Course Overview

Course Overview

Hello. This is Ronald Guijt. Welcome to my course on ASP. NET Core 2. It doesn't matter if you have used a previous version of ASP. NET Core or not, this course will get you ready to start building with this marvelous framework. We'll start with an empty project and build a business application from the ground up. On the way, I'll explain everything you have to know about. NET Core 2 and ASP. NET Core 2. The major topics of this course are exploring the new project structure, showing you that dependency injection is now essential, that middleware lets you create your own pipeline, and all the ins and outs about the new metapackage; working with packages and libraries, the different package sources that are now supported, how to bundle and minify, and the new light-up feature; understanding ASP. NET Core 2 MVC, showing off some great new additions to MVC such as tag helpers, view components, and unified controllers; setting up a web API, letting you create a REST-based web API using the exact same classes used in MVC; developing applications across frameworks and operating systems. In here, I'm explaining how. NET Core works, also together with the. NET platform Standard. And finally, deploying ASP. NET Core 2 applications where we will see that it is now possible to run your app on many different operating systems without the need to preinstall. NET Core. I assume that you have experience using a previous version of MVC, as well as knowledge about HTML and web programming in general. After watching this course, you will be able to start building ASP. NET Core 2 applications while understanding what makes it tick.

Exploring the Project Structure of ASP.NET Core 2

About This Course

ASP. NET Core is an exciting framework to create web applications in. NET. With the release of version 2, it has become even better. This first module is about the new project structure ASP. NET Core 2 brings. But first, some brief information about what to expect. ASP. NET Core was built from scratch by Microsoft. In the process, the decision was made to let go of some of the things you are familiar with if you've already developed with classic ASP. NET, the ASP. NET before ASP. NET Core. ASP. NET Core is significantly different than classic ASP. NET. This course assumes you have experience with a prior version of ASP. NET and MVC. It is designed to show you the new and changed things in ASP. NET Core and MVC compared to classic ASP. NET, taking the new features of ASP. NET Core 2 in account. A good way to learn ASP. NET Core is to immediately show you what it looks like in a working environment, in this case Visual Studio, without going through a lot of slides first. That's way I'll start out with File, New, Project and work on an app in C# throughout all modules of the course. Whenever I encounter something new, I will explain it using slides. While ASP. NET Core apps can be developed without it, a demo app in this course will be built in Visual Studio, which runs only on Windows. I chose that environment because it is the most commonly used. Examples of cross-platform tools you could use to develop ASP. NET Core apps including Visual Studio Code and JetBrains Rider. Globomantics is a company that provides a web application for conference organizers. The conference organizer maintains a list of conferences, and speakers can submit proposals for a conference, which then can be approved by the organizer. We're going to build this demo app from scratch. We will start right away with building the app in this module. First you'll see the changed project structure in Visual Studio. And I'm also talking about references, dependencies, and the underlying packaging structure. The Main method is the entry point many. NET applications use, including ASP. NET Core apps, and the Startup class is an enhanced way to configure the pipeline of your app, as well as dependency injection. That is an integral part of ASP. NET Core now. In the next clips, I'm taking the first steps to create the new project.

Starting a New Project

To get started with ASP. NET Core 2, you have to install the latest version of Visual Studio 2017; any edition will work. Note that Visual Studio 2017 has a Community Edition, which is available for free, but please check the licensing of this edition though. The Community Edition is not crippled in any way, but some user restrictions apply. At the time of writing, ASP. NET Core 2 itself is not included in the installation of Visual Studio. To get everything you need, run the ASP. NET Core installer, which you can download from dot. net. Let's start by choosing File, New, Project in Visual Studio. Then search for ASP. NET Core Web Application. You can choose different. NET Core versions here for your project, and as you can see, you can also create an ASP. NET Core app targeting the full. NET Framework. Don't worry about this too much for now if you don't get this. I will explain everything about the different frameworks later in the course. Just be sure to select ASP. NET Core 2 for now in the drop-down. The templates Web API and Web Application contain some already-to-go apps. There are also more advanced apps that use Angular and React. I want to explain everything from scratch, so I choose the Empty option here. Let's take a look at the project file next.

The Project File

Let's take a look at the project configuration file. You can do this by right-clicking on the project, after which you can select Edit Globomantics. csproj. I'm using C# for the demos, but in ASP. NET Core 2, Visual Basic. NET is also supported. In that case, you'll have a vbproj file. This project format has been around for a long time. If you've worked with classic ASP. NET, or any other type of. NET project, it should be familiar. However, the format of the file and how you work with it is different in ASP. NET Core. If you've worked with a previous version of ASP. NET Core that used the project. json file, that file is now gone, together with the export file. The new csproj file format replaces both of them. The reason that they are replaced is that project. json wasn't compatible with MSBuild, a build platform used in many DevOps environments. You can now just edit the project file without the need to unload the project first, and the structure of it is very different. For one, it doesn't contain any folder or file references. Let's right-click on the project and view all the files in the File Explorer. I create a text file in the File Explorer and switch to Visual Studio. You can see Visual Studio shows the file in the project. In ASP. NET Core, the Solution Explorer reflects what is on disk one to one. That means its files added to the project are now included by just placing the files in the project folder. This is also great news for conflicts in your source control system with a project file of your application. Since the file references are not in the project file anymore, you'll save a lot of time resolving conflicts with other team members. Back to the csproj file. In the TargetFramework node, the framework target we chose in a drop-down dialog when we created the project is visible. The value inside the node is known as a moniker. In our application, we use the moniker for a. NET Core 2 application.. NET Core is a relatively new version of the. NET Framework, which is significantly different from the full. NET Framework, which is around since the beginning of. NET. I will tell you all about it in module 5. You would probably think that an ASP. NET Core app always runs in. NET Core. Sounds logical, right? Well, in fact, you could also target the good old full. NET Framework if you want. That has monikers as well. The next node includes the wwwroot folder. I will talk about that in the next clip.

wwwroot

In classic ASP. NET, all files were served of the root folder of the project. You could just create a file in the root of the project and it will serve when requested by the browser. The serving of certain sensitive files like Web. config and Global. asax had to be blocked by using special routing rules. But these files had to be blacklisted. The new approach uses more secure whitelisting. In ASP. NET Core, nothing is accessible by the browser by default except when it is in a wwwroot folder. So typically you put images in here, but also CSS files and JavaScript files. Let's create a folder in wwwroot called img, and put the logo of Globomantics in it. We will use this in the layout page later on. As mentioned earlier, there's no need to add new items for everything you add to the project anymore. You can just use File Explorer to create the new folder, and place the new file inside of the folder. The next clip is about the final entry in the csproj file.

The Meta Package

The final entry in the csproj file is a PackageReference node. Every time you add a NuGet package using the dialog screen of Visual Studio, or whatever tool you're using, a PackageReference entry will be added here. Right now, there's only one package referenced, Microsoft. AspNetCore. All Version 2. This is one replacement for all the NuGet packages you had to reference in earlier versions of ASP. NET Core. In ASP. NET Core 1, having many smaller packages for each feature seemed like a good idea, but in practice this became a problem because all these packages tend to rely on each other. Upgrading one package to a new version could cause your whole application to go haywire. But if you still prefer things the old way, no problem, this is still supported. Unless otherwise mentioned, every feature of ASP. NET Core I use in this course is present in this so-called metapackage. The metapackage includes all packages by the ASP. NET Core Team. That basically means everything that is needed to use all features of ASP. NET Core. It also includes all Entity Framework Core packages and all Microsoft and third-party dependencies of both ASP. NET Core and Entity Framework Core. There is something else that is special about the metapackage. It is already installed on the target machine, so it doesn't have to be deployed with your app. This is a good thing because the package is huge. A new. NET Core 2 feature called a runtime store makes this possible. I'll explain more about the runtime store in the final module of this course. The next clip is about the Main method.

The Main Method

In classic ASP. NET, code in the System. Web assembly took care of starting the app and the Global. asax file had methods in which you could provide custom logic. The steps needed to start up an application are now determined by you, and that starts with the Program class. You're in the driver seat. The Program class contains a Main method, the entry point for the application. When the runtime executes the application, it looks for this method and calls it. Nothing new here. Most. NET applications start up using the Main method. The application starts initially as a command-line application. The Main method configures ASP. NET Core and starts it. The configuration is done by calling BuildWebHost, which is a method in the Program class that returns an IWebHost object. On that object, Run is called, and from that point onward, this application has become an ASP. NET Core application. BuildWebHost calls CreateDefaultBuilder on a static WebHost class. Since ASP. NET Core is open source, we can look at the source code of the CreateDefaultBuilder method to understand what it's doing. I downloaded it from GitHub. First, it instantiates a new WebHostBuilder. It configures it to use Kestrel, which is a web server which will host the app. It is an internal web server for your application, so it's embedded in your application. It also tells the application to use IIS integration. IIS is the external web server for this project and requests will be proxied between the external web server and the internal web server. This setup will relay requests from IIS to Kestrel, and vice versa. It also configures the content root, which is the project directory. With ConfigureAppConfiguration, the new settings system is configured in a way that it loads a settings file, appsettings. json, by default. ConfigureLogging sets up logging to the Console and the Debug window by default for the whole application, including the libraries it uses, such as MVC. UseDefaultServiceProvider will instruct ASP. NET Core to use its internal dependency injection container, and ConfigureServices is used to add types to the dependency injection container that I used by one of the ASP. NET components mentioned above. By default, a class called KestrelServerOptions is used to configure Kestrel. It is good to understand this code because chances are that you'll want to deviate from the default settings. For example, you want to use another dependency injection container than the internal one. That container library will provide an extension method on WebHostBuilder. You could then still use CreateDefaultBuilder. You can then just override the default service provider by the one you want to use. You can also deviate from the default external web server or logging framework, for example. You can see that on top of the configuration that is done with CreateDefaultBuilder, the WebHostBuilder is also configured to use a startup class, apply named, Startup. The Startup class is just a simple class, not deriving from any other class. The runtime will, by convention, call two methods. First, ConfigureServices, and then Configure. We'll look at these in the next few clips.

Dependency Injection

As we saw in the previous clip, the runtime executes Main, which among other things configures the Startup class. The runtime will call the methods ConfigureServices and Configure inside that class. The purpose of the ConfigureServices method is to configure dependency injection. Dependency injection in classic ASP. NET was optional. In ASP. NET Core, it forms an integral part of ASP. NET itself. The dependency injection mechanism depends on an IoC container. IoC stands for Inversion of Control. Typically when an application starts up, types like a class can be registered in a container. In dependency injection terms, these types are called services. Once registered, other types can ask the container for an instance of that type. Also, during registration you can determine what the lifetime should be of that instance. The lifetime of an object is managed by the container. A transient lifetime means a new instance of the type is created every time it is asked for. A scoped lifetime means the instance will live until the web request is completely handled. And singleton lifetime means once an instance is created, the same instance will be supplied every time until the app closes. Here we are back in our project to see how this works. At application startup, the hosting layer of ASP. NET will give you an opportunity to register types in a container by calling the ConfigureServices method in the Startup class. By supplying an IServiceCollection object as a parameter of it, you get the API to do the registering. In the next clip, I'm creating and registering some services for Globomantics.

The Startup Class: ConfigureServices

To get the data for the Globomantics app, we are going to call a web API that we will create in a later module. Let's first create a. NET standard Class Library, which will contain the classes that are shared between the web app and the future web API. I create three simple model classes under the Models folder. ConferenceModel contains properties for a conference and assumes a default start date of Now. And ProposalModel, of course, contains information about a proposal and correlates to a conference by using the ConferenceId property. StatisticsModel contains two statistic properties that we're going to use later in the course. Back in the web application, let's create an interface for the class that is going to be responsible for retrieving the conferences, called IConferenceService. Of course we have to add a reference to the shared library in order to make this work. The classes that implement this interface will contain an async method, GetAll, which will return an IEnumerable of ConferenceModel and a GetById method that returns one conference by ID. GetStatistics is used to return a failed StatisticsModel object. And finally, an Add method that adds a conference to the list. Here's a class called ConferenceMemoryService that implements the interface. In the next module, we are going to get the data from a web service, but we're going to get in-memory data for now. So I just create a list with a couple of in-memory conferences. I add to the list when Add is called, I return the list when GetAll is executed, I'm searching the list for the right ID in GetById, and calculate statistics in GetStatistics. Note that the interface dictates async methods because we need that for web API access. For that reason we have to wrap the results in a delegate that is passed to Task. Run. I'm also creating IProposalService, which is very similar to IConferenceService. The differences with IConferenceServices are that it only can get you proposals for a specific conference and it has an extra method needed to approve a proposal. Here's the accompanying ProposalMemoryService. It also has an in-memory list, this time with proposals that point to a certain conference with a key called ConferenceId. For the Approve method, it looks up the proposal in the list using its Id and sets the Approved property to true. To register the ConferenceMemoryService with a dependency injection container, I type services. AddSingleton of IConferenceService, ConferenceMemoryService. This means whenever some type asks for an IConferenceService object, supply an instance of ConferenceMemoryService. And we do the same registration for IProposalService. The AddSingleton method determines the lifetime of the object. The object, in this case, will live as long as the application runs. When we are going to create controllers in the third module, we can just create constructors in the controller, accepting an IConferenceService object, and the ConferenceMemoryService instance will be automatically injected. You'll see what I mean if I'm going to create the controllers. You can even inject a service in a view using the Razor syntax, @inject. So with dependency injection there is no need to new up a class, and there's no need to think about when an instance goes out of scope, because the container manages that. And because a class can be registered by using an interface it implements, there is no dependency between the types. The controller can inject an instance of any class as long as it implements the interface, and that opens up great opportunities for unit testing.

Unit Testing

When you test a method, you probably only want to test that method and not the methods of other objects it calls. So during a unit test, you can configure a container to inject a mock instance implementing the interface, that is an object that simulates the response of the actual object that is used when running the app. Our current ConferenceMemoryService is in fact a mock object, it doesn't really access the API. I could easily replace it with a class that does actually read data from the API just by creating another class that implements IConferenceService, and configure a container to inject a new type instead. The rest of the app doesn't change because there are no hard dependencies between classes because of the interface registration that decouples them. In the next clip, you'll learn how a request is processed.

The Startup Class: Configure

The other method in the Startup class is Configure. This method configures the HTTP request pipeline of ASP. NET. You can see it accepts an interface object of another service called IApplicationBuilder. That object is injected. The pipeline specifies how the application should respond to HTTP requests. When your application receives a request from the browser, the request goes through the pipeline and back. When there's nothing in the pipeline, nothing happens, so you need to plug in the stuff you need in the pipeline. This is in contrast to classic ASP. NET where the pipeline was prefilled with a lot of stuff, stuff you maybe don't need at all, making the app unnecessarily slow. The individual parts that make up the pipeline are called middleware. For example, the MVC framework can be plugged in as middleware, but you might do some authentication first. In that case, configure a piece of middleware before MVC, doing the authentication. Even the ability to serve up static files containing JavaScript and CSS, for example, has to be configured as middleware. While traveling through the pipeline, data about the request travels through the pipeline and gets read and manipulated by the middleware, and eventually the result is the response. The functionality of the web server is also accessible by the middleware because ASP. NET Core makes them available in the form of feature interfaces. Here's the bigger picture summarizing the components of an ASP. NET Core app. When a request is made, it first arrives at the web server, and I assume that you've used IIS for now. IIS then invokes the. NET runtime at initial startup, which will load the CLR and then look for an entry point, the Main method in your application, and it then executes it, starting an internal web server in your application. In our case, this is Kestrel. The Main method and the Startup class configure your app and the request is routed from IIS to Kestrel, pushed through the pipeline, configured by the Startup class, processed by all the middleware, which will produce the response and reroute it back to Kestrel and IIS, which will produce the response to the browser. This is much more efficient than the old System. Web approach, and it will surprise you how performant and resource friendly it is. Most versions of classic ASP. NET relied on an assembly in a. NET Framework called System. Web, which was tied to IIS. It is not suitable anymore for today's standards. It is around since classic ASP. NET version 1, and as ASP. NET progressed, stuffed with code to support all kinds of new features. Processing a request using System. Web isn't very efficient anymore for that reason in terms of memory usage and performance. Using the pipeline, you only plug in what you need, and everything you plug in is in separate assemblies exposed in NuGet. System. Web relied on IIS, and IIS is tied to Windows. That's the reason you can't run classic ASP. NET apps on other web servers than IIS, and only on Windows. Now that this limitation is lifted, in ASP. NET Core you can run apps on many external web servers for many operating systems. Keep in mind that there are actually two web servers, one web server that I call the external web server. This is a full-blown production-ready web server like IIS, Apache, or NGNIX on Linux, for example. There's also an internal web server, part of and hosted by your app. Requests to the external web server are proxied to the internal web server, and vice versa. Like the external web server, you're also not tied to using a specific internal web server. The most obvious choice is Kestrel because it's very fast, it is implemented cross-platform, and it has first-class support in ASP. NET Core. Kestrel is a lightweight web server; it doesn't have much more features than actually executing a request. That's why you need an external web server in production scenarios, which you can configure security, configure caching and compression, set up mappings to the right app depending on the URL, and many more things. If you've used OWIN and the Microsoft implementation Katana, this all probably looks familiar. The ASP. NET Core pipeline can be seen as the next version of Katana. Keep in mind that an important difference is that there is still a dependency on System. Web when using Katana in classic ASP. NET. If you want OWIN support for backwards compatibility in ASP. NET Core, you can activate it by using middleware, which is available in NuGet. In the next clip, we'll see the pipeline in action.

Middleware

So back to Visual Studio. In the Configure method, there is already some implementation. I'll begin with the app. Run line. We are creating a mini-middleware here using a lambda expression. The Run method of the injected IApplicationBuilder accepts a request delegate as a parameter. A request delegate is a delegate that can contain methods that have an HTTP context object as a parameter. The HTTP context contains a Response object. We write the text, Hello World, directly to the response. As the name implies, WriteAsnyc is asynchronous, so we can await it. And because we await it, the function has to be marked as async. Let's run this app. You can see the text appears in the browser, but when we inspect the page source, we can see it's just the text, no doctype indicator or HTML tags surrounding the text. Now I create another app. Run to output another text. When running the app, you can see the second text is not displayed. That is because a middleware should call the next one. If not, the middleware just terminates the pipeline. The Run method on IApplicationBuilder won't help us with calling the next middleware. But luckily, there is a Use method. This one can accept two parameters, the HTTP context as before and a next delegate. When I refactor the first app. Run to app. Use, and call the next delegate when I'm done, the second text will be shown in the browser. Now let's zoom into this concept some more in the next clip.

A Closer Look at Middleware

Besides IApplicationBuilder and IHostingEnvironment, which I will explain in the next module, I can also inject an ILogger object. Since logging was already configured in the CreateDefaultBuilder method, it will enable me to log to the console. As a generic parameter, you have to specify the class you're logging in. So the logger automatically has the right context. In the app. Use, I'm removing the output of the text and I log Before and After next is invoked. We also do some logging if the app. Run is called. Now let's change the execution environment from IIS Express to the application itself, which will run the app standalone without the external web server IIS, using only the internal Kestrel as its web server. You can see it starts up now as a console application. Look at the output the logger creates. Besides the logging of Kestrel itself, you can see the first middleware being executed. Next is called and the second middleware is executed. And then control is given back to the first middleware. Let's go back to this slide for a moment and notice the arrow going into the pipeline and the arrow coming out. Everything before the invocation of Next in a particular piece of middleware is happening in the incoming arrow, everything after in the outgoing. Middleware tends to be more complex than just writing to a log or outputting a text, and that's why middleware can also be written in separate classes and easily plugged into the pipeline by using extension methods. We'll see that in action in the module about MVC where we call the extension method useMvc on the applicationBuilder object. Now that I explained how a request is processed, I can also explain the launchSettings. json file in the next clip.

launchsettings.json

In the Properties folder there's a file called launchSettings. json. This tells Visual Studio what to do if you press the Run button. It by default contains two profiles. The profiles are visible in the drop-down next to the Run button if you're using Visual Studio. IISExpress will launch IIS Express, which will host the app and start a browser that will hit the URL. It also sets the environmentVariables ASPNETCORE\_ENVIRONMENT to Development. We can detect this environment in our app and react to it. For example, load full CSS files instead of minified ones. I will show you that when we go in to do the layout page for this app. There's also a section configuring IIS Express, setting authentication, an applicationUrl, and an sslPort. The second profile just has the name of the web app itself. It will just run the application as a command-line application. So, only the internal web server Kestrel will be used. We used this option in the previous clip. And also a browser will be started that hits the URL. These settings, as well as some of the settings used in the project file, can also be set by using the Properties dialog of your project.

Summary

To sum this module up, ASP. NET Core 2 is something entirely different if you're used to classic ASP. NET, and also quite a bit different when you've used a version prior to 2. Maybe the project structure is new to you, and files are not served directly from the project root, but in the wwwroot folder. Every ASP. NET Core app starts its life as a console app and gets promoted to an ASP. NET Core app using the WebHostBuilder class. And the Startup class has two important tasks, to set a dependency injection, an essential part in ASP. NET Core now, and to configure the pipeline through which all requests travel and responses are generated. In the next module, I take a closer look at the different kinds of packages there are and see how we can use the build and tooling to process client-side files.

Working with Packages and Libraries

Overview

This second module is entitled, Working With Packages and Libraries. We're going to take a look at the various package sources that are available to you. You'll also see how to create some simple bundling and minification tasks for our demo app with the built-in bundling and minification feature. Next, we'll take a look at more advanced task runners, like gulp, grunt, and webpack. ASP. NET Core 2 makes it easy to reference all their full. NET Framework packages; I will show you how. And finally, the light-up feature seems like magic when you see it for the first time. I will show you what it is and how it works. But first, the different package sources.

Packages From NuGet and NPM

NuGet is maybe your package source for all packages now, server, as well as client side. In this release of ASP. NET, Microsoft encourages developers to use NuGet for packages specifically meant for. NET, so no JavaScript and CSS libraries. For client-side packages, you can now use Bower. This is a client-side package manager many web developers in the world use, also developers that are not into Microsoft technologies. We'll see in a minute that it's now fully supported from within Visual Studio. There's also another package manager called npm, which stands for Node package manager. Node. js is a first-class citizen in Visual Studio now. For developers who create Node. js solutions in Visual Studio, npm is the source for server-side packages, but npm is sometimes also used for client-side packages. Npm is also used to install build tools for the client-side workflow. These tools are running on Node. js. Examples are the gulp and grunt task runners, which are also supported by Visual Studio now. NuGet, npm, and Bower are not the only package sources in the world. If you want to get packages from other sources than the ones Visual Studio supports out of the box, you could download an extension for Visual Studio called the Package Installer, or just use the command line, which is also an option if you don't use Visual Studio.

Working with NPM

For the demo app, I'm using Bootstrap, which is the most popular CSS library in use today. I'll grab it from Bower. I recommend to first create a Bower configuration file in your project called bower. json. You can use the template Visual Studio provides. To install Bootstrap, right-click on the project and select Manage Bower Packages. Bootstrap is immediately visible when you click Browse. I'm installing version 3. Now let's take a look at the bower. json file. The dependency is added. I could also manually enter packages here and Visual Studio helps me with IntelliSense. Npm also works with a configuration file called package. json. Visual Studio has support for it, it just doesn't have a nice, visual dialog to manage packages, like Bower has. Once it's done installing Bootstrap, take a look under the Dependencies folder. There's a Bower folder, and under that there is our package. Also notice that a lib folder is created under wwwroot when I install the package. It contains all Bootstrap files in the package. There's a lot of stuff there. That's great, but I only want the files I need under the wwwroot folder. I don't want the files that I don't use to be part of the deployment of the app. So let's do that again. I'm uninstalling the package. Now take a look at bowerrc, a file that becomes visible when you press the little arrow in front of bower. json in the Solution Explorer. The output folder is specified here. Let's change that to bower\_components and save it. Now when I install the package again, it is installed in that separate folder. But now, of course, nothing is accessible to the browser. Besides that, I'm like some bundling and minification on the CSS and JavaScript. In the next clips, you'll see how these things work.

Bundler and Minifier

Microsoft has created a tool bundled with ASP. NET Core to bundle and minify your CSS and JavaScript files. It is called Bundler and Minifier. It uses a configuration file in the project called bundleConfig. json. For the demo app, I'm creating a site. css file, which contains any CSS styles specific for this app. I'm placing it under the wwwroot/css folder. It contains the font setting and classes that we're going to use to display the page title and for displaying the statistics. Now I want to create one minified bundle containing the bootstrap. css file, which is under the bower\_components folder, and the site. css file I just created. Add a JSON File called bundleconfig. json and give it some content. You can specify a bundle by setting the output file with one or more input files. These files will then be stuck together and minimized, made as small as possible. I want the bundled and minified output file in the CSS folder. The input files, the files I want bundled and minified, are the bootstrap. css file, which is under the bower\_components folder, and my own site. css file, which is already in the wwwroot folder. Just like I have site. css in an unminified version in wwwroot, I also want bootstrap. css copied over in unminified form. Later on, we're going to load the full versions when developing and use the minified versions in production. For this, I create another configuration entry in bundleconfig. json, specifying the bootstrap. css file in the lib folders input, and bootstrap. css in wwwroot/css as output. But now, as an extra option, I turned the minifier off. If you don't use Visual Studio, you can run the bundler and minifier by typing the command dotnet bundle. If you do use Visual Studio, it's a good idea to install the bundler and minifier extension from the Visual Studio Gallery. Then the bundleconfig. json file is recognized by the Task Runner Explorer. You can run the bundling from there and trigger it when certain Visual Studio events happen. So I could run bundling and minification after a build, for example. When I run the task now, you can see the CSS files appearing in the css folder. With the plugin there's also extra tooling available when you right-click on a file that is relevant for building and minification. If you need more than just bundling and minification, then you could use task runners. I'm looking at them in the next clip.

Using Task Runners

Gulp and grunt are task runners that run on Node. js. They use plugins to execute tasks. Their features are defined by these plugins. There are many plugins available for them that can perform all kinds of tasks. Besides bundling and minification, you could, for example, transpile your JavaScript into another version, compress your images, generate documentation, or check your JavaScript for style errors using JSLint, for example. You first have to install gulp or grunt and their plugins using npm, and then you can write tasks in JavaScript that call the different plugins in a specific order, specifying configuration for them. Visual Studio recognizes the task files in the Task Explorer for gulp and grunt out of the box, and with the plugin called the WebPack Task Runner, it also works with webpack. Strictly speaking, webpack is not a task runner, but a module bundler, which is more sophisticated, but it can achieve the same goals. In exactly the same way as we did with bundleconfig. json in the previous clip, the tasks can then be executed from within Visual Studio or they can be bound to build events. Here's an example gulp task configuration. A file called gulp. js, which is convention, is defining the tasks. One is for minifying JavaScript. You can see it's a JavaScript function, which will be executed using Node. js. It first defines the source, or input files, which are in variables in this example. It then defines the output file, and then it calls uglify, which is a minified plugin for gulp. Further up in the file, it is being pulled in with the require statement. Finally, it instructs gulp to write a file to the root of the destination folder. Once defined, the task runner recognizes the file, and you can execute individual tasks or use Visual Studio events to execute the tasks. In the next clip, you'll learn about a great way to keep using full framework libraries.

Referencing Full .NET Framework Libraries

For the demo app, I'm going to need some extension methods on HTTP client that are not part of the ASP. NET Core metapackage. So I bring in a package called System. Net. Http. Formatting. Extension. As you can see, the package is for the full. NET Framework 4. It turns out this just works. In fact, you can reference any full framework assembly from ASP. NET Core 2, even if it targets. NET Core. Not all assemblies are going to work, however. They only work if the assembly only uses framework classes and methods that are implemented in. NET Core 2. The totality of classes and methods is called the. NET Framework API, and a particular set of API is defined in a specification called. NET Standard.. NET Core 2 implements. NET Standard 2, so as long as a legacy assembly you're referencing only uses. NET Standard 2, and not more, it will work. It turns out the majority of legacy packages will work, as long as they don't use Windows-specific things like WPF or Windows Forms. More about. NET Standard and the sharing of code in a later module. Next up is the light-up feature.

The Light-up Feature

The light-up feature enables hosting environments to inject features into your app. An example is when you use Microsoft Azure as a hosting environment, you probably want to use its Application Insights feature that enables you to keep track of your running app using a dashboard. To enable that in previous versions of ASP. NET, you have to add a NuGet package and maybe add some logic to your app. There's no need to do that anymore with the light-up feature. Azure can inject the logic into your app without the need for you to do anything. Here's how it works. It evolves around an interface called IHostingStartup. At app startup, the executing assembly is scanned for classes that implement this interface. When it finds one or more of these classes, it calls the Execute method on all of them. The Execute method is passed an IWebHostBuilder object. That's the same object we saw in action in the Main method to configure our app with logging, dependency injection, and the external and internal web servers. That's great, but it doesn't explain how Azure can inject a feature in our app. Apart from scanning the executing assembly, other assemblies that are configured in the ASPNETCORE\_HOSTINGSTARTUPASSEMBLIES environment variable will be scanned as well for classes that implement IHostingStartup. And of course, execute will be called on them as well. Environment variables are of course controlled by the hosting environment, so Microsoft has added such an environment variable to their servers on Azure. Finally, there are probably extra packages to be installed with your app by the hosting environment to make this work. This is the case with Application Insights. The DOTNET\_ADDITIONAL\_DEPS environment variable points to a JSON file that can contain these dependencies, and these are then automatically used by your application.

Summary

In this module, you saw that you can now use any package resource, and if you're using Visual Studio, the IDE will support you. We looked at how to create simple tasks with the bundling and minification feature. We also took a sneak peek at the more advanced, but also more complex, task runners. We referenced older. NET Framework packages. And finally, we saw the inner workings of the light-up feature of ASP. NET Core 2. In the next module, the building of our demo app really takes off. We're going to dive into MVC for ASP. NET Core.

Understanding ASP.NET Core 2 MVC

Module Overview

In this module I'm going to build most of the demo app using MVC for ASP. NET Core. You'll see how to set up MVC in a new project. Environments help you to distinguish between development, staging, and production. Controllers in MVC are unified, and you will learn what that means. Tag helpers are great to render tags in a very HTML-friendly way. I will show you the new way to do application settings, and view components are partial views on steroids. You'll see how that works as well.

Setting Up MVC

In this course, I'm only talking about MVC, and not about Web Forms, and that is because Microsoft chose to do no reimplementation of Web Forms in ASP. NET Core. You saw in the previous module that services are objects that can be injected in, for example, controllers. We need to register the services that MVC uses in the IoC container. MVC controllers rely on these services, and you can inject them yourself as well. For example, when you want access to the HTTP context object. The complete MVC framework is available as a piece of middleware in the pipeline, so I have to make sure that I add code so that the middleware in the pipeline is used. And while I do that, I also have to determine what routing I want to use. To set this up in the application, all you have to do is insert one line in the ConfigureServices method in the Startup class. AddMvc is an extension method that registers all necessary services. Next, we have to insert the MVC middleware in the pipeline. First I'll delete everything from the previous module in the Configure method. I just add app. UseMvc. This is enough to set up the MVC middleware, but what is missing here is a configuration of routes. There's an overload for that, which accepts an action delegate of IRouteBuilder. If you have the need for complex routing, it might be a good idea to create a separate method that you call here. For simple routing you can, just like me, use a lambda. To set up a default controller action route, you can use this syntax and add a default controller and action. This means that if no controller action is specified in the URL, in my case the conference controller's index action is hit, which I will create in a minute. There's also an optional parameter ID here. If it is specified in a URL it will bind to a parameter of the action called id, if present. If you start out with a controller named homeController with an index action, there's no need to write this code. When you write UseMvcWithDefaultRoute, MVC will automatically look for a controller called home with an action called index if a request is received that doesn't supply in the place's route. It's also possible to use attribute routing. You don't have to enable that explicitly anymore in the configuration. In the next clip, I'm configuring other middleware.

Configuring Middleware

Let's put some more middleware in the pipeline. One thing I forget easily is to add static file support. If we don't do that, none of our images and CSS files will be served. Let's run the app at this point without any controllers in place. All I get is a white screen. If I open up the F12 tools of the browser, I can see it returns a 404 HTTP status code, Page not found. If you want to make this visible on the page, add the StatusCodePages middleware. If I now refresh the browser, you can see a minimal error message. To create custom error messages, you can use UseStatusCodePagesWithRedirects, which will redirect to a custom error page. You can even add in a wildcard that puts the HTTP status code in a string. For now, I'll just use the standard status code pages. The order of the middleware is, in this case, not very important, but if you add middleware to do authentication, then you must add that before the call to useMvc, because you want to do authentication before the request reaches MVC. I'm not going to cover authentication in this course, there's a special course about this topic called Understanding ASP. NET Security. What about the famous yellowish page ASP. NET presents when an unhandled exception occurs? You know, in classic ASP. NET it was configurable in the custom error section of the Web. config. That page is now also middleware, activated by a call to UseDeveloperExceptionPage, which you probably only want to plug in if you're developing the app, and not in production. You can manage this by working with environments. That topic is up next.

Environments

If you're building applications professionally, then you most likely are working with different physical hosting environments running on different servers like development, staging, and production. In ASP. NET Core, this is supported by an environment variable called ASPNETCORE\_ENVIRONMENT. The value of that variable can be read in your code by using an object implementing IHostingEnvironment. You already saw how you can configure that environment variable at development time in Visual Studio, by using launchSettings. json. In production, you should set the environment variable in the operating system. In code, you can determine what the environment is by using an object that implements IHostingEnvironment. In the template we chose when we created the project, it is already injected in the Configure method, and the line of code that checks the environment is also already there. The IHostingEnvironment object has a method, IsDevelopment, IsStaging, and IsProduction. These three environment values are so common that the API supports it directly. But, you can use any string you'd like in the environment variable. With each environment, you can supply a string for your custom environments. You probably don't want to expose details of exceptions in production, so we only plug in the developer exception page middleware if the environment is development. Now, if an unhandled exception occurs, the page with the details of the exception will only appear when ASPNETCORE\_ENVIRONMENT is set to the string development. The Startup class itself also supports the different environments. You can add different Configure or ConfigureServices methods for each environment by adding the name of the environment to the method name, or just create one specific configure method for an environment name and let the rest fall back on the standard Configure method.

Unified Controllers

Earlier ASP. NET versions had a clear distinction between MVC, the framework for web user interfaces, and web API, the framework for REST web services, also known as APIs, that expose data only. Web API was created following the exact same model MVC uses. It, in fact, also uses the MVC pattern used in types that are in most cases named exactly like the types used in MVC. One of the reasons to build web API completely separate from MVC, even though it matches one to one, was because MVC relied on the System. Web assembly, which is tied to IIS and Windows. Microsoft wanted to untie a web API from System. Web, which among other benefits gave web API applications the ability to be self-hosted. Now that all of ASP. NET Core doesn't use System. Web anymore, it doesn't really make sense anymore to have two sets of types that do almost the same thing. In ASP. NET Core there is just one set of types to support the functionality of MVC and a web API. And because everything is unified now, having multiple names for the same thing also doesn't make sense, so the name web API has been dropped. The combined functionality of MVC and web API is now just called ASP. NET Core MVC.

Implementing Controllers

So let's create the main controller for the app. First, I create the Controllers folder and add a class called ConferenceController that derives from the controller base class. Let's give it an action method named index. You can see I'm creating async actions, which require no extra plumbing. The convention is to return IActionResult now instead of the ActionResult base class, which makes more sense. You can now just create your own action result without being tied to the action result abstract base class. This also has its benefits with unit testing. To get the conference data, I'm going to ask a dependency injection container for an object that implements IConferenceService. We can do that by just creating a constructor with a parameter of that type. We registered this class earlier with dependency injection so the container will make sure it fills the parameter with the correct object. I'm going to store this object in a private field. Let's set the ViewBag. Title to Conference Overview and type return View. Just like previous versions of MVC, View here is a method of the controller base class that news up a View result instance. I retrieved the model data from the IConferenceService object and passed that into the view. I also want to have support for adding a new conference, so I created an Add action with a nice ViewBig. Title, and I'm returning a View, passing in an empty ConferenceModel object. The user is going to enter the data in HTML form I will create in a second. When a user submits this form, the controller must catch that HTTP POST, so I create an action that responds to that using the HttpPost attribute. It accepts the now failed ConferenceModel. After checking if the model is valid, I tell the conference service to add a new conference, and then redirect back to the index. What is still missing is a controller for the proposals. I prepared it before recording so you don't need to watch me type the whole thing again. One difference with the conference controller is that when the index is called, I need a conferenceId. We're only going to show proposals for that conference. To get details for this conference, I can also inject IConferenceService besides the IProposalService. I can then construct a nice ViewBag. Title involving the conference name. Finally, it returns the view with the proposals for the conference ID that was supplied. It uses the IProposalApiService object for that. It also has a set of Add methods. When the Add action gets called, the conferenceId the new proposal is for is passed in. I'll show you how to do that when I'm adding the views. So I can give the view that information by filling the conferenceId property of the new proposal model object. I also have an approve action that just uses the Approve method of IProposalService, and then redirects back to the index action using the ID of the proposal we approved. Now let's create some views with tag helpers in the next clip. The principal of a layout page has

Tag Helpers

remained the same compared to classic ASP. NET, so let's design that first. We need a Views folder containing a Shared folder. In that folder I add a new item, look for the Layout Page template, and call it \_Layout. cshtml. I want this layout page to be the default for all views, so I add \_ViewStart. cshtml to the Views root folder and specify the new layout file. Here's the layout page for the Globomantics app. There are a couple of diffs that use the Bootstrap grid system for positioning. The grid system is there to position HTML elements in an easy way. There's a container, and inside the container are typically rows with columns. And columns have a certain width. All these things are controlled by CSS classes. In the first row we put the Logo image and the ViewBag. Title. The views for our controllers will be rendered in another row in the RenderBody section. As you can see, the Razor syntax is still there in ASP. NET Core MVC. But for rendering most tags, there is now a more HTML-friendly option called tag helpers. Where you in classic ASP. NET needed HTML helpers to render HTML tags to support the Views model, you can now use HTML attributes like asp-for to do the same. The benefits of this are this is a more HTML-friendly approach. When you hire a designer for your app, this HTML editor application probably can't make sense of HTML helpers, but tag helpers should be no problem. Also, maybe more importantly, with HTML helpers, adding HTML attributes to a tag results in ugly code. You have to new up an anonymous type and give it properties that represent the attributes. There are a lot of tag helpers supplied with MVC, but should you want to create custom logic that responds to tags or tag attributes, you can create your own by deriving from a tag helper base class. You can also invent entirely new tags. All built-in tag helpers are in the Microsoft ASP. NET MVC tag helpers assembly, which is part of the metapackage. To enable tag helpers for views globally, create a view called \_ViewImports. cshtml in the root of the Views folder. The Razor syntax in here will be automatically added to every view that's generated. Type the Razor syntax, @addTagHelper, followed by a string containing which tag helper to load. I want all of them, so I supply the wildcard star, followed by a comma, and then the assembly name the tag helpers should come from. If you don't want tag helpers everywhere for some reason, you can also put this directive in your view. Using the same method, it is also possible to register your own tag helpers. To do that, you can just derive from the tag helper base class and put the new class in a new assembly, and load it up using the addTagHelper command. MVC will automatically scan the assembly for types that derive from TagHelper. Back to the layout page. In a previous module, we created CSS files in the wwwroot folder. I want the individual CSS files only to load when the environment is development, because it is much nicer to have the readable CSS files when you fail with a design in the F12 tools of the browser while developing. There is a tag helper to determine the environment with a convenient name, environment. Within that tag, I've put a standard HTML link tag with the asp-href-include tag helper. Tag helpers can be entirely new tags like environment or attributes on existing tags. The asp-href-include tag helper enables me to specify a bunch of files with a wildcard, and MVC will render individual link tags for each of them. In this case, I'm selecting all files in the css folder in wwwroot, but I don't want the minified CSS file that contains all CSS, that's only for non-development environments. I can use asp-href-exclude to exclude it. In case of a staging or production environment, I only want that minified file, so this was the CSS. When you have JavaScript in your app you can do the same for the JavaScript files. Also on the place where I rendered the logo, the img tag uses a tag helper called asp-append-version, which enables cache busting of images. That means a version hash is calculated based on the image content and appended to the name of the image when the image is requested by the browser. This allows me to change the image, and when it changes I'm sure it will appear on the screen of my users immediately, and not show a version that was cached earlier by the browser. So to see all this in action, I quickly create an empty view for the index action of my conference controller, and run the app. When I press F12 in the browser, take a look at the Network tab. You can see the individual CSS files are loaded and the image has the version hash behind the image name. Now let's change the Environment to Staging. When I now run the app, you can see the minified CSS file is loaded. In the next clip, you will see some more tag helpers in action when I work on the index view.

More Tag Helpers

In the index view for the conference controller, I make the view strongly typed, saying an IEnumerable of ConferenceModel is its model. I use Bootstrap's grid system to create a nice layout. I first add a Bootstrap row with a column. Next I use an HTML table and render the header. Then I use the good old HTML helper, Html. DisplayForModel to render all the conferences. The template for a conference I'm creating in the DisplayTemplates folder. Despite the presence of tag helpers now, some HTML helpers are still great to use together with tag helpers. The template contains the row for the table with all columns. The last column stands out because I used a tag helper here. The a tag uses the asp-controller and asp-action tag helper to generate a URL to the proposals of a conference based on the controller action name. The index action of proposal controller wants a parameter named conferenceId, which I also supply with the tag helper asp-route. After the final dash of asp-route is the name of the parameter and the value passed in is the Id from the model. Next up is the index view for the proposal controller. Let's add the needed folder in the view, which is similar to the index view of the conference controller. Note the a tag, which uses the asp-action tag helper to generate a URL to the Add action in the proposal controller, passing in the conferenceId. The ProposalModel display template is similar as well. It renders an icon if the model is approved with the help of a Razor if statement. The Glyphicons library used is part of Bootstrap. If it's not approved, it renders a link that executes the approve action, passing in the proposalId. To make the icon work, the final step is to copy over the fonts folder in the distribution folder of Bootstrap to the wwwroot folder. Let's run the app to ensure everything is fine, and it is. Let's add the screens to add conferences and proposals next.

Form Tag Helpers

There's a hyperlink in the form of an a tag that points to the Add action of conference controller in the conference view. I can omit asp-controller. In this case, the Add action is called of the current controller, which is the conference controller. I apply the same trick to the index view of proposal controller, only now I have to provide a value for ConferenceId. Slight problem here, because the context of the model is a collection of proposal models here, so I add the conferenceId to the ViewBag in the controller action, and then use that value in asp-route-conferenceId. Now let's create the Add view for the conference controller. This job is much more pleasant now because I can now just use the HTML form tag instead of the awkward-to-use BeginForm HTML helper. I can specify the action to call when the user submits the form with the asp-action tag helper. In the form, I have Bootstrap rows that have a label HTML tag, which also has the asp-for tag helper, and an input, which has it as well. I have this combination for each property of the conference. And finally, I'm rendering a Submit button, which doesn't need any tag helpers. The Add view for proposal is done in the same way. The only extra thing I've added is a hidden input, which holds the conferenceId. Now let's run the app to see if we can add conferences. You can see it works perfectly, but when I go to the proposals list of the new conference, you can see there's a bug. I forgot to filter the proposals in ProposalMemoryService. Now that I have applied a fix, you can see that adding proposals now also works without a problem. Let's take a look at the new setting system next.

Application Settings

The configuration system has been re-architected in ASP. NET Core, but it still uses simple key value pairs to store the values. In classic ASP. NET, it is very common to put settings in the Web. config file. In ASP. NET Core, the Web. config file is gone. With the new configuration system, the out-of-the-box configuration sources can be JSON, XML, and even INI, and environment variables are also supported. Microsoft has made an effort to decouple the actual settings on the configuration system itself. In other words, the settings can travel through the app on their own, and the traveling is of course done using the now essential dependency injection. When I discussed the Main method, I showed you that createDefaultBinder calls a method called ConfigureAppConfiguration. Here is the code for that method. It gets a hostingContext and a config object. The config object is of the type IConfigurationBuilder. You can give this object various sources for configuration settings, and it will compile all the data in these sources down to one key value list. When using CreateDefaultBinder, a file called appsettings. json is added as a configuration source. There is a second JSON file loaded in the same way called appsettings. env. EnvironmentName. json. Env. EnvironmentName is an expression that fills in the current environment name by using the HostingEnvironment object provided by the hostingContext. The configuration system just overrides the settings done earlier if duplicates are found. So first the appSettings. json file is read, and if a JSON file for a specific environment is present, which contains the same settings, it will simply override the settings. In this source code, you can also see that user secrets are added to the configuration container. User secrets are used to keep things like database passwords out of source control. Please see my course, Understanding ASP. NET Security for more details. Also, all environment variables are added to the key value list, as well as all command-line arguments. Let's see how to bring this knowledge to our app in the next clip.

Strongly Typing Application Settings

Now let's see how all of this can work in our demo app. I can just add an appSettings. json file and put in values. I don't need to configure anything to get it loaded because that is already done in the code that ConfigureDefaultBuilder executes. I want certain conferences to be displayed in bold when the number of attendees exceeds a certain value. I want that value to be configurable as a setting in appSettings. json, so I create a root for the app and put the value 3000 in it with the setting BoldConferenceAttendeeThreshold. Some features of ASP. NET Core for external libraries can also take their settings from the key value list. An example is a logging feature. I can configure it to log only when the log level is Warning or higher, like this. I want the Globomantics value to be available strongly typed throughout the app. Now I create a class called GlobomanticsOptions that has a property with the same name as a setting in the appSettings. json file. Now all I have to do is bring the class and the JSON file together. You can do that by asking the dependency injection container for an IConfiguration object, which contains the key value list. I'm doing that in the constructor of the Startup class, and I make the object available as a private field. In ConfigureServices, I can then call Configure on the IServiceCollection object, with as a generic parameter the class we just created. The method parameter accepts an IConfiguration object, so I could just pass the whole object in, but I want it to only take the app specific settings into account. So, I tell it to use a section of the configuration file. The section, in this case, is everything under the root for Globomantics. The effect is that our settings object is now available for injection because the property name in our object is matched with the configuration setting that has the same name. In the ConferenceModel view template, I can now inject an IOptions object of our Configuration class. We need to add usings for the namespace IOptions is in, as well as the namespace for GlobomanticsOptions. I'm now replacing the marker for AttendeeTotal with an if statement. A settings object is made available using its Value property, so I compared the AttendeeTotal of the conference to the BoldConferenceAttendeeThreshold value of the configuration object, and surround the AttendeeTotal with a strong HTML tag. And I'll leave the strong tag out when the threshold isn't met. Let's run, and you can see that it works. This is great to get started with settings without writing a lot of code, but what if you want to deviate from the way ASP. NET Core configures the settings system. In that case, go to your Program class and just add a call to ConfigureAppConfiguration on the WebHostBuilder, like this. It accepts a context and a ConfigurationBuilder object, and with that, you can add all the setting sources you want. The next clip is about view components.

View Components

In this clip, we're going to build the little statistics section on the Conference Overview page. The idea is that this is going to be a reusable component, so I can use it on other future pages of the app. The first thing that comes to mind is to use a partial view, but I have to calculate the statistics. Putting logic in a view like that isn't the best idea, and when I do the calculating in the controller, it won't be reusable. In previous versions of MVC, you might have used a child action for this, but this was a relatively costly operation because a child action goes through the whole controller lifecycle. A view component can help because it is like a partial view, but with a mini controller, which is more lightweight than a full controller. The mini controller can do the calculation, put the results in a model, and return a view, and I can render it in Razor syntax by calling component invoke async. MVC looks for view components in a folder with a surprising name, ViewComponents, so let's create that first. In it, I put a class called StatisticsViewComponent, the type derives from ViewComponent. It should contain a method returning an IViewComponentResult called Invoke. It returns an IViewComponentResult asynchronously, which is the view component version of IActionResult. The view model for the view is the statistics view model we created earlier. The IConferenceService has a method to get it, so I inject it into the constructor and return a view with the data in the Invoke method, just like I would do in an action of a normal controller. MVC expects the view in the Components folder. It makes sense putting it on the Shared because it's supposed to be reusable. First I create a folder called Components. Under that, I create a folder with the name Statistics. The suffix, ViewComponent, is omitted. I didn't specify a view name when I typed ReturnViewStatement in the ViewComponent class. In that case, MVC looks for a view called Default. cshtml, so let's create that. The view just renders the simple overview of the statistics. The last thing to do is to switch to the index view of the conference controller and render the view component by calling Component. InvokeAsync with the name of the view component as a parameter. Because it's an asynchronous method, we have to await it. When I run the app, you can see the view component appears, and the first version of the Globomantics app is feature-complete. But there is more. You can also let the InvokeAsync method of your ViewComponet exept perameters, and fill those parameters in when you render it. Let's make the ViewComponents caption computable using this technique. I first add a parameter to Invoke basic Just a simple string with a caption in this case. Then I put a value of it in caption of the ViewBag. In the ViewComponets view I swapped the static caption with a ViewBag caption. Now in the IndexView of conference where I rendered a ViewComponent I set the caption by supplying an anonymous object with a caption property. Using this way you can also pass multiple parameters if you want. When I now run you can see it works. There's another way to render ViewComponents. You can also use tag helpers instead of razor syntags to render them. Just by using vc: the name of your ViewComponent. You can pass in the parameters by using attributes. Note that when the parameter consists of multiple words with camelCasing you write them in the tag with dashes in between the words. When you want to render your ViewComponent with TagHelpers, you have to include the assembly of the project in the ViewImports. cshtml file.

Summary

In this module, you saw how to set up MVC using the Startup class. In that class, you registered the services of your app in the IoC container and set the pipeline up. Environments like development, staging, and production are set up using an environment variable and supported throughout ASP. NET, enabling you to load, for example, certain JavaScript files or plugin a piece of middleware only when a certain environment is active. MVC and web API controllers, as well as the other types from these frameworks, are now unified. MVC and web API together continue under the name ASP. NET Core MVC. Tag helpers are a great way to render HTML tags in an HTML-friendly way where you needed to use HTML helpers before. Application settings are now handled very different, but more flexible and suitable for dependency injection. And finally, we looked at view components, which enable reusable components in an efficient manner. In the next module, we're going to build the web API for the app.

Setting up a Web API

Module Overview

This module is about creating and accessing a web API with ASP. NET Core. We're going to change the architecture of the app. I'll start by showing you how. Then I'll talk about why you will want a web API. Next, some theory about REST, after which I'll show you how the web API is implemented. A cool feature of web APIs is that they can support multiple formats, and I will show you how to try the web API out and how to access the web API from a. NET app, in our case, the Globomantics web application.

Distributed Architecture

In the previous module, we created an architecture that looks like this. The browser generates a request that is processed by a controller. That controller uses an IService interface abstraction. We use dependency injection to inject an IProposalService, for example. The only implementation of IProposalService we had was the ProposalMemoryService, which maintained an in-memory collection of conferences. Database access is outside the scope of this course. If that was not the case, we could have of course used a database to retrieve stored conferences. In this picture, all of the functionality is in one application. The new architecture looks like this. We're going to create new implementations for the IService interfaces that will call a REST-based web service instead of using in-memory data. REST-based web services are also known as web APIs. The web API uses the exact same framework as our web application, so it also has controllers that process the requests, they are just not returning HTML, but data, and they get the data from something I call a repository, abbreviated as repo. These objects are also injected into the controllers with dependency injection using an interface abstraction. We will move the in-memory functionality to these repositories. In the next clip, I'm talking about the reasons for a web API.

Why a Web API?

But why should you want a web API instead of putting all the logic in the web application? First of all, you might have other applications that need the exact same logic to retrieve data. The web app might not be the only application available for your users. Maybe you also have a mobile app, for example. Using a web API, you only have to implement the data retrieval logic once. It makes it possible to share that logic across different apps, or maybe you want to give external customers a chance to create their own apps using your data logic, or integrate it into their existing apps. A reason could also be that the opinion of your team is that the web application should not access data, but only do stuff for the browser. A separate application should fetch the data for it, creating a clear separation between the two concerns. In practice, you're going to use a database of course. Having the web application retrieve the data means that the web application should have a connection string and maybe a username and password for the database. Since the web application faces the internet, that could be a security risk. If an internal web API manages the database connection, that risk is mitigated. Finally, in a bigger organization, you might have multiple teams, maybe front-end and back-end teams. It might be more practical to separate the logic for this reason alone. In the next clip, some REST theory.

REST

REST is an abbreviation, meaning Representational State Transfer. There are a few properties of REST explained by the Richardson Maturity Model, shown here. Web services using SOAP and other forms of remote procedure call are shown here as The Swamp of POX, where POX stands for Plain Old XML. Where SOAP is mostly ignoring the underlying protocol, REST is using the semantics of the transport protocol, which is typically HTTP. The first of the REST properties is that the methods in the service are not directly exposed. In other words, the caller of the service doesn't have to know how the service is implemented. This is in contrast with, for example, SOAP web services. Instead, all data, and data is known as resources, are available as specific URIs. What you want to do with the resource is partly determined by how you call the URI. An HTTP request is composed of a verb together with a URI. The most used verbs in HTTP are GET, POST, PUT, and DELETE. A pattern commonly used in REST is that GET verbs get you data, POST will introduce new data, PUT will update it, and DELETE, well you guessed it, will delete it. Hypermedia controls are on the next level. This is a way to get URIs from the service as part of the response. Hypermedia controls work kind of like hyperlinks in a web page. You can click to a next relevant page. For example, in the REST model when creating data with a POST call, the response returns a unique URI where the new resource is located. It's also common to have a well-known URI that exposes some starting points in the service. In this way, it's possible to discover the URIs while using the service, and to serve the service, so to speak, without knowing all the URIs up front. This mechanism is known as HATEOAS, Hypertext as the Engine of Application State. In the next clip, we're implementing the Globomantics web API.

Implementing the Web API

Because ASP. NET Core and the web API framework are now consolidated in ASP. NET Core MVC, setting up a web API project is very similar to setting up a web project. Just start an empty ASP. NET Core project like we did with the web project and build it up like before. I'm only covering the things that are different in this module. I've already created a project simply called API. In ConfigureServices in the Startup class, I register three different repositories, one for the conference, one for the proposal, and one for the statistics. The implementation of these is the same as the services we created in the previous module, so they use in-memory data. One notable fact is that the statistics repository asks for an IConferenceRepo to do its work. This just works. Dependency injection is not limited to the controller. The container will also support all classes that the controller uses, and all layers below that. GetStatistics uses the conference repository to get all conferences, and then calculates the statistics. Back to the Startup class. In the Configure method, you can see that it appears that I use MVC without any form of routing. I am using attribute routing for this one. The conference controller uses the route v1/controller. Words in square brackets are expressions. In this case the controller is called ConferenceController, so the word conference without the controller suffix will be used here. Using an expression like this will ensure that there will never be a mismatch between the route and the controller name. The action is not mentioned in the route. That's what we want, because a REST service should respond to HTTP verbs. If we mention the action here, the calling party must know the implementation details of the web API. That is like remote procedure call and SOAP services. That is not what we want here. The v1 suffix is a way to version your web API. If version 2 comes out with altered controllers, you can change v1 to v2, and then you can make both v1 and v2 available at the same URL providing backwards compatibility. ConferenceController gets the conference repo injected and has a GetAll action. It returns an IActionResult. When no conferences are found, it will return a NoContentResult, which will result in an HTTP status code 204, giving the caller a clear indication that nothing was found. If there are conferences, an ObjectResult is returned. This will return an HTTP status code 200 response with the conferences in the request payload. This action responds to the GET HTTP verb. That's the default. If you don't specify it otherwise, it will be available with a GET. Also, there's a naming convention and an action that starts with GET will be accessible with a GET. If it starts with the word POST, it will be available with a POST, etc. The next action is called Add. That doesn't start with the name of an HTTP verb, but I wanted to respond to a POST since it inserts new data, so I've put the HttpPost attribute above it. It accepts a ConferenceModel object as a parameter and it returns it. This is normal in a web API. The conference is probably changed by the adding, in this case an ID is determined by the repository, so the only way to let the caller know what the ID is is to return it. The Add action doesn't return an IActionResult, but the new ConferenceModel directly, which is supported. However, this maybe isn't the best solution. If all goes well, there's no problem. The HTTP status code 200 will be returned with a new conference model. But what if an exception occurs? In that case, the action will always return an HTTP status code 500, internal server error, a very generic error message that doesn't give any hints to the caller what might have gone wrong. That's why the proposal controller has a different approach for the Add action. The IActionResult object it returns is called CreatedAtRoute, which will result in an HTTP status code 201, which means created. As we will see in a minute, it also returns a URL where the new proposal can be retrieved. This is level 3 of the Richardson Maturity Model in action, hypermedia controls. The URL is generated by specifying the action GetById for the current controller. The parameter needed, in this case proposalId, is also supplied. Apart from the URL, the proposal itself, of course, is also returned. Also notable with the Add method is the FromBody attribute before the parameter. This forces the model binder to go look in the request body for this data. The rest of proposal controller is similar to the conference controller, except for the approve action, which responds to an HttpPut verb. To let the routing system know it has to look for a supplied proposal ID in the request, I have supplied a proposalId expression as a parameter to the HttpPut attribute. This will be matched with a parameter the action has. The Approve action will try to approve the proposal. If that works, it will return the approved proposal. If it generates an InvalidOperationException, it will assume that the proposal was not found and return an HTTP status code 404. You should make this error checking more comprehensive in production scenarios. This is just to give you an example of how it is done. The implementation of statistics controller is very simple. It has just one action that gets the statistics from the repository. In the next clip, I'm talking about content negotiation.

Content Negotiation

Let's start the web API from Visual Studio. I'm using the console app profile here to run it. Before we access our new web API from the web application, let's first see what it does. I'm using Postman here, which is free for individuals and small teams. You can download it from getpostman. com. The web API runs on port 5000, so I generate a GET request to localhost, and I'm hitting v1/Conference. When I execute the request and select the Raw tab, you can see the list of conferences comes back in JSON format. When I take a look at the headers of the response, you can see the service notifies me of the fact that JSON is returned with a content-type header. Typically, web APIs support content negotiation and can support multiple data formats. Using the Accept header, I can request a certain format for the return data. I didn't do that for the request I already did. In that case, the first format available is used, in this case JSON. Let's do the request again, but now with an accept header requesting XML. When I execute the request, you can see JSON is still returned, and that is because ASP. NET Core just supports JSON out of the box. If the requested format is not supported in the web API, it will still use its first, and in this case, only format available. To support XML, I install the NuGet package Microsoft. AspNetCore. Mvc. Formatters. Xml. In the Startup class in the ConfigureServices method, I can now use an extension method on AddMvc called AddXmlSerializerFormatters. When I now run again and issue the same request, XML is returned. We just added a media formatter to the API. Many media type formatters can be added to your API, like this one, using a variety of formats. You can also create them yourself to add support for a proprietary format, for example. In the next clip, we're doing some further investigation of the output of the web API.

Examining Web API Output

Let's try to add a proposal using Postman. As a reminder, the code in the controller looks like this. The route of GetById is a named route. I refer to this route in the CreatedAtRoute result, passing in an id. Let's see what happens when I generate a request for this action. I have to set up a POST request because the action responds to that HTTP verb. I use the v1/proposal URL. Remember, you don't specify the action name here, REST services should respond to HTTP verbs instead of action names. In the body, I include the new proposal in JSON format. I'm adding a new proposal for conferenceId: 1 here. I have to let the web API know that I'm sending JSON by setting the Content-Type header to application/json. When I execute this request, the response has an HTTP status code 201 Created. The new proposal is returned from the service with a fresh ID. And when we take a look at the headers of the response, there is a location header pointing to the new proposal. This is what I meant by level 3 of the Richardson Maturity Model, hypermedia controls. This is just a limited example. In practice, you should supply the caller to the web service with as much hypermedia information as you can. In the next clip, we're going to access the web API from the web application.

Accessing a Web API from a .NET Application

Let's modify the web application so that it calls the web API. I've created another implementation of the IProposalService interface. The new class is called ProposalApiService. It asks the dependency injection container for an HttpClient object, which I have registered in the Startup class. The registration is slightly different this time. I'm telling the dependency injection system how to construct the HttpClient object by specifying a lambda. In the lambda, I'm just newing up the HttpClient within the constructor the base URL for the web API. I put it in as a plain string here, but it's a good idea to get it from a configuration file instead. Let's take a look at the code of GetAll in the proposal API service. It uses the HttpClient object to create a GET HTTP request to v1/Conference, which will trigger the GetAll action in the web API, returning the list of conferences. The request is done asynchronously. As soon as a response comes back, I'm checking if the status code indicates success, in this case a 200. If that's the case, I'm pulling the result out of the response object by using an extension method from the NuGet package we installed earlier. ReadAsAsync has a generic parameter specifying to which type it should convert and receive data. I'm returning the IEnumerable of ConferenceModel later on in the method. If there's no success status code, I'm throwing an exception with a reason phrase. This is a description of what went wrong. This is a simplified version of error handling. In production, you probably need a more comprehensive way of handling errors. Implementation of all methods that have to get the data from the web API is done in the same way throughout the web application. The Add method uses PostAsJsonAsync to do an HTTP POST to the supplied URL. The body of the request will contain the conference model. The Approve method generates a PUT request to the proposal controller. You might remember that we marked this action with the HttpPut attribute. So that was ProposalApiService. I also created a ConferenceApiService class implementing IConferenceService. In its implementation it uses all the concepts I explained with ProposalApiService. Nothing new. Now if I want the web application to work with the web API instead of the in-memory data, I can just go to the Startup class and tell the dependency injection containers to inject ConferenceApiService instead of ConferenceMemoryService if a class asks for IConferenceService. In the same way, I change ProposalMemoryService to ProposalApiService. This is so easy to do because we don't have a direct dependency on the concrete service implementation. There is a layer in between, an abstraction in the form of an interface. Before we hit Run, be aware that if both web application and API are started using their console mode, they will both try to use port 5000, which won't work. So I'm configuring the web application to use port 5001 by using the UseUrls method on IWebHostBuilder in the program class. When I now run the web app and web API together, you can see it works as before, only now requests are generated to the web API.

Summary

Here's a summary of this module. A web API is a web service that we call REST-based. Whether you create a web app or web API, in ASP. NET Core you use the same classes. Despite that, the structure of a web API is different. A web API can support multiple formats. You can use content negotiation to pick one. The testing of a web API is easily done with Postman, and to access the web API from. NET code, HttpClient is the class to use. In the next module, we're taking a closer look at. NET Core and how it works.

Developing Applications Across Frameworks and Operating Systems

Module Overview

Where the previous modules were more hands on, this module contains a bit more theory. You'll learn what the roles of the full. NET Framework,. NET Core, and the. NET Core CLI are, and it will also become clear why. NET Core applications can run cross-platform where full. NET Framework applications cannot. In a moment I will start by explaining how the. NET Framework developers have used since the beginnings of. NET works. You will also learn about Mono and. NET Core and how they are different from the traditional framework. Code sharing is done using the successor of PCLs, a. NET Platform Standard, and I will give you some guidelines helping you to decide which framework to use. Lastly you will see how the. NET Core CLI runs your ASP. NET Core application.

The Traditional .NET Framework

Since the beginnings of. NET, the mechanics around running an app are the same. It doesn't matter if you have an ASP. NET app, a WPF app, or a Windows Form app, there's always some kind of bootstrapper that invokes. NET. In a desktop or console app, this is the executable file itself, which is started by Windows. In a traditional ASP. NET app, this is IIS using an ISAPI DLL. The app you have written in C#, or another language supported by. NET, is compiled in Visual Studio to an assembly. An assembly is an EXE or DLL containing Intermediate Language, or IL. OSs and CPUs don't understand IL, so on the machine the assembly is deployed to, the IL still has to be compiled to native code, and that is called JIT compiling. JIT stands for Just In Time. The IL is compiled to native code on the machine the assembles are deployed to just before it executes. The JIT functionality is contained in. NET itself, in the part that is called the CLR, or Common Language Runtime. The CLR is reasonable for almost all functionality in. NET, such as the loading of assemblies, the checking of types, and garbage collection. So the. NET Framework is something that has be installed on the machine your app runs on or else it simply won't run. The other part of the framework consists of a big stack of classes and other types. That's in the part that is called FCL, or Framework Class Library. It contains all types needed for Windows Forms, WCF, WPF, Web Forms, as well as types usable across these frameworks, like file handling, reading and manipulating XML, drawing, and cryptography. All apps use some of these classes. In a Web Forms app, for example, all of the components like buttons, checkboxes, etc. on the form are in the FCL, and also more basic classes like the ListCollection class are in the FCL. The advantage of this is that your app only has to contain the stuff you write. Everything else is already installed when the. NET Framework was installed. The CLR is specifically designed to run on Windows. Also, some of the classes in the FCL are specifically for Windows. System. Web is an example of an assembly containing classes that are tied to IIS, and therefore Windows. But next, you'll learn about another. NET Framework called Mono.

Mono

Mono is an open source version of the. NET Framework developed by the community. It uses the exact same principles that Microsoft. NET Framework uses, and was started before Microsoft started its open source journey. It's compatible with the Microsoft. NET Framework. You can create assemblies on a Windows machine using the Microsoft. NET Framework on Visual Studio and run them on Mono on a Linux machine. So an important difference with the Microsoft. NET Framework is that it is cross-platform. The versions are available for Windows, Mac OS, and Linux. It's also used as the basis for Xamarin, which runs. NET on Android and iOS. In the next clip, I'll talk about NuGet.

NuGet

NuGet is essential for. NET Core, the topic for the next clip. So before I go on, it's important you understand it. Having everything you need inside the framework, like the Microsoft. NET Framework or Mono, is an ideal scenario, but in practice it's almost always practical or needed to use some library of classes that is not in the FCL. So you could just download third-party DLLs from the internet and ship them together with your app. To make things easier, NuGet was introduced by Microsoft. NuGet is a central place to get these libraries, and Visual Studio has tooling to search, download, install, and update them. NuGet has become the platform to quickly introduce new libraries and even whole frameworks. Developers of these libraries and frameworks can easily apply new versions or bug fixes to NuGet, and as a developer working on an app, you can easily update them with one click in Visual Studio. Microsoft, as a developer of the FCL and the CLR in. NET Framework, doesn't have the luxury to roll out new versions and updates so easily. Every time they wanted to introduce something new or update stuff, they had to release a whole new version of the. NET Framework or update an existing. NET Framework, and that's something you don't want to do very often because you know or can imagine it's a hassle to do the update on every single machine running. NET. So Microsoft began to use NuGet for assemblies that would normally maybe be in the FCL. An example of a framework using this principle is MVC. MVC is installed as a NuGet package in the application, and not machine-wide like the FCL. This enables different applications to use different versions of MVC without the need to install different versions of the. NET Framework, and distributing MVC via NuGet gave Microsoft the ability to update MVC out of band with the. NET Framework, which enabled MVC to evolve much faster with more frequent updates. And that was the upbeat of a completely modular framework class library with. NET Core, and. NET Core is the topic for the next clip.

.NET Core: CoreFx and CoreCLR

Through the years,. NET has been reinvented multiple times. This slide just shows the current versions, not even the ones that are phased out already. But wouldn't it be a lot easier if we have a version of. NET for all platforms, not only for Microsoft that has to keep all these tags up to date, but also for us developers that have to learn and maintain all these versions. The purpose of. NET Core is to be the one. NET version that rules them all, and it all starts with ASP. NET. Another motivation for. NET Core is the need to reduce the overall footprint. From an ASP. NET perspective, using System. Web isn't really an option anymore. Also having a huge. NET Framework on a machine that causes versioning problems and contains lots of stuff you don't need, it's cumbersome. And in this cloud-driven world, being fixed to Windows isn't of this time anymore. To me, the most exciting feature of. NET Core is that it can run across operating systems. Just like the full. NET Framework,. NET Core also consists of two parts, a common language runtime, which is now portable, that goes by the name CoreCLR, and a class library called CoreFX. CoreFX contains a set of types common across every. NET application. It doesn't include complete frameworks such as WPF or Web Forms, like the FCL in the full. NET Framework. There are, for example, classes to manipulate files, and classes that are collections, like a list. The different assemblies of CoreFX are individually distributed via NuGet. Everything else you need outside of CoreFX you have to get from NuGet as well, such as the MVC framework.. NET Core is what Microsoft calls cloud optimized. It basically means that it's lightweight. It is not only much smaller than the full. NET Framework, it also contains optimizations. Running an ASP. NET app has never been faster and more efficient in terms of resources. It's also great that. NET Core is open source software. That means besides Microsoft, the community is working on it as well. Let me talk about code sharing next.

Sharing Code with the .NET Platform Standard

I talked about how great it would be if there was one. NET Framework for all application types, but that's sort of a utopia right now. In reality, there are different. NET versions that won't go away soon. The ability to share code across these platforms is a must. Let's say you are developing an app that has to have both a web and a mobile front end, then you could be writing an ASP. NET Core application and at the same time a Xamarin application. Of course, we want to share as much code as possible between the ASP. NET Core application and the Xamarin application. The first thing that comes to mind if you want to share code is to create a class library assembly in the form of a DLL, but the assemblies are bound to a specific. NET version they are compiled against. So a DLL compiled against a full. NET Framework 4. 5 will not work in, for example, Xamarin. Why? Because Xamarin uses a different. NET Framework, and thus has a different API. So it could be that you use a. NET API in the assembly that isn't supported in Xamarin. For that reason, Microsoft came up with PCLs. A PCL is a Portal Class Library. It's just like a normal DLL assembly. The difference is that you specify which platforms it should support in advance. Visual Studio will take that into account while providing you with an API while you're programming. The smallest API of the selected frameworks will be provided. The problem with PCLs is that they are tied to a specific version of the platforms they are built against. So when a new version of a platform comes out, you have to rebuild and redeploy the assembly to support that version.. NET Standard can be seen as a successor to PCLs. When creating a class library, you are now binding your assembly not to a specific platform, but to a. NET Standard version number. Every. NET platform, legacy, current, or future, will implement a certain version of the. NET Standard. The version number of. NET Standard represents a certain API. Because assemblies targeting a. NET Standard version number are not directly bound to platforms anymore, new versions of platforms can come out, it will always be backwards compatible with the version you're using without the need to rebuild and redeploy your assemble. So let's say you've built a class library for. NET Core, implementing. NET Standard 1. 6, and you want to reference it in a project that is built against a. NET Core version that implements. NET Standard 2. 0. This will just work because the runtime knows that 1. 6 is 100% backwards compatible with 2. 0. The lower the. NET Standard version number, the less API it has, but the more. NET platforms you're able to support. Let's imagine that the API for version 1. 0 only contains a way to write to the console. This is not really the case, of course, but let's imagine. Every platform has it implemented, so every platform is able to support version 1. 0. Programming such an application won't be much fun because Visual Studio will only let you program with the console. The way you use version 1. 0, you will miss out on the cool, newer APIs of newer versions. But, every. NET platform will be able to use your assembly because all platforms are backwards compatible with. NET Standard 1. 0, and when you target the newest version of. NET Standard with your assembly, you have the latest and greatest API available, but only up to date or recent. NET platforms will be able to support it. The step from version 1. 6 to version 2 of. NET Standard is huge. The number of available APIs has gone from around 13, 000 to 32, 000. Most of the additions were APIs that were already available in the full. NET Framework. When only. NET Standard version 1 was around, a lot of library creators were reluctant to bring their product to. NET Standard. It was simply too much work. Now that. NET Standard 2 offers an API that is a lot closer to the full. NET Framework, it has become much easier to port existing libraries to. NET Standard. But not all libraries available in NuGet target. NET Standard, and that's why there's a compatibility mode that allows full. NET Framework libraries to be referenced in. NET Core applications, as long as the API the library uses is available in. NET Core. You saw that in action when we added the NuGet package with the extension method on HttpClient. Let's see how to configure frameworks in an ASP. NET Core application in the next clip.

Configuring Multiple Frameworks

Let's go back to the ASP. NET Core world. Here's the project file of the web app again. In the TargetFramework node you can see there's one platform targeted, netcoreapp2. 0. This means, obviously, a. NET Core 2. 0 application. Here's the project file of the shared DLL we created as a. NET Standard library. It targets. NET Standard 2. 0. That means that it can be referenced from all frameworks that support. NET Standard 2. 0. So. NET Core 2. 0 can use it, but also the full. NET Framework 4. 6. 1, and Xamarin. iOS 10. 14, for example. And because. NET Standard is always backwards compatible, it will also support all versions that come after these version numbers. Let's say we need this library to work with full. NET Framework 4. 5. 1. It won't work because 4. 5. 1 is not compatible with. NET Standard 2. 0. In that case we can target multiple frameworks at the same time. They are specified using monikers, which are just strings that stand for a version of a framework. To let this library support multiple frameworks, change the TargetFramework node to plural, TargetFrameworks, and then separate the different frameworks with semicolons. So here I add net451, which stands for the full. NET 4. 5. 1 framework. When you do this, the API you can use in the application will be automatically limited to what the framework with the smallest API offers, so in this case,. NET Framework 4. 5. 1. When I save this, all NuGet packages specified in the project file are now restored for both frameworks. To illustrate, I'm installing the AutoMapper NuGet package. On the dependencies in the project, you can see that there are sections for each framework. It's also possible to install certain packages for specific frameworks. There are a couple of things you have to take care of, and the tooling doesn't really support you when you target multiple frameworks. To learn about all the ins and outs, please see Rick Strahl's blogpost at tinyurl. com/multiframeworks. In the next clip, we'll look at the. NET CLI.

The DotNet CLI

. NET Core comes with a command-line tool called the. NET Core Command- Line Interface, or. NET Core CLI. The primary role of it is to run your. NET Core app. For other frameworks you have to use some other way to run your app. Mono, for example, also has CLI tools to start your app, and when building against a full. NET Framework, the output of your application will be an executable that Windows recognizes as a. NET app, for which it will host the CLR for you. The executable part of the. NET Core CLI is contained in a command-line application simply called. NET. In order to run your application, it needs the ability to execute Intermediate Language, or IL. So instead of Windows or IIS executing the app, the. NET command-line application having its own process executes it. A. NET application will need a CLR, and the. NET Core CLI will host it for you instead of the operating system in earlier. NET versions. The. NET Core CLI will look for an entry point in the form of a Main method and run the application from there. Because a separate native process now runs the app instead of the Windows operating system, the. NET Core command-line application can be developed for multiple operating systems and CPU platforms. There are implementations for Windows, Linux, and Mac OS, and on Intel and ARM CPU architectures. So now the app you developed in Visual Studio can run cross-platform, and as an ASP. NET developer, you have a front row seat because ASP. NET apps are among the first application types that can run with the. NET Core CLI. The. NET CLI can not only run your apps, but also, for example, restore NuGet packages that are specified in the project. json, so it has more tasks to help you with the development of your app. It is also an SDK, or Software Development Kit. When you run a. NET Core app in Visual Studio, all Visual Studio does is execute the. NET Core CLI. When you press Run, for example, it will execute dotnet run, which will put the. NET CLI to work. The run command is called a verb. Its logic is present in an extension that will be executed by the. NET Core CLI. In this way, the. NET CLI is easily extendable with other commands. There are, for instance, extra verbs available to support migrating databases in Entity Framework. Let's look at the built-in verbs. We already talked about dotnet run, which will build the application and then execute the assembly while it hosts the CLR for it. While running the app, debugging is also supported. The. NET Core CLI executes all applications like command-line applications. It starts them by executing the Main method, and the application can then set up and configure other frameworks, like ASP. NET. The. NET CLI also has SDK commands like dotnet restore, which will download the required NuGet packages. That command is also used by Visual Studio. And there are also commands for just building the app and publishing the app, among others. The. NET Core CLI is just a simple command given on the command line, so it doesn't really need Visual Studio to execute. You can also execute it directly by just typing in the commands or using some other tool to invoke it. Let me do a small demo with the. NET Core CLI. When I type dotnet -h at the command prompt, you see the verbs I can use. One is the new verb, which can produce many sample applications. When I type dotnet new, you can see the options. One option is console, which produces a very simple. NET Core command-line application. Let's try that one. There are two files, Program. cs containing a Main method that just outputs Hello World, and a project file, which targets. NET Core 2. 0. As part of dotnet new, the packages needed are already restored, but you could also explicitly type dotnet restore. Now I can just type dotnet run. This will build the project and run it using the project file and source code.. NET CLI will host the CLR for this application. Now when I go to the bin folder of the tiny application, and then to the Debug folder, you can see that. NET has created a folder for the target framework. You can see there is no EXE file, just a DLL. That is because the. NET CLI is supposed to start here, not Windows. To run the application using the DLL, just type dotnet with the DLL name.

Summary

In this module, you learned what the difference between the full. NET Framework,. NET Core, and Mono is. All three have advantages and disadvantages, and I talked about these. We also talked about the. NET Platform Standard and why it is important if you want to share code across the different. NET platforms. And I explained what exactly the. NET CLI is and the role it plays in developing. NET apps in a completely new way. The next module is about publishing your ASP. NET Core applications.

Deploying ASP.NET Core 2 Applications

Module Overview

This module is about deploying ASP. NET Core applications. First, I'll go into the differences between Framework-Dependent, or FDD, and Self-Contained, or SCD, deployments. Then you'll learn what Visual Studio does when you publish a framework-dependent app. After that, two excursions about the runtime store and Kestrel and native dependencies. That knowledge will come in useful when we're going to take a look at publishing a self-contained app. Next I'll show you how to deploy to different operating systems, first IIS and Windows, then Microsoft Azure, and finally, Linux. After that you can practically deploy to any supported platform because the basics are the same every time. But let's start with the differences between framework-dependent and self-contained deployments.

Framework-dependent and Self-contained Deployments Differences

After writing a. NET Core app, you can choose if you want framework-dependent, also known as portable, or self-contained deployment. Framework-dependent means that the deployment relies on the presence of. NET Core on the machine it runs on. We know that concept from the full. NET Framework. The only difference is that there is no huge framework class library present. The other option is to have a self-contained deployment. Using this option, there is no machine- wide installation needed of. NET Core, since. NET Core itself will be deployed as part of your app at the time you publish it.. NET Core with CoreCLR and CoreFX, as well as the runtime that is needed to run your app are all in your app's deployment. Here are the advantages and disadvantages of a framework-dependent deployment. You don't have to configure the operating systems you're deploying to up front because the deployment will run on all supported operating systems. All output files are assemblies that the preinstalled. NET CLI can handle. The deployment size is small, because. NET Core is assumed to be in the target machine. The disadvantages are that the correct. NET Core version much be preinstalled. That could cause problems when different apps use different versions of. NET Core. Concerns could be, will the different versions play well together, and, is the newer version really backwards compatible? With a self-contained deployment, you as the app publisher has full control about the. NET Core version used because it is part of your app's deployment. So there's no question about if the app will run with its. NET Core version. Also, side-by-side execution of apps using different versions of. NET Core just works. Disadvantages are that you have to know and configure the operating systems your app is supposed to support up front. The size of the deployment compared to a framework-dependent deployment is quite a bit larger, so each deployment will require a lot more disk space. Let's do a framework-dependent deployment in the next clip.

Framework-dependent Deployment

I'm going to deploy just the web app in this module. So I've switched from the API services back to the in-memory services. Let's see how to publish your app from Visual Studio if you want a framework-dependent deployment. Right-click on the project and select Publish. Next, choose Folder. Then I have the option to specify a path where the app will be published to. Press the Publish button and Visual Studio will execute the. NET CLI with the publish command. After it's done, the publishing profile is saved, and you can modify it by pressing Settings. On the Connection are the settings we already chose, and on the Settings you choose which Visual Studio configuration to publish, and you can select the Target Framework version. You will only see options here that are configured under the Target Framework section in the project file. The final option, Target Runtime, is only used in self-contained apps, and is now fixed to Portable, which is your framework-dependent app. In a folder where we did the deploy to, you can see there's no executable, just the DLL. There are also no DLLs containing. NET Core. This is because we deployed the app as a framework-dependent app; that's the default. It is like publishing an app for the full. NET Framework. It is assumed that I have. NET Core preinstalled on the target machine. Like the command-line application I demonstrated earlier, I can execute the application by opening a command prompt in this folder and then type dotnet, followed by the DLL name. You can see Kestrel being started. In the next clip, we'll take a look at the runtime store.

The Runtime Store

When we take another look at the folder we deploy to, there is no sign of the metapackage; that's the big package that I talked about when we began developing the app containing everything needed for an MVC app. The metapackage is a package with a lot of dependencies. You probably won't need everything it contains. So it seems like a waste to deploy this very big package along with your app. The whole point of having many small packages is that you could include only the stuff you need. Luckily, we don't have to include the metapackage in our deployments, thanks to the runtime store. It has similarities with the Global Assembly Cache, or GAC, that the full. NET Framework uses. The runtime store is a centralized location for NuGet packages. Everything that is inside the store doesn't need to be deployed with an application, it is just assumed that it is already on the target machine just like the full framework assumes that all assemblies in the system namespace are already on the target machine. The ASP. NET Core metapackage was already installed when. NET Core 2 was installed on the target machine, so it doesn't have to be redeployed with your application anymore. The number of files copied over to the publish target is significantly decreased this way. Packages inside the runtime store are typically ngen.. NET normally compiles assemblies to binary code just before the application runs, called Just In Time compilation, or JIT, causing a slow first start of the app. Everything in the runtime store is pre-JIT'd using ngen. Ngen is a tool to do the pre-JIT'ing. So this should make your app startup faster. It is also possible to add your own NuGet packages to the runtime store. For that, you'll have to create a manifest, which is basically a csproj file listing all included packages. Using the. NET SDK, you can then add these to the store. And when you're publishing an app, you can then specify which manifest it should take into account. Everything that is inside the manifests is then not published with the app, it is assumed to be in a target machine already. In the next clip, I'll talk about native dependencies and Kestrel.

Self-contained Deployment

I mentioned Kestrel is a very performant, lightweight web server that is contained within your application. It consists of a. NET assembly, but also a component that is native for a certain operating system. That native component, in the case of Kestrel, is an open source library called libuv. We should include that native component when we're deploying the app. So now we have a problem, because a frame of dependent deployment can be used on many operating systems and CPU architectures, and they all need their own version of libuv. Let's see what the. NET CLI did when it published the framework-dependent application. You won't find libuv in the output directory of your publish. That's because libuv is included in the runtime store, together with the metapackage and all dependencies. But what if you're using other packages in your applications that have native dependencies? For these, a folder will be created called runtimes, containing child folders for each supported RID. RID stands for Runtime Identifier. An RID is a combination of an OS and a CPU platform. Inside the RID folder, there is a native folder, which contains the native bits for that OS/CPU combination. So when your application has a native dependency and it is published framework-dependently, the. NET Core CLI will just publish the native files for every RID it supports. You can find the list of supported RIDs at tinyurl. com/netcorerids. Let's see how publishing works for a self-contained app in the next clip.

Rolling Out to a Platform

When publishing a self-contained app,. NET Core and the runtime will be distributed with your app and no installation of. NET Core is needed on the server that runs it. Also, the publishing process will provide for a way to start your app. Since ASP. NET Core is cross-platform, in the case of a self-contained app, it must be known to which RID you are planning to deploy before publishing. That's because there's now a part of your application that has to run natively on the platform, the. NET runtime, and all of its native dependencies. That's why I'll need to add a RuntimeIdentifiers node in order to deploy in self-contained mode. If RuntimeIdentifiers is present, your application will be automatically deployed as a self-contained app. In the RuntimeIdentifiers node, I have to list all the RIDs I'm planning to deploy to. I add the two RIDs I'm going to need for this module. One is the Windows 10 platform running a 64-bit CPU, and one is the RID for Ubuntu Linux on 64 bits. Now let's add another publishing profile with a different output directory. On the Settings, you can now see that I have a choice for which configured runtime, which is in fact the OS, I want to deploy. These are the configured values in the project file. I choose Windows 10 explicitly here and publish the app to a Windows subfolder. When looking at the output directory, you can now see that we have a lot more files. All the native parts of. NET Core itself and all dependencies are also published, and we have a native executable that contains the. NET runtime. The native dependency needed for Kestrel, libuv, is also present, and all assemblies in CoreFX are there as well. Because the app has to be self-contained, there is no dependency on the runtime store, so all packages we depend on are published, including everything that is in a metapackage. When deploying self-contained apps, using the metapackage could be a disadvantage. It publishes a lot of DLLs you maybe don't need at all in your applications. If deployment size matters to you, it is maybe a good idea to just specify every dependency in the project file explicitly, just like in version 1 of. NET Core. I can just execute the EXE by double-clicking on it. So to execute a self-contained app, the. NET CLI is not needed because the publishing process provides for a way to execute the app natively in the OS. Let's publish again, but now for Ubuntu using a different profile configured with a separate folder. You can see the same effect as the Windows publish we did, but now. so files are generated, which are native to Linux. Again you can see that the native dependency to libuv is part of the deployment, as well as all assemblies. Let's see how to deploy the published output in the next clip.

Deploying to IIS

This is going to be the shortest clip ever because once you have run either a self-contained or a framework-dependent deployment, you can just copy over the output files to the operating system and run. Really, that's it. Of course with a framework-dependent app, you have to have. NET Core preinstalled. Running an application by itself, like I did so far, always runs just Kestrel as a web server. In production, you need more features than Kestrel gives you. You want to configure security or load balancing. Kestrel is just a very fast, but simple web server that doesn't support these features. You need a full-blown, production ready web server for that. So in the next clip, we'll look at how to deploy to IIS.

Configuring IIS

Making use of a full-blown web server like IIS will give you a lot more configuration options, but also, running just a command-line app could be a problem in production. What if the app crashes? While using IIS as the front-end web server, IIS will also do the process monitoring required and restart the app if necessary. Let's take another look at the output folder of the framework-dependent deployment we did. There's a web. config here. That's strange, because I mentioned earlier that ASP. NET Core doesn't use the web. config anymore. That statement still stands. The web. config is meant for IIS. It configures it. It is there because the default web host builder configured IIS as the external web server. The web. config registers a handler called AspNetCoreModule, and in the aspNetCore configuration node, it tells it to start the application using the. NET CLI where there's argument to the DLL name. In the web. config for the self-contained app, the same module is registered, but the. NET CLI is not used. Instead, the executable is mentioned. The first thing you want to do when you want to deploy to an IIS server is to make sure that AspNetCoreModule is installed on the machine you want to deploy to. You can download it from GitHub, and the package you download also includes. NET Core itself, though you can deploy framework-dependent apps on the server if you want. So what is AspNetCoreModule? It facilitates something called a reverse proxy process. When the request comes in, IIS can proxy the request to a separate process. That means it can just take the request and forward it to another process, after which IIS gets a response from the process and sends the response back. So this is great for your ASP. NET Core application, which is a command application that can handle HTTP requests. The cool thing is that IIS will manage the process for us with the help of AspNetCoreModule. It will start and stop the command-line application with Kestrel when needed, taking care of concurrency, etc. It does all that in the background. In the IIS management application, your app is just another website, and you can configure it using all the great features in IIS. So let's configure the app in IIS. The configuration for framework- dependent deployments and self-contained deployments is identical. Start the IIS Manager, right-click on the root server node, and choose Add Website. Type in the name of your app. An application pool will be automatically generated for this app. In the Physical path textbox, specify the path to the location of the published app. I've copied over the portable version of the published app to this location. As a binding, I use port 8080, and press OK. Now go to the App Pools node and double-click on the application pool of your app. On the. NET CLR version, choose No Managed Code. That might sound strange, but remember the app now has the CLR running in an exclusive process. It's still managed code, we just don't need IIS to host the CLR for us. When I hit OK and point my browser to localhost port 8080, you can see the application runs. So now I can use all great features of IIS for my ASP. NET Core app, just like any other web application because it runs in front of it. If you don't like copying over the project and configuring it by hand, there's also the Web Deploy option in Visual Studio. With it, you can deploy straight to IIS from Visual Studio. IIS has to be configured with the Web Deploy addin, and the right ports must be opened in the firewall in order to make this work. In the next clip, you'll see how to bring the app to Microsoft Azure.

Deploying to Microsoft Azure App Services

Deployment to Azure is easy using its App Services. Your application will run on a virtual machine in the cloud. You can't see it, but the virtual machine runs IIS, and AspNetCoreModule is already installed, so the deployment process used in the background is basically the same as deploying to IIS on the local server. A new site has to be created in IIS, and the published content has to be uploaded to the server. All this can be done by creating another published profile in Visual Studio. Instead of File System, I now choose Microsoft Azure App Service. It might be necessary to log in to your Azure account first. You can then create a new app in Azure, giving it a name. The URL will be that name with. azurewebsites. net behind it. Subscription lets you select your Azure subscription. Resource Groups are containers needed to, for example, group all services for one application, and the App Service plan is basically the virtual machine you are deploying to. You can house multiple web projects on the same App Service plan, and the App Service plan is the thing you pay for in Azure. When I press Create, the web app in Azure is created, and when I then click Publish on the newly-created publish profile, a few moments later the app is running in the cloud. See for yourself. Direct your browser to aspnetcoreglobomantics. azurewebsites. net. Of course in the Azure portal, you have all these great cloud options available at your fingertips. For example, you can scale out by setting a value on a slider. This will make your app run on multiple virtual machines with an automatic load balancer in front of it. So by setting it to two instances instead of one, it can handle a lot more requests. Autoscaling based on, for instance, a CPU load, is also available. I'm going to deploy to Linux right after this clip.

Deploying to Linux

I'm focusing on deploying apps to Linux in this clip, but it's also possible to develop the app itself using Linux or other operating systems, not being Windows. Visual Studio Code and JetBrains Rider are examples of development environments with an increasing number of features to support this. Here I am on Ubuntu Linux. I just copied over the framework-dependent deployment of the web app. By the way, you can see that views are now precompiled in a DLL by default in ASP. NET Core 2. There are a couple of easy steps involved to install. NET Core on Ubuntu. You can find instructions for that on this URL. On that same site are also instructions for many other operating systems. It won't surprise you that the application will just run by executing. NET with the DLL name again. When I let the browser hit Kestrel's URL, you can see it works. Using a self-contained deployment is even easier. Just copy it over and run, no need to preinstall. NET Core. In the next clip, we'll take a look at a production-ready web server for Linux called NGNIX.

Running with Nginx

It's great that a Linux app runs self-hosted now, but just like a Windows, I want a full-featured web server to configure SSL, domains, and load balancing, though it would be great if I could set up a reverse proxy in a production-ready web server, just like I did in IIS with the ASP. NET Core module. Linux has a web server available called NGINX, so I installed NGINX on this Ubuntu Linux machine. Just like many applications on Linux, NGINX is configured using text files. I configured IIS earlier with a web. config, but NGNIX doesn't know about the web. config. I've opened up a new console window, so the demo app is still running using Kestrel. In the configuration folder for the enabled sites of NGINX, I have the configuration for the default site. In this case, I want the default site to be the demo app. You can see it listens on port 80, the default HTTP port. The location is slash, which is the root. I'm setting up a reverse proxy to localhost:5000. This means incoming requests to port 80 will now be sent to port 5000, and the responses that a web app will generate will go out on port 80 again. I save the configuration and restart NGNIX. When I open a browser and hit localhost using the default port 80, you can see it works perfectly. There are some more things to take care of in production, because right now the reverse proxy works, but the process my app runs isn't monitored. You have to set that up too. To study the details, go to tinyurl. com/aspnetcorenginx.

Summary

In this final module, you learned the options you have when it's time to deploy your application, namely framework-dependent or self-contained. You saw how to self-host the app with Kestrel, and you can host your app on a production-ready web server by using a reverse proxy in IIS with AspNetCoreModule. Deploying to Azure is a breeze with the Visual Studio tooling. Running ASP. NET Core applications on Linux requires a couple of more steps, but it runs beautifully, even with the full-blown web server NGINX in front of it. Thank you for watching this course. I hope you'll have a blast developing applications with ASP. NET Core 2.