ASP.NET CORE 3.1

SUCCINCTLY

BY **SIMONE CHIARETTA**
AND **UGO LATTANZI**

SUCCINCTLY EBOOK SERIES

Syncfusion®

ASP.NET Core 3.1 Succinctly

By
Simone Chiaretta and Ugo Lattanzi

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-205-8

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, VP of content, 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. He has authored several books and articles, and has coauthored *OWIN Succinctly* and the previous editions of this book with Simone Chiaretta, both of which were published by Syncfusion.

When he's not working, he loves to spend time with his children, Tommaso and Bianca, and his wife, Eleonora. He also enjoys snowboarding and cycling.

Simone Chiaretta

Simone Chiaretta is a web architect and developer who enjoys sharing his development experiences—including almost 25 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*; *What's New in ASP.NET MVC 2*; and *Front-end Development with ASP.NET Core, Angular, and Bootstrap*; all published by Wrox. He has also coauthored *OWIN Succinctly* and the first two editions of this book with Ugo Lattanzi, all of which were published by Syncfusion.

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

When he's not writing code or blog posts, or taking part in the worldwide .NET community, he is either training for or participating in Ironman triathlons, or lately, taking care of his newborn Marco.

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

About ASP.NET Core 3.1 Succinctly

There are only a few years that one can consider as landmarks of a profession, and 2016 was one for software development. That is when Microsoft released .NET Core, its third major development library and SDK. But this time, it was not just for Windows, and it was not a closed-source product. It was an open-source SDK that allowed 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, which has gone through some improvements in the meantime. This book guides readers through the foundations of the library, covers its basic features, and covers the new version of the web application frameworks ASP.NET MVC, Web API, and Blazor. At its end, this book shows 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 3.1. When showing how to develop with IDEs, it uses Visual Studio 2019 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 with .NET Core. In addition to offering new features, it also adopts a significantly new approach to web development.

The first chapters go through the history of Microsoft's web stack to show the motivations that led to this framework. Later, the book 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 3.1 is a modular, cross-platform, cloud-optimized version of the .NET Framework, consisting of the CoreCLR and the implementation of the .NET Standard Library 2.1. Two of the main features of this library are the ability to install only the features that are needed for the application you are building to reduce its footprint, and the possibility of installing the library itself within the application. This makes it possible for applications built with different versions to coexist 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 to be cross-platform, completely open source, and without the limitations of backward compatibility. Like .NET Core, ASP.NET Core is 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.

ASP.NET Core comes with many interesting features that we are going to see throughout this book, like an integrated dependency injection system and a new application framework that unifies the programming models of ASP.NET MVC and Web API.

.NET Standard

The .NET Standard won't be covered in detail in this book, but since it has been mentioned (and you have probably heard about it), let's define what it is.

.NET Standard is a specification that defines the set of APIs a .NET platform must implement. Its aim is to simplify the sharing of code and libraries among different implementations of .NET. The latest version of these specifications, .NET Standard 2.1, includes all the APIs available in the .NET Framework 4.6.1 and more.

.NET Standard 2.1 is supported by:

- .NET Core 3.0 and later
- Mono 6.4

- Xamarin.iOS 12.16
- Xamarin.Mac 5.16
- Xamarin.Android 10
- Any future version of UWP and Unity

This means that libraries developed for .NET Standard 2.1 can be used in any of those platforms. It's a big advantage for library developers, since they no longer have to develop different libraries to target different platforms.

For you as a developer, the main takeaway is that .NET Core 2.1 has feature parity with the "full" .NET Framework 4.6.1, and this makes it possible to port older applications to .NET Core.

There is a caveat: notice that the list doesn’t include any version of the .NET Framework, since the latest version of the .NET Standard it supports is 2.0.

According to the new strategy by Microsoft, the way forward is .NET Core, and the .NET Framework, whose latest version will be 4.8, will be only developed in maintenance mode, with no new features coming, and only updated for security and compatibility.

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 ASP.NET Core came to be, it's important to see what came before, and how the web development world has changed during the last 20 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 web server. The framework also provided other services to implement basic state management, caching, and other lower-level concerns.

The second group comprised a large number of WinForms application developers who were forced by changes in the market to approach 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 integrated development environment (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 with a feature-rich, 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 for several reasons:

- 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.
- 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, Microsoft released a new web framework called ASP.NET MVC in 2009. 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 completely 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.

Although ASP.NET MVC solved the problem of the slow release cycle and removing the HTML markup abstraction, it still suffered from a dependency on the .NET Framework and **`System.Web`**, which kept 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 webpages 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 under different web servers.

OWIN and Katana

With all these modular frameworks spreading around, there was the risk that developers would have 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. OWIN defines the components of a modern web application and how those components interact.

Microsoft released an OWIN-compliant web server in 2013, 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 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 led 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:

- You couldn't run .NET applications on a non-Windows system.
- The dependency on the full framework made .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 quickly.
- The complexity of the .NET project system prevented the development of .NET apps outside of Visual Studio.

It became clear that the best 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.

The future of .NET

With the release of the first pre-releases of .NET Core 3 and .NET Framework 4.8 in 2018, Microsoft announced that the future of the .NET platform will be only in .NET Core. This means that version 4.8 is going to be the last major version of the .NET Framework, and after that, it will only receive security and compatibility updates.

That’s why .NET Core 3.x also includes all the libraries for developing modern, desktop-based applications with WPF and Windows Form.

After 3.1, the next version of .NET Core will be called simply .NET 5 (skipping version 4 to avoid any confusion with the .NET Framework).

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 build a simple application.

Installing .NET Core on Windows

Installing .NET Core on Windows is pretty easy. With Visual Studio 2019, 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.

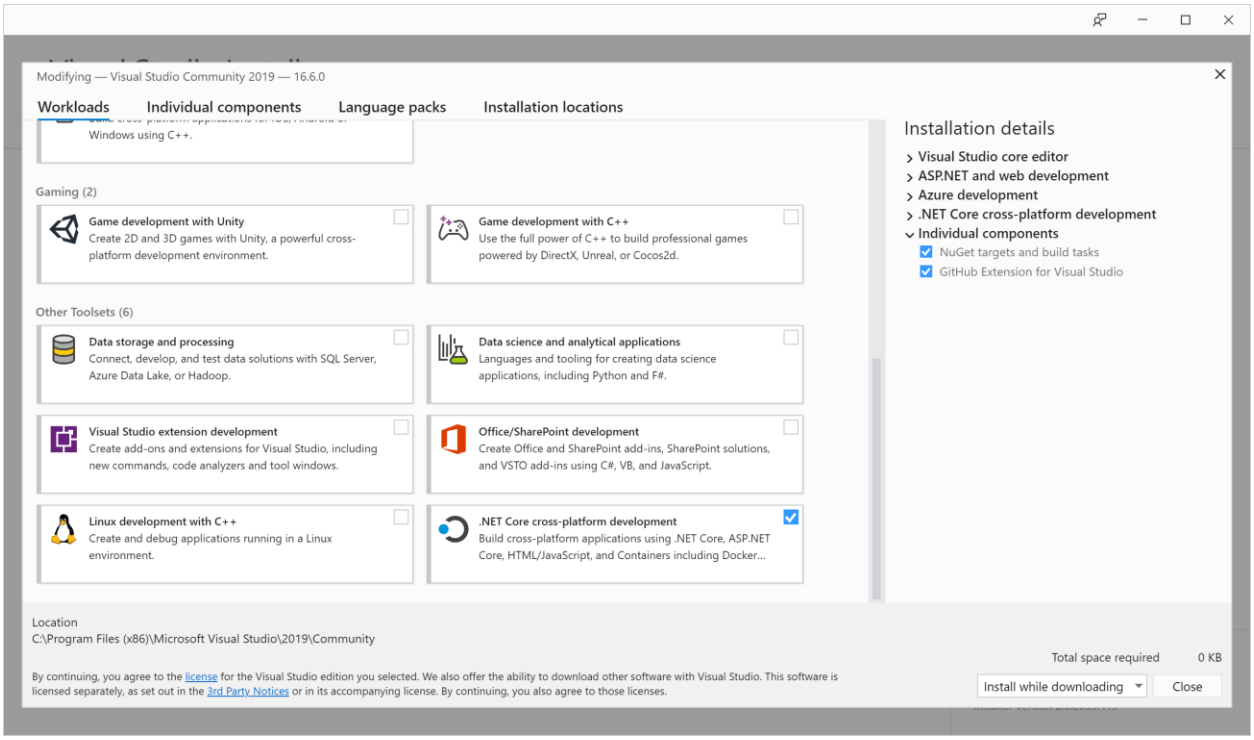


Figure 3-1: Visual Studio Installer

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 way of installing, but the process boils down to the same general steps:

1. Install prerequisites and configure the package manager of your distribution.
2. 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.

There are no prerequisites for macOS, so you can install the official SDK for macOS by downloading it directly from the official .NET Core website.

On Mac you can also just install Visual Studio 2019 for Mac, and you will have .NET Core installed as part of the IDE.

On Linux, 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 each of the tools with which you can build .NET Core apps.

Building the 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.

Code Listing 3-1a

```
dotnet new console --output HelloWorldApp
```

This command creates a **`new`** application of type **`console`** and puts it (**`--output`**, or **`-o`** as shortcut) in the folder HelloWorldApp.

Now, to run the application you have two options. You can move into the folder where the application has been created (**`cd HelloWorldApp`**) and launch the **`dotnet run`** command to build and run the project. Alternatively, you can use the **`--project`** option and run it directly from the parent folder.

Code Listing 3-1b

```
dotnet run --project HelloWorldApp
```

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

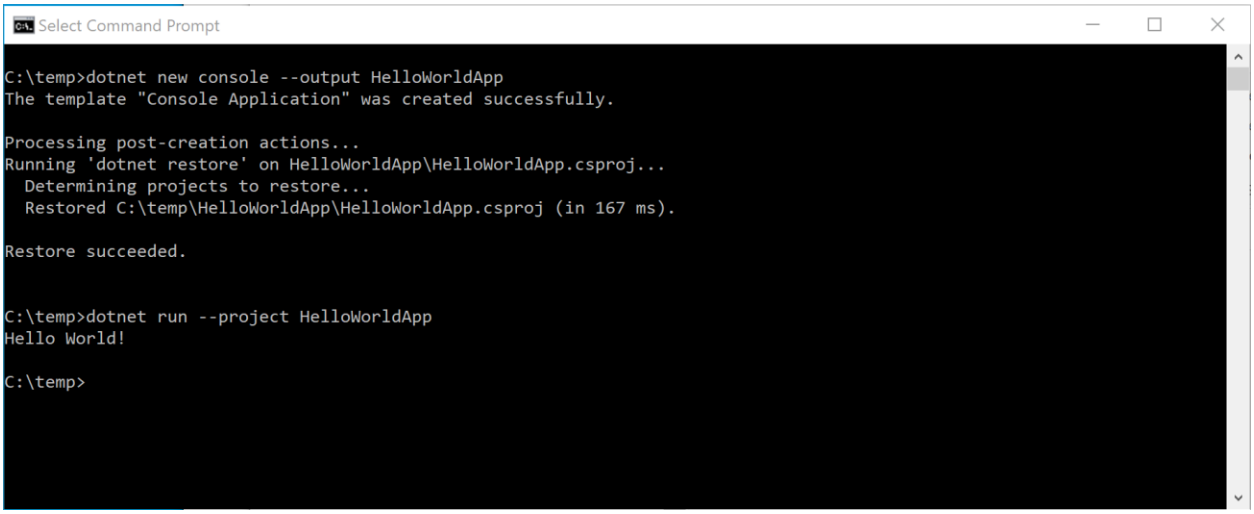


Figure 3-2: dotnet CLI

Let's dive into what happened here.

The **`dotnet new`** 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; it just prints the **`Hello World!`** string to the console.

Code Listing 3-2

```
using System;

namespace HelloWorldApp
{
    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 would even discuss them. You probably 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 Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

This is pretty basic, and it just says that the application targets the **`netcoreapp3.1`** framework.

Visual Studio

Another way you can build the same console application is by using Visual Studio. First, go to the **New Project** dialog (or from the splash screen), choose **Console** from the project types drop-down, and select **Console App (.NET Core)**, as shown in Figure 3-3.

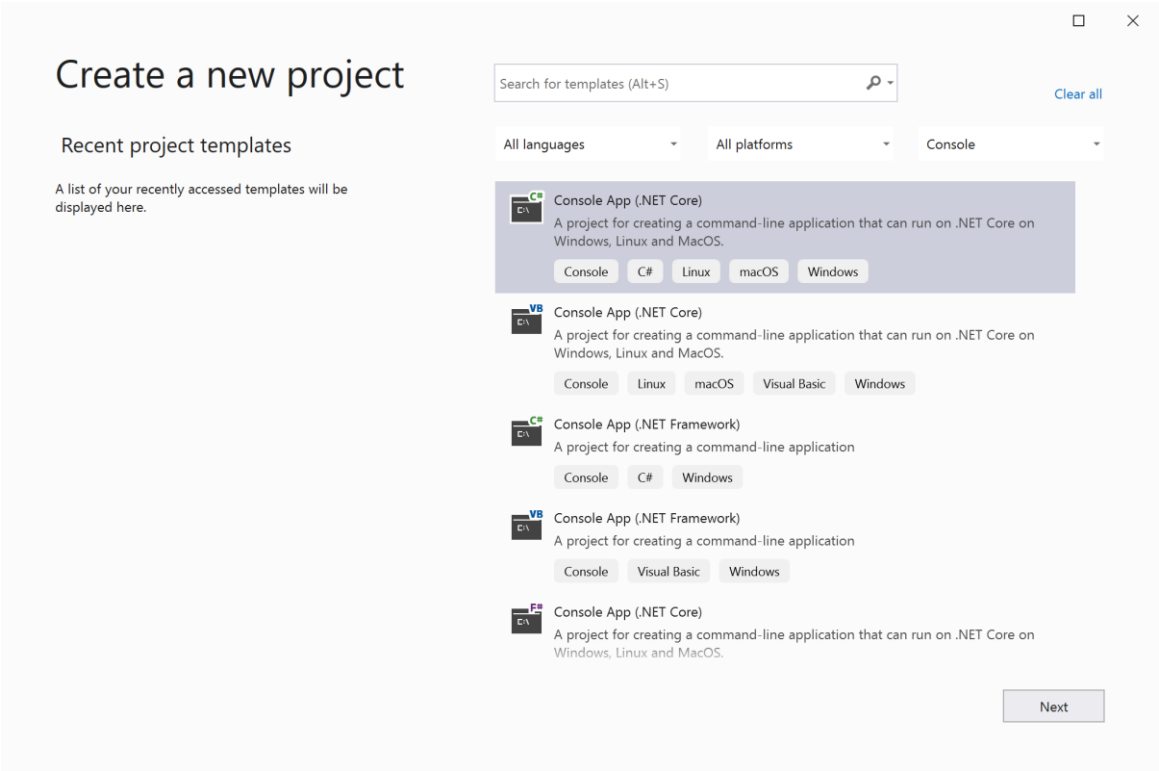


Figure 3-3: New Console Application Using Visual Studio 2019

Once the project has been created, you can edit it as you normally would, and run it. Figure 3-4 shows Visual Studio with the console application loaded in.

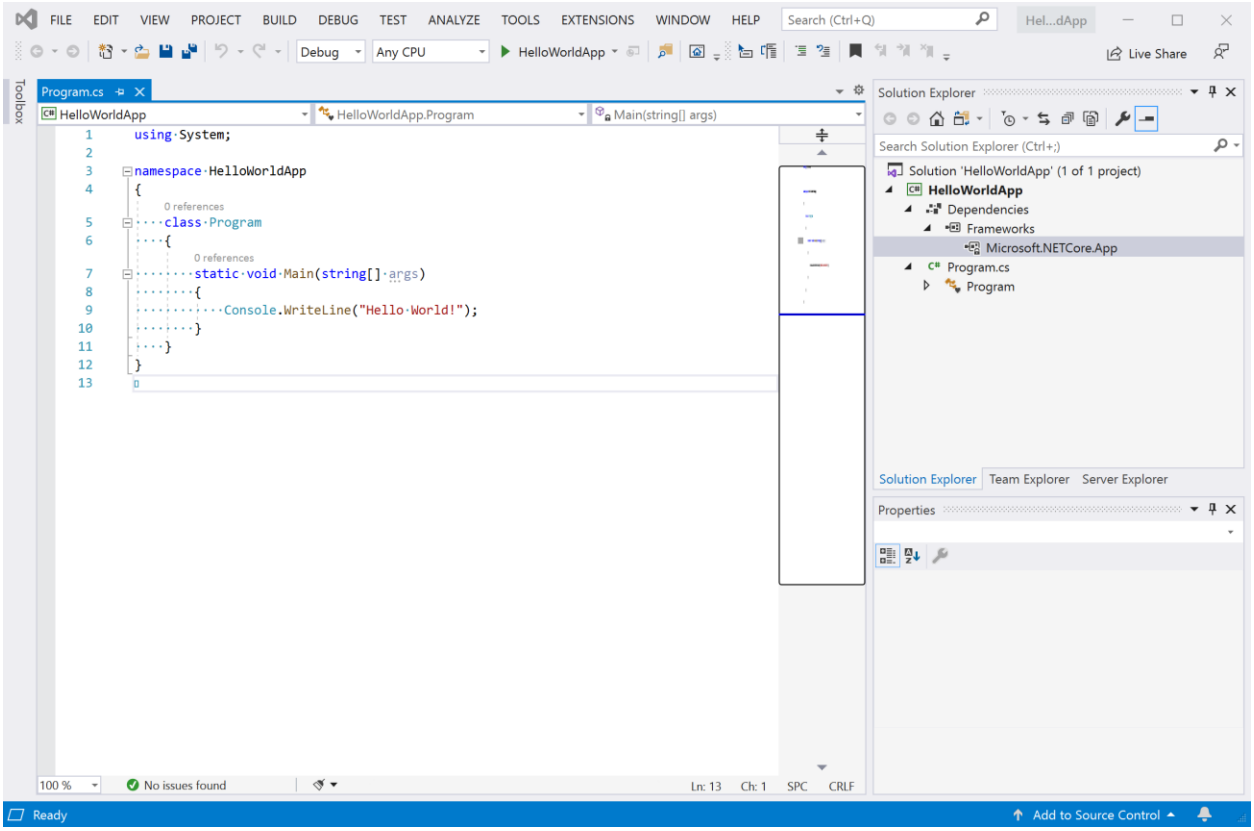


Figure 3-4: Hello World Application

Note: If you used .NET Core 1.1, you will remember that even a project as simple as "Hello World" had zillions of dependencies. This was because of the modular approach of the framework, staying true to the "reference what you need" approach. It proved to be a bit too much, so starting with .NET Core 2.0, all the basic dependencies are included in the `Microsoft.NETCore.App` metapackage. You can trim it down if you want to, but it at least simplifies the initial application development.

Conclusion

In these chapters, we've shown why 2016 is one of the few landmarks in our industry, along with Microsoft's release of Classic ASP in 1996, and the release of the .NET Framework and ASP.NET in 2002.

.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 changed 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 versions based on ASP.NET 4 and ASP.NET Core 2.x.

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 2019. If you prefer the lightweight Visual Studio 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.

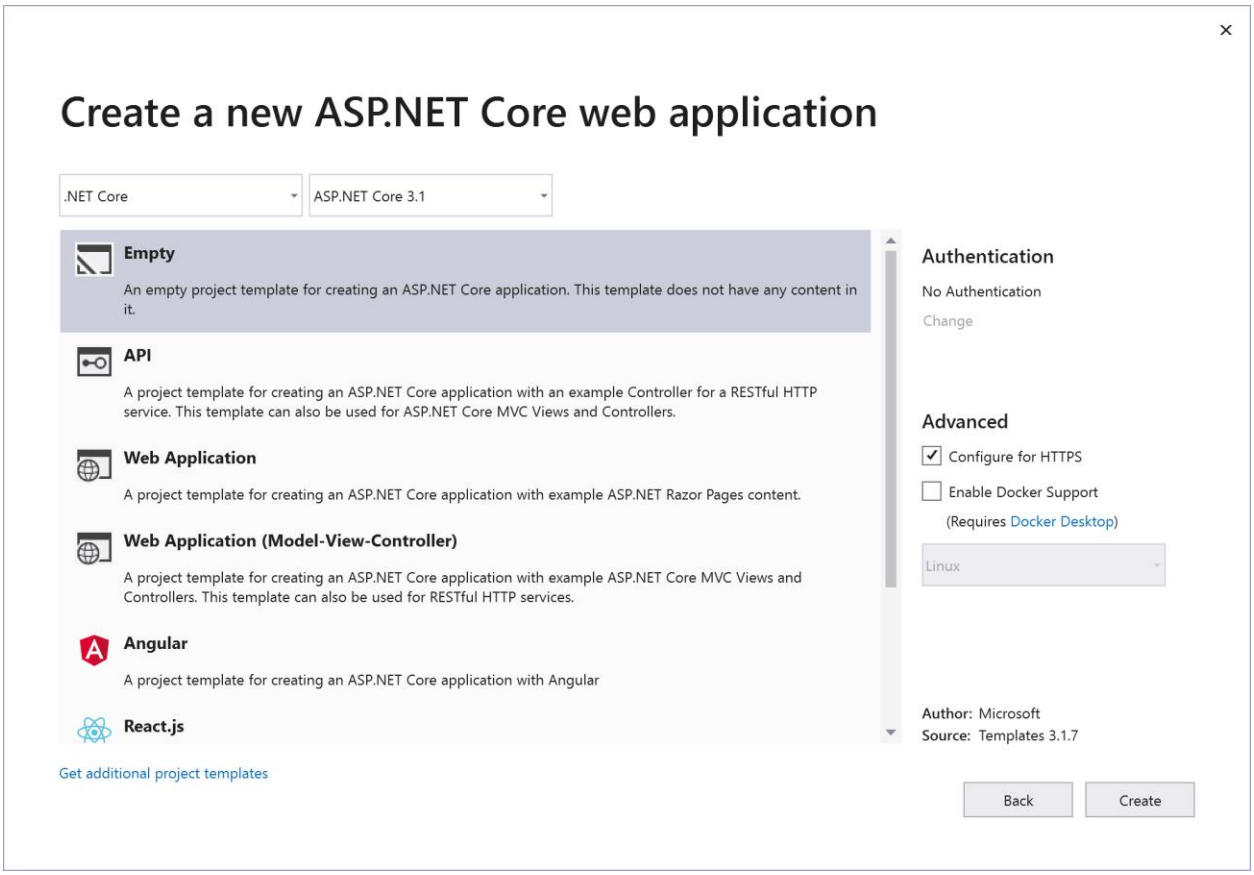


Figure 4-1: New Web Application Using Visual Studio 2019

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 previous chapters, the .NET Core host can only run console applications. So, the web app is a console application, too.
- ***project*.csproj**, where ***project*** is the name you provided in the creation wizard, is an XML-based project configuration file. It contains all our package references and some 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 the ASP.NET Core application (previously we were using the global.asax file).

The *project*.csproj file is not visible in the Visual Studio Solution Explorer window, but you can easily see what's inside by right-clicking on the project and selecting **Edit *project*.csproj**.

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;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
```

```
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }}
```

As you can see, this class is simple, as the only important method is **`CreateDefaultBuilder`**, which configures all the needed services for you.

If you are already using previous versions of ASP.NET Core (or you read the previous edition of this book), you'll notice that this class has changed significantly. Especially from v2 to v3, the **`WebHostBuilder`** has been deprecated in favor of the more generic **`HostBuilder`**.

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;

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

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

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

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapGet("/", 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 book OWIN Succinctly, available from Syncfusion.

The last lines of the listing are registering a request delegate (responding with the text string **`Hello World!`**) for a **`GET`** request on the root of the site (**`/`**). This is done via the endpoint routing middleware.

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:

- Developers no longer have an excuse not to use it, whereas before it was basically left to their judgment.
- You don't need to use third-party libraries.
- 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.

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, one class that is responsible for formatting text, and another 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 into 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. “Uncle Bob” Martin, systems built this way adhere to one of the five SOLID principles, the dependency inversion principle:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
B. Abstractions should not depend on details. Details should depend on abstractions.

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 the *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 to 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 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 them. It is actually easy. It all happens in the **`ConfigureServices`** method.

Code Listing 4-2

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

The parameter that 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 lightweight 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 are reused throughout the rest of the application. Such services typically hold an application state like an in-memory cache. They are added via the method **`services.AddSingleton<IApplicationCache, ApplicationCache>()`**.

Using dependency injection

Let's see an example of how DI 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, which is the only thing the consumer depends on.

Code Listing 4-3

```
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 4-4

```
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 new parameter to the constructor and a private variable to hold the reference of the external dependency.

Code Listing 4-5

```
private readonly ILogger<HomeController> _logger;
private readonly IClock _clock;

public HomeController(ILogger<HomeController> logger, IClock clock)
{
    _logger = logger;
    _clock = clock;
}
```

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

Code Listing 4-6

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

    return View();
}
```

You also have to change the view **Home/Index** to display the value.

Code Listing 4-7

```
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>@ViewData["Message"]</p>
    <p>Learn about <a href="https://...">building Web apps with ASP.NET
Core</a>.</p>
</div>
```

The complete file for the **HomeController** is provided in the following listing.

Code Listing 4-8

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IClock _clock;

    public HomeController(ILogger<HomeController> logger, IClock clock)
    {
        _logger = logger;
        _clock = clock;
    }

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

}
```

Tying it all together via the configuration

If you run the project now, you will get the following error: **`Unable to resolve service for type 'Services.IClock' while attempting to activate 'Controllers.HomeController'`**.

This is because you haven't configured the IoC container yet. It needs to be configured 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.AddControllersWithViews();

    // 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 of the interface.
- The configuration that binds the interface and implementation together and informs the container of their existence.

Environments

Most applications we deploy need 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), pre-production, and so on. In this book, we are showing only what comes out of the box with ASP.NET Core, but you will see how to easily add new environments.

One of my favorite features of ASP.NET Core, which is included with the framework, 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 syntax (more info at MSDN) and preprocessor directives.

This approach is tricky, and it requires you to build the application differently for each environment because the config transformation and the preprocessor directives are applied at compile time. Further, 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 detect that the application is running, 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 development mode. If it is **Staging**, you are running the application in a staging mode, and so on 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 click 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:50071",
      "sslPort": 44323
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The **iisSettings** section contains all the settings related to IISExpress, while the **profiles** section contains the Kestrel configurations. If you are familiar with the JSON format, you can edit all these values in Visual Studio by following the steps depicted in Figures 4-2 and 4-3.

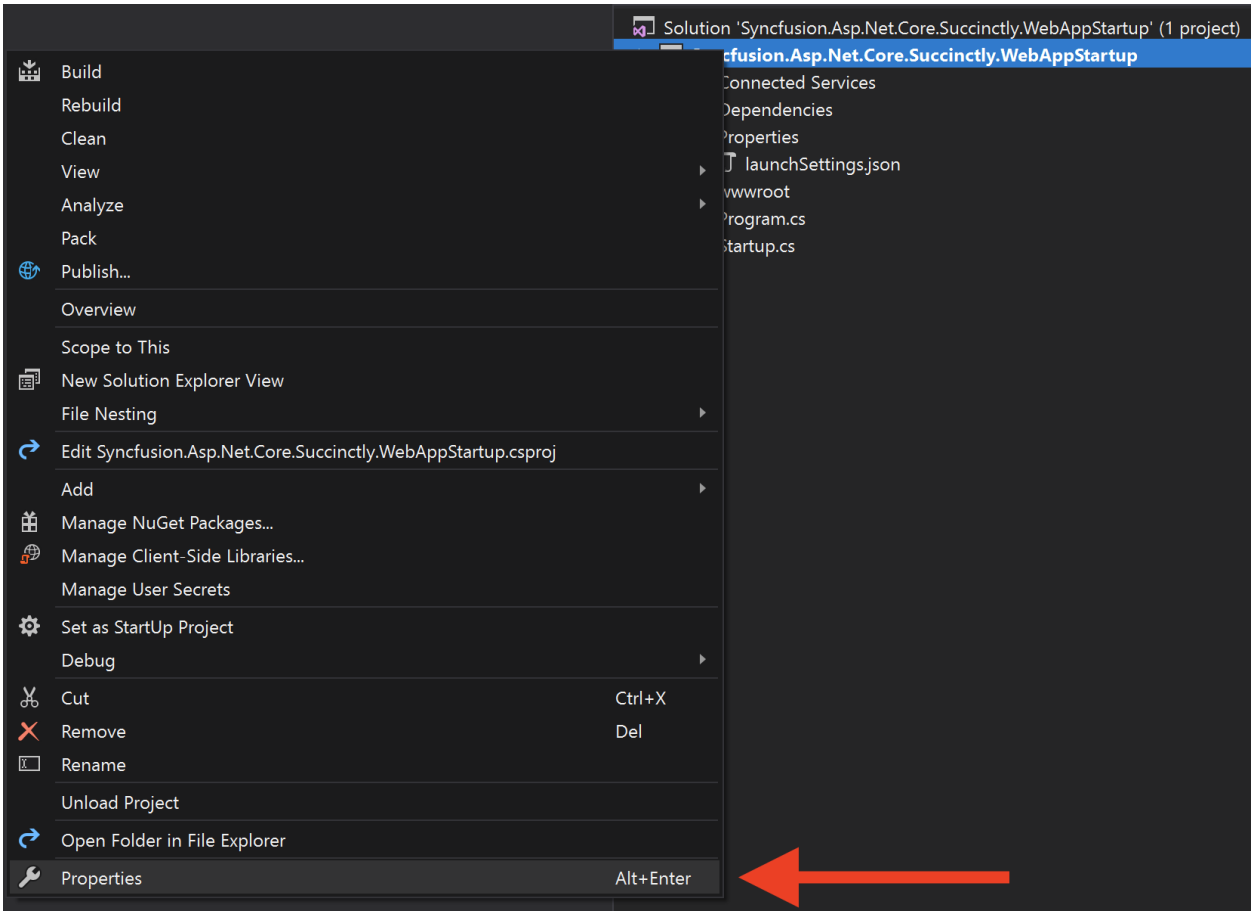


Figure 4-2: Changing IIS Express Settings, Step 1

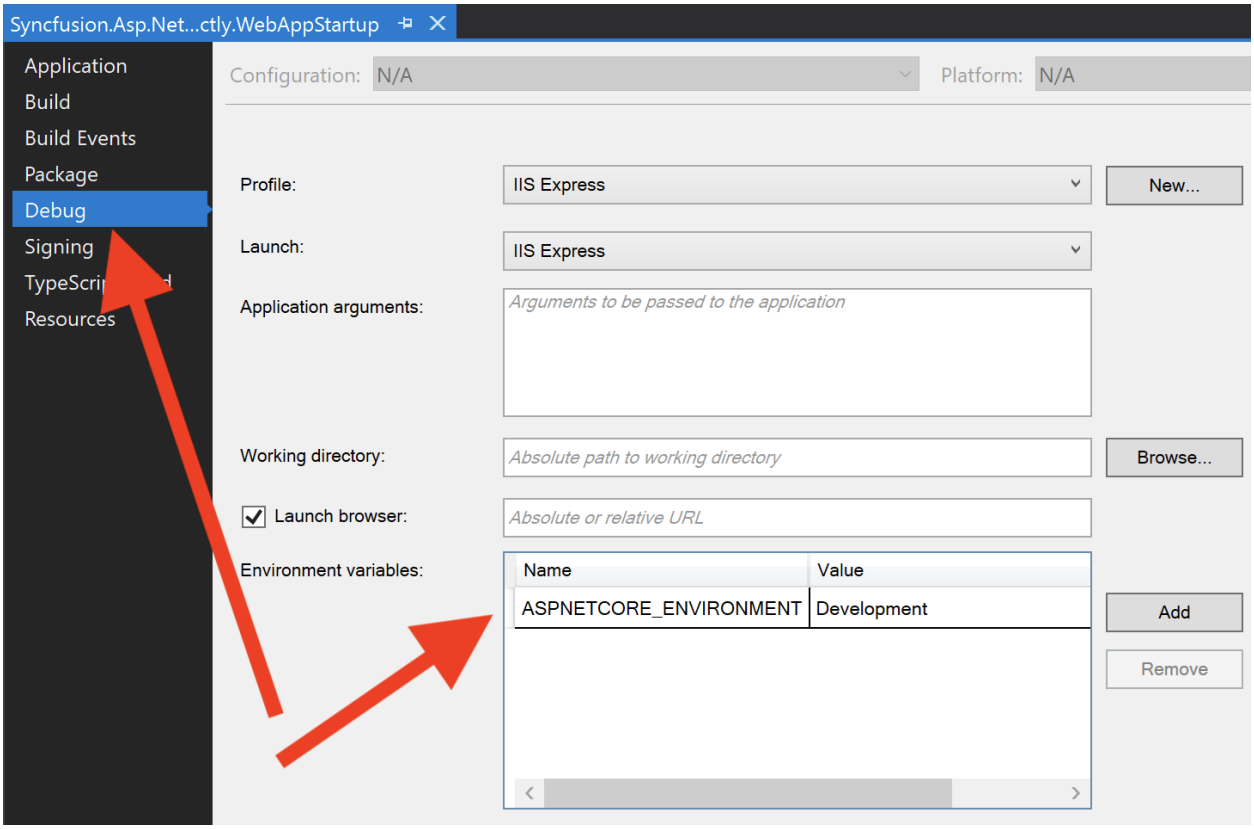


Figure 4-3: Changing IIS Express Settings, Step 2

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 the **`profiles`** section with our settings.

If you want to run your project outside of Visual Studio (by using the command line) and you want to force a specific profile defined in **`launchSettings.json`**, you can do this using a specific argument (**`dotnet run --launch-profile Development`**).

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 to see the stack trace of an error if 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 the **`Configure`** method and use it to change the behavior of your app. This is already done for you by using the ASP.NET Core project template.

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;

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

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment())
                  {
                        app.UseDeveloperExceptionPage();
                  }

                  ...

            }
      }
}
```

In this example, the **DeveloperExceptionPage** middleware is used only if our application is running in development mode, which is exactly what we want.

What we did in this class can be repeated in any part of your code as a controller, a service, or whatever needs to be different 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 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 even 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, and 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)
    {
        CreateHostBuilder(args).Build().Run();
    }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
}
```

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 **Configure** method 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 methods **IsUAT()** or **IsQualityAssurance**. So how can you create one?

If you think about it, the answer is pretty easy. You assign a new value to the **ASPNETCORE_ENVIRONMENT** variable using a set command in a command shell (**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, keeping with 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 Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions;

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

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment() || env.IsQualityAssurance())
                  {
                        app.UseDeveloperExceptionPage();
                  }

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

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 use a specific middleware.

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, but thanks to the new metapackage called **`Microsoft.AspNetCore.App`**, you don't need to install them each time you need one. The metapackage already includes all the most-needed packages for a web application, including **`Microsoft.AspNetCore.StaticFiles`**.

Now that you know this, you can edit the Startup.cs file to add the specific middleware, **`UseStaticFiles`**.

Code Listing 4-15

```
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 Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
      public class Startup
      {
            public void ConfigureServices(IServiceCollection services)
```

```
            {
            }

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment() || env.IsQualityAssurance())
                  {
                        app.UseDeveloperExceptionPage();
                  }

                  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 static files are served from a special folder called wwwroot.

If you created the project via the Empty template, you have to create it by yourself; otherwise it will already be there.

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 an ASP.NET Core application could be very useful is a web app that is a single-page application (SPA).

Here is Wikipedia's definition of an SPA:

> A single-page application (SPA) is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages.

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, and Vue) 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 in 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`**.

Code Listing 4-16

```
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 Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions;

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

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment() || env.IsQualityAssurance())
                  {
                        app.UseDeveloperExceptionPage();
                  }

                  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 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 Syncfusion.Asp.Net.Core.Succinctly.Environments.Extensions;

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

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment() || env.IsQualityAssurance())
                  {
                        app.UseDeveloperExceptionPage();
                  }

                  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 amount of time and return the best possible feedback to the user.

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

We have dedicated a section later in this chapter to logging frameworks, so 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 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.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

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

        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 in Code Listing 4-18 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).

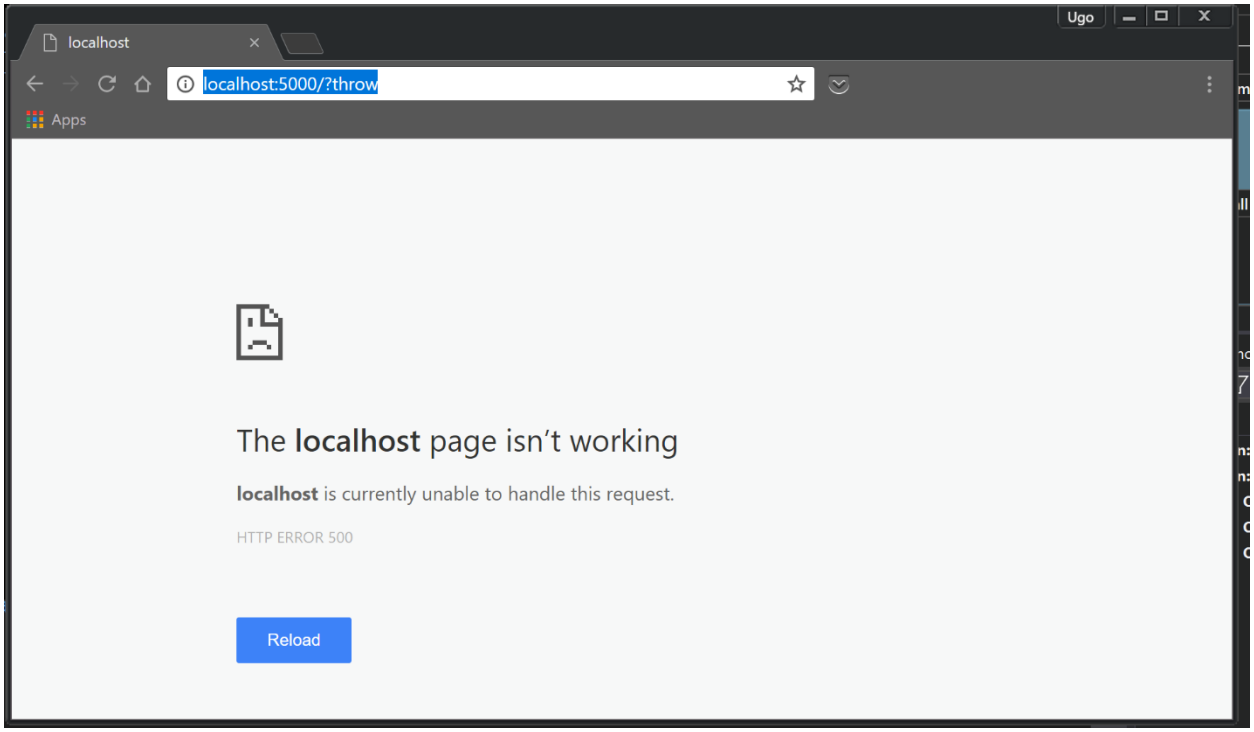


Figure 4-4: Default 500 Error

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 YSOD ("yellow screen of death") generated by the previous version of ASP.NET. To use the new, fancy error page, thanks to the metapackage **`Microsoft.AspNetCore.App`** already mentioned, it is enough to add just one line of code into your Startup.cs file.

Code Listing 4-19

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

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

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

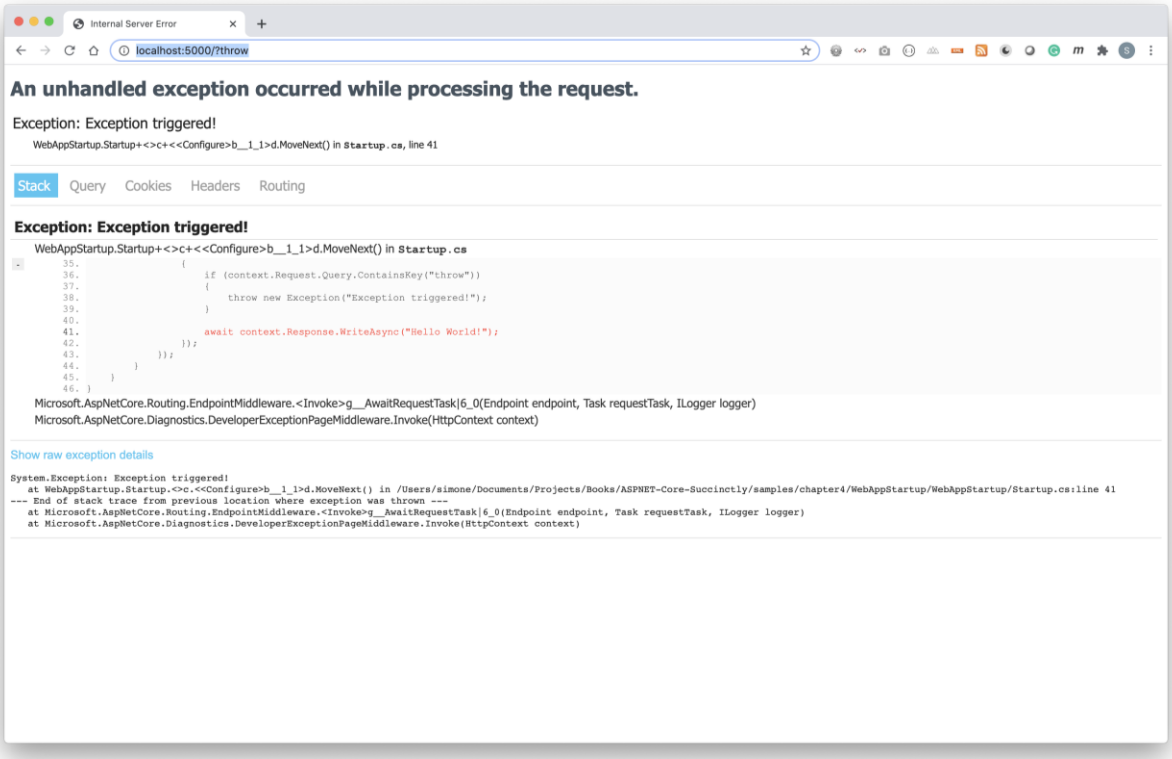


Figure 4-5: Developer Exception Page

On this page, there are four important tabs:

- **Stack**: Contains the stack information (line of code and callstack).
- **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.
- **Routing**: Contains information on the current route.

User-friendly error page

For reasons already explained, you can't show the error information in 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

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

Finally, your error management could be like this.

Code Listing 4-22

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

Configuration files

The way ASP.NET Core handles configuration files changed significantly with the new version. Before ASP.NET Core, you used the **`AppSettings`** section of the web.config file, but since the first version of ASP.NET Core, web.config is not needed anymore, unless you are hosting your application on internet information services (IIS).

If the **`AppSettings`** section is not needed anymore, how can you store your information? The answer is simple. You use external files (you can have more than one). Fortunately, there is a set of classes that helps 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 we 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 remains to hydrate the C# classes with the values from the JSON file. If you called your configuration file, appSettings.json, it comes out of the box (thanks to the new **WebHostBuilder**), and you can skip this code block.

Otherwise, you have to modify the Program.cs file in order to specify your configurations (you could have more than one), like this.

Code Listing 4-25

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
      public class Program
      {
            public static void Main(string[] args)
            {
                  CreateHostBuilder(args).Build().Run();
            }

                    public static IHostBuilder CreateHostBuilder(string[]
args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
```

```
                    webBuilder.ConfigureAppConfiguration((builderContext,
config) =>
                    {
                        config.AddJsonFile("mySettingsFile.json", optional:
false, reloadOnChange: true);
                    });
                });
      }
}
```

Now you have to move into the **Startup** class in order to read the configuration, and this is possible by requiring the **IConfiguration** interface from the constructor.

Code Listing 4-26

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Syncfusion.Asp.Net.Core.Succinctly.Environments;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
      public class Startup
      {
            private readonly Configuration _myConfiguration;

            public Startup(IConfiguration configuration)
            {
                  _myConfiguration = new Configuration();
                  configuration.Bind(_myConfiguration);
            }

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

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment())
                  {
                        app.UseDeveloperExceptionPage();
                  }
                  else
                  {
          app.UseStatusCodePagesWithRedirects("~/errors/{0}.html");
```

```
                }

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

Manage different environments

We already explained the importance of having different environments and how to manage them using 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 4-27

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

And look at appSettings.Production.json.

Code Listing 4-28

```
{
    "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 on how to handle multiple configuration files. To do that, change the code you wrote previously to this.

Code Listing 4-29

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                    webBuilder.ConfigureAppConfiguration((builderContext,
config) =>
                    {
                        IWebHostEnvironment env =
builderContext.HostingEnvironment;
                        config
                            .AddJsonFile("mySettingsFile.json", optional:
false, reloadOnChange: true)


.AddJsonFile($"mySettingsFile.{env.EnvironmentName}.json", optional: true,
reloadOnChange: true);
                    });
                });
    }
}
```

Now, for a development environment, you will get **`my-development-db-name`** as the database name; otherwise, it will be named **`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 the 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 4-30

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

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 4-31

```
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 4,300 packages.

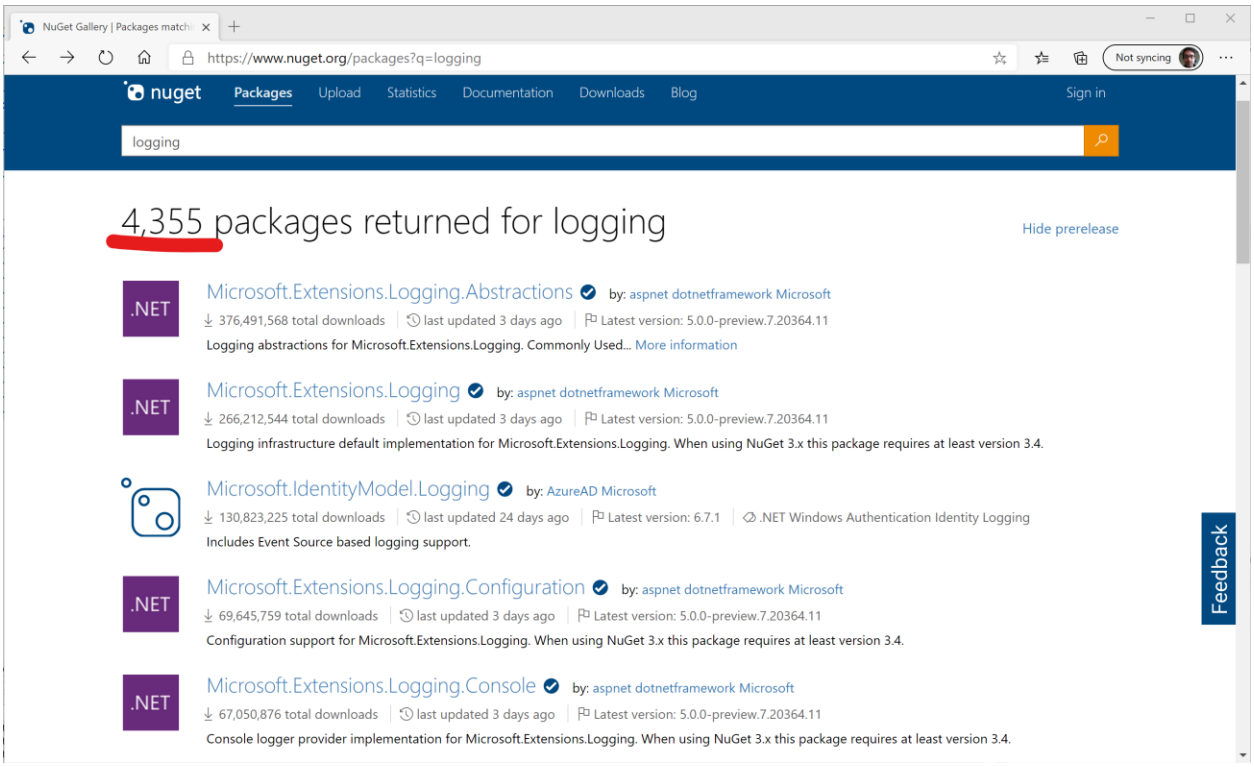


Figure 4-6: NuGet Logging Packages

Of course, not all of the packages 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 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, and Serilog.

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.

Configuring logging

First of all, you have to choose the output of the log you want—for example, console application, TraceSource, or EventLog.

Kestrel could be very helpful to use the console output, but thanks to the metapackage **`Microsoft.AspNetCore.App`**, you are ready to use the following providers:

- **Console**: The output will be printed into the console application (useful for Kestrel).
- **Debug**: The output will be printed into the debug windows in Visual Studio.
- **EventSource**: Useful if you like to analyze your logs using PerfView.
- **TraceSource**: This one is useful if you are running on .NET Framework instead of .NET Core. You can learn more about it here.

The logging output can be configured in the Program.cs file.

Visual Studio (and Visual Studio Code) offers an incredible feature called IntelliSense, so it suggests all the available options to us.

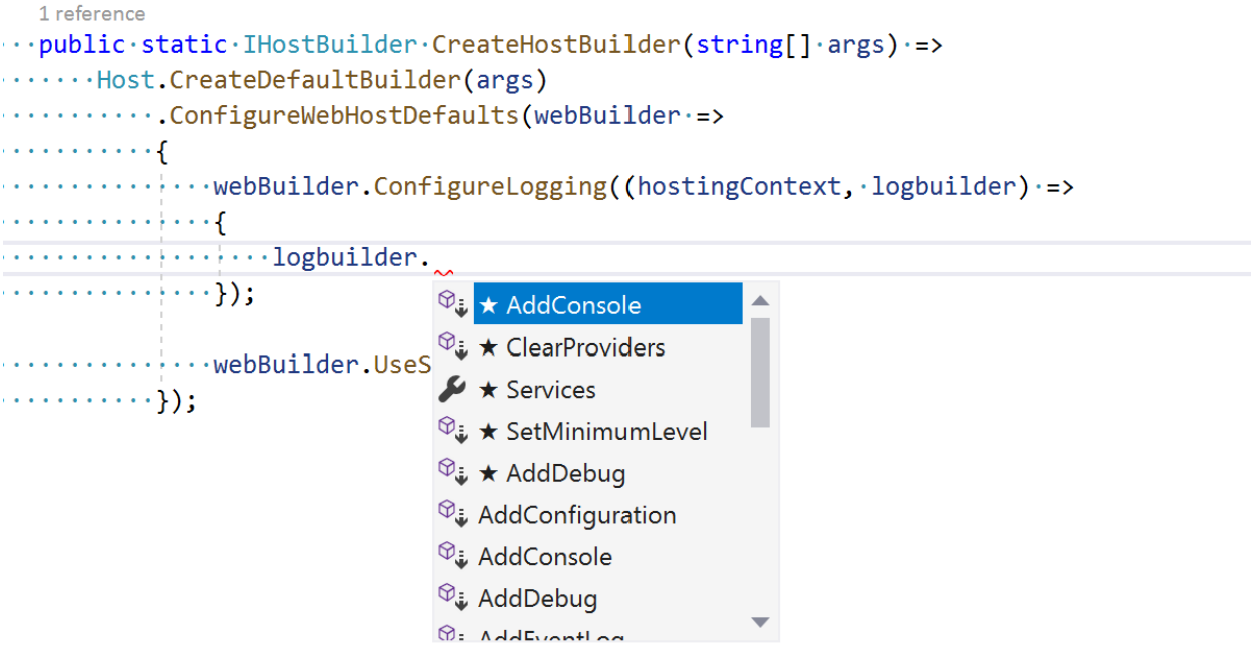


Figure 4-7: Configuring Logging

Let's see the Program.cs file.

Code Listing 4-32

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

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

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
      public class Program
      {
            public static void Main(string[] args)
            {
                  CreateHostBuilder(args).Build().Run();
            }

            public static IWebHostBuilder CreateHostBuilder(string[] args)
=>
                  Host.CreateDefaultBuilder(args)
                        .ConfigureLogging((hostingContext, logbuilder) =>
                        {
                              logbuilder.AddConsole();
                              logbuilder.AddDebug();
                        })
                        .UseStartup<Startup>();
      }
}
```

You didn't really have to add those lines since, by default, an ASP.NET Core application already has Console, Debug, EventSource, and EventLog (on Windows only) enabled. But that's what you would need to do to add other logging providers.

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 following output.

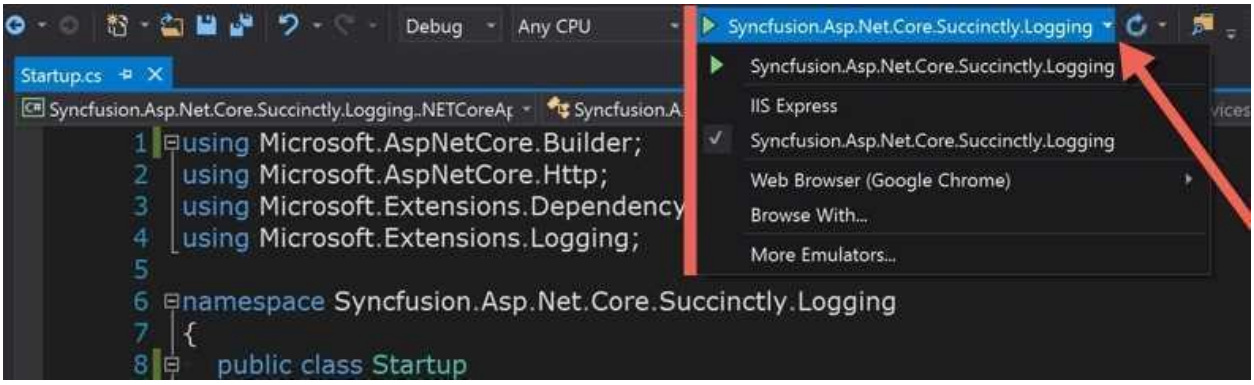


Figure 4-8: Running a Web Application Using Kestrel

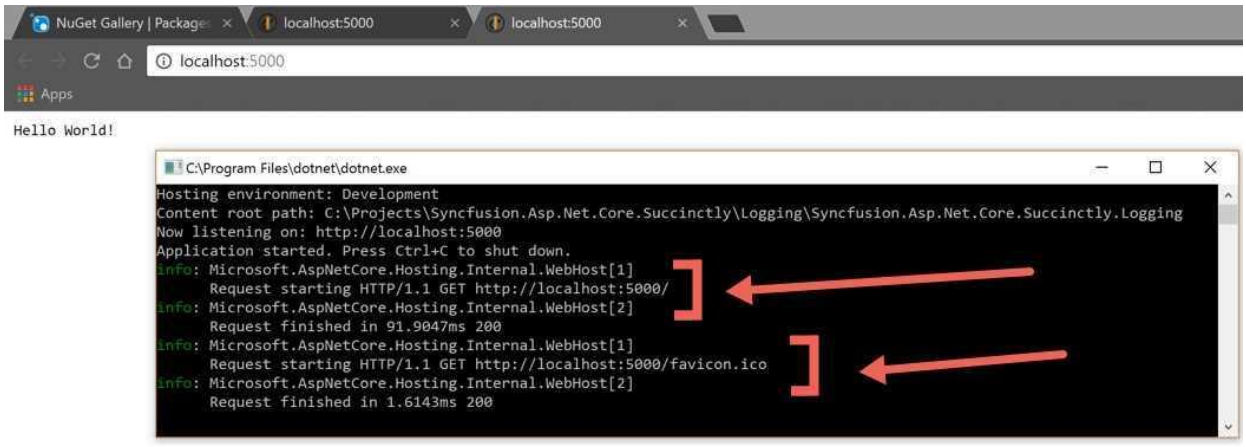


Figure 4-9: Web Application Console Output

Each request is logged twice: first when the server receives the request, and second when the server completes the request. In the figure, 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.

You can do it using the **`ILogBuilder`** interface, as shown in the following code.

Code Listing 4-33

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

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.ConfigureLogging((hostingContext,
logbuilder) =>
                    {
                        if
(hostingContext.HostingEnvironment.IsProduction())
                        {
                            logbuilder.SetMinimumLevel(LogLevel.Warning);
                        }
                        else
                        {
                            logbuilder.SetMinimumLevel(LogLevel.Debug);
                        }

                        logbuilder.AddConsole();
                        logbuilder.AddDebug();
                    });

                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Personally, I don't like this code, because you have to use an **`if`** statement to change the logging configuration. Fortunately, **`ILogBuilder`** offers another way to change the logging behavior via the configuration file.

Code Listing 4-34

```
webBuilder.ConfigureLogging((hostingContext, logbuilder) =>
{

logbuilder.AddConfiguration(hostingContext.Configuration.GetSection("Loggin
g"))

    logbuilder.AddConsole();
    logbuilder.AddDebug();
});
```

In this way, it is enough to specify the correct logging configuration into your configuration file.

Code Listing 4-35

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

More information about logging with ASP.NET Core 3.x is available here.

Add the 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 4-36

```
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 a custom logger

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
- Elmah.io
- Loggr
- NLog
- Slack
- MongoDb

Health checks

Health checks are a new feature introduced with ASP.NET Core 2.2 that allows an ASP.NET Core application to report its own health status over an HTTP endpoint. These endpoints can be used by networking or orchestration infrastructure to decide whether to direct traffic to the application.

For example, a load balancer might decide to take the application out of the pool if it becomes unhealthy, and wait till it becomes healthy again before re-including it. Also, health monitoring tools can check and report the status of applications.

Basic checks

By default, ASP.NET Core provides only a basic health check, which tells whether the app is running or not. But more advanced probes can be developed to check, for example, memory or disk usage, and even dependencies such as database connections or external services.

Let's see how to configure the basic health check. All that is needed is to add two lines of code, one in the **`ConfigureServices`**, and one in the **`Configure`** methods.

Code Listing 4-37

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    ...
    app.UseEndpoints(endpoints =>
        {
            ...
            endpoints.MapHealthChecks("/health");
            ...
        });
    ...
}
```

The first configures the service to register different health checks, and the second registers the middleware to respond on the endpoint **`/health`**.

Database checks

This basic probe does nothing more than return **`Healthy`** if the application is running. But more sophisticated probes can be added from ASP.NET Core directly or by either writing your own (by implementing the **`IHealthCheck`** interface) or using some contributed by the community, like the **`AspNetCore.Diagnostics.HealthChecks`** package. This way you can provide health checks for database engines, queues, caches, and so on.

To add the SQL Server check, modify the call to **`AddHealthChecks`** by specifying the additional check you want to run.

Code Listing 4-38

```
services.AddHealthChecks()
  .AddSqlServer(Configuration["ConnectionStrings:DefaultConnection"]);
```

Since this is a community feature, you need to install the NuGet package **AspNetCore.HealthChecks.SqlServer**.

This just checks that your app can connect to the database server and can run a simple query (the check in this package executes **SELECT 1**), but doesn't say anything about your specific database. If you use Entity Framework to connect to your database, you can also check that the database has been correctly created with all tables needed.

Code Listing 4-39

```
services.AddHealthChecks()
    .AddDbContextCheck<MyAppDbContext>();
```

Also this health check needs an additional nuget package, this time from Microsoft: **Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore**.

Application readiness and liveness

In some cases, an application might be functioning, but not be ready to respond to requests. This could be an application with a long startup time, for example initializing a cache, or similar long operations. In this case, we can differentiate two type of states:

- Readiness: Indicates when an app can respond to requests.
- Liveness: Indicates when the app is functional.

Normally the readiness check is only performed when an application has just been started, to make sure the load balancer or orchestrator can start directing traffic. After the application becomes "ready," only the liveness check is performed.

Typically, while a Liveness probe would return **Healthy** or **Unhealthy**, the readiness probe could return **Degraded** if the application is not ready to process requests yet.

Additional customization

The health check system is widely customizable. You can specify to run the health endpoint on a separate port by passing its value in the call to the **MapHealthChecks** function.

Code Listing 4-40

```
endpoints.MapHealthChecks("/health", port: 5002);
```

Another behavior you can change is the output format. The default behavior is to return **Unhealthy** if one of the checks fails, and **Healthy** if all the checks succeed. However, you might want to also see the details of all checks, or have your custom output for a specific load balancer or monitoring software. You can do this by specifying the custom response writer.

The following code listing shows how to code a simple writer that returns the status of each individual check.

Code Listing 4-41

```
endpoints.MapHealthChecks("/health", new HealthCheckOptions()
{
    ResponseWriter = WriteResponse
});

...

private static Task WriteResponse(HttpContext httpContext,
    HealthReport result)
{
    var sb = new StringBuilder();

    sb.Append($"Overall status: {result.Status}"+Environment.NewLine);
    foreach (var entry in result.Entries)
    {
        sb.Append($" - {entry.Key} : {entry.Value.Status}" +
Environment.NewLine);
    }
    return httpContext.Response.WriteAsync(sb.ToString());
}
```

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 also 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, and 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 its own book.

Web API

A 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 with the help of the application protocol.

Probably the most well-known example is Facebook, which allows users to share content and posts, manage pages, and do more from any client, such as a mobile app or desktop. 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 from classic web navigation, except for the result. In normal HTTP navigation, when we use a browser to call www.tostring.it, the server returns HTML. For 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, called Web API. This library doesn't exist with the newest version, and you can handle APIs and classic requests using the same framework, because Web API has merged into the MVC framework. This is the primary difference between the old and new ASP.NET.

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 we saw in the previous chapters; you don't have to install it because it is included in **`Microsoft.AspNetCore.App`**.

Once it's installed (using either the project templates or the dotnet CLI), register the service and configure it within your web application like this.

This is actually done for you by the template. Notice the important lines are the ones that configure the controllers. In **`ConfigureServices`**, the controllers are registered in the DI system, and at the end of the **`Configure`** method, the controllers are mapped via the endpoint routing system.

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.AddControllers();
            }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
      }
}
```

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, which uses different HTTP verbs according to the type of action needed.

Suppose we have the request **`api/users`**.

Table 1: Handling the Request api/users

Resource Sample	Read (GET)	Insert (POST)	Update (PUT)	Partially Update (PATCH)	Delete (DELETE)
Action	Gets a list of users	Creates a user	Update users with batch update	Batch-updates users only with the attributes present in the request	Errors or deletes all users, depending on what you want
Response	List of users	New user or redirect 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

If we want to manage a single user, the request should be **`api/users/1`**, where **`1`** is the ID of the user we want to work with.

Table 2: Handling the Request api/users/1

Resource Sample	Read (GET)	Insert (POST)	Update (PUT)	Partially Update (PATCH)	Delete (DELETE)
Action	Gets a single user	Return an error because the user already exists	Update 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 redirect to the URL to get the single user	The updated user (complete object) or redirect to the 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 deviating 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 the APIs from the standard controllers.

Code Listing 5-2

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

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

            [HttpGet("{id}")]
            public User Get(int id)
            {
                  var users = new[]
                  {
                        new User() {Id = 1, Firstname = "Ugo", Lastname =
"Lattanzi", Twitter = "@imperugo"},
                        new User() {Id = 2, Firstname = "Simone", Lastname
= "Chiaretta", 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.

`[ApiController]`

This attribute configures the behavior of the controller in a way that is more suited for web APIs, like automatically returning HTTP code 400 for invalid requests.

`[Route("[controller]")]`

This says that this controller can manage all the requests whose URL is named like the controller. Requests to the URL **`/users`** will be handled by this controller. If you want the more standard behavior where web APIs are always under a **`/api`** path, you can change the attribute to **`[Route("api/[controller]")].`**

`[HttpGet]`

This attribute specifies the verb for the action it is decorating. 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.

Hypertext Transfer Protocol (HTTP/1.1) doesn't specify limits for the length of a query string, but limits are imposed by web browsers and server software. You can find more info here.

This approach is not so different from what you saw before. The only difference that the data comes from the body instead of the query string.

Code Listing 5-3

```
// 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",
Twitter = "@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 the following:

- **`[FromBody]`**: This specifies that data comes from the payload of the HTTP request.
- **`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).
- **`return new EmptyResult();`**: Basically, this means status code 200 (OK). The response body is empty.
- **`return new NotFoundResult();`**: This is the 404 message that is used when the requested client can't be found.

Testing APIs

A browser can be used to test the API because APIs are being implemented over HTTP, but there are other applications that can help with this, too. One popular application is Postman, which you can download for free here.

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

Retrieve users (**`/api/users`** using **`GET`**).

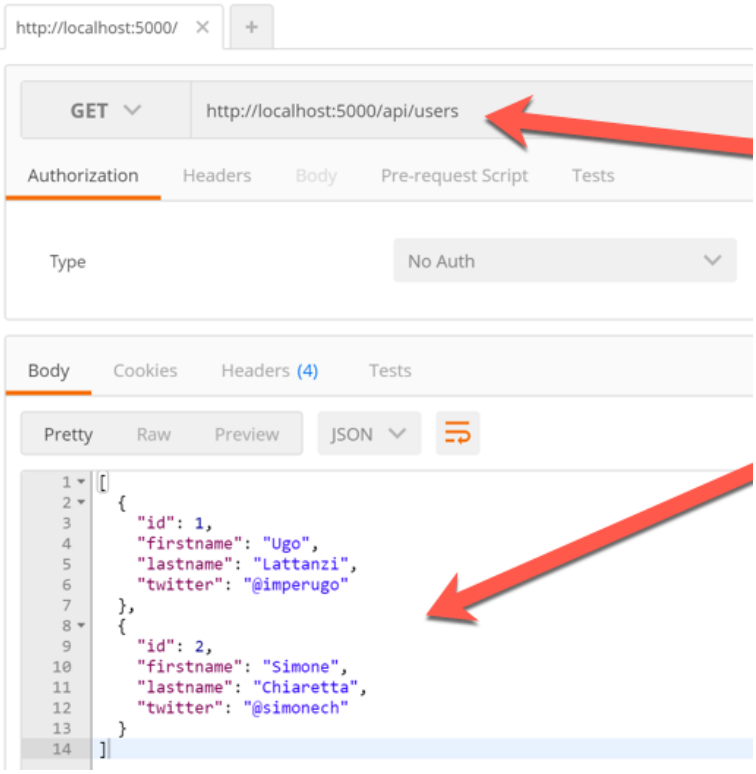


Figure 5-1: Using Postman to Test REST Endpoint-1

Retrieve one user (**`/api/users/1`** using **`GET`**).

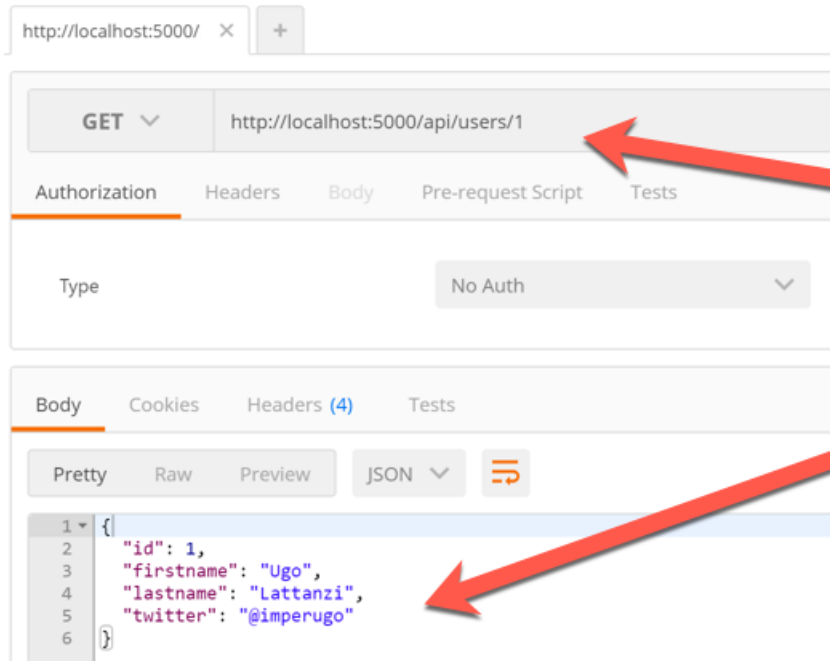


Figure 5-2: Using Postman to Test REST Endpoint-2

Create user (**`/api/users`** using **`POST`**).

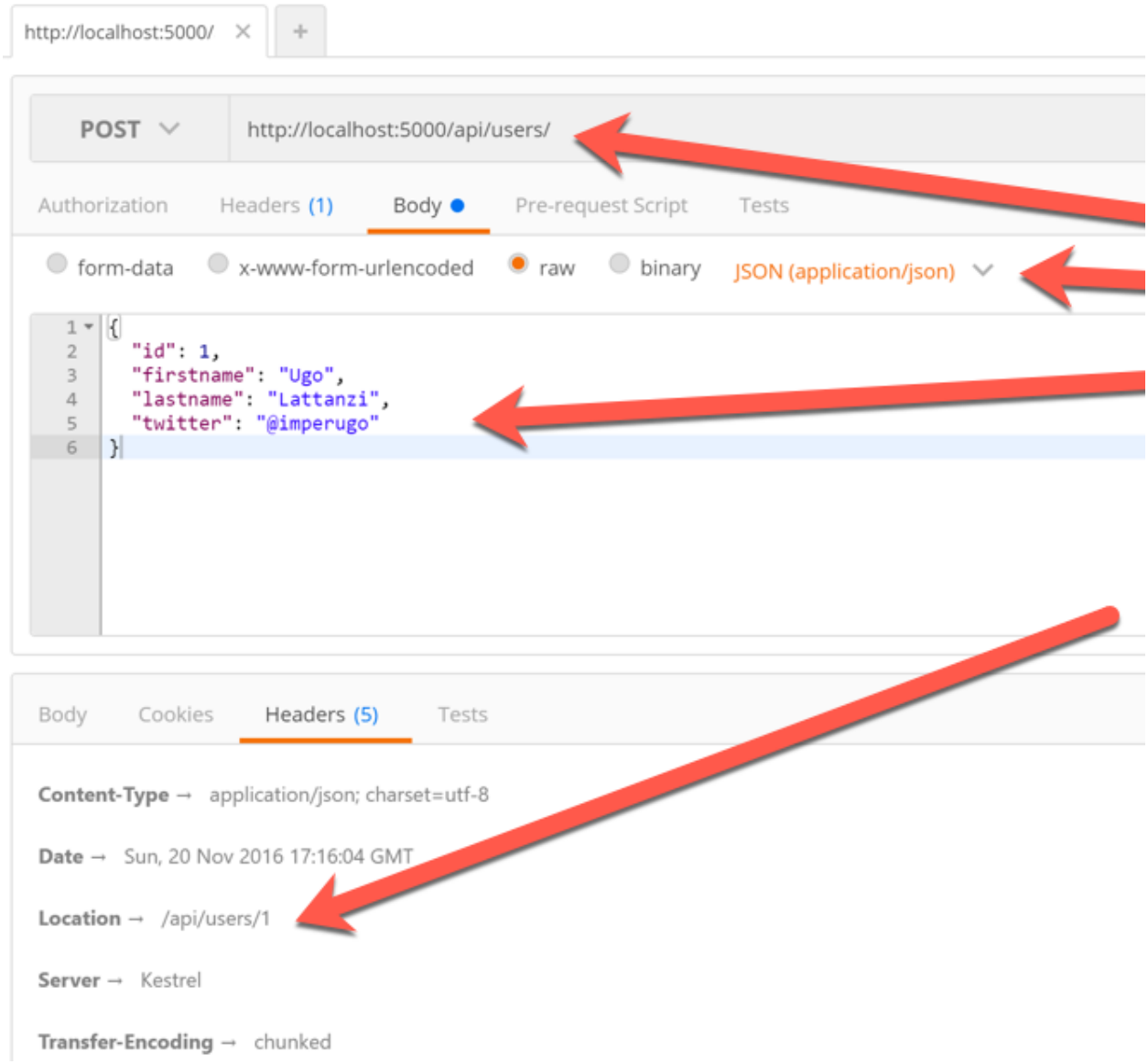


Figure 5-3: Using Postman to Test REST Endpoint-3

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

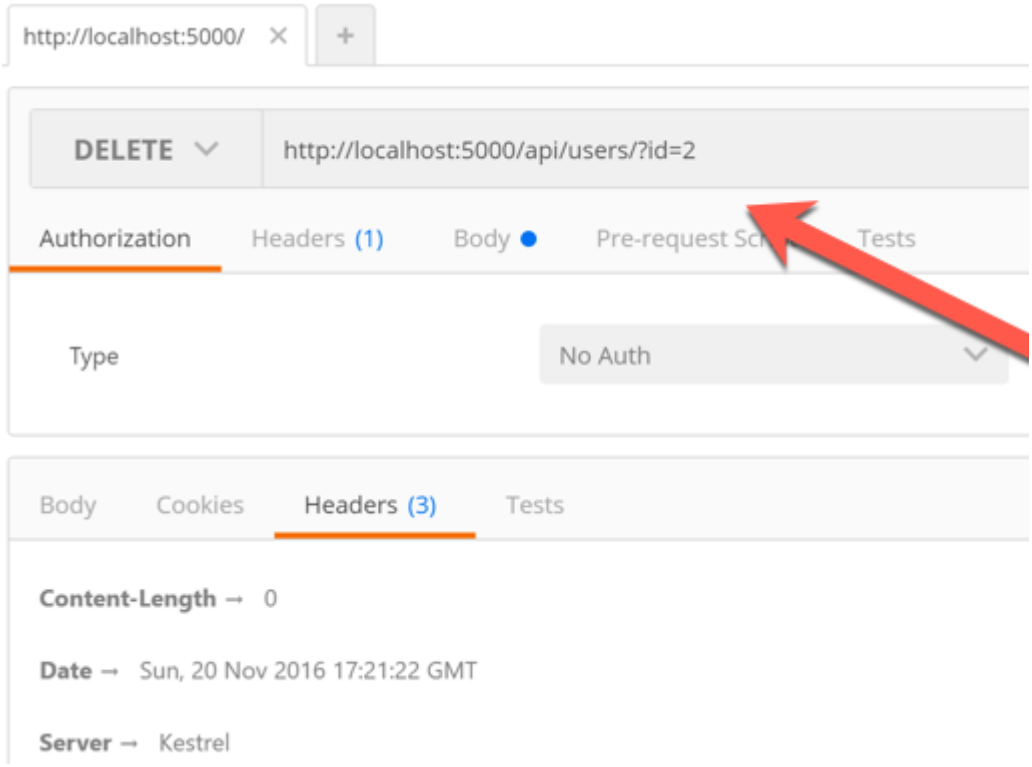


Figure 5-4: Using Postman to Test REST Endpoint-4

Documenting APIs

Another interesting feature of web APIs is the ability to "self-document" the methods. This can be done using a standard called OpenAPI or Swagger.

ASP.NET Core doesn't come with a library for Swagger, but as usual you can add the functionality using a third-party NuGet package. There are two main libraries available, Swashbuckle and NSwag, and they are configured in the same way. The following examples use NSwag.

Here are the steps to enable NSwag in a Web API project:

1. Install the **`NSwag.AspNetCore`** NuGet package.
2. Configure the services needed inside the **`ConfigureServices`** method, with **`services.AddSwaggerDocument()`**.
3. Register the Swagger API middleware, in the **`Configure`** method with **`app.UseOpenAPI()`**.
4. Register the Swagger UI middleware in the **`Configure`** method **`app.UseSwaggerUi3()`**.

You can now browse to http://localhost:<port>/swagger and see the specification of your APIs.

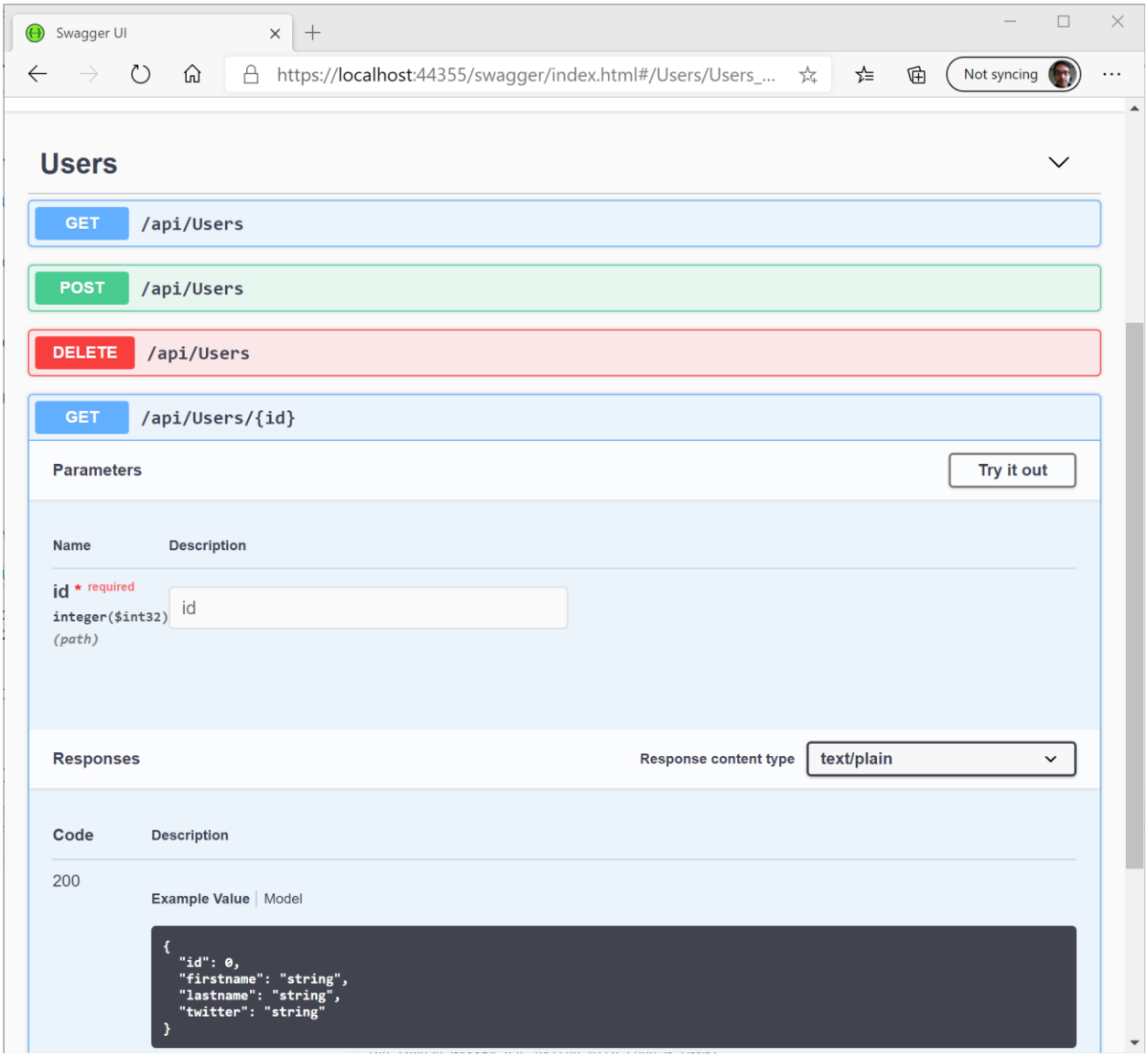


Figure 5-5: Swagger Documentation

gRPC

The trend of recent years, with regard to application architecture, is to develop microservice applications. This kind of architecture offers several advantages:

- Highly maintainable and testable.
- Loosely coupled.
- Independently deployable.
- Organized around business capabilities.
- Owned by a small team.

The microservice architecture enables the rapid, frequent, and reliable delivery of large, complex applications. It also enables organizations to evolve their technology stack more easily.

As you can see, the number of advantages offered is really high, but let's see an example.

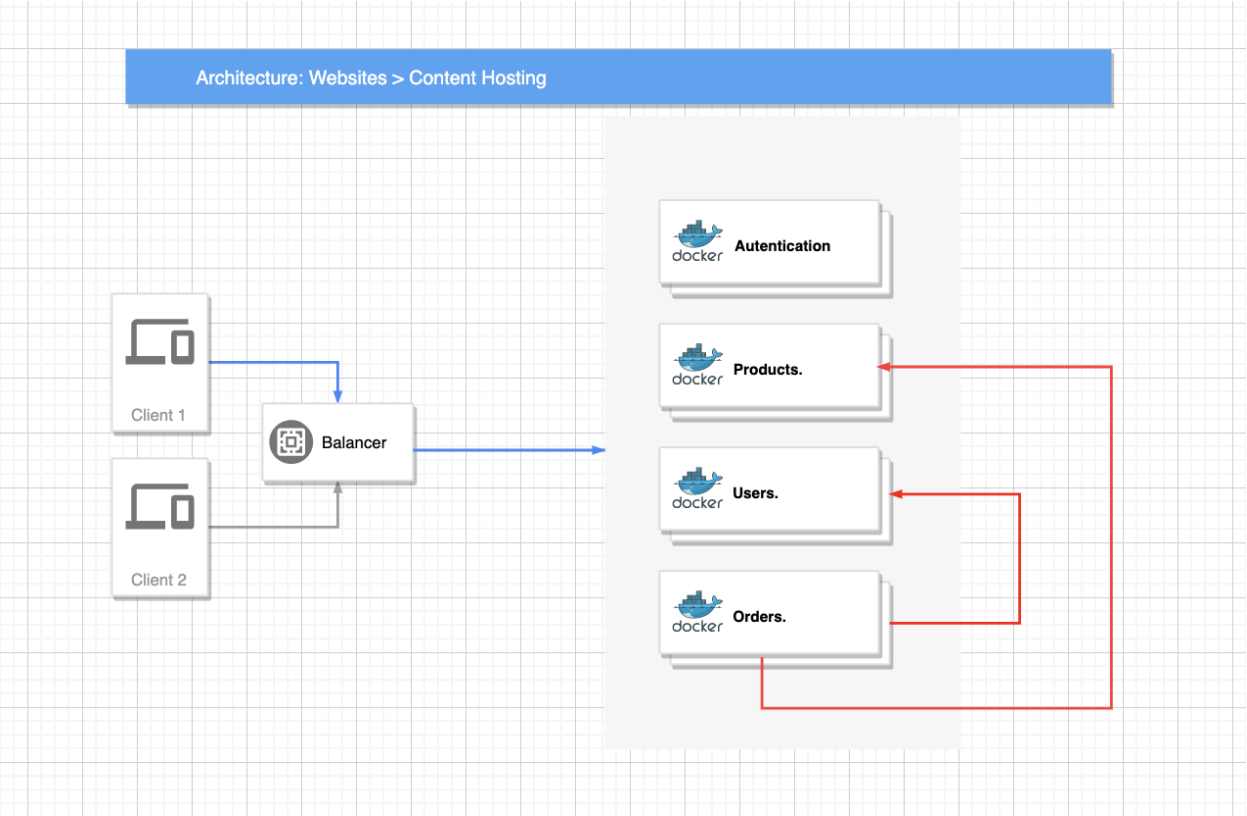


Figure 5-6: A Typical Microservice-Oriented Application

In Figure 5-6, there are four different microservices, each of them with a specific domain and scope and developed by four different teams, probably with different programming languages.

The team that is managing the authentication probably doesn't know where the products are, their structure in the database, or how to retrieve them. The same applies also for orders, users, and so on. This doesn't mean that the microservice managing the orders doesn't need the user information or the product information.

In fact, if you think of a real scenario, the order probably has knowledge of the user who created it and the products bought.

The question here is: what is the best way to retrieve the product information from the order microservice?

Retrieving the data directly from the database isn't a solution because it breaks all the benefits we discussed. The team needs to know where the production information is, its structure into the database, and so on.

It is clear that the order's microservice should ask of the product microservice what it needs, but how? A possible solution is to expose a set of REST APIs used to share the information. This would work perfectly, but it would negatively affect performance.

REST is perfect for the browser, but it has a performance cost that would be too large for internal communication. First of all, the size of the bytes transferred through the request, and then the REST over HTTP, isn't bidirectional. This means that the microservice A needs to create a request to microservice B in order to retrieve the information, and vice versa: two connections and two requests.

In 2015, Google started to develop a protocol that solves the REST problems, and named it gRPC (Google Remote Procedure Call). You can see its definition directly on the official site.

gRPC is a modern, open-source, high-performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications, and browsers to backend services.

The important thing to know now is that gRPC is built on top of HTTP2 (so it can use bidirectional communication), and the serialization is based on Protobuf in order to reduce the amount of data transferred on each request.

If you browse the website, you can see that almost all important languages are supported (Android Java, C#, C++, Go, Kotlin, Node, Python, and so on). This proves that gRPC can be used in mixed scenarios where each microservice could be developed with a different programming language.

gRPC and ASP.NET Core

.NET, starting from version 3, supports gRPC. The same way we do for other types of projects, here we have to install a specific package into your project. The needed package depends on which side you are working on because, exactly as it happens for the REST over HTTP, there is a client (the microservice that is asking for the information) and a server (the microservice that is returning the information).

Let's start with the server part.

First of all, install in your application the package called **`Grpc.AspNetCore`** from NuGet. Your .csproj file should have the following line.

Code Listing 5-4

```
<PackageReference Include="Grpc.AspNetCore" Version="2.29.0" />
```

Now that you have all your packages available in your project, the first step is to create the signature of the service that will expose the data to another microservice.

Because gRPC is strongly related to Protobuf, we need to create a proto file in our project.

Tip: A good practice is to create a file for each service, for example products.proto, users.proto, and so on.

We want to expose our product to another microservice, so our proto file would look like this.

Code Listing 5-5

```
syntax = "proto3";

package products;

// The product service definition.
service Products {
  // Sends a product
  rpc GetProduct (ProductRequest) returns (ProductReply);
}

// The request message containing the id of the requested product.
message ProductRequest {
  string productId = 1;
}

// The response message containing the product information.
message ProductReply {
  string productName = "";
  string price = "";
}
```

The proto syntax is not a part of this book, so please read the reference here.

The package **`Grpc.AspNetCore`** includes some tools to generate all the needed C# classes at build time, so you have to edit the .csproj file and add these lines.

Code Listing 5-6

```
<ItemGroup>
      <Protobuf Include="**/*.proto" />
</ItemGroup>
```

Now build your project and create your first gRPC service.

Code Listing 5-7

```
using System.Threading.Tasks;
using Products;
using Grpc.Core;
```

```
using Microsoft.Extensions.Logging;

namespace Server
{
    public class ProductsService : Products.ProductsBase
    {
        private readonly ILogger _logger;

        public ProductsService(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<ProductsService>();
        }

        public override Task<ProductReply> GetProduct(ProductRequest
request, ServerCallContext context)
        {
            _logger.LogInformation($"Sending product with id
{request.productId }");
            return Task.FromResult(new ProductReply { ProductName = "Bike",
Price = "5000€"});
        }
    }
}
```

With the service implemented, now it needs to be registered in the application by modifying the **Startup.cs** file.

Code Listing 5-8

```
namespace Server
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddGrpc();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();
```

```
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapGrpcService<ProductsService>();
            });
        }
    }
}
```

The Client

Now that the server is ready to return data over gRPC protocol, let's see how to invoke a request and read data.

First of all, we have to copy the **`product.proto`** file we created for the server in the client (a good way would be to share that file among all the needed microservices) and install all the needed packages.

Like you did for the server part, modify the .csproj, including all .proto files.

Code Listing 5-9

```
<ItemGroup>
      <Protobuf Include="**/*.proto" />
</ItemGroup>
```

Once all packages are installed, the .csproj file contains the following lines.

Code Listing 5-10

```
<PackageReference Include="Google.Protobuf" Version="3.12.3" />
<PackageReference Include="Grpc.Net.Client" Version="2.29.0" />
<PackageReference Include="Grpc.Tools" Version="2.30.0">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles;
analyzers</IncludeAssets>
</PackageReference>
```

In order to demonstrate that gRPC is not related only to the web world, you are going to invoke the service from a console application.

Code Listing 5-11

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

```
using Grpc.Core;
using Grpc.Net.Client;

namespace Client
{
    public class Program
    {
        static async Task Main(string[] args)
        {
            var channel = GrpcChannel.ForAddress("https://localhost:5001");
            var client = new Products.ProductsClient(channel);

            await UnaryCallExample(client);

            Console.WriteLine("Shutting down");
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        private static async Task UnaryCallExample(Products.ProductsClient
client)
        {
            var reply = await client.GetProduct(new ProductRequest
{PproductId = "12345" });
            Console.WriteLine("Product: " + reply.ProductName);
        }
    }
}
```

Notes

gRPC is easy to use, offers good performance, and allows you also to stream information between the server and the client thanks to the HTTP/2 protocol.

On the gRPC Google repo, there are tons of examples for any scenario, starting from a classic server to client iteration like the previous examples, to using gRPC over Blazor, thanks to the gRPC Web.

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.

However, the registration in the **`Startup`** class must be slightly changed.

The **AddControllers** must be replaced with **AddControllersWithViews**, which also registers the services to handle views, and all the other concerns that were not needed in the APIs.

Also, the **MapControllers** inside the **Configure** method must be changed. It registers endpoints based on attribute routing and is not suited for the standard routing of ASP.NET MVC. In order to use the conventional routing, you need to replace it with (or add in addition) the controller route method.

Code Listing 5-12

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

With these changes, you can start creating a controller. 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 5-13

```
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, not JSON.

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

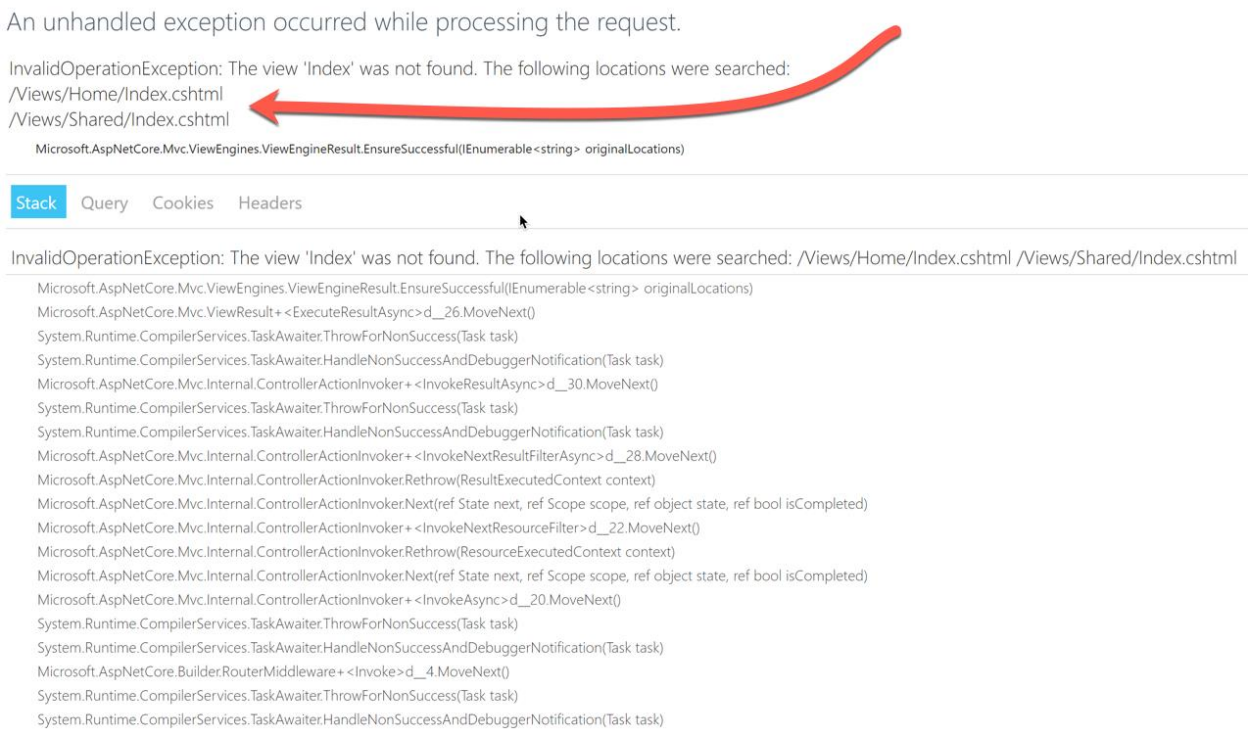


Figure 5-7: Unable to Find the MVC View

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.

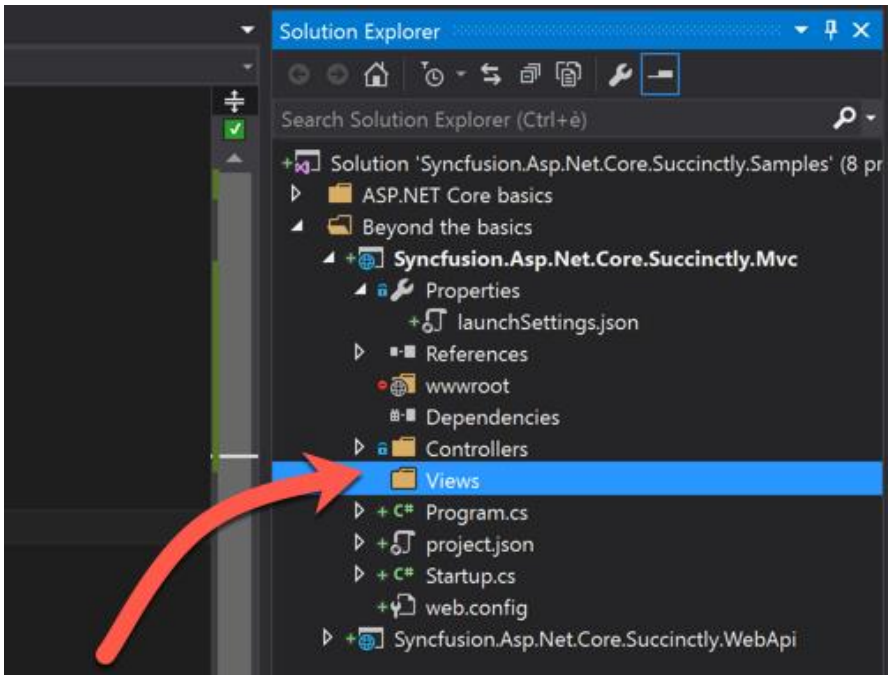


Figure 5-8: The Views Folder

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. Visual Studio 2019 makes it really simple to do.

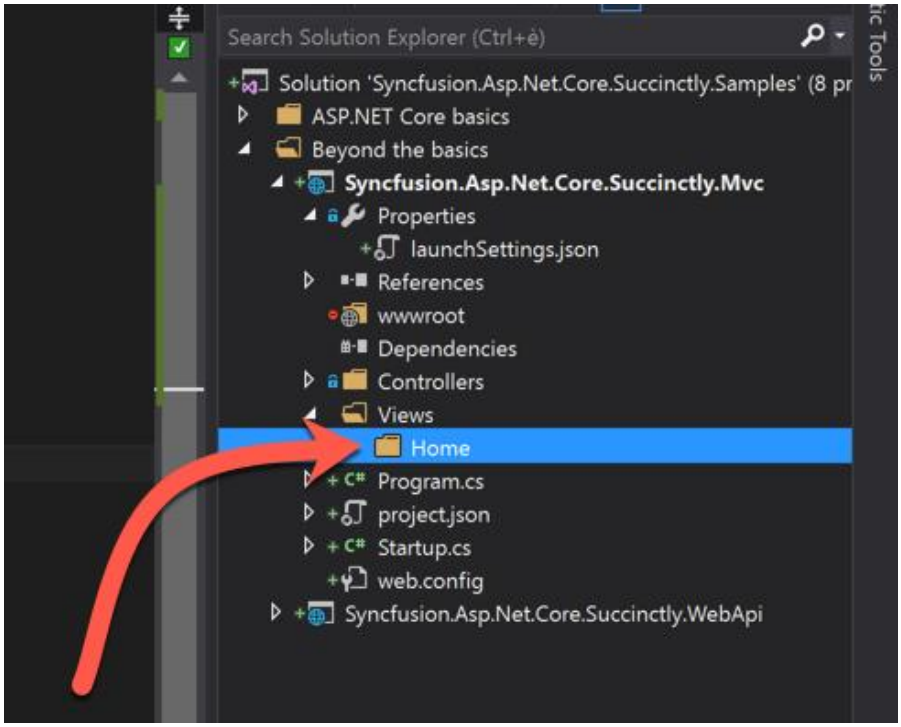


Figure 5-9: The Views Folder for the Home Controller

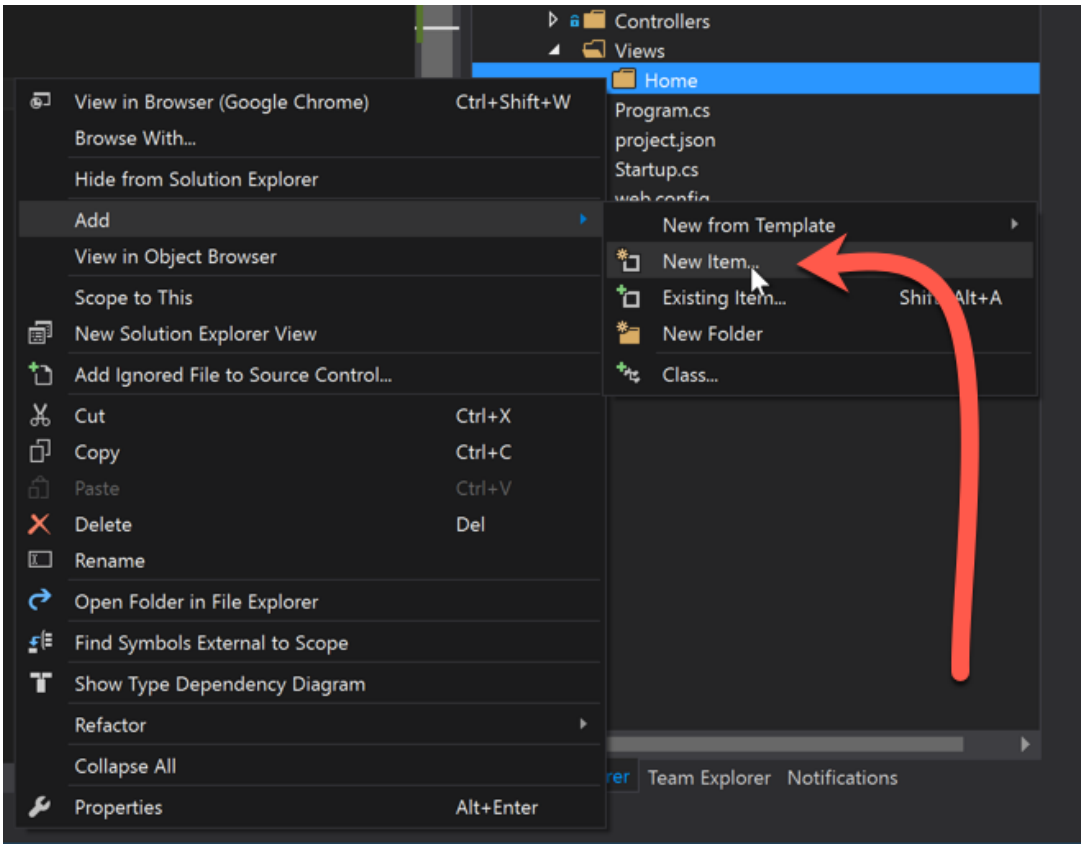


Figure 5-10: Adding the View-1

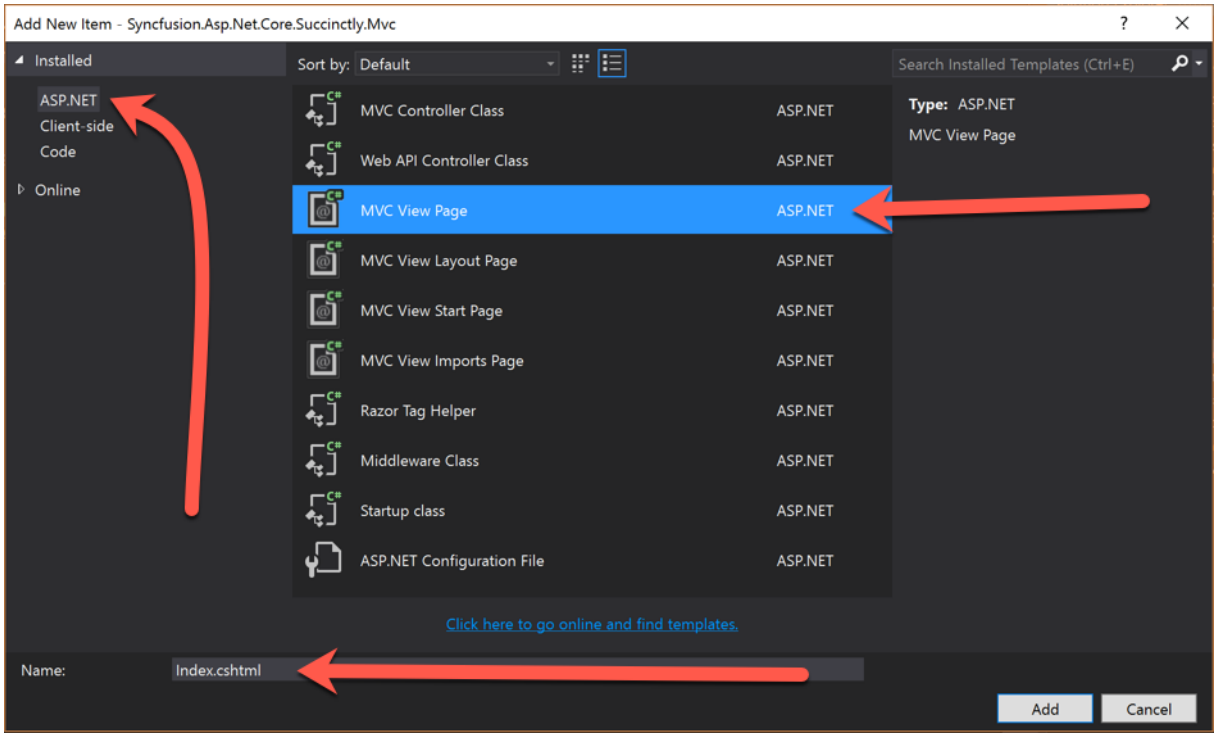


Figure 5-11: Adding the View-2

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 here.

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 5-14

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

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

Figure 5-12: 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 elaborate and to use something from the server, you have to go back to the controller and send the data to the view.

Code Listing 5-15

```
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 =
"Lattanzi", Twitter = "@imperugo"},
                new User() {Id = 2, Firstname = "Simone", Lastname =
"Chiaretta", Twitter = "@simonech"},
            };

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

Notice that we have reused the class **`User`** 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 5-15

```
@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 5-16

```
@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>
```

The output should be something like this.

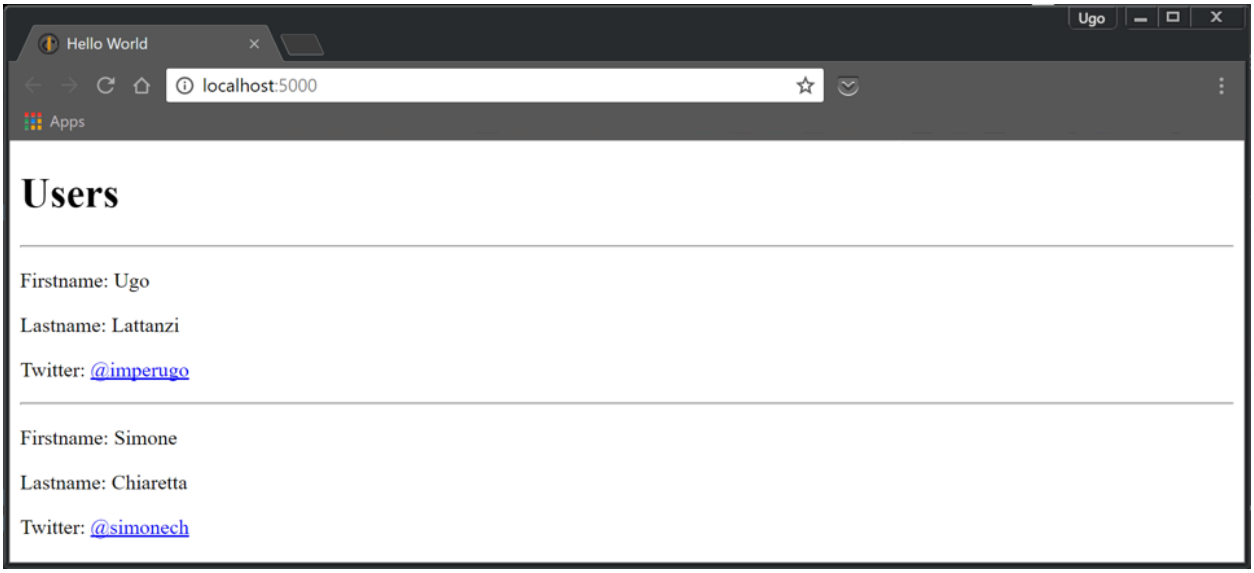


Figure 5-13: Server-Side Page Rendered Using MVC for a List

Routing

You just saw a small introduction to ASP.NET MVC. Notice that **`HomeController`**, combined with the **`Index`** action, handles 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 the following code.

Code Listing 5-17

```
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.AddControllersWithViews();
            }

            public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
            {
                  if (env.IsDevelopment())
                  {
                        app.UseDeveloperExceptionPage();
                  }

            app.UseEndpoints(endpoints =>
            {
                 endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{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 the following URL, the output should be the same:

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

In you want to manage a URL like **`http://localhost:5000/products`**, it's enough to 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 very much 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 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 HTML Razor syntax:

```
@Html.TextBoxFor(m=>m.FirstName, 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`**. 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.

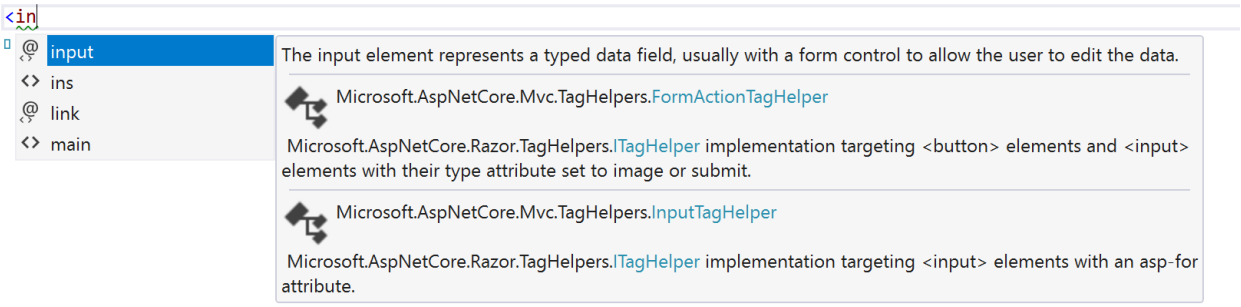


Figure 5-14: Visual Studio IntelliSense for HTML Tags

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.

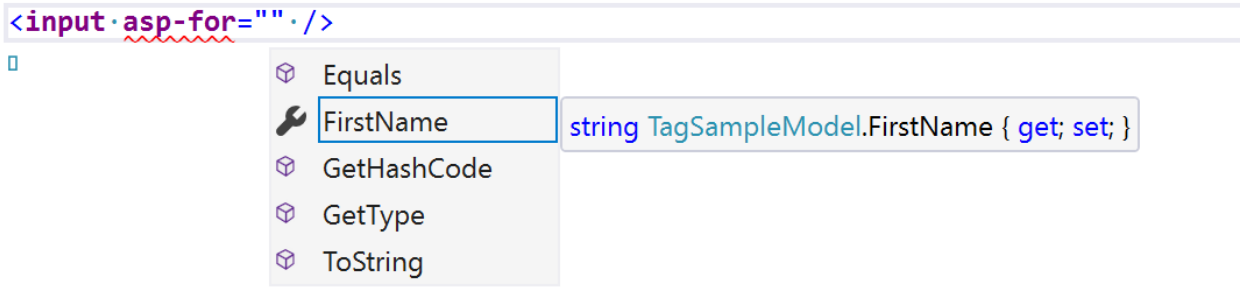


Figure 5-15: Visual Studio IntelliSense for MVC Model

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

```
<input class="form-control" />

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

Figure 5-16: Difference Between the Two Tags

In the first line, the element is brown with attributes in red. In the second, the element is purple with the special attribute also in purple (the normal attribute is still red). 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 re-implementing 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 link to controllers and actions, manage cache, render different HTML elements based on the environment, and manage script fallback or CSS files. You can see some of these tag helpers used in the View\Shared_Layout.cshtml file in the default project template.

Code Listing 5-18

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - ViewHelpers</title>
    <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index">ViewHelpers</a>
                <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-
sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
```

```
    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - ViewHelpers - <a asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @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 the following.

Code Listing 5-19

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

And creates a working a tag, like this.

Code Listing 5-20

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

Start by creating a **UrlTagHelper** file inside the MVC project. The name is important, as it is what defines the name of tag (in this case, **`url`**). 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**: Contains information on the current execution context (even if it is rarely used).
- **output**: Contains a model of the original HTML tag and its content, and it 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 it. This is done by adding a reference in the _ViewImports.cshtml file.

Code Listing 5-21

```
@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 5-22

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

namespace MvcSample.TagHelpers
{
    public class UrlTagHelper: TagHelper
    {
        public override void Process(TagHelperContext context,
TagHelperOutput 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 (**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 5-23

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

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.

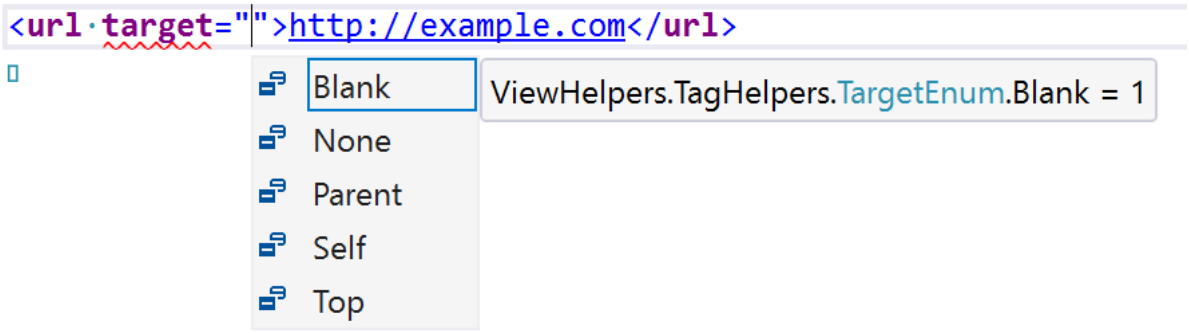


Figure 5-17: HTML Property Suggestion

This is the code of the updated tag helper.

Code Listing 5-24

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

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.ToString().ToLowerInvariant());
            }

        }
    }
}
```

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

View components

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

A partial view 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 of both view and class with the logic.

If you used a child action in previous versions of ASP.NET MVC, view components more or less solve the same problem in a more elegant way, as their execution doesn’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, and conditional login panels.

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`** method 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 5-25

```
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 5-26

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

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

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

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

Code Listing 5-27

```
@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. All view components are also registered as tag helpers with the prefix **vc**.

Code Listing 5-28

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

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

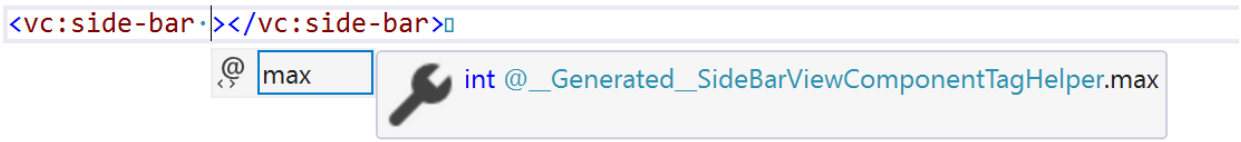


Figure 5-18: IntelliSense for Tag Helpers

Razor Pages

If you have to create dynamic pages but you don't want to use MVC, or you are looking to use a different approach, ASP.NET Core 2.x introduced a new way to create dynamic markup. It is called Razor Pages, and it uses the powerful Razor engine, without the part related to the model and controller that you already saw in the MVC section.

It is built on top of MVC, but its services must be registered specifically, similarly to how the registration of Web API was different from ASP.NET MVC.

Code Listing 5-29

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

    public void Configure(IApplicationBuilder app)
    {
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Now the application is almost ready to host some Razor pages, but in order to do so, you have to follow some conventions.

The first one is where the pages are located. You have to put them into a folder called **Pages** in the root of your application. To help with this, if you are using Visual Studio, you have a helpful context menu when you right-click on the folder.

Follow these steps to create your first Razor page.

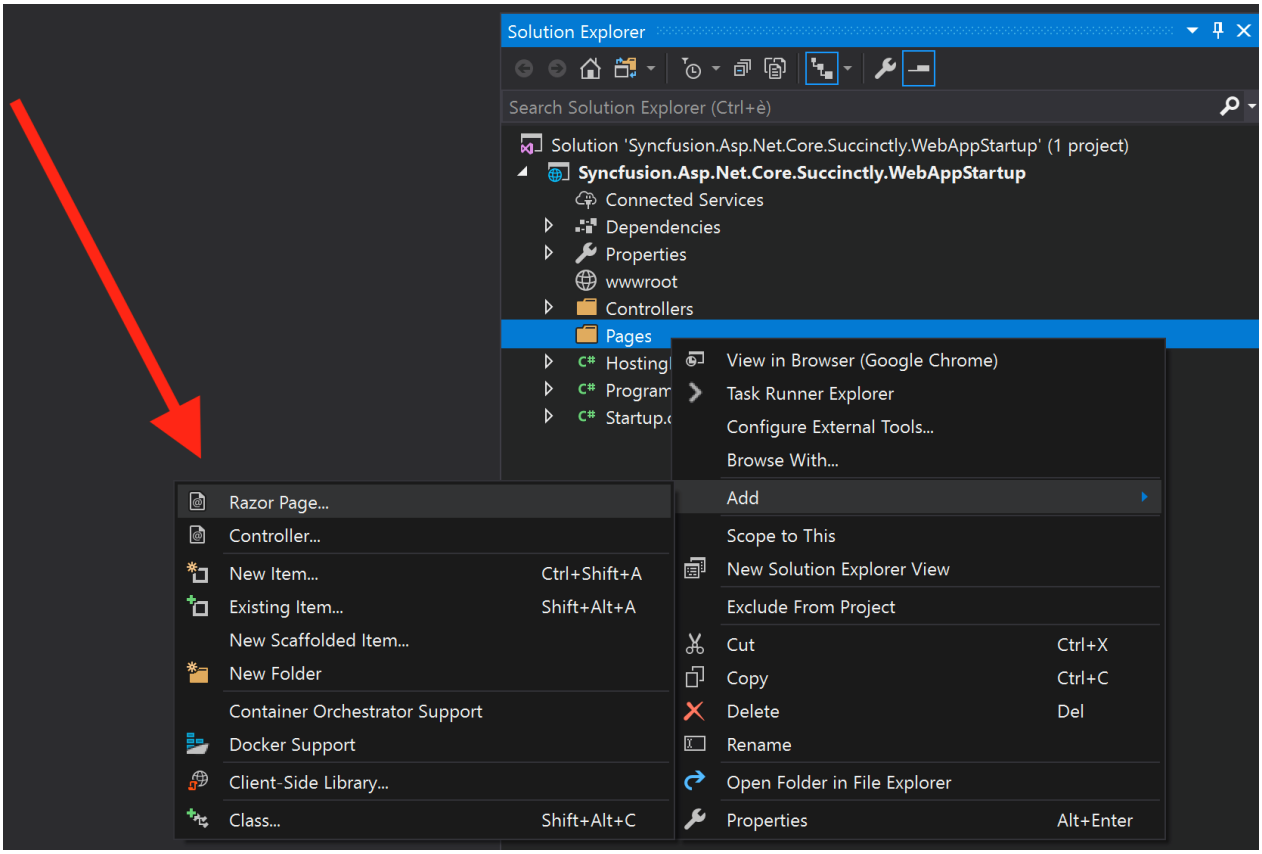


Figure 5-19: "Add Razor Page" Menu Item

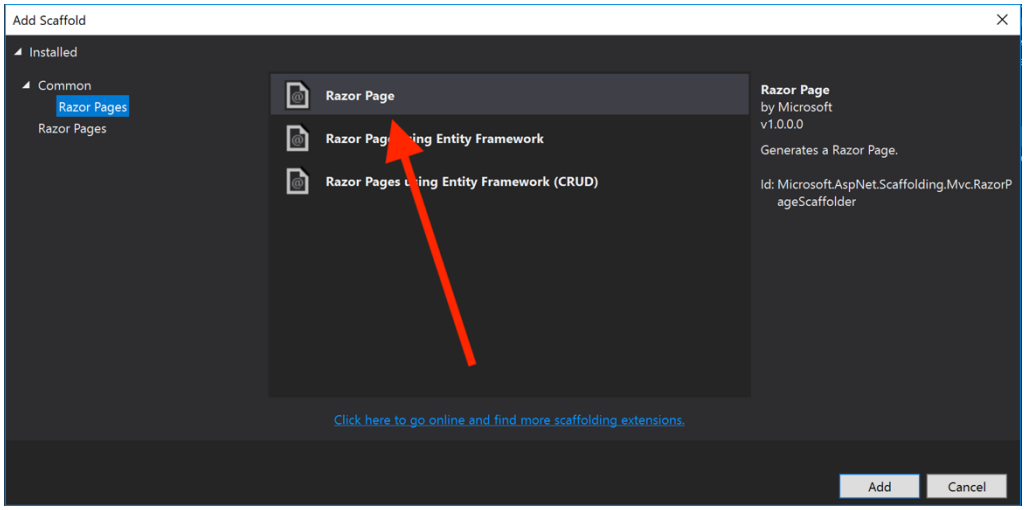


Figure 5-20: Add Scaffold Page Dialog

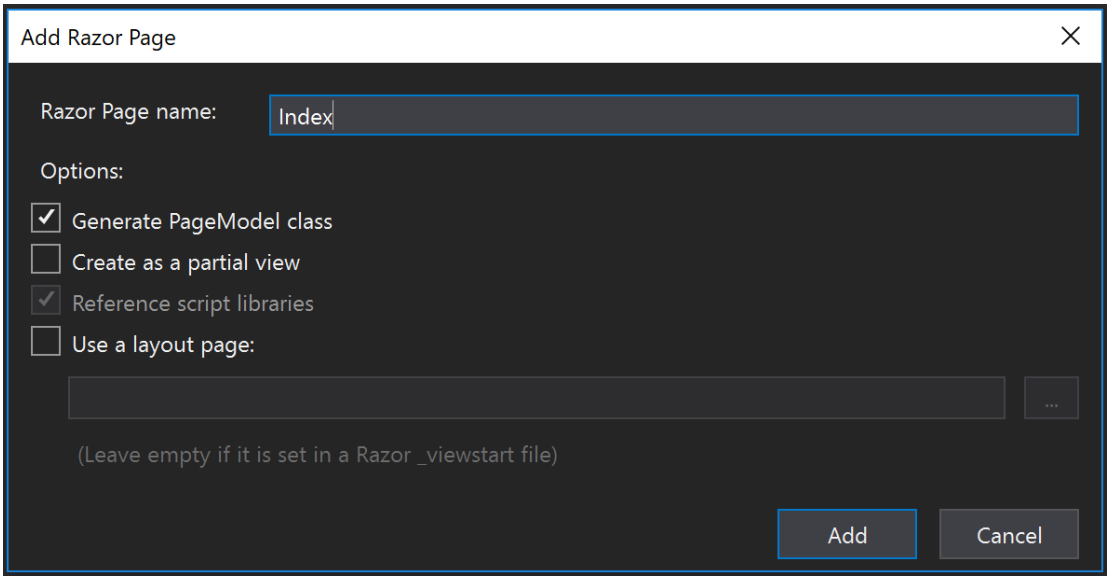
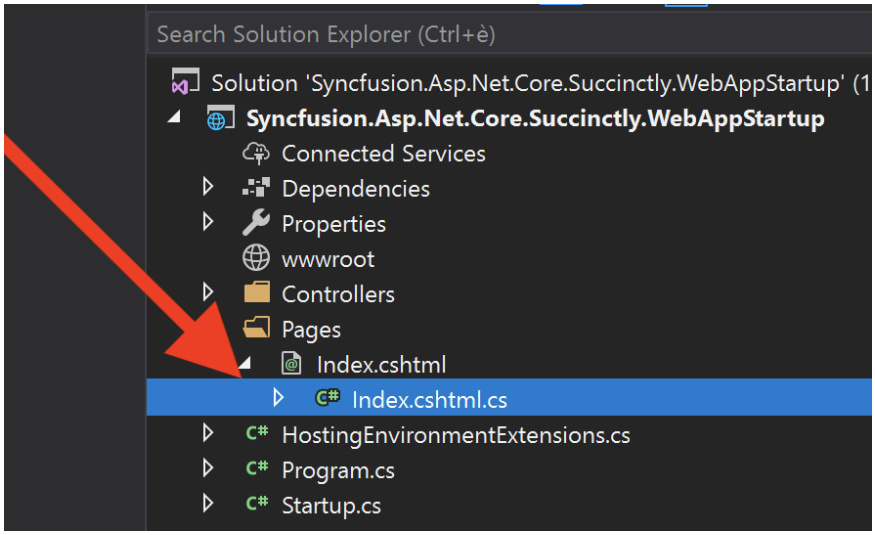


Figure 5-21: Add Razor Page Dialog

After going through these steps, an empty page is created in the **Pages** folder. If you now run the project, you can see your page in the browser. The **`Index`** page shows up by default; otherwise, you have to write the name of your page without the extension in the URL http://localhost:5000/myRazorPage.

Before going on and creating something more useful than a blank page, go back to the Solution Explorer and take a look at the file called **Index.cshtml.cs**.

It is somewhat reminiscent of the code-behind used with Web Forms, but it is not the same. In fact, if you open the .cs file, you don't have all the things you had with Web Forms (**`Page_Load`** and so on). This .cshtml.cs file contains the **`PageModel`** class.



5-22: PageModel class

If we have to make a comparison, we could say that this file is like a combo between the MVC controller and its model. If you read the MVC part, you surely remember the logic behind the controller, where you create an array of users and return that model to the view. With a Razor page, it's not much different.

Code Listing 5-30

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Syncfusion.Asp.Net.Core.Succinctly.WebAppStartup.Pages
{
    public class IndexModel: PageModel
    {
        public User[] Users {get;set;}

        public void OnGet()
        {
            Users = new[]
            {
                new User() {Id = 1, Firstname = "Ugo", Lastname =
"Lattanzi", Twitter = "@imperugo"},
                new User() {Id = 2, Firstname = "Simone", Lastname =
"Chiaretta", Twitter = "@simonech"},
            };;
        }
    }
}
```

Now go back a few pages and compare it to the code in Code Listing 5-6; you will notice a few changes. The code is inside an **OnGet** method, while with MVC it is implemented in the **Index** action method of the **HomeController**. Also, the **Index** method had the attribute **[HttpGet]** to indicate which action has to be executed for an HTTP **GET** verb. As you can imagine, with Razor Pages, the name of the method serves the same function as the attribute used with ASP.NET MVC. The **OnGet** method is executed for **GET** requests. If you have to handle **POST**, the method would be **OnPost**, and so on for all other verbs.

So, the name of the method changes, as well as the way to pass the model to the view. In MVC, you create a **ViewModel** object and return it with the **View** method, while with a Razor page, you just assign it to a property of the **PageModel**.

The Razor markup is exactly the same because the only difference between the two frameworks is how the view is instantiated.

Code Listing 5-31

```
@page
@model Pages.IndexModel
@{
    Layout = null;
```

```
}

<!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.Users)
        {
            <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>
```

The routing is based on the physical location of the file inside the **Pages** folder, so it's slightly different from how it works in ASP.NET MVC.

Table 3: Sample Files and Their URLs

File name and path	Matching URL
/Pages/Index.cshtml	**/** or **/Index**
/Pages/Contact.cshtml	**/Contact**
/Pages/Products/Index.cshtml	**/Products** or **/Products/Index**
/Pages/Info/Contact.cshtml	**/Info/Contact**

We can summarize by saying that the folder **Pages** never appears in the URL, and the structure of the URL is made by the structure of folders and files created inside the **Pages** folder.

This is only an introduction to Razor Pages, just like it was for ASP.NET MVC. If you want to go more in depth into the topic, you can do so with the official documentation.

Single-page applications

Single-page applications (SPAs) are a very hot topic, and it is now possible to create both Angular-based and React-based applications directly from within Visual Studio, and also from the **`dotnet`** CLI.

With Visual Studio you have the option of choosing the template inside the **New ASP.NET Core Web Application** window, as shown in Figure 5-23.

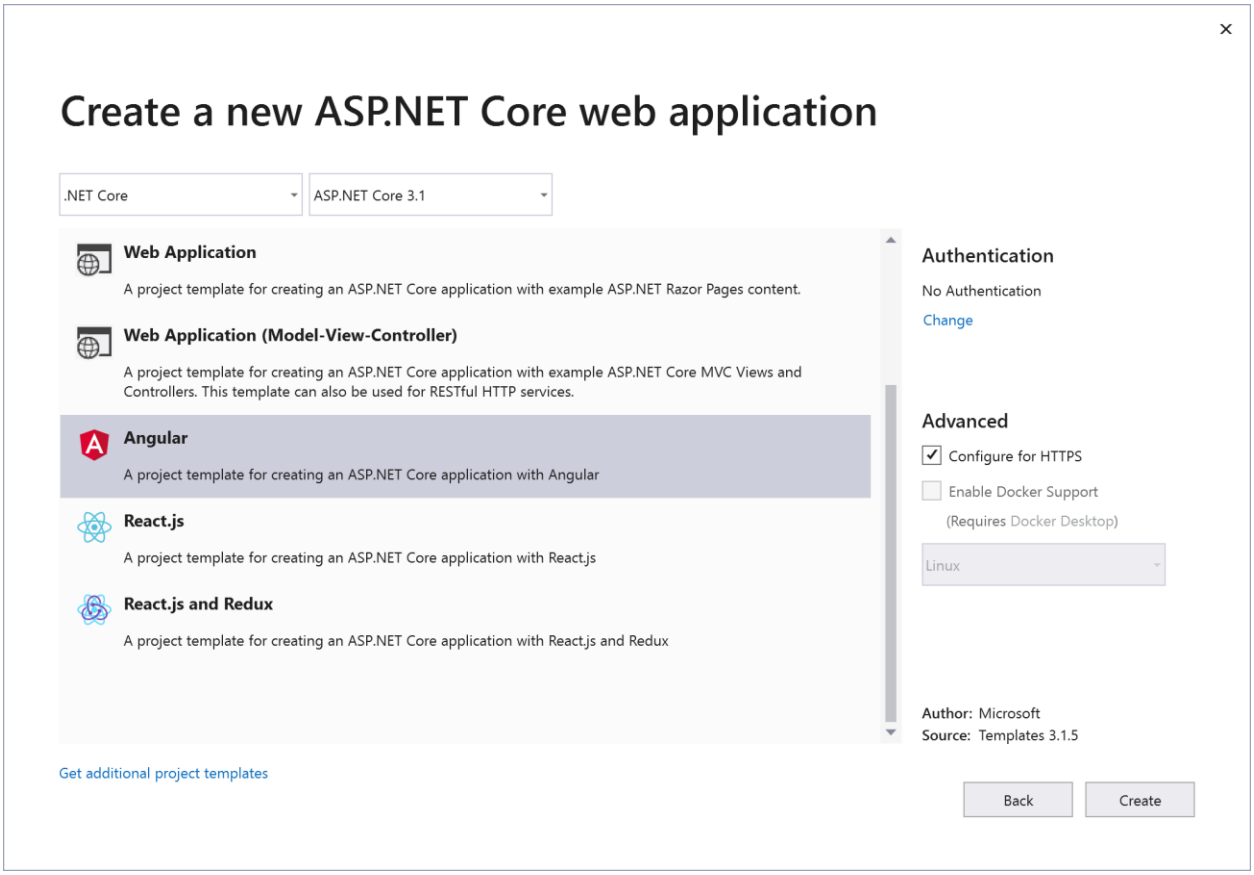


Figure 5-23: New ASP.NET Core Web Application

With the **`dotnet`** CLI, you can create an SPA by choosing one of the SPA templates:

- **`angular`**
- **`react`**
- **`reactredux`**

Whichever method of creation you choose, you get an application that uses a feature called JavaScriptServices.

JavaScriptServices

JavaScriptServices is a set of features that helps use client-side technologies within ASP.NET Core applications. For example, it can be used, among other things, to run JavaScript on the server interoperating with Node.js, to automatically build client-side assets using Webpack, and to simplify the usage of various SPA frameworks (like Angular and React).

The Angular application template

By choosing the Angular template, you get a sample application with a few pages, tied together with Angular routing, one of which connects to a Web API. Figure 5-24 shows the structure of the application.

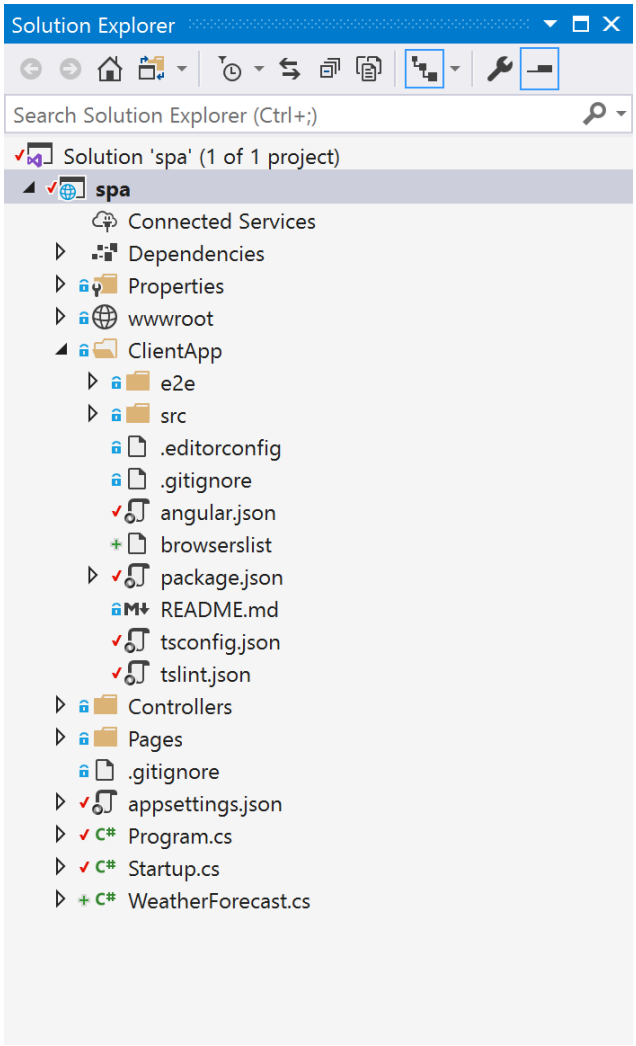


Figure 5-24: SPA with Angular and ASP.NET Core

The **ClientApp** folder contains the whole Angular application, created using the Angular CLI. This way, you can work on Angular as if you were using a normal client-only Angular application, even without using Visual Studio. And if you are used to building Angular applications using the CLI, you can still do it. In debug mode, the application is even served using the `ng serve` command of the Angular CLI, so that changes to the Angular files are automatically processed and updated on the client application.

The **Controllers** folder contains the controllers, like any other Web API application.

The **Pages** folder contains the few server-side pages that might be needed, like, in the sample, the Error page. They are just Razor Pages because such a simple page doesn't require the full MVC structure to be created.

With the structure set up like this, now you can start easily developing your SPA, as the ASP.NET part is just like any other ASP.NET Web API.

SignalR

Sitting next to SPAs, there is another class of applications, called real-time applications, where data is pushed from the server to the browser without any operation performed by the user.

This kind of interaction is possible thanks to a feature of HTML5, implemented nowadays in all modern browsers, called a WebSocket, which allows a full duplex connection between server and client.

ASP.NET Core has a framework that helps you build these kinds of applications in an easy and reliable way. This framework is called SignalR Core. Unfortunately, real-time applications are pretty complicated, and even a simple introduction would fill half of this book. However, you can find more information on SignalR in the official ASP.NET Core documentation.

Blazor server and Razor components

With .NET 3.x, Microsoft released a new framework for developing client-side interaction directly in C# and .NET Core, without the need to know JavaScript. This new framework is called Blazor.

Blazor can run directly in the browser, thanks to the fairly new web standard called WebAssembly, and can also run on the server.

Yes, you read correctly: Blazor allows you to write an application in C# with .NET Core that, while running on the server, can interact with the browser like any SPA can do.

But it is not as crazy as it sounds. It is actually a pretty clever solution.

Blazor server hosting

Before explaining how the Blazor server works, it is important to understand how all client-side, JavaScript component-based frameworks, like Angular or React, work.

The components are rendered into an in-memory representation of the browser's DOM (document object model), called the render tree. Whenever something changes in the application, like a button is pressed or some new data arrives, this in-memory representation changes. It is compared against the current document model by a rendering engine, which updates only the parts of the tree that have actually changed. This is done to make the update faster and keep the UI responsive. As you can see, there are two subprocesses in these client-side frameworks: one that takes care of updating the components (the actual application), and one that works behind the scenes and just updates the UI.

Blazor took this concept and enhanced it by splitting the rendering engine into two additional parts: one that takes care of creating the rendering tree and computing which parts have to be updated, and one that applies these changes to the DOM.

In the client version of Blazor, both parts of the rendering engine are part of the .NET Core-based Blazor runtime running inside the WebAssembly, effectively behaving like the other JavaScript libraries.

In the server version, the part of the rendering engine that computes the tree and its changes runs in .NET Core, on the server. The changes are pushed to a small JavaScript library running inside the browser, whose role is just updating the DOM based on the information it receives from the server, via SignalR. And this same channel is also used by the client to pump the events (button clicked, text changed, and so on) to the server.

This separation is what makes it possible to write a Blazor application and later decide to move it between client and server.

Blazor server app

Now that you have a basic understanding of how Blazor works, let's explore how a Blazor server application is made by creating it with Visual Studio.

Blazor is a very wide topic, and as done for SignalR, the next section is just a basic introduction. If you are interested and want to learn more, I recommend you read the book *Blazor Succinctly*, also published by Syncfusion.

Blazor applications can be created both using Visual Studio and via the dotnet CLI. In Visual Studio, you start from the **New Project** menu item. Unlike all the other projects in this book that started from the ASP.NET Core Web Application template, for Blazor you need to select the specific **Blazor App** template.

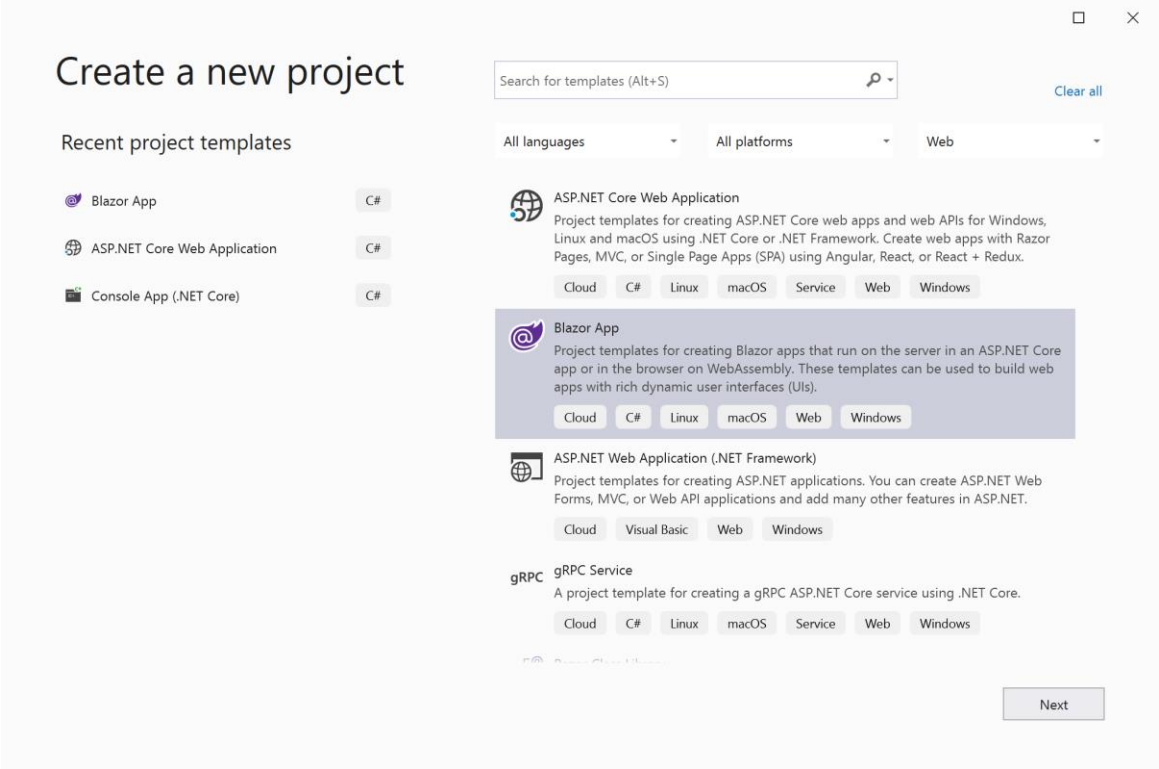


Figure 5-25: New Project Dialog

Once you've specified name and location for the new app, you have to choose between client and server app.

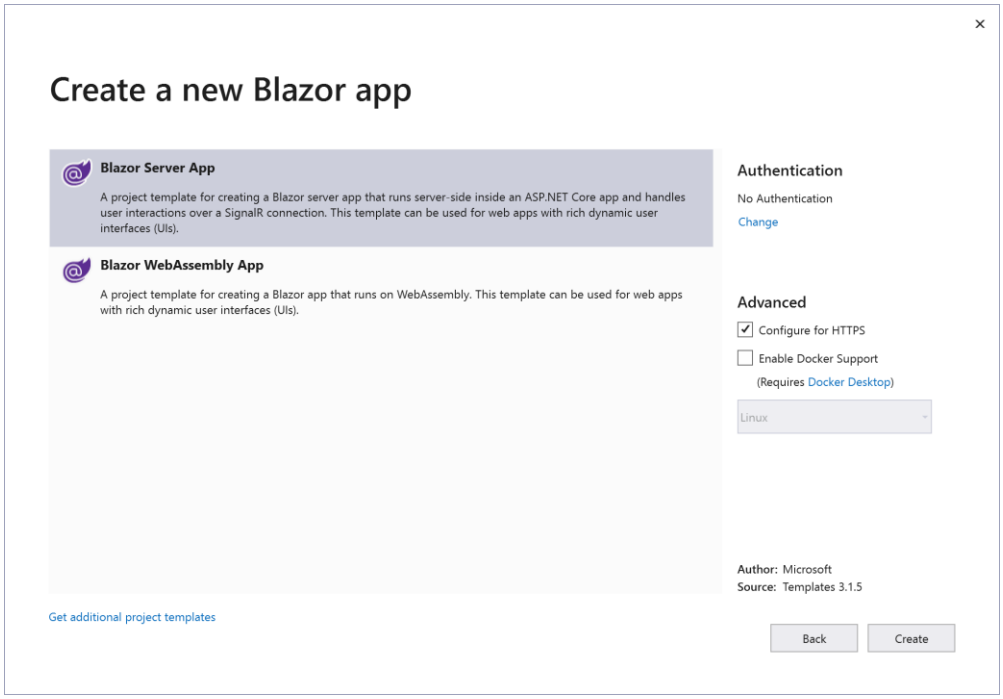


Figure 5-26: New Blazor Template Dialog

The project generated from the template is similar to the one of the Angular template: a simple SPA with navigation, a simple counter, and a fake weather information page.

The structure of the project is also similar.

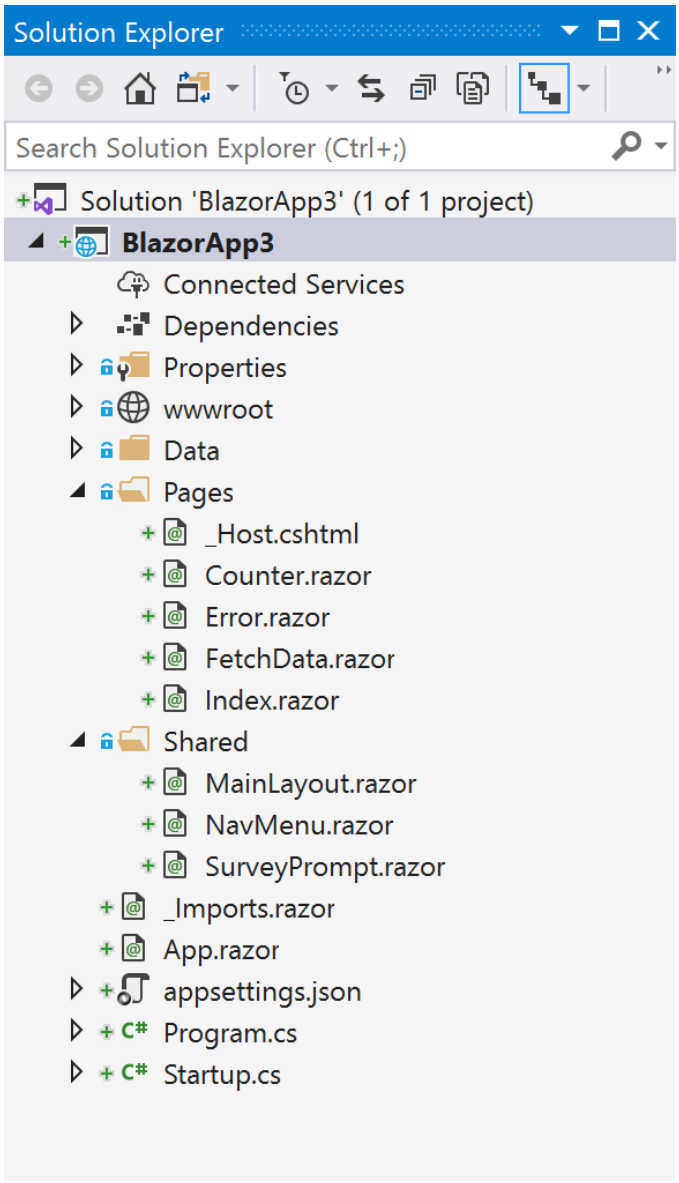


Figure 5-27: Blazor App Project Structure

Let's explore each of the elements of the application.

The first step is to instruct the runtime that this is a Blazor app. This is done by adding a few lines of code in the Startup class.

First, add the Blazor service in the **`ConfigureServices`** method and then configure the SignalR Hub used by Blazor to communicate with the client. This is inside the endpoint configuration lambda in the **`Configure`** method.

Code Listing 5-32

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddServerSideBlazor();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
...
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });
}
```

The next step is specifying inside the _**Host.cshtml** file, the name of the Razor application to execute.

Code Listing 5-334

```
<app>
    <component type="typeof(App)" render-mode="ServerPrerendered" />
</app>
```

This instructs the runtime to execute the root component called **App**. It is defined in the **App.razor** file. Notice the .razor extension: this is what defines our Razor component.

Code Listing 5-34

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData"
DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

The root component defines, among other things, the routing of the application and the default master layout view used by all components that do not define one explicitly.

As soon as the application is requested via the browser, the **Index** component gets rendered, inside the default layout.

Code Listing 5-35

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

The first line is what defines the route to which the component responds, in this case **`/`**.

You'll also notice, at the bottom of the file, that the component includes another component, called **`SurveyPrompt`**, with a **`Title`** parameter.

Razor component

After the overview of the sample application provided by Visual Studio, let's see in detail how a Razor component is done.

Razor components are very similar to the Razor views used in ASP.NET MVC or Razor Pages, and they obviously use the Razor syntax. In addition, they can also contain a **`@code`** block that can contain fields, properties (used as input parameter), and functions (that also act as event handlers).

The following example is an enhanced version of the counter component, where the increment can be passed as parameter and the initial value is a random number.

To create the Razor component, use the **Add** > **Razor Component** menu item, as shown in Figure 5-28.

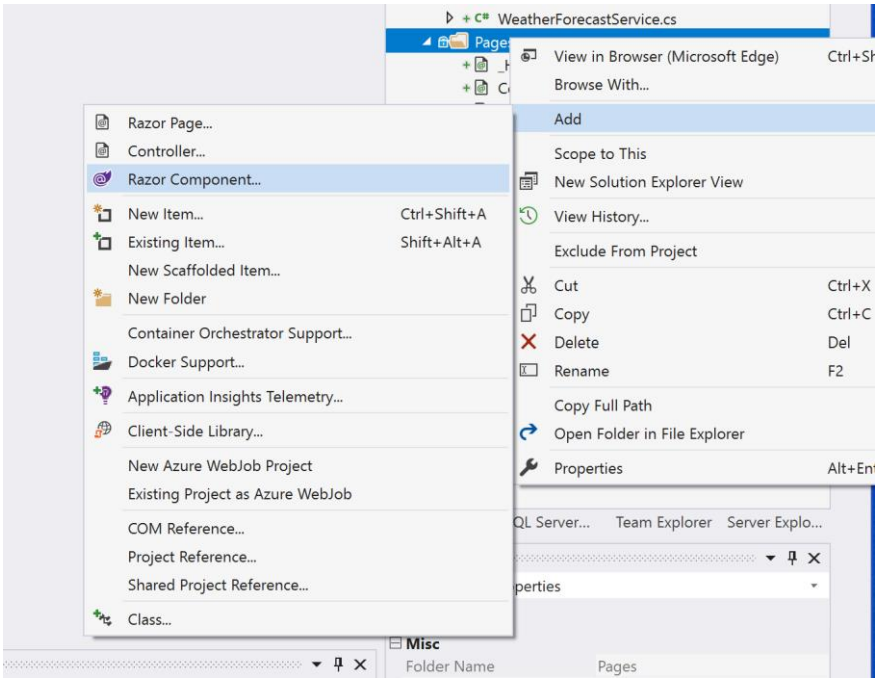


Figure 5-28: New Razor Component

The following listing shows the complete code of the component.

Code Listing 5-36

```
@using System.Security.Cryptography

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount += IncrementAmount;
    }

    protected override void OnInitialized()
    {
        currentCount = RandomNumberGenerator.GetInt32(0, 10);
    }

    [Parameter]
    public int IncrementAmount { get; set; } = 1;
```

```
}
```

From top to bottom, we can see:

- **@using**: Define the **using** statement, like any Razor-based file.
- **@onclick="IncrementCount"**: This registers the function **IncrementCount** as handler for the **onclick** event.
- **IncrementCount**: The event handler that was registered previously.
- **OnInitialized**: The event handler for one of the application lifecycle events.
- [**Parameter**]: Defines a parameter that can be passed when the component is included in the parent.

Finally, you can include this component in any page by simply using its file name.

Code Listing 5-37

```
<RandomCounter IncrementAmount="3" />
```

Now, browsing to the app, the counter will start from a random value between 0 and 10, and every time you click the button, it will increment the value by 3.

Worker service

All the projects created so far are responding to user interaction and work only when someone requests them. With ASP.NET Core, you can also create background tasks that run indefinitely by implementing a hosted service.

At its core, a worker service is simply a class that implements **IHostedService** interface, with just two methods: **StartAsync**, which is used to start the background task; and **StopAsync**, to gracefully terminate and dispose of all resources in case of a normal termination of the process.

To simplify the most common start and stop behavior, the framework provides the **BackgroundService** base class. This class only has one method to override, **ExecuteAsync**, which should contain a never-ending loop with the task to execute and a call to an **async** method (like IO operations or simply a delay) to allow other services to run.

A service worker application can be created, as usual, both via a project template in Visual Studio and via the **dotnet** CLI.

In Visual Studio, you have to look for the Worker Service project template, and with the **dotnet** CLI, you need to run **dotnet new worker**.

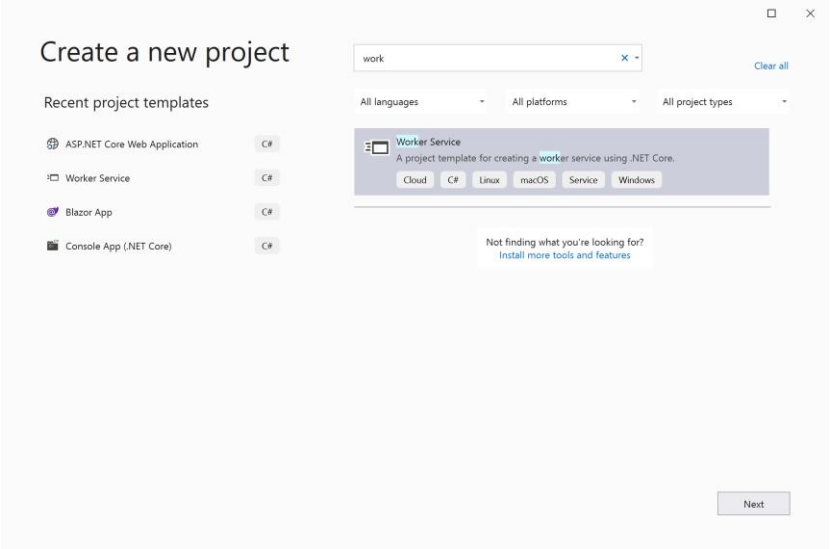


Figure 5-29: Worker Service Project Template

Once its created, the project contains just two files, and, unlike all other web projects, which are based on the **`Microsoft.NET.Sdk.Web`** SDK, it is based on **`Microsoft.NET.Sdk.Worker`**, which has a reduced footprint.

The first file is Program.cs, which contains the registration of the hosted service.

Code Listing 5-38

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((hostContext, services) =>
        {
            services.AddHostedService<Worker>();
        });
```

The second file, Worker.cs, contains a background service implement using the **`BackgroundService`** base class.

Code Listing 5-39

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

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

```
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

The whole logic is implemented in the **ExecuteAsync** method inside an infinite loop that is only terminated when a cancellation is requested. In addition, if you look at the constructor, you can see that DI is used also with this type of service.

You can also use a service worker inside a normal web application. Nothing changes in the way you develop either application. The only additional step is to register the hosted service inside the **ConfigureServices** method in the **Startup** class of the web application, after having registered your controllers and other services.

Code Listing 5-40

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddHostedService<Worker>();
}
```

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 and will demonstrate how to deploy apps and 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 experiences for both local servers and Azure.

How to deploy 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.

Self-contained applications (and not just DLLs) require a different approach for hosting them inside IIS. 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, we need to install a specific module that does the "reverse proxy" work and makes sure the application is running. This module is called **`AspNetCoreModuleV2`** 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 already 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.

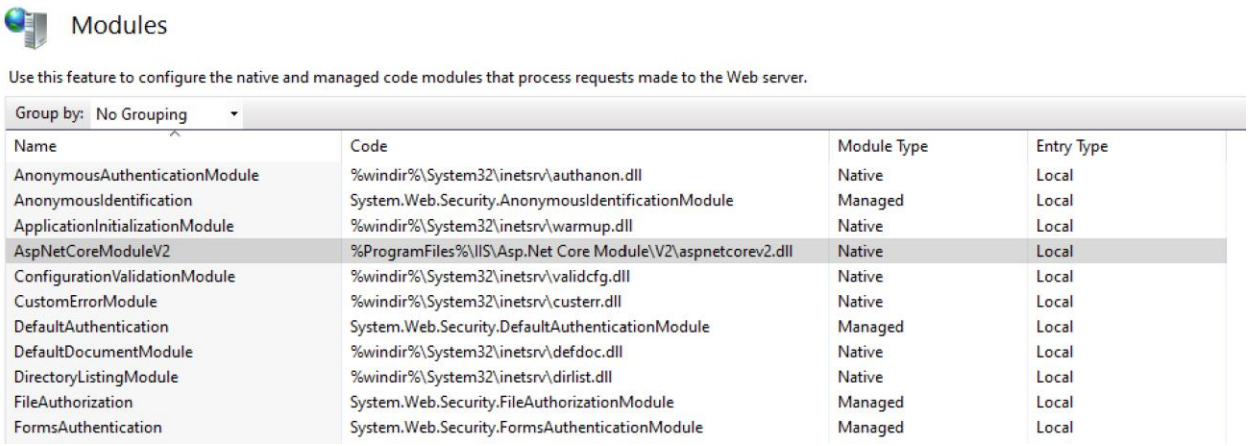


Figure 6-1: IIS Modules

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 shown in Figure 6-2.

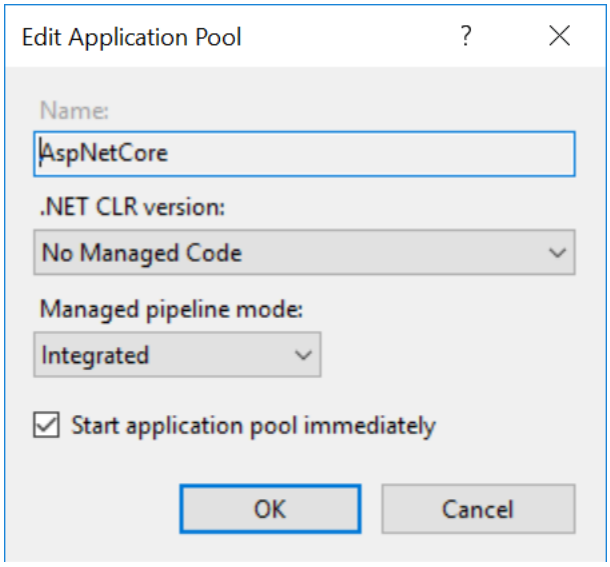


Figure 6-2: IIS Application Pool Basic Configuration

The configuration of the **`AspNetCoreModuleV2`** comes from the web.config file in the root of the folder.

Code Listing 6-1

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*"
          modules="AspNetCoreModuleV2"
          resourceType="Unspecified" />
      </handlers>
      <aspNetCore
        processPath="dotnet"
        arguments=".\PublishingSample.dll"
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\stdout"
        hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

Publishing the application is done using the **`dotnet`** command-line tool. 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 netcoreapp3.1
      --output "c:\temp\PublishFolder"
      --configuration Release
```

The **`publish`** operation also runs all the MSBuild targets and can, for example, run the bundling and minification of scripts and styles. It also automatically creates (or updates if one is specified) 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 ability to publish to a remote server and perform incremental updates. An alternative 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.

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, 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, even if you have to create new resources.

From the **Solution Explorer**, right-click the project name, and then select **Publish**.

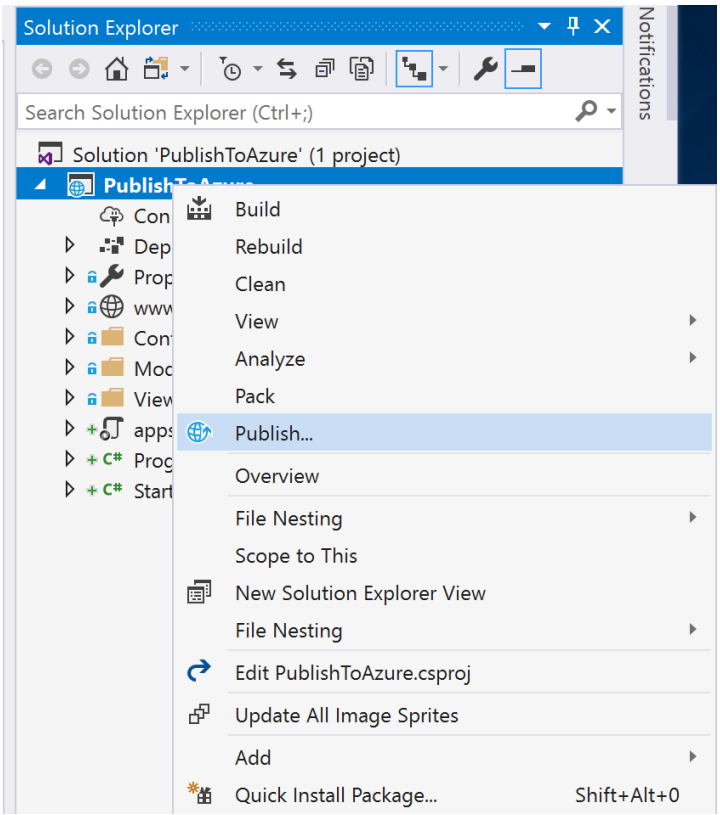


Figure 6-3: Publish Web Application Using Visual Studio

Next, you can choose how to publish your application. You can choose from among various targets, both local and remote, like Docker Registry or Azure.

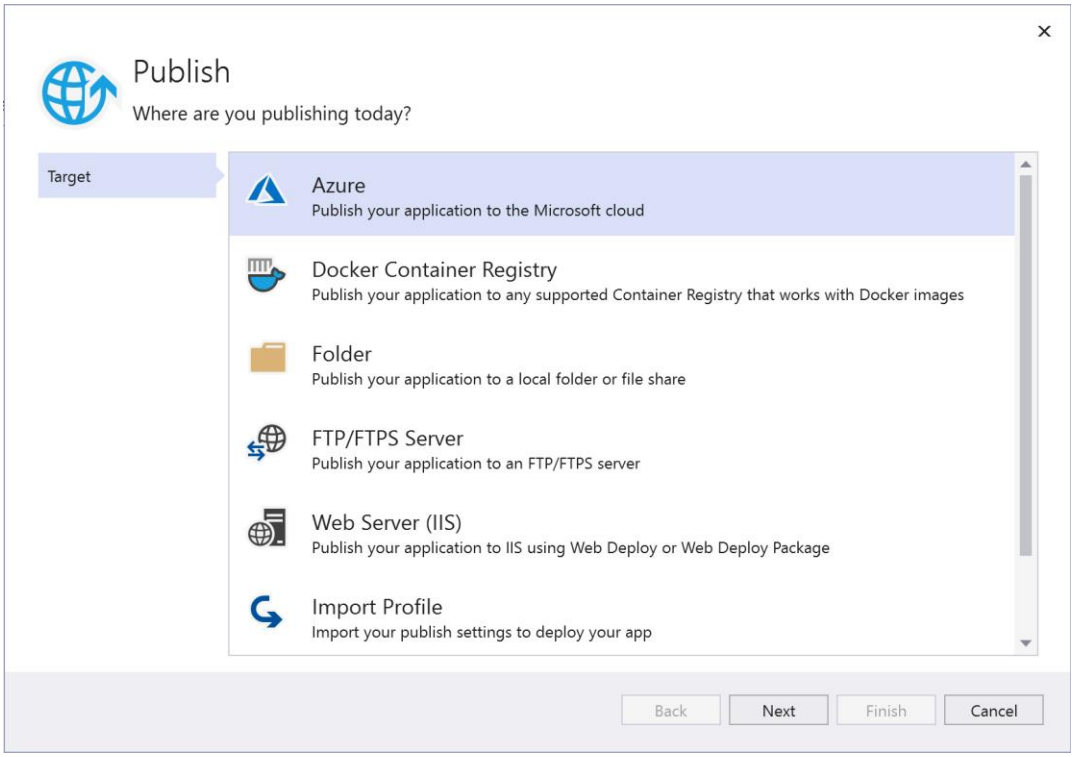


Figure 6-4: Select Publishing Target

When choosing Azure, you choose which Azure service to use. You can choose the “traditional” App Service (Windows or Linux), or you can also decide to publish your application as Docker image, directly as App Service Container, or just to the Azure Docker Registry.

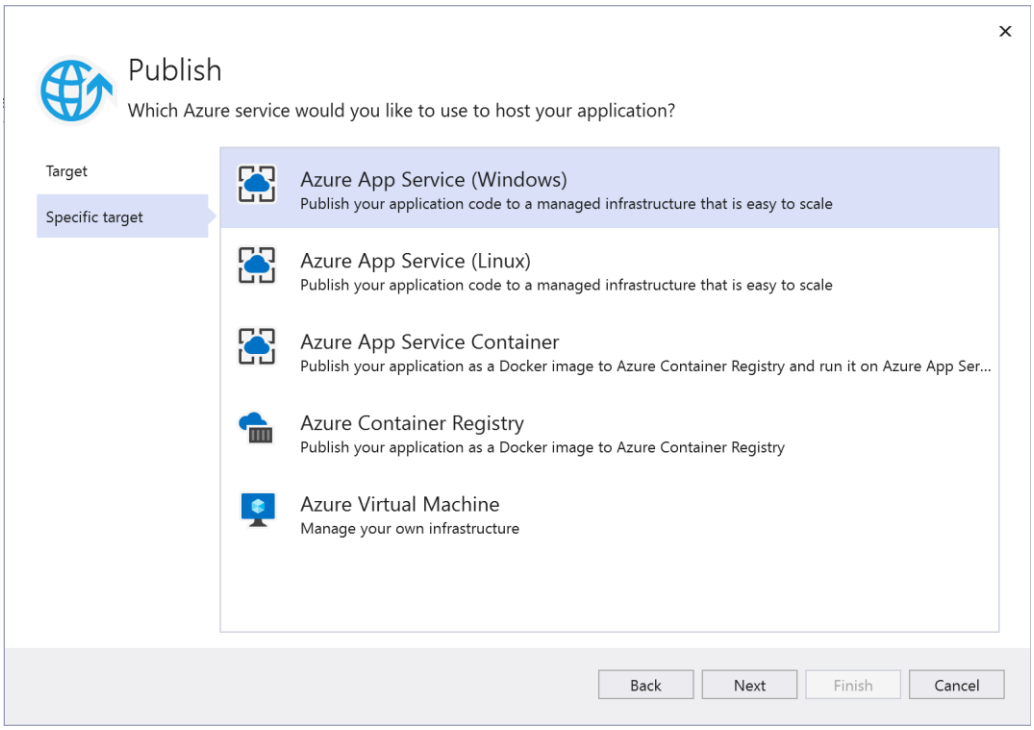


Figure 6-5: Select Azure Service to Use

Next, you can choose to reuse a previously created service, or you can create a new service directly from within Visual Studio.

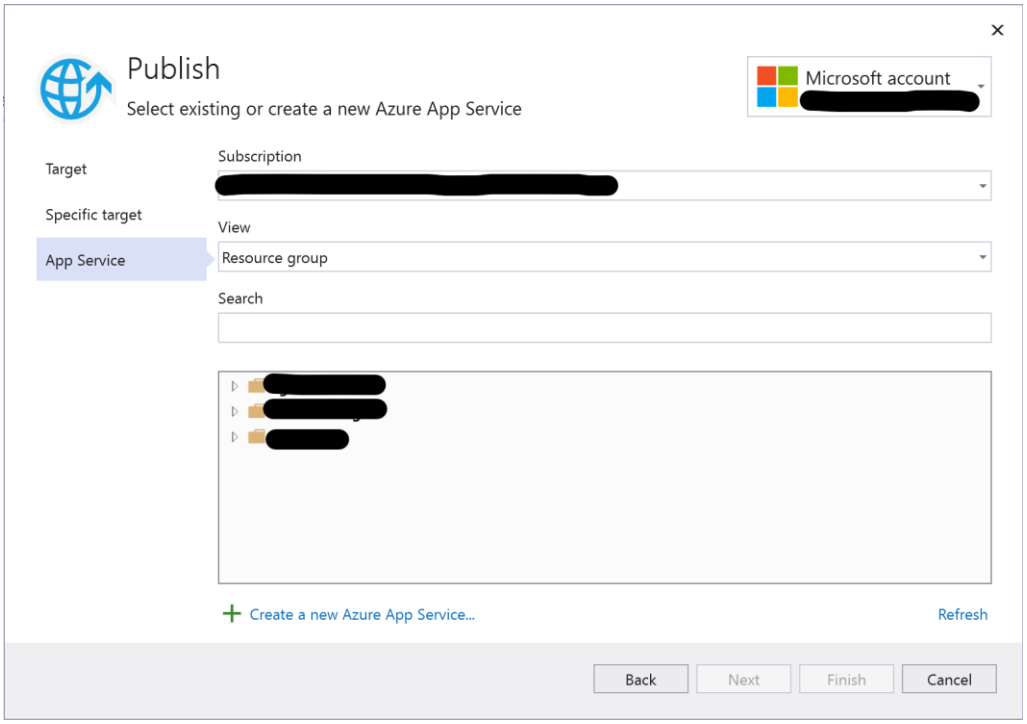


Figure 6-6: Select Existing or Create New Service

If you choose **Create a New Azure App Service**, in the next screen you can create an app service in which to host the web application. When running some tests, I recommend also creating a new resource group and a hosting plan (use the Free size), so that you don't interfere with any other apps you might already have on Azure. This way, once the test is complete, you can just delete the resource group, and everything will be deleted as well.

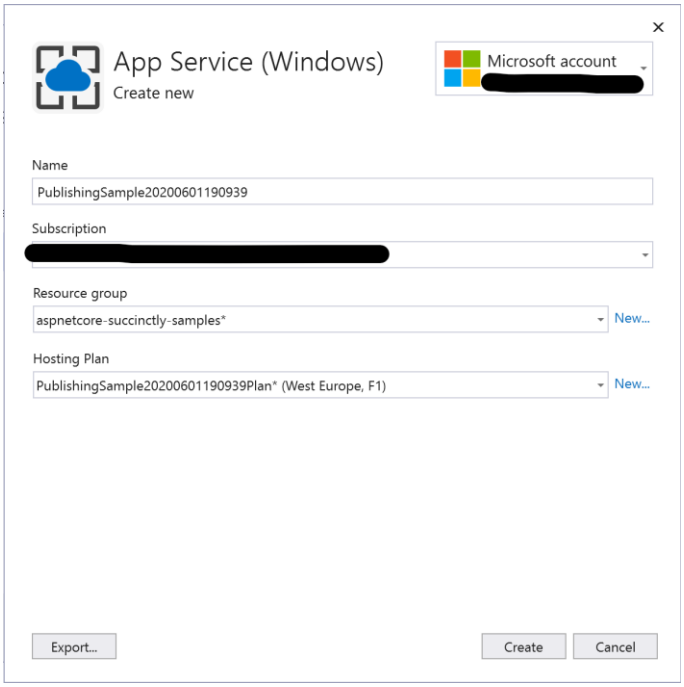


Figure 6-7: Create App Service

Now, click **Create** and wait for resources to be created on Azure. Once this is completed, you go back to the same screen as Figure 6-6, but with the newly created service already selected.

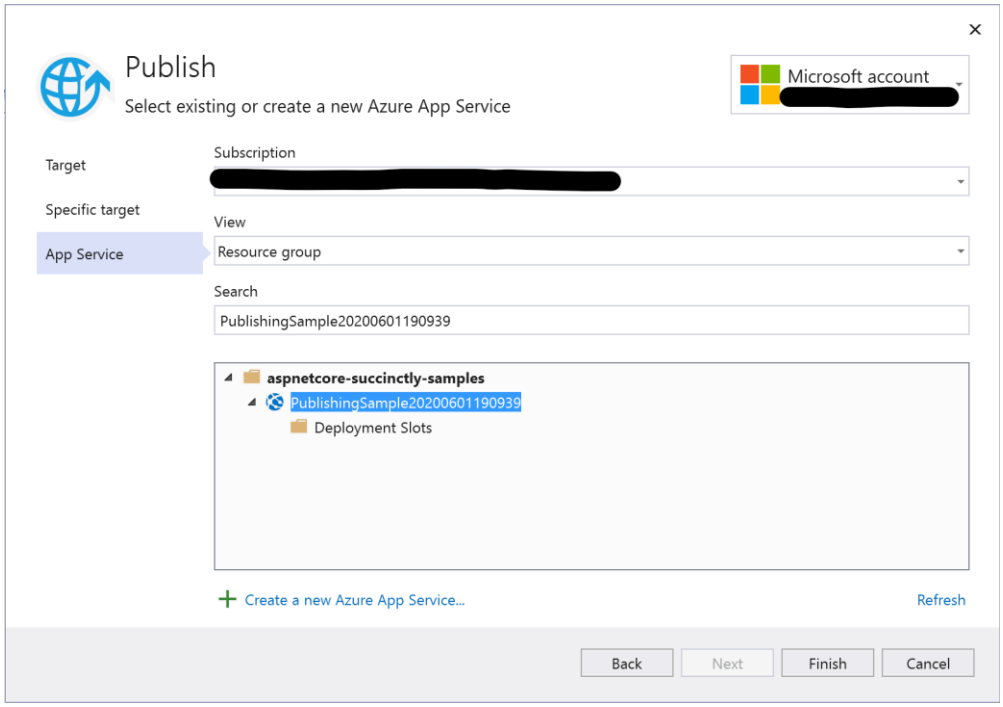


Figure 6-8: New Service Created

Now you can click **Finish**, and go to the last screen, from which you can change publishing configuration, add dependencies on other Azure services (like DBs, storage account, or Application Insights), and publish the application.

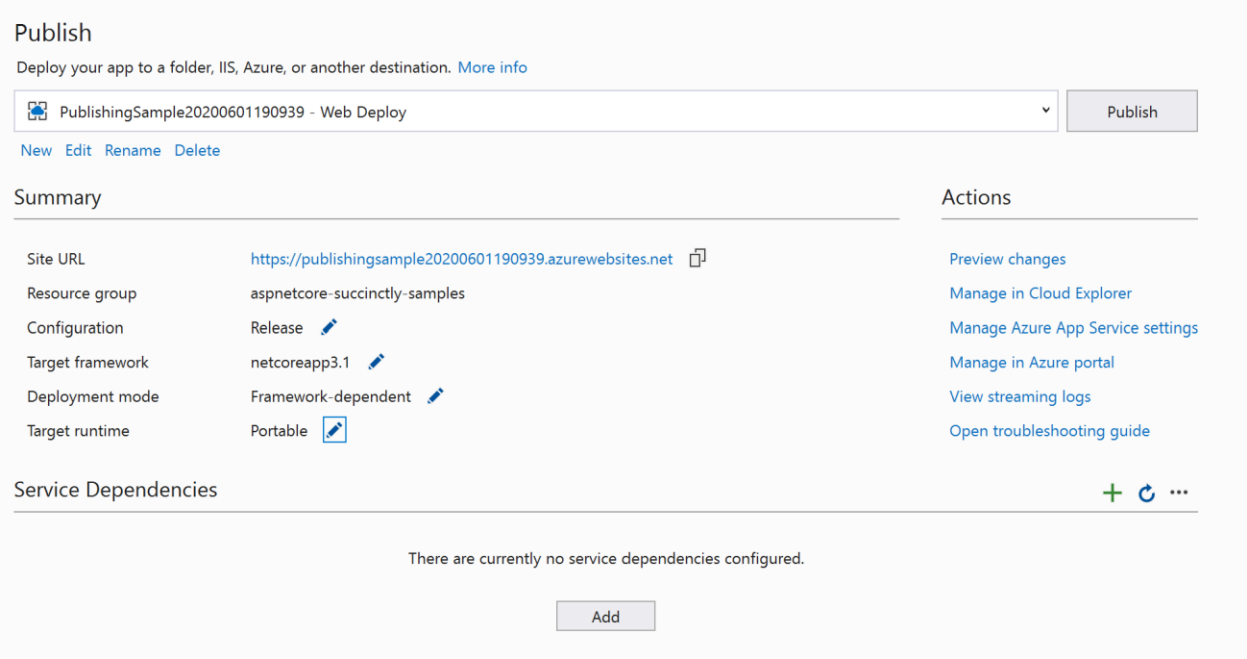


Figure 6-9: Publish to Azure

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.

Deploy to Azure via the command line

Publishing from Visual Studio is fine if you are just doing some tests, or for a one-off project. But if you are working on a project where you'll deploy multiple times, you might want to deploy via an automated process.

You could use the DevOps features of Azure, but you could also deploy directly from the command line by using the **`publish`** command.

First, you have to create a publishing profile as you would when deploying from within Visual Studio. Then, from the command line, you can type the following line (where **`WebProject.csproj`** is the name of your project and **`Azure`** is the name of your publishing profile):

```
dotnet publish WebProject.csproj /p:PublishProfile=Azure
/p:Password=<passwordOmitted>
```

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 this book, we said that ASP.NET Core is a cross-platform framework, but apart from a simple example in Chapter 3, 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 and to build and 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. Some of the commands available from the **`dotnet`** tool are:

- **`new`**: Creates a new .NET Core project.
- **`restore`**: Downloads all dependencies from NuGet (this is rarely needed, as the **`build`** command does it for you).
- **`build`**: Compiles the projects.
- **`publish`**: Generates a self-contained folder used for deployment.
- **`run`**: Runs a project, building it if not already built.
- **`pack`**: Packages the project as a NuGet package.
- **`msbuild`**: Serves as a proxy to 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 when an argument is specified, 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; and with **`classlib`**, it creates a library project targeting by default .NET Standard 2.0.

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.

The .NET Core framework 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`**). 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 the Microsoft Docs site for more information.

The runtimes and application support can 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

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.15-x64;ubuntu.14.04-
x64</RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

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.10.15-x64
dotnet publish -r ubuntu.14.04-x64
```

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

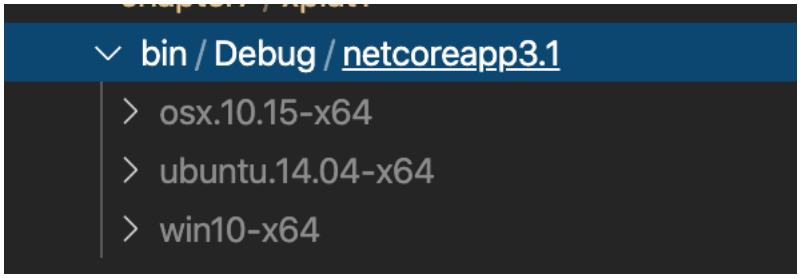


Figure 7-1: Folder Output

If you used a version of .NET Core before 3.x, the contents of these folders will surprise you. While in previous versions you would have just one executable per folder, with .NET Core 3.1 you get around 200 files and it is about 70Mb in size. The reason is that, starting with .NET Core 3.0, the **`publish`** command generates a self-containable executable with all the dependencies as individual files. This way you can run the application even on machines where the .NET Core runtime is not installed.

You can generate a single file (instead of 200) with the build option **`p:PublishSingleFile=true`**. Now all the libraries of the .NET Core framework are bundled into a 70MB executable, which is used to launch the cross-platform library.

You can also choose to generate a runtime-dependent executable by specifying the **`--self-contained false`** option. In this case, you depend on the framework installed on the machine, but the size of the application is just a few KBs and consists of only two files: a cross-platform library and a platform-dependent executable to start the library.

The **`dotnet`** CLI is extensible, so although it has very basic features now, it can be expanded with external libraries. You can 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 2017, you can use any modern text editor for which an OmniSharp plug-in 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 six popular text editors: Atom, Sublime Text, Vim, Brackets, 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 download the version for your operating system here. It's available for Windows, Mac, and 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 the extension pane in Visual Studio Code and typing **@recommended** in the search bar.

Once the extension is installed, you can open an ASP.NET Core project created using the **`dotnet new mvc`** command by selecting its folder. Visual Studio Code understands that it's a .NET Core project and will show a warning like the one in the following figure.

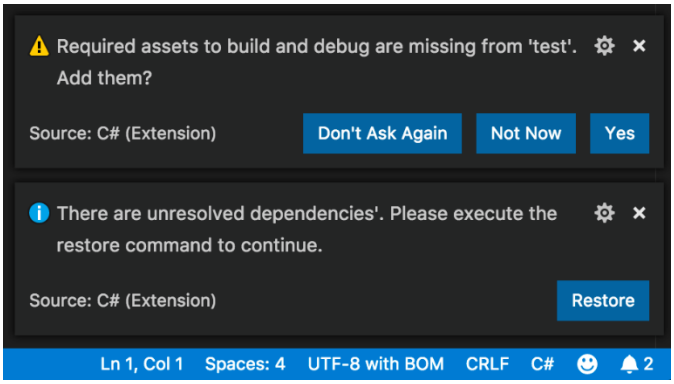


Figure 7-2: VS Code Warnings

The warning says that some configuration files are missing. Once you click **Yes**, two new files will be added to the .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 2019, if not more so.

For example, you have IntelliSense and code completion.

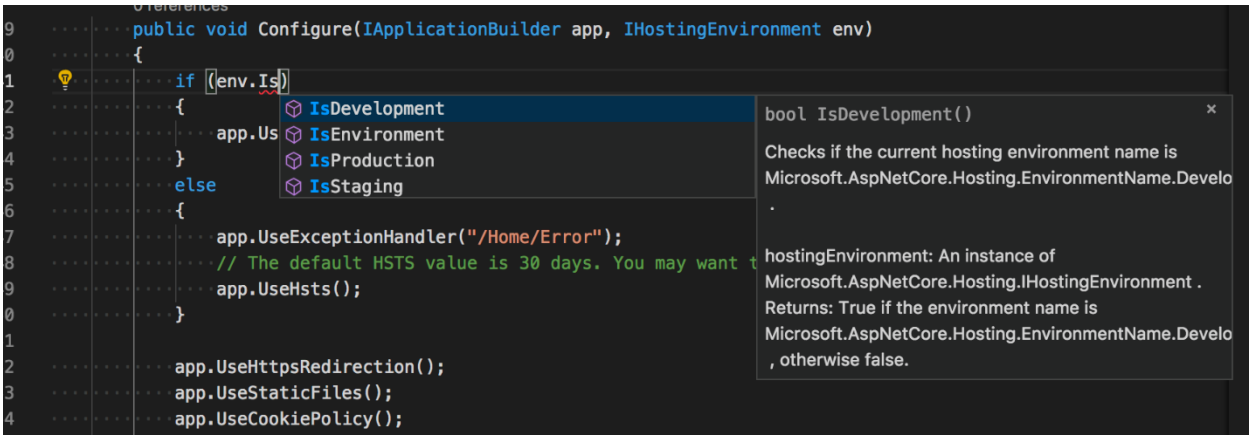


Figure 7-3: VS Code IntelliSense

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 7-4 shows all the locations in the UI where suggestions appear.

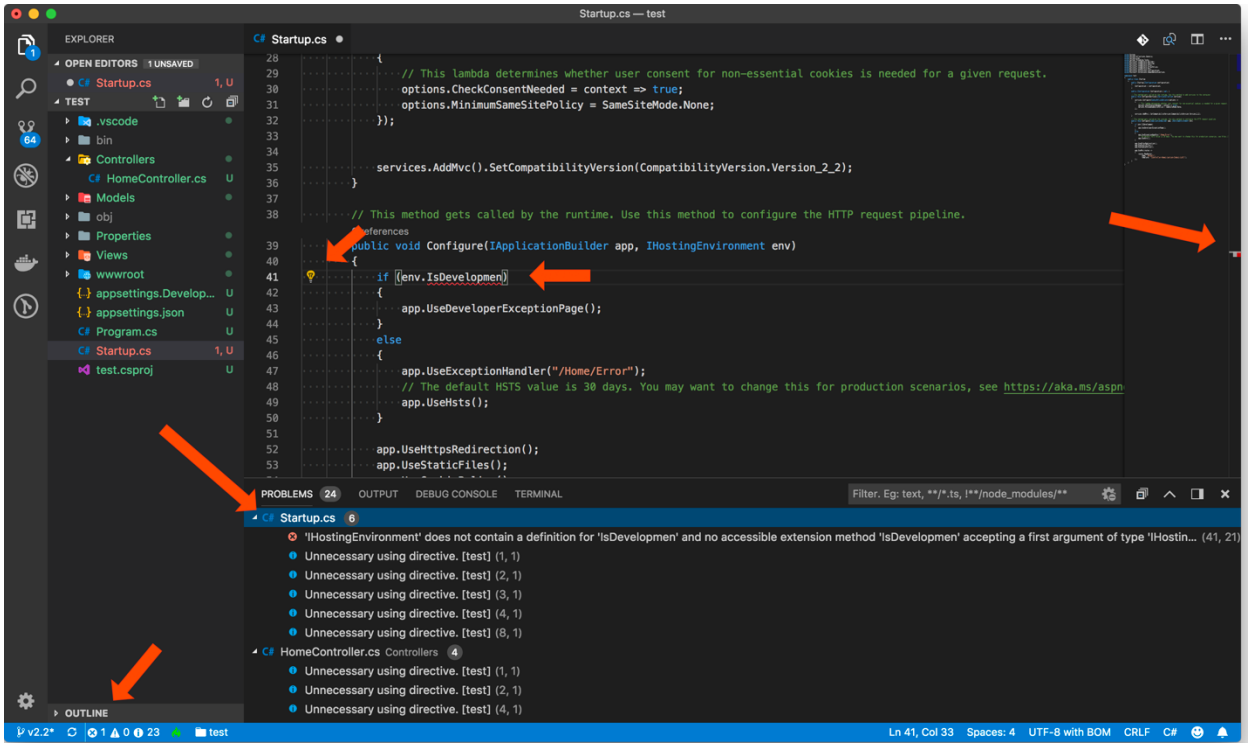


Figure 7-4: VS Code Suggestions

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.

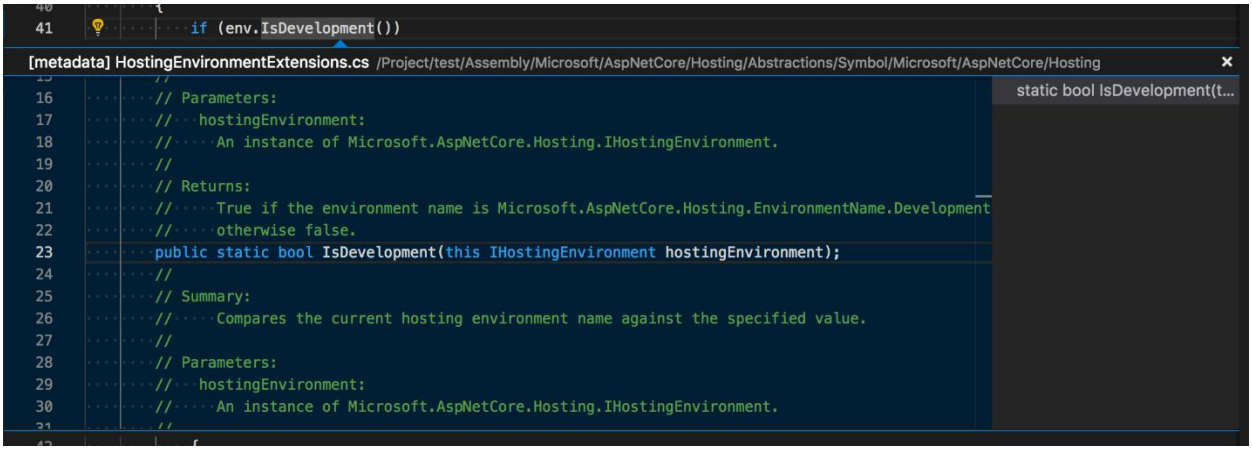


Figure 7-5: VS Code Navigation

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-shaped icon in the left sidebar), and click the **Debug** icon (the green Run icon, just like 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 with any other code debugger.

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

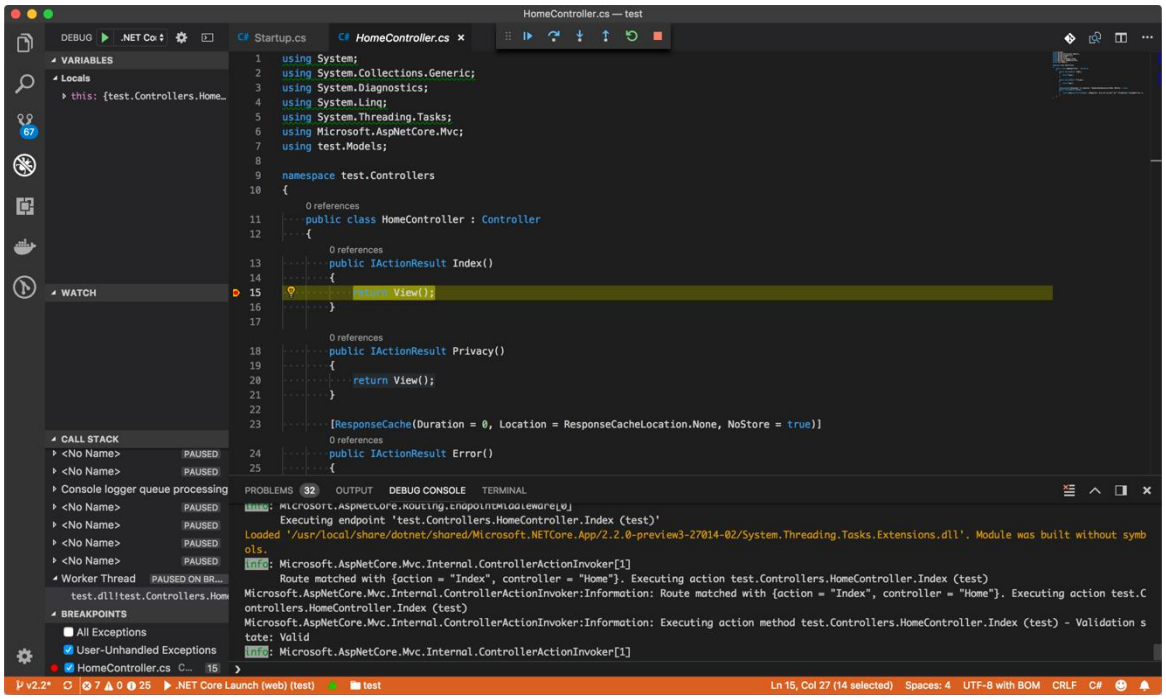


Figure 7-6: VS Code Debugging

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

While the release of the first version of .NET Core was a bit troubled with delays and changes of direction, version 3.1 is stable, implements the .NET Standard 2.1 (and therefore everything that was available in .NET 4.6.1), and is supported by great tooling in Visual Studio, the **`dotnet`** CLI, Visual Studio Core, and even Visual Studio for Mac.

As mentioned in Chapter 2, the .NET Framework (the closed-source, non-cross-platform framework we have been using since 2001) will not be upgraded anymore, and in November 2020 Microsoft will release the new version of .NET Core under the name of .NET 5, making it clear that the future of the .NET platform is based on .NET Core.

If with version 2.2 you could still debate whether it was a good idea to start a new project on .NET Core, now with version 3.1, .NET Core is the only way go.