Course Overview

Course Overview

Hi, everyone. My name is Gill Cleeren and welcome to my course, Building Your First ASP.NET Core 2.0 MVC Application with Visual Studio 2017. I'm a solution architect for mobile and web projects. Late 2016, Microsoft launched ASP.NET MVC as the future of ASP.NET. About a year later, ASP.NET Core 2.0 MVC, so the next version, has been released. Microsoft has extended the protocol already quite a lot and they will keep adding more and more features to .NET Core. This course is aimed at giving you a hands-on way to learn about ASP.NET Core 2.0 MVC. You'll build together with me a full application in Visual Studio 2017. You'll see how to apply the MVC pattern, build views, work with a real database, and much more. Some of the major topics that we will cover include building a fully working application starting from File New Project. Learn about the new features in ASP.NET Core 2.0 MVC such as stack helpers, access data in a database, using Entity Framework Core 2, include ASP.NET identity to allow users to authenticate to your site. By the end of this course you'll know how you can build a fully working, real-world application using ASP.NET Core 2.0 MVC in Visual Studio 2017. Before beginning this course you should be familiar with some C# and some HTML. I hope you will join me on this journey to learn ASP.NET Core 2.0 MVC with the Building Your First ASP.NET Core 2.0 MVC Application with Visual Studio 2017 course here on Pluralsight.

Introduction

Module Introduction

Hi there, and welcome to this Pluralsight course named, Building Your First ASP.NET Core 2.0 MVC Application with Visual Studio 2017. My name is Gill Cleeren and I'll be your trainer for this course. The title of this course is already giving away what I'll be teaching you in this course. We'll be spending quite a lot of time writing code in Visual Studio 2017, creating an ASP.NET Core 2.0 MVC application and the nice thing is that you'll be able to follow along with everything that I'm doing in this course, but more on that later in this module. At the end of this course, you'll have understood how ASP.NET Core 2.0 works, how MVC works, and how Visual Studio 2017 can be used to create a real application that you can use as the basis for other applications that you'll need to build in your professional life. I'll start with an overview of what exactly you'll be learning in this course and what to expect in terms of gained knowledge after completing the course. I'll be spending some time as well to explain to you already what ASP.NET Core is all about and why it's such a big deal for Microsoft as well as for you, a .NET developer. Finally, we'll be finishing this module by taking a look at what you'll need to have installed on your machine to follow along with this course. Let's get started.

Course Overview

As promised, I want to give you a quick overview of what to expect from this course so that you know what you'll be spending your next couple of hours with. As mentioned already, this course will teach you how can build a small application with ASP.NET Core to MVC by using the latest version of Visual Studio 2017 with all updates installed at the time of the creation of this course, which is late 2017. For this course, I want to focus on the doing rather than the talking. That's why we will start with File New Project. I'll be spending quite a lot of time in the actual coding of the application so that by the end of the course we'll have a working a ASP.NET Core 2.0 MVC application. Since I am convinced that you will learn more by doing instead of just listening and watching me do all the hard work, I suggest that you follow along with everything that I'm doing in this course. Now to do that, you'll need to make sure that you download the course files for this course. Please go to the course page on the Pluralsight website and download the exercise files. In there you'll find for each module a start and an end solution. Although this is a beginners course, I can't really explain everything so from my side also I do assume some knowledge from your side as well. Since we're developing a website, I assume you have some basic knowledge of HTML and CSS. We'll be writing our code in C# so I expect you have at least some basic knowledge of that as well and in general some affinity with web development will come in handy as well. Note that you don't have to be an expert C# or web developer to follow; far from it, just some basic knowledge in these areas will do. Note that you also don't need any previous knowledge about ASP.NET MVC or ASP.NET MVC Core to be able to follow along with this course. Even if you've never heard of the MVC pattern, don't fear, as I'll make sure that you'll understand everything to follow along. Of course, if you have ASP.NET MVC knowledge already, some things will be a bit easier for you to follow. However, quite a few things have changed in ASP.NET Core and in the case that you are already familiar with ASP.NET MVC, you can use this course to get an understanding of what has changed with the introduction of ASP.NET Core and ASP.NET Core MVC. Now we will as mentioned be building a small application. Let's talk a bit about the application we'll be working with. We are creating an application for Bethany, one of our best customers and Bethany has a small, but very successful shop that sells pies. Since she wants to make her shop even more famous, she has decided her online presence needs to be expanded and so she'll need a web application. In this course we are building an application for Bethany that will contain a home page with some of the delicious pies that she offers. We can also from the site take a look at the details of a pie. The site also will contain a feedback form and we'll see how we can make sure that only logged-in users can access the feedback form.

Demo: Taking a Look at the Final Application

As the first demo of this course, let's take a look at the finished application that we'll have created by the end of this course. We're now looking at the final application that we'll be building throughout this course for the application for Bethany's pie shop. The application starts by showing a list of some delicious pies that Bethany is offering on her site. We can click on one of the pies and then we see the details of that particular pie. It looks very delicious to me. We can also click on the Feedback button here to send some feedback to Bethany. As you can see, before sending feedback, I need to either register or log in. I have already created an account beforehand so I can just simply log in. I've entered the credentials, I can now hit the Log in button and as you can see, I'm now logged in. There's a Log off button showing. I can now also click again on Feedback and I can now send feedback to Bethany. I can enter my name, my email, and also the message I want to send and I click on Send Feedback, the information is stored, and you'll see a thank you page. That's the application that we'll be building throughout this course together. I do want to point out the following before continuing. A longer version of this course already exists in the Pluralsight library where I add quite some more features to the site than I add in this course; however, that course was created with ASP.NET Core 1.0 and quite a few things have changed between ASP.NET Core 1.0 and 2.0. That is why this course on ASP.NET Core 2.0 has now been created. You can see how the setup is done using ASP.NET Core 2.0 and all other features that we aren't covering in this course still work exactly the same in version 2.0. So for those features, please take a look at the longer version of this course, which you can find using the link on the slide you see here.

What Is ASP.NET Core?

Now that you have an idea of what to expect from this course, let's explore what this ASP.NET Core thing is all about then. Now let's start this exploration with a short history lesson. No fear! Not one like the one you hated in school. Anyway, things started for ASP quite a long time ago. ASP, which stands for Active Server Pages, was already created in the late 1990s. At that time there was no .NET just yet. So we had to use script code in combination with HTML to create interactivity. Yours truly has actually done his first app development endeavors using this very technology. Microsoft has then launched .NET and with the launch of .NET they also created ASP.NET and ASP.NET WebForms. WebForms was mainly aimed at making web development as easy as Windows desktop development. We were able to use real C# code in code behind files in combination with markup HTML that you can drag onto the designer interface. At the time that was a great technology, but the created HTML was lacking seriously. Developers wanted to get more control over the markup they created and Microsoft gave developers a new tool and that tool was called ASP.NET MVC. Aimed at being able to create nice HTML again in combination with a more test-oriented approach, ASP.NET MVC allowed us for years to create maintainable code bases for web applications. ASP.NET has received a lot of updates over the years, but was still based on the same foundation as WebForms. Now because of this dependency, it still pretty heavy to run and was still very monolithic and it was also tied to Windows. Microsoft therefore started the .NET Core stack aimed at cross-platform and lightweight development. In 2016 after many beta releases the first release of ASP.NET Core was released and in August 2017, Microsoft has released the first major update to ASP.NET Core, ASP.NET Core 2.0. ASP.NET Core is built on top of .NET Core and therefore inherits its benefits. As already mentioned, ASP.NET Core is capable of running cross-platform and it's thus not tied to Windows only. It's therefore possible to run your ASP.NET Core code base on Windows, Linux, and Mac. ASP.NET Core is basically a rewrite, a different branch so to say. It's not based on the code base of previous versions of ASP.NET MVC, which also explains why some features are not available in ASP.NET Core. ASP.NET Core is a platform that we'll be using to build our application. Now the name is coming from the commonly used pattern, the MVC pattern, which stands for Model View Controller. The pattern has been around for quite a number of years and is basically an architectural pattern that promotes the separation of concerns. Instead of writing our code in one big, un-testable block, we will separate the concerns so that code can more easily be tested in isolation. Now by using this approach we will end up with more testable and maintainable code. Don't worry about fully understanding this pattern just yet; I'll be explaining the different blocks in more detail later in this course.

Getting Your Machine Ready

Now to finish this introduction module I want to make sure that you have everything installed to follow along with this course. As you may have understood from the title of this course, I'm using Visual Studio 2017 for this course to build the application. The tooling that I'm using is the latest version available at the time of the creation of this course, which is late 2017 and the latest update for Visual Studio 2017 is update 4, which brings some minor changes in the tooling. Of course, you'll need a browser as well. If you're following this course from a machine running an operating system other than Windows, well things will still work for you. How great is that? Remember that I said that ASP.NET Core runs cross-platform? Well, this is one of the places where you'll benefit from this. If you want, you can use Visual Studio Code as well for anything that I'm doing in this course on a Linux machine or on a Mac and of course, also on Windows. Other OmniSharp-ready editors including Atom, Sublime, and even Vim can also be used, but as mentioned in the demos of this course, I'll be using Visual Studio 2017.

Demo: Getting Your Machine Ready

In this demo, let me guide you quickly through everything that you need to have installed to be able to follow along with everything I'm doing in this course. So as mentioned, in this course I will be using Visual Studio 2017 to create the application. If you don't have Visual Studio installed, you can go to visualstudio.com and from there you can download the Community edition for free, which contains everything that you need to do ASP.NET Core development. The Visual Studio installer has changed dramatically compared to previous versions. You'll want to make sure that you have the .NET Core cross-platform development workload installed as you can see here. Visual Studio now installs everything based on these workloads and there is a workload specifically for .NET Core cross-platform development. Here on the right you can see that indeed the Visual Studio setup will also install .NET Core 2.0. If on the other hand you already have Visual Studio installed, you can go to microsoft.com/net, click on Downloads, and there you'll find the .NET Core SDK. As you can see at the time of the recording, the latest version is 2.0.3. These are the tools that you need to follow along in this course. If you want to make sure that everything installed correctly, open Visual Studio and click on File New Project. In there, click on .NET Core, select ASP.NET Core Web Application, hit OK, and in the template window, make sure that ASP.NET Core 2.0 is available in this drop-down list. When that is the case, you're ready to go.

Summary

And so we have arrived at the end of this first module. Although we haven't covered that much just yet, I want to point out the most important takeaways from this module. Microsoft is making a lot of investments in .NET Core and ASP.NET Core. It's clear that the future of .NET is in .NET Core and of course, by learning about ASP.NET Core and ASP.NET Core MVC, you're in a perfect position to benefit from this. With ASP.NET Core, Microsoft is also making it possible to build and run ASP.NET MVC applications in a cross-platform way, which is single-handedly the biggest change we've ever seen in the world of ASP.NET. Let's start with the creation of the app we've looked at in this module. We'll learn about the configuration and the setup of the site. I hope to see you again in the next module. Thanks for watching!

Setting up Your Site with Visual Studio 2017

Module Introduction

Hi there and welcome to this module titled, Setting Up Your Site with Visual Studio 2017 here on Pluralsight. My name is Gill Cleeren and I'll be guiding you through this module. So far, we haven't really started with the creation of our site just yet, but that will change in this module. By the end of this module we will already have configured the application correctly. Quite a few things are very particular in the area of the configuration of the site so let's dive in and learn more about this. A small overview of what we will be doing in this module is in place. We will start with File New Project and we will take a look at what gets created by executing the template. You will learn more about the different files that get created when you execute the template. After that we will explore the configuration of the site, which is as mentioned, something entirely new in ASP.NET Core.

Exploring the Project Structure

As promised, let's start with exploring the project structure of a new ASP.NET Core 2.0 application. We'll take a look at the new files that are being created by Visual Studio and see what their use is. Creating a new ASP.NET Core 2.0 MVC project can be done using Visual Studio 2017. As you can see in the screenshot here with .NET Core installed, I have the ability to select ASP.NET Core Web Application under the web node. When we hit OK, Visual Studio will give us the option to select the template we want to execute. Different templates exist depending on the .NET Core version you have selected at the top of the window. We're using here ASP.NET Core 2.0 of course and then you should see the list of possible templates you can see here in the screenshot. Since I want to make sure that you understand all that is going on, we will start from the blank slate. I will select the Empty template in the demos. Other possible templates that come with Visual Studio 2017 include the template to generate an API, a more complete MVC application, and some templates that already come with React or Angular enabled by default, but as mentioned, we will start from the Empty template. When we let Visual Studio execute this template, we'll get the project structure we see here in the screenshot. As you can see, not a lot has been generated for us, which was to be expected since it was the blank template that we selected.

Demo: Creating a New Application

Instead of discussing slides what happened when Visual Studio executed the template, let's go to Visual Studio, do a File New Project, and take a look at what got created. I'm here in Visual Studio and we are going to start with the creation of the application. So we'll go to File New Project and we'll make sure that we have .NET Core selected and then we'll be able to select ASP.NET Core Web Application. I'm going to name my application BethanysPieShop. Hit OK and now Visual Studio presents us with the list of templates it can find for an ASP.NET Core web application. You can install additional ones, but the list you see here is the list which is currently installed with the installation of Visual Studio. We have the Empty template and that's the one I'll be starting with because then I can really explain you everything that is going on. As a template specifically for a web API, there are two templates, as you can see which target a web application. The one on the right, so the one with Model View Controller, uses MVC, whereas the one on the left uses the new framework in ASP.NET Core 2.0 which is the Pages framework and it's something that I won't be using in this course. There's also a template which already includes Angular, React, and React.js in combination with Redux, but as mentioned I'll start from the Empty template. So I'll hit OK once again and here we now see the result of executing that template. Let's take a look at the Solution Explorer. Under Dependencies we will quite logically find some dependencies. Let's expand the NuGet node and there we'll see a package Microsoft.AspNetCore.All. A little side note here is that a lot of things, in fact, almost everything in .NET Core and therefore also in ASP.NET Core is now a NuGet package so you'll see a lot of packages typically referenced in the .NET Core application. As you can see here, there's in fact only one which is referenced and it's called the MicrosoftAspNetCore.All package. As the name is already given away, it adds in All, it probably means that it has some extra dependencies itself. That's correct. The AspNetCore.All package is what is called a metapackage. It's a package that in itself references other packages. If you take a look here, well let me expand this a little bit, you'll see that many other packages are referenced here. The direct result of this is that we as developers will not have to create a lot of extra package references since a lot of packages have already been referenced. Now since we're talking about dependencies, I do want to show you this as well. When I right-click on the Project file and I and edit the project file so the csproj file, I can indeed see here that there is a package reference to Microsoft.AspNetCore.All. Now since we have the csproj open, do note how clean and simple this file has now become. It's now easy to manually edit this file yourself. That's probably also why there is now an entry in the right-click menu to directly open and edit the csproj file. Another thing of interest is this wwwroot folder here. This folder is called the web root. It is the folder where we will be storing static files. So JavaScript files, image files, CSS files and so on. That means that all of our MVC code will actually go in the root BethanysPieShop folder where its static files will typically be in a structure under this wwwroot folder. So this means that under the wwwroot folder, I can, for example, create an images folder and I can put in there an image called image1.jpg. That image can now be accessed via www.bethanyspieshop.com/images/image1.jpg. We will be adding static files in this folder very soon. Finally there are two very important classes, the program and the startup class that we'll need to investigate a little bit more. Let's first return to the slides for a minute.

Site Configuration

Now that we have seen the different files that were created, let's start with the actual configuration of the site. In the previous demo, you have seen that at the root of an ASP.NET Core application we had the two very important files, namely, the program.cs and the startup.cs. As mentioned in the demo, these two classes have a very distinct task in the lifecycle of an ASP.NET Core application. The program class in our application contains a main function, as you can see here. This may sound a bit weird, but yes, an ASP.NET Core application is in fact a console application and therefore it contains a main function that will execute. Although for our application we won't really have to change anything in this class, it's interesting to know what's happening here. In the main function, a host is being set up, which will configure a server and a request processing pipeline. We are calling CreateDefaultBuilder which will set up the application with some defaults. One of the things that it'll do is configuring a web server called Kestrel. By default you ASP.NET Core 2.0 application is thus hosted by this internal web server. Typically you will still access a site over IIS and that link is also enabled by default here. Another very important thing that is happening here is that we are specifying the startup class using the UseStartup method. We are passing as star parameter the startup class, which will be the type that actually performs the startup or I should better say, the configuration of the application. In the startup class we will actually be doing two things. Defining the request pipeline, and configuring all services that we need throughout the application. The startup class is required to define two methods, which will be automatically invoked. These are the ConfigureServices and the configure methods. In ConfigureServices we will be registering services for our application. Now let me explain this in a little bit more detail, since it is crucial to understand. The ASP.NET Core uses a simple built-in dependency injection system. Through dependency injection we can achieve a loosely coupled architecture within our application. Instead of instantiating classes directly, dependencies will be injected into the consuming classes. We'll see how exactly this is done later in the course, but most of the time this will be passing dependencies through the constructor. This approach is then called constructor injection. ASP.NET Core comes thus with a built-in dependency injection container, a component that will be helping us to register and retrieve dependencies. This container is available through the IService provider interface and the services collection you see here as a parameter is the collection of services it manages. We can in this ConfigureServices method therefore add services to the collection of services managed in the application. Now services is a very broad term and it is in fact exactly that. Services are objects that have a certain functionality for other parts within the application. We'll be able to register system services as well as our own, which we will be doing later in this course. In the Configure method, the request pipeline will be set up and the request pipeline consists out of a number of components chained behind one another. These are the so-called middleware components. These components will, let's say, intercept or handle an incoming HTTP request. Each component will be able to alter the request or by the way, also the response can be altered, or simply pass it on to the next component in the request pipeline. Here in the sample kit we are registering four middleware components. Along the way through this course we will be registering more middleware components. First, I'm using UseDeveloperExceptionPage. Enabling this will therefore enable a page that is comparable with the yellow screen of death page that we had in previous versions of ASP.NET. The UseStatusCodePages method add support for text-only headers for common status codes. UseStaticFiles is another middleware component I'm registering here and that will enable the ability for my site to serve static files such as images and CSS files. Finally, UseMvcWithDefaultRoute sets up MVC middleware using the default routing schema. This may sound a bit confusing, but don't fear, we will explain all of this in much more detail throughout this course. Since these components will be in a pipeline, the order in which they get added is important here. Typically your call to the MVC middleware component needs to be at the end of the pipeline. So to summarize, when an ASP.NET Core 2.0 application is starting, quite a few things are set in motion. The application starts with the call to the main function in the program class. There we specified which was our startup class using the UseStartupCall. In the Startup class, first the ConfigureServices method is called automatically by the system and it allows us to register services that we'll want to use in our application. Then the Configure method is called where the request pipeline is set up. After that, our application is up and running and it's able to handle incoming requests.

Demo: Site Configuration

Now that we know how an ASP.NET Core application starts, let's return to the demo and configure our site. So here's a program class, which is of course in the program.cs file. An ASP.NET Core application is now in fact a console application. So that means that an ASP.NET Core application now has a static void Main. It may still sound a little bit weird; I totally understand. In the static void Main there's a call to BuildWebHost which will basically set up the environment. In the BuildWebHost we are calling CreateDefaultBuilder on the web host class. The later does quite a few things. Let's take a look at the documentation for this method. As mentioned, the CreateDefaultBuilder will do some setup for the application based on some defaults. One of the things it will do is it will create Kestrel, a built-in web server that actually runs Kestrel, a built-in web server that actually runs your ASP.NET Core application. Although your ASP.NET Core application runs on an internal web server, it can still be used in combination with IIS using IIS integration. That's another thing that CreateDefaultBuilder will do. It will set up this integration. There are quite some other things that CreateDefaultBuilder will do for you and we'll come across some of the other defaults that will be set up by this method later in this course. Next is a very important call to UseStartup by default in here and it's pointing us to the startup type. This will basically specify the Startup class as the startup type for our application. Let's take a look at the startup next. Here you see the default created Startup class. As mentioned, it contains two very important methods, ConfigureServices and Configure. Both methods are automatically called by ASP.NET Core. In the ConfigureServices method we will add services to the dependency injection container. Through the passed-in IServiceCollection I'm capable of registering services in my ASP.NET Core application. ASP.NET Core is very lightweight and therefore also modular and that's a big difference between ASP.NET Core and previous versions of the framework. Since everything is now modular, basically nothing is enabled by default so we will have to enable what we need and we can do that for many things by registering the service in the dependency injection container and basically enabling the feature. In this ConfigureServices method I'll therefore be registering services of the framework that I'll need as well as my own services and I will do that in the next module. For now let me start by making sure that MVC is enabled for this application; MVC itself needs to be added to the services collection. I can do that by calling services.addmvc. Now basically my application knows about MVC. Next we have the Configure method. In this method we will set up the request pipeline of our application. We should really see this as a pipeline in which we have a sequence of components that will work on the incoming request. They will be able to inspect the request, change it, or pass it on to the next component in the pipeline. It's also possible for components to actually short-circuit the request. If a component can handle a request, it may do so and it will not pass it on to the next component, but it will short-circuit, so basically send it back to the client. I'm going to start by deleting the default generated code here so I can add several middleware components I need in my request pipeline. I'll start by adding support for the DeveloperExceptionPage. By default in an ASP.NET Core application you will not screen the yellow screen of death that you used to see in old ASP.NET applications. By default we will not get any feedback when something goes wrong. By plugging into the request pipeline, this extra middleware component it will actually make sure that when something goes wrong in the application, we will get an exception. This is of course something that you should only use while we're developing the application. Next I'm also going to add a middleware component that is going to show information about the status of the request, for example, a status 400 or status 500, if that would be returned by my application. The next middleware component I'm going to plug in is support for static files. The middleware component for static files will do exactly what it says. It will go and search by default in that wwwroot folder and return a static file from there. That can be an image, an JavaScript file, a CSS file, and so on. Finally, we'll also plug in the MVC middleware component. I'm using for now, UseMvcWithDefaultRoute. I will talk about what that routing is there later in the course. There's one more important thing here. I did mention that this pipeline should be considered a sequence of components working on the request to route a response. Therefore the sequence in which you add them is important. UseMvcWithDefaultRoute should definitely be placed after UseStaticFiles. UseStaticFiles is typically one that will short-circuit the request and therefore basically not bother MVC for requests for images for example. So that will bring down the load on the server because those requests can be handled by the static files middleware. So keep that in mind. The sequence in which you add the middleware components is important here. So at this point, the basic configuration for my application is ready.

Summary

You now already have a good understanding of how ASP.NET Core 2.0 MVC applications are typically structured and if you have knowledge of previous versions of MVC, you'll have noticed that things have changed quite a lot. In this module you've also heard for the very first time that ASP.NET Core 2.0 comes with dependency injection support built in. In the oh so important startup class, you've seen how in the ConfigureServices method we have the ability to register services with the dependency injection container. With the site now ready and configured, it's time to start writing the code for the first page end to end. That's exactly what we will be doing in the next module. Thanks for watching!

Creating the List Page

Module Introduction

Hi there, and welcome to another module of the Building Your First ASP.NET Core 2.0 MVC Application course here on Pluralsight. I'm still Gill Cleeren, so that has not changed since the last module. In this module we will create the first list page for our application and for that we are of course going to follow the principles of the MVC pattern. So we'll be creating the different building blocks throughout this module. As always, let me give you an outline of what we will be looking at in this module. We are using ASP.NET Core 2.0 MVC and of course, that fully relies on the MVC pattern. So for that very reason, I want to make sure that you understand the concepts of MVC and we'll kick off the module with a short overview of the building blocks in an MVC approach. In the next part, I will be taking you the creation of the page that we are going to build in this module and we'll see how each building block is to be created using ASP.NET Core 2.0 MVC. We will start from the ground up and so we'll first create the model and repository clauses. Next, quite important in the whole MVC story is of course, the controller and it will look at how we can create a controller for the list page here, and we only have the controller, we won't really see anything just yet so we will be learning how we can add the view in the next part and we'll learn about Razor and other view-related topics here as well. Finally, we'll want to make our view look a bit nicer. To do that, I'll show you how you can add in your project some Bootstrap code. After completing the steps in this module, we will already have a fully working MVC page up and running.

Hello MVC

As promised, we'll start the module by making sure that you understand the MVC pattern and its building blocks correctly so let's do that first. The letters MVC stand for Model View Controller. The MVC pattern is an architectural pattern that has been around for quite some years. It's not something you need for ASP.NET, far from it even. It has been created with the idea in mind that code or classes should be doing one thing and one thing only. In other words, to improve the separation of concerns. This results in more loosely coupled systems and the net result is that we get software that is much more testable and maintainable. In MVC we are thus working with tree building blocks. The controller, the view, and the model. When we are sending a request to an MVC application, the request will arrive on the controller, which we'll see soon is nothing more than a class in our code. It has a very important role in that it will update the model or retrieve data from the model. Secondly, it will also select which view is needed. The view as we'll see is nothing more than a template which will receive the model data and display that to the user.

Creating the Model and the Repository

Now that you have a basic idea of the MVC pattern and its building blocks, let's start making things more practical. In the remainder of this module we will be creating the different building blocks and we'll start with the one at the very foundation and that is the model. Let's take a look at the model and the repository. Now we have been referring to the model already quite a few times, but what exactly is that model then? We refer to the model as the group of classes that make up the domain data. So typically these are plain old CLR classes or POCO classes. Next to the domain data, the model also contains the classes that manage the data. The later wouldn't typically be a repository class, but more on that in just a second. The model should typically have a simple API and its main goal is hiding the nitty details of how we manage or interact with the actual data from the consumer. We don't want to spread the code that does the actual data interaction throughout our code base. We want to group that code inside of the model. Like I said, part of the model is of course the domain classes. In our code we'll need to represent pie instances in our application. The model class Pie is a class part of the domain of our application. You can see the Pie class and we will be using that one in the demos in just a second. The class itself has no base class; it is a simple POCO class. Just the domain classes alone do not make up the model. We'll also need the classes that will perform the actual data persistence. This will be handled by creating a repository class, which is a class that abstracts away the details of how the persistence is actually happening. If really translated, as mentioned before, we don't want to litter our code of the application with code that persists objects. All that code should be placed inside of a repository class, which can then be used from our application. I said that the model should expose a simple API towards the rest of the application. Well, that's what you can see here. We're looking here at the contract of the repository called IPie repository. This interface defines what our repository will do initially. It defines a method called GetAllPies, which is returning an IEnumerable link Pie containing probably all the pies in the data store. Next it also defines a method called GetPieById, which accepts the id of the pie we'll want to retrieve. Pretty simple, right? In the upcoming demo we will be creating this interface and we'll create a mock implementation of the interface for now, which will contain some hard-coded data. One thing before we go to the demo I still need to tell you is about the registration of the dependency. In the previous module we have already seen that in the ConfigureServices method of the startup class we can register system services. That worked because of the dependency injection system that comes with MVC and the good news is that we can use this system also for our own services. Instead of creating an instance of the repository in our code somewhere and thus creating a tight coupling, what we'll do is we'll register our repository with a dependency injection system. All other classes that will want to make use of our repository can then ask the dependency injection system for an instance of the repository. That's exactly what dependency injection is all about and this way we'll create more loosely coupled code and remember, that was one of the goals of MVC.

Demo: Creating the Domain and Repository

Okay, time for a demo. In the first demo of this module we'll create the model. So I'll be creating the domain classes and the repository. So let us now start with the creation of the model. I'm going to put all the stuff that's related to the model in a folder called Models. I think that's a good name. I'll typically put in there the classes that make up the domain of my application. In my case, that at this point would be the Pie class. My Pie class needs a number of properties. I'm going to paste these in. You can find all the things I'm pasting in during this and other demos in the snippets file in the downloads of the course. There's a separate snippets file per module. At this point my pie has an id, a name, a short description, a long description, a price, a url for the image, and a thumbnail, and a Boolean that returns whether or not this pie is currently the pie of the week. Now of course my model is more than just my domain. I also need the classes to work with that domain. They also are part of the model. And I'm going to create first an interface, the iPieRepository. Change it into an interface and currently I want it to support that this will return all pies in a method called GetAllPies and I also want to be able to retrieve pies per id so I create a GetPieById method as well to which I pass the id of the pie, and that will return of course a pie. At this point I'm going to implement this interface in a class called the MockPieRepository, which is also still part of the model of my application. I call the MockPieRepository because I'm going to work with some mock data for now. Of course this class needs to implement the iPieRepository and we'll have Visual Studio implement the interface. There we go. I'm going to paste in a little bit more code. Let me show you what I've pasted in. I have a private list of pies, which is going to be initialized in the constructor if it is still null. In this private void InitializePies method I'm initializing the list again with some hard-coded data. If you're following along, take a look at the snippets file for this module. The GetAllPies method is simply going to return that pies list. In the GetPieById method, I'm going to return from the pies list the first pie for which the id is equal to the passed-in id. So that'll be p.Id equal to pieId. If I can't find it, I will simply return the default. So make sure that you've saved all of this. Now this MockPieRepository, I'll need to use it from different locations within my application so I'll need to register it with a dependency injection container in my application and for that we have to go back to the startup class. Remember that in the ConfigureServices method I have access to the dependency injection container of ASP.NET Core and I can register extra services in here. So what I'm going to do is I'm going to call services again and I'm going to call AddTransient here. AddTransient means that whenever someone is asking for an iPieRepository, a new MockPieRepository will be returned. AddTransient really means in terms of lifetime that whenever an instance of iPieRepository is requested, we get a new instance every time. Next to AddTransient we also have AddSingleton. AddSingleton basically means that only one single instance is going to be created of the specified type and the same instance is always going to be returned. There's also AddScoped. AddScoped is a bit of an in-between. Per request it will always return the same instance, but as soon as the request goes out of the scope, the instance is removed and with the next request, a new instance is going to be returned. So we're now ready with the model. Let's now go back to the slides.

Creating the Controller

Now that the domain layer is already shaping up nicely, let's turn our attention to the controller. In the entire MVC story, the controller has a very central and crucial role. You could see the controller as the manager within the MVC pattern. It will use the view and the model to in the end create a response that we'll want to send back to the client. The controller will thus be responsible for responding to user interaction. When the user has for example clicked on a button and a request is sent to the server, that request will be picked up by the controller. It's therefore the controller, which has the end responsibility to create the response based on the user interaction. In doing so, the controller is the party that will directly interact with the model and by using the model, the controller doesn't need to know how the actual data persistence works. As you have already seen, it's the role of the model to hide these details from the controller. Let's see what the controller looks like then. Here on the snippet on the slide, we can see a very simple controller called PieController. As mentioned, a controller is nothing more than a class and it will inherit from the base controller class that comes with MVC. In MVC the controller's name is typically suffixed with controller as well. That's a convention in the MVC framework. I've said before that it is the controller's role to respond to an incoming request. In fact, to be correct, it's actually a method that will be invoked on the controller when a request is received. Such a method is called an action or an action method. This method will execute and typically in the case of MVC, it will return at the end a view as a result. Therefore the return type is also ViewResult. This will then make sure that the viewer will get to see your web page as the result for sending the request. Of course, a real action method on a controller will typically do more than just returning the view. Here on the slide you can see a more real-life example of the PieController. For starters, the controller is now making use of the pieRepository, which as you can see, it's receiving through a constructor parameter. In fact, it's the built-in ASP.NET Core dependency injection system that is taking care of this. An instance will be injected into the controller automatically. You'll see this in the upcoming demo in more detail. In the list action method we are still returning a view, but this view is now receiving some data, in this case the list of pies exposed on the pieRepository. This data will then be used by the view. Remember that the view, as we'll see soon in more detail, is in fact a template. It'll receive the model data that it will need to display correctly.

Demo: Adding a Controller

Let's now return to our demo and add the controller. Just like a place, everything that has to do with the model in the Models folder, I'll also be creating a folder for my controllers. So I'll do that first. I'll create a folder called Controllers. In there, we'll start with the creation of our first controller. I'm going to go to Add and Add New Item here, select Web and then select MVC Controller Class. We'll leave the name HomeController; that's a good name to start, click on Add, and here's our first controller. As you can see, a controller is a class that inherits from the base controller class and the convention also says that the controller should have a name that ends in controller, but that is just like I said, a convention. Now the controller is of course going to be using the repositories to talk with the model. So the first thing I'm going to do is I'm going to bring in a private field for the pieRepository. I'm going to create a new read-only iPieRepository and I'm going to call that the pieRepository. In the constructor I'm going to initialize this pieRepository. Now without the dependency injection what you would typically do is say my pieRepository is a new MockPieRepository. Now because of dependency injection, we'll not be doing this. Instead what I'll do is I'll say that this HomeController will need an IPieRepository when it's going to be created. Now because of the fact that in my startup class I had specified that when a IPieRepository was required, a MockPieRepository was going to be returned. Dependency injection will automatically inject an instance of the MockPieRepository here. That is called constructor injection. I simply specify that this controller needs an IPieRepository and a dependency injection container will try to fulfill this dependency. So I'm going to say that my local pieRepository is equal to the passed-in pieRepository. You can already see that an index action method has been generated, but we first need to see a couple more things about the view before we can continue writing the code in this controller. Now let's go back to the slides for just a minute.

Adding the View

We now have the M for model and the C for controller. That means that the only thing we're missing is at this point the V for view. Of course, that's a very important one. We can write awesome code in the controller, but as long as there's nothing to visualize, the user won't be able to use any of our code. So let's take a look at the views in this part then. So what exactly is the view part then? I've already mentioned that the view should in fact be considered a template. It contains mockup code that we'll use to visualize the data of the model. If the data, for example, would be a list of pies, well in that case, our view could contain code that will loop over all the pies in the list and visualize each of them. For example, as rows in a table. In ASP.NET Core 2.0 MVC, view templates are available as .cshtml files. These are files that contain the mockup code. In the context of ASP.NET Core 2.0 MVC we can distinguish two types of views, a plain or regular view and a strongly typed view. You'll see what these mean in the upcoming slides and you'll see in a demo that most of the time in the case of ASP.NET Core 2.0 MVC they'll be using strongly typed views. In the views, we'll be making use of Razor, a mockup syntax that allows us to use some C# functionality in the web pages we are creating. Our views are as mentioned, .cshtml files. By convention, views are placed in the Views folder, which is a folder under the root of the application. In there for each controller that we create in the Controllers folder, a corresponding sub folder is created. So if you have a home controller we'll also a folder Home under the Views folder. In that folder then, each view is represented by a .cshtml file. Note that this is purely convention and if you really don't like this, you can change it; however, I won't be doing that in this course. So for each controller we need to have a sub folder where the .cshtml files will live, but how does the controller then know which view needs to get rendered? Now consider the example controller again here we have with the Index action method and we are returning a View. Again, by convention the view that will be shown here is called Index.cshtml so the same name as the action method. If this view cannot be found, an exception will be thrown. How can now the controller make sure that the correct data gets passed to the view? Remember that we have said earlier that the view will receive data to visualize. It is basically a template. Well, there are several ways to pass data. Probably the most basic one is using something called the ViewBag. The ViewBag is a dynamic object that we can add data to from within the controller. As you can see here, I'm adding data to the message property and since the ViewBag is dynamic, I can add as many properties with the name I want. Our view can now also access this data and thus become a dynamic view. Here we can see a very basic view, which is almost pure HTML, but notice the highlighted line here. The line is prefixed with the @ symbol and that is a signal that this is Razor syntax. It's a sign to say that this line actually contains code that needs to be evaluated. You'll see me using quite some Razor code in the demos. Now after the @ symbol we see again ViewBag.message and so what happens here is that my view code now gets access to the same ViewBag object as the controller and can read out the properties that were specified in the controller code. This way the controller and the view can now exchange data. One major problem is the actual dynamic aspect here. And real applications will need to cost quite a lot since all the past data is of the base object type. Because of the fact that ViewBag is dynamic, we won't use it all that often and instead, most views that we'll create will be so-called strongly typed views. In the code that you see here, I have an action method again; the list method, and now it is not just returning a view. No! Instead it's returning view, passing in a list of pies using the repository as a parameter. This results in the list now being set as the view data.model property of the ViewResult that's being returned here behind the scenes. This view will now get access to the list as well. Let's take a look at the view code now. To create a strongly typed view, at the top we have to include the type of the object that's going to be passed in. We're doing this using the Razor model keyword followed by the type which is here, an IEnumerable in Pies. With the typing in place we can now access the model. So the data that is passed to the view. What I'm doing here is again I'm using Razor code to loop over all the pies in the model and so that model was the list. I'm using a foreach Razor expression here, which is just a C# expression. Inside the loop I'm creating more HTML elements and again through Razor expressions I can now access the properties of each individual item in the list such as the name, the price, and the category of the pie. If we create all of our views like this, so including all the html for each page, it would be quite a hard job to update our site if we want, for example, to add a new page to a menu at the top of the page. We would have to update all pages manually; how bad is that? To solve this, ASP.NET Core MVC comes with a layout file, which is a template that our views can refer to. It'll contain the general mockup and other views can place their content inside of the layout template then. Such a template will typically be placed in the shared folder. ASP.NET Core 2.0 MVC will look in that very location, which is again a convention. In our application we'll have just one layout file, but know that a real application will typically have more than one. Here you can see an example of a layout template. It contains the full-page mockup code and it will also contain a placeholder, which is again a Razor expression. We are using RenderBody here, which indicates where the view's content should be placed. Now one more thing here. Our views will now need to point to the layout file they want to use. Now we can do that in each view directly, but typically what we'll use is a so-called ViewStart file. This is another cshtml file that will be searched for automatically by ASP.NET Core 2.0 MVC when a view is being rendered. Now in the ViewStart file we typically place code we want to execute when just any view is being called. So at the ViewStart, in there I then specify which layout file should be used.

Demo: Adding the View and Layout

We've covered quite a lot of ground here regarding the view. So now we turn to our application and add a view, a layout, and a ViewStart file. In the previous demo we had already created the home controller, but that's the one that you see here. We already had in there the dependency that was going to be injected so the iPieRepository was being injected into this controller via the dependency injection. Now we are going to focus a bit more on the view part. As you can see here, currently we have an IActionResult Index which is going to return a view. That view of course at this point does not exist just yet so I'll now create a new folder called Views where all my view files are going to reside. So let's add a new folder and the convention is that this folder is called the Views folder. Since we are here in the Home controller, I need to create a sub folder called Home and all the views that have to do with the home controller are going to live in this very folder here. So I have an Index action method where I'm calling return View. ASP.NET Core MVC is going to search for an index.cshtml file in this very folder here. So we can go to Add New Item. Select MVC View Page, set the name to index.cshtml, click on Add, and a new view file is being added. Now I could start adding basically static HTML here and then this view would be invoked via that index method. Of course, we'll want to pass some data from this Index action method to our view. As mentioned, we have two options to do that. I can specify the ViewBag here. The ViewBag is a dynamic so I can add any property I want. So for example, I specify the Title property and set that to Pie overview. In my view let's start with deleting auto-generated code here and I can now access that data as follows. I can now say add ViewBag.title. The @ symbol indicates that this is a line of Razor code. The ViewBag basically wraps a dictionary and all the data that I set in the controller is going to be available in the view as well. As you may have seen when I was typing, there was no IntelliSense, which is kind of logic because it's a dynamic. Visual Studio does not know which properties I have defined on this ViewBag in the home controller. All the data that's being passed from the controller to my view is also going to be passed as object so we don't have any type information. That is why we will typically use a strongly typed view. So I'm going to be passing some data directly here when I call the View method. What I'm going to do is I'm going to say that I want to retrieve all the pies to build up that list of pies and I'm going to use for that the injected instance of the pieRepository and I'm going to call on that getAllPies method and I'm going to say that I want them to be ordered by name. There we go. This list of pies, which is currently an iOrdered enumerable is then going to be passed to the View method. Now in the view I can specify that the model for this view is of type IEnumerable in Pies. Let's make this view valid html. So I'm going to specify a doc type, an html, a head, a body, and as we can see, that ViewBag title is now moved to the body. Now since my view is now typed in an IEnumerable of Pies, I now can write a Razor line that is going to loop over all the pies in the list. I'm going to do that with a foreach expression, which is going to say that for each pie in the Model and notice now this is a capital, so this Model property that is available to the view, basically is of type IEnumerable in Pies, and that's of course the same data that was passed here to the view in my controller. In here I'm going to create a div and for each pie that I'll have, I'll want to show the name of the pie and on a new line in an h3 I'll want to show the price of the pie and I'll want to format that as a currency. If we've done a good job we should actually now be able to run the application and actually see a list of pies. Let's try that. It's a little bit bragging, but I think I've done a good job. I can see both the title Pie overview, as well as a list of the pies. That was the title here, which is still being passed via the ViewBag and then I have the list of pies, which is being rendered by this foreach expression here. Now if I have a lot of views and a real application typically will have a lot of cshtml view files, it's not a good idea to always have to fully include the full html mockup. You'll typically have in your views parts which are going to be reused so we'll need to use a template and in MVC we can add a template by creating a layout file. Since the layout is going to be used by multiple views, we're going to place it in a shared folder. That's again a convention and that's a folder that is going to be served by MVC when it is rendering your views. It's doing that automatically. So we'll add inside of the Views folder another folder and we'll call that Shared. In there, I'll create a new layout file and that will be a template file. Select New Item here and under Web, select MVC View Layout Page. By default it's called \_layout and I will leave that for now. This as you can see, contains the full html skeleton and note that this now also contains a call to @RenderBody. The RenderBody is a placeholder which is going to be replaced with the actual output of the view. So when my view is going to use that layout template, the output of this view is going to be inserted right in this very position here. Now since my template contains this full html skeleton, I can go to my own view and delete all of this because that's now contained in the template. Now of course, at this point my view does not know it needs to use that layout template. I need to point it to that template. I'll use again some Razor code here in which I'm going to specify that the layout property of this view should be set to \_Layout and this is pure C# so I'll need to end that line with a semicolon. When this view is now going to be called by my controller, it's automatically going to search in the shared folder for my template I have defined here. Let's try that to make sure that everything still works. The result is still the same; that's good news. Now having to write this code in each and every view is a bit of an overkill, is it not? So what I'll do is I'll add a so-called view start file. Directly in the Views folder we'll add another New Item and we'll select MVC View Start Page here. This file is by default called \_ViewStart. As the name already implies, the ViewStart file is going to be called automatically when MVC is rendering your view. It's called at the start of the rendering of the view. So when my index view here is going to be invoked, it's going to search for the ViewStart in the Views folder so in its parent folder. And in there we can basically write code that we want to invoke upon rendering of each and every view. In this case we're setting here that the layout should be set to that same layout. In my index I can now remove this code because it's going to be invoked automatically anyhow. The result is of course still the same and I'll prove that to you. I would never dare to lie to you. As you can see, the result is still exactly the same, but basically my view is more or less ready for now, but I'm going to make one more change here. I'm a bit bothered with the fact that in effect I'm using two ways to pass data. Is it not possible to create one type that is going to contain all the data for my view? Well, that is something that is very often done and what we are going to create is a model for the view, which is going to wrap all the data that my view will need. I'm going to basically create a view model. So let's do that. I'll add another folder here where I'm going to put all my view models. Let's call this folder ViewModels. And so what I typically do is I create in here a class and I'm going to call it HomeViewModel and in there I'm going to list all the data that I need inside of my view. So that would be the string Title as well as the list of pies, and make sure that my Pies type is known here. There we go. Now I'm going to use this inside of my home controller. In here, well let's remove this use of the ViewBag and instead create a new homeViewModel. Add another using statement here and I'm going to initialize this. Well, let's set the title to Welcome to Bethany's Pie Shop, and let's set the Pies to the pies I've retrieved from the repository. Of course I need to make a small change here as well. I'm not going to be returning the pies, but I'm going to be returning that homeViewModel. Something is going to break, I hope you understand. Well, let's go to the view and we should make a change here. This is not going to be typed in IEnumerable in Pies any longer. No, this is going to become BethanysPieShop.ViewModels.HomeViewModel. This I can replace by now saying I'll go into Display here, the Model.Title and I'm also not going to be looping over the model, but I'm going to be looping over the Model.Pies. Now the result should still be the same, but let's make sure I haven't made any mistakes here. And there we go. We also see the updated title so we are effectively making use of that view model here. Now let's go back to the slides.

Styling the View

In the final section of this module, we'll focus on making our view look a little bit better from the visual perspective. As you may remember, this is what our view should look like, although at this point it doesn't really look like that. Although it does work, it's not really just the same. That's of course due to the fact that we haven't added any styling just yet. Now in the upcoming demo we'll be using Bootstrap, a very popular CSS library. Now Bootstrap is of course a client-side package that we need to add to our project. Managing packages such as Bootstrap can be done using NuGet, but most often they're added to a project using a client-side package manager such as Bower or npm. Visual Studio and ASP.NET Core currently still support adding packages using Bower. Microsoft has, however, announced that support for Bower will be removed and replaced in the next update. We will, however, in this course still use Bower while we await what Microsoft announces for the next version and update the course once it's released.

Demo: Adding Style with Bootstrap

In the final demo of this module we are going to see how we can in ASP.NET Core 2.0 MVC add client-side packages using the Bower client package manager and we'll see how we can add Bootstrap to our application. If you think back of the demo that I showed you in the beginning of this course where I showed you the finished application, well this doesn't look anything like that one, but it's just basically some styling that we'll need to apply. Let's do that now. So we're now back here in Visual Studio and what I'm now going to do is I'm going to style my application using Bootstrap. Now to bring in client-side dependencies, client-side packages, let's say, I can actually use different approaches. I can use npm to work with packages, but at this point, Visual Studio 2017 still supports Bower by default. Bower is just like npm, a client-side package manager that we can use to manage our client-side packages such as Bootstrap. So what I'll do is right-click here on my project again and I'll add a Bower file. Select New Item and let's search for Bower. They have a Bower configuration file that we'll use to list out all the dependencies we have in our application and here's what a file looks by default. In its dependencies collection I can basically now add multiple dependencies. Now I'll specify that I'll need Bootstrap here. As soon as I save this, keep an eye on the wwwroot folder in Solution Explorer. Notice that I can now expand this and this how created a lib folder and in that lib folder we can now find Bootstrap. Bootstrap itself has a dependency in jQuery so that was included by default as well. Now that we are working in that wwwroot folder, I'm also going to be adding my own files here. I'll add some images and I'll add those in a folder called Images. That seems to be a good name. I'll add an image called pattern.png, which is going to be the background pattern in my application. Now while we're at it, I'm also going to be adding some custom CSS. I will typically place my CSS in a folder called content and in there I'll create a new CSS file. Select the Style Sheet and let's call this site.css. In this CSS file I'm simply going to be setting some padding and a background image, which is going to be repeating and it's going to be pointing to that pattern image I had just added. Let's save all that. Now since we have a layout template I can also use this to reference my CSS files. I'll add a reference to both Bootstrap as well as my own CSS file. Let's save that. I'm also going to update this a little bit to use some Bootstrap layouting. I'm going to introduce a little bit of Bootstrap here and it's going to be used to position things a bit better on my page. There's some basic Bootstrap code used basically for positioning and what I'll also do is I'll go back to my index cshtml and I'll update this code. I'm going to replace it with another snippet here. That is going to now use the image thumbnail of the pie, the price, the name, and the short description, again using some Bootstrap code. If we now run the application again we should see a much better-looking application. Let's try that. There we go. Now we start seeing those delicious pies. We also see the name, the price, and a short description, as well as the background, which is repeating. I'm pretty happy with the result; I hope you are, too. Let's now summarize this module.

Summary

Whew! That was quite a lot of work, but now we have a working page that fully supports the MVC pattern. We've in this module started from the model, we've added the controller, and finally we've added the view. While doing so we've also seen the layout file as a template and the ViewStart as a way to link the view with its template. Finally, we've gone back to our page's view code and updated it so that it looks quite a lot better and we've used Bootstrap for that. I hope you enjoyed creating that page together with me. In the next module we are going to update our code again so that we'll use a real database and we're going to use Entity Framework Core for that. Let's do this!

Adding Data with Entity Framework Core

Module Introduction

Hey there, and welcome to another module of pure ASP.NET Core 2.0 MVC fun. I'm Gill Cleeren and I'll also be guiding you through this module. In this module we'll keep working on our application and as the title of the module has given away, we'll be adding support for Entity Framework Core to our application. So far we have been using mock data so by the end of this module we'll have support for a real database in the application. Let's kick off the module again by taking a look at what we will be covering in this module. We'll start the module by exploring Entity Framework Core or EF Core for short. Once we have enough knowledge about the framework, we'll see how we can add it to our application. After that, I will show you how we can through the use of EF Core, initialize the database from code. Finally, we'll finish the module by looking at the approach we'll need to follow when we are modifying a model in the code. I will see that we can do that using something called migrations. Let's get started.

Hello EF Core

So as promised, we're going to kick off things in this module by exploring what Entity Framework Core really is and what it can do for our application. You probably already have heard of the Entity Framework Core framework, perhaps in the context of ASP.NET MVC where it was also often used. Entity Framework Core is really the next version of Entity Framework, the ORM or Object Relational Mapper from Microsoft. And ORM will allow us to work with data in the database without having to write directly any SQL statements in most cases and basically allows us to work with the types that will behind the scenes be persisted in the database. Just like previous versions of Entity Framework, EF Core will allow us to work with LINQ statements to interact with our data. EF Core is a very lightweight framework and can be a good companion in combination with ASP.NET Core 2.0 web applications. Just like .NET Core and ASP.NET Core, EF Core is also working in a cross-platform way so it will still allow you to interact with the database on Linux or on other platforms. Also, again just like .NET Core, EF Core is open-source. You can find all the source code on GitHub. EF Core out of the box supports SQL Server; that is basically the default for ASP.NET Core 2.0 applications, but we're definitely not limited to using only SQL Server. EF Core allows us to work with other databases as well, even non-relational databases are supported. While that is not the focus of this course, it's interesting to keep this in mind if the requirements of a project force you to use another database. One thing that's also important to keep in mind compared to previous versions of Entity Framework is that EF Core only supports a code-first approach. A database-first approach is not supported any longer. If you are still a bit unclear on the role and goal of an ORM such as EF Core, let me explain this in a bit more detail. In order to work with data in a database we can write SQL statements in our code or calls to our procedures from our code. While that approach has worked for many years, an ORM can make the life of a developer much easier. Developers typically like to work with classes and objects and that's where EF Core comes into the picture. The ORM sits between our code and the database and we'll translate operations we do with objects into changes in the database. This enables us as developers to focus on the code part and not so much on the database side of things. In our application, we have a type called Pie with all nested properties. Through EF Core this type will be mapped to a table in the database where you can see the properties appear as columns in a table. EF Core will typically take care of creating this table for you. Based on the properties of the Pie class, it will figure out what types it needs to give to the created columns. EF Core uses quite a few conventions for this as well. For example, on the Pie class, the property with the name id or pie id will be default become the primary key in the table in the database. In our code we can now perform operations using pie instances and EF Core will update the corresponding table in the database accordingly.

Adding EF Core to the Application

Now that you have an understanding of what the purpose of EF Core really is, let's see how we can now include support for EF Core in our application. I'll be showing you most of the changes we need to make to our application in the upcoming demo since as you'll see, we'll need to perform quite a few steps to make things work. Let's start by listing out the steps we need to follow to add EF Core to our application. Of course, we'll need to create domain classes. Remember that I said that EF Core only supports code first. Based on the domain we create, we will ask EF Core later to create a database. Secondly, we'll need to create a database context class. This class plays a very important role in the way that we work with Entity Framework. It can be seen as the intermediate, let's say, between our application's code and the actual database. We'll see how to create this class as mentioned in the demo. Next we'll need a way to let our code know how we can connect with the database and quite logically we'll be using a connection string to do this. In previous versions of ASP.NET we typically placed this inside of a web.config file. That has changed to .NET Core as well and you'll see in the upcoming demo that we now can place this inside of an appsettings.json file. Finally we'll also need to make some more configuration changes in the startup class again. Remember that by default, ASP.NET Core doesn't know anything about Entity Framework so we'll need to make some configuration changes for it to support EF Core. You can see a snippet of the context class that we'll need to create. This class inherits from the DbContext base class; that is a requirement. Secondly, for each domain entity that we'll want EF Core to manage and sync with the database, we'll need to create a DbSet. At this point in our application that's just a single one, but of course typically you'll have more than just one.

Demo: Adding EF Core

Alright, time for demos. In this demo we'll add support for EF Core and so we'll add the DbContext, we'll update our existing mock repository to now work with EF Core, and we'll also need to make some changes to the application's configuration. Let's get started! So in this demo I'm going to show you how I can use a real database in our application and we're also going to replace the MockPieRepository with a repository that is going to use the actual database. I will be using Entity Framework Core for that. Now to work with EF Core, the first thing that I will do is I'll create a class that is going to be my database context, my DbContext, and since this has to do with the model, I'm going to put that in the model's folder. I'm going to create a simple class and I'm going to call that AppDbContext. Now a context is basically, let's say, the intermediate between your code and the actual database. It's a bit like the traffic agent that moves data back and forth between your code, your application, and the actual database, but to become a database context, my class needs to inherit from the built-in DbContext. That's a type that comes with Entity Framework Core. Now it's required that my DbContext has a context of DbContext options, otherwise it won't work. Now to pass in these options, I have the ability to either override a method called on-configuring, but I'll use the other option and that is via a constructor argument. So let's create the constructor and as the parameter I'll pass in DbContextOptions in my own AppDbContext, and these options need to be passed to the base type. In my AppDbContext class I'm going to specify which types need to be reflected in the actual database. So each type that I'll want to create as a table, I'll specify here as a DbSet property. So at this point I want EF Core to manage my pies in the database table Pies. So at this point my AppDbContext is more or less ready. EF Core is now aware of the Pies type and it will later on also create the corresponding table in the database. Now we're going to move away from using the mock Pie repository and we're going to use a repository that is going to use this DbContext. So I'm going to create a new Pie repository. There you go! Of course this needs to implement the IPieRepository interface. In here I'll now need to have access to my AppDbContext, and how do I get access to it? Well, like we've already done before. We'll pass it in via constructor injection and this may ring a bell, is this already known in a dependency injection container? Well, not yet. We'll do that in just a second. So in here I'll pass in an appDbContext and of course I'll make that local instance equal to the passed-in instance. There we go. Next we'll implement the interface and in the GetAllPies method, I'm going to of course use my DbContext to return all the pies from the context. Now what this will do behind the scenes it will check if the pies collection on the context has already been populated. If not, it will actually load in the data from the underlying database. In the GetPieById, we'll again use the context and its pies collection to retrieve the pie that matches the requested pieId. So now we have our AppDbContext, we have a pie repository that actually uses it, let's now go to the configuration of the application. I'll need to register in my ServicesCollection that I'll be using EF Core. That's what I'm doing here. I'm calling services.AddDbContext and I'm passing in my own context type. In the options at this point I'm specifying also that I'll want to use SQL Server. Now when we work with a SQL Server database we'll need a connection string to connect with that actual database. If you have done ASP.NET development before, you'll probably think of a connection string, which is going to be specified in the app.config file. Now ASP.NET Core does not use web config files anymore. It uses a new configuration system, which is based on named value pairs and those configuration settings, so available as named value pairs can basically come from different sources. The most typically used source is an appSettings.json file. Let's say a comparable file to web.config where I will place in a text-based file configuration information. So what I'll do is in my application I'll add a new item, and that's going to be an appsettings file. It says that here is an ASP.NET configuration file. Let's add that and as you can see, there is already a connection string placeholder in there, let's say, that says that the default connection, so that will be the name of the connection, is going to use in this case localdb. Localdb is a development-oriented SQL Server instance that can be installed along with Visual Studio 2017. I'll also need to specify the database name here. Let's call this database BethanyDemo123. Now how this is file then been read out? Well, for that I need to go back to the program.cs. Do you remember that we had this CreateDefaultBuilder call here which was going to be invoked automatically? Let's take a look at the documentation again for this one. In here you'll find that a file with the name appsettings.json is automatically going to be read out. So if you keep the default settings or the default name for this file in this case, ASP.NET Core is automatically going to read out the appsettings.json file and it's going to convert that into configuration information that we can access from the application. I now need to go back to my startup and in here I'll need to pass in the read-out configuration information. Now this information is going to be injected in the startup via dependency injection so this IConfiguration instance contains all the information that was already read out by the program class. I can now use the configuration instance to ask for a connection string. The name of my connection string was DefaultConnection. So I'll pass in here the name DefaultConnection. Now in our Services collection we are also still using the MockPieRepository and let's replace that with PieRepository. So I've now introduced EF Core, let's now see what the result of this work is. It seems that we're getting an error and if you think about it, isn't that kind of logic? It says that it cannot open the database BethanyDemo123, which is logical. We haven't created the database just yet. Let's return to the slides for just a minute and when I will come back in the next demo, we'll create the actual database.

Creating and Initializing the Database

As you saw in the demo, we've made quite a few changes, but we aren't seeing our pies anymore. The reason is pretty simple. We have not data in the database just yet. Let's now see how we can initialize our database. EF Core can be used to perform this initialization. In our code we can check, for example, if data is already in the database and if no data is present, well then we can insert pies in the pie table in our database. Since ASP.NET Core 2.0, this initialization code should actually be moved into the program class. I'll show you how we can do this in the demo.

Demo: Creating and Initializing the Database

Well then, time for another demo. Let's now add code that will initialize our database with some pies when it is still empty. So we're kind of stuck here because we didn't have the actual database just yet and in using EF Core, we'll probably want EF Core to also create a database and manage it from there on. So we'll have to ask EF Core to create a database for us first and to do that we'll use a migration. Now migrations can be executed via the Package Manager Console. If you don't have it open, go to File, Other Windows, and then in there you'll find Package Manager Console. Make sure at this point before running the migration that you have built your application. That is necessary for EF Core to find all the types that you have in your AppDbContext. So what I'll do is I'll add a migration here using the command add-migration. I need to pass a name and as this is the initial migration, I'm typically going to name it InitialMigration. Hit Enter and wait a couple of seconds. The net result is that there is a folder Migrations, which has been created and in there, there was a migration added. There's an Up method in here, which is going to contain all the code to create the database table. In this case that is just one table, Pies, with the description of all the columns. A second method, Down, is simply going to drop the table. Now at this point we have created the migration, but we haven't executed it so we still don't have a database. The next command I'll use is update-database. What this has now done, it has compared the current state of the database with the migrations in my application. Of course in my case it saw that there was no database just yet so it created the database and executed the migration. Let's run things now and see what the result is. Well, it seems that we're not getting an error anymore so we can actually log into the database, but as we can see, it seems that we have lost all the pies. I find that quite sad. But this is of course because we didn't have any initial data in our database so let's do that next. I'll create a class and I'll call it DbInitializer. Now this is just a class that I'll use to populate my database with some initial data. I'll make it a static class and in here I'll use a static method called Seed and I'll want the AppDbContext to be passed along here. Now the Seed method is only going to be used to initialize my database with data if there is no data in there just yet. So what I'll do is I'll ask the context if it already has some data. In this case I'll check if there are any pies already and if that's not the case, well then I'll add some. I'm going to paste in another snippet here that is going to load into our database some pies. This is again a snippet that you can find in the snippets with the downloads of this course. Finally, I'll also need to make sure that my context will submit the changes to the underlying database because at this point the changes are just in the AppDbContext. So what I'll ask my context is to save the changes. Now what are we going to call this Seed method from then? In previous versions of ASP.NET Core we would typically have called this from the startup class. Microsoft now recommends against doing that and the best practice is now to call this from the program class where we are doing all the setup of the application. So let's go back to the program class and there we'll call this Seed method here. I'm going to comment out this line and replace it with a couple of lines extra. Make sure that the using statements have been introduced, and basically what I'm doing here is I'm using the dependency injection container to get access to the AppDbContext, which I need to pass to my DbInitializer. Then I can own my DbInitializer, called a Seed method, passing in the AppDbContext. I think we're good. Let's try writing the application again and we should now see a long list of pies. Oh, now I'm happy! All these delicious pies are showing and they have been loaded via EF Core from the actual database. Now let's go back to the slides.

Modifying the Model

In the final part of this module I want to discuss how we can cope with changes that need to happen in the model. How can we then update the database model from code? Let's take a look. Imagine that we need to make a change to the Pie domain class, for example, a new property needs to be added. Since EF Core has created the table in the database, we shouldn't manually go and update the table in the database, but still we'll need to update the database, right? EF Core comes with something called migrations to support this and migration will be created by EF Core and it contains the code that it will execute to bring the database model back in sync with the domain model. Creating migrations can be done using the Package Manager Console.

Demo: Modifying the Model

In the last demo of this module, let's take a look at how our application can cope with the change that we need to make to the model. We'll see how we can work with migrations to update the database. Let me start this demo by showing you the actual database that was created by EF Core. Now to view a database which has been created in localdb, you shouldn't be using your Server Explorer; instead you should use SQL Server Object Explorer. In there you'll see BethanyDemo123 and that defines a couple of tables. There's the Pies table and an EF Migrations history table. This is a table that is maintained by Entity Framework Core itself to keep track of the migrations it has already applied on our database. Currently in the Pies table there are the columns which correspond to the properties we have on our Pies type. Well, let's now go to our pie type and let's make a change to this type. For example, let's add another Boolean IsInStock and let's now assume that I want to make sure that my database remains in sync with my model. So what I'll need to do is well, first I need to a build. That I need to do in any case. Then I can go back to my Package Manager Console and do add-migration. Let's type add-migration again and give it a name PieModelChanged, for example. EF Core has created another migration called PieModelChanged. We can see here that it has figured out that the IsInStock property has been added on the Pies so it needs to create another column for that as well. When we now run update-database again and we go back to SQL Server Object Explorer, let's refresh, and let's now go back to BethanyDemo123. In the Pies table we'll see that the corresponding IsInStock column has been added and that is how we keep the database and the model in sync. Changes in the model have been synchronized to the underlying database.

Summary

And we have reached the end of another module. Let's do a small recap of the most important learning takeaways from this module. We have seen that with EF Core we can add support for working with the real database in our application. EF Core is the .NET Core version of Entity Framework. One of the biggest changes that we have seen is that EF Core only supports a code-first approach, which is what we have of course used in this course as well. We've also seen how we can make sure that the model and the physical database remain in sync through the use of migrations. Up next is adding navigation to our site. We'll need more than one page of course as well and that's exactly what we'll be doing in the next module. Thanks for watching!

Adding Navigation to the Site

Module Introduction

Alright, I'm ready for another module in the Building Your First ASP.NET Core 2.0 MVC Application course here. I'm Gill Cleeren and in this module we are going to extend the application a little bit more. Up until now the application just had one page and for a real application that's a pretty low number. So what we'll do in this module is see how we can navigate to another page in our application. I'll be explaining how we can add navigation in an ASP.NET Core application. Good traditions never stop! So we'll start this module as well with a small overview of what we will be learning in this module. First, I'll start by explaining to you how navigation in ASP.NET Core MVC works. Next, we'll make some changes to our application and we'll add support for navigation.

Understanding Navigation in MVC

So before adding support for navigation in the application, let's understand how things work in ASP.NET Core 2.0 MVC. If you think about the traditional way of browsing to a page, basically what happens is you're going to make a request to the server. For example, a request for File1. What happens on the web server is that the server will search for that file and when found, that file will then be sent back to the client. What's important to understand here is how the request is handled. A request is made for a file that physically exists on the disk and when the requested file is found, that file is simply returned to the client. Now when we're working with ASP.NET Core MVC, things are happening completely differently. Remember that when using MVC, requests are handled by action methods on controllers. So when a request comes in, MVC will need to map the request to the correct action on the correct controller. There's no link anymore between the incoming request and a physical file on the disk. The question then remains, how will ASP.NET Core MVC know which controller and which action it needs to invoke? Well, the answer is a process called routing and it comes with ASP.NET Core 2.0 MVC by default. So does routing work then? Routing is based on routes, which may sound logical. Each route is a URL pattern and that will be mapped to a handler. The handler in the case of MVC is the action on the controller. The fact that we can use patterns for our routes means that we don't have to type all the possible routes for our application. What will happen is that the URLs are compared with these patterns and when a match is found, in fact, I should say when the first match is found, that pattern will be used to trigger the corresponding action. Take a look at the basic URL you see here on the slide. http://www.bethanyspieshop.com/Pie/List When we look at this URL, we can see that it's made up out of a number of parts. First you have the host, bethanyspieshop.com, as well as Pie and List. These parts are known as segments. Typically the first part will point to the controller and the second part will point to the action. Of course we have to somehow explain this to the thing in MVC that controls all this and that's the routing engine. It is not something that works by default, this mapping thing. We'll have to create a route that says that when a URL with a pattern with two segments is encountered that we will assume that the first one is the controller and the second one is the action. You can see the pattern here; it's controller/action and each of these segments is surrounded with curly braces. So the route /Pie/List will therefore be handled by the List action on the PieController. Now as you can see here, at this point, this action doesn't specify any parameters just yet. Sometimes we'll need more segments in the route. Imagine that you want to view the details of a pie with id 1. The request for this could be /Pie/Details/1 where now the last segment is new. We'll need an extra segment in the route pattern. As you can see here, the id segment has now been added. And that request that we just saw can be handled by this action here. The Details action now has an id parameter and by adding this extra segment it's now possible to get this value into the method parameter. This should already give you an id about routes in ASP.NET Core 2.0 MVC. Let's see how we can create those in our code. Now you may be thinking, how did we actually see our page while we didn't create any routes just yet? Well, the answer is that we have asked ASP.NET Core to configure our site with a default route. By calling in the startup UseMvcWithDefaultRoute, automatically a route has been added, a default route in fact that will work for most applications and you can see the route here. It contains three segments; one for the controller, one for the action, and one for the id parameter. While a lot of applications don't really need a lot more and can perfectly work with this default route, we typically will create our own routes. By the way, don't worry about home and index that you see here in this snippet. Those are defaults and I'll explain those in just a minute. In the code snippet that you see here, I'm taking control and I'm going to create my own routes. Instead of using UseMvcWithDefaultRoutes, we are going to use UseMvc and through the Options pattern we are going to register a route with MVC. Remember that a route needs a pattern and then when a request comes in, MVC will match the request with the patterns we have defined. Here the template is defined and the route is also getting a name. The pattern should actually look familiar. It's the same one as ASP.NET uses by default. The idea of managing the template myself here is that we might in fact need more than one template and so we'll need to have full control. Now what about the values that you see behind the controller and action then? Well, those are route defaults. For the controller this is set to home and for the action it's set to index. When a request comes in that doesn't specify a value for the controller or the action part, MVC will use these default values. This will allow us to send the request to the root of the site and automatically the index action on the home controller will be invoked. The third segment also has a noticeable character here, the question mark. By adding this value the segment becomes optional. It's not required to be part of this request in order for this to still be a match. Because we added this, this route can be used for both requests that include the id segment as well as routes that don't. The two URLs here you see here on the slide will both match the route. One has the id segment and the other one does not. But since this segment was defined as optional, we'll have a match for this route with both requests.

Adding Navigation

Now that you understand the basics of how ASP.NET Core 2.0 MVC handles navigation, let's take a look at how we can add navigation to our site. To create a link in our views, we'll use something completly new in ASP.NET Core 2.0 MVC called Tag Helpers. A tag helper consists out of mockup code that will get executed server-side. When we use it in our mockup code it'll therefore trigger the execution of code. This way for Razor code we can trigger certain functionality to happen. ASP.NET Core MVC comes with quite a few tag helpers built in and we'll see some in this module already. It's also possible to create custom tag helpers. Tag helpers are an extension point. If you've used previous version of ASP.NET MVC before, you'll notice that this all sounds very familiar to HTML helpers and that's correct. Although HTML helpers still work in ASP.NET Core, most HTML helpers have been replaced with tag helpers. Now the reason that I'm now all of a sudden explaining to you about tag helpers is that we'll be using tag helpers for navigation as well. Take a look at the snippet you see here on the slide. I'm creating a link using a plain anchor tag, an a tag. I want to have ASP.NET Core MVC generate a link based on the route or routes I have in my application. I can do this using a tag helper; in fact, using two tag helpers here. The asp-controller tag helper is set to Pie and the asp-action tag helper is set to List. As mentioned, tag helpers trigger execution of code. So when this view code here is running, it'll trigger execution of code that will look at the routes that we have in our application and based on these, it will generate a correct outgoing link. Here you can see the generated code for the link we've just created. Based on the routes we have in the application, the correct link is generated and that's all because of the fact that we've used tag helpers. The link indeed now navigates to the Pie controller and the List action. To work with navigation, ASP.NET MVC comes with some more tag helpers. We've already seen the asp-controller and the asp-action tag helpers. We'll also get the asp-route-\* tag helper where we can replace the asterisk with the name of the parameter, for example, id and pass it a value. The asp-route tag helper can be used to force ASP.NET Core to use a specific route, so by name. Now this later tag helper you probably won't be using that much.

Demo: Adding Navigation

Now that you have a good understanding of routing in ASP.NET Core and that you already know the basics of tag helpers, let's return to our application and include support for navigation. So let's now add navigation to our application. This is going to be a demo in which I'll show you quite a few things and the first thing that I'm going to start this demo with is showing you how we can add routes. So back here in the startup class and I hope you remember that in the very beginning of the course, I added this UseMvcWithDefaultRoute middleware. This middleware plugs in MVC in the request pipeline, but also configures it so that there is one route, which is going to be handling all our requests that I'm going to remove and I'm going to replace it by app.UseMvc and to UseMvc I'm going to now be passing my routes. The route I want to add, I can add using routes.MapRoute. A route can have a name, which in this case I'll set to default and in any case, the route will need a template. That template is then going to be used to match an incoming request with this pattern. Now the route that I have added here is the same route that was basically created by UseMvcWithDefaultRoute. So I haven't really made any changes here, but of course in a more complex application we'll use several routes by calling MapRoute several times with different routes. But for this application we'll keep things simple and we'll just keep one route. I have two other courses on ASP.NET Core here on Pluralsight and there I will add several routes so if you want more information about that, please refer to these courses. Now I do want to show you navigation, but at this point my application just has one page so it's kind of hard to show you navigation really. So what I'll do is I'll create a detail page, which will contain the details of a pie and I will be able to navigate to that page by clicking on a pie in the overview. So let's go back to the HomeController and let's add another action method called Details. This Details action method expects an id. I'm going to use my pieRepository to check if I can find that pie. If not, I'm going to return a 404 not found, otherwise I'm going to return a view, passing in that pie. So that's again going to be a strongly typed view. Of course we'll also need to create the corresponding view. So again in the home folder because we were still in the home controller, I'm going to add a new view, which is going to be my details view. As we move to defaults here and type this view to BethanysPieShop.Models.Pie instance. I will display the name of the pie, an image, the price, the name, the short description, and the long description. So now we can actually introduce navigation. So from my index view I'll want to navigate to this details view. When I click on the pie name I'll want to trigger navigation. So I'm going to start by introducing a link here. Now I want to navigate to the Details action on the HomeController. In previous versions of ASP.NET MVC I would have used an HTML helper. Now ASP.NET Core uses tag helpers for this. Now to enable tag helpers I need to introduce a file called the view imports file. Let's do that first and I'll add that view imports file here in my Views folder. Go to New Item here and under Web select View Imports File. We'll leave the default name, click on add, but this View Imports file as the name is really giving away, will contain imports that I will use throughout my application's views. It will basically enable or import stuff into the view. What I'll add here is my tag helpers. I'll basically say to MVC, import all tag helpers in the Microsoft ASP.NET Core MVC tag helpers assembly. When I save this file I'll be able to use tag helpers throughout my Razor views, and I will need those to actually make sure that the correct outgoing link is going to be created. When I click on a pie name, I'll want that the link gets generated that points to the HomeController and the Details action. Notice that asp-action and asp-controller as well as the anchor tag itself have become bold. This is IntelliSense that notices that we're now using a tag helper. This only works because I have this @addTagHelper in my view imports file here. That has enabled my tag helpers in my application. I will also need to specify to the details of which pie I'll want to navigate. That I can do using the asp-route- and then I can pass in my route value. Now the route value that I'll need will be id and why id? Because in my HomeController, my parameter here is id. This value will get matched by model binding. We'll see model binding in the next module, but for now keep in mind that this thing will actually search for the corresponding values on the incoming request and I'll set that to, well, that will be the pie.id. Remember that I always had to fully qualify the namespaces here. Now I can use the ViewImports file here also to add some using statements which will then make some namespaces known throughout my views. I will do that with the BethanysPieShop.Models and View.Models namespace. Let's do a build first now and now I can clean up my view code a little bit. Again, I remove this and notice that HomeViewModel is now still known. I can do that also with the Details view and remove this qualifier here as well. Now since we're talking about navigation I want to do one more thing before I show you the running sample. Let's go back to the layout and let us already introduce a navigation bar at the top, which is simply going to be an HTML5 NEF element, which is going to contain a UL for all the links in the navigation bar. I'm going to introduce the first link here, which is going to be a link to the index action on the HomeController, which simply says Bethany's Pie Shop and as you can see by the highlighted values, this is also using a tag helper. Let's run the application again and see if our navigation is actually working. I can click on a pie here. Notice at the bottom that we are in fact navigating to Home/Details/1 and here are the details of that delicious pie showing now. Also I can click here and that will take me back to the localhost with the port number. Now as you can see, ASP.NET Core does not place /Home/Index behind localhost. That is because in my route I have specified defaults. Home is a default for controller and Index is a default for action. The question mark behind id indicates that this value is optional and you can see that because we have been using the same route all along, both for the index action as well as for the details action. Let's now summarize this module.

Summary

Let's summarize what we have seen in this module. We started the module by looking at ASP.NET Core 2.0 MVC will actually route an incoming request to an action method on a controller. We've learned that patterns are being used for this to check if there's a match or not. Secondly, we've looked at how tag helpers can be used to create the correct outgoing links so that the generated HTML can be used to navigate to another page. We're making good progress here. In the next module we'll add yet another page. We'll create a form where the user can send feedback to Bethany's Pie Shop.

Creating a Simple Form

Module Introduction

Hi there, and welcome to another fun module in the Building Your First ASP.NET Core 2.0 MVC Application with Visual Studio 2017 course here on Pluralsight. I'm still Gill Cleeren and I'll be guiding you through this module. The module title is already giving away what we will be doing in this module, is it not? Creating a simple form is exactly what we'll be doing here. I'll be showing you how we can add a contact form to our site and along the way you'll see how we can use MVC to validate the entered data and also store it in the database. Creating a form will thus be the focus of this module. You'll see soon that in ASP.NET Core 2.0 MVC the creation of a form will typically be done using tag helpers again. We've already looked at tag helpers in the previous module and you'll see more use of these here as well. Just creating the form is not enough; we'll need to also make sure that the data entered by the user is correct before we'll store it in the database. We'll see how this is done in the part, Model binding and validation.

Using Tag Helpers

First things first. Let's see how we can create the actual form and as said, we'll again be using tag helpers for this. Just a small recap of what we saw in the last module. Tag helpers are a new features of ASP.NET Core MVC and in most cases they replace the HTML helpers that you may have used in previous versions of MVC. They're a server-side thing. A tag helper will typically be a new tag or an attribute that is only used in Razor markup code and therefore will not appear in the client-side code. However, by including tag helpers in our Razor code we will trigger the execution of code. Remember that in the last module we used the asp-controller and the asp-action tag helper and that in turn resulted in the creation of the correct link. Creating that link was done by code that sits behind the tag helpers. ASP.NET Core 2.0 MVC comes with quite a few tag helpers already built in, but custom ones can also be created. When we want to create a form in MVC, we will typically also use tag helpers. MVC comes with quite a few tag helpers out of the box that help us in the creation of forms. The actual form tag is a tag helper. That may sound a bit confusing since the form tag is an existing HTML tag and now all of a sudden it has become a tag helper? Well, yes, that's correct. Microsoft has basically extended the form tag and when we use it, code will execute. You'll see me using the form tag helper in just a minute in the demo. The input tag helper is also a built-in tag helper and it can be used to link an HTML input with a property on the model that we're binding to. Other form-related tag helpers that come with ASP.NET Core 2.0 MVC are the label tag helper, the Textarea tag helper, and the select tag helper. Also, several tag helpers are included that are related to the validation of data, but we'll see those later in the module. You can see a small sample of me using the form tag helper. As you probably can see, the form tag itself is nothing special, but it can be used in combination with the asp-action attribute, which will then point us to the action that needs to be invoked on the current controller and the form is posted back to the server. Here's another example. I'm using the label tag helper here. On the label tag I've added the asp-for attribute, which has its value set to Name. Name is a property on the model that we are binding this form to. When this tag helper executes, again, code is executing and that will result in this HTML here being generated. In this case, a for attribute is being generated, which has its value set to Name.

Demo: Creating the Form Using Tag Helpers

Now that you already have a deeper understanding of tag helpers, let's spend some time in the new demo where we'll be creating the form. So in this demo we are going to add a form where the user can add feedback to Bethany's site and in that form we will of course will be using some tag helpers. Now I will be storing all the feedback that is entered via the site in the database so therefore I will create again a domain object, feedback, and the repository. Now to save us some time I've already done that and you can see the result here. So here you see the feedback class, which simply defines an id, a name, an email, a message value, and a contact name, which is a Boolean. I've also created this interface, the IFeedbackRepository in which I simply have one method defined, which allows us to add feedback. In the FeedbackRepository, which of course implements the IFeedbackRepository, I can inject the DbContext and I use that context to add feedback into my database. In the end I will then call SaveChanges, which will persist the data to the database. Now since I want EF Core to also manage the feedback items, I have created a DbSet in the AppDbContext just like we've done before with the pies. Then just like with the PieRepository I've also added my FeedbackRepository into the dependency injection container in the ConfigureServices method of the startup class. Finally, I've also run a migration via AddMigration that resulted in the creation of a new migration class and you can see the FeedbackAdded Migration class here, which is simply going to create the table Feedbacks. I've also run an UpdateDatabase. So now we're ready to create a screen where the user can enter feedback. Let's do that next. So I'm going to start with the creation of a new controller and I'm going to call that my Feedback Controller. I'm going to create a new item, select MVC Controller Class here and name my controller the FeedbackController. That controller will of course have a dependency on the feedback repository. And you should already be getting used to this. I'm going to inject that feedbackRepository dependency via constructor injection into this controller. So I'm going to add an IFeedbackRepository parameter here and call that feedbackRepository, and of course set the local field equal to the injected value. There we go. The template already created this Index action method and we'll leave that; we'll actually use that for now. And of course, I still need to create the corresponding view. So let's do that first. I'm going to go to my Views folder, create a new subfolder called Feedback. That's the name of the controller and in there of course I now need to create my new view and I'm simply going to call that Index. Now at this point this action method is simply going to return the index view; however, I'm going to create still a strongly typed view. I am going to type this to a feedback instance. So I'm going to set the model to Feedback. Now you may wonder why am I doing that? Well, I am going to in a form now capture all the data so that a feedback instance can be created via model binding and that will be sent back to the server later on and we'll see that very soon, don't worry. Now in here, since I'm going to capture data, I'm going to create a form. Now as you can see, Visual Studio has already detected that I'm using the form tag helper. Now this form tag helper does allow me to specify to which action I'm going to be posting my form. I'm going to use for that the asp-action attribute, as you can see, that also gets highlighted. I'm going to be posting back to the Index action method. Now you may be wondering that was the one that returned the view. I'll explain what is happening here in just a second. For now let's focus first on the creation of the form. In this form I wanted to do a post to my server so I'm going to set the method to post. Now for layout purposes I'm also going to set the class to form-horizontal. As you can see, I'm getting some IntelliSense help here from Visual Studio that has detected that I'm using Bootstrap and the role should also be set to form. There we go. Now in my form I want to capture all the data to create a new feedback instance. So let me start by creating fields to capture the name of the user. I've pasted in a small snippet here. In here you can see that I'm using a label and I've specified the asp-for to be Name. That will simply link this label that is going to be created with the input that is also going to be used to capture the name. Next I've created an input, again a tag helper and I specified asp-for to Name. Now behind the scenes, the data entered in this input is going to be linked with the name property of the feedback that is due to model binding and the fact that the tag helper is going to create the correct generated HTML, but we'll see that in just a second. Now I suppose that you seeing me type is not the most exciting thing of all; I've decided to paste in all fields as well using another snippet. As you can see, I've done pretty much the same thing for the email, the actual message that is going to be using the text area tag helper, and I'm also going to be using a ContactMe checkbox where the user can check if he or she wants to be contacted or not. Of course, I need a way to submit my form and I'm simply going to do that using a plain input type submit. So a plain button basically. Now at this point I think my form is pretty much ready. I have used tag helpers to generate the correct HTML. Now before I actually run this, let's go back to the slides for just a minute so I can explain to you a little bit more about model binding.

Model Binding and Validation

Although we now already have a form in place, we still need to be able to send information from the client back to the server so that the MVC application can then persist the information. In this part we'll see how this process is handled using model binding. We'll also spend some time looking at how we can perform validation so that we can let the user know that their entered data is perhaps not correct. Let's take a look, shall we? First, let's take a look at how we from the client can get the entered values into the parameters of the action within the controller. MVC will match the data coming from the HTTP request with the arguments that our action method needs. This process is called model binding. Using model binding, the objects which are required as input parameters for action methods are created based on the data that it gets from the HTTP request. Imagine that we have an action method that requires an id as parameter. MVC needs a value to be there in order to be able to call this method. The process called model binding will try to find a value for this id parameter. When we do a request to /Pie/Details/1 we actually want that 1 to go into the id field. Model binding will do this for us and for this purpose it will use something called model binders. Model binders are components which help providing the values from a certain location, let's say, in the request. For example, there's a model binder that will search in the form data of the request. Another model binder will search in the route for variables and a third one will search in the query string. This list of model binders will actually be invoked in this sequence and so as soon as a match is found, the process will stop and pass the value to our action method. Imagine that we have a route where the id segment has been defined. In this case the value id will come from the route variable model binder and it will provide us with the correct value. The model binding not only works for simple values, but it also works for complex types such as an entire object. In this case the model binding engine will look at the target object's properties and will search for these instead, again with the same model binders. If all Bethany's shoppers would be entering the correct data in the form, the world would be a better place. However, I strongly believe that it won't be happening any time soon so we should be validating the data coming from the client. We'll need to perform validation from the model where we are making sure that the data being sent can be used for binding it to the model we're expecting. If that is not the case, we'll need to provide meaningful error messages. ASP.NET Core 2.0 MVC's model binding engine can perform a check to see if the values provided by the model binder meet the requirements of the model object. We can do this using an explicit call in the action method to ModelState.isValid. This will return true if the model object has no validation errors and false if it has any. Only when the model is valid will be saving the feedback item to the database and we can then for example redirect the user. If, however, the ModelState.isValid returns false, we are returning the user to the same view. ASP.NET Core 2.0 MVC will out of the box perform some basic validation already for us. It will, for example, already check the type of data that is being sent back. That it can do on itself. However, most applications will typically require more complex validation to happen. One of the ways that we can do this is by adding validation attributes on our model classes. ASP.NET Core comes with quite a few attributes built in already, which can be used to cover quite some scenarios. Here in this snippet you can see that I'm using the Required and StringLength attributes. When the Required attribute is being used and no value is provided when the model binding is happening, the validation will fail. Similarly, when we use the StringLength attribute and the user has entered a value, which is greater than what was specified, validation will also fail. ASP.NET Core 2.0 MVC has more validation attributes on board. We've already seen the Required and the StringLength attributes. The Range attribute can be used to make sure that the numeric value is within a certain range. To the RegularExpression attribute we can specify an regex, a regular expression, which will need to be matched by the value of the property. If there's no match, an error will be raised. Finally, the DataType attribute and its possible values can be used to make sure that the entered value is required to be in a specific type such as phone number, email, or url.

Demo: Validating the Model

Let us now go back to our demo and add validation on our model so that our form will be more complete. So now back here in our form that we created in the previous demo. Now you may be wondering, what is actually happening when the user hits this Submit Button? Well, then of course, the form will be posting back to the Index action method on the FeedbackController and don't forget that I made this a strongly typed view. Basically the view knows that all the fields are going to be bound to a feedback instance. Now let's go back to the controller. If you remember correctly, we did already have this one Index action method, but this one was going to be called when a Get request is going to be sent to this controller. Now we'll also need to be able to cope with a post request being sent to this controller. So I'll add another action method that is going to be able to handle this. So I'm going to create a new action. I'm going to set the return type again to IActionResult and I'm simply going to call that also Index. Why? Because of course I specified that I'm going to be posting back to the Index action method. Now I do need to specify that this action method is to be invoked when a post is being received and I'm doing that by adding this HttpPost attribute. Now if I want, I could also attribute this one with HttpGet, but that is not really necessary; that is basically the default. Now of course Visual Studio is complaining at this point. It says that I have two Index action methods, but keep in mind that this one needs to get in some data. I'm going to be expecting a Feedback object here. Now that feedback instance is actually going to be created using mobile binding. Basically what is happening is that at this point ASP.NET Core has through the system of model binding, looked in all the form data that is being sent with the request and it has searched in there for data that corresponds with the properties I have here on my feedback. So it has searched for a feedback id, name, email, message, and ContactMe. And those were of course the names of the fields that were generated by the tag helpers. That means that if everything went well and the user has entered the data, that this feedback instance is going to be sent back to my controller. Because I have that feedback now, I can simply pass it on to the feedbackRepository. I can invoke the AddFeedback method, passing in the feedback instance. Now when the user has entered that information, well, what am I going to do then? Well, maybe I should thank him for sending some feedback. So I'm going to return the user now, not to another view, but I'm going to redirect him to another action method. I'm going to use for that the RedirectToAction method here available on the controller and I'm going to specify the action method I want to send him to. I'm going to use the FeedbackComplete action method. Of course, I don't have that action method just yet so we'll need to create it. It's very simple to do of course. So I'm going to create a new action method that's going to also return an IActionResult and we'll call that, of course, FeedbackComplete. That has to be the same name. This one simply returns a view. So when this went okay, then the user is going to be redirected to this action method. Now of course we didn't create a corresponding view yet so let's go back to the Feedback folder and add a plain view and we'll call that, of course, FeedbackComplete. In there, I'll simply show a thank-you message. Now there's one thing I haven't done yet. I haven't added this feedback form to the navigation of the application. So let's go back to the layout and add another li here that is going to allow the user to navigate to the feedback controller and its Index action method. But before we run the application I'm going to add a breakpoint here in the post index action method so we can see the data that is being passed in. Alright, we're ready to run the application. Let's do it! So the first thing seems to be working already. The Feedback button is showing in the navigation; let's click it. There's my form. Now that we're here, let us take a look at the source of the page and here we indeed see that a form, which is going to be posting back to /Feedback has been created. The label's for property has been set to Name and the input has its name and id also set to Name. That was due to the tag helper of course. So we can now enter some data here in our form and now we're ready to send some feedback back to Bethany. Let's click the button and there we go. The breakpoint is being hit so the Post action method is indeed invoked. Let's take a look at the feedback instance that has been created here. As you can see, the data entered in the form has through model binding been passed into this instance. I can now add that feedback object to my database and then I will be redirected to the FeedbackComplete action method. And there we go. We now see indeed that FeedbackComplete is going to be invoked. Now at this point, as you may have guessed, it is still pretty easy to enter some invalid data here. I can leave out the name, I can enter an invalid email address and so on. So what we'll do next is we'll add some basic validation. I want to keep things simple; I'm only going to be using some attribute validation by adding some attributes here on the feedback type. In my other ASP.NET Core MVC courses I'm using a lot of other options to perform validation. Please take a look at those courses if you want more information on validation. I'm going to again use a little snippet here that is going to add some attributes. I'm going to bring in the correct using statements. There we go! As you can see, I've now added some attributes. For example, the name has a required attribute, which as you may guess is going to require that the name is being filled in before validation will succeed. The StringLength attribute allows me to specify what is the maximum amount of characters that the user can enter here and you also can pass an error message, which you'll see appear in just a minute. I'm also using the DataType attribute here. Now the DataType in itself does not really do any validation. It is just there to let the browser know what data is to be expected and depending on the browser, this may be handled in a different way. Then we also have the RegularExpression attribute which as you may have guessed is expecting your regular expression, in this case to validate the email to be a valid email address. Now how is this validation going to be triggered then? Well, let's first go back to our controller. Now validation on the model is actually going to be triggered when the model binding is happening. So basically before the index action method is going to be invoked already. Now when model binding happens, a side product is being created called ModelState and ModelState is going to contain the errors that were encountered when validating the object. ModelState also defines a property called isValid so basically we are going to specify that only when ModelState says that the validation has succeed through the isValid property, then and only then are we going to add our data to the database and only then also are we going to be redirecting the user to the thank-you page. Otherwise, well then I'm simply going to return the user to the same page that he was on and we're going to pass back the same data. Then the user gets to see the same view again with validation errors. Now these validation errors also of course need to be displayed somehow. Let's go back to our view and we have a couple of options to visualize the binding errors. The first option is using again another tag helper called asp-validation-summary. I have to apply that onto a div and what this will then basically do is it will look at the bound model, so the passed-in feedback instance and all validation errors that ModelState now contains will be shown here in the ul with per-validation error a new line. Alternatively, I can also right next to the invalid input display the error message. I'm using for that the asp-validation-for attribute and I specified the value to be Name. This way ASP.NET Core knows that validation errors for the Name value should be displayed here. I've done the same thing with the other input fields. Now let us run the application again and see if our validation is actually working. Now let's go back to the Feedback page, and well, let's try sending feedback directly and as you can see, here the values are being displayed in the validation summary and the values right below the inputs are using the spans with the asp-validation-for tag helper. So we've now correctly implemented the form, we've added model binding, and we've also added some basic validation. As mentioned in my other ASP.NET Core courses, I go much deeper in both model binding and validation. Let's now go to the summary of this module.

Summary

We've reached the end of yet another module. I hope you enjoyed this one again and learned a couple of new things. Let's recap what we have covered in this module. We've created a form and we allowed the user to enter data in our application. Through model binding this data became available to us via the parameter without us having to manually retrieve the data from the form fields. That is a great time saver. After that, we have seen how we can do validation. We've covered attributes here, which allow us to do some basic validation on the model properties. In the next and already last module of this course we will add support for ASP.NET Identity to allow the user to sign into the application. I hope to see you there as well.

Adding Login Capabilities

Module Introduction

This is the last module that we'll be spending together for now, but hey, let's make the best of it, shall we? Welcome to this module of Building Your First ASP.NET Core 2.0 MVC Application. In this module we'll be looking at how we can secure the application. Most specifically we'll be taking a look at how we can make sure that not just anyone can use the feedback form that we have created in the previous module, but instead, only logged-in users can do this. Let's take a look at the agenda for this module first. We'll start the module by exploring the API that we'll use to work with security, which is the ASP.NET Core Identity API. I'll give you the needed information to get started with this API. I'll give you the needed information to get started with this API. Next, as we have done before, once we have understood the fundamentals, we'll add support for authentication to the site and make sure that as mentioned, only logged-in users will be able to send feedback to Bethany.

Exploring ASP.NET Core Identity

So let's kick things off in this last module by taking a look at the ASP.NET Core Identity API. So what exactly is ASP.NET Identity? Well the ASP.NET Core Identity framework is basically a membership system that's built into ASP.NET Core. Using ASP.NET Identity we will be able to authenticate and authorize users. Authentication has to do with knowing who is trying to log into our site. Authorization will allow us to manage which permissions a certain user has to have to access a certain resource, so checking if the user can see a certain page. Furthermore, it can allow us to manage roles, claims, and more advanced options related to identity management in our ASP.NET Core application. To do this it can even allow us to work with external providers such as Google or Facebook; however, we won't be discussing the later here in this course. Refer to my Enterprise ASP.NET Core course here on Pluralsight to learn about this. Here we're limit ourselves to adding an ability to log into the site. To manage the authentication and authorization information, ASP.NET Identity can work with SQL Server out of the box and that's also what I'll be doing here in this course. To manage the authentication and authorization information, ASP.NET Identity can work with SQL Server out of the box and that's also what we will be doing in this course. In the next demo we'll take a look at what it takes to add support for ASP.NET Identity in our application. We'll first need to update the AppDbContext. It will now need to inherit not from DbContext, but instead from IdentityDbContext. Next we will see that a lot of the work we need to perform is configuration work. We basically need to add support for ASP.NET Identity in our application by configuring it in again, yes, our startup class. I'll show you how this is done in the next demo. Apart from the basic configuration to just support ASP.NET Identity, it also allows us to further constrain, for example, the user registration. If you want to for example, force users to use a strong password, which we should by the way, we can configure identity so that it doesn't allow simple passwords. You can also use the options to configure how security cookies should behave such as the expiration time and we can also configure user options such as the fact that the email used upon registration should be unique. Here you see an example of this configuration code. In the code you see here, which does go in the startup class, we are using identity options to configure that the user password requires a digit, the length should be at least 8 characters, and it should also contain a non-alphanumeric character. I'm also requiring that the user who is registering on the site should do so using a unique email address upon registration so when we try from code to register the user, ASP.NET Identity will perform all these validation checks and registration will only succeed if all conditions are met.

Demo: Adding ASP.NET Identity

Alright, let's now return to our demo application. As mentioned, we'll first need to perform some work to actually add support for ASP.NET Identity to the site. That's exactly what I'll be doing in this first demo. Now the first thing that I'll do to add support for ASP.NET Identity into my application is changing this DbContext base class into IdentityDbContext. Of course, I need to bring in ASP.NET Core Identity Entity Framework Core. This already gives away that Identity has a dependency on Entity Framework Core. Now this is generic class and it can pass in the type I want to use to represent the user. I'm going to for now use the built-in identity user. IdentityUser is a class that already defines the base properties that you'll want to capture from a user such as the email, the username, the password, hash, phone number, and so on. Of course, we can expand this and I am doing that in my Enterprise ASP.NET Core MVC course where I'm actually adding extra properties to this IdentityUser class. Take a look there if you want to have more info on that. Now since we want to use Identity, well, like we've done before, I also need to configure it in the startup of my application. In the ConfigureServices method I'm going to use the services.AddIdentity method, which is basically going to add the default identity system configuration. I'm passing in the user and role type I want to use for this. I need to bring in our using statement, there we go, and I also point out that it should use my AppDbContext to basically store the information. You'll see that when you use Identity that you don't really have to write any queries to work with users with roles and that sort of thing. It's all wrapped in an API, but of course, underneath the data needs to be stored somewhere and then it's worthwhile pointing out with this AddEntityFrameworkStores. It's pointing out the data store that we'll use for that very purpose. Next in the Configure methods I also need to configure authentication middleware. For this I'll need to use app.UseAuthentication. Make sure that you put this before MVC. Like I said in the beginning of this course, this is a pipeline in which components are being placed in a sequential order and it also will be used in a sequential order when a request comes in. Authentication needs to be placed before MVC because of that reason. Let's build everything now. Alright, that has succeeded, and although we haven't done any changes to the model, we need to update our database so it will also contain the tables that come with ASP.NET Identity. So we'll go again to the Package Manager Console and we'll add another migration. I'll call this migration IdentityAdded. This is quite a large migration as you can see, because quite a few tables will have to be added. Let's first run UpdateDatabase and that will take a look at the result. So UpdateDatabase went successfully. Let's go back to SQL Server Object Explorer and take a look at our database. And now in the Tables list we'll indeed see quite a few tables which have to do with ASP.NET user and role information. Those are the tables entirely managed by the ASP.NET Identity API so you won't really have to be doing any querying on that. That is all done through the API. Let us now return to the slides.

Adding Authentication to the Site

Now that the site is configured with ASP.NET Identity, it's time to start using it. Just configuring it won't really do a lot of miracles now, will it? Let's see how we can add support for authentication to the site in the next part. So what do we need to allow users to authenticate in our site? To allow users to log in and register we'll need to create the corresponding views. These will be simple views, nothing really special about it. So we'll have to create the view using tag helpers again where the user can enter the relevant information to register or to log in. These views will be linked to a controller that will typically include to manage all login-related functionality. This will typically be called my account controller. Finally, we'll also have to make some changes in the model and the view models. We'll see how all this is done in the next demo in just a minute. In the account controller that we'll create, we'll be using some of the classes that come with the ASP.NET Identity API. The UserManager class is used to manage all interactions with user objects in the data store. Basically think of this class as your way to create users, delete users, and in general, do all interactions regarding user objects. It will also take care of doing all relevant changes in the data store. So in our case, the SQL Server database. Next, the SignInManager class will be used for user authentication and related actions. It defines methods such as password sign-in async which accepts a username and a password. When we trigger this method it will try authenticating the user. It will return a so-called sign-in result, which will contain a succeeded property indicating whether or not authentication attempt was successful. Many other methods are available on this class such as sign-out async, change password async, confirm email async, and much more. What you'll need to remember here is that these two important classes of the Identity API are offering us an abstraction over the details of working with authentication and user management. The Identity API makes this difficult task a lot easier and since underneath it's built on top of membership, you can rest assured that the implementation of your security-related features here is done in the right way when using this API.

Demo: Adding Authentication to the Site

Now that we know what we need to create next, let's return to the application and first add the required views. So the Login view and the Register view. Next we'll also create the mentioned AccountController and we'll need to make some changes to the model as well. Let's get started! To add support for authentication, so allowing the users to register or log in to our site, I'm going to start with the creation of a new controller and let's call that controller the AccountController. So let's get into our Controllers folder and add a new controller. Select MVC Controller here and let's call that AccountController. In here I'm going to be using the two classes that I mentioned already in the slides. I'm going to be using the SignInManager and the UserManager quite a lot. Of course I have to bring in some using statements here again. The SignInManager will allow me to do stuff such as checking whether users can log in, allow also users to sign in, to sign out, and many other options. In my Enterprise ASP.NET Core course, I cover quite a lot of features of the SignInManager. Secondly, the UserManager then allows me to work with users. I can check if a user is a role. I can create users, I can enumerate users. A lot of possibilities to work with users in the data store. Of course, I'll need an instance of these and you already know the drill. We'll use dependency injection. You may be thinking, well, I did not add any extra services to access the SignInManager or the UserManager. That's correct. They are automatically available via ASP.NET Core. So you don't really need to add anything; they will just be injected in the classes where you need them. Now what will my AccountController need to do? Well, I think we're going to have to be able to log in, of course. So I'll replace this Index method with LogIn and it is simply going to return the View. Let us first create of course the view already. So let's go to Views, create a new folder, Accounts and in there I'll create the corresponding view. So first we'll create the LogIn view. Now in this screen I'll want to capture the information entered by the users so the login information, a username and a password that is. Now I don't really have a domain class that can actually be used for this. So like we've done before, I'll create a login view model. So it's going to be simply a new class and I'll call it LoginViewModel. On this LoginViewModel I have defined a username and password property and I've also used some attributes again. Both values are required and I'm using one that I haven't used before, the Display attribute, which will override that in the UI, user name without a space will be shown. So this value instead will be shown. In here I use DataType.Password to indicate that a password is to be expected. That will result in an input type password being generated. I hope you also understand why I'm using a view model here. This data, I don't really have a domain class and I also don't really want to create a new domain object just for this. This is simply data that is used in this particular view so I create a new view model for it. Just go back to our login and type this now to my new LoginViewModel. Now to save us again some time I will paste in the form I will use to allow the user to log in. There's really nothing special going on in this form. Let me take you through it. I have a link at the top that also redirects the user to the Register action method on the AccountController. We haven't created that; we'll do that next. Then I have a form where the user can log in via the Login action on the AccountController. I have a validation summary again and I'm again using the Label tag helper and the Input tag helper. I'm also doing validation so that the validation on the login view model will be triggered. At the bottom I simply have an input type submit. Now of course we need to capture that data in my AccountController again. I'll need to create a second login method that is going to be triggered when a post is being sent to this controller. Again to save us some time I've used a small snippet here. Notice that I'm receiving now a loginViewModel and I'm first going to check if the entered data is valid. If that is the case, well then I'm going to use the userManager and I'm going to ask it to search if we already have a user with the specified username. If that actually returns a user, well then I can try to sign in the user. I'm going to use that user and I'm also going to pass in the entered password. If that all goes well then I'll redirect the user to the Index action method on the HomeController. If not, well then I'll simply show a generic error saying User name/password combination not found. I add that as a custom error on the ModelState and that error will then be shown in the validation summary that we have in the view. I will fast forward now to a point where I have added also the registration methods. Here you can see the get Register action method. And the post version as well, of course. I'll first again check if the entered data is correct. If that is the case, well then I am going to now manually create a new identity user. I then ask the userManager to create that user using that user information of course and the provided password. If that all goes okay, then I redirect the user to Home Index. If not, then I'll simply return the user to the same View, passing again that loginViewModel. I've also added the Logout action method which is also going to be a post action method. In here I ask the signInManager to sign out the user and also to redirect the home page of the site. Let us now add to the layout a Registration and Log in button. I've added two new links here, one that goes to the Register action method and another one that goes to the Login action method. Let's try running this and see if we can already register successfully in our site. At the top you can now see my two new buttons. Well, let's try registering. Let's add the username and a password. Hit the Register button and I am indeed being redirected to the Home page so I think registration has succeeded. Let's take a look in the database. If we now go to ASP.NET Users and we ask to view the data, well then we'll see that one user with the username Jill has been created. So let's try logging in then. Hit the Log in button, and I think we're logged in, but hmmm, there seems to be a bit of a problem. I don't really see that I'm logged in as a user. Let's try fixing that. I'm going to go back to the layout template and instead of simply showing these two buttons, I will only want to show them when the user has indeed not been logged in. Otherwise I want to show the Log out button. Checking whether or not the user is signed in or not is something that is supported by the SignInManager. So I will actually need access to the SignInManager directly here in my Razor code. Now it is possible in ASP.NET Core MVC to also do dependency injection directly in the Razor code. For that I can use the @inject keyword, which is going to allow me to inject from the dependency injection container an object directly into my view. In this case I'll need access to a SignInManager in IdentityUser and the instance is going to be called SignInManager. That's a local variable basically. As you can see by the red squiggly, it does require me to add another using statement here. Now I can use that SignInManager in my Razor code. So what I'll do is I'll check if the user is already signed in. If that is the case, well then I'm going to create a new form, which is going to post to the Log out method on the AccountController. If the user has not been logged in, well then I'll simply show those two buttons again. Let's try running this again and see if we now see that we're logged in and if we can also successfully log out. So let's hit the Log in button, log in with the credentials I've created before, hit Log in, and now we indeed see that the Log off button has appeared and when I click that, well then I can again register or log in.

Authorization

Although we have now allowed a user to register in our application, we're not really using this information just yet. You could say that at this point we've only used the authentication part. We should probably also look at authorization part. We'll want to allow only logged-in users to access a certain resource. We can restrict access to a controller and all its action methods by using the Authorize attribute on the controller or on the action method themselves as you can see here. When a request is now received from an action in the methods of this controller, ASP.NET Core will first check if the user making the request is already logged in and if that is not the case, the request will simply not be allowed. Note that we're using the attribute in its most basic form here. We can, for example, also using this attribute, pass a role which is allowed and then ASP.NET Core will check if the user is part of the given role. If you're interested in knowing more about this, please check again my Enterprise ASP.NET Core MVC course here on Pluralsight, which dives much deeper in this subject.

Demo: Adding Authorization to the Site

In the final demo of this course we are going to use authorization in the form of the Authorize attribute to ensure that only logged-in users can access the Feedback view. It was one of the requirements that only logged-in users would be able to send feedback to Bethany. At this point I can simply as a non-logged-in user, click on the Feedback button and send feedback. Let's fake that by using some authorization in the application. I am here on the FeedbackController and like I said, I only want authenticated users to access this controller. So what I'll do is I'll add the Authorize attribute without any extra parameters. Let me bring in the correct using statement and let's now run the application again, and now what we'll see is that when I click the Feedback button, I will indeed be redirected to the login page. So it's only accessible now for logged-in users. That's exactly what the business requirements specified.

Summary and Course Closing

We've reached the end of the last module of this course. Let's recap what we have seen in this module. We've covered the ASP.NET Identity API and we have seen that with this framework we can authenticate users in our site. We can thus now know who is trying to access the site. Next we can use Identity to also authorize users. This means that we can check which resource the user has access to. Now before closing, I also want to remind you about some of my other courses in this series where you can learn more about ASP.NET Core and ASP.NET Core MVC. In the Building Your First ASP.NET Core Application course here on Pluralsight we build a similar application to this course; however, more features are added to the site so you'll have a more extended site after watching that course. Keep in mind that that course has been created at the time of ASP.NET Core 1.0 so things like the project structure are different. Next I also have an advanced course called Building an Enterprise Application with ASP.NET Core MVC. This course dives deep in many topics that make up ASP.NET Core MVC. This course is created with ASP.NET Core 1.1. So everything in that course is fully compatible with everything that you see here in ASP.NET Core 2.0. And that concludes this course. I want to congratulate you on finishing it. I hope you enjoyed watching the course and that you have learned a lot of new things that you can use in your professional career. Thanks for watching! Bye for now.