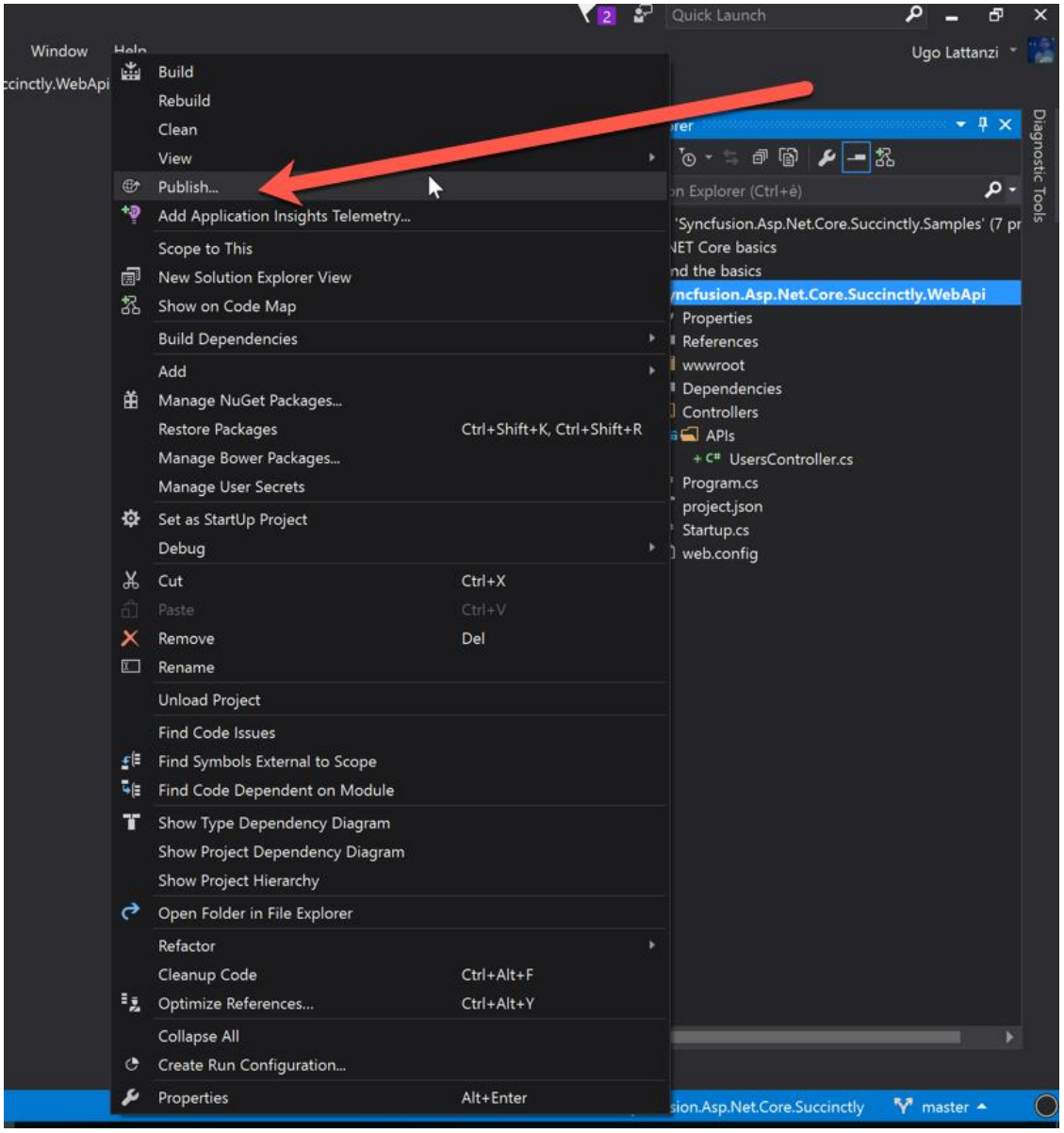

Figure 6-3: Publish Web Application Using Visual Studio

Connect to Azure.

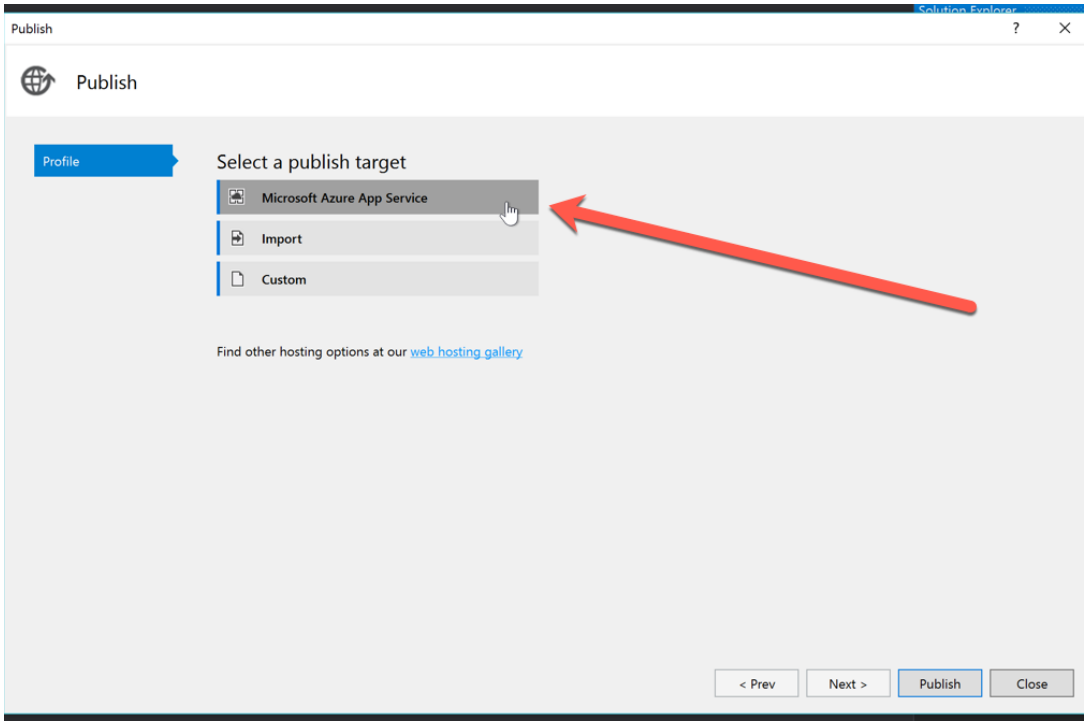

Figure 6-4: Select Publishing Targets

Log in if needed.

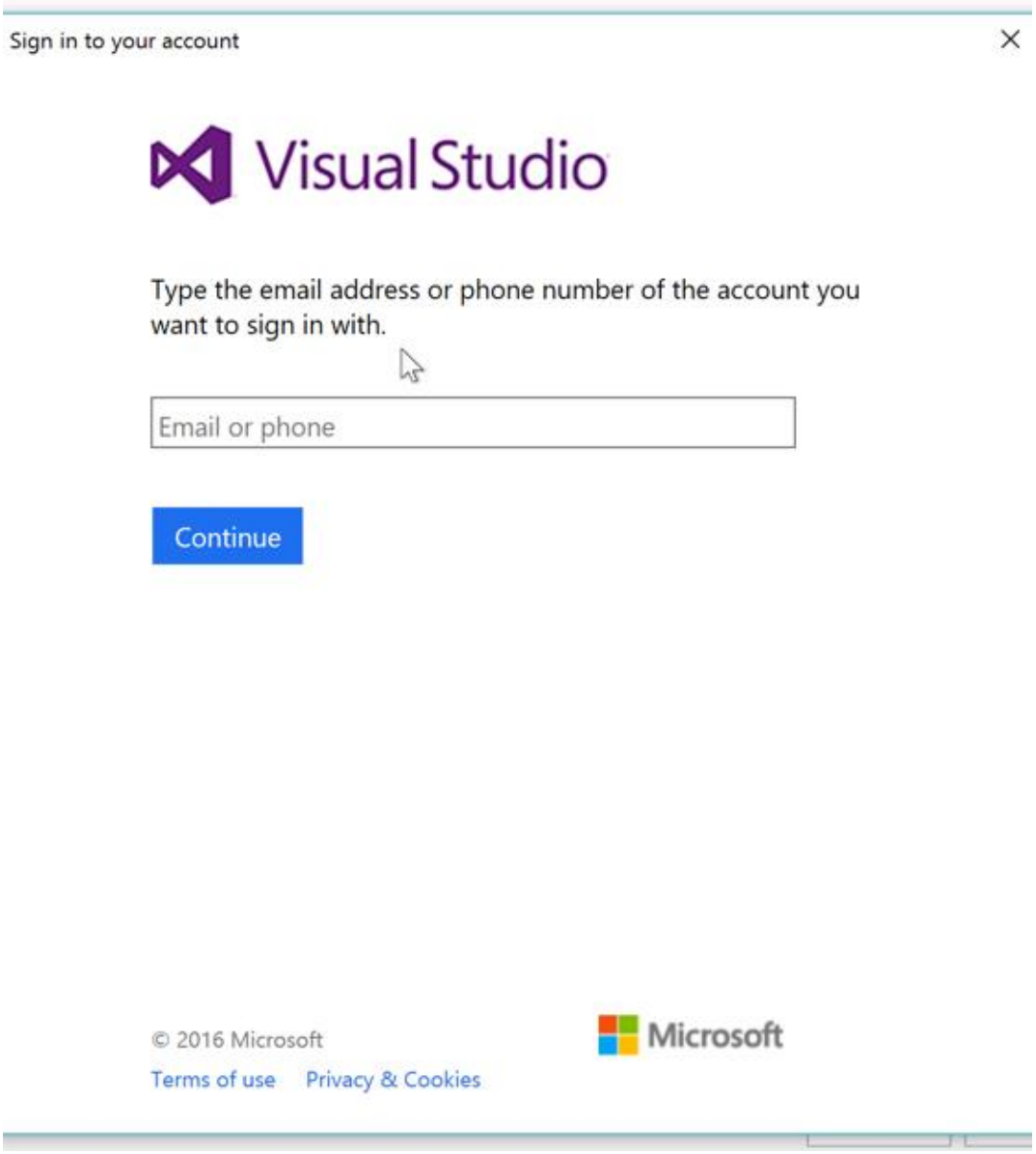

Figure 6-5: Azure Credentials Input

Select the App Service where you want to deploy.

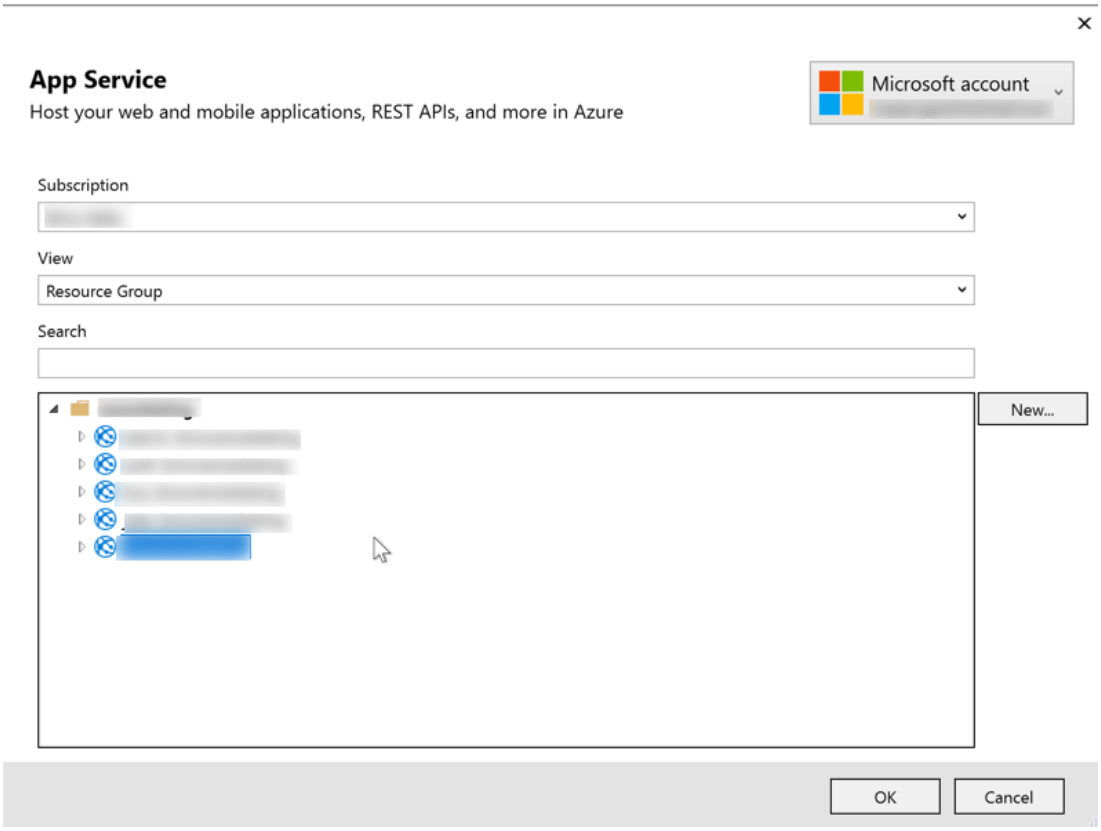

Figure 6-6: Azure App Service Selection

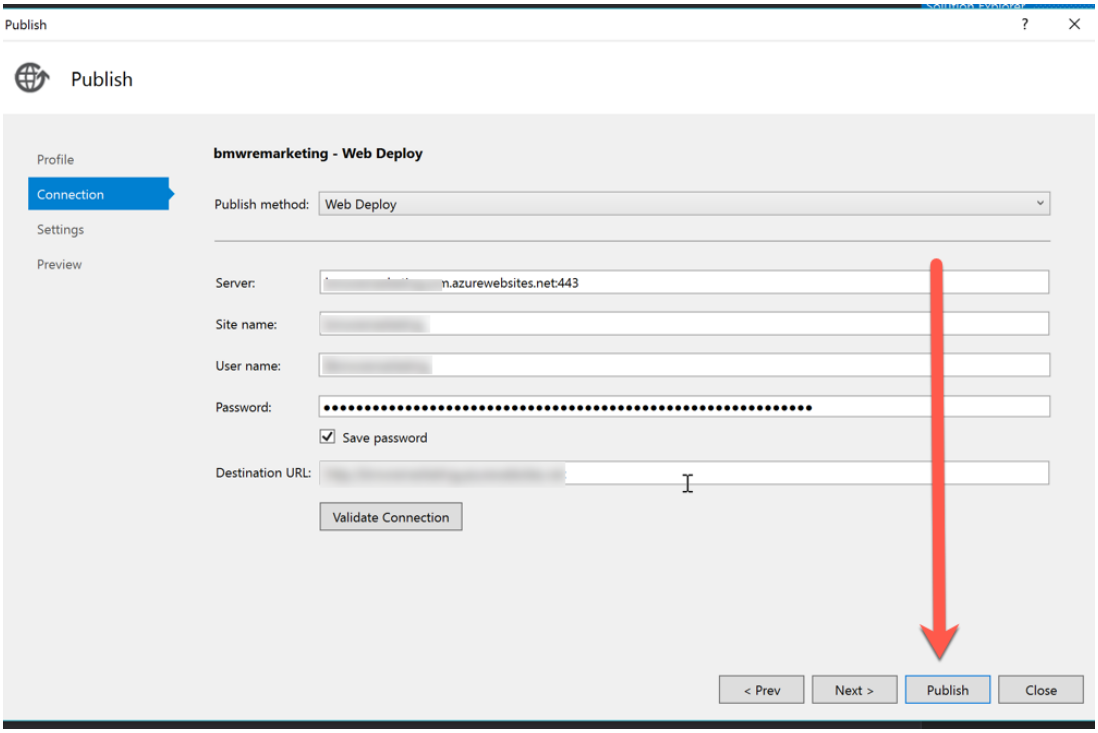

Figure 6-7: Publishing

At the end of the publishing procedure, Visual Studio will open the website with a browser, and, if everything is fine, you should see your application running on Azure.

Conclusion

In this chapter, we discussed how easy it is to deploy ASP.NET Core applications on IIS on premises and how it's even easier to do it on Azure. No wonder everyone is now moving to the cloud.

Now that you've built an application with Visual Studio and deployed it for the world to see, the next chapter goes more in depth on how to build apps without Visual Studio.

Chapter 7 Tools Used to Develop ASP.NET Core Apps

At the beginning of the book, we said that ASP.NET Core is a cross-platform framework, but we are almost at the end of the book, and apart from a simple example in Chapter 1, we've always used Visual Studio on Windows.

This last chapter closes the gap between platforms by showing how ASP.NET Core applications can be developed without Visual Studio.

Using dotnet CLI

Throughout this book, you have used the dotnet command line tool to create projects, to build, and to publish applications. In this chapter, you will go more in depth and see more advanced usages of the tool.

The dotnet tool is called by first specifying the command, followed by the arguments specific to the command, and finally the options.

The commands available from the dotnet tool are:

- **`new`**, to create a new .NET Core project.
- **`restore`**, to download all dependencies from NuGet.
- **`build`**, to compile the projects.
- **`publish`**, to generate a self-contained folder used for deployment.
- **`run`**, to run a project, building it if it's not already built.
- **`pack`**, to package the project as a NuGet package.
- **`msbuild`**, used as a proxy for the standard MSBuild command.

Let's start by looking in greater detail at the **`new`** command. When called without any option, it just lists all the possible types of projects it can create, but by specifying an argument, it directly creates a project. Using **`console`**, it creates a C# console application; with **`mvc`**, it makes a full-featured ASP.NET Core application with MVC; or with **`classlib`**, it creates a library project.

For both **`build`** and **`publish`**, you can specify the framework, the runtimes, and the configuration (debug or release) you want for compiling. These options are not very useful if you built your application only to target one framework or runtime. However, they are meaningful if, for example, you want to create a utility that can run on multiple machines as a native application.

If you want to build a sample console app for Windows and for Mac, you have to first specify in the project file that you want to support additional runtimes.

.NET Core supports a lot of different systems, but each of them has a very specific identifier, which is in the format `[os].[version]-[arch]` (e.g, `osx.10.11-x64`). But despite the templated look, they are unique strings, so before using a new RID, you have to make sure it's supported and has the right name. Refer to the RID Catalog on Microsoft's docs site.

The runtimes and application support have to be specified inside the **`RuntimeIdentifiers`** property of the project file as a list of RIDs separated by semicolons (`;`). Leave no space after the semicolon or the restore will fail.

Code Listing 7-1

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp1.0</TargetFramework>
  <RuntimeIdentifiers>win10-x64;osx.10.11-x64;ubuntu.14.04-x64</RuntimeId
entifiers>
</PropertyGroup>
```

Once the file has been modified, run `dotnet restore` again. This will download all packages—for other runtimes too—as shown in the following command prompt window.

Figure 7-1: Command Prompt Output

You can now publish an app for all three runtimes, thus creating self-contained folders that contain native binary executables that can be copied directly to the target machine.

To publish them, run the **`dotnet publish`** command three times, one per runtime you want to build.

Code Listing 7-2

```
dotnet publish -r win10-x64
dotnet publish -r osx.10.11-x64
dotnet publish -r ubuntu.14.04-x64
```

These commands create three folders under the bin folder of the project, one per runtime.

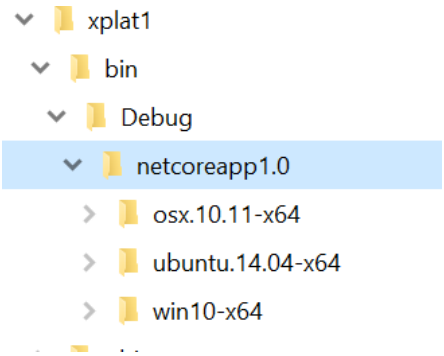

Figure 7-2: Folder Output

The dotnet CLI is extensible, so although it has very basic features now, it can be expanded with external libraries like the **`iis-publish`** tool seen in Chapter 4. Expect its features to grow as time passes.

Developing ASP.NET Core using Visual Studio Code

With the CLI, you can create a new project first and compile and publish it later, but the CLI doesn't help you develop the application. If you don't want to use full-fledged Visual Studio 2015 or 2017, you can use any modern text editor for which an OmniSharp plugin is available.

OmniSharp

OmniSharp is a set of open-source projects working together to bring .NET development to any text editor. It's made from a base layer that runs Roslyn and analyzes project files. This layer builds a model of the project that can be queried via APIs (REST over HTTP or via pipes) from text-editor extensions to display IntelliSense, autocomplete, suggestions, and code navigation. At the moment, there are OmniSharp extensions for five popular text editors: Atom, Sublime Text, Vim, Emacs, and Visual Studio Code, the new cross-platform, open-source text editor developed by Microsoft.

In this chapter, we are going to show how to develop a simple ASP.NET Core application using Visual Studio Code and the OmniSharp extension.

Setting up Visual Studio Code

Installing Visual Studio Code is easy: Just go to code.visualstudio.com and download the version for your operating system: Windows, Mac, or Linux.

Visual Studio Code is a general-purpose editor that relies on extensions to support specific languages. To develop and debug ASP.NET Core applications, install the C# extension by clicking on the extension pane in Visual Studio Code and then typing **`@recommended`** in the search bar.

Once the extension is installed, you can open an ASP.NET Core project created using the **`dotnet new -t web`** command by selecting its folder. Visual Studio Code understands it's a .NET Core project and will show a warning like the one in the following figure.

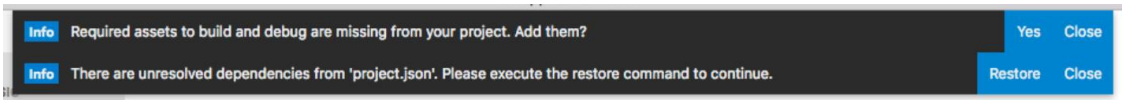

Figure 7-3: VS Code Warnings

The first warning says that some configuration files are missing. Once you click **Yes**, two new files will be added to **`.vscode folder`**: **`launch.json`** and **`tasks.json`**. These two files tell Visual Studio Code how to compile a .NET Core project and how to launch a debugging session. But don't worry too much about them, as all the correct values are added by the C# extension.

Developing with Visual Studio Code

Now that all is ready, developing an ASP.NET Core application with Visual Studio Code is just as productive as doing it with the full version of Visual Studio 2017, if not more.

For example, you have IntelliSense and code completion.

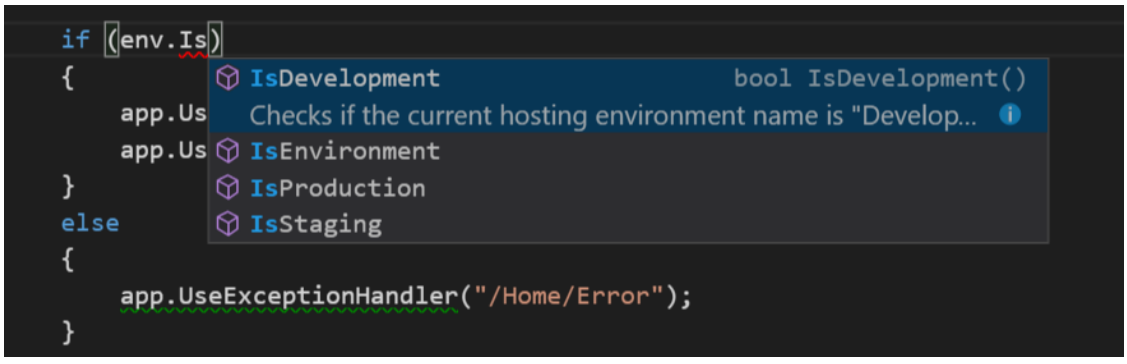

Figure 7-4: VS Code IntelliSense

You also have the same linting and refactoring suggestions. They appear as squiggle underlines (red or green) with messages in the bottom panels and icons in the status bar. Figure 7-5 shows all the locations in the UI where suggestions appear.

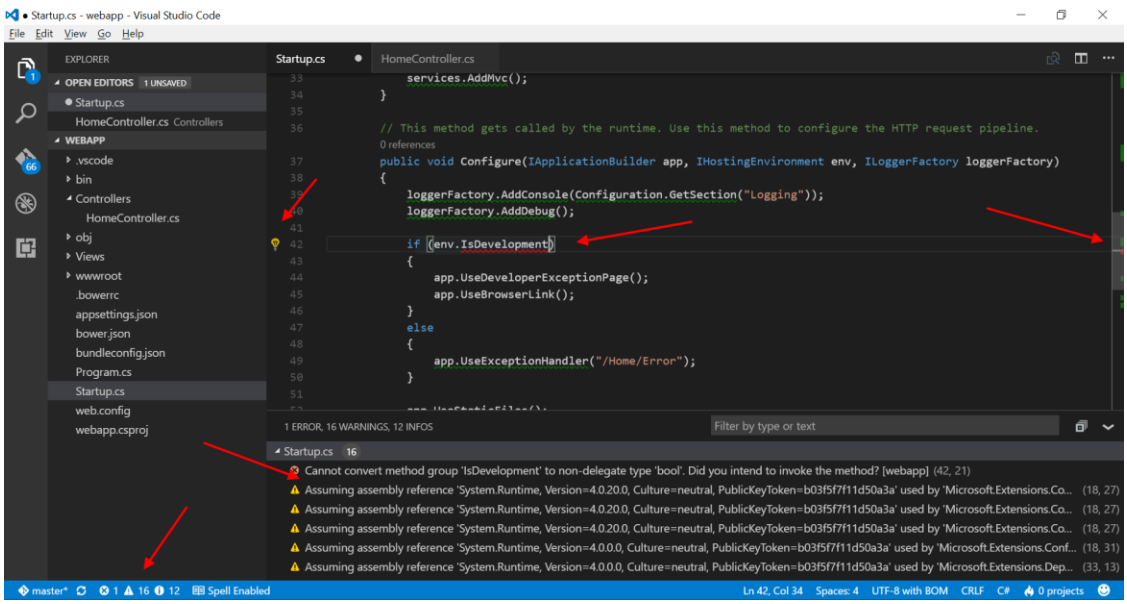

Figure 7-5: VS Code Suggestions

Another example of a good Visual Studio Code feature is code navigation. Like any other editor, you can go to the definition of a variable and peek at it without leaving the current file.

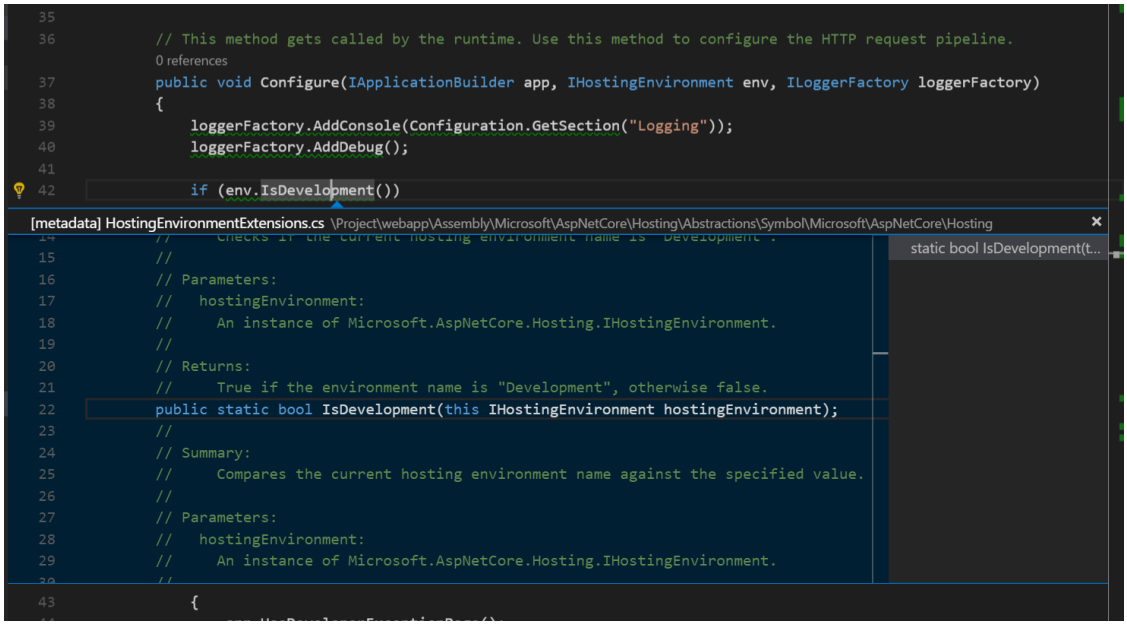

Figure 7-6: VS Code Navigation

Debugging with Visual Studio Code

Once the application has been developed, it's time to make sure it works. From within Visual Studio Code, you can launch the application, set breakpoints, and inspect variables.

To do so, go to the **Debug** panel (by clicking the *bug* icon), and click on the *debug* icon (the green *run* icon, same as in Visual Studio) to launch the application. To set a breakpoint, click next to the line number in the editor. Then you can step through the instructions like any other code debugger.

The figure below shows the debugging interface of Visual Studio Code while debugging the **`HomeController`**.

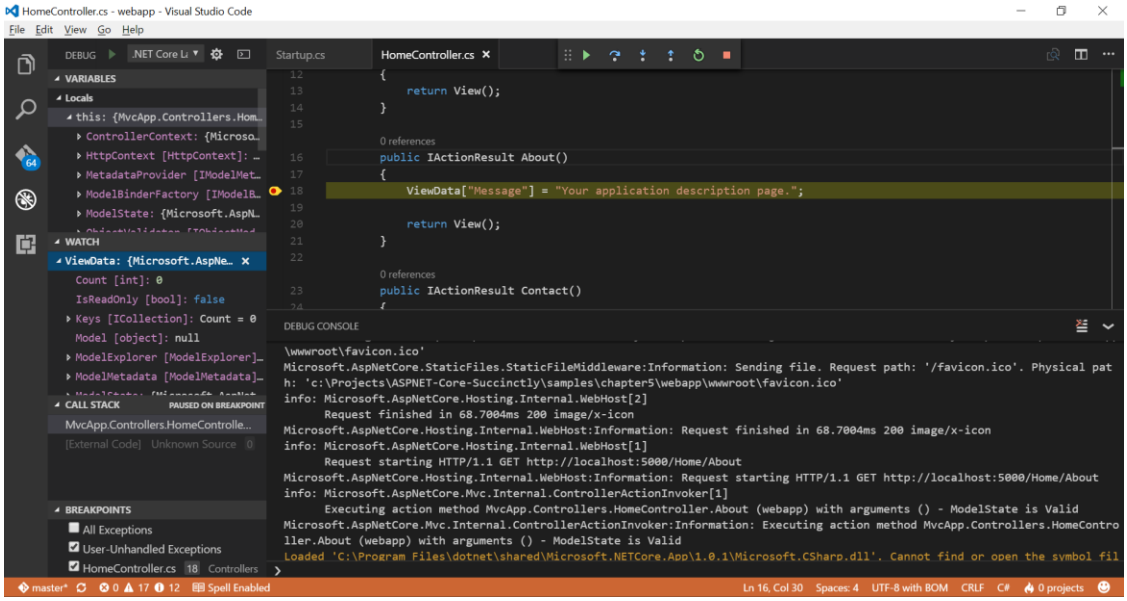

Figure 7-7: VS Code Debugging

You can basically do everything you can normally do with the full version of Visual Studio, but in a lighter way and on all operating systems.

Conclusion

In this chapter, you saw that you do not need a Windows machine with Visual Studio to develop ASP.NET Core applications. You can use a Mac and develop using the CLI and Visual Studio Code.

In addition to what we've shown in this chapter, Visual Studio Code can do many more things. It is a Git client, a Node.js and client-side JavaScript editor and debugger, and it is also a very good Markdown editor. In fact, this whole book has been written in Markdown within Visual Studio Code, and later converted to Word format using Pandoc.

A Look at the Future

The release of .NET Core has been a very difficult one, with many delays and changes in direction. While the framework is stable, with .NET Core 1.1 released in November 2016, the tooling is still in development.

When we wrote this book, we relied on previews that were made available at the Connect(); event of November 2016, so some of the screenshots or procedures in the examples might be different from what is currently available.

What can we expect for the future of .NET Core?

First, unlike previous frameworks, we have to expect a continuous release of enhanced tools, both for the CLI and for the tooling inside Visual Studio. Second, on the framework side, there will be the second big release of .NET Core 2.0, implementing .NET Standard 2.0.

But rest assured that Microsoft sees .NET Core as the future of .NET for the next 10 years, so this is the right time to jump in and start learning this new technology.