Course Overview

Course Overview

Hi. This is Scott Allen, and this is my ASP.NET Core Fundamentals course. ASP.NET Core is the latest web framework from Microsoft, and it's engineered to be fast, easy, and work across platforms. In this course, we'll build an application from scratch and see how to work with ASP.NET and databases to build a website of restaurant information. During the course, you'll learn about Razor Pages and API controllers. We'll be using the Entity Framework and create and execute Entity Framework migrations. We'll also see how to integrate client‑side libraries into ASP.NET Core and learn some of the behind‑the‑scenes internals. Along the way, you might even hear a story about my childhood. By the end of the course, you'll be ready to build your own applications using ASP.NET Core. But before we begin, make sure you're comfortable with the C# language, HTML, and have some general experience at web programming.

Overview of the 3.0 Upgrade

Here is a fun fact to know before you begin this course. I recorded this course using ASP.NET Core version 2.1. Since I recorded the course, Microsoft has released ASP.NET Core 3.0. There are no significant changes between version 2.1, which is the version I recorded with, and version 3, which is currently the latest version and probably what you're using if you always download the latest versions. But there are some small differences between 2.1 and 3.0 that you will notice as you work through the course. In a couple of places, I have updated the videos with notes to make these changes clear, and something else I've done is update all the source code for this course to the latest version in my GitHub repository for the course. You can find that at github.com/OdeToCode/OdeToFood, and it's here where you will see where I applied some of these changes to the code, like changes to use Bootstrap version 4, which is also a newer version of Bootstrap than what I used in the course. Bootstrap, being just a CSS framework, doesn't have a significant impact on the code that you write, but again, there are some changes and some differences. You'll also find a README file in this repository where I've placed some additional notes about the course, and inside of here I've also tried to annotate places where you might see some changes in the course due to the different versions. And if you run into any problems, please let me know in the Discussion forum for the course. And thank you, I hope you enjoy this course.

Drilling into Data

Introduction

Hi. This is Scott Allen, and welcome to my course. In this first module, we're going to create an application and start showing some data on the screen. This is going to be a fast moving module, and I will skim over some topics that we'll come back and explain the more detail later in the course, but the idea is to show you what it takes to get a basic web application together using ASP.NET Core. Let's get started.

Creating the New Project

In this course, I'm going to be using Visual Studio to work on this project. Since .NET Core works across platforms, including Linux and Mac, you might be using a different environment like Visual Studio Code or Visual Studio for the Mac. So while I'm using Visual Studio, I am going to include some notes for those of you using different environments in this GitHub repository, OdeToCode/OdeToFood. This is also a place where I can keep you informed of changes in the environment or in the framework since I've recorded the video. But our first order of business is going to be to create the OdeToFood application that we want to work on. So I'm going to come into Visual Studio, and in Visual Studio I can say File, I want to create a new project, and from the .NET Core templates, I want to create an ASP.NET Core Web Application. I'm going to place this into a specific directory, which is the OdeToFood directory, and I'm going to name the project OdeToFood. Once I press OK, I'm going to be taken to a second screen where I can select the specific type of template that I want to use to create this application. I'm going to tell Visual Studio that I want to use the .NET Core Framework, and specifically ASP.NET Core 2.1 on top of .NET Core, and I want to build a Web Application, not a Web Application (Model‑View‑Controller), a Web Application. And I'm going to leave all the other current settings at their default setting. So we will configure this for HTTPS, we're not going to enable Docker support, we're not going to start with any authentication. With that, I should be able to press OK to create this project. And while that's being created, let me flip over to the command prompt just to show those of you who aren't using Visual Studio how you can achieve the same result. I'll include more details on GitHub, but here's the important part. Since .NET is a cross platform framework, all other tooling that you need to build and create projects also works across platform through an application known as the .NET CLI. So once you've installed .NET Core on the Mac or on Linux or on Windows, you should be able to go to your command prompt, your shell, your terminal, and be able to type dotnet. Dotnet ‑h would show you some more of the commands that you can use with .NET, but if I do dotnet new, I have the ability to select a template that will create the exact same project that I just created in Visual Studio. Specifically, if I'm in a folder named OdeToFood and I execute the command dotnet new razor, the project that is created by that command will be the same as the project that's created by Visual Studio. If you want some more tips on how to create the project in non‑Visual Studio environments, again, visit my GitHub repository for the project. But for now, I just want to come back to Visual Studio, I want to hit Ctrl+F5. Ctrl+F5 is the default Visual Studio keyboard shortcut to run my project without a debugger attached. You can also execute this command by going to the Debug menu in Visual Studio and selecting Run Without Debugging. This will start a web server with my app. It's also going to launch a web browser that appears on a different window. So let me drag this over to the screen, and this is the application we're going to start with. Its running on localhost port 44372. Your port might be different, but this is over HTTPS. And it has just a few basic features where I can view a Home page, an About page, and a Contact page. This is the starting point for our application.

Editing Razor Pages

With the application running, I'm going to make my browser window as narrow as possible because I want to have the browser window and Visual Studio on screen at the same time. And I just want to point out that we have a responsive design, because when the viewport becomes this narrow, the menu collapses into the hamburger icon here, which I can still click on to get to the menu. And this behavior is provided by Bootstrap, so in this project is Bootstrap version 3. I also want to point out that when I navigate to the different pages in this application, like the Contact page or the About page, these will go to URLs like /Contact and /About, and in our project, ASP.NET Core is mapping these requests to Razor Pages. These pages live in the Pages folder, and when I am viewing localhost, some port, /about, ASP.NET Core is rendering the About page. So Razor Pages have a .cshtml extension, but that extension is not required in the URL. And like many topics in this module, we're going to come back and go into more details about Razor Pages later in the course. But for right now, I just want to point out that if you have the application running without the debugger attached, you should be able to make changes in the project and just save files and come out and refresh the browser and see those changes reflected in the browser. Now one change I want to make is to add a link into my menu to get to the restaurants that we're going to be working with in this project. And you'll notice that in the About page, there's no mention of OdeToFood, there's no menu links, so where does all this other stuff come from? Well, most of the Razor Pages in this project, like About and Contact and Index.cshtml, which, by the way, represents the Home page of the application, they all plug into a special page known as a layout page. You can find this page by default in the Shared folder. It has the name \_layout.cshtml. We'll find out later in the course that there's a number of different special razor pages, and most of them will be named with a leading underscore. So it is layout.cshtml that renders everything for my application that is inside of the head tags, so the links to my style sheets like Bootstrap, and also defines the opening body tag and the menu that I want to work with. So, if I browse through layout.cshtml, I should come across this section, which contains some Bootstrap 3 styles to be a navigation bar, and here can see the text Home, About, and Contact. That's exactly what appears in my menu, and that's because this is the menu for the application. If I want to add an additional item that I can plug in another list item, let's add an anchor tag with the text Restaurant, and you'll notice this one additional piece of information here, asp‑page. This is what's known as a tag helper in ASP.NET Core. Again, another topic that we'll go into more detail about later in the course, but for now just know that a tag helper helps me to render HTML on the server. What the asp‑page tag helper will do is set the href, or my anchor tag, to point to a Razor Page that I have in my project. So, /Index goes to the Index page. This is the About page, the Contact page. Again, I don't have to use the .cshtml extension, but let's say that we are going to have a Restaurant page. In fact, let's say that we're going to have an entire folder full of pages related to restaurants, and inside of that folder I'm going to have a Home view, or perhaps we can call it a List view, that will show the list of restaurants that the application knows about. And now all I need to do is save layout.cshtml, refresh my browser, and let's open up the menu and I can see yes, I now have a Restaurants link. That Restaurants link isn't going to work just yet. The next thing we have to do is actually add that Razor Page.

Adding a Razor Page

We've said that this new menu item should point to a List page in a Restaurants folder, so let's go to our project, and in the Pages folder I want to add a new folder, and we will call this Restaurants. I then want to right‑click on Restaurants and say that I want to add a new item, and in this case what I want to add is a Razor Page. Now for those of you who are not using Visual Studio, I'll give you some tips on how to generate this page in the next clip. But regardless of what you're using to create this Razor Page, we want the name of the page to be List.cshtml. And in Visual Studio, I can say the name is just List, and what this will do is create really two files inside of my folder, List.cshtml and List.cshtml.cs. Now first of all, it's technically my razor Page. You can see it has what's known as an @page directive that tells ASP.NET Core, yes, I am a Razor Page, and then it has another directive, the @model directive with a lowercase m, and this directive says that the model for this page is of type ListModel. In other words, an instance of the ListModel class is the object that this page will use to display information. If we want to display restaurant information, then ListModel will need to contain restaurant information. And ListModel is the class that we can see defined in the .cs file, which is nested behind List.cshtml. So you can see this forms in Visual Studio a little tree node that I can collapse and expand. Now if you're familiar with ASP.NET MVC and the model‑view‑controller design pattern, then I just want to point out that List.cshtml is very much going to be the view in the MVC design pattern, whereas the code that I place into my .cs file for the ListModel, the methods here are going to be very much like controller actions and the properties are going to be very much like the models that I would pass into a view. And if you don't know about ASP.NET MVC, then don't worry. You don't need to know about the history of this framework to be successful with this course. But right now, all I want to do is make sure that this List Razor Page is working properly, so let me add a header to the page and we'll just say Restaurant. I'm going to save that page. I'm going to come out, I'm going to refresh the browser to make sure that ASP.NET Core picks up the existence of this page. And with the browser refreshed, I'm going to click on Restaurants and hopefully see that I am rendering this new Razor Page that I created.

Using the Scaffolding Tools

For those of you who do not use Visual Studio, and even those of you who do, I want to make you aware of another tool that you can use to create or scaffold new items into a project. So any time in Visual Studio when I'm right‑clicking on something and I want to add a new item or I want to add a new scaffolded item, many of the options that you see here in the Visual Studio user interface are also available to create and scaffold from the command line. This is done using a tool, which is the dotnet aspnet‑codegenerator. In the REAME file of the OdeToFood GitHub repository, I have placed some instructions on how you can install this tool. I just want to run it here to show you that you can use this tool to run a generator. There's several generators available, for areas, for controllers, for identity. The one we're using is the one for Razor Pages. So if I say aspnet‑codegenerator, I want to create a Razor Page, and I'm going to pass in ‑h for a help flag. The help will tell me the two arguments that I have to pass along are, one, the name of the Razor Page that I want to create, and two, the template to use to create this Razor Page. So later in the course when we have entities and we have data access classes, I'd be able to use this tool to say generate me a page that allow someone to edit a restaurant and automatically save the restaurant when all the validation checks pass and the user has clicked the Save button. For this scenario, we're just starting off with an empty Razor Page, so I would use the Empty template. And then there's a number of options that I can pass along. For our purposes, there's only two options we need to pass along. One is useDefaultLayout. That basically means I want to create a page that looks like the About and Contact pages that are in this project. They don't emit HTML head tags and body tags. They're just going to have content that plugs into the existing layout page. And the other option I can use is ‑outDir to place my new Razor Page in the appropriate location. So in the previous clip, I added a Razor Page using Visual Studio. If I wanted to generate that same page using this command line tool, I would say generate a page named List using the Empty template, use the default layout, and place this into the Pages\Restaurants directory. So after I execute this command, I would then have my list.cshtml Razor Page and my list.cshtml.cs page model.

Injecting and Using Configuration

Before the end of this module, we want our restaurant list page to display a list of restaurants. But before we get there, I want to take a moment to explore the relationship between this Razor Page and the Razor Page's PageModel because typically it is the PageModel that will perform data access and do all the hard work to put together the data that the Razor Page will display. Here's a very simple example. Let's give this class a public property. I'll make it of type string, and I'll call it Message. And then inside of the OnGet method, which is what responds to an HTTP get request, I'm going to simulate some data access by saying Message =, and let's just provide the message. Hello, World! So this is a simple hello world example. I'm going to save the PageModel class and come over to the page itself, and now any information that that PageModel exposes I have access to through an @Model property, so @Model with an uppercase M. The @ sign, by the way, allows me to transition from writing HTML to writing a C# expression or a C# code block. Again, we'll talk about that more later in the course. But when I say @Model, the object I'm looking at is an object defined by that class in the List.cshtml.cs file, so I should be able to find Message here in the IntelliSense. And if I save my Razor Page at this point and do a refresh, I should be able to see the message on the screen that was put together by my PageModel, which is Hello, World! Let's take this one step further. One easy data source that I can access right now is my application configuration. One of these sources of configuration for this application is apppsettings.json. As we move through the course, we'll find other configuration sources that we can use. But anything I place into this file is something I can access at runtime. So, for example, if I add a Message property to this JSON file and I say that Message is equal to Hello from appsettings, let's say I want to display this message in my Razor Page. Well, the whole idea in the relationship between a Razor Page and it's PageModel is that the Razor Page is strictly about display. All I want to do inside of the cshtml here is write simple C# expressions to place stuff into my markup. Therefore, it's going to be the PageModel's responsibility to fetch this message from somewhere. So something I can do in ASP.NET Core is give my PageModel a constructor, and my constructor can take a parameter of type IConfiguration. In order for this to work, I do need to bring in the namespace Microsoft.Extensions.Configuration. I can do that in Visual Studio by pressing Ctrl+period, and Visual Studio will detect that, and I just press Enter to add that using statement automatically. So let's call this config. And now I can also press Ctrl+period in Visual Studio to say please save IConfiguration off into a private field. So, again, I do that by pressing Ctrl+period and selecting an option from the menu. So I now have access to configuration throughout the rest of the PageModel, and here inside of OnGet, instead of saying Message = Hello, World, I can access the application configuration, and I can ask for values from appsettings.json by going to the configuration and indexing in like I would into a dictionary. So I can say, please give me a value for Message. And now if I save everything and refresh, we should be seeing the value that is stored inside appsettings.json, which we do, and we can see a very basic pattern that's going to repeat itself throughout the course. I'm going to use the PageModel to inject services that will give me access to the data that I need and then use those services, like the configuration service, to fetch data and add that to properties that I will then bind to inside of my Razor Page. Next, I want to take that pattern and apply it to Restaurants, but we're going to have to build out some abstractions for our restaurants first.

Creating an Entity

Now that we have a page to show restaurants, we're going to need to define what a restaurant looks like. Now, in a simple project, I would simply add a new folder to the OdeToFood project, perhaps name that folder entities or models or core or domain and define a Restaurant class inside of that folder. But I want to show you how to work with your models or your entities in a separate project. This can be useful particularly in larger applications where you need to reuse this code. So I'm going to right‑click on the Solution, and I want to add a new project, and I want to make this a .NET Core class library. Again, this is something you can do from the command line with dotnet new. Well, let's call this project OdeToFood.Core. It's going to represent those classes and types and processing algorithms that are the core of the business or the core of the application. And what is this application primarily concerned about? It's primarily concerned about restaurant, so I went to delete Class1.cs that is in the project and instead right‑click and say that I want to add a class, and let's call this class Restaurant.cs. Or rather, that's the file name with the Restaurant class inside. This will be a public class, and we just need to define the properties that we need to store or the information that we want to store about a restaurant. So let's define a property to store the name of a restaurant. Let's also just generically define a Location. Of course, you could make that an address that includes a street and a postal code and a city and a country. But with everything I'm going to show you, you can extrapolate out from Location to that more complex data structure. I also want to add some sort of identifier field because ultimately we're going to store these restaurants inside of a database, and every restaurant will have an ID that is an integer. And then, just to make things interesting, let's store information about the type of cuisine that a restaurant serves. So I will create a new enum named CuisineType. The default type will be None, and then let me just add a few additional cuisine types. So just going by the most popular cuisines that are served here in America, there's Mexican, there's Italian, and there's Indian. Now, typically. I keep every type in a separate file. I can do that in Visual Studio. If I put the cursor on CuisineType, press Ctrl+period, there's the refactoring option to move this to CuisineType.cs, which is good. And now I just want to add a property to my Restaurant of type CuisineType, and let's just call it Cuisine. I'm going to press Ctrl+K, Ctrl+D to do some formatting. That will also remove the unnecessary usings that I have, and this is the restaurant data we're going to work with.

Building a Data Access Service

Now that we have a Restaurant type, let's create something that we can use for data access that will store and retrieve restaurant objects. Before we get into the complexities of working with a real database though, I want to set up something very simple that will only work in test and development, but implemented in a way that will allow me to easily swap in a real data access class using a SQL database in the future. And to do this, I'm going to add another project to the solution, also a class library, and we're going to keep our data access classes separate from our entities and from the UI or the web application part. We will call this OdeToFood.Data. Inside of this, again, I'll delete the Class1.cs, and what I want to do is add a new item, and in this case, what I want to do is add an interface definition. So what we're going to do is start with some in‑memory data that will only work during development, but we'll hide our data access behind, let's call it RestaurantData. That will be the name of our abstraction. I'm going to make this a public interface, and we're going to start with a single operation. I want an operation that can return an IEnumerable of Restaurant. And in order to be able to access Restaurant, I'm going to need to add a reference to OdeToFood.Core. I'm going to use Ctrl+period in Visual Studio and select Add reference from the menu over here on the left. For those of you not using Visual Studio, I just want to show you what that will do. If I open up the OdeToFood.Data.csproj file that represents this project, this file will keep track of all my NuGet package references, as well as all my project references. So you can see that we have now referenced using a relative path an OdeToFood.Core.csproj file. That will give us access to the types defined inside of OdeToFood.Core from OdeToFood.Data. Later, we're also going to have to add references from OdeToFood, the web application, to these other class libraries that we've been creating. But for right now, I'm going to close this, discard my changes, and continue programming my interface. So I want a method that will GetAll restaurant. And for right now, I'm going to implement that using a class that I will call InMemoryRestaurantData. This will implement IRestaurantData, and I will use Visual Studio to quickly add the methods for me. So Ctrl+period, and select Implement interface. That will add the GetAll method for me. And now, once again, I'm going to do something that is only really going to work inside of a development environment. It's just something I'm going to do to give us data to focus on building other parts of the application until we come back and start working with SQL Server. What I'm going to do is add some data just using an in‑memory list. So let's give this class a constructor. Let me also provide some more room on the screen here. And let's say behind the scenes we're going to have a List of Restaurant, and that will be a variable named restaurants. And inside of the constructor, I can say restaurants is a new List of Restaurant, and let's go ahead and populate our list with some data. So let's create a new Restaurant. I'm going to make up identifiers here, so with an Id of 1, the Name, the Scott's Pizza, the Location as Maryland, and the Cuisine equals a CuisineType of Italian. And now I'm just going to take a moment to add a couple more restaurants. You can do this on your own and make up any data that you want. We'll come right back when I'm finished. Now that I have some sample restaurants in place, let me implement GetAll using a simple LINQ query. So I could say from r, which is a restaurant, in the restaurant collection that I have. Let's do some simple ordering, so I could say orderby r.Name. We'll have them in ascending order. And then we will select every restaurant. With that, I'm going to save my data access file. We're one step closer to putting data on the screen. There's just a few more chores we have to take care of first.

Registering a Data Service

On the restaurant list Razor Page, in order to show the restaurant data, we're going to need access to the InMemoryRestaurantData class that we just created. However, I don't want to write code inside of the Razor Page that instantiates this class directly. Instead, I want to program to an interface, which is IRestaurantData. This is going to allow me to swap out the data source later in this course, and we'll move over to using SQL Server seamlessly. ASP.NET Core is going to help me out here because ASP.NET Core follows the dependency inversion principle. All I need to do is tell the framework that whenever a component like a Razor Page needs something that implements IRestaurantData, it should provide that component with InMemoryRestaurantData, specifically. To tell the framework about this, I need to go into the Startup.cs file. So this is a bit of code that we'll cover later in the course that executes when the web application is first starting. And one of the methods that is in here is the method ConfigureServices. This is a method that ASP.NET Core will invoke to say tell me about all these services and components that you need. Tell me about your IRestaurantData and what should implement that interface. So it's inside of here where I want to walk up to this service collection, and I want to say that I want to add a singleton. You might be familiar with a singleton design pattern. It means I want a single instance of a specific service for the entire application. This is never something that you'd really want to do with InMemoryRestaurantData because behind the scenes InMemoryRestaurantData relied on a single List of Restaurant. And if we were to use a singleton instance of this class in production or with real users, we're guaranteed to have data corruption or an exception. Because a List is not thread safe, it's not going to be able to process multiple requests. So what I'm doing right now is just for development, but I'm going to say add a singleton instance whenever someone needs IRestaurantData. Then please provide the following component. But before I can do that, I'm going to need to add a reference from OdeToFood to OdeToFood.Data. So I've now established a project reference from the web app to the Data class library. And now I should also be able to say please provide an InMemoryRestaurantData instance. I'm going to do this as a singleton so that later when we start making data changes, everyone's going to see the same set of restaurants. But like I say, you only want to do this for development and test. And later in the course, we're going to swap out the implementation to work against a real database. But for right now, with the service registered, over in My Razor Page, which was Pages, Restaurants, List.cshtml, I want to go to the PageModel. And now, in addition to asking for IConfiguration in the constructor, since ASP.NET Core also knows about IRestaurantData, I should be able to ask for an instance of something that implements IRestaurantData. I do need to bring in the namespace OdeToFood.Data. Well, let's call this restaurantData. I'm going to use Ctrl+period to create and initialize a field and save that value off. My Razor Page now has access to a service or a component that knows how to fetch the restaurant, so the next step will be to fetch the restaurants and put them on the screen.

Building a Page Model

Remember that the goal for a PageModel class, like the ListModel that we're working with, is to respond to an HTTP request, like a get request to /restaurants/list that will invoke this OnGet method. Later in the course, we'll also see how to handle post messages with OnPost. But in this case, we're going to have a get request that's essentially saying give me a list of restaurants. So it's the responsibility of this PageModel class to use the services that are provided to put together the information that we need to render in the view. And the information that the view is going to consume will be the information that I place into public properties on this PageModel. Because from the view, I can reach properties like Message, and I can use them inside of markup and C# expressions to display that value. So all we really need to do with Restaurants is another property. Let's make it an IEnumerable of Restaurant. In order for this to work, I need to add the namespace OdeToFood.Core, and I'm going to call this property Restaurant. Fetching the restaurants is an easy operation. I can say that Restaurants equals what happens when I go to restaurantData and ask it to GetAll. At this point, my PageModel is providing all the information that the page needs. It has a collection of restaurants. There's public string Message. Now we just need to do some work in the Razor Page itself to display all this information.

Displaying a Table of Restaurants

Inside of our Razor Page, let's take the message that we're currently displaying, place that into a div, and I'm going to leave that at the bottom of the page. What I want to do is take the restaurants that I know about in my system and display them in a table. Later in the course, we'll change this around. But for right now, it would be nice to have a simple table, perhaps even style this with the Bootstrap table style, and for each restaurant display a table row that has the restaurant information inside. And the key phrase that I use there is for each restaurant. Inside of the cshtml file for a Razor Page, I can write markup, but I can also switch over and use the full power of the C# programming language. So if I have a collection, like a collection of restaurants, and I want to loop through those restaurants to create a table row for each restaurant, then I can write a foreach statement. All I need to do is use the @ sign to transition over into C# code, and I can say foreach restaurant in Model.Restaurants. Remember, Model with an uppercase M is what I can use to access my PageModel. And once I use Model, I can reach any property that I have exposed for my PageModel. So, for each restaurant, I want to create a table row, and this is where in a Razor Page I can just seamlessly transition between C# code and markup. So, for each restaurant, let's write out of table row. Now I'm ready to write out table cells that contain restaurant information. So inside of here, let me switch back into C# mode, and I'm going to use a simple expression with this local variable restaurant and just say let's display the restaurant name at this location. Let's have another cell that displays the restaurant location and one last table cell that will include the restaurant cuisine. Now you can make this table a lot fancier. You can add, for example, table headings, and later in this course we're going to look at a different approach for displaying this data. But for right now, I want to save all my files. I want to come back to the browser and do a refresh, because what I'm hoping to see are the restaurants that I made available from the InMemoryRestaurantData source, which I can now see on my page. So this data is all in memory, but later in the course we're going to be working with a real database, and I'm not going to need to change anything inside of my Razor page. It's just going to rely on that IRestaurantData abstraction.

Summary

In this first module, we've covered quite a bit of ground to display a list of restaurants. But the key part we've learned is how to use a Razor Page to accept an HTTP request and respond to the request by putting together data for the page to display. There's a nice separation of concerns in Razor Pages. All of your heavy lifting takes place in C# code in the PageModel, while the cshtml file contains just simple expressions and HTML markup. In the next module, we're going to continue building this application out by adding a search form and the ability to drill into the details for a restaurant.

Working with Models and Model Binding

Introduction

Hi, this is Scott Allen, and in this module, we are going to add search and movie‑detail features to our application. Along the way, we are going to learn about how to navigate between Razor Pages, and we'll get our first glimpse of an ASP.NET Core feature known as model binding. Model binding helps us take input from a user, like the input in an HTML form. Let's start with a refresher on HTML forms.

Working with HTML Forms

An HTML form allows us to receive information from the user, like the form here, which has only a single input for the user to enter their first name. In markup, we could make this UI by starting with a form tag and then including the Save button. This could be an HTML button element or an HTML input element, and in a simple form, we'll have just one of these with a type of submit. The type of submit tells the browser when the user clicks the send information back to the server. And what information does the browser send? Well, it will send along all of the inputs from inside the form, including any drop‑downs, any radio boxes, check boxes, and so on. In this form, we have just a single text input with the name fname, short for first name. That name is important because it gives us a name to associate with the value that the user types. And now the question is, where will the browser send this data? Well, if we don't specify any additional information in the button or the form, then the browser will send an HTTP get request back to the same URL that served this form. So if the user is viewing the page /account/edit name, submitting this form will send a get request back to /account/edit name. One way to change this destination is to add an action attribute to the form. So this form would send a get request to /update. And it's important to remember that the default is to make a get request, and we can change the type of the request using the method attribute. So now we're explicitly telling the browser what method to use. Now we never want to use a get request to edit any data, ever. A get request should never modify data back on the server. When the browser sends a get request, it has to place the input key and value pairs into the query string, like so, query string fname=Scott. So I've type Scott into that input box. The nice thing about this is that the user can bookmark this URL and return to this page with that parameter at any time. But you'd never want to bookmark a page that would change your name when you click on that link, just like you'd never want a bookmark that would submit an order with your credit card to buy airline tickets. A form like this that issues an HTTP get request is good for a search operation though. A search form is taking user input to search or read information. It's not trying to update my profile. So an HTTP get is a good approach to use when you're searching and reading. What we really want for an update scenario like this, when I want to update my first name, we want a post operation. A post will send data back to the server with key‑value pairs in the body of the message, not in the query string. We're going to take a closer look at form posts when we begin to modify data in this course. For now, let's work on a search form.

Building a Search Form

In the first half of this module, we want to work on giving the user the ability to search our restaurant. So on the Restaurants list page, I want to have a search form where the user can enter some text, press a button, and then we only want to display restaurants where the name of the restaurant matches, or at least partially matches, the text that the user has given us. And to do that, we are going to need a form. So inside of my List page, I'm going to use a Visual Studio Code snippet to add a form. This automatically adds action and method attributes. But the method attribute here is a post. I want to change this to a get. A search is a read operation, and the get method is ideal for a read operation. Not to mention this is going to place the search term into the URL in the query string, so the user will be able to bookmark search results. I'm also going to remove the action by leaving the action off. We will return to the URL that rendered this form, which is /restaurants/list. So this list Razor Page is going to be responsible for implementing the search feature. Now, inside of the form, I'm going to do some formatting, so I'm going to add a div with a Bootstrap 3 class of form‑group. In the GitHub repository for this course , I'll give some examples of what this should look like in Bootstrap 4. But inside of that div, I'm going to create another div. This is going to be an input‑group. And now I can have my input. The type could be type search or type text. I'm going to make it type search. The class, this will be a Bootstrap form‑control, and we'll start the initial value off as the empty string. We also need to give this input a name, but I'm going to defer that for just a little bit. We'll come back to it. Next, I want to provide a button, a button with the type equal to submit that tells the browser to submit this form. But again, to make things look just a little bit better, I will place this into a span that is an input‑group button. So this button should appear just to the right of the input. So now we will have a button. It will be a default‑looking button. And I'm going to place an icon here using Bootstrap 3 hlyphicons. And the glyphicon that I want is the search glyphicon. Let me save this Razor Page. We'll come back to the browser and refresh. I should now have a form or a UI that I can work with and where I can enter a search term and press the magnifying glass to search.

Finding Restaurants by Name

Before we can implement this restaurant search form our a Razor Page, we're going to need a data service that supports searching by name. So let's swing over to the OdeToFood.Data project. I want to open up IRestaurantData. And there's various different approaches that we could use to provide this search by name functionality. I even have an entire course dedicated to this type of topic. But what I'm going to do is take a very simple approach. In the application, let's assume instead of calling GetAll, the application, when it wants to list restaurants, is always going to say GetRestaurantsByName. So we'll always be getting the restaurants by name and passing in a string parameter that represents the name of the restaurant or the partial name that we went to match. And the way we can implement this method is if you pass an empty name or a null name, we'll return all the restaurants in any case. But I now need to implement this method, so let me copy the method name, come down into the InMemoryRestaurantData service, and paste this in as the name to replace GetAll. And yes, this is going to take a name parameter. We could even provide a default value here so that the parameter becomes optional. And to implement this method, I just now need to add a where operator. So let's first check if the parameter is null or empty. So where string.IsNullOrEmpty, this name parameter or someone has passed in a name, so we need to make sure that r, which is the restaurant, r.Name, starts with this value. We could do an exact match and say r.Name = name, but let's do something a little more fun and use StartWith. With that, I can save the data service. Let's close that file. We're now ready to go into the PageModel and start talking about how we're going to implement OnGet with this search functionality.

Binding to a Query String

As we implement the search functionality, we're going to learn about a very important topic in ASP.NET Core, which is the topic of model binding. Model binding is how ASP.NET Core helps you out by finding things and information that you might need inside of a request. So, for example, there's two different ways I can reach this OnGet method, and I'm talking about in general. A general way to reach this OnGet method is to arrive there as the result of a request that is not doing a search. For example, if I just refresh /Restaurants/List, I'm sending off a GET request without a search term, and I just want to see all the restaurants. Same thing would happen if I came into the menu and clicked on Restaurants. In that case, I don't need model binding because I don't need any information about the request or any information from the user. But what happens when I type into the search field? Now I need to know that value so I can invoke GetRestaurantsByName and pass in the appropriate name. This is where model binding comes into play. The first step in model binding is to give your inputs a name. So we did not give our search input a name. I'm going to give it a name now, and let's just call it searchTerm. And now I know inside of my OnGet method what I'm looking for is something called searchTerm. One way to find that value would be to go to the HttpContext property. So inside of every Razor Page, you'll have a context property, and it represents information about this HTTP transaction, both the request and the response. I can look, for example, at the request. And if I want to dig into HTTP headers to find some sort of access token, I can do that. I can also look at the path, and I can look at the query string. So if I know the search term is going to be in the query string, I can look for it there. But any time you're using the HttpContext to find something in a request, particularly if it's a form input or a query string parameter or an HTTP header, there's a better way to do that, and that's model binding. All I need to do is tell the ASP.NET Core framework that I need an input that is named searchTerm. There's a couple different ways to do that. One is to simply add a parameter to my OnGet method named a searchTerm. And now what ASP.NET Core will do when it invokes this method is go out through the requests and try to find something that is a search term. It'll look in posted form values, it will look in the query string, it'll look in HTTP headers, and you can even implement your own custom model binder. Just remember the goal of model binding is to move information from the request into an input model. But this name does have to match. So searchTerm here and searchTerm is the name of my input. Another way to do this is to use searchTerm as a property, but we'll come back and talk about that later. But right now, assuming I have a searchTerm, I'm going to pass that into GetRestaurantsByName. And I do just want to point out if we arrive at this method without a searchTerm, so let's say the user has just selected Restaurants from the menu, with the ASP.NET Core model binder will do if it does not find a search term is simply pass a null reference for the string value. That works with reference types. If for some reason this was an integer parameter, which is a value type, then ASP.NET Core would have to throw an exception. It would say I did not find a searchTerm, and since I cannot pass null, I'm going to throw an exception. But with strings, that's kind of like saying searchTerm is optional. And to test this, I'm going to save all my files and come out to the browser and do a hard refresh on this page, /Restaurants/List. What I'm hoping to see, and I do, is that I still have a list of all three restaurants. But what happens if I search for the string Sc, and now I have just Scott's Pizza? And if I go out and search for just an L, I see just my one Mexican restaurant. And where do these values appear? Where are they passed to my application? They're past in the query string, so /Restaurants/List query string, searchTerm=L. Let's try searchTerm=ss just by typing directly in the browser. Scott's Pizza appears there. And if I remove the searchTerm entirely, we'll go back to the list of three restaurants. But now let's go back to doing a search for Sc, and you'll notice something peculiar about this form, which is that the value Sc does not appear in the input. It might be nice from a user's perspective to keep my searchTerm in this input, so I know what I've searched for, and I could even come up here and modify it. Currently, that's not happening because when we render the input, we're always setting the value equal to an empty string. In order to solve this problem, not only do we need to know the searchTerm inside of the OnGet method, we need to know the searchTerm here that the user entered. We're also going to need to know the searchTerm when we render this input so we can specify the correct value. Let's see how to do that in the next clip.

Using Model Binding and Tag Helpers

Inside of my code base, searchTerm is what I would call an input model. I'm using searchTerm as a parameter during OnGet, and I expect to receive that parameter from the user. And that's different than the two properties I have here, Message and Restaurants. I would consider these output models. They're not here to receive information. They're here for me to populate with information so that when the page renders, I can bind to this output model and display information. But now I'm in a situation where I want to use searchTerm not only as an input model, but I also need an output model for searchTerm so I can populate value and show the user what searchTerm is in effect for a particular request. So one way to do that would be to add another property to my page model, a public property of type string, and let's call the SearchTerm with an uppercase S and T. And one way to populate this property would be to say that SearchTerm = lowercase searchTerm, the parameter, inside of my OnGet method. But there's an easier way to do this. Any time I have a property on my page model like the SearchTerm, it can function is an output model or an input model or both. So what I want to do is use searchTerm the property as both an input and output model. And the way I can do that is to add a special attribute to this property called BindProperty. BindProperty tells the ASP.NET Core Framework when you're instantiating this class and you're getting ready to execute a method on this class to process an HTTP request, this particular property should receive information from the request. So just like ASP.NET Core went looking for something called SearchTerm when I had that parameter name on the OnGet method, it is once again going to go looking for something called SearchTerm to populate this before invoking OnGet. This is also model binding. And so now down here inside of my OnGet method, I'll be able to say use that particular searchTerm. Now there's just one catch. By default, ASP.NET Core is only going to bind input properties during an HTTP POST operation. We're not doing a POST. We're doing a GET. Later in the course, we'll be doing a POST. But there is a flag I can use to control that behavior. So I can say that I want to bind this property, and, yes, I want that to support a GET request. So now when the user comes in and searches for something like a capital S, that request should populate the SearchTerm. I'll be using that SearchTerm to find restaurants. Now I just need to set the value of my input to that SearchTerm value, and the user will be able to see what they just searched for. There's an easy way to do that. There's a special tag helper that works with model binding in ASP.NET Core. It also works with input tags and label tags. And that is the asp‑for helper. So if you haven't noticed, all of the tag helpers that Microsoft provides out of the box start with asp‑ or at least most of them do. We've already seen asp‑page. That tag helper changes the href on an anchor tag to point to a specific page in the system. Asp‑for is a way of saying this input is for the following property that you'll find on the model. So inside of asp‑for, I write an expression that assumes I'm already working with the page model. So this can be a little bit confusing. But down here, when I need to write an expression to loop through the restaurant, I have to get to my Model property that gives me my PageModel instance, so literally an instance of this class. But when I'm using this tag helper, asp‑for, the Model part is already assumed here. So I don't have to write Model dot. I'm already working against an instance of my PageModel, so I can just say SearchTerm. That's the property I want on my PageModel. What asp‑for will do on the server with an input is not only set the name of this input so that it works with model binding and will populate SearchTerm, it will also set the value of this input to match the value in my output model property, which is SearchTerm. So you can almost think of asp‑for doing two‑way data binding. Asp‑for sets the name so input model binding works and one of my properties will receive this value. But also when we output, which is when we render this HTML from the server, the asp‑for helper will set the value for output. And so now what I should see if I save all my files, and let's just renavigate to Restaurants and look at the full list. There's my three restaurants, and I want to search for Ci, not continuous integration, but Cinnamon Club. And now I can see the input is populated and still has the Ci there, so I could change this if I wanted, perhaps making the search more specific. And I can also now cancel things out, press the Enter key, and I'm back to looking at a full collection of restaurants. So this is the magic of ASP.NET Core model binding. We're going to use a lot more model binding in this course, especially when we start to edit restaurants because we're going to need the user to enter all the information about a restaurant, and we'll see how that works with HTTP POST operations. But for now, let's turn our attention to passing information between pages. What I want to be able to do is click on a restaurant and go to more details about that restaurant. But that means some other page, some details page, will need to know what restaurant I want details about.

Building a Detail Page

The goal now is to provide a link for each restaurant in this list of restaurants that will take the user to the details for a restaurant. Now we don't store a lot of information about each restaurant, but you'll be able to extrapolate this out to have a detail page that will show everything that you know about a restaurant. You can even add your own custom properties and attributes if you want to this project. I'm going to stick with something very simple because what we really want to focus on is how does the list page communicate with this detail page because if I click on a link for Scott's Pizza, I somehow need to tell the browser to go over to this other URL and load the specific restaurant, Scott's Pizza. As a first step, let's just add our detail page. So I want to right‑click the page's Restaurants folder, say that I want to add a Razor Page. I'm going to pick the simple Razor Page template and click Add. Yes, I do want a PageModel class. I do want to use the default layout. And let's just call the page Detail. once Visual Studio has finished scaffolding this, I can think a little bit about my output model. Well, since I want to show the details for a specific restaurant, it would make sense to have a public property of type Restaurant, and we could name this property Restaurant. All I need to do here to make this work and compile is bring in the namespace OdeToFood.Core. And now before we get hung up on all the details of how to find out which restaurant we're supposed to load, let's go ahead and build out a simple UI. So first of all, instead of just showing a header that says Detail, since we're going to show the details for a specific restaurant, let's show the restaurant name here. So I need to go to my PageModel class. From there, I'll find the Restaurant property, and then I can display the name. And since we want to show everything that we know about a restaurant, let's also display the Model.Restaurant.Location. And I also want to show Model.Restaurant.Cuisine. There's one additional bit of information I'd like to show, which is let's show the ID of the restaurant that we have, so Model.Restaurant.Id, and I'll just preface that with the text Id. And so we're building a very simple display. You can make this a lot fancier, but I will add one more bit of UI. When I'm on the detail page, presumably I came here because I was on the list of restaurants. It might be nice to provide a navigation button that will take a user back to the list of all restaurants, So let's have an anchor tag that points back to the list page. Remember, I can do that with the tag helper, asp‑page. That tag helper will set the href here. So I want it to point to a Razor Page that's in the same folder with the name of List. And we can say All Restaurants will be the text here. If I want to make this even a little bit fancier, this is optional, I could give this some bootstrap styling to look like a default button. And now I'd like to take a quick peek at what this page would look like, but it's going to be difficult without a restaurant because my PageModel class does not set the restaurant. Therefore, the restaurant property will be null, and this expression Model.Restaurant.Name will throw a null reference exception. So let's go into the PageModel and at least set restaurant to a new empty restaurant for now. And this is somewhat typical of the way I like to work. Let's start with simple steps and verify that we're on the right path. We're going to come back in the next clip and try to get a restaurant ID and eventually load this from a database. But for right now, I just want to go to the browser and try to go to /Restaurants and then /Detail and see if I can get this page on the screen, which I can. So I have a restaurant with no name, no ID, no location, and no cuisine, but that's okay. We're going to fill in those details. I also have this button or this link that will go back to a list of all restaurants, and that's good. Now that we have a UI in place, let's see how we can pass along a restaurant ID from the List page the Detail page.

Linking to the Details

The detail page is going to need to know the ID of the restaurant to detail. So let me add a parameter to the OnGet method, which will be the restaurantId. I'm going to add that as a parameter here, not a property, because to me this is strictly an input model. Yes, we do display the ID on the page, but that ID is going to come from our restaurant model. And our restaurant ultimately is going to come from the database. To me, the restaurantId that I need to query the database, that's strictly an input, so I'm going to have a parameter here. We know from previously in this module that I can pass in a restaurantId using the query string. But I went to explore a couple different techniques for passing a restaurantId to this page. And so that we can see what the restaurantId s, before we start querying the data source, I'm going to populate our restaurant property, the Id property, with this incoming restaurantId. That will allow us to see what this value is. So with those changes in place, displaying the restaurantId, now it's time to take a look at how can I create a link or a button here that will take me from one of these restaurants to the details for that restaurant? Let's turn our attention then to the list. What I want to do is add another table cell off here to the right side of the table, and inside of the cell we're going to add some actions, as we move throughout the course. So for right now I'm going to add something that will allow me to get to the details. The next thing we'll add later is something that will allow me to edit a restaurant. But for right now, details. And feel free to add a simple link with text. I'm going to try to do something just a little bit fancier. Yes, I do want a link, so I'm going to use an anchor tag. But I'm going to use some Bootstrap classes to make this look like a larger than normal button, and all this button is going to be is another glyphicon. So this time I want to use a glyphicon, but I want to use the glyphicon known as the zoom‑in icon. And now, to reach the Detail page, all I need is an href. So one way to specify the href is to hard code this. So this would work. I could say, when the user clicks this, go to /Restaurants/Detail, that will get me to the Detail page, and then I can say include a query string where restaurantId equals and use a little C#expression here to say restaurant.Id. So this will certainly work, but this isn't the approach I want to use. I want to use an approach that uses tag helpers. So we already know that a tag helper like asp‑page, has the ability to set my href attribute so it points to the correct page. And in general, these tag helpers are the way to approach this problem because the tag helpers know about your software, they know what pages are in place, and they're going to know things about the structure of that page, like what inputs that page requires. And these tag helpers are going to be a little more flexible. So if you change something about the structure of your page, the tag helpers can automatically update the URL. You're going to see exactly what I mean by the end of the next clip. But for now, let's just see if we can get us to work. So I do know I want the href to point to the page, which is the Detail page. But it's not enough just a point to the Detail page, I also need to pass along this data, this piece of information, which is the restaurantId. And that's where another tag helper comes into play, the asp‑route tag helper. Now, this tag helper is a little bit different than the other tag helpers that I see in the IntelliSense window. Because these other tag helpers, the name that you see here like asp‑controller, that is the name that you have to use to invoke this tag helper. I have to say asp‑controller and provide some value inside of here. But asp‑route is a little bit different. It's a little more dynamic. What I can do is use another dash here and then include the name of the parameter that I want to pass along. So in other words, if I wanted to pass a search term parameter to some page, then I could say asp‑route‑searchTerm equals and then some value. And what asp‑route can do is figure out how it can pass searchTerm to the other page, knowing what it knows about that page. And you can pass as many of these asp‑route parameters as you need. So if I also needed an asp‑route‑name, I could add that here, too. We just need a single route and that single route is restaurantId, and the value will be the ID of the restaurant of the current row that we're on. Let me put this one to the next line so it's a little bit easier to see. And so now I have these two pieces of information here specified with tag helpers. I want to reach the Detail page and I want to pass along somehow a restaurantId that is equal to this value. Let me save all these files and let's see if this works. So I'm going to come over to the browser page, let me refresh the list. I'm hoping to still see my three restaurants, but now have a link on the right. And I have an HTTP 502.5 Process Failure. I'm going to leave this in the video because this error does pop up now and then, at least with this version of .NET Core and this version of Visual Studio. And I just want to give you the tip that typically this problem just goes away if I refresh again, and if it doesn't go away, the most I've had to do is just rebuild my solution, and in this case, it's not a problem with your code or with ASP.NET Core. This is a problem I only see during development, when I'm making changes and refreshing the browser a lot. So it's a glitch that happens in the coordination between IIS Express and ASP.NET Core and Visual Studio, where Visual Studio is trying to restart my application in the background when I make changes. Once I do a rebuild and I refresh the browser, everything's back to normal. And now I have my little zoom‑in icon on the right, and if I click on this, I can see yes, that went to /Restaurants/Detail query string RestaurantId=2. So my tag helpers, asp‑page and asp‑route, they decided the best way to pass a restaurantId to the Detail page is by placing the restaurantId in the query string. And I can ask for now any restaurant by any ID that I want. So when I ask for restaurantId 10, we'll be querying with a restaurantId of 10. But this is just one way that I can pass information from the List page to the Detail page. Another way, instead of using a query string, would be to place that ID into the path of the URL itself, so something like /Restaurants/Detail/19. Currently, that would result in a 404 error. Let's see how we can get this to work in the next clip.

Specifying Page Routes

In order for the URL/Restaurants/Detail/19 to work, I need to change the Detail page so it can respond to this path because the 19, the restaurantId, it's now in the URL path instead of being in the query string. And that is something I can do with the @page directive that is at the top of the Razor page. What I can do here is define a string parameter that describes to ASP.NET Core what the route or what the path should look like to reach this page. Now the path is always going to start with /Restaurants/Detail, unless I configure ASP.NET Core differently, or unless I start a path with a slash here and say no, actually, the way you're going to reach this page is to get to /food/place, but that's not what we want to do. What we want to do is introduce an additional segment for the URL path that's going to contain a parameter, which is the restaurantId. And for that, what I do is I specify the name of something inside of curly braces. So if I specify id, what I'm saying is that in order to reach this page, this Detail page, you have to use /Restaurants/Detail/ some identifier, that might be the number 12, or it could be a string like XYZ126. There's actually constraints that I can put on these parameters, like the int constraint that says this should be parsible into an integer. I can also make this parameter optional, so I could say with a question mark that you can reach this page by going to /Restaurants/Detail without passing an Id ,and you can also go to /Restaurants/Detail/3. But the important part to remember is that now the third segment of the URL, so whatever comes after /Restauranst/Detail, that third segment is what ASP.NET will consider to be a parameter named Id. Now if I were to leave this parameter name as just Id, then what my tag helpers over here would do is still put restaurant.Id into the query string because these tag helpers are smart enough to look at the page and say what parameters do you take in the route? Not the query string, not posted foreign values, not a header, this is something has to go into the route or the path of the URL. And so what I want is actually restaurantId, and this is not an optional parameter, I have to have this restaurantId to do my job, and yes, it does have to be an integer. So with that change in place, and this is the only change we've made to introduce this parameter restaurantId, let's come back to the browser and do a refresh on /Restaurants/Detail /19, where we used to get a 404 result, but now because ASP.NET Core understands to pass this value, parse it out of the URL, and pass it to my page as a restaurantId, we're now looking at the details for a restaurant with an Id of 19. And if I go back to the list of all restaurants, I should now be able to go to Scott's Pizza by going to /Restaurants/Detail/1. Now you must realize I didn't have to change anything in the List page to generate this new URL format. That's because the tag helpers, asp‑page and asp‑route, they generated this link with a restaurant Id in the path instead of the query string because those tag helpers, they know my pages, and they understand they're for amateurs. And again because this parameter is required, I cannot reach the Detail page without having some identifier in this place, and that identifier has to be a number, an integer. So we now have all the pieces in place where we can actually retrieve a restaurant from the data source and display it. Let's work on that next.

Fetching Restaurants by ID

What we need next is a data source that supports querying restaurants using an identifier. So let's go over to the OdeToFood.Data project. The first thing I'll do is modify our interface definition because this is what our Razor Page uses, the interface. I want to define a method that will return a restaurant. We can call it GetById. We pass in an id parameter. And with that in place, I'm going to copy that and bring it down to our InMemory implementation. What I want is a public method that returns a restaurant called GetById. And the implementation can be very simple. Let's go to restaurants. Let's use the link operator SingleOrDefault. So either give me the single restaurant that matches the following predicate, or give me a default value for restaurant, which is the null value. But let's look for a restaurant named r. Given r, I want one where r.Id equals this incoming id value. So we either return the single match for this, or we return a null. Back in the Detail page, I can now add a constructor. And just like we did with the list page, I can inject IRestaurantData. I need to bring in the name space, OdeToFood.Data. Let's give this the name restaurantData and use Ctrl+period to place this into a field on the page. And I would now have the ability inside of OnGet to say that the restaurant that my page is going to bind to should equal restaurantData.GetById, the following restaurant, and I'll pass in the restaurantId. Let's save this file. Come back to the browser. I'm going to refresh /Restaurants/Detail/1, and what I'll see are the details for Scott's Pizza, so we did retrieve this. And just to do another check, I do see Cinnamon Club, and everything here appears to be working. But what happens if we have a curious user who sees this number at the end of every URL and says, well, what happens if I look for something with a 10? What's happening here is our data layer is returning a null restaurant. It didn't find the restaurant in the data source. And we're passing that null reference into our Razor Page, which is writing these expressions, which dereference that pointer and throw an exception. What can we do about this situation? Let's look at that next.

Handling Bad Requests

We certainly don't want to show an end user this type of error page if they stumble upon a restaurant that doesn't exist. Perhaps they found some link on the Internet somewhere and followed it for /Restaurant/Detail/ some identifier that was deleted last year. What we'd rather do is to show them a page that says sorry, we couldn't find that restaurant. One way to do that would be to modify Detail.cshtml because I can write if statements inside of my page. So I could say something like if model is null, let's show this div that says sorry, we didn't find the restaurant, else show the restaurant. But writing if/else statements inside of a CSHTML page is quite messy. I would suggest you avoid that type of solution whenever possible. If I need to write some sort of if/else statement, I would rather do that in a CS file like inside of my page model itself. But what could we do inside of OnGet to prevent this type of screen that's going to show an error? Well first, let me tell you that, by default, if we don't return anything from OnGet, what ASP.NET Core will do is always render this CSHTML. However, we have control over that decision. What we can do from OnGet is have a return value, and we can specify the return type as IActionResult. So we can think of OnGet as an action that takes place inside of this page. What we're going to return from here is a result of this action. And there's many types of objects inside of ASP.NET Core that implement IActionResult. And inside of a Razor Page, there's many helper methods that will produce an IActionResult. For example, there is a helper method that is named Page. You can see here a Page method returns a PageResult, and a PageResult implements IActionResult. So if I just say return Page, we're going to have the exact same behavior that we had before, which is we're going to try to fetch a restaurant, and then we're going to return this action result, which tells ASP.NET Core please render this page or take the CSHTML, Detail.cshtml, and render that, in which case, when the user is looking for something that doesn't exist, we're going to have some sort of error. But I can also do something like this. I can say if the Restaurant = null, now I want to return a different type of result. I don't want to render the page. So what can I do? Well, I might want to redirect to somewhere else. And you can see there's a number of methods here that allow me to do a Redirect or a RedirectPermanent. So that's the difference between a 302 and a 301 status code. And there is also a RedirectToPage. So this is a way of saying let's not render Detail.cshtml. Instead, let's return a 302 status to the browser and tell it to go look for this other page. Let's tell it to go look for a page called NotFound. Currently, that page doesn't exist, but we'll implement it here in just a second. So if the restaurant is null, we're going to go to the NotFound page. Otherwise we can render Detail.cshtml. Let's implement NotFound. I want to Add a Razor Page. And this time what I can do is add a Razor Page that doesn't even have a PageModel class. We don't really need any logic associated with NotFound, at least the NotFound that I want to use. All we need really is some markup inside. So let me click Add. And once Visual Studio creates this, I'm going to come in and say Your restaurant was not found. And perhaps what we could do is include an anchor tag that points to the List of restaurant. With bootstrap, we could make this a btn. We'll make it a primary‑colored button, so it stands out a little bit, and include the text See All Restaurants. Let me save all files in the project and come out to the browser. What I'm going to do is refresh /Restaurants/Detail/10. And now, instead of seeing an ugly exception, I see /Restaurants/NotFound. I see the message that this restaurant is no longer available, and I can go back to my list, and I can still navigate to the details.

Summary

In this module, we implemented search and detail features. But more importantly, we learned about model binding from the query string and from URL parameters, or what ASP.NET Core would call route parameters. In the next module, we're going to build on this model binding knowledge and give our users the ability to edit restaurants and create new restaurants.

Editing Data with Razor Pages

Introduction

Hi. This is Scott, and in this module, we are going to give our users the ability to edit restaurant data and create new restaurants. We're going to need to build more forms in HTML and use more model binding. But we're also going to see some new topics, like model validation. Let's get started by adding a new page to the application that will allow users to edit an existing restaurant.

Creating the Restaurant Edit Page

In the previous module, we added a new Razor page to the application to detail a restaurant, and to reach that Detail page, we added a link here in the list of restaurants. I want to do something very similar to edit a restaurant. In fact, a lot of the early work on editing a restaurant is going to look just like the work we did to detail a restaurant because we need a link, we need to retrieve the restaurant, we need to put some restaurant information on the screen. So let's get started here in our List Razor page. And I'm going to copy and paste the anchor tag that we used to get to the Detail page and just change some of the code in here. Instead of going to the Detail page, this link is going to go to the Edit page. We'll create that in just a moment. And instead of the zoom‑in icon, I want the edit icon. Those are all the changes I need for the List page. Now let me add a new Razor page, which is going to be the Edit page, so I will name this Edit. I do want a PageModel class and a layout page, and click Add. As soon as Visual Studio is finished scaffolding this, I will want to add a constructor to this page so that we can inject the data service that we need. That is going to be the IRestaurantData data service. I'll need to bring in the namespace OdeToFood.Data. We will call this restaurantData, and we will save this off into a private field. I expect to use this data service to create a property that my view can bind to. This property will be of type Restaurant. And that's from OdeToFood.Core, and I'll give this property the name Restaurant. Now what I need to do is retrieve that restaurant inside of the onGet method. Just like with the Detail page, I will expect us to receive a restaurantId. And also just like the Detail page, I want that ID to be in the URL. So over in the cshtml file in the @page directive, I'm going to say add an additional parameter to the route, which we will call restaurantId. And, yes, once again, I can make sure that's an integer. So far everything that we're doing is sort of a review from what we did to build the Detail page. And now I can say that the Restaurant is what happens when I go to my restaurantData service, and I ask to get the restaurant by the ID, which is restaurantId. Now just like with the Detail page, we might have to handle the condition where the user ends up here after passing a restaurant ID that no longer exists. We'll know that's the case if Restaurant comes back from the data service as null. And if this is the case, what we want to do is redirect back to the NotFound page that we created in the last module. And in order to do that, I need to start returning results. So I'm going to change the return tight from void to IActionResult. If the restaurant is null, we will RedirectToPage, specifically the NotFound page. Otherwise, during OnGet, we'll return our cshtml file, which should present to the user a form that allows the user to edit a restaurant. In fact, we can come back here and we can say, instead of just Edit, I can say you're about to edit Model.Restaurant.Name. Let me save all files in the solution. Come back to the browser. I want to refresh, hopefully see our list of three restaurants with an additional icon, which we do have. And if I drill into Scott's Pizza, then I do get the /Restaurants/Edit/1. I can see the header says Editing Scott's Pizza. Now all we need is a form where I can change that name. So everything we've done so far is sort of like a review of what we've done in the previous modules. But now let's start drilling into the details of a form that will post data back to the server.

Building an Edit Form with Tag Helpers

Now let's build the form that we will use to edit the restaurant data. So I want a form with a method equal to post, because I do need the browser to send a post operation. We are editing data. That's not an operation that you want to do with an HTTP get request. And to send the post, I'll add a button of type submit. Let's also give this some Bootstrap styling so that it appears as a button with the primary color, and with the text saved, the user will know where to click to save the restaurant information. Now, I want the user to edit the restaurant name, the location, and cuisine, but I also want this form to include the ID of the restaurant as a form value, because I want the form to include all the information that I need to put together to update a restaurant. But I don't necessarily want the user to edit the ID, and that's why when I add an input for the restaurant ID, I'm going to make it a hidden type. And then just like we did in the last module, I can say that this input is asp‑for Restaurant.Id. And you might remember from the last module, asp‑for can do at least two jobs. One job is to set the name attribute of this input so that when the form is submitted, ASP.NET model binding can say, ah, I see this value is the value that should be placed into the Restaurant.Id property. Asp‑for can also set the value of this input, so if I'm editing the restaurant with an ID of 3, the value will be set to 3 for me. I don't have to set those attributes myself. Now I need the visible inputs. So let's creative a div with a class of form‑group. And for the first input, let's edit the restaurant name. I will need a label to tell the user this is for the restaurant name, and I can do that again using asp‑for, so this is a label for Restaurant.Name. What asp‑for can do in this case for me is set the for attribute to make sure that this points to the appropriate input, and now we can have the input, and the input is also a asp‑for Restaurant.Name. In this case, I don't have to set the type attribute for this input, that's something else asp‑for can do for me. It can look at the property that I'm binding against Restaurant.Name, see that that is a string property, and say, oh, you must want an input type = text so you can edit a string. We'll see later how we can influence the generation of that type attribute. And I'm also going to give this a little Bootstrap styling, so this is a form control. So this block will edit the Restaurant.Name. I'm going to copy that block and paste it, because I really want the same code, but now we're going to edit the restaurant location. So I just need to change Restaurant.Name to Restaurant.Location, and now I'll paste it once again. This time we want to edit in this block the Restaurant.Cuisine. However, I don't want the user to have an input, especially a text input where they type in the name of the cuisine, I want to make sure they're using a specific cuisine that is available in this enum. So ideally they will select from a list of options where those options are None, Mexican, Italian and Indian. So what I really want to use here is not an input, but an HTML select. That will build essentially a drop down list for me. And I can give this a class of form‑control, and I can still use asp‑for to say this is for Restaurant.Cuisine. But I also need to provide the options that are going to be available inside of this select. So when the user clicks on the select, what appears here? One way to do that is to hard‑code or hand‑code those options into the select. Another way to do that is to use another tag helper that I'll introduce you to, and that tag helper is asp‑items. So if I can bring up the IntelliSense here. So you can see that asp‑items takes an IEnumerable of SelectListItem, so a collection of SelectListItem. A SelectListItem is an object that describes one of the options in a select, so it has two significant properties. One property says, here's the text to display, and the other property says, here's the value to submit when that particular entry is chosen. So the SelectListItem is going to build my options for me, and ideally, I'll have one SelectListItem for None, so it will display the text, None, submit the value 0; show the text, Mexican, submit the value 1; Italian is 2; Indian is 3; and those are just the default values that are given because I didn't explicitly specify them. They're the values assigned by the C# compiler. So I need to build this collection of SelectListItem, and one way to do that is to use a property that is available inside of a Razor Page, which is the Html property. It's of type IHtmlHelper, and because of that, we call it the htmlHelper. Many of the methods that are available on the htmlHelper are methods that help you generate little snippets of HTML, or just general methods that you might find useful inside of a view, for example, GetEnumSelectList. What this can do is build a collection of SelectListItem based on an enum type. So using reflection, it can walk through my enum of CuisineType and see there's an option None, which is 0; and Mexican, which is 1; and so forth. So I could right here inside of the Razor Page say, let's get this EnumSelectList using my CuisineType, and I would have to bring in a using statement to make sure that CuisineType is available, but I'm not going to use this approach. As simple as that operation is, it's also an operation that I could do inside of my PageModel, and any time I have the ability to build something like the model that my cshtml file needs to use, any time I have the ability to do that inside of my PageModel instead of inside of the view itself, I want to do that inside of my PageModel, because quite often in the future that will make things easier to change. It also makes that particular operation easier to test. So, in other words, what I want is a property that I can use with asp‑items, so a property that is of IEnumerable of SelectListItem. That's exactly what asp‑items needs. All they need to do is bring in the namespace, which is Microsoft.AspNetCore.Mvc.Rendering, and I can call this, let's call this Cuisines, because it contains the list of cuisines that I need to bind again. So now I have two output properties, or two properties I combined against, the restaurant that we're editing and this additional model, Cuisines, which helps me build an HTML select. Now all I need to do is build this inside of my page, and just like the restaurant, I'll need to provide that information inside of OnGet. Now the obstacle you might encounter is that if you try to find that same htmlHelper here inside of the PageModel for a Razor Page, you won't find a property named Html. That's only available when I'm inside of the cshtml file. However, just like we did with the restaurantData, we can tell ASP.NET Core about other services and other components that we need to do the job. So I can ask ASP.NET Core to please provide me with an htmlHelper that I can use. And once ASP.NET gives me that helper, I can save it off so I can use actually Ctrl+period to say, please create and initialize a field, htmlHelper. Now it's saved off just like my restaurantData service. And inside of that OnGet method I can say that Cuisines will be equal to htmlHelper.GetEnumSelectList, and then just provide my enum, CuisineType, as a generic type parameter. Now I have a piece of code that if in the future I need to modify this or I need to change some of the text that is appearing, I can do that here in my PageModel instead of trying to place that into the view. And now, coming back to the view, I can say that asp‑items equals what happens when we go to Model.Cuisine. So, again, it's a little bit confusing, but asp‑for always assumes that you're working on something that is on your model, so you never have to say model dot to get to the PageModel, I can just say Restaurant.Cuisine. But asp‑items, it works against a data source that could come from anywhere, so I have to specify no, go to the model and then look at the Cuisines property. And with these changes in place, I should be able to save all files, let's come back to the browser and refresh, and I now have a form where I can edit the name, Scott's Pizza; the location, which is Maryland; and notice the cuisine. The option that is selected here is Italian, which just happens to be the cuisine for Scott's Pizza. So not only have I built a list of the available options, but the asp‑for tag helper makes sure that the initial option that is selected matches the value that is on my model. So with this form in place, let's prepare our application to make use of the form post that happens when I click on Save.

Model Binding an HTTP POST Operation

In order for the Save button to work, I'm going to need to add a few things to the application. First of all, inside of our Razor PageModel, I'm going to need a method that responds to an HTTP post, so OnGet responds to a get request and presents a form to the user. Our OnPost method will respond to a post and take the form information and try to save it into the data source. We're also going to need a data source that supports updating restaurants. So let's start there, with the IRestaurantData interface that the Razor page uses, let's add another method, it's going to be a method that returns a restaurant, let's just call it Update, and it's going to take the updatedRestaurant. Why does this return a restaurant? Because in the process of updating a restaurant, some data sources might change some of the properties or some of the values that are in that restaurant, so if you have some attribute that is generated from the database, like a concurrency value or a timestamp, returning a restaurant, perhaps even a new restaurant object, would propagate those values back to the caller. We won't need to do anything quite so sophisticated inside of our InMemory data source, but I'm also going to add one more method, a method that returns int, and let's call it Commit. Many data sources out there act as units of work. They collect all the changes and additions and deletions that you're making to a data source, and then reconcile those changes when you invoke a method like Commit, or SaveChanges, or CommitTransaction. This is certainly true of the entity framework. After I make a change to an entity, I can have the entity framework persist those changes in the database by calling SaveChanges on a DB context, we'll see how that works later. I want to expose that same sort of functionality through this interface by having a Commit method. Now again, with an InMemory data source, we could try to implement IRestaurantData in a way where commit actually mattered, but I'm just going to do the simplest possible thing that can work. Remember, the goal here is not to provide a real data source that works in memory, this is only for testing and development, and it's really only here to allow us to work for a bit until we can bring in a database and start using the entity framework. So, a method named Update, that will take a restaurant parameter, we can call it updatedRestaurant, and here's what we can do in the implementation, and let me just make sure the name is correct, which is Update, and first we're going to look inside of the list of restaurants that we have for this updated restaurant, and try to pull that out. So I'm going to say that restaurant = restaurants, and I'll use the SingleOrDefault operator to say, given a restaurant, r, give me a match where r.Id equals the updatedRestaurant's Id. Now, if a restaurant is not equal to null, that means we found a match inside of our list, and what we can do is simply copy information from the updated restaurant into the existing restaurant that is in the collection. So, for example, I could say restaurant.Name = updatedRestaurant.Name, and restaurant.Location = updatedRestaurant.Location, and finally, restaurant.Cuisine = updatedRestaurant.Cuisine. So I've copied over all the information that the user has given me, and for InMemory collection, that's really all the work that we need to do. So let's just return the restaurant object that we found with the new information inside. I also need to provide an implementation of Commit, since our InMemory data isn't really transactional, what I'll just do is return 0. So this method doesn't really mean anything until we start using a real data source; we will do that later in the course. Now what I should be able to do is come into my Razor page, and we want to implement an OnPost method, so, a method that can return IActionResult called OnPost, and what parameter do we want to take here? Well, we could take in a restaurantId, we could even make sure that this restaurantId matches the Id property of the restaurant that were binding against from the form, but I'm not going to do anything quite so fancy, I'm just going to assume that the Id of the restaurant that is in the form will be the Id of the restaurant that we want to save; and how do we know the Id of the restaurant that is in the form? Well, just like we did in the previous module, I want to change this restaurant property to be an input and an output. It's an output because given a restaurantId, I find this restaurant during the OnGet method and populate this property so that we can build the form to present to the user, and then when the user makes modifications and clicks the Save button, I expect the inputs and the select of the form to populate this restaurant, and in order to do that, I again need the BindProperty attribute. So we used that before, and we allowed it to happen during a get with supports get, in this case, we only want this to happen during a post operation, and so when the user clicks on the save button, this restaurant should be populated with information from the form. What that will allow me to do during OnPost is say, dear restaurantData, please update the following restaurant, and because update returns a restaurant, I could assign that return value to the restaurant property and inspect the restaurant to see what changes have taken place. But if I want to do the simplest thing that would possibly work, I would also say dear restaurantData, please now commit those changes that I asked you to make, and for right now let's just return the page and see what happens. So ,there's still several problems that we have to address in this OnPost method, but I just want to take things one step at a time here. Let's go back out to the list of all restaurants, I just want to start fresh from the browser. Let's go in and try to edit Scott's Pizza. Let me just say this is Scott's Pizza 2, we'll change the cuisine type, click Save, and it looks like the changes have been made, I can see now I'm editing Scott's Pizza 2, but there are some problems here that we need to address. First of all, cuisine is now empty. What happened to my cuisine? I can also completely blank out the name and click Save, and that operation goes through without an error, and if I come to the list of all restaurants, I can see I now have a restaurant with no name, which is bad. Let's see how to fix these issues in the next few clips.

Adding Validation Checks

Let's first address the issue where I lose my list of cuisines when I edit a restaurant. So if I change the cuisine here from None to Mexican, and click Save, I've completely lost my list of cuisines; yet, that change was sent to the data source. So my no name restaurant does have a cuisine type of Mexican, and the problem here is that I only populate this cuisine's property to bind against during an OnGet request, what happens during an OnPost request? Well cuisines has never populated, so over in the cshtml file, when I say asp‑items = Model.Cuisines, cuisines is going to be null. So the simple fix here is to have this particular piece of information generated on both a get and a post, and the reason I left that problem in here is just to prove that ASP.NET Core is stateless. There's nothing in the framework that's going to magically and automatically rebuild the options in my select. I need to take responsibility for building the select items, and I need to do that during OnPost if I'm going to present this form to the user again. Note that I don't necessarily need to do that for restaurantData, in fact, since we're not using this property, after I've called restaurantData to update, I could remove this assignment to the restaurantData, that restaurant information is still going to be available because it's the restaurant information that was sent back from the user. In other words, when I changed the cuisine here from Mexican back to None and click Save, why is a location populated? Well, it's populated because the restaurant.location property is populated; that happened due to the magic of model binding, and in fact, what we can find out is that the asp‑for tag helper that we're using to populate our inputs with values, can really rely on the raw data that the user has provided, so data that might not even fit into my model. I'll show you an example of that later in the course, but for right now, adding this line of code inside of OnPost should fix my list of cuisines. Now, what about the empty name problem? Well, during an OnPost operation, you need to perform some sort of model validation, you can never trust input from the user, so you need to make sure that the input the user is providing is valid. One way to do that, the hard way to do that would be to place if statements here. So I could say if the Restaurant.Name is not empty and it's not null, and if the location is not empty and not null, and if the length of the name is beneath a certain threshold, and only if all those statements are true; only then should you do the restaurantData.Update. But ASP.NET Core and .NET in general, provides an easier way to do common validations. You can have these common validation checks done for you by adding attributes to your model objects. So, for example, here on the restaurant, I want to make name a required field, no one should be able to enter a restaurant that has a null or empty name, so I'm going to add a required attribute. This is from the System.ComponentModel.DataAnnotation's namespace, I'm going to bring that in, and if you explore this namespace and look through the documentation for this namespace, you'll find a number of useful data annotations. So, for example, I can make name required. I can also say that the maximum length of the name should be 80 characters, and I can do these as separate attributes. Some people even like to combine these, so it looks something like Required comma, and StringLength 80. I could do the same thing for location. Perhaps I want location to be required and also to have a maximum length of, let's, say, 255 characters in this case, and there's also built‑in validations to make sure that numbers are within a certain range, to compare two passwords, and to make sure that a string fits in the right pattern for a regular expression. I will also point out for more complex validations, there's an interface that you can implement, IValidatableObject, this will allow you to write custom code on your model or on your entity to perform validation checks. Now ASP.NET Core whenever it does model binding, so when it's binding form inputs to bind properties like restaurant, it's going to look at the restaurant type and see these data annotations and automatically run checks to make sure these annotations pass. We'll see how to inspect the result of that validation in the next clip.

Using Model State and Showing Validation Errors

To take advantage of the data annotations that we just added to Restaurant, we need to use a data structure known as ModelState inside of ASP.NET Core. So any time that ASP.NET Core performs model binding, so it moves data into my model like a Restaurant or even into a simple integer parameter like restaurantId, the framework will keep track of everything that happened and record that information inside of this data structure called ModelState. I have access to ModelState through a property that is available on my Razor Page called ModelState, and I can walk up to this data structure and index into it if I want to find out specific information about, let's say, the model binding that occurred for the value with the name Location. I can check, for example, does this have any errors? So if Location is a property on the restaurant and it has a required attribute, and the user passes an empty string, that's going to raise an error that gets included in this errors collection. I can also look at AttemptedValues. So if the user tried to enter the string XYZ into a property of type int or float or some numerical value, the only way to see what the user tried to type is by looking at this AttemptedValue because it can store anything. I cannot look at my model property to see a possible value. But these are the types of low‑level operations that you typically don't need to worry about. Tag helpers like asp‑for use this information all the time. Typically, all we care about is walking up to ModelState and asking, is this valid? Is all the information that got passed in via model binding, is it all valid? Did it pass all the validation checks? And only if that is true, do I want to go to the data source and say I have valid data, let's save it into the data source. If the ModelState is not valid, what I need to do is present this edit form again and allow the user to fix any errors that have occurred. Let's try this out. I'm going to save everything, come back to my browser. And I do want to point out now that we're editing data and we're making changes to the application, at some point you're going to refresh the browser and all the original restaurant information is going to appear, and that's because we're using an in‑memory data source. The in‑memory data source, as I mentioned earlier, is only good for testing and development. When the application restarts, we lose all the changes that we've made. So we're looking at the original restaurant information again. Once we hook up to a database, that particular problem will go away. Let's now try to edit Scott's Pizza. And what I want to do is blank out the Name and blank out the Location. And now when I click Save, it doesn't look like anything happened. I don't know if the restaurant was saved or the restaurant was not saved. I can come back to the list of restaurants, and this looks good because it looks like I did not save invalid information, but I also didn't tell the user that there was a problem. So if the user tries to submit a blank name, I need to provide an error message, and that's where the tag helper asp‑validation‑for comes in to be helpful. So let's create a span with a Bootstrap class of text‑danger and use the tag helper asp‑validation‑for. What I can say is that I want validation messages that might be in ModelState for Restaurant.Name. And what asp‑validation‑for will do is look in the ModelState, see if there's any errors for Restaurant.Name, and if so, display the associated error message for that particular error. Now the data attributes that we have used like required and string length, they provide some default error messages that we will see on the screen now that we're using a validation span. But let me copy that span and also paste it here where we can display validation messages for Restaurant.Location. And finally, I will also place that here for our select. First, let me fix the formatting here so that I don't have this character on a separate line, and then I'll paste in my span, and then I just want to change this from Restaurant.Name to Restaurant.Cuisine. Let's save all the files, come back to the browser. What I'm going to do is try to submit a restaurant with an empty Name again, and I can see the Name field is required. Also, if I do this with Location empty, the Location field is required. So now the user knows when they are editing that they need to provide a name and location. But there's still one more thing that we can add to our edit page that will make it function a little bit better. I can see this if I make a change, click Save, and then I'm sitting on this page where I can see the changes that I've made. And if I try to refresh this, as some users will try to do, I have a warning to confirm my form resubmission. Let's talk about that next.

Following the POST-GET-REDIRECT Pattern

In Web development, it's generally a bad idea to leave the user on a page that is displaying the results of an HTTP POST operation. And that's because of post modifies data. In this scenario where I am editing restaurant data, maybe this isn't too bad, but imagine I was creating a restaurant or submitting an order with my credit card. If the user or the browser decides to refresh this page, that would resubmit the form by sending another HTTP POST that could create another restaurant, a second restaurant, an unintended restaurant, or charge my credit card twice by submitting a duplicate order. And even the browser knows that resubmitting the form may not be the right thing to do. So when I hit Refresh on this page at the end of the last clip, the browser popped up a warning dialogue, saying, are you sure you want to resubmit this form? Any action that you just took would be repeated, do you want to continue? And so generally in Web development, we want to avoid this warning dialogue. What we want to do instead is when the user clicks on Save, and if the post is successful, and I saved that data, I then want to redirect the user's browser over to another page, another resource, and have that page displayed as the result of an HTTP GET request. A GET request doesn't modify data at save, so I can refresh and even bookmark that page. So what could we do in this scenario where a user is editing a restaurant and they click Save? Where could we go that would be safer? Well one thing we could do is redirect the user over to the details for this particular restaurant. That would not only allow the user to see this restaurant and see the changes that they've made, they're in the application and they're saved in the system, but the details page is also something I can display with a GET request, and that is a safe request. So we just need to make a modification here inside of our page model, where if the ModelState is valid and we have saved and committed the changes to the restaurant, we don't want the logic to continue to fall through and rerender this edit page. Instead, we again want to redirect to another page. In this case, the page will be the detail page that we created earlier in the course. Now, remember this detail page needs information about the restaurant in order to find the restaurant that it needs to display, specifically the detail page, you might remember, needs a parameter named restaurant ID. So I'm going to copy this identifier, bring it back to my page model, and show you that the second parameter I can pass here to redirect to page after the page name is an object that contains route values. So what I need to do is have an object that I pass to redirect a page that will have a restaurant ID property and set that property to the ID of the restaurant. One easy way to do this with C# is to create a new anonymously typed object. So just the new keyword and then opening curly brace. I'm going to paste in restaurantId that I copied out, and say that restaurantId should equal Restaurant.Id. What ASP.NET Core can do is look at this object, see there's a RestaurantId property, it now knows it needs to pass this information along to the other page. It will also know that detail expects a restaurantId, so it can place that restaurantId into the path. Let's see how this works. I'm going to save all my files. Let's come back and go to the list of all restaurants and, again, because restaurants is an in memory collection, my restaurant collection has been reset. The application has been reset. But I can go in and edit Scott's Pizza. Let's change the cuisine to None. Let's change Scott's Pizza to Scott's Pizza 3, click Save, and I have now been redirected to /restaurants/detail/1. I can see the changes that I have made are in effect, and I'm now on a page that is safe to refresh because after the browser was redirected, it issued a get request here. I'm refreshing a get request. So this pattern that I just put in place is commonly called the Post/Redirect/Get pattern, PRG, and it's a simple pattern to follow whenever you have formed post operations in your application.

Building a Create Restaurant Page

Now that we have the ability to edit a restaurant, let's add a feature that will allow a user to create a new restaurant. What I'd like to add is a button here to the list of restaurant at the bottom that will allow the user to create a new restaurant from scratch. Now, in many projects, the create scenario and the edit scenario will be two different Razor Pages. There's edit, and then there's create. But what you'll find typically is a lot of your markup and some of your logic even is duplicated between edit and create. They're trying to do the same job just from a different starting point. What I'd like to show you is how you can use this Edit page to both create a new restaurant, as well as edit an existing restaurant. And we'll do it all from the Edit page. We will start, however, in the List page where I want to add a button for the user to click on. So let's create an anchor tag using the asp‑page directive to point to the Edit page. I will use the text, Add New, and let's add some Bootstrap styling. We can make this a button using the primary color. I'm going to save that file and come out to the browser and refresh already because I just want to show you that this button does appear and I can click on this button, but I stay on the list. Why is that? Because the Edit page demands and requires a restaurantId. That's useful when I'm editing an existing restaurant. But when we're creating a new restaurant, I don't have a restaurantId that I can pass along. So this is a scenario where I could make restaurantId an optional parameter. And I can do that by placing a question mark here at the end. So now ASP.NET Core knows that I can reach the Edit page by going to /Restaurants/edit/3, which is a restaurantId, or /Restaurants/edit with no additional parameter. And then, in order for this to work, I'm going to need to make some changes in my page model. So currently, I assume I always receive a restaurantId, and I'm going to go out and look up a restaurant by that ID. Now I'm going to make this nullable, in which case restaurantId might be null. So I only want to look up the restaurant if restaurantId has a value. Only then will I go to the database and say, please get this restaurant by ID. And now I have to look at the value property to pull the actual integer value out. Otherwise, what should I do? Well I don't need to look up the restaurant from the data source, so perhaps I could just initialize a new empty restaurant or even provide some defaults here for the user. For example, if I knew the user's location based on their profile, I could go ahead and specify a default value here for the location. Since that value will populate the form input for location, it might save the user some time, they won't have to type it in. But in our application, since I really don't know anything about the user, I'm going to leave that out. So now I should be able to reach the edit page without passing an ID, in which case will be editing a new empty restaurant. Let's try this out. I have saved all my files. I'm going to refresh the browser. Now I want to add a new restaurant, and I am editing a restaurant with no name as yet. Now, before this can work and we click Save, we're going to need to update some of our OnPost logic and our data source so that we can create and add new restaurants.

Adding Create to the Data Access Service

Let's first take care of our data source. I'm going to swing over to OdeToFood.Data, and open up IRestaurantData.cs. This is currently where we have both our interface and our in‑memory restaurant data implementation, we're going to change that in the next module, but I want a new method that will return a restaurant. We can call it add, we can call it create, but it's going to take a restaurant parameter, which is the newRestaurant, and this is a method that I now need to implement inside of InMemoryRestaurantData. I'll place this right here before the update. So, a public method, that return to restaurant named Add, it takes a newRestaurant, and for the most part, this implementation should be pretty simple. I can walk up to the list of restaurants that we have and say that I want to add this newRestaurant, but there is one additional piece of bookkeeping that I should take care of. In the next module, we're going to hook up to an actual SQL Server database, and with the SQL Server database, I plan on having the ID of the restaurant generated by the database. Currently in this InMemory database, we're just making up our own IDs. So, to somewhat simulate what a real database would do, I'm going to generate an ID here, but again, this is only for test and development, you wouldn't want to use this logic in production. So this newRestaurant.Id should equal, and let's go out to the existing restaurants and find the maximum ID. So I will use the link Max operator and say, given each restaurant, r, I want to look at the Id property to find the Max Id, and then + 1. Relatively simple logic there, and then we can return this newRestaurant that we have modified and given an ID. With that in place, I can now turn my attention back to the page model, where we have some work to do inside of OnPost, we need to figure out if we need to call Update or Add.

Handling Create vs. Update Logic

Inside of the OnPost method, one way to tell if I should be updating a restaurant instead of adding a restaurant is to look at the restaurant ID property. If the restaurant ID is 0, assuming our database will never use an ID value of 0, we could say any restaurant with an ID value less than 1 or equal to 0, that must be a new restaurant that we need to add to the data source. Otherwise, if the restaurant has an ID and that ID is greater than 0, then we need to update an existing restaurant. Now before I implement this logic, I just want to point out that I first have to check if the ModelState is valid, and then I would have to check if the restaurant ID is greater than 0. And for me personally, I don't like to use nested if statements, if I can avoid them. So I'm going to rewrite this method just a little bit. I'm going to take these bottom two lines of code, which are the lines that repopulate my Cuisines property, and then tell ASP.NET Core to render the current page. I'm going to take them and paste them inside of my if statement, but then change my if statement to say if the ModelState is not valid. So if a ModelState is not valid, let's rebuild our Cuisines and then show the form to the user again so they can fix any errors. Now let me take this logic and cut it out of the if statement and paste it below the if statement. So, again, what I'm really just trying to do is avoid nested if statements, I'm still going to have some slightly messy logic because some people will not like to have multiple returns inside of a method. But we will work with this approach for now. But now I can add the if statement to say, okay, the model is valid, and now I need to check if the restaurant ID is greater than 0. Because if the restaurant ID is greater than 0, I want to update an existing restaurant. Otherwise, what I want to do is go to restaurantData and tell it to add a new restaurant. This is the type of logic that you might also place inside of your data layer somewhere, but we're going to have it in the Razor Page, but regardless of which approach we use, we need to call commit to flush those changes into the real data source, and then we can still redirect to the detail page once we've successfully added a restaurant. Let's try this out. I'm going to save all the files. Let's come out to the browser and go back to the list of all restaurants. And then I want to say that I want to add a new restaurant. Let's add a restaurant, Mancini's. I could say this is in Stockholm and the cuisine is Italian. Click Save, and I now have a new restaurant with an ID of 4. If I go to the list of all restaurants, that restaurant appears here. Of course, that's only going to stay here until the application resets again. But we'll fix that in the next module. And then there might be one additional nice feature to have. Once the user clicks the Save button, it might be nice to present a message here to make a very explicit that, yes, you did just save some data. But that would, again, require me to pass along some information between two pages from the edit page over here to the detail page. Let's look at how to do that next.

Confirming the Last Operation

So when we commit our restaurant changes and redirect to the Detail page, we like the Detail page to show some sort of message like restaurant saved. So how can we pass along information from this Edit page to the Detail page when the Detail page is made using a separate http request? Well, one way to do this is to include information in the request itself, so somehow encode the fact that there should be a message by passing a parameter in the URL or in the query string. Right now, we're passing the restaurantId In the URL, but if I were to add some other property to this anonymous object like Message=true or Message= and some string, ASP.NET Core would also look at that property and say we need to pass this information along, and if that information doesn't fit into the URL because the Edit page isn't asking for that sort of information in the route like it is for a restaurantId, then ASP.NET Core would take that parameter and put it into the query string. So passing along information in the query string, that's certainly one approach we could take, but we would have to be careful that the user doesn't bookmark that page at that point, because when they bookmark that page, that information or that flag will be included in the query string and the Detail page will still display that message, even though the user has not, in fact, just created or edited a restaurant. What we really want is some temporary data to say show this data for right now, but forget about it in the future. And in fact, ASP.NET Core does include a data structure that's available to us named TempData. TempData is a bit like a dictionary of key value pairs, so I can walk up and index into TempData, then I can say I want to place the following information into TempData with the key of message. I want to put an object in here, which is a string which says something to the effect of Restaurant saved. And if we wanted to get fancy, we could have, instead, place this call to TempData inside of the if and the else and say Restaurant updated or Restaurant added, depending on which action we actually took. But let's make this simple and just say TempData message = Restaurant saved. And as the name implies, TempData is truly temporary. On the next request to this application, any page in the application can look into TempData and see this message and pull it out, but after that, the TempData will disappear, so we don't have to clean this up. We're simply putting some information in for the next request for the next page to show and then that temporary data will disappear. Now the only question is, how do we display this inside of the detail page? One way to do this would be, let's say, at the bottom of the Detail page to add a statement to say if there was TempData sub message available, then let's show a div with some information inside. That's one approach, but it turns out, in ASP.NET, I can also bind to TempData, which is very interesting, it's a little bit of a cleaner approach. So inside of detail, the page model, in addition to the Restaurant output property, I want to create another property of type string, call it Message, and then place the attribute on this called TempData. When ASP.NET Core sees the TempData attribute, it's going to go into the TempData structure for me and look for something with the key of message, whatever this property name is. If it finds a value for that key in temp data, it's going to take the value and assign it to this property assuming the types match, which in this case they should. And so now inside of our markup, this can be a little bit cleaner, I could say if Message is not equal to null, and of course I have to get to Model.message to reach into my page model, and I also have to make sure that I spell Message properly with the proper capitalization. So if we have a message, let's display a div and give this a Bootstrap class of alert, alert‑ let's call it info and just include Model.Message, whatever we pulled out. Let me save all my files. Let's come back and go to all of our restaurant. The application restarted, so we lost Mancini's, but that's okay. Let's just edit an existing restaurant. I can change the cuisine, click Save, and I can see now my message appears, Restaurant saved. That would probably look better at the top of the page, but I'll leave that as an exercise for you to try. But I also want to point out if I refresh the page, my TempData disappears. So if the user were to bookmark this page after editing or creating a restaurant, they're not going to come back to this page using that bookmark 3 months from now and see that message and start to worry that they just modified something in the system. It's temporary data and it's now gone.

Summary

In this module, we learned how to edit restaurants and create new restaurants, but those restaurants are still being saved just in‑memory. Now we're ready to hook our application up to a real database so those restaurants can stick around a little bit longer.

Working with SQL Server and the Entity Framework Core

Introduction

Hi, this is Scott. And in this module, we will replace our in‑memory movie data source with a data source based on a real relational database, in my case, Microsoft SQL Server. To do this, we will rely on the Entity Framework to insert data, update data, and query data. We will also see how to use the Entity Framework to generate a database schema and keep that schema in sync with the C# entities we're writing.

Installing the Entity Framework

When you work with the Entity Framework from an ASP.NET Core project, at least one created from one of the Microsoft templates, there's nothing additional that you have to do to work with the Entity Framework in this project. And that's because if I look at the NuGet dependencies that are associated with this project, I see the package Microsoft.AspNetCore.App, which is what we call a metapackage. This package exists to bring in other packages, and if I do a search for Entity Framework, or just Entity F, I can see that this AspNetCore.App metapackage brings in additional NuGet packages that bring in all the Entity Framework functionality that we might need; however, we're using a slightly different setup. Yes, ultimately we want to use the Entity Framework from the OdeToFood project, but we've also separated out our data access into OdeToFood.Data, and this particular project does not reference any NuGet packages currently. This is just a simple, humble class library. When I build out the classes that are going to use the Entity Framework, I want to put those classes into OdeToFood.Data, and that means ultimately this project is going to need a reference to those Entity Framework NuGet packages. So I'm going to right‑click on the project, say that I want to manage NuGet packages. This is something you can do from the command line using the dotnet add command. What I want to do is go out and browse and search for Entity Framework Core. So I want to make sure I'm using the EntityFrameworkCore, not just the EntityFramework. I'm going to install the latest stable version, which for me is 2.1.4. To do that, I need to accept the license agreement. And then as soon as that is finished, there's actually some additional packages that I want to install. This version of the Entity Framework, EntityFrameworkCore, supports a variety of different databases, and the package that I just installed is the core of the EntityFrameworkCore, if you will. When I go to work with a specific type of database, like Microsoft SQL Server, then there are these additional NuGet packages that will add in support for a specific database provider like SQL Server. So I'm also going to install Microsoft.EntityFrameworkCore.SqlServer. Once this is finished installing, there is one additional package that I want to install, and that is a package named Design. That package is squeezed into the second location here in the list and it is Microsoft.EntityFrameworkCore.Design. Let me also install this package. While that's installing, let me right‑click and open up OdeToFood.Data.csproj so you can see what this looks like. What I have done is installed these three packages, which you could also do by modifying the .csproj file directly, but I am now ready to work with the Entity Framework from my OdeToFood.Data project.

Implementing an Entity Framework DbContext

Working with a database from the Entity Framework Core is relatively easy. All we need to do is implement a class that derives from an Entity Framework class named DbContext and then add properties to that class containing the information that we want the Entity Framework to store in a database. We know we want to store restaurant information, so let's go into the OdeToFood.Data project. I want to right‑ click, and I went to Add a new Class. And I will call this the OdeToFoodDbContext because ultimately we want to inherit from the Entity Framework Core's DbContext class, so inherit from DbContext. All I need is the proper character, and I need to bring in the namespace, which is Microsoft.EntityFrameworkCore. And I want this to be a public class. This will give me access to the class from outside of this project, and now I just need to add properties with the data that I'd like to store in the database. I know I want to store restaurant information, so let me create a new property, and it's going to be of type DbSet of T, where the T, the generic type parameter, is going to be Restaurant. Let me bring in the namespace, which is OdeToFood.Core. Let's call this Restaurants. And I will just point out that DbSet, this is the type I need to use to tell the Entity Framework that I want to not only query for restaurant information from a database, but I also want to be able to insert, update, and delete restaurant information. Now you can have a DbContext class that is very sophisticated in the data that it stores, not just a single entity like this Restaurant, but dozens and dozens of entities, so accounts, employees, invoices, sales. And, of course, you can have relationships between these entities. We're not going to do anything quite so fancy in this course. Julie Lerman's courses on Pluralsight will cover the Entity Framework in more detail. What I want to do is move ahead and see how we can start using this DbContext in our application. But before we can start using it, we're going to need to get a database set up. And ideally, I'll be able to tell the Entity Framework to use this information about the types of information that I want to store. So if the Entity Framework knows that I went to store restaurant information in a database, then I want the Entity Framework to go out and issue the commands necessary to create a schema that can store this information. We're going to take the first step in that direction in the next clip.

Using the Entity Framework Tools

The Entity Framework can create and manage a database for us using information that we provide in this DbContext. So once the Entity Framework knows that we want to store restaurant information, the framework can create or update a database to store all the information that we need about a restaurant. The framework can change the schema of the database so that the database can store all the information available for a restaurant. This is a feature of the Entity Framework known as migrations. In order to create a migration, I need to use the dotnet command line interface. We talked about this tool at the beginning of the course, and I just want to revisit that tool again now, since we're going to need to use it to create a migration. So the dotnet command is a part of .NET Core, and it's a cross platform command line interface, so you'll have access to .NET wherever you have the .NET Core framework installed. And with this tool, I can create new applications. I can also build existing applications, and there's a whole variety of commands that are available. If I execute dotnet and pass in a flag to say, give me the help screen, what we're looking at here is a list of all the SDK commands that are available. This includes the ability to do things like run my unit tests and add NuGet package references to a project. And then down here at the bottom, you'll notice there's also additional commands from bundled tools. So the dotnet command line interface has an extensibility mechanism that allows other developers, developers from Microsoft or developers like me, to create extensions or plugins for the .NET command line interface. These particular bundled tools are tools or extensions that automatically come with the .NET CLI. The tool that we're interested in is the Entity Framework Core command line tool. So if I type dotnet ef, what I'm running now is the Entity Framework Core .NET Command Line Tools. I'm running them through the .NET CLI. And here's the commands that I can execute with dotnet ef. I can manage a database or I can manage my DbContext types or I can manage my migrations. Let's try DbContext firstW. hat I want to do is try to execute dotnet ef dbcontext, and this command will come back and tell me that I didn't provide quite enough information to do anything useful. There's additional commands here that allow me to express what do I want to do with my DbContexts? Do I want info about a specific context or do I just want to list the available contexts that are in my project? Or do I want to scaffold a DbContext? What that allows me to do is point this tool at an existing database and have the tool generate code for me to match that database. That's a very interesting scenario for a lot of people who are working with .NET Core and using an existing database. But let's just try to see if the Entity Framework tools can find my DbContext. So I want to do dotnet ef dbcontext, and then type list, and this comes back with the result, OdeToFoodDbContext. This is perfect. It has found my DbContext. Let's try another command, dotnet ef dbcontext info, so I want some additional information about my DbContext. And now I have an error. And here, ultimately is what the error is trying to say. Yes, the Entity Framework can detect that I have a DbContext inside of this project, and I do want to point out that I was running this command, dotnet ef dbcontext, from the folder that contains the OdeToFood.Data CS Project file. So I am in the same folder as the project, and from this folder the Entity Framework Tools can see that I have a DbContext derived class, but there's nothing useful that myself or the Entity Framework can do with this DbContext until it is associated with a database provider. So the Entity Framework supports different types of databases. There's MySQL, there's MS SQL Server, there's various others. And until you tell the Entity Framework, this is the type of database that I'm going to use, and in fact, here's a specific database that I want you to look at, until we reach that point, there's not much the Entity Framework can do. So what we need to do at this point is add some additional information into our project that will give the Entity Framework information about the database that we want to use. Once that is done, then we can move on to using the migrations feature.

Using Other Databases and Tools

One of the trickiest parts of getting set up with the Entity Framework is getting connected to a database. We're going to need to connect to some database to use the Entity Framework and to create migrations. So let me give you a few tips here. Since I'm using Visual Studio, I'm going to be using Microsoft's LocalDB. This is a version of Microsoft SQL Server that is installed when you install Visual Studio. And unless you heavily customized your installation, you should have a LocalDB installed. You might also want to go to your Visual Studio installation and make sure you install some of the data tools because these graphical tools can make working with the database a little bit easier. Now if you're not using Visual Studio and you're not using Windows, don't worry, there are other databases that the Entity Framework can work with. If you do a search for Entity Framework Core database providers, chances are you will end up on this page on docs.microsoft.com, which maintains a list of the current database providers supported by the Entity Framework. So, in addition to SQL Server, there's also many popular databases like SQLite and MySQL. The good news is, once you've established your database and you've worked through how to connect and how to add a connection string to the Entity Framework, all the application code that we're going to look at is going to be the same as what I write. I also want to point out that if you're not on Windows, another option is still to run Microsoft SQL Server, and to do that by using Docker images that Microsoft provides. So there's Docker images that will allow you to run SQL Server on Linux or with Docker for Mac or Windows. Again, you can do a search for this particular quick start. You should be able to find all the information you need to get SQL Server running in a container. And, again, for those of you not using Visual Studio, there are some great tools out there that will allow you to work with Microsoft SQL Server. One example is the command line tool, mssql‑cli. Just like the dotnet‑cli, this is a cross‑platform tool, so this works on Windows, as well as Mac OS and Linux. It allows you to connect to the database and issue queries, it has IntelliSense. It's really an overall great tool. There's also extensions that you can plug into Visual Studio Code that will allow you to query SQL Server. Again, what I'm going to do is use Microsoft's LocalDB. So if I come back into Visual Studio, what I want to do is go to View, SQL Server Object Explorer, just to show you that I should be able to see LocalDB inside of this Object Explorer. So this is an instance of SQL Server that is running. I already have several databases inside of my LocalDB instance and the specific instance that we want to use is LocalDB\MSSQLLocalDB. So when we create our connection string to tell the Entity Framework, here's the SQL Server that you're going to point to, we need to use this name for the name of the server. So LocalDB inside of parentheses, and then the backslash MSSQLLocalDB. Once we have a connection string configured, we can have that connection string passed into our DbContext, so it knows what database to go to. And once our DbContext is hooked up to a database, we can also start to add migrations and manage our database through this project. Let's start working through those goals next.

Adding Connection Strings and Registering Services

During development, I'm going to store my connection string inside of the appsettings.json file that is the configuration for our web application. You might remember earlier in the course, we added a message here, and I showed you how to pull this message out and display this message on a web page. Now what I want to do is provide enough information to give my DbContext a connection string to get to a localdb, a SQL Server. So I'm going to add an entire section here inside of appsettings.json called ConnectionStrings. For our application, we're only going to have a single connection string, but there's some flexibility here if you continue to build on the application to have multiple connection strings. Now I provide a key value pair. So this is the name of my connection string. Let's call it OdeToFoodDb, and here's the value. And this is the piece that will be different for different database technologies. If I want to connect to my localdb, which is Microsoft SQL Server, I will have a connection string that looks like this. First of all, I specified the data source or the server or the SQL instance. And this uses the value that we just looked at inside of the SQL Object Explorer. So I want a Data Source=(localdb), inside of parentheses, and then \\ because I need to escape this inside of the JSON file, MSSQLLocalDB, and then, I went to specify a specific database. Now this database doesn't exist on this particular server yet, but we will create it. And the way I specify a database name is to use the term Initial Catalog. So my Initial Catalog will be, let's call it OdeToFood. Now the Entity Framework will know what server to go to, the data source. It also knows what database to use, the Initial Catalog, and then the third piece of information that you typically find inside of a connection string are the credentials that the Entity Framework can use to make a connection. In the case of Windows and SQL Server, what I can do is to say I went to use Integrated Security. This is essentially a way of saying use my Windows identity to connect to the SQL Server. For most other database technologies, you'll need something like a username and a password. But for localDB, this should be all that I need. Integrated Security=True, Initial Catalog is OdeToFood, and the Data Source is (localdb)\\MSSQLLocalDB. Now the next question would be how does this connection string reach my DbContext so that my DbContext knows how to connect? Well, that's done through an area of the the application that we have also looked at previously in the course, and that is ConfigureServices. Inside of ConfigureServices, we've already told ASP.Net Core any component that needs IRestaurantData, give them this InMemoryRestaurantData component, and that allows me to take an IRestaurantData as a constructor parameter nearly anywhere in the application that I might need this. I want to do something very similar with the Entity Framework. In fact, the Entity Framework provides some methods that I can use to describe the DbContexts that my application is going to use. And that would be the method AddDbContext. And there's actually two of these here, AddDbContext and AddDbContextPool, which is new for this version of ASP.NET Core, and I would suggest using AddDbContextPool. This will pool your DbContexts in an attempt to reuse DbContexts that have been created while the application is alive, which can lead to better performance and scalability. So I want to add a pool of DbContext, specifically an OdeToFoodDbContext. And what that will allow me to do is just take an OdeToFoodDbContext as a constructor parameter, just like right now, I'm just asking for IRestaurantData, and some object just magically passed in by ASP.NET Core. We'll see later in this module, now I can just have a constructor that asks for an OdeToFoodDbContext, and one will magically be constructed and passed into me. But this DbContext is going to need some options. So let's write a small lambda expression that will customize the options for this DbContext. And specifically, what I want to do is invoke a method, which is UseSqlServer. This method is in the namespace Microsoft.EntityFrameworkCore. And this is the method that is installing my database provider. So I'm not using SQLite. I'm not using MySQL. I am using Microsoft SQL Server with this DbContext. And now the required parameter to use SQL Server is the connection string. So I have to answer the question specifically, which SQL Server do you want to use in the entire universe of SQL servers? Well, let's go to our Configuration property, which has been populated with information from appsettings.json and other configuration sources and ask it to get the connection string. And now I can just ask for a connection string by name, so if I name this OdeToFoodDb, then I can copy that string, bring it back to Startup.cs, and paste it in here. And so now I'm saying, I want to use SQL Server, and I want to use SQL Server with this connection string that you will find somewhere in our configuration sources. And I have now configured an application where I can use my OdeToFoodDbContext to talk to my localdb. But there's one more bit of code that I need to add. Over here in my OdeToFoodDbContext, I need to provide enough code so that the framework can pass in the connection string and the other options that this DbContext needs to know about to work with the database. And I can do that by adding a constructor to my DbContext and have this constructor take an object of type DbContextOptions of T where T is OdeToFoodDbContext. We'll call this options. And yes, there's lots of options inside of these options, but there's really nothing that I need to do with the options here inside of my code. What I can do is simply forward those options to my base class constructor, so the DbContext constructor, and that constructor can figure out what to do with all the information inside of this object. So now I have an OdeToFoodDbContext that can take connection options and provider options. I have services configured so that I can ask for an OdeToFoodDbContext inside of any component. Now I just need to take a step towards having a workable database using the Entity Framework migration.

Adding Database Migrations

Now that we have our connection string and configuration in place, let's go back to the command line where I want to try this operation again. dotnet ef dbcontext info. I'll do that on a clear screen. And what we're going to discover is that I still have an error, and here's why. What we're trying to do is work with the DbContext and work with this project OdetoFood.Data. And when I run the command dotnet ef DbContext info, the only information the Entity Framework has available to it is the information that is in this project, OdeToFood.Data. But OdeToFood.Data does not contain the startup configuration, ConfigureServices. That's over here in the web project. And OdeToFood.Data does not have appsettings.json, which includes the connection string. And this is just one of the situations that you're going to face if you separate your data access components and your DbContexts from the rest of the application, but it's not a problem. There's actually several different ways to solve this error that we're seeing. One approach is to add a class to my OdeToFood.Data project that implements the IDesignTimeDbContextFactory of OdeToFood DbContext. Now we've seen this term design time on a few occasions in this module. And when the Entity Framework is talking about design time, it's really talking about now, compile time, that period of time when I'm developing and designing and before I get into production. And what a class that implements IDesignTimeDbContextFactory would do is provide all the information that the Entity Framework would need to spin up a DbContext and have it connect to a real database. In other words, IDesignTimeDbContext needs to specify a database provider, like we do here with UseSqlServer, and then provide a connection string. And the idea is that particular class will only do this during design time. We'll use something completely different perhaps at runtime. Another way to solve this problem is to simply tell the Entity Framework tools look, I'm working with this project, OdeToFood.Data, but my startup project that configures all my services and my database providers and my connection strings, it's over here and another folder. So why don't you use that to spin up the environment and then start doing work with a real DbContext? Let's try that. What I want to do is execute dotnet ef dbcontext info again, but this time add a ‑s parameter, which is short for startup project. And my startup project is back up one folder. We will go into the OdeToFood folder and specify OdeToFood.csproj. And now when I run this command, the Entity Framework should find enough information to come back and give me more info about my DbContext. Now it can tell me that when I'm using the OteToFood DbContext, I'll be using Microsoft SQL Server with a database named OdeToFood. And the specific SQL Server I'm using is localdb MSSQLLocalDB. So now that Entity Framework is working, and now what I want to do is use migrations. Migrations are all about keeping a database schema in sync with the models in my application code. So right now I have a DbContext that wants to store restaurant information. So what I need is a migration that will create a database schema that allows me to store restaurant information. And then perhaps later, I'll add another class for invoices or I'll add a few additional properties to that restaurant class. At that point again, I'll need to migrate my database schema so it keeps in sync with the information on it that I went to store. I'll need to migrate my database schema to keep it in sync with the information that I want to store as defined by my C# classes. So again at that point, I could add another migration. So every time I make a change to one of my models, my C# objects that I'm using to store information into the database, what most people would call entities, any time I make a change to my entities, I can run these Entity Framework migrations to create the schema changes that I can apply to my database. So here's how this works. We're going to add our first migration. I'm going to say dotnet ef migrations. You'll notice what I can do is add, list, remove, or script a migration. What we want to do right now is add a new migration, but notice you can also generate a SQL script if you want to perform your migrations using SQL. But I'm going to do dotnet ef migrations. I want to add a migration. I now need to provide a parameter that is the name of this migration. Let's just call it the initialcreate because this is our first migration. And once again, I need to specify my startup project so that the Entity Framework has a database that it can look at to say oh, this database doesn't exist. I need to create a migration that includes everything. So now when I execute this command, what this command will do is add C# code into my project that describes the type of database schema changes that need to be applied. So this command executed successfully. Let's come back into Visual Studio. What I want to do is look inside of my OdeToFood.Data project, and I can now see a Migrations folder. There's a few different files in here, which are mostly just the Entity Framework doing some bookkeeping. The most interesting file here is the initialcreate file. So that's the name that we gave our migration. And in here, we can see that the Entity Framework created some C# code that invokes an API on this object migration builder that will do things like create new tables. In this case, the Entity Framework sees that I have a class called restaurants, and I'm using that in a DbSet. So it's going to create a table named Restaurants, and this table is going to have columns that match the names of the properties that are in my restaurant. So my restaurant C# class has Id, Name, Location, and Cuisine properties. Therefore, my table has Id, Name, Location, and Cuisine columns. Notice the types of the columns will match the types of my properties and that Entity Framework is even looking at my data annotations to see that oh, the name of a restaurant should have a maximum length of 80, and it shouldn't be nullable. It's a required field in this application. So we won't make that a nullable column. There's also APIs available inside of here to create indexes and also to execute arbitrary SQL. But what I want to do next is show you how you can use the dotnet ef tool to update a database with this migration that we just created. Let's do that next.

Running Database Migrations

Before we apply the migration to our database, let's open up our database. And in the Object Explorer, I just want to point out there's nothing up my sleeve and there is no OdeToFood database inside the collection of databases here. Now let's go back to the command prompt. What I want to do is use dotnet ef and this time go to the database command. So here's what I can do with the database command. I can drop a database or I can update a database to a specified migration. So still working from the OdeToFood.Data folder, what I want to do is update my database, specifically the database that is pointed to by the connection string in my OdeToFood project. So I want to do dotnet ef database update and once again specify a startup project that points back to my OdeToFood.csproj. Let's execute that command, which, by the way, there's additional flags I could add if I wanted to have some additional verbosity. But even with the defaults here, we should see some SQL DDL statements roll by, some statements about creating tables, and there they go. And I can see what the Entity Framework has done is a number of things, but it has created a table named Restaurants with the Id, Name, Location, and Cuisine. It's also, by the way, decided that since I have a property named Id, that's going to be the primary key for this table. So all of this could happen just by convention. If you don't like these conventions, there is a C# API that you can use to tell the Entity Framework to behave differently. There's also attributes that you can apply to your entities to say that this property name maps to this column. Perhaps you want to call it something different, something other than Name, or have some sort of prefix. But now that this command has executed successfully, I should be able to come back to Visual Studio. I should be able to refresh the list of databases that I have here, and I now have OdeToFood. If I drill into this, I now have a table, dbo.Restaurants, and there's also a table that the Entity Framework keeps here to track the migrations that have been applied to this database. So now my database is in sync with my C# code. In the future, if I make a change to the restaurant, if I add a property or remove a property or change one of the data annotations on that entity, I can go through the same series of steps that I just followed here. I could add a new migration and then use dotnet ef database update to apply that migration to my database and get it back in sync with my entities. So now we have our DbContext, and we have our connection string, and we also have a database. But before we start using this database, let's implement one more operation in IRestaurantData that you typically see in a database application, and then we'll be ready to start using our database.

Implementing a Data Access Service

Inside of IRestaurantData, we have create operations, read operations, and an update operation. What we don't have yet is a delete operation. Let's go ahead and add that before we build another implementation of IRestaurantData that uses our DbContext to talk to a real database. So I want something that will return a restaurant, it's named Delete, and perhaps all we need the caller to pass in is just the id of the restaurant that they want deleted. Let's provide an implementation of that operation in our InMemoryRestaurantData. But before I do, I'm going to place the cursor on the class name, hit Ctrl+period in Visual Studio, and I'm going to move this type to another file, InMemoryRestaurantData.cs. So this follows a convention that I'm accustomed to, which is to keep each type in a separate file. And since we're about to implement another implementation of IRestaurantData, I want each implementation and then the interface itself each to have their own file. So now I can put the cursor on the interface and press Ctrl+period and say go ahead and implement this interface. Visual Studio is smart enough to say oh, I see you don't have a delete method yet Let me go ahead and add that. It did that here at the bottom of the file. And now I just need an implementation. So let's go out and find the restaurant that we want to delete inside of the collection of restaurants that we have. So I could ask for the first or default restaurant. We're given a restaurant, r. The Id matches this incoming id. And then if my restaurant variable does not hold a null value, I must've found a restaurant. So let's go to our Restaurants collection and ask it to remove this object. And then we can just return restaurant. And now I should have a project that builds successfully, which it does. And I have an interface, IRestaurantData, that I now need to implement using my OdeToFood DbContext. Let's do that next.

Saving and Commiting Data

Now what I want to do is add a new class to the project, but without going through the usual Visual Studio interface. So this class will be SqlRestaurantData that implements IRestaurantData. And instead of the usual interface, I'll again put my cursor on the class name, press Ctrl+period, and say that I want to move this type to its own file. So now I have a class in its own file without right‑clicking on the project, so SqlRestaurantData. It's going to implement IRestaurantData. Let's go ahead and use Ctrl+period again and say Implement interface. And now we can work through each of the methods on IRestaurantData. So first of all, in order for SqlRrestaurantData to do anything meaningful, it's going to need an instance of OdeToFoodDbContext. And like everything in ASP.NET Core, we want to avoid constructing an OdeToFoodDbContext ourselves. Instead, I just want to ask for this as a constructor parameter. So I'll take a parameter named db of type OdeToFoodDbContext. I'm going to use Ctrl+period to initialize a field with that db value. And the beauty of all this is through the magic of ConfigureServices, ASP.NET Core is going to come across some component or some Razor Page that needs something that implements IRestaurantData. We're going to configure in SqlRestaurantData to be injected for IRestaurantData. And ASP.NET Core is going to see oh, the SqlRestaurantData needs a DbContext. Let me work with the Entity Framework to grab one of those and pass that into the constructor. So there's nothing else I have to do to receive a DbContext. And now when someone wants to add a new restaurant, this is a very simple operation with the Entity Framework. I can go to my Restaurants property, and I can say Add here. I can also actually just say Add right off of the DbContext because the Entity Framework is going to be smart enough to say oh, you're passing in an object of type restaurant. You must want that added to the Restaurants collection, which is really the restaurants table inside of the database. So Add this new restaurant that someone has passed in. And since we are forced to return a restaurant, let's just also return that newRestaurant. The commit operation with the Entity Framework also very simple. With the Entity Framework on the DbContext, there's a method, SaveChanges. All I need to do is invoke this method. This method does return an integer, which represents the number of rows affected in the database. So I'm just going to go ahead and return the value returned by SaveChanges. Now there's the delete operation. What I'm going to do is first go out and find the restaurant to delete. This is very similar to our implementation in the InMemoryRestaurantData. But let me do that just by going to the GetById method that we already have available, so find me a restaurant. And then if that restaurant is not equal to null, I know this restaurant exists in the database, then I went to go up to the Restaurants DbSet property on my DbContext and tell it to remove this restaurant that I've just queried for. And at that point, we can return the restaurant. And I do want to point out operations like db.Restaurants.Remove and up here under add the db.Add newRestaurant, nothing will change in the database until someone calls commit. That's when the Entity Framework tries to reconcile all the changes that I've asked for with the database. So it says oh, you've added two restaurants and removed one. Let me issue two insert statements and a delete statement. That's what happens during SaveChanges. And all those changes happen atomically. Now let's implement GetById. This is also a very simple operation with the Entity Framework on the Restaurants DbSet or on every DbSet really. There is a Find method where I can just pass in the primary key or the key values for a particular entity, and the Entity Framework will figure out how to track down that entity inside of the database. So for restaurants, there's a single key. It's the id property. I just pass that into Find. If the Entity Framework can find that restaurant, it will return the restaurant. If it cannot find the restaurant, it will return null, and that's perfect behavior for what we need. Let's also look at GetRestaurantsByName. So now we can write more of a LINQ query. So we could say from r, which is a restaurant, in db.Restaurants where r.Name starts with this incoming name or the name is empty. So these expressions are actually C# code that the Entity Framework can translate into a SQL query against the database. So all of the heavy work occurs inside of the database, including the orderby. So I can say let's order by the name of the restaurant and then select that restaurant and then return this query. Now you might have noticed the little red squiggly here. We were not getting much help through IntelliSense. All of that was because I did not have System.Linq included, that namespace. That brings in all the extension methods that are the magic behind this particular query operation. And now you'll notice that the query looks fine. There's no red squiggles. And I should also have IntelliSense help inside of here. So when I say r. something, I know I'm looking at a restaurant, which is good. Now for update. An update operation with the Entity Framework can look different depending on what you want to achieve. But here's the approach I'm going to use. First of all, I'm going to use a method on my Restaurants DbSet, which is the Attach method. And what I want to attach is this updatedRestaurant. So Attach is a way of telling the Entity Framework I'm about to give you an object that represents information that is already in the database. But I want to give you this object because I want you to start tracking changes about this object. So I'm not adding a new restaurant. I'm not trying to retrieve this restaurant from the database. I already have all the restaurant data here inside of an entity object, and I'm just going to attach it to the DbContext so that the DbContext can manage that particular restaurant, that entity. And then I'm going to tell the Entity Framework that the state of this particular entity, this restaurant, is modified. So I can say EntityState.Modified. And to make this code look a little bit better, I can get rid of the namespace portion and then use Ctrl+period to bring in the namespace, which is Microsoft.EntityFrameworkCore. So essentially what we have done, which is I now have to return the updatedRestaurant to return a restaurant. But what we've done is we've attached this entity to the Entity Framework. We've told the Entity Framework that it is modified. So the Entity Framework has to assume that everything about this restaurant has changed except for the ID. But the name might be different, and the cuisine might be different. And so now when someone calls SaveChanges and the Entity Framework is reconciling all the changes I've made, it's going to come to this particular restaurant and say I need to issue an update statement. I need to make sure all those attributes in the database for this particular entity match the values that are in the properties right now. And that's what will happen when someone calls SaveChanges. And with that, we now have a complete implementation of SqlRestaurantData. We should now try it out in the application.

Modifying the Service Registration

To use SqlRestaurantData inside of the application instead of our InMemoryRestaurantData, there's really just one change we need to make, and that's inside of Startup.cs. Here inside of ConfigureServices, we used to say use InMemoryRestaurantData whenever you need an IRestaurantData, but now I want to use SqlRestaurantData. I also want to make very sure that I change this from AddSingleton to AddScoped. A singleton DbContext or a Singleton RestaurantData would be very bad for the application. Typically, the way you want to use your data access components if you're using the Entity Framework behind the scenes is to have your services scoped to a particular HTTP request. So every time the framework hands out a SqlRestaurantData, that's going to keep handing out the same instance of SqlRestaurantData for a single request when the next request comes in that might be a new RestaurantData, but this allows the DbContext that's working behind the scenes to collect all the changes that are needed during a single request. So I now have services.AddScoped, and I'm using a SqlRestaurantData implementation. I should now be able to save everything, do a build, come out and refresh my application, and all my restaurants are gone. And that's because we're looking at the database now, and the database is completely empty. So let's add a new restaurant. I'm going to add Scott's Pizza back into the application. I'll give this a Location of Maryland, set the Cuisine to Italian, click Save, and it looks like I have successfully saved a restaurant in the database. Once I'm finished recording, I'm going to go through and add back the other two restaurants, but I should be able to come into my database, that OdeToFood. Let's go to the Restaurants table. I'm going to right‑click and say, let's View the Data inside of here, and there is my restaurant stored as the first record in the table. I should now be able to add more restaurants, which I'll do offline. I should be able to drill into the details for this restaurant, and I should be able to update this restaurant. The only other operation we need to add here is the ability to delete a restaurant, which we'll do in the next module.

Summary

In this module, we learned how to use the Entity Framework to talk to the database. We used EF migrations to not only create a database schema, but also create the database itself. And we now have the ability to create, read, and update data in a database. In the next module, we'll implement the delete feature and look in more detail at some of the UI features of ASP.NET Core.

Building the User Interface

Introduction

Hi, this is Scott, and in this module, I want to show you some more details and tips for working with Razor pages. We're going to learn about some of the special pages in ASP.NET Core, including the Layout page, ViewStart, and ViewImport files. I also want to show you how to create reasonable components in the UI using partial views and view components.

Using Razor Layout Pages and Sections

For our first topic, I would like to revisit the Layout page in this project. You might remember at the very beginning of this course, we wanted to modify the Application menu to add this link to the restaurants list, and to modify that menu, we had to go into the Shared folder and look at \_layout.cshtml. Let me give you three quick tips about this file that might clear up some confusion. First of all, this file does not need to live in a folder called Shared, but that is something that you're commonly going to see in ASP.NET Core projects. Anything that you place in the shared folder, the idea is that this is a component or a page or a view or a file that can be shared amongst all the other pages, and all of the pages in this application will plug content into the Layout page. That's what we're going to look at in more detail here in just a bit. But first, these tips. So first of all, you'll typically find layout inside of the shared folder. Secondly, this technically is not a Razor page. A Razor page is a page like list or edit or detail that we've created and even NotFound.cshtml is a Razor page. Notice that NotFound.cshtml does not have a page model, so there is no CS file behind this Razor page, but it is a Razor page because it has the @page directive and one of the capabilities the @page directive provides this file is the ability to respond to a URL, in this case, it would be /restaurants/notfound, but the Layout page does not have a page directive. Technically, this is just a Layout view or a Razor view, and developers who work frequently with ASP.NET Core MVC, they also can produce UI using cshtml files and they typically refer to those cshtml files as views. So, yes, it can be a little bit confusing, but sometimes you're going to hear things referred to as a view, sometimes as a page. but just know, technically, from ASP.NET Core's perspective, you can only be a page if you have the @page directive. The third tip I want to give you is about this leading underscore in the file name. You'll notice a few files inside of pages that have this leading underscore like the CookieConsentPartial and ViewImports and ViewStart, and this leading underscore is not required, but again, this is a convention that you'll see many ASP.NET Core developers follow. The leading underscore is an indication that the CSHTML file does not exist to render a view on its own, it's typically a component that is required by other views or plugs into something else. So while I might redirect a user to the NotFound view, I would never redirect the user to the Layout view or to ViewImports or ViewStart. We'll see the meaning of those special files later in this model. Right now, we're talking about the special file, \_layout. It's the job of layout to control the structure of my user interface. So what are the common elements that need to appear on every page in the user interface? What scripts and CSS files do I need to include on every page? That's the type of content that goes into layout. So you can see I had scripts at the top, I have a navigation menu inside of here, and then at the bottom, we'll be sure to place a copyright onto every page and that's why the copyright appeared here at the bottom. And so, now there is two valid questions to ask at this point, first of all, how does a page like restaurants list no where to plug in content like this table of restaurants? And then a second good question would be, how could that restaurant list page choose a different layout page or choose not to use this layout page? In other words, exactly how does this layout page come into play? We're going to answer that second question in the following clips, but right now I just want to show you that any time you're rendering a page like I'm rendering the List page or the Detail page, the content that is the markup that is produced by this page, when there is a Layout page in effect, will always be placed into the Layout page wherever you have a call to RenderBody. So this Layout page has a call to RenderBody just inside of a div that is a Bootstrap container, and that's why when I'm rendering the restaurant list, that list will appear just before we get to this horizontal divide that is specified in the Layout page. But what if I have a Content page like list.cshtml that also wants to play something into the footer like down here with copyright? So, for example, this message that we're displaying from appsettings inside of list.cshtml, what if I wanted that to appear beneath the horizontal divide? Well, that's a situation where render section can be helpful. Inside of a layout view, I can invoke another method called RenderSection, and I can have multiple calls here, so each section will have a name. Let's call this one the footer section. So what I'm doing inside of the layout page is saying, I'm going to allow other content pages like list and detail and edit to plug content into the section that I'm going to call footer, and I'm also going to pass a second parameter here to say that this section is not required, in other words, the list and the Edit page, they're not required to provide me this section. And now, if the List page wanted to render a message into this footer, all it needs to do is define a section using @section. So right here is where we're displaying the message, the message that we read from configuration. I'm just going to wrap that inside of an @section where I provide a name, so I'm going to provide the name, footer, and then I use curly braces just like I would for a C# code block. And now, if I save all my files and refresh the browser, we should see Hello from appsettings jump from above the horizontal line to below the horizontal line, and then we'll go into the footer, which is what has happened. So now I know that everything inside of the Razor page cshtml is going to be rendered out wherever we have RenderBody inside of the Layout view. Unless I have that content wrapped with an @section directive, then I can render into these alternate sections that a Layout view might provide. Now to answer the next question, how does a view select this Layout view, let's implement the delete feature that our application still needs.

Implementing a Delete Restaurant Page Model

I want to give the user the ability to delete a restaurant. So I'm thinking we'll have an icon here to the right of each restaurant that the user can click on, and this will take them to a page where they can confirm that they do, indeed, want to delete this restaurant. Before we add that icon, let's create the page. So I want to right‑click on the Restaurants folder and add a new Razor Page. This time I'm going to add one with a little bit of a twist, and that twist is I'm going to uncheck the box that says Use a layout page. So all the pages we've created so far use the default layout page. This one, the Delete page, will not. Let me click to add this to the project. Visual Studio will scaffold out the page. And before we start looking at the markup inside of the CSHTML file, let's work with the page model. Now, ultimately to delete the restaurant we're going to need an IRestaurantData service. We've been using that service throughout the course, so let's go ahead and inject IRestaurantData into this page model. I'll need to bring in the namespace OdeToFood.Data. We will name this parameter restaurantData, and I'll hit CTRL+period to save that off into a field. And now here's what this page model is going to be responsible for. When there is a GET request, we're going to present the restaurant to the user, so we're going to show information and ask the user if they're sure they want to delete this restaurant. The page we show them will also have to include a form because we do not want to delete the restaurant until the user posts some information back to the server and says yes, I confirm this delete. Remember, we never want to modify data during a GET request, so I want to create an OnPost method. It's inside of there where we will do the actual deletion. And like we talked about earlier in the course, after we do the delete, we're going to want to do a redirect and issue a GET request for some other page so the user is not sitting on the result of a post. That's why I know OnPost is going to return some sort of action result. OnGet I suspect also will need to return an IActionrResult. I also expect both of these methods to require a restaurantId. We can force this restaurantId to be in the route when we move over and modify delete.cshtml, just like we did with the detail page. We want this page to respond to a URL like /restaurant/delete/3, where the number 3 is the restaurantId. So first, let's respond to OnGet. I know that I'm going to need to display a restaurant. So let me go ahead and create a property of type Restaurant, and that means I bring in the namespace OdeToFood.Core. We can call this property Restaurant, and this gives me the model that I combined against from my CSHTML file. Now we just need to populate this. So let's say that Restaurant = restaurantData. Let's go out and get the restaurant by this restaurantId. And there's always a chance that restaurant is not in the database, so if the Restaurant is null, we will return a result that tells ASP.NET Core what we really want to do is redirect to another page, specifically the NotFound page. Otherwise, we did find a restaurant, so let's render the page. Now, inside of OnPost, I can go ahead and ask my restaurantData data source to delete the restaurant with the following ID, and remember that returns a restaurant to me, the restaurant that has been deleted, and that might be helpful because I can use that information in a message. But let's go ahead and tell restaurantData to commit those changes. Now, of course, there is a chance that we arrived here, and the user either passed us a bad restaurantId or maybe some other user deleted the restaurant before this user could get to it. So let's just check and make sure the restaurant is not equal to null. In other words, if it is equal to null, we can redirect to the page once again, the NotFound page. Otherwise, what do I want to do? Well, I don't want to just allow the page to render and display a page as the result of this POST request. So let's instead redirect to a safe page and issue a GET request to show the list of all restaurants. And just like we did with add an update, we could put a message here into TempData, just so the user knows that they really did delete something. So let's say that restaurant.Name deleted. That should be all the C# code that I need in the page model. Now let's turn our attention to the page itself.

Implementing the Delete Markup

Let me copy my parameter name restaurantId because the first thing I want to do as I move into delete.cshtml is add to the @page directive to let ASP.NET Core know that this particular page is going to require a restaurantId in the routes, so /restaurants, /delete, /some restaurantId. Now remember when we created this page, we told the scaffolding framework do not use a Layout page, and that results in two differences that we have not seen in other Razor pages. First of all, there's this code block here, so @{ is a C# code block, and inside of this code block, I can invoke methods that are available through a Razor page, as well as set properties on my page. What this is doing is setting a property on my Razor page that says Layout = null. In other words, it's telling the ASP.NET Core framework I do not want to use a Layout page. We're going to investigate that property in more detail in a bit, but for right now, I just want to point out that we also have a page that has to render a DOCTYPE, and an HTML element, and a head element, and a body element all because we're not going to use a Layout page, which typically takes care of those elements for us. Now in just a little bit, we're going to switch this page over to use our Layout page, but for right now, I'm going to pretend like we're moving ahead without the Layout page. So first of all, on this Razor page, I want some sort of header. I'll let the user know they're about to delete something. Let's create a div with Bootstrap styles to say that this is a dangerous operation you might perform here, and are you sure you want to delete, and we can go to our model, to the Restaurant property, and say do you want to delete this restaurant name, and now we can give the user a Yes button to click on. That will be inside of a form whose method will be equal to post. We want to make sure this is a post operation. Now by default, if I do not specify the action attribute here, the form will post back with the same URL that it came from. So if the user is viewing /restaurants, /delete, /3, and they click on a button to submit this form, this form will post back to that same URL, and we should receive this post in our PageModel. So let's just give the user a button, the type will be submit. We can also give this button a styling to make it look a little bit dangerous, and add the text Yes. So I will click on that button if I want to delete; otherwise, what do I want to do, maybe we need a Cancel button here also. So let's use an anchor tag with the asp‑page helper that will point back to the List page. We will use Bootstrap to style this link to make it appear as a button. So a button, a default‑looking button, and just use the text Cancel. Now all we need inside of our restaurant list is a way to navigate to this Delete page. So let me make a copy inside of our restaurant list of the second icon that is here, the Edit icon. I'm going to copy and paste that, we just need to change some parameters around. So I want to go to Delete, and I do still want to pass along restaurantId, and I just want to change the glyphicon. So instead of glyphicon‑edit, let's use glyphicon‑trash. Save all my files, refresh, and we now have an icon to click on. Will it work? Stick around and find out in the next clip.

Using \_ViewImports and \_ViewStart Files

Let's see what happens when I click to remove a restaurant, and the first thing we'll notice is that the Bootstrap styles we included in the Razor page are not having any effect. And that's because the Delete page does not include Bootstrapped.css. In our application, It's the Layout page that includes Bootstrap.css for everyone else, so our Delete page either needs to include a link to Bootstrap.css or use the Layout page. We'll fix that in a moment. For right now, let's see if the Cancel link works. Yes, it brings me back to the list of all restaurants. And if I click Yes, that I do want to delete this restaurant, it does appear that the restaurant is deleted, it's no longer in my list of all restaurants. But now let's go back to the Delete page and see what we can do with this Layout property. This Layout property gives the Delete page the power to explicitly set the Layout page that it wants to use. So for example, I could say that I want to use a Layout page named Layout2. Now our application does not have a Layout page named Layou2, but that's okay because the error that ASP.NET Core gives me is very informative. It's going to tell me that it could not find a layout view named Layout2. And it looked in the following locations, in/Pages/Restaurants, in the root of all Razor pages, just the /pages folder, and also in the Shared folder. This is one of the reasons why developers will place layout pages into that Pages/Shared folder because the ASP.NET Core framework will always look inside of that Shared folder. This should also be a clue to you that you can have multiple layout pages inside of an application. So I could have a layout view, and again, I'm using the terms view and page somewhat interchangeably, but I could have a layout view inside of my Pages/Restaurants folder that is just for those restaurants, or perhaps I have multiple layout pages that are different themes for my website, and the user can pick which theme they want to use. But in either case, explicitly setting the Layout property inside of my Razor page will always tell ASP.NET Core what layout page to use. What if I do not have this property here? Now my Razor page looks like other Razor pages, like the List page here that doesn't explicitly set the Layout page to use. And by the way, I should go ahead and remove the DOCTYPE, and the html, and the head, and the body, and all this markup that we don't need, since we're using a Layout page now. When I do not explicitly specify the layout, I do receive the \_layout view that this application has. How does that happen? Well that happens because of a bit of a magical file that exists here inside of the Pages folder, and that file is \_ViewStart.cshtml. When ASP.NET Core goes to render a view or a page, it's also going to look for an \_ViewStart file, and when it finds a ViewStart file, it's always going to execute the code inside of the code block here before executing the underlying page. So if I just want the default layout view for every page in this application to be \_Layout, I can place this block of code inside of \_ViewStart, and this bit of code will execute to set that property before the code inside of an individual Razor page will execute, and that applies to every page that is in this folder, as well as every sub folder like my Restaurants folder. And while we're talking about special files, another file here that starts with a leading underscore is \_ViewImports file. There's a couple different things that happened inside of \_ViewImports. One is that I can add using statements that will be included in the compilation for other Razor pages in the project. So if I always want types inside of the OdeToFood namespace to be included when I'm writing code inside of my cshtml files, this using statement will help me with that. I can also set my default namespace, and this is how I add in tag helpers to make them work. So the tag helpers that we've been using in ASP.NET Core are tag helpers that come from Microsoft. These are tag helpers like asp‑page and asp‑for, and those tag helpers would not work unless I have this @addTagHelper directive either on the individual page that is using the tag helper or here inside of \_ViewImports, which will apply these settings to all my pages. And what this directive is saying is use all the tag helpers that you can find inside of this assembly named Microsoft.AspNetCore.Mvc.TagHelpers. Now in this course, we're not going to create a custom tag helper, but if you were to write your own tag helper, you would also have to register that, I would have to say @addTagHelper. I would probably just say register all the tag helpers that you can find inside of the OdeToFood assembly, and that would allow me to use my own tag helpers inside of my Razor pages. Some of the other underscore files that you see inside of here are what we call partial views. We'll talk about them in just a bit, but for now, we know a little bit more about how layout pages work, and how the Layout page is automatically set for me inside of \_ViewStart. This is why my Delete page now has the proper look and the proper Bootstraps styling applied visually. But we also know now that I can override that if I want to by specifying the layout property directly here inside of the page itself.

Reusing Markup with Partial Views

In the last clip, I mentioned the term partial view. Let's look at what a partial view can do. First of all, a partial view, as the name would imply, only renders a part of your content, and this is useful in a couple different scenarios. One scenario where using a partial view can be useful is if you have a complicated user interface. Let's say you have a large, complex model that requires a lot of work to present to the user. You might consider breaking that complex model into pieces and have partial views render each piece of the more complex model. The restaurant entities that we're using are relatively simple in our application, but imagine a more complex restaurant model, one that perhaps included the menu for the restaurant, the chefs for the restaurant, as well as the restaurant's contact information and their operating hours, and perhaps some restaurant reviews. Instead of trying to express the markup for all of that model in a single Razor page, perhaps I break off pieces of that restaurant so that the reviews, for example, are rendered by a different component, a partial view. So one way to use partial views is to break up a large or a complex page into simpler pieces. Another scenario where you might want to consider partial views is when you need to re‑use chunks of HTML. So imagine I wanted to display the summary information for a restaurant in several locations throughout the application. If I want to make that display consistent, I would create a partial view to encapsulate that UI. So let's add a Razor page, and what's a little bit confusing here, again, this differentiation between a page and a view, I'm not really going to add a Razor page, but I can still go through this dialogue in Visual Studio where I say Add New Razor Page, and then I will say do not generate a PageModel class. That's because a partial view really doesn't put together an independent model. Whenever you're using partial views, you can pass model information into your partial views from the primary page, so I do not want a PageModel class, and in fact, I want to select this checkbox that says create this file as a partial view. I'm going to call it \_Summary. As I pointed out earlier in the course, the leading underscore is not required, but it is a signal to other ASP.NET Core developers that this is not a primary view or a primary page, this is a partial view that I would only render from other views or other pages. Let's click Add to place this into the project, and I've placed this into the Restaurants folder, \_Summary.cshtml. So this is going to be a partial view that I really can only use from other pages that are in the same folder. The first thing I'm going to do is delete everything that was placed into here, including the @page directive, it's very important to delete the @page directive. I do not want this to be a Razor page, I want it to be a plain, simple view, and I do want to give it a model. This partial view is going to display the summary information for a restaurant, so I'm going to expect that someone is going to pass this partial view a restaurant, and this is one of those situations where the type restaurant is not in scope, so I either have to have a using statement here or use the full namespace, but let's go ahead and add a using statement for OdeToFood.Core, which is where a restaurant lives, and now the compiler will be happy. So what would a restaurant summary look like? Well let's turn to Bootstrap and use a class of panel and use a default‑looking panel. In Bootstrap 4, panels become cards, but I'm going to give this panel a panel heading, and inside of the heading, let's use an h3 element that displays the restaurant name. So again, @model, lowercase m, defines the type of the model that will be passed to this partial view, where @Model uppercase M is the model property where I can get to the model. So let's display the restaurant name inside of an h3, then we can have a panel body with the mundane information, so a class of panel‑body. We can show first the Location as @Model.Location, and then show the Cuisine type, which will be @Model.Cuisine. And now perhaps just like we have in the list view, I want a collection of action buttons, perhaps place them inside of a panel‑footer. So let me come over to the List view, and copy the collection of buttons that we have, bring those over into the \_Summary, paste them in, and all I will need to do here is change from restaurant.Id to model.Id. So just copy and paste that three times. So I now have a partial view called \_Summary in my Restaurants folder that I can use from any other page to display information about a restaurant. Let's look at how to use this partial view next.

Rendering Partial Views

Let's use our new partial view here on the list of restaurants. So instead of displaying all our restaurants inside of a table, let's display our restaurants as a series of panels. So that means for each restaurant in my model, I want to show this partial view \_Summary. So over inside of list.cshtml, one thing I can do here is take my table and completely remove it from the view. Instead, what I want to do is a loop through each restaurant in the Restaurants collection that came in as part of my model, and now it can render each partial view. I can do that with a tag helper named partial. So all the tag helpers we've seen so far, tag helpers like asp‑for and asp‑page, all these tag helpers appear as HTML attributes. Partial is a tag helper that appears as a full element, and when I use the partial tag helper, there's a couple parameters that I have to pass along, and I can pass those parameters in what looks like HTML attributes, but just remember, all of this is processed and rendered on the server, so there is no partial tag being sent to the client. This partial tag is going to be replaced with the contents of my partial view \_Summary, and the way I specify that is by setting the name. What do I want to render? I went to render \_Summary. And because this partial view is in the same folder as this list Razor page, ASP.NET Core will easily find this partial view. I could also place this partial view into the shared folder, and then I could render this partial view from anywhere. The second bit of information that I need to specify is the model, what model do I want to pass into this partial view? Well, I want to pass in the current restaurant that I am looping through, and with that, I can close my tag off. And I also want to point out inside of list.cshtml itself, this partial view has helped to simplify this view somewhat. We've removed a lot of code here and placed it into a partial view that is solely responsible for just showing an individual restaurant. Let me save all the files. Let's come out to the browser and test this with a refresh. And what we have again isn't the fanciest UI, but I now have my restaurants in a series of panels, and the action buttons here are still available. Now partial views work very well in this situation where I already have a model, and I just want to hand off that model or a piece of that model to a partial view to render. So as long as I can give the partial view the model information it needs, everything works fine. What happens if I want to have a reusable piece of UI that I can use from everywhere, and I want to make that a little more autonomous, a little more independent. What I want is something like a partial view, but a partial view that has the capability to do its own data access. Well, that's when I can turn to view components in ASP.NET Core. Let's look at those next.

Implementing a ViewComponent

Now if someone came to me and said there's a new requirement for this application, what we want to do at the bottom of every page in the application is show the total number of restaurants that we know about. So count the total number of restaurants in the database, and display that number at the bottom. Well, as soon as someone says for every page in the application, I'm thinking about the layout view because I want to add this feature once in the layout view instead of modifying every single individual Razor Page. But there's a bit of a problem. A layout view is just like a partial view. There is no page model for a layout view. There's no easy way for me to do data access. And really, anything dynamic that the layout view might have to display has to be information that comes from the Razor Page itself. In fact, we're kind of looking at an example right here where the layout page is setting the title for a page. So because the layout view renders the head tag, it has to render the title tag. But the layout view, since it's in place for every page in the application, it doesn't know what the title should be. Someone has to pass that title to the layout view so the layout view can render the proper title. I can do that by placing an entry into this ViewData data structure that ASP.NET Core provides. So every Razor Page and partial view and layout view share an instance of this ViewData data structure, so anything I place into ViewData can be read by any of the other views that are involved. And this is a dynamic dictionary, so I can place really any type of object into this ViewData that I want. I can place a string value in, I can place a restaurant, any type of object. And typically, what you'll see is that pages like Detail.cshtml, they will set the title of the page by placing a title into ViewData here in the opening code block. Now perhaps you've noticed that our List page does not properly do this. The title of this page is just ‑ OdeToFood. And that's because this expression, which I'm going to copy from Detail.cshtml, that expression does not live here inside of this code block . I'm going to add it. And here I could say this is a list, or I could just say this is Restaurants. And now if I refresh my List page, it's now Restaurants ‑ OdeToFood. We should really do the same thing with the DeleteView since we never set a title here either. So ViewData is one opportunity where a Razor Page can pass information to a layout view. But this requirement I'm facing really is not a requirement that I want to impact my Razor Pages. I don't want each Razor Page in the application to count up restaurants and pass that information to the layout view. I want something that can independently and autonomously go out and build whatever model is needed and render a piece of UI that I can place here into the layout view. In ASP.NET Core, that's the type of operation that you can perform with a view component. Building a view component is almost like building a little Razor Page, except a view component is more like building something that uses the model‑view‑controller framework instead of Razor Pages. Let's see how this works. I'm just going to create a new folder in this project, and I'm going to call it ViewComponents. So this will be all the different view components that I place into the application. Inside of this folder, I want to add a class, and let's call this class the RestaurantCountViewComponent. A view component in ASP.NET Core derives from a base class, which is the ViewComponent class. I need to bring in the namespace Microsoft.Asp.NetCore.Mvc. And a view component, just like the page model in a Razor Page, can have a constructor, or I do dependency injection. So if I need to count restaurants, I can ask for IRestaurantData, and I'll do the same thing that I do in a Razor Page, so bring in the namespace OdeToFood.Data and then save this value off into a private field so I have access to that field later in the processing. Now, one significant difference between a view component and a Razor Page, actually, there's several significant differences, but one significant difference is that a view component doesn't respond to an HTTP request, so I'm not worried about get or post or any of those types of operations. A view component is more like a partial view. It's going to be embedded inside of some other view that is simply going to say please go render this view component. So when that rendering happens, ASP.NET Core is going to call a method named Invoke. So let's create a method that returns an IViewComponentResult named Invoke. IViewComponentResult, by the way, very much like an IActionResult. It simply encapsulates what's going to happen next. And typically, the only thing that can happen from a view component is that you render some sort of view. So we'll have an IViewComponentResult, but now we need to think about our data access. So currently, there's no efficient way to count restaurants from an IRestaurantData service. Let's go into our data project and see if we can add this quickly. So in our interface, let's have a method that returns an int and call it GetCount, GetCountOfRestaurants. Let me save that file, and that should produce two compiler errors in my application, one with InMemoryRestaurant and one with SqlRestaurant. So first, let's go to InMemoryRestaurant, and let me just hit Ctrl+period here to implement the interface that will have added that operation here somewhere in the file. And turns out, it's down here at the bottom. And all I really need to do is just return the count that we know about in our list of restaurants. Let me close that file, and we will open up our SQL implementation where once again I'll go to the interfact, hit Ctrl+period. I want to implement this interface. Let's scroll down to the bottom of the file. This time it placed it a little more into the middle. But now I can say return db.Restaurants.Count. So one thing to think about is that this operation, this view component, if I place it onto the layout page, it's going to add a query to every page in the application. So if you're in a performant‑sensitive scenario, you might want to cache this value. But we are really just going to focus on building the view component itself. So let me build my model, which is the count of restaurants, so a very simple model. But I can go to restaurantData, tell it to give me a count of restaurants, and now, unlike a page where I can simply say return page to render the associated cshtml file, I need to explicitly say I want to return a View. So somewhere on the file system, there's going to be a cshtml file that represents the view for this view component. I'll show you how to specify that in just a bit. But this is how I pass my model information along. So, again, this is another significant difference between view components and Razor Pages. View components work a little more like the MVC framework inside of ASP.NET Core. You have an action method that builds a model, and you pass that model to a view by returning some sort of view result. In a Razor Page, I would simply set that count as a property on my page and then bind to it from the cshtml file. So slightly different syntax, but a lot of the same separation of concerns here. You can almost think of this as the controller. The count is my model, and here's a View that is responsible for displaying that count. And the bigger picture here is I have this widget, this component, this service that I can use from a layout view. It can do its own data access using whatever services it needs. It could build its own model, and it can render its own view, which will plug into the layout view. We'll see how to do that next.

Rendering a ViewComponent

Now I need to create a view for my View component, Since I plan on using this View component for my layout page, that means this View component could be used from anywhere, any page in the application. So I'm going to create the view for the View component in the Shared folder. So ViewComponents have a very specific convention by default on how to locate views. What I need to do inside of this folder is add another folder called Components, and then inside of Components I need a folder that matches my ViewComponent name. So inside of Components, I want to add another folder, and this is going to be called a RestaurantCount. So I don't need the words ViewComponent on the end here, but I do need the folder name to match the first part of the view component name. If you've used ASP.NET MVC, the Model‑View‑Controller framework, this is very similar to how the framework finds a view for a specific controller. So now inside of this folder, I can place one or more views that are dedicated to my View component. If I'm not specifying the specific name of a view here, like now I want to render the count view and pass along the count model. If I'm not specifying the name of a view here, the name of my view will be default. So coming back into the folder, I want to right‑click and say that I want to add. And now, instead of going through Razor Page, let's go to New Item. What I went to find here is just a regular Razor View. Basically, I just want a cshtml file. I don't want a page model or a CS file behind this, and I did want to name this. I wanted to give this the name default.cshtml. I should have done that before creating it, but I'll simply rename it here. So now I have a view file. Let's erase everything that is in here, set up my model. What do I expect my View component to pass to me? As a model, I expect an integer. And once again, I will have access to an @model property that will allow me to use and present that information. Let's create a div, perhaps with some Bootstrap styling to make appear in a well. And I'm going to close the div tag, fighting with Visual Studio here just a bit. And let's say there are @Model restaurants here. And perhaps we could even include a link. So include a link to /Restaurants/List, and display the text, See them all. So now I have my ViewComponent class that uses the restaurantData service to build a model, which is count, and we tell the ASP.NET Core framework, we want to render the default view using this model, the count. ASP.NET Core should find this view. When it executes this view, it'll receive just this snippet of HTML. And now it's up to me to pick the location where I went this rendered. For rendering, it turns out that I can use a special tag helper to render this view component, but first I'll have to do what we talked about earlier, which is to go into my ViewImports file and say that I want to register some tag helpers from this assembly, so from my OdeToFood project, and I'm telling ASP.NET Core, load up that assembly and include all the tag helpers that you find. What that will allow me to do is come into the view or the page, wherever I want to use this view component, and I'm going to place this into the layout page, since we have to show this on every page. And let's go to the bottom, let's say inside of the footer, and I'm going to use a tag helper, vc:, and then include the name my view component. So a couple things here. First of all, this is the name of the view component without the two words view component at the end, so it's just restaurant‑count, and it's also what we would call Kebab cased. So instead of using an uppercase c here, if you have uppercase letters in your view component name, replace those with lower case letters and put a dash in front. I don't have to do that with the first letter, but restaurant‑count, all lowercase, will tell this tag helper, go out and find a view component named restaurant‑count. Execute this code and place the HTML that it produces right here. Now what if you have a view component that requires parameters? Perhaps I have a view component that's going to display the current weather. So one of the parameters that you have to pass in is the zip code. I can ask for a parameter here using invoke, and then I can actually pass that parameter by name, as part of this tag helper. So zip code or postal code and set that value and pass it in. With restaurant‑count, we don't need any parameters, but I'm going to save all my files. We're going to refresh the list page, and hopefully down here at the bottom of the page I can now see the message, There are 3 restaurants here. See them all! And that does seem to work. And that should appear on every page. So even the About page, which has nothing to do with restaurant, since this view component can go out and do its own data access and build whatever model it needs, I can use this view component from anywhere. And just to summarize, that's the primary difference between a view component and a partial view. A partial view always depends on model information coming from its parent view. A view component can be completely independent.

Scaffolding a Complete Set of CRUD Pages

Now that we've built out this application that is using the Entity Framework to talk to a database and we have views that will allow me to list restaurants, detail restaurants, edit, update, and delete restaurants, let me show you a feature that is available through the scaffolding tools on both the command line interface and here in Visual Studio, I just want to show you that we could have code‑generated most of this functionality just by setting up our entity, which is the restaurant and our DbContext. This can be useful for applications where you need to move really quickly and get a prototype in place for someone to use. What I'm going to do to demonstrate this feature is to add a new folder to the application. Let me just call it R2 for restaurants version 2. And I'm going to right‑click on this folder and say that I want to add a new scaffolded item. Once I click this, I get to select a template, and what I'm going to select his Razor Pages using Entity Framework CRUD, where CRUD stands for create, read, update, and delete. So what this scaffolding feature can do is implement all the same functionality that we've implemented in this course. And for some projects and some applications, it makes for a good starting point. But now that we've been through the process of building this ourselves, you should be able to understand the code easily. First, I specify a model class. The model is going to be my restaurant. I specify a data context. That's going to be my OdeToFood DbContext class. I don't want to create these as partial views, I do want to use a layout page, and I'm going to leave this blank because I do want to use the layout page that is specified in the viewstart file, and I'm going to click Add. This is going to dump four or five Razor Pages into this folder. And when it's complete, let's just open this up. Yes, I can see I can create a restaurant, delete a restaurant, get the details for a restaurant, edit a restaurant. So notice, create and update are two separate pages using the scaffolded approach. And then there's the index page, which is the default page for this folder that displays a list of all the known restaurants. In fact, we could go there if I just go to local host with the port that I need and go to just R2. When you go into a page's folder like this, ASP.NET Core will automatically look for a Razor Page named Index.cshtml. So in our implementation, if we had named the list Razor Page as Index.cshtml, we could reach here without specifying /restaurants, /index, or /list. We could go to just /restaurants. But here in the scaffolded code, you can see a table very much like the table that we built. I can click to edit this restaurant. If I go to the cuisine dropdown, notice this scaffolded approach sometimes is not perfect. You'll need to do some work. You'll need to do some customization. But if you're comfortable getting started with this approach and comfortable using a DbContext directly inside of your page, scaffolding out these views can be a good way to generate the boilerplate HTML that you're going to need.

Summary

in this module, we learned about layout views, partial views, and view components. Layout views control the structure of your application, while partial views and view components give you two different approaches to building reusable UI widgets in your application. In the next module, we'll continue thinking about the user experience and front‑end technology as we look at how to work with JavaScript and other client technology.

Integrating Client-side JavaScript and CSS

Introduction

Hi. This is Scott, and in this module, I want to give you some tips for working with client‑side libraries and ASP.NET Core. Specifically, I want to show you how to work with JavaScript library, it's like JQuery, and how to work with CSS frameworks like Bootstrap. We'll see how to manage and include these client assets into a project, and we'll also see how our application can support client‑side validation features. Along the way, I want to add some new capabilities to the application, including an API endpoint that we can use to fetch data using JavaScript in the browser. To get started, let's look at how ASP.NET Core serves static assets like JavaScript and CSS files to the browser.

Serving Static Files and Content from wwwroot

One topic we need to look at in this module is how do we serve up the static assets that we typically use for client‑side development, and by static assets, I mean those files that make up libraries like Bootstrap and jQuery. These are the JavaScript files, CSS files, images, fonts, and other assorted files that we typically have as part of our web project. By default, in ASP.NET Core, these files have to live in a special folder. Now, in ASP.NET Core, you can configure and reconfigure any type of behavior that you want, so it's not 100% required that these files live in this one special folder, but by default, the way ASP.NET Core is configured and set up is that static files can only be served from inside of this special wwwroot folder that is in the project. So, for example, if I have an image file, like this OdeToFood.png file here in the root of the web project, ASP.NET Core will not serve up this file, it will not have a successful response to a request that is asking for OdeToFood.png. So here in the layout page for the application, I have an image tag that is asking for OdeToFood.png from the root of the website, and if I save this and refresh the browser, I tried to place that image down here below the copyright and you can see I have a broken link. As far as ASP.NET Core is concerned, that file does not exist. And this behavior is good. This means that someone cannot request something like my startup.cs file or my appsettings.json file because ASP.NET Core is not going to serve up content that is not inside of wwwroot. But if I take this PNG file, OdeToFood.png, and I'm just going to click and drag it into the wwwroot folder, and by the way, you could download this file from GitHub if you want to try this also, but now OdeToFood.png is in the root of my website. As far as ASP.NET is concerned, if a request arrives for OdeToFood.png at the root of the website, ASP.NET Core will be able to find this file. It will also be able to find a favorite icon. And you'll also notice this is where we have other static assets stored, for example, JavaScript libraries like jQuery and jQuery Validation, as well as the JavaScript library for Bootstrap. This is where we could also store CSS files, like the CSS file for the OdeToFood site. So this is what I could open up and edit and modify if I wanted to change something like the default padding for the body of OdeToFood. But I want to show you that with OdeToFood.png now inside of wwwroot, I should now be able to refresh and see the layout page display OdeToFood at the bottom below the copyright, and that image request is successful. Here's another example. If we go to the homepage, and let's right‑click on one of these images and say open this image in a new tab, we'll see the image is actually an SVG file, banner2.svg, and that's at localhost: some port /images/banner2.svg. So I would now know if I want to modify this image to come into the wwwroot folder. There's my Images folder, and there, indeed, is banner2.svg. A little bit later in this module, I'm going to show you a trick that allows you to serve static files from other folders inside of your project and from the file system. But for right now, you just need to know that this is the default behavior. The wwwroot folder serves as the content root for your website. So, images and CSS files and JavaScript files, they all need to live somewhere inside of wwwroot. And if we want to take advantage of features like client‑side validation using jQuery Validation, jQuery and these other script files need to be loaded from these folders inside of wwwroot. Let's see how we do that in this ASP.NET Core project.

Using ASP.NET Core Environments

I just want to point out that here on the homepage for OdeToFood, we have this Bootstrap carousel that rotates images and rotates panels as we sit here and look at it, and in order for this to work and in order for me to be able to click on this to manually rotate a panel, this requires some JavaScript to be in place, both jQuery and Bootstrap.js. So the question is, where and how do these JavaScript files get loaded by the project? Well, first of all, I do want to point out, again, that inside of wwwroot, there is a lib folder, which is where the jQuery library and the Bootstrap library and all the JavaScript that I need, this is where it all lives. So somewhere in my project, I need to load these JavaScript and CSS files from the wwwroot folder. And the ideal place to do that would be from the layout page of the application. So we've already looked at and talked about the layout page. Every other Razor Page renders inside of this page essentially, so if we look here at the top of the layout page, you'll notice inside of the head tag we have links to bring in the Bootstrap CSS Framework, as well as our own custom site.css file. Now those two links are wrapped inside of an environment tag, which I'll come back and talk about in just a minute. But before we talk about that, let's scroll to the bottom of the layout page where I can show you this is where we have script tags to load jQuery, to load Bootstrap.js, and to load our own custom site.js file. So this layout page will include in every other Razor Page all of the CSS and all of the JavaScript that we need. It will load these files starting at the root of the website, which, again, means we're going to have to place those files into wwwroot. Now, as far as this environment tag goes, environment is a special kind of tag helper in ASP.NET Core. So, we've also looked at and talked about tag helpers previously in the course. We've looked at tag helpers like asp‑page to help generate a link, asp‑for to help generate forms. Environment is just a tag helper that, instead of looking like an HTML attribute, it looks like an HTML tag, but it is processed on the server, and what this tag helper will do is look at the current operating environment and will only emit the HTML inside of here if a condition is met. In this case, the condition is that the environment would have to be development. Down here, this next environment tag is what happens if we have an environment that is not development. So with ASP.NET Core, you can set up as many environments as you like. You might have an environment for development and one for production. You could also have one for testing and one for staging, but typically the big difference is between development and production. In development, you might have, but typically the big difference is between development and everything else. When you're in a development environment, you want to see stack traces and detailed error messages. When you're in production, you want things to operate fast and you don't want to show users in production detailed error messages that might leak information. So what we have here is a situation where, when we're in development, we're going to serve up our JavaScript and CSS files from files that are on the web server and locally on our file system, but when we're not in development, so we're in production or some other non‑development environment, we're actually going to try to serve those same files like jQuery and Bootstrap from a content delivery network. So typically the content delivery network will serve up those files with a little less latency and be a little bit faster and hopefully cached for your end user. And this is all just done with the environment tag, which will render or exclude the inner HTML based on this condition. Now, one of the questions you might have is, when I run this application from Visual Studio, how do I know what environment I'm operating in? Well, by default you will be in a development environment. You can also explicitly open up in the Properties section of your project this launchsettings.json file. This is a file that can be used from the command line with.NET run, as well as with Visual Studio, and this file controls some of the different environments that you can operate in. By default, when I run here inside of Visual Studio, Visual Studio is going to launch my ASP.NET Core application and run it inside of IISExpress, and when it does that with the IISExpress profile, by default, it's going to set an environment variable for that process and set this ASPNETCORE\_ENVIRONMENT variable to Development. This is the variable that controls your environment. If this variable is missing or not present, ASP.NET Core assumes a production environment, which means you might not receive the detailed error messages that you want, but if I want to see how my site would render in production, I could change this value to production, then save the file and relaunch the website with CTRL+F5. And the other way to launch this website, this second profile down here is essentially doing a dotnet run, and you can see by default we'll also set up an environment of Development for that scenario, too. So now that we know a little bit about how scripts get loaded and how we can use environment tags to switch which scripts are loaded between development and production, let's see how we can take advantage of some additional scripts to add features to our application, like client‑side validation.

Enforcing Validation on the Client

Currently in the application, if I go into edit an existing restaurant or add a new restaurant, I do have validation checks in place, so if a user tries to create a restaurant with no name and no location, we will show some error messages that say the name is required, the location is required, but these validation rules only work on the server. In other words, when I type into the restaurant name input, there's nothing that updates on the browser to now know that the name is here. This is passing validation. Instead, I have to go back to the server and send that value back, and now we have that error message cleared. And this is unfortunate only because, as part of my project by default, I'm already set up with jquery‑validation and jquery‑validation‑unobtrusive included in the project. jquery‑validation is a jQuery plugin that provides client‑side validation, and jquery‑validation‑unobtrusive is a validation plugin authored by Microsoft that ties together ASP.NET Core validation with jquery‑validation. It happens through the magic of the asp‑for tag helper. Essentially, if I look at the markup that is rendered, essentially, if I look at the markup that is rendered for an input like location, I'll see some additional HTML 5 attributes like data‑val = true and data‑val‑required, what jquery‑validation‑unobtrusive will do is come through and find these attributes, and feed these rules into the jquery‑validation plugin so that I can have true client‑side immediate validation of my inputs. Unfortunately, those scripts are not included by default; however, they're very easy to add. So one approach to adding them would be to go into my Layout page and just say that I want to enable client‑side validation everywhere, and what I could do is in addition to including JQuery and Bootstrap.js, I could go ahead and include jquery‑validation and jquery‑validation‑unobtrusive, those script files. However, there's another file in this project, which is in the shared folder, which is the ValidationScriptsPartial.cshtml file in the shared folder that already has all the markup that I need to include these scripts for client‑side validation. So here's a link to jquery‑validate and here's a link to jquery‑validation‑unobtrusive, and this is done using the same pattern that we saw before. So when we're in development, we will serve those files up from the local web server; otherwise, if we're not in development, we will serve those files from the ASP.NET Content Delivery Network. So as the name of this view implies, this is intended to be a partial view that I can render from inside of another view where I need this client‑side validation support. So, yes, I could render this from the Layout page and just include these scripts everywhere in the application, or I could try to be very efficient and say I only need this validation support when I am editing or adding a new restaurant, which for us both happens from the same page, which is Edit.cshtml. Then the question would be how and when do I render this from my Edit.cshtml page? Well, you might remember previously, when we were working in the Layout page, we talked about the Layout page having different sections that you can use to define different render areas for your views. So we had to find a section named footer that allows some other Razor page to inject content into this specific location inside of the Layout page. Well, there's another section defined at the bottom here, right before the closing body tag, named Scripts. So if I have any Razor pages in my system that want to inject their own custom scripts and have those script tags appear at the bottom of the page, then one way to do that is to render those scripts inside of a Scripts section. So let's go to Edit.cshtml, and let's say that we want to render a section called Scripts, so anything that I place inside of here will appear at the bottom of the page. But what I want to do is render a partial. So again there is a tag helper for that. In this case, the tag helper, much like environment, is a full tag. That's the tag partial, and I just need to provide the name of the partial that I want to render, in this case it's going to be validationScriptsPartial, and then close off my partial tag. So what we should do when we render our Edit page is inject into the Layout page in the Script section this partial view, so ASP.NET Core will be able to go out and find that partial view, and just take this markup, and place it where I have a call to partial. So now if I come back to the Edit page, let's issue a get request and reload this form, and let me click Save, and yes, I get the error message that the name field is required, but now when I start typing into that input, the client‑side validation is active, it's detected that there is something that exists in the input, so it gets rid of that error message. If I change this and start removing the characters that I've typed, now I'm back to an empty input and the name field is required. So what we're seeing is a couple of things. Inside of this ASP.NET Core project, we are seeing how to provide client‑side validation. It's all about the asp‑for tag helper working on the server side in conjunction with jquery‑validation on the client side. But We're also more generally learning how to work with scripts and other static assets from our ASP.NET Core project. So now what I want to do is take a look at a slightly more advanced scenario, one where we want to call from the client to an API endpoint on the server to retrieve some data and use some JavaScript to format and display that data.

Loading Restaurants from the Client

To show how we're going to work with an API from the client side, I want to create a brand‑new Razor Page, just to simplify things and keep this functionality in a separate page so it's easier to find. So right‑click on the Restaurants folder. I want to Add a Razor Page. I'll just use the default template here, and I want to add a page that has the name ClientRestaurants. I'll leave the defaults here in place. I do want to generate a page model. I do want to use the layout page. I'm just going to click Add. And while this is loading, I just want to point out that we could do this using Angular or using React or using Aurelia or using Vue.js, any number of different client technologies. I'm going to try to keep things very simple and just use jQuery and a jQuery plug‑in. But what I want to demonstrate are the same concepts that you had used with these other frameworks. So, how to build an API endpoint inside of ASP.NET Core, how to invoke that API from JavaScript, and use the endpoint to fetch data in JSON format. That's what ClientRestaurant is going to do. So we're not going to do any server processing. I'm not going to need an OnGet method here in my page model. I'm really not going to need a page model at all. Instead, what I want to do is come into the markup into the CSHTML file and do something that typically I wouldn't recommend. What I want to do is go ahead and say that I want to render into the Scripts section of the layout page an in‑line script that I'm going to include here in the page. So typically, I recommend that you add all your JavaScript to external.js files because those files are typically easier then to author and to test and to maintain, and they're also easier to bundle and optimize, whereas embedding script into your Razor Pages, if you do that too much, everything gets a little bit messy inside of the project. But I'm going to do this for this demo just to keep things as simple as possible. What I want to do is write a script that assumes we have jQuery loaded on the page, and I know we do, because the layout page will include jQuery for me. So let's write a function that will be invoked by jQuery when the DOM is ready. And ultimately what I want to do is use jQuery to go out and make an Ajax call to an API endpoint that we will build. So let's assume in the future that I will be able to come into this server and go to /api/restaurants, and that will be an API endpoint that I can go to which will return all the restaurants to me in JSON format. We can also build an API where /api/restaurant/3 would bring back the single restaurant with an ID of 3, but going to /restaurants will bring back all the restaurants. And with the Ajax settings object, I'm going to specify that this request should be sent with an HTTP method of get because we're not, again, updating data, we're just reading data. And if you need a refresher on using jQuery Ajax methods, I'm sure there's many courses on Pluralsight that will cover that topic, but this method returns a promise. So with a promise, I can say .then, execute the following function. I expect the response to that API call to be passed to this function. But for right now, the question is, what do I do with this response? And this response, when successful, will contain a lot of restaurant information, but the question is for right now, what do I do with this response? Well, ultimately, I want to nicely format that response into a table that a user can view. But for right now, to just see what's happening, let's log that response in the browser developer tools. And let's go ahead, before we even build our API, and see how this page behaves. So I want to go to /Restaurants/clientrestaurants, and let's go ahead and open up the developer tools, which I could do with CTRL+SHIFT+I, or from the menu here, and we're going to go look at the console. And what I can see here is an expected error, which is the server responded with a status of 404 Not Found, and that was on a call to /api/restaurant. So the next order of business will be for us to build an API endpoint that will respond to this request.

Implementing an API Controller

Creating an API is relatively simple. In fact, with ASP.NET Core, the scaffolding tools will help me create this API. What I'm going to do is add another folder to this project, a folder with the name of Api. I want to keep this separate from the Razor Pages and the rest of the application, and I do want an uppercase A in the Api. And inside of that folder, what I want to do is add a controller. So I believe that the best way to generate server‑side HTML is to use Razor Pages. That's the superior technology if you're rendering HTML from the server and you're receiving form posts from the client. However, when it comes time to build an API, an API that returns text or XML or JSON, the clearly superior technology for that is going to be to use controllers from the ASP.NET Core Model‑View‑Controller framework. And what I'm going to do is use the scaffolding tools right now inside of Visual Studio, but you can also do this from the command line. I'm going to use the scaffolding tools to say, yes, go ahead and build me in API controller with actions using the Entity Framework. So this will be a controller that has the ability to create, read, update, and delete restaurant data. Just like previously in the course, I showed you how you could scaffold out Razor Pages that give you CRUD functionality for a given entity. This time we're going to do it, but for an API. So my model is going to be my OdeToFood.Core restaurant. My data context class is going to be the OdeToFoodDbContext that we created earlier. And the controller that I want to create will be the RestaurantsController, that's going to be the name of the controller. I'm going to clic Add, and we'll take a look at this code and see how this behaves here in just a moment. But in ASP.NET Core, a controller and a Razor Page, they're very conceptually similar. I can use either a controller or a Razor Page to respond to an HTTP request. It's just that Razor Pages are better suited to produce an HTML response, whereas MVC controllers, they could also respond to HTTP ,but they are better suited for building APIs that return JSON or XML. All the controller is is a class that derives from a base class controller base. Just like a Razor Page, a controller can receive dependencies via dependency injection. So you can see we are injecting the Entity Framework DbContext into this controller, once the MVC framework has determined that this controller is needed and has to be instantiated. And at that point, very similar to Razor Pages. I'm going to have methods, which we will refer to as actions, that respond to different types of HTTP requests. So if I have a GET request for /api/Restaurants, which is what we're trying to do from jQuery, that's going to invoke this method, GetRestaurants, which will return all the restaurants that are in my DbContext. So for right now, we're not going to worry about paging and sorting, we're going to imagine that there's a manageable number of restaurants in the database. Down here's another method. This method is going to respond to a request like /api/Restaurants/5 and return a single restaurant that matches this by ID, so the restaurant with an ID of 5. The code is going to be very similar to the code that we wrote for the Razor Page that shows the details for a single restaurant. I also have methods that will respond to an HTTP PUT request to put a restaurant, that is update an existing restaurant. There will also be a method that responds to an HTTP POST request. When building an API, a POST request is typically what we use to create a new resource, create an entity, so this is going to create a new restaurant. This is slightly different than forms programming, where we use a POST request to do both create and update. When you design an API, you typically make more use of HTTP methods and HTTP status codes. So there's other courses here on Pluralsight that we'll go through in more detail, building APIs with ASP.NET Core. This is just a quick introduction, and I just want to demonstrate that indeed, if I go to /api/Restaurants, I should be able to see restaurant information even if I do that from my browser. So let me open up another tab. And now I went to copy /api/Restaurants. Let's go to that here on the server. And indeed, I can see all my restaurant information being returned, but as a JSON array. And if I go to /api/Restaurants/2, I will see just a single restaurant, the restaurant that has an ID of 2. So our API endpoint appears to be working correctly. And let's just see what happens now if we come back to our client restaurants page and do a refresh. You might remember, inside of the then callback for my Ajax call, I did a console.dir on the response that came back from the Web server, and indeed, here in the browser, I can see we're logging out the three restaurants that came back as the result of an API call. So now I know that the code inside of ClientRestaurants, or the JavaScript code, is working correctly. I'm calling the API. I'm getting back some JSON data. The JSON data is legible. Now we just need to figure out how to format that data into something that the end user would like to see.

Using DataTables

Now that we have data flowing back from the API endpoint in our application, it's time to take this raw data and place it into the UI. There's many different approaches that you can use to do this. One approach would be to use a fully featured framework like Angular, or View, or React; but I'll leave those topics for other Pluralsight courses. What I'm going to build on top of is a relatively simple jQuery plugin named DataTables. So here I am on datatables.net. On this page, you can read more about DataTables and the different features and how easy it is to use, but ultimately, this is a jQuery plugin, so this is going to be JavaScript code that relies on jQuery already being present and loaded into the browser; and now the question is, how can I take the DataTable script and add them to my application? There's also several different approaches that you can use to answer this question, but right now let's go to the download page, and I do just want to point out that there's different versions of DataTables that will integrate and work with other frameworks, like Bootstrap 3. So this is a jQuery plugin, and the use of jQuery is required, but you can optionally also borrow some of the styling and themes from frameworks like Bootstrap 3 or 4. I'm going to select Bootstrap 3 since my application is built on top of Bootstrap 3, and what I'm doing on this download page is setting the options to customize my download. So, for example, if I also wanted to download jQuery, I could select that here, I'm going to select just DataTables, I'm not going to add the ability to fully edit the controls in my DataTables, and I'm not going to select one of the many extensions that is available for DataTables. Instead, I'm going to scroll down to where I can pick a download method, and here we can see a few different options. So one approach is I could go to download here, and then click on the download files button and manually extract and copy the files that I need into my project into the wwwroot folder, of course, so I could serve these static files. Another approach, by the way, for Visual Studio users, is to use a relatively new feature in Visual Studio 2017; what I could do is right‑click on my wwwroot folder and say that I want to add, and then select Client‑Side‑Library. So in the past, Visual Studio and ASP.NET have relied on different technologies to download and install client‑side libraries, like Bootstrap and jQuery. In the not so distant past, one of those technologies was a package manager named Bower. Unfortunately, Bower is not so well maintained anymore, the JavaScript world has moved on to use other package managers, and so this client‑side library feature in Visual Studio 2017 addresses that issue for those users that don't want to use some external package manager, they just want to use Visual Studio, and what I can do is say that I want to go out and find the library named datatables, and there's all sorts of libraries available through this extension, this client‑side library extension in Visual Studio, you can get to Bootstrap and jQuery, and as we can see here datatables. So I want to download datatables version 1.10.19. I can select all the files that I want or I can select individual files, and Visual Studio will copy and install these files into wwwroot/datatables. So again, one approach is to download from the website, extract files and copy files into my wwwroot folder. Another approach is to use Visual Studio to essentially take those same steps for me, but ultimately what I want are the datatables JavaScript files somewhere inside of my wwwroot folder so I can serve those as static assets. What I want to do is show you another approach. The first approach I'll show you is how to take the link and the script tags that are generated here for me, and use these inside of my application to serve up datatables directly from a content delivery network, and what I want to do is make sure that I select the CDN option so the download option assumes that you've installed the correct files into the root of your website or into a folder called DataTables, that's not our case, what we want to do is use these files directly from a content delivery network, which is datatables.net. So I'm going to take this little bit of HTML, and copy that HTML and bring it back to my project, and now the question is, where do I want to install those? If I was using datatables throughout the application on a number of pages, I would probably add this markup into my layout page. That way, the datatables library and style sheets are available everywhere throughout the application, no additional work is required. If, however, I have a situation like what I currently have, where datatables will only be used on one specific page in the application or a small number of pages, what I might consider is just adding that markup to the scripts section of the one page or the several pages that are going to use datatables. That way I'm not sending the script files down to the client's browser, just for the client to view pages that aren't going to use datatables. But on the other hand, when the client does come to a page that is going to use datatables, all of a sudden the browser is going to need to make these two downloads. Now, when I do this, I do want to do something very similar to what is in the layout page, which is I might only want to load these files from a CDN when I'm in my production environment. So we will have one environment tag that will control what happens when I'm in development mode, and then another environment tag that will exclude development, and in other words, be the environment and the markup that will render when I'm in production and staging in every environment that is not development. So when I'm running in production, this environment tag will contain the link and the script that I need for datatables to be loaded into the client's browser. The question is, what do I do during development? Perhaps I still want to serve up these files from a local source, which means I still need to get these files onto my local file system, and the question is, what's the best approach to do that?

Managing Client Libraries Using npm and NodeJS

Now, again, our goal is in addition to serving data tables from the CDN when we're in production, I want the ability to have data tables locally on my web server's local file system when I'm in development. This might make debugging easier, it also might be that I want to bundle and minify data tables and jQuery, and all the other script files that I have together into a single file for the client to download, and in order to do that, I don't want to have these individual downloads from different CDN's, I want everything locally so I can use some different tools for bundling and minification. So the question is, how do I get data tables locally? Well one approach is certainly you can download a zip file and extract that zip file, and copy files around on your file system. Another approach is to use that client library manager that is now available inside of Visual Studio, but I want to show you the approach that I have been using for the last couple years. So coming back to datatables.net, when we're on the download page, we can actually see several different options here to use data tables from a CDN to download a file directly, and then there's a series of package managers. So Bower might be the package manager that you're most familiar with if you've been using Visual Studio for a number of years, because Visual Studio and ASP.NET MVC used to use the Bower package manager to deliver client side libraries like jQuery and Bootstrap; however, we've moved on past Bower. These days I would strongly urge you to consider using Yarn or NPM. These are both package managers that are typically associated with Node.js, and in order to use NPM or Yarn, you'll have to have Node.js installed on your computer. But I will tell you that NPM in particular, years ago used to be the package manager that I would use to download modules and packages to consume from a Node.js application, but over the last couple of years, NPM has expanded its scope, it now includes many client side libraries like Bootstrap and jQuery, as well as things like Angular and React. So you can use NPM to install nearly anything JavaScript or CSS related that you might need for the client or the server, and what I want to show you in this module is how to use NPM to install data tables, and what I want to show you in this module is how to use NPM to install data tables locally, and then I'm going to show you how to serve those files from the local file system when you're in development mode. So the first thing you're going to need to do if you're going to use NPM is make sure that you have NPM installed, which does require you to install Node.js, and I'm not going to go into a lot of detail about how to use Node and NPM, you'll find those topics covered in other Pluralsight courses. But if I type node and I enter into a prompt like you see here, then I can be relatively confident that I have Node.js installed, and now I can enter JavaScript expressions, like 2 + 2, and Node.js is really just a JavaScript execution engine, so it's going to evaluate that expression and show me the result 4. So I'm going to hit Ctrl+C to cancel, and now that I know I have Node.js installed, I want to try my luck with NPM, that should also be in my path if Node.js installed successfully. So when I run NPM, which is the Node package manager, I can see a number of different commands that I can execute, but ultimately what I want to do with NPM is use NPM to install data tables, and also use NPM to keep track of what is installed inside of this project, so other developers, when they download or clone my source code from a repository, they'll be able to run NPM and install the same set of packages that I'm using. In order to do that, NPM relies on a special file named package.json, and I can create that file by typing npm init. This will initialize a new package.json file in the same folder, so I want to be in the same folder as the root of my web project, so at the same level as odetofood.csproj that is inside of this folder, and now I'll receive a series of prompts asking me what I want to place into this package.json file. I'm just going to take all the defaults. So the default package name is the same as the folder name that I'm in, which is odetofood, and that's fine. I'll take the default version, and description and entry point, I'm just going to press Enter until we get to the end, and I should now have in this folder with my csproj file, a brand new file named package.json. So the way things typically work with NPM is that you will check into source control your package.json file, and this file will contain a list of all the dependencies that your application has, that is, the different modules and packages that you need installed with NPM. When other developers bring this repository down, they can run a command, which is npm install. What npm install will do is look in this package.json file to see what's needed, and go out and download all the files that are needed to satisfy the dependencies in package.json. But the first step in this would be telling NPM that I need data tables and that I need to take a dependency on data tables, and that's actually the command that is given to me on datatables.net when I go to the NPM tab here, under Pick a download method. It's going to give me the command that I need to execute with NPM to install datatables.net with Bootstrap support, and save that or list that as a dependency of my project. So I'm going to copy that command, bring it back to the command prompt, paste it in, press Enter. What NPM will do is go out and download datatables.net and install that into a folder here named node\_modules. So if I open up this folder in Explorer, I'll be able to see node\_modules, and inside of node\_modules, there are files related to datatables.net, that would be the core functionality. Here's datatables.net, the Bootstrap support that is needed so additional JavaScript and CSS files there, and because datatables has a dependency on jQuery, and datatables tells NPM about that, NPM also download jQuery for me. So I could be loading and using jQuery from this folder instead of my www.rootfolder if I wanted to. Let's just swing over to Visual Studio real quick because I want to show you inside of our project, two things; first of all, here's the package.json file. This is a file that I would add to source control. I'm just going to clean things up in here a little bit. A lot of this information is not information that is required, so everything from license to description, I'm just going to remove those lines and leave in just the project name, the version, and then here are the dependencies for this project. So this is what NPM looks at to see, oh, you need datatables.net‑bs. Let me go out and download that if it's not already in the Node modules folder. So typically, people do not check node\_modules into source control, they only check in package.json, and any time you come out to the command line and run npm install in a folder with the package.json file, it's going to look at the dependencies and download them, and place them into node\_modules if they're not there. But now, I have the JavaScript and CSS files required to use data tables on the file system, how can I now reference them from my project, specifically inside of this environment that is set up to load HTML for the development environment?

Managing Production Scripts and Development Scripts

Thanks to npm, we now have all the files that we need for data tables with bootstrap support on the local file system. However, I do want to point out if you come into the Solution Explorer in Visual Studio, you won't be able to find the node\_modules folder here. That's because Visual Studio will hide that folder by default. It considers it sort of an infrastructure folder, kind of like the bin folder and the obj folder. It's full of files that can be recreated using an npm install, just like you can recreate the obj folder by doing a build in Visual Studio. So Visual Studio doesn't show node\_modules by the fault, but it can be useful to have that folder in your Solution Explorer, and one way to have that folder appear is to come up here in the Action bar for Solution Explorer and click on the icon that will show all files. Now I can see these hidden folders, like the obj folder and node\_modules. And one of the reasons why I want to make this folder visible is because it provides an easy drag‑and‑drop approach to adding the links that I need into my HTML markup. So if I want to add links to serve up data tables during development time, what I can do is go into the datatables.net folder inside of node\_modules, find the js folder, and this is going to be the unminified core version of datatables. So if I take that file and drag it over here, I now have a script whose source is set to node\_modules, datatables.net/js, then it's jquery.datatables.js. Perfect. Now again, we cannot serve files from the node\_modules folder by default in ASP.NET Core. But I'll show you in the next clip how to get around that limitation. I'm also going to open up the datatables.net‑bs folder, which is the bootstrap support for datatables, and I need two files from here. First of all, there's the CSS file. I'll take the unminified version and drop it into my page. That gives me a link for the CSS file. I also need the JavaScript file, which is dataTables.bootstrap. I want to add that after the core js file and after jQuery, but the layout page will take care of that for me. The scripts section comes after the jQuery script reference. But I'm going to drag dataTables.bootstrap.js over, and I now have the two scripts and the one CSS file included for my development environment. They're going to be coming from the node\_modules folder. Now if I want to make this a little more permanent, I can right‑click on node\_modules and say please include this folder in my project. That means even when I do not have Show All Files selected, this folder is going to appear. And sometimes this also helps with Visual Studio when it's trying to provide IntelliSense for JavaScript and CSS files that you're using from the node\_modules folder. So if you want IntelliSense for datatables and for the bootstrap functionality and anything else that you've npm installed, just right‑click node\_modules and add it to the project. Now at this point, I do want to point out our files are a bit scattered around on the file system. The file new project template for .NET Core placed jQuery and bootstrap into folders in the wwwroot folder. I'm now using npm and installing additional script files and CSS files into node\_modules. If this was a proper project that I was working on for a team, I would want to clean things up, and I would want to switch everything over to using npm. So I would delete libraries and CSS frameworks from the wwwroot folder. And instead, I would npm install jQuery and save it as a dependency and npm install bootstrap and save it as a dependency and use npm to manage all the client assets that I use inside of this application instead of having some in wwwroot and some in node\_modules. I'll clean that up after I'm finished recording this module. So if you want to try it on your own and compare the results, you can do that. But right now, let's just see if we can get the application working in a development environment and see if we can get data about our restaurants on the page inside of a data table.

Serving Files from the node\_modules Directory

As I mentioned earlier in this module, the default settings created by the project template that we used to create OdeToFood will only allow static files to be served from the file system if they live inside of this wwwroot folder, but this is just the behavior that is configured into this application. If you do not want to serve any static files, you can remove this capability from ASP.NET Core. If you want to rename the wwwroot folder, you can configure ASP.NET Core to use a different folder name. If you want to add additional folders for static assets, you can also do that. And me, as I said, I've been using MPM to manage my client‑side and frontend dependencies for a number of years. And what I've done in the past is create a NuGet package that makes it very easy to serve up static assets that live inside of the node\_modules folder in an ASP.NET Core application. So what I'm about to do is add a reference to a NuGet package that will allow me to serve files from node\_modules. We also have to make one little code change in the application. But first of all, let's right‑click OdeToFood and say that I want to manage NuGet packages. What I want to do is Browse for odetocode, and use this package, which is OdeToCode.UseNodeModules. Again, you can add this reference by editing the CS project file directly or by using the .NET command line interface, but once this package is installed, I'm going to show you the one quick change that we have to make to this project. What I need to do once this is installed is come into the Startup file for my application, and we haven't talked about the Startup fall in great detail, we have done some work inside of ConfigureServices, but in the next module, I also want to describe to you in more detail what this Configure method will do because it is this Configure method that is very important to controlling the behavior of my ASP.NET Core application, including if I'm going to be using the wwwroot folder or not because it is Configure where I can install HTTP processing components known as middleware in ASP.NET Core. So these are pieces of software that I install into a processing pipeline, so that any time an HTTP request arrives at my ASP.NET Core application, it has to run through a series of these components that figure out what to do with this request. One of those components is known as the static files component or the static files middleware, and what that particular piece of middleware will do is look at an incoming request and say, hmm, this is a request for jQuery.js, can I find that file somewhere on the file system inside of the folder where we're serving static files, and if it finds that file, it will serve and return that file from the web server. What I want to do has had another piece of middleware that, just like the static files middleware, will look on the file system for a specific file, but this time instead of inside of going to the wwwroot folder, it's going to go to the node\_modules folder. And that piece of middleware, that component is something that I put together in the OdeToCode.UseNodeModules NuGet package, and the name of the middleware or the name of the extension method that I used to install that middleware into the processing pipeline for ASP.NET core, UseNodeModules, I call that on the IApplicationBuilder interface, which is passed to me. Now, this particular extension method, UseNodeModules, does require a parameter. It requires a parameter of type IHostingEnvironment, which will provide the middleware with information about the environment it is operating in. And fortunately, ASP.NET Core passes an IHostingEnvironment into the Configure method. So what I can do is take this variable env, which will hold the hosting environment, and pass that along and through to UseNodeModules. So this is my processing pipeline. Again, more details about this in the next module, but this is going to allow my application to serve static files from wwwroot, it's also going to allow my application to serve static files on the NodeModules folder, and it's also going to support a cookie policy, where we're going to display something in the user interface, telling the user that we use cookies and asking them to accept the use of those cookies before we proceed. And this middleware is also going to set us up to use the model‑view‑controller framework, which will include Razor pages, and API controllers, and all these other components that we route to inside of the application. But with all these pieces in place, I should be able to come to the Client Restaurants page and be able to load this page with all of these new scripts, including data tables, and not experience any 404 errors, and it looks like we're good there. There's no errors appearing in the developer tools. So at this point, we're ready to come into Client Restaurants, and now finally use this data tables plugin to show the data that we have retrieved from the API endpoint.

Creating Sortable, Searchable Data Grids with DataTables

Data tables has an easy‑to‑use API that also provides a lot of flexibility and the ability to customize a lot of settings if you need to. What I'm going to show you is a very simple approach to taking some data that you might get back from a JSON API call and place that data into a nicely formatted table. In order to do that, we're going to need a table. So let's create a table element that has a Bootstrap class of table, and also give this an id, of, let's say, restaurants. That's all the work I need to do with that table. Data tables will take care of the rest. So I copied the id of this table because down here, once I receive the response, what I want to do is use a jQuery selector to go out and find that element by id, so the element with the id of restaurants, and invoke the dataTable extension. When I do that, there's different options that I can pass in with an options object here. What I'm going to do is pass in an object, and first of all, say that the data you want to use to populate this table is this response object. And then I can also specify the exact columns that I want to use. I do that by passing some sub objects. Every sub object will have a key of data where I can specify the name of the attribute on the response that contains the data that I want, so I want you to take the name of each restaurant and use that in the table. I also want to display the location, so let's add location. And I also want to specify the cuisine for each restaurant. And with those settings in place, let me save ClientRestaurants.cshtml, and let's come back out to the browser and see what happens when I do a refresh. And I can see the three restaurants that I have in the database. Data tables has formatted that information into a table. It has also provided the ability to page information if there was a lot of restaurants. I also have the ability to search my data table, so I can search and see just the restaurants, for example, in London. And I also have the ability to sort. So if I wanted to place this into a specific sorting order, like ascending order with a name, this is all just core functionality provided out‑of‑the‑box by data tables without me setting any additional options. But I do want to show you a little bit about what it's like to customize data tables. So notice my cuisine is not showing up as a user friendly display; instead it's showing an enum value directly from the database. So what is 3? Three is really Indian cuisine, but data tables is not going to be able to display the associated text for that value without me doing some additional customization and additional work. Now, I do want to let you know that one way, the proper way to do this, perhaps, would be to serialize all the available cuisines into JSON format and send that information also to the client. What I'm going to do for right now, though, is, just to get this to work, is to hard code this value. So I'm going to say that cuisines is an array where a value of 0 would be the unknown cuisine, and I happen to know that the value of 1 in the database would be Mexican, then there is Italian, then there was Indian. You might remember this from the beginning of the course. So now I have my values. Again, you might make an API call to get these values instead of hard coding them into your script. But once I have these values, one of the things I can do with data tables is for any given column, provide a custom render function. So I'm going to add a value here, render, and say that this is a function that will take a piece of data, what data tables will do is pass in the raw value that it's looking at, which is going to be a 0, 1 or a 2, whatever that enum value is, but I will say let's take that data point and let's return cuisines sub data. So what this will do is take the value 2 and turn it into the text Italian. Let me add the one additional closing curly that I needed to complete this object. Let's save this file, come out to the browser and refresh, and now, instead of displaying raw numeric values, we do display the text of the cuisine, Indian, Italian, Italian. This is just an example of how easy data tables can be. But the bigger picture here is that it's also easy to build API endpoints using ASP.NET Core, call those API endpoints using jQuery or Angular or any JavaScript Framework that you want to use, even the built‑in Fetch API, and then process that JSON data on the client side.

Summary

In this module, we took a closer look at how to use some client‑side technologies with ASP.NET Core. We used the Node Package Manager, npm, to install some frameworks and jQuery extensions that we want to use. And we also saw how to install a custom piece of middleware to serve static files directly from the node\_modules folder that an npm install will create for us. Finally, we got to see how to build an API endpoint using a controller with the ASP.NET Core MVC framework and how easy it is to process JSON data from the browser. In the next module, we're going to take a closer look at the middleware that we talked about in this module. And we're going to take a look at some of the internals of how ASP.NET Core really works and how it processes HTTP requests.

Working with the Internals of ASP.NET Core

Introduction

Hi, this is Scott, and in this module, I want to show you some internals, and demonstrate how ASP.NET Core works behind the scenes. Having some knowledge of the internals is extremely helpful when you need to troubleshoot problems in an application, and this knowledge also gives you more confidence when using ASP.NET Core. We'll talk about request processing with middleware, how an ASP.NET Core program gets started, look up more details of the configuration system, and see how to log diagnostic information. We'll get started at the start of our application in program.cs.

Exploring the Application Entry Point

Let's learn a bit about the internals of ASP.NET Core by starting at the very beginning. How does OdeToFood, this project, this application, how does it launch and how does it execute? If you've been programming .NET for any amount of time before ASP.NET Core, then you might know that there is a convention in .NET if you want to write a command line application, that is an application that you can run from the command line. That convention is to have somewhere in your project, a class that has a mean method, a public method that is static, and the typical convention is to place the static mean method inside of a class called Program. You can change this behavior using some compiler switches, but this is the convention that you will see most .NET developers follow. So our OdeToFood application, which is really going to be running on a web server when it's in production, we can think of it as an application that we can run from the command line because it has what we call an entry point. The static mean method is the entry point, that's where execution will start when the OdeToFood process starts running. To run OdeToFood from the command line, I can use the dotnet run command. When I execute this command, and I don't provide any additional arguments, dotnet run is going to look in the current folder or a project file and say, oh, you want to run this project, let me build this project, launch a process, and look for the entry point to start your logic executing. And when I press Enter to execute, this is very much like Java, and Python, and Ruby, and Node.js, if I want to execute something from the command line, I go through a launching process like Java, Python, Node, or .NET to start my application logic running. And you can see a few things are happening, dotnet run is going to look up launch settings from launchSetting.json, we talked about that file a little bit previously, and then it's going to start this application running, and this application really wants to be a web server. It wants to listen for connections and process HTTP messages. In this case, my application is up and running and listening on 2 different ports, port 5001 for a TLS secure connection and port 5000 for regular HTTP traffic. And now if I come over to a browser, which I already have open to localhost port 5001/Restaurants/List, I can now see a list of all my restaurants. And coming back to the command line, I can see my OdeToFood application has some output here. These are logging statements that give me a little bit more information about what's happening, for example, a route matched with the page /Restaurants/List, I can see the Entity Framework dumping out some of the SQL queries that are executing, for example, SELECT Id, Cuisine, Location, and Name from Restaurants, as well as a SELECT COUNT from Restaurants. And a couple questions that would come to mind are how is this logging established, how can I change the logging if I want to? Those are some of the topics that we're going to look at in this module. So first of all, let's just come back and remember that launchSettings.json is influencing how my application is behaving, specifically, it's setting ports here. So if we come back into Visual Studio, and go to the Properties folder, and look at launchSettings, yes, I can see a section for iisSettings, that's not what we're doing when we're running from the command line. I can also see a section for configuring how to run this under IISExpress, which is what happens by default in Visual Studio when I run this project. But this third section here, the OdeToFood section, which is what would happen if I made this selection in Visual Studio, this is effectively the section that is used when I use a .NET run. It's the settings that will be in place from Visual Studio if I select the OdeToFood option instead of the IISExpress option. In both cases, we can see the ASPNETCORE\_ENVIRONMENT variable is Development, and here we can also see the applicationUrl being set to exactly those ports that we saw, port 5001 for TLS, port 5000 for an unencrypted connection. So those are the settings that .NET is using as it's starting up to launch my process, and in the process itself, since mean is my entry point, the first thing we're going to do is use an ASP.NET Core API to create a WebHostBuilder. Specifically, we're going to create a DefaultBuilder, so a builder is something that knows how to create a web host, and behind the scenes, what the builder is going to do, the default builder, is set up some of the default services like the Configuration service. It's also going to put some default logging services in place like a service that will send log messages to the console, so I can see those messages displayed when I run this directly from the command line. We're also going to tell this default builder to configure this application using the following Startup class, so we're passing that as a generic type parameter, but this is the Startup class that we've been inside of a few times. And ASP.NETCore knows it's going to instantiate this class and invoke two methods, one method which is ConfigureServices, this gives us the ability to put our own custom services into ASP.NET Core, and then have those services injected into pages, and controllers, and anywhere we need them. So throughout this course, we've been adding things like our DBContext, as well as our RestaurantData service. And then the other method ASP.NET Core will invoke on the Startup class is the Configure method. The Configure method is going to establish what middleware will execute for each incoming HTTP message. We're going to talk about Configure in more detail in the next clip. For right now, let's just circle back to Program.cs, and I'll show you that once we create the default builder and tell the default builder to use this Startup class to configure itself, then we're going to build the builder, which gives us a web host instance. That web host is going to know how to listen for connections, and process, and receive HTTP messages. All we need to do is tell that web host to run. Once we tell the web host to run, everything else that happens in this application is going to be determined by the code that we have inside of Startup.cs.

Processing Summer Corn with the Allen Family

As I've mentioned before, the configure method of our startup class is where we install middleware. We install middleware generally by invoking extension methods on an object that implements the IApplicationBuilder interface. In this case, ASP.NET Core passes that object as a parameter, we'll name it app inside of this method, and so the method UseDeveloperExceptionPage installs the developer exception page middleware into this application's pipeline. The UseStaticFiles method installs the static files middleware into the processing pipeline for this application. But you might be wondering, what exactly is a middleware component? We've talked about them in a very abstract sense, but let me try to make things a little more concrete by telling you a story. When I was a child, every summer, my parents would buy fresh corn to freeze for the winter, and I don't mean a little bit of corn, this was a lot of corn, hundreds of ears, and preparing this corn to freeze required a lot of work. So as a family, we made an assembly line. I was the first person in the assembly line, my mom was second in the assembly line, and my father was the last person in the line. As the first person in the assembly line, I would take an ear of corn from a pile of hundreds of ears, and I would tear off the husk. If the corn inside look bad, I could throw away this ear of corn. otherwise, if the corn looked good, and most corn would look good, I would pass along the ear of corn to my mom. Mom's job was to use a fine brush to remove the rest of the silk from the outside of the corn and clean it up. When she was done with that, she would pass the ear of corn to my father. My father was last in line, and his job was to take all the corn that we'd been passing down, put it into batches, and place that corn into a boiling pot of water to blanch. Blanching the corn prepares the corn for a good freeze, and that process takes a few minutes, but once this was done, my father would take those hot ears of corn and place them into an ice bath to cool them off, and at that point, this process and this assembly line would start to reverse. My father would pass those blanched ears of corn back to my mother, and my mother, using a small paring knife, would cut the corn off of the underlying cob. Eventually, mom would pass the cut corn over to me, and I would take this corn and place it into plastic bags for freezing. So I was the beginning and the end of this processing pipeline for corn. Now what I just described is exactly what middleware will do, but instead of processing ears of corn, middleware in ASP.NET Core will process HTTP messages. So imagine a configure method like the configure method that we were just looking at, but instead of code, like app.useStaticFiles, imagine app.useLogger, and then app.useAuthorizer, and then app.useRouter. So we've installed three pieces of middleware into a processing pipeline, just like my family, there were three of us in a corn processing pipeline. Now the first ear of corn arrives, which is really an HTTP message, and this HTTP message wants to make an HTTP POST request to the URL/corn. The first piece of middleware in this pipeline, the logger, might just want to inspect the request, and record some diagnostic information, and if the request looks good, the logger will allow the request to move along to the next piece of middleware, which is the authorizer. The authorizer might look at the incoming request, and make sure there is a cookie or an access token in the request. If the token doesn't exist, the authorizer can halt processing, just like I used to throw away a bad ear of corn, and the authorizer might do this by throwing an exception, or perhaps return a status code back to the caller that indicates this request is unauthorized, but if the request is good, the authorizer can pass the request along to the final component in the pipeline, which is a router. Now the router might take this request and try to find something inside the application that can process the request. So if this is a POST/corn, the router might go looking for a razor page in the pages folder named corn, or the router might go looking for an MVC controller that can handle this request. If so, the router is going to pass this request along, and eventually some razor page or some controller is going to produce a response. That response might be JSON data, or it might be HTML, or XML, or it might be an error page, but whatever the result, here's something that is important to understand about middleware. The middleware pipeline is bidirectional. Requests, flow in, and responses flow out, just like in the family food processing pipeline, corn would flow in one end, be processed, and flow back out the other end. So each piece of middleware in this pipeline has the ability to inspect the incoming request, and it's also going to know when the outgoing response is occurring. So, for example, the router might pass this response back to the authorizer, but the authorizer might not care about the response, all it cares about is authorizing an incoming request. The logger, on the other hand, when the authorizer allows the response to flow back to the logger, since the logger knows exactly when this request started, and now it knows when the response is going out, perhaps the logger just records how long it took to process this response. So this is what we are defining inside of the configure method of the startup class, we're defining exactly what components need to exist in this processing pipeline. and if we went to serve static files, then the static files component has to exist somewhere in this pipeline to be able to look at an incoming request and look at the URL, and see if that URL is going to match something that exists in a folder on the file system, and if we want to route messages to controllers and razor pages, we'll need a piece of middleware that knows how to route these requests. So middleware is extremely important in ASP.NET Core, and middleware will completely define exactly how your application will behave once it's up and running. Let's go back to the configure method and look at this in a little more detail.

Exploring the Application Middleware

Let's look at the middleware that is installed in the current application. Inside of the Configure method, if we're in a development environment, the first piece of middleware that is installed in the pipeline is a piece of middleware known as the Developer Exception Page Middleware. Anytime you admit add middleware to the front of the pipeline, you're doing that for one of two reasons. Either you want the middleware at the front of the pipeline so it's the first to receive an incoming request, or you want this middleware at the beginning of the pipeline so it's the last piece of middleware that will execute once the response is flowing out of the server and back from the middleware. Remember the middleware pipeline, just like the core processing pipeline, is a bidirectional pipeline. Requests come in. Responses go out. The Developer Exception Page Middleware is first because it really cares about what's going to happen in the outgoing response. It doesn't care anything about an incoming request. All the Developer Exception Page wants to do is call to the next piece of middleware, whatever that may be, allow the requests to flow through, and then to see if any other piece of middleware later in the pipeline throws an exception. If so, this exception page middleware wants to handle that exception and display a page that will help developers to debug this problem. So we only install the Developer Exception Page when we're in development, because the Developer Exception Page Middleware will catch an unhandled exception and render HTML about that exception, so things like a stack trace and in what file and on what line of code did this exception occur? That's too much information to use in production, so if there is an exception when we're running in production or when we're not in development, we're going to use a different piece of middleware, the Exception Handler Middleware, which is also listening for an exception to come out of the rest of the middleware. But now, instead of showing detailed information about the exception, UseExceptionHandler allows you to render some other page or view in your application that can show the user a friendlier error message and an error message that won't include sensitive information like file names and line numbers. Another piece of middleware that we install and only in production because of caching issues is the HSTS Middleware. HSTS Middleware essentially instructs the browser to only access this information over a secure connection. If for some reason you don't want to use a secure HTTPS connection, perhaps because you have a proxy sitting in front that takes care of the encryption for you, then you can remove the HSTS Middleware, as well as the HTTPS Redirection Middleware. This is going to send an HTTP redirect instruction to any browser that tries to access the application using plain HTTP. Then we have the Static Files Middleware and the Node Modules Middleware. We already know that Static Files attempts to serve a request by responding with a file that's in the wwwroot folder, and the Node Modules Middleware was a piece of middleware that we installed to serve static files from the node\_modules folder. Almost all middleware is going to take some options of one kind or another if you want to customize the middleware. For example, UseStaticFiles will take an object that is of type StaticFileOptions. So if we just take a quick look at StaticFileOptions, some of the parameter's behavior that I can set will be to specify things like the default content type, and I can even specify a file provider. Notice this is of type IFileProvider, so we're passing in an abstraction. And we can implement that interface to look for static files anywhere. And if we wanted to look in the database for a particular file or go to some storage in the cloud like Azure Blob Storage, we can specify a file provider to have static files work in whatever manner we need. The next piece of middleware here would be to UseCookiePolicy. This is a piece of middleware that's going to help track if the user has consented to the use of cookies. So you might have noticed the first time you ran the OdeToFood application, there's a message that appears that you can customize to specify your cookie policy. Once the user accepts your cookie policy, there's going to be a cookie set for this application, the AspNet.Consent cookie that will let you know, if the user ever returns here, that the user has already consented. So this works with some other services inside of ASP.NET Core that you can check with to make sure the user has consented to the use of cookies and tracking. And finally, the last piece of middleware is UseMvc Middleware. This is like the router middleware that we talked about earlier. This is the piece of middleware where, if nothing else, has served a file or done any useful work and started to return, UseMvc Middleware is going to look at the incoming request and then try to route that request to render a Razor page or to invoke a controller that will render a Razor View. So remember, each piece of middleware in this pipeline has the ability to stop processing and not call into the next piece of middleware in the pipeline. For example, if UseStaticFiles sees there's a request for /image.jpeg, and that request matches a file, image.jeg in the wwwroot folder, then the Static Files Middleware can just render that image and return that image back to the client, and we never have to call into the next piece of middleware in the pipeline. So in that case, we would never get to the MVC middleware, which would try to route the image.jpeg request to a controller or a Razor Page. And that's why the order of this middleware is important. Likewise, you could also have a piece of middleware here, let's say, that is checking for some sort of cookie or authorization token, and if it doesn't find that token, throws an exception. In that case, you can be sure there's no other middleware later in the pipeline that will execute. Now some of the other pieces of middleware that you might use and commonly see when you're programming with ASP.NET Core include app.UseAuthentication. So this piece of middleware is going to attempt to establish the identity of a user. And so you typically see this piece of middleware later in the pipeline so that anything that comes afterwards has a chance to know who the user is. Anything that comes before app.UseAuthentication might not be able to establish the identity of a user. There's also app.UseSignalR, so if you want to use SignalR to do real‑time WebSocket communication, there's a piece of middleware here to handle those types of connections and requests. And there's also, for someone who's using a SPA framework, which is a piece of middleware that essentially allows you to say, if no one else has responded and done anything in this middleware pipeline, then let me use this piece of middleware and a little bit of configuration to tell you what to do. This is typically where you want to render your index.html page, so the primary opening page that is the single page in your single‑page application. And you can see just in the IntelliSense, there's various other pieces of middleware that you might come across, so UseCors, which allows you to specify cross‑origin resource sharing headers to the browser. And some of the things I'm talking about now are obviously covered in other Pluralsight courses, so using SignalR and building SPAs with the ASP.NET Core. What I want to do next is just give you a little demonstration of what it might look like to write your own piece of middleware.

Building Custom Middleware

To make the concept of middleware really concrete, I want to show you how you could implement a simple middleware component. Now typically, if you're going to implement a real‑world middleware component, you'll write a full C# class definition. I'm not going to go into that level of detail. You can find other courses on Pluralsight about ASP.NET Core middleware that will go into that level of detail. I want to show you the simplest possible approach. I will warn you, you do have to be somewhat comfortable with delegate types in .NET and lambda expressions. Both of those topics are covered in my C# courses. But let's take a look at implementing a simple piece of middleware that will just say hello world in a response. One way to write a simple piece of middleware is to invoke a method on IApplicationBuilder named Use. You can see in the IntelliSense that Use takes a function of RequestDelegate and RequestDelegate. That means I pass to Use a method that will take a RequestDelegate as a parameter and return a RequestDelegate. Now I can implement this all with a single lambda expression. I'm going to pass along a named method that might make this a little bit easier to understand, and the name of my new method will be SayHelloMiddleware. This is a method I can now create in Visual Studio just by placing my cursor on that name, pressing Ctrl+., and saying that I want to generate a method Startup.SayHelloMiddleware. And if I scroll down now, what we should see is a method that takes a RequestDelegate as a parameter and returns a RequestDelegate. This brings up the question, what is a RequestDelegate? Let's place the cursor here on the symbol RequestDelegate. I'm going to press F12 to go to the definition. And we'll see that a RequestDelegate is just a method, any method that takes an HttpContext as a parameter and returns a Task. And this is really the core of what middleware is all about. A RequestDelegate is a method that is going to handle an HTTP request. Everything about that HTTP request is represented by this HttpContext object that is passed as a parameter into the RequestDelegate. From this context, my middleware can inspect the request, it can stream results into the response, it can set the status code and the content type of the response, and it can do this all asynchronously because we do return a Task. So back here inside of SayHelloMiddleware, let's work on returning a RequestDelegate. And what I'm going to do is now just flip over to using lambda expressions and say that we're going to return a lambda that represents a method that's a RequestDelegate. So a RequestDelegate takes an HttpContext object is a parameter. Let's name that parameter ctx for context. And so this is the bit of code that would be invoked for every request that is arriving at this application. And with that HttpContext, again, I can now inspect the request. So if I wanted to look for, let's say, a specific HTTP header or an access token, or a cookie, I can do that, and I can also access a response object. Let's say that we just want to write hello world into the response every time this method is invoked. What I could do is use WriteAsync and say that I want to say Hello, World! And we will do that every time through this method. This is an async method. I do want to await that method, which means I need to make my delegate an async delegate. And now let's just see how the application's going to behave. So notice I'm installing SayHelloMiddleware here in the middle of the processing pipeline. So SayHelloMiddleware happens before Static Files and before Node Modules and before MVC, so this bit of code will be invoked on every request that arrives at our application. If I now come over to a browser that is running the application, and let's do a refresh on the home page, what I see is Hello World! And now it doesn't matter what URL I request from this application, the only thing my application can do is say Hello, World! That's going to happen even if I request a nonsense URL like /foo. If instead I were to move my middleware component down here after MVC, then the MVC routing would be active before I have my SayHelloMiddleware. And that means I could still serve Static Files, I could still serve files from Node Modules, and I could even have Razor Pages and API controllers work. We will only reach the SayHelloMiddleware if nothing else above it in this HTTP processing pipeline was activated and produced a response. I can kind of simulate that. Let's move the SayHelloMiddleware back to the middle of the pipeline. But I want to change it around so that the SayHelloMiddleware only responds with Hello, World! if I go to the URL /hello. The way I could do that is to use an if statement. So I could say if the incoming request as a path that starts with the following segment /hello, only then do I want to write into the response the text, Hello, World! Otherwise, I want to allow other processing to happen. So if we're not at /hello, I want that request to travel to the next piece of middleware and potentially reach Static Files or Use MVC and have the rest of the application working. Well, that's where this parameter to SayHelloMiddleware becomes important. Remember, every piece of middleware is really a RequestDelegate. So the method that we have returned here is a RequestDelegate and this parameter, also of type RequestDelegate, represents the next piece of middleware in the request processing pipeline. So if I have determined that my middleware wants to handle this request, I can simply write into the response and do nothing else. Essentially, once this piece of middleware exits, we will start flowing back to the beginning of the pipeline again, just like we did in that corn processing example that I talked about earlier. So if the request path does not start with /hello, I want to allow the next piece of middleware to handle this request. So what I can do is invoke this next parameter, which requires an HttpContext. I'll just pass along the context that is available to me in this lambda expression. And since this is asynchronous, I can also await the next piece of middleware. Once I await the next piece of middleware, I've sent that message along to the rest of the processing pipeline to figure out. And after I've awaited next, I could inspect things about the response. For example, I could look at the ctx.Response.Stat to see if we've produced a successful response or not somewhere else in the pipeline. But this is really all I want my middleware to do. I want it to respond with Hello, World! if we are at /hello; otherwise, I want to allow other middleware to do the processing. And now I can have this middleware installed anywhere in the pipeline and still be able to reach my Razor Pages and API controllers because control will pass through this middleware to the next piece of middleware if we're not at the proper URL. So coming back to the browser, let's request /foo, and what I should see is a 404 response. So now I had a request that went all the way through the entire middleware pipeline, and it never found a component that was able to satisfy this request. There were no controllers, there were no pages, there were no static files. But if I go to, let's say, the home page, so just the root URL, that works, I can now go to /Restaurants/List and see my list of restaurants. And I should be able to go to /hello and hit my hello world middleware. So that's a quick concrete example of how middleware works in ASP.NET Core. Again, it does require you to be comfortable with delegates and lambda expressions. But if you ever want to implement something that is a cross‑cutting concern in your application, say something that's always going to check for an HTTP header or potentially set a response header for every response, those are the types of behaviors that you want to implement using middleware.

Logging Application Messages

If you run this ASP.NET Core application from the command line, every request that you send to the application will produce some logging output here in the console window. You can also see this output inside of Visual Studio when you run from Visual Studio if you open up the proper window. So inside of Visual Studio when the application is running, I can go to debug and say that I want to view the output window, and once I have the output window open, I can say that I went to see output from the ASP.NET Core web server. This will show me the same information, and this logging information can be quite helpful when you're trying to debug an application, particularly if you're trying to figure out why some piece of middleware might be executing or not executing, why isn't this URL reaching an API controller or razor page, and what kind of commands is the entity framework sending to SQL Server? All this information can appear in your output log if you configure the proper settings. So some of these settings are in appsettings.json. You'll notice there is a logging configuration section inside of here. By default, we're going to execute with a log level of warning. That means if there's any warnings or errors, I'm going to see information in my output log, but there's also other log levels that you can use. I'm going to leave it to the documentation to describe what those settings are and how you can change them. I'll leave a link in the README for my GitHub repository associated with this course. What I want to focus on is what I think is a more important question, which is how can I produce my own custom log messages? Well on ASP.NET Core, logging, like many other features, can be implemented as a service that is injected into various components elsewhere in the application. So there is a service, a logging service, that I can inject into API controllers, or razor pages, or view components, or even middleware components, that allows me to write into whatever logging syncs are available. Let me demonstrate by going into the restaurant's list page, what I want to do is come into the page model, and let's just write a message when there is a get request to the page model. So the first thing I'm going to do in the constructor is inject another service. I want to inject an ILogger of ListModel, so the logging service can take a generic type parameter that represents the class that it's going to be associated with, and this can sometimes help you differentiate log output. If you're trying to find the log output from a specific razor page, for example, the list model versus the edit model. What I need to do is bring in the name space Microsoft.Extensions.Logging, I'm going to do the trick where I just use Ctrl+. here to say, create and initialize a field logger for me, and at this point, I now have a logger field available to me throughout the razor page. So inside of OnGet, I could say logger, and then I want to log a warning, I want to log an error, I want to log something informational. Of course, if this is actually output or not will be influenced by the log level setting that you have inside of app settings. So to make sure this appears, let's do logger and log an error that says Executing ListModel, and of course, I could put additional information in here, but let's just see if we can get this basic message to appear. I'm going to save that file. Let's open up the output window again and look at ASP.NET Web Server. The way to think of this is to think of ILogger being customized for this list model class, and later I'll show you how this allows me to find output from this ListModel a little bit easier in the logging output; and what I should see in the output window is that I am executing ListModel, and I can see this came under the fail category because we logged this as an error, and if I ever want to see all of the output that came through ILogger of ListModel, what I can do is search for the type, OdeToFood.Pages.Restaurants.ListModel. What looks like the array indexing subzero here is what happens when you're logging against a parameter with the generic type like ILogger of ListModel, but it would be pretty easy to search my logging output for Restaurants.ListModel, and see all of the logging statements that were output by this razor page against that ILogger. Now something else that I want to point out that we've touched on briefly before is that sometimes my logging, I want it to be a little more detailed, so I might want more detailed logging when I'm in development mode, but when I'm in production, I might not want informational logging, I might want to cut this back to just warnings and errors, so as not to overflow log files, and there's an easy way to accomplish this with the configuration system of ASP.NET Core. Let's take a closer look.

Configuring the App Using the Default Web Host Builder

One of the neat things about ASP.NET Core being open source is that if I want to dig into the internals of something like the CreateDefaultBuilder method, I can go to GitHub and find the source code for that specific method. So let's say I want to figure out what is CreateDefaultBuilder doing. What sort of logging is it configuring by default? Because obviously there's some logging output that is happening. What are the configuration sources that it configures by default? Well, if I come into the aspnet/AspNetCore repository, it's not always this easy, but let's search for CreateDefaultBuilder and see if we can find a method named CreateDefaultBuilder. Well, here is one inside of WebHost.cs, which sounds like a pretty promising candidate, and if I scroll down, indeed, I will find the implementation of CreateDefaultBuilder that I'm looking for. It's this one down here that is currently starting around line 148. First of all, I can see it setting up a number of different configuration sources, so one configuration source is the command line, another configuration source is appsettings.json, and yet another configuration source is appsettings.EnvironmentName.json. So one thing that I need to point out is that the configuration sources in the ASP.NET Core are hierarchical. So if my configuration sources are the command line first, and then appsettings.json, and then appsettings.SomeEnvironmentName.json. So this is literally the environment name that I'm operating in. So this would be appsettings.Production.json or appsettings.Development.json. Then when I go to the configuration service in ASP.NET Core and say, please tell me the configuration value for the setting named message. If that message exists from the command line, ASP.NET Core will use that value, but if that setting also exists in appsettings.json, then the value from appsettings.json will win, and that's the value we'll use. But I can also override that particular setting by adding a setting into appsettings.Development.json, and then I also want to point out that another configuration source is environment variables, and that can be extremely useful. If I have a database connection string that has a password inside, I don't necessarily want to put that in appsettings.json and check it into source control, I want to keep that setting outside of source control, that's why ASP.NET Core has user secrets that are available for development, and in production, we would typically place that connection string into an environment variable. So I just want to show you how these different configuration sources can work. Currently, inside of appsettings.json, we have our logging levels, and we also have this message that says, "Hello from appsettings!". You might remember that we retrieve this message and we display this message, when we show a list of all our restaurants, we show this down here at the bottom of the page, "Hello from appsettings!". We do that because inside of the ListModel, where we just add as a logging, we asked the IConfiguration service to give us a configuration value for the key named message, and because this CreateDefaultBuilder by default, added appsettings.json to the list of configuration sources, we were able to retrieve this message, "Hello from appettings!", but I just want to point out if I take that value, and open up a file that is hidden here behind appsettings.json, which is appsettings.Development.json, and as we just saw in the CreateDefaultBuilder source code, this file, any settings that I have in here will override the settings that are in appsettings.json, but only when I'm in the development environment. So inside of here, in development, I can see that the logging level by default is tuned so that I should see more information. So the default level here is debug, and anything coming from the system or Microsoft logging sources will be at the informational level. So again, more details on those different levels in the documentation that I'll link to, but let me also paste in here that I want to change the message from "Hello from appsettings!" to "Hello from Development!!". I'm going to save this JSON file. Let's come back to the browser and let me issue a refresh, and down here I can now see the message is Hello from Development!! Essentially, the settings in appsettings.Development.json override the settings that are in appsettings.json, but only when I'm in development. Otherwise, appsettings.Development.json won't exist as a configuration source because CreateDefaultBuilder is going to add appsettings.Production.json if I don't have a named operating environment. Notice, also, there is some logging configured in here by default, and of course, all these settings in the DefaultBuilder are settings that you can override and change, The default logging will be configured using this logging configuration section that will be pulled from one of the JSON files or some configuration source that is already configured, and then we will add the console as a destination for our logging statements. That's how I can see output when I do a .NET run. We'll also add the debugger as a destination for our logging statements so I can see things inside of Visual Studio. And so now we know a little bit more about how CreateDefaultBuilder works, how it configures the configuration system itself and how it configures logging, and we also see a very important characteristic of ASP.NET Core that is taken advantage of in many different scenarios, which is the ability to override different configuration settings by placing settings into different files or different configuration sources. Just for fun, see if you can set Message as an environment variable from the command line and then use .NET run on the application to display that custom message from the environment variable. Remember, launchSettings.json can influence what environment you're operating in. When you run using .NET run, you'll be using the settings that are specified in Ode to Food, and you'll be using this ASP.NET Core environment. I'm also going to go back and set the environment back to development for running under IIS Express.

Summary

In this module, we took a look at some additional details about the internals of ASP.NET Core. We saw more features of the configuration system, and we saw how we can flexibly switch between development and production and have different settings in place for those two different environments. We also added some logging to the application, and we wrote our own custom simple piece of middleware and came to understand a little bit more about how middleware works and why the ordering of middleware is important. In the next module, we're going to turn our attention to the last topic for ASP.NET Core, which is how to publish and deploy an ASP.NET Core application.

Deploying ASP.NET Core Web Applications

Introduction

Hi, this is Scott, and in this module, we are going to look at deploying an ASP.NET Core application. I want to show you how to publish an application into a folder that will contain all the files that you need to run the application in any environment where .NET Core is installed, but we will also publish a self‑contained application that has no dependencies on framework and runtime installs. I'm also going to show you how to get your ASP.NET Core application up and running using Windows Web Server IIS, and if you want to see how to run ASP.NET Core applications inside of Azure, I'll refer you to my Getting Started with Azure for .NET Developers course, I cover that scenario in detail.

Publishing Apps vs. Deploying Apps

When we talk about deploying an ASP.NET Core application, we're really talking about putting this ASP.NET Core application somewhere in a server‑hosting environment. And since ASP.NET Core works across different platforms, the hosting environments these days can vary widely. So you might want to deploy to a Windows server running IIS, or you might want to deploy to a Linux server running Nginx or Apache. I want to run through a couple different scenarios with you. But before we do that, I want to make sure you understand the distinction between deployment and publishing. Publishing a .NET Core application is something you have to do before you can deploy that .NET application The publish step makes sure that all the files and everything you need is put together into a specific location so you can take all the files that you need and then deploy them. I also want to point out in this module, we're mostly going to be working with the command line tools to do the publish operations. The command line tools will work anywhere, on Windows, on Linux, on macOS. If you're using Visual Studio, you also have the ability to do a publish operation by right‑clicking on the project and selecting Publish. Truth be told, most of the options over here for a publish target really require two steps. First, there's a publish operation in Visual Studio, and then there is a deployment, for example publishing and deploying to IIS or to an Azure virtual machine or to an Azure app service. The closest operation to what we'll be doing with the command line would be to publish to a folder. It is this selection that allows you to go through the UI here and say I want to publish this application into a specific folder. And what we'll see when we go to the command line tools is there's a few different options that we can pass along. So these publish options, these advanced settings, they're available in Visual Studio UI, as well as from the command line. One of the options is which configuration to publish. Do I want to do essentially a release publish or a debug publish? Then there's also the target framework. This specifies what framework has to be in place for this application to run. By default, when we set up this project using a new template, the project was set up for us to run on what we call netcoreapp2.1, that framework, so .NET Core 2.1. And our deployment mode, again, something we can select from the command line tools also, by default is framework‑dependent. That means the destination machine, the server, wherever we're going to copy the application files to, it will need to have a framework preinstalled on it, in this case .NET Core 2.1. But we could also build and publish what's known as a self‑contained application. A self‑contained application doesn't depend on a framework being installed on a server. All the files that are needed to run the application are in one place in one folder, and all we need to do is copy them to the destination system, and the application will run. When we have a self‑contained deployment, we will also be forced to select a target runtime because if this publish operation is going to put together all the files that we need on the server for the self‑contained deployment, then we need all the correct files that are specific to a given environment for .NET Core, for example all of the files for a 32‑bit Windows environment or a 64‑bit Windows environment or on OS X or on a 64‑bit Linux environment. So we're not going to be using Visual Studio in this module. We're going to use the command line. Visual Studio might be a little bit easier than the command line the first time, but the command line is going to give you that low‑level control that you can take advantage of, particularly if you want to automate this process.

Using dotnet publish

Here we are at the command line, and the command that we want to explore is the .NET publish command. I'm going to pass in the help flag so we can see what options are available here. And you'll notice that many of the options for the command line map to options that we just saw in the Visual Studio UI, so the ability to specify an output directory, and a target framework, and a runtime identifier, and a configuration. And that's because behind the scenes what Visual Studio is doing is really using .NET publish in the background. So let's try to publish our application. I am inside of the same folder as my OdeToFood.cs project file, so the main project that is the web application, and what I want to do is a .NET publish operation, and I'm only going to pass along one option to start with. I'm just going to specify the output directory, so ‑o. Let's place the publish output into a folder in my temp folder, and let's place it into c:\temp\odetofood. When I press Enter, .NET publish will make sure that all the NuGet packages are restored. It will make sure that everything is built to the latest version with the source code that is on the disk here, and then it's going to take everything that I need and publish it into the c:\temp\odetofood folder. Let's take a look at that folder and see what is inside. So scrolling up here, the first thing you'll notice is that there's a number of assembly files in here, so .dll files. These assemblies represent dependencies that .NET publish found for my application, dependencies that are not part of the target framework. So we do not need the DLL files that represent .NET Core itself, but we do need the DLL files that we might have referenced through NuGet packages, for example, OdeToCode.UseNodeModules, because those DLLs might not be on the target system when we deploy. You'll also notice the assemblies are here from the other projects that the web application references, so OdeToFood.Core and OdeToFood.Data. I also want to point out that our appsetting files have been published into this folder, both the development and the production file. I also want to point out that OdeToFood.Views.dll exists here. So any Razor View that is in the project or any Razor Page that is in the project is compiled and packaged up into a DLL, much like our regular C# code is packaged up into OdeToFood.dll. So you will not find a Pages folder here in the published output, instead, all of our views and pages are compiled into this binary. You will also find, as I scroll down a little bit, a web.config file, we're going to come back and talk about that one later, and a wwwroot folder. So all of our static content that existed inside of wwwroot in the project has been copied over to this folder, and that means any images that we need are here in the published folder, so when we deploy, those images will be available on the hosting server. Also, any JavaScript files and CSS files, the entire wwwroot folder from the project was copied over. Now you might remember there's another folder that contains static assets like JavaScript files that we need, at least if we're running in a development environment, and that is the node\_modules folder. That folder is currently not here inside of this published output folder. I'm going to show you how to fix that. Of course, we typically only use node\_modules when we're in development, not in production, but it would be a good idea to have node\_modules available here in this published output, and I will show you how to do that before we leave the module. But right now, what I want to show you is if I push the current directory onto the stack and switch over into the OdeToFood directory, what I want to show you is that we used to run a project in this course by saying dotnet run, but dotnet run assumes that you're in a folder with a cs project file. Something you'll notice that is missing from this published output is any of the source code files, so there is no Program.cs or Startup.cs or csproj file, all of the C# code has been compiled into a DLL. So then the question is, if we cannot use dotnet run, how do we launch the application from the command line like we used to? Well, I still use the .NET CLI, but now I can specify the name of the assembly that I went to execute. And what .NET will do is load up that assembly, look for the entry point, which we know is the static main method, and then start executing the code inside. So if I try this here, we're actually going to end up with an error, and if I scroll up, I will show you that the error actually is because the node\_modules folder does not exist. We load up that use node\_modules middleware regardless of what environment that we're in, development or production. So that middleware is always going to be looking to use the node\_modules folder, and if it doesn't exist, that middleware is going to throw an exception and our application fails to launch. Now it would be possible from this folder just to run an npm install because the npm package.json file was included in the published output. So if I ran an npm install here, npm would look inside a package.json, see that I have a dependency on jQuery and Bootstrap, and jQuery validation would create the node\_modules folder and install all those packages. Then I could run the application, and it would be happy. But running npm install by hand from the command line, typically not something that you want to do if you're going to automate your build, so let me just show you a few quick steps that you can take to make sure that the npm install always happens so that the node\_modules content is published into the output folder.

Using MSBuld to Execute npm install

To make sure that we have the node\_modules folder available in the published output for OdeToFood, I'm going to make some changes in the cs project file for OdeToFood. In Visual Studio, I can right‑click a project and say that I want to edit the csproj file directly. If you're using Visual Studio Code or some other editor, you can just open the project file in the editor. And there's just two changes that I want to make inside of this project file, and there's two pieces of XML that I want to add to this project file and two pieces of XML that I want to remove. And just to give you some background, the cs project file in .NET is what we call the MSBuild file for a project. That's because there's a tool for .NET named MSBuild that has been around since the very beginning of .NET that understands how to process a csproj file and execute the instructions inside to build a project, and publish a project, and do many other things. Since MSBuild has been around since the beginning of .NET, over the years it's collected many features, so it's very extensible, it can do a lot of things, sometimes even MSBuild can be a bit overwhelming with the number of different approaches and features that you can use with it. But here's what I went to do, I want to automate the npm install for this project whenever there's a build taking place, and then I want to make sure that the publish operation knows to include the node\_modules folder that is created by an npm install. So first let's take care of the npm install step itself. What I'm going to do is paste a small bit of XML into the csproj file, it doesn't really matter where this exists, but what I'm adding to the project file is a new target for MSBuild. So what MSBuild is looking for when it opens up a cs project file will be different targets to execute. There's many targets included by this SDK that we don't see that are in effect for this project that we don't see in this file, but what we're doing is adding another target. We're telling MSBuild that the name of this target is PostBuild, and what we want you to do is execute the steps inside of this target after this other target that's defined for you. That other target is ComputeFilesToPublish. What we want to do after that other target completes is execute npm install. Now I could make this fancier, I could make sure this only happens in a certain way during debug or during a release configuration, I could also check to make sure that node and npm is installed before trying to execute this step, but we're just trying to do the simplest possible thing that can work for our scenario. So do an npm install, that will give me a node\_modules folder. Now, unfortunately, the publish operation by default is going to ignore a folder named node\_modules, so what I need to do is bring in another little snippet that says, and provides instructions to say, I have some content that I want to include in this project, and that content exists in node\_modules. Take that folder and everything inside of it, and when you go to publish, I want you to always publish that content and preserve the newest files. So that's going to make sure that when we do a dotnet publish, not only do we have an npm install that executes, but we also copy over the contents of node\_modules into the published output. Now because this Content group exists, I'm now going to delete some XML. You may or may not have to do this, but any time you're inside of the Visual Studio Solution Explorer and you right‑click folders to say, include this in the project or exclude this from the project, and what we did earlier in the course was right‑click node\_modules and say Include this in the project. Any time you perform that operation, sometimes there's some strange artifacts that are left behind in the cs project file telling Visual Studio include this and don't include that, and that's the problem that we're facing here with this ItemGroup. It's trying to include some individual files from the node\_modules directory. What I want to do is replace this content, all of the content items in this ItemGroup, with the content that I specified here to just say, take all of node\_modules. And for me, when I'm in Visual Studio, and for me whenever I perform these operations for whatever reason, it added two separate ItemGroups with node\_modules and individual files. I just want to delete all of those, remove those, and leave behind just this content that says take all of node\_modules. So now we have a step and a target that will do an npm install. We're telling the project system to include node\_modules in the published output. I am going to save this file and let's return to the command line. Now what I want to do is pop back into the folder where I was executing dotnet publish. Let me use the up arrow and try to find where I executed that command. Here it is, dotnet publish into c:\temp\odetofood. Let's execute this again. I should see some npm outputs scroll by, and I do. That's a good sign that we executed the npm install. And now let's look inside of the temp\odetofood folder and take a quick look to see if we have node\_modules, which we do, and that's good. I should now be able to do a dotnet on OdeToFood.dll and start this application running, but before I do that, let's talk about the difference between this dotnet publish operation, which depends on a framework being installed on the host system, and a self‑contained application.

Building Self-contained Applications

Now that we have node modules included in the published output, let's take a look at an additional option for dotnet publish, which is sometimes very useful, and that is the ‑‑self‑contained option. Without this option on publishing output files that depend on a target framework already being installed on the server or the hosting environment where I'm going to move this application to, but there are some situations where, depending on a pre‑installed framework, has some drawbacks. For one, you might want to take a dependency on a framework that is newer than what you have in production. We know sometimes production environments can move slowly. We also might be in a situation where there are multiple applications deployed into a production environment, and we want to make sure that if someone upgrades or patches a framework on that machine to fix something in application A, it doesn't impact our application, application B. We want everything that this application needs to be in the published output so we can copy everything that this application needs over to the other server. Of course, there are also some drawbacks to having a self‑contained deployment. If there is some sort of security patch that can be applied to a target framework in a hosting environment, if all the applications are depending on that target framework on a shared installation, I can upgrade that framework once and be sure that all the applications are using the latest bits. But if I have a self‑contained application somewhere on the system, I can have situations where that self‑contained application will be using different bits than the shared target framework install. Well, let's take a look at what this does. If I try to publish with ‑‑self‑contained here, I'm going to eventually run into an error, and that error is that I need to specify a runtime identifier, or turn off the self‑contained flag. And this makes sense. If I want to create a self‑contained application, now what dotnet publish has to do is include an entire runtime in the published output. And the question is, which runtime should dotnet publish include, because there are many different versions of the runtime. There's the version that runs on 32‑bit Windows and 64‑bit Windows and 64‑bit Linux. Which one of those is the runtime that I want? I need to know where I'm going to deploy this application and what the target system looks like. What is the architecture, 32‑bit or 64‑bit? What's the operating system? Once I have all that figured out, I can then select the runtime identifier that I need. And I can do that by executing this command again and passing a ‑r option and then a runtime identifier. So what is a runtime identifier? Well, if you go to your favorite search engine and search for .NET Core Runtime IDentifier, you should see a link to the .NET Core RID catalogs or Runtime IDentifier catalog. This will list the small mnemonics that we can use as runtime identifiers. So, for example, if I want to target 32‑bit Windows, very generic, I'm not being specific and saying it's going to be Windows 10. I'm just saying a generic 32‑bit Windows environment. Then I would use win‑x86, and of course, there's also the 64‑bit version, win‑x64. And looking through this list, you can find a generic Linux runtime identifier or what we call the portable identifier, and the same for macOS. It's going to be on here somewhere down this list. What I'm going to specify is a runtime identifier of win‑x64. So back at the command line, win‑x64 is going to be my target system, because I'm going to deploy this application on the system that I'm currently executing on, which is 64‑bit windows. And now it's going to take a little bit longer to do this publish operation. And what we're going to find in the output folder is quite a bit more output. There are many more dll files and assemblies inside of this folder now, because we have the entire framework in this folder. So not only do I have my dependencies like OdeToCode.UseNodeModules. I also have system.xml.dll that provides all the XML support for .NET Core. Yes, of course, I still have my wwwroot folder, but I also have files that will be specific to 64‑bit Windows like WindowsBase.dll. So I cannot take this application now and deploy it onto a Linux system. I would need to do a separate publish operation and produced the output for Linux x64, and I could take those files and put them on Linux. Also, now if I tried to do a dotnet on OdeToFood.dll, well, it might work, because I am on a system where the. NET CLI is installed. But because we want this application to be completely self‑contained, I might copy these files to a system that does not have the .NET CLI. It doesn't have a framework or an SDK installed, and because of that, in addition to finding OdeToFood.dll in this folder, and what I'm looking for actually is in the output folder, so let's look inside of OdeToFood, what I should see in this folder now is an OdeToFood.exe.. So now if I want to run my application, I can type Ode to Food and then OdeToFood.dll. So essentially OdeToFood.exe replaces the dotnet command in the. NET CLI. This application is now self‑contained. So now that we have a published, self‑contained application, the next step after publishing is to do a deployment. Let's try that next.

Deploying to a Web Server

One of the wonderful things about APS.NET Core is that when it comes time to deploy, you have many different options. If you do a search for ASP.NET Core Nginx, you should find a link to this page, which is how to host ASP.NET Core on Linux with Nginx. So if you're in a Linux environment, now that you know how to publish an application, you can take that published output, combine it with this documentation, and figure out how to get your application up and running with Nginx. You can also search for ASP.NET Core and Apache. Apache, another popular option on Linux. And this documentation will walk you through how to configure Apache to host your ASP.NET Core application. What I want to do in this course is show you how to host ASP.NET Core on Windows with IIS. It is, fortunately, a simple operation. And we're also going to get our application running with a SQL database that is not the same as the LocalDB that we're using for development. Now, one caveat about IIS deployments, even though we have created a self contained application that includes an entire runtime deployment so that we don't have to depend on any other bits at runtime, there is still a component that you need to install on a server or IIS to execute ASP.NET Core. The download for that component is known as the .NET Core hosting bundle. You can see there's a link right here at the top of this article for hosting ASP.NET Core on Windows with IIS, and what will happen when you follow the links is you will eventually download an MSI installer that will install something known as the ASP.NET Core module. Sometimes you'll see that referred to as just ANCM, ASP.NET Core Module. The ASP.NET Core module, as the name implies, is a module the plugs into the IIS processing pipeline and facilitates the hosting of an ASP.NET Core application with IIS. I also want to point out that I am using ASP.NET Core 2.1 and with ASP.NET Core 2.1, I do need the ASP.NET Core module installed, and this model is going to allow me to do what's known as out of process hosting. Coming up in future versions of ASP.NET, we'll be able to use this ASP.NET Core module and do in process hosting so that my application code is actually running inside of the worker process for IIS, and that can lead to some performance benefits. It's really a configuration change to move between out of process and in process. I'm going to show you the basic steps that you can use with IIS to get up and running with ASP.NET Core in general. And by the way, again, if you want to deploy to Azure, my Azure Getting Started for .NET Developers will show you exactly what to do for an ASP.NET Core project. But with that being said, I do want to make one small change to the project itself that's going to make this a little bit easier to deploy. I'm going to come into Startup.cs and temporarily what I want to do is remove the use of an actual SQL server database. We will add this functionality back in later, but just to get the site up and running, I want to comment out this line of code that says use the SQL RestaurantData service for anything that needs an IRestaurantData interface. And I'm going to replace that and say that we want to use InMemoryRestaurantData with any component that needs IRestaurantData. So you might remember at the beginning of the course, InMemoryRestaurantData did not access a database, it accessed a list that was held in memory. This means we're cutting out the database access temporarily, and we're just going to focus on getting the site up and running in IIS and then turn on the SQL database. So with that change in place, let me come back out to the command line. What I want to do is just one more time do a .NET publish with these latest bits, these latest changes, and while that is happening, I'm going to flip over to the IIS manager. Let me bring this window over here onto the screen where I am recording. And first of all, I'm going to click on the server and I'm going to go to the Module section. And by the way, this demo is going to move relatively fast. It will feel very fast if you haven't done a lot of work with IIS in the past, but I'm really trying to avoid talking about IIS in detail. There's lots of other Pluralsight courses that go into IIS for developers and IT admins. What I want to do is just focus on this deployment. But I will show you, if you come into the server and you go to the module section, I can open this up, and I just want to verify that I do have that ASP.NET Core hosting bundle installed. I can be sure that is installed if I have the ASP.NET Core Module Version 2 installed as a module here at the server level. That is in place, so we are good. Next, what I'm going to do is go to our list of sites. Currently we have a default website in place, but I want Ode to Food to be a new site that is a part of the server. So I'm going to add a website. Let's give the site the name of OdeToFood and the physical path will be the path to our published output. So I'm going to enter in c:\temp\odetofood. Of course, in production, you'll want to make sure that that folder has the proper ACLs, and it's a bit locked down. But again, all of that is covered in other courses. And also, I'm going to bind this to an HTTPS port, the default HTTPS port, which is port 443. And to access the site, we're just going to be using localhost. Because I want to use HTTPS, I must select an SSL certificate to use. Now, this will be a bit unorthodox and something you would never do for a production website, but as long as I'm just running locally, I can select the IIS Express Development Certificate that is already installed on this machine and that certificate is going to work well, because it's already trusted and set up well to work with localhost. You won't be able to access this website remotely and have a trust relationship with the server. But again, I don't want to go into the details of how to create and install SSL certificates. Those topics are already covered well in other courses here on Pluralsight. With these settings in place, I'm going to say, yes, go ahead and start the website immediately. We're going to press OK. And now let's come over into a browser, and what I want to do is go to HTTPS and then localhost without the port number, because it's going to be the default port number. And I have my application running. So a deployment to IIS is very simple. All I need to do is take my published output and point a website to that published output. Once I have the ASP.NET Core hosting bundle installed, everything's going to be relatively smooth. But before we declare this a complete success, we still need to get a database up and running. And before I get the database up and running, I also want to show you something that is happening behind the scenes with IIS and ASP.NET Core.

Exploring web.config and How IIS Hosting Works

One of the questions that you might ask after that last demo ism how does IIS even know what to do with an ASP.NET Core application? All I did was point the IIS physical path to my published output, and somehow IIS just magically knows how to run my application. How does that work? Well, let's actually again move over into the published output, the OdetoFood directory, and I want to again call your attention to this web.config file that lives in this folder. And if you've done any ASP.NET development in the past, you'll know that web.config used to be the configuration file for ASP.NET. In ASP.NET Core we do not use web.config anymore, at least not by default. You could always plug in a custom configuration provider that looks for a file named web.config. But web.config as a default configuration source no longer exists in the world of ASP.NET Core. However, web.config is a configuration source for IIS. So when we point IIS to this folder, one of the first things IIS will do is look inside of web.config and ask the question, how am I configured, what should I do? Let's open up this web.config file, and let me bring this over onto the screen, and I just want to show you, it's a very minimal configuration. But what this is telling the web server, which is IIS, is that I want to install a new handler, that is something that handles incoming HTTP requests. And what I want you to do is give this handler the name aspNetCore, and I want you to take every request that comes in for any path to this website, using any verb, GET, PUT, POST, DELETE, PATCH, HEAD, anything. What I want you to do is take all those requests and forward them to the aspNetCoremodule. So that is the aspNetCoremodule that is installed by the hosting bundle for .NET Core. The aspNetCoreModule then knows that it needs to take that request and somehow get it to the middleware that is inside of my application. And it's going to do that differently for out‑of‑process hosting versus in‑process hosting. But ultimately it is the aspNetCoreModule that receives every request coming in to this IIS server for this site and will make sure to deliver those requests to my application. So here's a bit of additional configuration for ASP.NET Core here in .NET Core 2.1. This is telling that module that the way to start my ASP.NET Core application is to run the OdeToFood.exe program and give it the argument OdeToFood.dll, which is my application. That's the assembly that contains my application logic. So we already know it's OdeToFood.exe that works with a self‑contained application. Otherwise, this could just be using the .NET CLI and say just execute .NET OdeToFood.dll, and that will give you the process that you need to use to process these requests. So this is some of the magic that is happening behind the scenes with IIS in combination with the ASP.NET Core, and it's how IIS knows what to do with my application. It's because of this little bit of configuration and the aspNetCoreModule. It also means that behind the scenes IIS is essentially just running OdeToFood.exe, or it's running .NET OdeToFood.dll. And that means if you have any problems trying to get this deployment to work inside of IIS, one of the easiest things to do when you are sitting on the server like I am right now is just try to run OdeToFood.exe from the command line. There's a good chance you'll see the same error that IIS is seeing that you're trying to debug. Perhaps it's a configuration error. Perhaps a configuration file is missing. Perhaps the node\_modules folder is missing like we saw earlier. That would've prevented this from working inside of IIS. But I can see here we are up and running. We're listening on a different port because we're using a different server now. But if the application works here running from the command line, there is a good chance that it will also work from IIS. And just to be clear, what we're doing here when we run OdToFood directly, we're using a web server that is built into ASP.NET Core known as Kestrel. Kestrel can sit here and listen for requests, just like IIS does. Kestrel by default's going to be listening on Port 5000 and 5001. Now you could use Kestrel and configure your firewall to allow remote requests to reach this particular server, but Microsoft still recommends that you do not use Kestrel in a production environment. They still want you to take your ASP.NET Core application and run it behind a production‑ready, secure, hardened web server like IIS or Apache or NGINX. So I'm going to shut this down, but that's just a quick troubleshooting tip. And now that we have the application up and running, let's take the next step, which is to get the application working with SQL server

Setting up Automatic Entity Framework Migrations

Now that the application is running in IIS, we want to hook up the database. So the first thing I'm going to do is come back into the OdeToFood project, comment out the line where we said use InMemoryRestaurantData, and re‑add the line of code that says yes, please use the real SqlRestaurantData service. So now, before our application can execute successfully, we're going to need a working database. To get a working database, I'm going to do this in two steps. The first step you might think of as how do I create this database? How did I get it started? Well you might remember from earlier in the course, we would use. NET migrations to create a database. That might be something that you don't do in production because you're just not going to be able to connect to a production database and run the dotnet command to execute migrations against that production database. So what you might do instead, as I flip over into the data directory for this application, is I might do a dotnet ef migrations script. Remember we talked about this command earlier in the course. I want to generate a script for all of the migrations that are there, and I want to specify a startup project for this of OdeToFood\OdeToFood.csproj. So that's the startup project, which is my web application. Doing this, dotnet ef will be able to generate a script for me that I can copy and place into a file and perhaps hand over to a DBA and say this is how you create the database. Then all I need is the connection information that I can plug into a configuration file or a configuration source somewhere that allows my application to access that database. But I'm going to take a slightly different approach. The approach I'm going to take is I want to add some code into the application that will automatically execute these migrations against whatever database it's configured to connect to. And if that database doesn't exist, the application will create the database itself. Of course, this will require the application to have access to the database in the dbcreator role, which again is not always something that happens in a production environment. You typically try to minimize the privileges available to an application. But since I cannot cover every possible scenario, I'm just trying to show you some of the common approaches to integrating with the database. And then we're going to carry through with this full demo by just enabling an automatic migration feature. The way to do this is to execute migrations when this program is starting up, and the ideal place to do that then would be inside of program.cs here at the entry point before I start running my web server. In other words, I want to separate the steps of building a web host and running the web host into two distinct steps, so I have a place in between where I can execute migration. So I'm going to say host = CreateWebHostBuilder and build that host and then, some time later, do a host.run to start the server process executing. In between, this is where I want to be able to execute migrations against that database. And fortunately, the DbContext in Entity Framework Core provides an API that allows me to execute migrations against a database. It's just going to require a little bit of code. But what I want to do is tuck this away into a separate method perhaps. So let's say that I want to invoke a method called MigrateDatabase, and I know I'm going to need this host object because the host object will give me access to some of the services that I need, like the DbContext. And now I can use Visual Studio and Ctrl+period on this symbol to generate this method for me. So let's now implement this method. In order to migrate a database, I need access to my DbContext object. So over in OdeToFood.Data, what I need is access to OdeToFoodDbContext. I need a new instance of this class to do the migration. But I cannot just use the new keyword and construct an instance of this class myself because I don't know exactly what DbContext options to pass along. All of that has been configured inside of the ConfigureServices method of this application, specifically what type of database to use. Is it SQL Server or some other database? What is the connection string to use? I want to rely on all that information that is already configured into my application. In other words, I want to effectively resolve a dependency from the service provider that is in ASP.NET Core that is typically used to inject dependencies into things like Razor Pages and controllers that we've seen all throughout this course. And that's not hard to do. The first step is I'm going to add a using statement and grab ahold of something known as a scope. And I can get a scope by going to host.Services. So I'm asking my web host about its collection of services, and I want it to create a scope for me. I'll explain what a scope is here in just a moment. But this CreateScope method, it's an extension method in a different namespace, which is Microsoft.Extensions.DependencyInjection. So I press Ctrl+period inside of Visual Studio, and let's bring in a using statement for that namespace. And now I have a scope. A scope ensures that I can grab a service from the service provider and then clean that service up in the appropriate fashion afterwards. Remember, services are registered with a lifetime. So we have singleton services and transient services and HTTP request scoped services. When I get a service, when I get my DbContext out of the service provider, I want to make sure it maintains the proper scope. And I can have this using statement and the scope effectively clean up the services that I'm using when I'm done if they do indeed need some cleanup. So from this scope object, I have an API that will allow me to get to an OdeToFoodDbContext. And the way I do that is I'm going to store that in a variable called db, but I go to the scope and I go to the service provider and I say please get me the required service, so the service that I absolutely need, and that is the OdeToFoodDbContext. I'm going to need to bring in a namespace for this. It's over in the OdeToFood.Data namespace. So I'll bring that in with Ctrl+period, and I now have a fully initialized DbContext. I just noticed that I'm missing a semicolon up here. Let me add that so everything compiles correctly. And now that I have a fully initialized DbContext that can talk to the database, let's use the API to execute the migration, which is I go to the Database property that represents the database. You can see I can do different things here like begin a transaction. I can delete the database. But what I want to do is execute a method, again an extension method, named Migrate. So I need to use Ctrl+period here and bring in a namespace, Microsoft.EntityFrameworkCore. What this method will do, as the IntelliSense documents, is that it will apply any pending migration for the context to the database, and it will create the database if it does not already exist. So now every time my applications start, if there have been any migrations that have been added to the application, specifically to the data project, which we've referenced, those migrations will execute as soon as the application starts up. Of course, in a real production environment, you might want to take some additional steps to make sure that your database is backed up before you apply migrations just in case something goes wrong. But I'm going to leave some of these important details as exercises for the viewer. But now that I have the ability for my application to create the database that I need, all I need to do is allow this application to connect to the proper database.

Connecting to a SQL Server Database

What I want to do is allow my application to work with a real Microsoft SQL Server Server instance that is on this machine, and by real instance, I mean, it's not a LocalDB. It is an instance of Microsoft SQL Server Server that has been installed. I have the instance that I want open in Microsoft SQL Server Management Studio, and the first thing I want to do is create a login that my application can use. So I don't want to create the database. The application is going to be able to do that, but I do need to create in login for my application and give that login the ability to create a database. So the first question is, what sort of login do I want to create? Well, I can certainly create a login for my application that has a username and a password. Then all I need to do is create a connection string that uses that username and password to connect to this database, but I want to do something a little bit different, something that could possibly be a little bit safer, and that is to use Windows Integrated Security to allow my application to connect to this database. So it turns out, again, this isn't of course an IIS, but every website that runs an IIS runs in something known as an application pool. And an application pool, every application pool has an identity, so a real windows identity. So one of the application pools that I have here is OdeToFood, and since this was created to use a default identity, which is the application pool id, that means if I can just use this application pool id and add it as a login to my SQL Server instance, my application running in IIS should be able to connect to the SQL Server database. So over in SQL Server Management Studio, the way to do this is to say that I want to create a new login, and the login is going to be IIS AppPool\, and then the name of the application pool, which is OdeToFood. At this point, I don't want to click Search. It's actually been one of my pet peeves over the years that you cannot find this login name if you do a search. I just want to type in IIS AppPool\OdeToFood, and say yes, this is the login I want to use with Windows authentication. Click OK, and this login should appear down here. Now I want to right‑click that login, and of course, all of this can be done through Transact‑SQL scripts also, but again, this isn't a module on IIS or SQL Server. We're moving really quickly through a lot of different topics here, but hopefully it will give you some insight into what you can do in different environments. Right‑click on this login, and what I want to do is go into the properties, because now I want to set a server role. I want to make sure that this login is in the role of dbcreator. This will allow my application to issue, create database statements, and once my application issues create database, it will own that database and be able to do anything it wants inside of that database. Again, this might be more privileges than you want to allow in a production environment. Instead, what you might want to do is pre‑create the database and give the application just reader and writer access, but again, unfortunately, I cannot cover every combination and permutation of different options that are available. So now my applications should be a DB creator on the SQL Server. So for the next step, I just need to give my application a connection string so that it can reach the SQL Server. So you might remember, we have the ability to have different app settings for different environments, and because I do not have a username and a password in my connection string, I feel comfortable placing the connection string into an app settings file. Effectively, what I want to do is copy the existing app settings file, and then we're going to paste that again into the project and change the name. So I want to change the name of this to appsettings‑production.json. And the way this copy/paste worked, I actually also ended up with appsettings.production.development, which makes no sense. So let me delete that version, and I also missed the period, so let's do appsettings.production.json. Now, if I had an actual SQL user that I wanted to use, that is a login, with credentials like username and password. I would not want to store that password in this JSON file, especially if I was going to check this JSON file into source control. That would be a problem. Instead, what I could do, because of the ASP.NET Core configuration system, is set that connection string in an environment variable. That keeps the password out of source control entirely, but all we need inside of appsettings.production is a connection string that points to a slightly different database. So let me remove the logging statements here. Actually, this is a good logging statement for production. We'll stick with that. It's a warning. So let me remove this logging statement and allowed host, and the messages. All I really want to override when I'm in production is the connection string for OdeToFoodDb. The data source will no longer be LocalDB. It's going to be this local machine, though still. The initial catalog can still be the database OdeToFood, and I do want Integrated Security=True. That means my application will connect with the application pool identity, which since that has been granted access to the SQL Server and the dbcreator role, we should be able to execute the migrations and then connect to the database and start saving data and reading data. So let me save all the files. We're going to need to take one additional step here, which is now to publish this. So let me use the up arrow and go back and find my last dotnet publish that went in to c:temp\odetofood, execute this. Now, there might be some locked files that prevent me from writing everything I need into the published output directory. It might be that IIS has some files locked because the application is running. If that's the case, I just need to do an IIS reset either through IIS Manager or here from the command line. It's literally iisreset. Of course, you will need to be running as an administrator in order to execute that, but it looks like we have everything copied over. We have the new OdeToFood.dll. We should also have the new appsettings.production.json file over there, and I just want to do a quick check inside of the published output to see that indeed we do have, at the very top here it will be appsettings.production.json. So with that in place, let's come over to a web browser that is pointing to They should to https://localhost. I'm going to issue a refresh, which might take a little bit longer than normal. We will establish a connection to the database, execute those migrations, but it looks like I have successfully retrieved this page. There are no errors on the page, so I'm going to assume that we're connected to a database, and yes, I can even see the statement, there are 0 restaurants here. That means we did create the database so of course, there's no data inside of it, but we can certainly add a new restaurant, and we could call this Now in Production. That's a very nice restaurant to eat at, by the way. Its location is right here. The cuisine, we'll just say it's none, it's everything, that's production. We save that restaurant. Look at all restaurants. I now have a real SQL Server database and I'm deployed to IIS so I have a production environment here.

Conclusion

In this module, we covered a wide variety of topics, everything from the dotnetpublish operation to deploying to IIS and configuring SQL Server. I hope this gives you some tips and ideas for your own ASP.NET Core application deployments. And this concludes my course. I hope you enjoyed this course, and I hope it will make you a productive ASP.NET Core developer.