Introduction to ASP.NET MVC 4

Introduction

Hi, this is Scott Allen and this is the first module in the course design to demonstrate everything you need to build applications with ASP.NET MVC. This course is primarily aimed at developers who are new to ASP.NET MVC, but even if you've been working with the framework I hope to show you a few tricks. We'll be starting at the very beginning in this first module. I'll first show you how to install all the tools that you need. We'll be using Visual Studio express 2012 for the web which is a free tool for building web applications, but you can use other editions of Visual Studio 2012 if you want to follow along. By the end of this first module, you'll see how to create and run your first application and also have an understanding of the design goals of the MVC framework. So, let's get started.

Web Platform Installer

What we're looking at now is a machine with a fresh new copy of Microsoft Windows 8. I don't even have any development tools installed yet. So, I will go to a web browser and search for web platform installer. This will work on Windows 7 also by the way. If you haven't moved to Windows 8, but when I do that search I should see a link that I can click on to go to a page where I can download Microsoft Web Platform Installer. As this button implies this is a free download. It's a tool that you can just tell it to run and it will go out and be able to download software products and frameworks and libraries and install them on your computer. It just takes a little bit of time to initialize, but what we're going to look for is Visual Studio 2012 web edition. When I install that it will also install a web server that I can use to host my web application on this machine and a SQL server instance that we can use to save data for an application. So, now that it is initialized, I can come into the search box that's in the top right here and I will search for 2012 web and click enter. The very first search result here is Visual Studio Express 2012 for Web with Windows Azure SDK that is the product that I want. So, I'll click add. You click add. You can go through and search for other products that you might want to install, but you'll notice if you click on the hyperlink here. This will show you everything it's going to install here that includes Visual Studio 2012, some Windows Azure tools. It also includes IIS 8 Express and Microsoft SQL Express LocalDB Edition. We'll be talking about in using those two tools during this course and just selecting that one item will give me everything that I need. So, I will close this out and go ahead and tell the Web Platform Installer that downloads everything I need and install this. I'll need to accept some license agreements and chances are, it might need to reboot the machine once or twice, but we'll come back once installation is finished.

The Tools

Now, it's just a few minutes later and the Web Platform Installer has completed. It didn't require any reboot on this machine which is always nice. So, we're ready to use Visual Studio for the first time. But before we do, let's just a take a step back and talk about the tools we're going to use in this course. What we've done so far is set up the Web Platform Installer and you can launch this tool again at any time if you want to install more software. But we've already installed the three essential tools that we need to move forward. The first is Visual Studio. Visual Studio is where we will write task and debug our code. It includes editors, debuggers and shortcuts for all the technologies that we need to bring together in a web application. It can handle C sharp code. It can also handle JavaScript HTML and CSS. We've also installed SQL server. Later in this course, we'll see how to persist and query data that's stored in a SQL server database. And finally we've installed IIS Express. IIS Express is a light weight web server we can use during development and it doesn't require any special privileges to run. As a web server, this is the component that will respond to HTTP request that come in from a web browser and it responds by executing the logic that we build inside of our MVC application. We won't interact with IIS Express on a regular basis. It just pretty much sits behind the scenes and faithfully hosts and executes our application. The first step in using all these tools is to just launch Visual Studio.

New Project

Once studio is installed, you should be able to find it by searching your Windows start menu. And Windows 8 the start menu is an enormous screen full of squares and rectangles, but I should be able to find VS Express for web. I'm going to right click that and tell Windows to pin it to my task bar. That makes it really easy to launch from a desktop view in the future and then just click on the icon to get started. Visual Studio Express doesn't want you to register and obtain a product key. Again this is a free product, but you'll have to provide some basic contact information to obtain the key and keep using VS Express. But right now I'm going to cancel this and just move forward, but you'll probably want to register right away. The first time that you launch Visual Studio, it might take a little extra time to perform some initialization. But every start up after this will be considerably faster. Now, we're ready to go. Every great application that's built using Visual Studio starts by clicking on the new project link or you can also go to file new project. And inside of here you'll see templates create all sorts of different applications, Silverlight Applications, class libraries, of course the one we're interested in, is an ASP.NET MVC 4 Web application. You can create this using Visual Basic or for this course we'll be using C sharp. I'm going to give the project a name I'll call this OdeToFood because I want to build a website where I can enter restaurant reviews and I can select any folder on my hard drive even when it doesn't exist yet or I want to save this project and be able to work on it. Now, I can click okay and Visual Studio will ask me what template I want to use to create this project. In the future, you might see additional templates in this dialog box and the idea is that you'll pick the template that is the closest match to the type of application you want to build. The empty template is great if you know exactly what you want and you want to start from scratch. The internet application template is the one I'll be selecting. We'll talk about some of these other templates later in the course, but the internet application template comes with the default home page and all of the user interface pieces and infrastructure needed for users to register on the site and then log on and log off. For the view engine, I'm going to leave the default value here of Razor. A view engine in MVC is a component that works inside the application to help you produce the HTML that you need to send back to the client's browser. Razor is the preferred view engine for ASP.NET MVC, it's the one I strongly suggest you use. I'll also let Visual Studio go ahead and create a unit test project for me. The project will have the same name as my web project which is OdeToFood. It will just have dot Tests appended at the end. And with that I'll click okay and Visual Studio will go ahead and start putting together my project. One thing that's new with ASP.NET MVC 4 is all of the little pieces that go into this project. All of the different libraries like jQuery and jQuery UI. They are all added as NuGet packages. We'll be talking about NuGet later in this course too. But it makes all of these pieces very easy to update. So, if there is an update to one of let's say the JavaScript Libraries in my project. I can easily go out and install that update without manually downloading and copying files around. We'll see that, but for now, Visual Studio has created the project. It's ready to go. So, let me run the application, I can click the play button up here in the tool bar to run this application inside of Internet Explorer that will have Visual Studio compile everything that will launch IIS Express and also Internet Explorer and there we can see the application as running. That's a good sign. You can see that the template we pick, the internet application template, it give us a home page with some default colors and styles. We have a menu with working links. So, I can go to an about page. I can also go to a contact page. Notice the URL here is a nice friendly readable URL slash home slash contact. There is no file extensions present. And if we view the source code to this, we can see its nice clean looking HTML. In fact, that's HTML file because we have an HTML 5 doc type and MVC 4 includes some of the HTML 5 best practices, for instance, specifying the language from my mark up. Also specifying the characters set encoding. It turns out that it's important to have that meta tag with the character set encoding specified. If you want to avoid some strange and subtle cross site scripting vulnerabilities. We also have a meta view port tag by default and this is important for mobile devices. You can think of this as an instruction for how the page should render. So, without this meta tag, a mobile browser will assume that your page needs over 900 pixels to display properly. So, it will zoom out to fit all of the content on the screen. But because it's so zoomed out, the user cannot work with the site until they zoom in. But with this tag in place, we are telling a mobile browser that our site will adapt to the width of the device. So, please don't assume anything about how many pixels we need. This all works because of some instructions that are in the style sheet that are included in this application by default. I have a few other videos on Pluralsight that use MVC 4 with mobile devices and jQuery mobile, if you're more interested in that topic. We also have a link to Modernizer JavaScript library. Modernizer will ensure our HTML 5 mark up works with older browsers that were around before HTML 5 was given any thought. So browsers like IE 6 which is over 10 years old now. We can be sure that using the new HTML 5 elements like header and section and nav, they will appear properly on that older browser. Later in the course, we'll explore where all of this comes from and how to modify it. For now, let's just be happy that MVC 4 is keeping up with the latest standards and has given us a running web application to work with. But right now, I want to close the browser and return to Visual Studio to start poking around inside of this project.

Models, Views, Controllers

Here inside the Visual Studio you'll have a window with the title Solution Explorer. It is the Solution Explorer window that allows you to get to all of your files, C sharp files, JavaScript files, CSS files, icons, images and everything that goes into your application. When we created this application using the internet project template, Visual Studio went ahead and populated our application with all of the folders and files that you see here. That's why we already have a running application with contact in about links working. Three of these folders have a special significance, the controllers, the models and the views folders. Let's talk about those names for a minute. The ASP.NET MVC framework derives its name from the model view controller design pattern. This design pattern has been present in software applications for several decades at this point and it's a design pattern to follow when you want to separate the responsibilities of the components in your user interface layer. The C and MVC is for controller. A controller is a software component that will be the target for some external stimulus. In the case of a web application that external stimulus is usually an incoming HTTP request. So, when someone launches a web browser and points it to the slash home slash about location of my application. That incoming request needs to go to a controller that is in my application. When the controller receives the request, it's responsible for building a model that M in MVC. It's the model that contains all the information that you need to present to the user to satisfy that incoming request. In the case, of slash home slash about, the model might just be some information about the website or about the company or the people behind the website. Another example would be a controller for handling request to view recipes. You might have a recipe controller that builds a list of the most popular recipes. That list would be the model. The controller then selects a view to display the model. Views in the MVC design pattern are very simple objects. Think of them as templates. They take pieces of data from the model and they place them into a proper location on the page. If the model was a list of recipes, then the controller might select a view that will take that list and display the recipes inside an HTML table. That end result is that you isolate the behaviors in your UI into one of these three categories, model, view or controller. A view would never need to know how to call into the data access layer because the model already contains all the data it needs. Meanwhile, a controller would never need to know about where to place an error message or how to color it because that's the responsibility of the view. That isolation that you achieve with the MVC pattern makes it easy to maintain and change your application moving forward because the code inside of each of these pieces is very focused and easier to understand. You can make a change in the controller about where to get your list of recipes or how to calculate what the best recipes are and that shouldn't impact the view which is only worried about displaying the recipes. I do want to point out that the MVC design pattern doesn't dictate what type of data access you use. You can use web services, relational databases, file system, document databases, any form of storage behind the scenes. And it also doesn't dictate what your business objects or domain layer should look like. In fact, the MVC design pattern doesn't care if your application has layers or not. It's simply a design pattern for building a user interface and nothing more. So the MVC framework is designed to help you follow this MVC design pattern by giving you tools and classes to build models, views and controllers. The MVC framework also has some additional goals. One of these goals is to embrace the web and to be able to work very closely with web technologies like JavaScript, HTML and CSS. There is no large obstructions here that try to shield you from knowing things like what HTTP verb is being used to place a request. We'll learn about those things in this course. There is also nothing that makes it harder for you to work with the designer who can help you style and make your site look good using CSS. Another goal for MVC is to run on top of the core ASP.NET run time. The ASP.NET run time is been around for over ten years now. And it's proven itself to be secure and stable and optimized for performance. If you're already familiar with ASP.NET and you know how to work with HTTP modules and handlers and caching and diagnostics, then you'll feel right at home in the MVC environment. A third goal for the framework is to be extensible. We'll look at many extensibility points as we work through this course. If you do not like how the MVC framework behaves in some certain area, chances are, all you need to do is plug in the right component to make it behave a little bit differently. And finally the last goal of the MVC framework is to be testable. We will look at unit testing and tester and development techniques in this course and we'll see that MVC framework went to a lot of trouble to try and obstruct the ways of some of the thornier issues in testing user interface-oriented code. But right now what I want to do is return the Visual Studio and make some of these concepts a little more concrete.

Making Changes

Let's look at the details of what happens when we click on the about link in our application. The browser is going to send off a request to slash home slash about and that request will reach our development web server here on the local host which is this machine and that request will be received by IIS Express. IIS Express is running and it's in the background. You can see it if you go to the system tray area of the task bar. Sometimes it hides itself here because it's shy. But if you click on the little arrow, you can see that the icon looks like stacked blue pizza boxes and I can right click it to see more details like show me all the applications that you're currently running. I can see it's running my OdeToFood application and it's running it on port 56470. That's just a temporary port used for development. ISS express is going to take that request and deliver it to my MVC application. Inside of every MVC application, there is a routing engine which takes request and tries to deliver them to the proper component. We'll look at the routing API in the next module, but for now you're just going to have to trust me when I tell you that the request ultimately ends up inside of a class that's inside of the controllers folder. In fact, it ends up in a class called home controller because by default a request for slash home something will come to the home controller and a request for slash home slash about will end up inside of this about method inside of the home controller. So you can see there is a naming convention in play here. Where if you just name things correctly, request will find their way to the proper place and this isn't the only scenario where naming conventions play a role in MVC. The about method doesn't have much work to do and by the way we may call this an action in ASP.NET MVC. So the public method inside of a controller are actions. This action doesn't have much work to do. It's not even really building a model. But let me put a break point here by clicking out to the left of the editor window and getting that little red dot. Now, we can run it with the debugger and see execution actually pause here when I click on the about link. The debuggers are a great way to step through your code and see what it's doing and inspecting variable values. Let me click on about and you can see we hit this breakpoint. And now I can press F10 to step through code one line at a time. I can hover over things to view their values and so forth. The line of code we're on now is about to return a view. A view is one type of action result that you can return from an action that tells the MVC framework what to do next. Maybe you want to redirect the user or maybe you want to return some data in a format for JavaScript to consume it. Returning a view is telling the MVC framework that the next thing you want to do is render a view, but what view will the MVC framework render? Well there is also a naming convention in play here too. The MVC framework is going to look inside of the views folder of this application. Notice there is a folder called home that matches the controller I'm in, that home controller without the controller part of it. But these are the views for the home controller and if I expand that, I can see there is a view called about, that matches the action that I'm inside of. The about action of the home controller and if I open this, what we'll see is the information that is pretty much what we'll see if I hit F5 to continue and let this view render. This about view has a dot CSHTML extension that is just the default extension for Razor views. Notice that that view that appears on the left hand side doesn't include everything that you see in the browser. It doesn't include the text, your logo here or the register link or the log on link or home about and contact. It really just includes the content that is specific to this about action. All these other pieces including the copyright at the bottom, those are things that we need to appear on every page in this application. So, they have been factored out of the individual views like about dot CSHTML. We'll see how to build what's known as a lay out view that contain all that common mark up. For now, what I want to focus on is just how do I deliver additional information from the controller to this view and then have it displayed on this page. So, let me stop debugging and come back into the home controller and we'll make some changes here inside of about. There is a couple of different ways to get information to the view. One approach is to use the approach that we see here in the about action and that's simply to put information inside of the ViewBag. ViewBag is a dynamically typed object in C sharp. That means you can add any sort of property to the ViewBag and it will be available inside of the view to pull out and retrieve and display. That's currently what's happening with this message property. So it could also say something like ViewBag.Location equals Maryland and the United States of America and do a build and that value will be available inside of ViewBag also. If I want to display that, I could come into about dot CSHTML and let's remove what is currently inside of here and come up. We're already displaying ViewBag.Message inside of an h2 tag so let me just have a div that says location is at ViewBag.Location. The amp sign is a way to tell the razor view engine here is a C sharp expression. I want you to evaluate it and take the result and write it into the response right here where I'm placing this particular expression. So by having that in place there and saving the view, I should be able to come back out to the browser and refresh and see that we now have location mailed in the USA. So this is one approach that a controller can use to pass along information to a view but let's look at a different approach, an approach using what we call a strongly typed model. Every MVC project will start with a models folder inside. You can use this folder when you create new models for your application. Although in reality, models can live anywhere even in a different project that you reference from this project. But we use the models folder for now. All I need to do is right click this. I want to add a new class. I'll call this the about model. And we'll keep things very simple. This is a class, I'm just going to give it two properties. I'll add them using a code snippet in Visual Studio so I can just type P-R-O-P and hit tab twice that will expand out to a property. I'll give the first property a type of string. We will call this property name and I'll give us another property. This one also type string will include the location. And let's say that this about model represents everything that we want to show when the user visits the about page. So what I'll need to do is come into the HomeController and instead of using the ViewBag, I'll declare a new instance of our about model. The about model isn't a different name space that I'm currently in. So for this to be legal C sharp code, I need to include the name space that this is defined in which is OdeToFood.Models. An easy way to do that is just to use this drop down menu here that's sometimes hard to get a hold of, and click on this to add the using for OdeToFood.Models. A quick way to do this is when your cursor is on that unexpected identifier. Just hold down the control key and hit period. That will drop down the menu then select it using enter. That adds the using statement for me and we now have legal C sharp code. Now, usually when you build a model, you need to do some sort of data access or some calculations. But right now, we'll just counter up something. We'll save the data access stuff for later. I'll give this a name, you can use your name and I will say that my location is still Maryland in the United States. And now that the model is complete, I can tell the MVC framework that when it renders this view, you use this model. So pass this model along to the view. The next step would then be to modify the view to use this new model. So let me come into the About.cshtml view. The first thing I'll do up here at the top is tell the view about this model. So I'm going to use what's known as a directive which is at model and that's a lower case M, very important that you use the lower case M here, tell it that the model it should expect will be an instance of about model. And again in order for the name space to work, I'll have to hit control period and drop down and say that this isn't OdeToFood.Models.AboutModel, in this case it doesn't add a using statement, just adds the pooling name space qualified type name which is just fun. And now instead of going to ViewBag which we never knew quite what is in ViewBag. There could be a message property, there might not be a message property. Instead of using ViewBag, I will use the model property of this view. The model property knows what we're passing in as a model and quite often you can get IntelliSense, let me do a quick build by doing shift control B making sure the project builds, and then maybe when I hit period, I can see the IntelliSense. I can see there is a name, that's what we'll put out here. And I can see there is a location. So, let me switch the server to the model.location just to save everything, do another build, amount and refresh. And we're now building this view using a strongly typed model. And this is a very common approach to MVC development. So, let me just reiterate what we did here. This model property with the capital M at Model that will represent the model object that I passed in to this view from the controller action. And in fact, I told Razor that it should be expecting a model of type about model and now visual studio and Razor and everyone in the world knows that the object it should be receiving is an object of that type that it should have name and location properties. And what we've seen so far is really the essence of MVC, I have a request arrive for slash home slash about that will be delivered by the framework to this about action on my home controller. I build a model, pass it to the default view. When that view renders, it picks apart that model and puts all the pieces in the right location for-- to display correctly to the user. So, I can see this working, but how can I be a little more assured that this will always work for me? That's where unit testing come into play. So, let's look at unit testing with ASP.NET MVC next.

Unit Testing

As I mentioned previously, one of the goals of the MVC framework is to be testable. Specifically, your controller should be testable. I should be able to exercise the logic inside of the controller action from a unit test and be able to write assert to make sure that the logic is behaving correctly and I should be able to do that without involving a web server or HTTP messages, or my network card or other things that make testing complicated. Fortunately, when I created this application, we told Visual Studio to create a unit test project for us. And if I look in the solution explorer window, I'll see there are actually two projects in my application solution. The first one is OdeToFood that's the web application and the second one is OdeToFood.test. This is my testing project. Inside of the testing project are some test the Visual Studio added when I created this application, these aren't the most fantastic tests in the world, but they do demonstrate how to implement unit test. This is a new feature for Visual Studio 2012 by the way. In previous versions of Visual Studio the free express edition did not include a unit testing tool, but it 2012 we have the framework known as MS test. So, let me open up the one set of test that it's inside of here the home controllers test. And you can see that tests are identified by attributes. With MS test you need an attribute on every class that contains a test, that's the test class attribute that you see. This attribute helps the test runner which we'll use in just a bit to find all the test inside of this project. And then every individual tests inside of a test class has a test method attribute. What a test runner is going to do is once it finds your test, it will instantiate every test class and then invoke every method that's a test method inside of that class. It will then check to see if each of those test passes or fails. You can tell the test runner when something fails. Inside of every unit test, you typically have three steps. Each step can have one or more lines of code. The first step in the arrange step. This is where you create objects that you want to test. In this index test, we're looking at the arrange act just consisting of just instantiating the home controller. That's the object that we want to test. Thanks to the testability of ASP.NET that we see we can instantiate our controllers from anywhere and not have them fail just because they aren't processing a real HTTP request. After the arrange step comes the act step. This is where you set things in motion. You usually invoke a method or set a property, do something to the object that you want to test and have it produce a result which you'll capture and examine in the third step which is the assert step. This is where you rate an assertion that should prove whether or not something work correctly. In this test we're just asserting that the home controller produced a result or the result .ViewBag.Message was equal to some specific string that's here. The test is just making sure that ViewBag.Message property is set the way it expects it. As I said, this isn't the greatest in the world. There is no real logic or need to test it that exact string as a match, but it does demonstrate how to use the test framework and to test API and you can see the arrange act assert steps that are common to almost every unit test. The assert API by the way includes a number of different assertions you can write. This assert is the R equal assert. It's to make sure that two objects are equal. But you can also assert that something is true or something is false or something is not null. There are many different types of assert. If the assertion fails, the test will fail. To see what test are passing and what test are failing. I just need to get the test runner going. I can do that by going to the test menu and saying run all the tests, the shortcut key there is control R followed by an A and when I do that I'll see the test explorer appear, test that have a green icon next to them are passing test that's good. Test that have a red icon next to them are failing test that would be bad. Let's see if we can get a failing test, right now we only have three tests and they are all passing but I can break something easily. I'll come over into the home controller and remove this bit of code that is passing the model in to build the view result. That means my view will not receive a model object and if I run the application when it's in this state, and go to that about action, we'll get a runtime error because the view needs that model when we don't pass a model in to build the view result we'll get a null model property and when we try to-- dereference that by doing model.name we'll get the object reference not set to an instance of an object exception. I could avoid this with a unit test, a unit test that makes sure that the about action of the home controller sets a model object and that the model object is of the right type and having a unit test in place for that will mean I can see that it's failing before it ever try to push this somewhere where someone will see it, I'll catch this error before the application ever runs. Let's try to do that. If I come over into the home controller test, I can see there already is an about test here. But the about test is just making sure that the about action is producing a result, any result and that the result is not null. And so, the change I made in the home controller will still not break this test. Let's run all the test again. All the test are still passing. I need to do a little more in this test. I need to really assert that result.model is not null. That's the important part that the view needs. We need a result but that result also needs a model associated with it. So with that change in the test and we run all the test again, now we have a failing test. If I click on the failing test, I'll get down here in the bottom of the test explorer window, a StackTrace I can click on the top link here, it will take me right to the line of code that failed, in this case it failed because result.model is null and the assertion is making sure that it's not null and to fix that, I just need to go back to the controller and put the code back in place so I actually pass a model along, now I can hit control RA to run all the test again. And we should come out with three passing tests. So this is all the time we have to talk about testing right now but we'll return to this topic later in the course. This is just a quick demonstration that was intended to show you how easy it can be to test the logic in your presentation layer, specifically the logic that you have inside of your controllers. Controllers are relatively plain C sharp classes. We can instantiate them without having a web server or a web browser or an HTTP request running. And compared to the view, it's the controllers that are doing all the hard work. Views are just simple in the MVC design pattern we use them just as templates to control the placement of data to most of the unit testing effort in an MVC application, it was against the controllers and the models.

JavaScript and CSS

So far, we've started to see how views, models and controllers work together and we've also seen how you see this to write unit test for an MVC application. But I also mentioned earlier that another goal of the MVC framework is to embrace the web and that means being able to use standard web technologies like HTML 5, JavaScript, and CSS 3. We've already seen a bit about how to work with HTML. We know the razor views that are in our views folder emit this HTML and we've already seen that the project does set up to use HTML 5. But there's a couple of other folders we haven't looked at yet, the scripts folder and the content folder. Inside of the scripts folder you'll find JavaScript files. The JavaScript files here include the popular jQuery library as well as some other jQuery plug ins we'll be using later in this course, plugins like jQuery validation and jQuery UI. And the content you will find the CSS style sheet for this site. This file site dot CSS is the file that's responsible for all the color and font choices that you see in the application. Let me make some basic changes to the body style here and see how that's reflected in the application instead of a white background color, let's go with 3, 3, 3, and instead this or that color let's change that over to white. So essentially we're flipping the background and foreground colors. And now once I save that file and refresh this page, you can see that influence is how at least that top of the page displays. But let me make a few more tweaks to that file and come right back. Now I finished making a few tweaks to my CSS file. And I've and added an image to the project. And this is what the applications looks like now. It's still not the prettiest application in the world but this is what we're going to be working on throughout the rest of the course. And hopefully we'll be adding features and making look it better. If you are interested in looking at the source code for this project, just go to odetofoodmvc4.codeplex.com. From here you can come into the source code tab. And from in the source code section you can download the latest source code. You can also click this History link and go back and see what the code look like at specific point in time like this point in time with the first video is nearly over.

Summary

In this first module I've demonstrated how to use the Web Platform Installer to create a development environment or you can build ASP.NET MVC Web Applications. With just a few clicks we were able to install Visual Studio, SQL Server and the MVC Framework. We still how to create a new MVC application and we talked about the Model, View, Controller design pattern. In the rest of this course we will be again drilling into specific areas of the MVC Framework. And see how to build Controllers, how to build Models, send story information in SQL Server. We'll take a very close look at Razor and also how to use jQuery plugins to add Ajax features to our application. We'll be writing more unit test and then in the end we'll deploy our application live to the internet. Thanks to Microsoft Windows Azure websites. So don't go away, we're only getting started with this excitement.

Controllers in ASP.NET MVC 4

Introduction

Hi, this is Scott Allen and in this module we're going to take some time to understand how controllers work in the mvc framework. Specifically we're going to look at the routing rules which deliver in the incoming requests for from the web to our controllers and then look at controller actions. You might remember from the introduction that the controller actions are the public methods on our controllers. They have the ability to respond to an incoming http request from the web. We'll also be taking a look at action filters which can introduce pre and post processing to an action and then see how we can input data to an action with action parameters and output different types of results from an action using different action results.

Routes and Controllers

One of the questions that we left unanswered in the introduction was this, how does the sp dot net know how to deliver a request like slash home, slash about to our home controller. And the answer to that question is in the routing engine. The routing engine is a core part of asp dot net, it's not tied to the mvc framework, you can use the routing engine to route requests for web forms, WCF services, really any type of resource but in mvc we use this routing engine to direct requests to our controllers. To do this we give the routing engine a map to follow using a map route API. A route map allows us to provide a friendly name for the route, a pattern for the route and default parameters for the route, the most important pieces to look at are the pattern and the default. Think about the goal of the routing engine, its job is to examine a URL and figure out where to send it for processing. When it examines the URL it needs to pick out pieces of data from that URL, little hints about where to send the request. In an MVC application we want the URL to specify the controller and action to invoke and perhaps some other data so we give the routing engine a pattern it can use and apply to find parameters in the URL. The words inside those curly braces will be the parameter names, if the routing engine doesn't find the specific piece of data in the URL like the controller name or the action name it can use one of the default values that we specify in the parameter defaults. Let switch into visual studio and experiment with this. Inside a visual studio let me introduce you to the global dot asax dot cs file. This is a bit of a magical file in asp dot net. If I open it up you can see we have a class here derived from http application and this allows us to hook into hook into some application level events like application start, this method will be magically invoked by asp dot net before you process the first http request. So when your application starts running the code here will execute one time before any of your controllers start executing and so this is where we put in some configuration that the application needs like the routing configuration. The routing configuration is done by this line of code, route config dot register route, what we pass in is the global routing table the table that contains all the routs for the entire application. It will be empty initially but when we call register routes that will add entries into the routing table. I can get into the definition of this method if I just put my cursor tight here on the method and press F12. Now we are inside of a class called route config, this class actually lives in the app start folder, you can find it right here. It's given to me by default by visual studio when I created this application so all this code was already written for me and if you've been working with mvc previously this is a little bit different in mvc 4 than it was in previous versions because all of the start up code that used to be inside of global dot asax dot cs has now been factored into classes that are in this app start folder. All these classes are called from the application start method in global dot asax dot cs. Here's the code that you saw on that previous slide, we're walking up to the route collection and saying let's add a new route, we'll call it default and here's the type of URL you're going to look for so if you see something that looks like slash home slash index, treat home as the controller name, index as the action name and you can also look for an ID but if there's not one that's okay we've provided a default value for it. And in fact if you don't see an action name just treat the default action name as index. When the routing engine picks apart a URL into these pieces it builds up a data structure that it passes along to the mvc framework and then the framework can look at that data structure and say what did you find for a controller and what did you find for an action and it uses that information to direct the request tour controller like the home controller. In fact that information is available to us, it's available any where throughout the request. Let's swing over to the home controller and see if we can inspect these values. Instead of building a message that says something about modify this template, let me build a different sort of message. What I want to do is see what kind of controller we're inside of and I can do that by going to the route data data structure, this is the data structure that's built by the routing engine, it'll have a collection of values inside and I can basically walk up to the values and say what do you have for a controller value, I could also ask it what it has for an action value and finally I can say what do you have for an ID value if any route data dot value sub ID and I can put all these pieces together into a message, let's use string dot format and I want parameter 0 which will be the controller name, parameter 1 which is the action name and parameter 2 which will be the ID. So controller action ID and this is now the message that I'll display on the home page so instead of modify this template we'll put a message here and let me run the application. And what we should be able to see that just by going to the root of the application, just local host slash we get a value of whom for controller and a value of index for the action and what happens if we go to slash home, same values, home and index. So I can go to slash home slash index slash 5 1 2 and you can see you get the value 5 1 2 for an ID. What we're going to see in just a little bit is that it is actually very easy to get that ID parameter, we do not have to look inside of route data explicitly but for just right now I want to demonstrate the whole goal of the routing engine is to pick apart that URL and then build this data structure that is available not only to the mvc framework to figure out where it's going to go but also to us, it's available inside of controllers, route data is also available inside of a view. Typically our code is not going to be looking into route values like this, it's the mvc framework that does that to find a controller and you'll notice that the value of the controller which is home tells the mvc framework to go out and append controller to that word home controller and look for a class by that name. It's a case and sensitive search.

Actions and Parameters

Now that we know a little bit about how routing works let's go into route config dot cs and define a new route, this is only something you need to do if a default route doesn't work for you but let me show you a scenario where a new route can help. Let's imagine that the user expects to be able to come into the application and search for a cuisine by its name, so French cuisine or Italian cuisine, in this case at second entry in the URL the second segment of the path isn't really an action identifier it's more of a parameter, it can be French or German or Italian. So the default route really won't work very well for us because we don't want to add an action to our controller for every type of cuisine that we have we just want that path sent to the action as a parameter. So I'll define a new route for this and I need to be careful where I define this route because the order in which rotes are added to the global route collection is significant, what the routing engine will do is evaluate each mapped route that we place into the route collection the first one that matches the URL will win. The default route entry that we have here is very greedy; it matches nearly any URL that you want to throw at it. It matches slash home slash index, it matches slash home, we've even seen it work when you just go to the root of the application and it can do this because it provides default parameter values to anything that's missing in the URL. IF we want to define our own route we'd probably want to place it in front of that default route and make that URL a little more restrictive. So for example let me map a route and we'll give it the friendly name of cuisine and we'll say the URL to match for this has to start with the word cuisine and then it can have a parameter which we'll call name which will be the second segment of the URL and you might say how can this work there's no control, there's no action. Well, the mvc framework does need to know what the controller will be and what the action will be so let me add those as defaults. I'll create a new anonymously typed object and say controller equals the cuisine controller, the default action, let's give it a default action of search and name will just make it an empty name and that defines our new route. Now if we have a request come in for slash cuisine the routing engine should see that, that matches the cuisine route and send things off to a cuisine controller. But if a request comes in for slash home, slash index that doesn't match the cuisine route it doesn't start with the word cuisine so we'll go to the next route entry which is the default route entry slash home slash index will match that one. As a side note let me briefly mention the purpose of this routes dot ignore route method call, the routing engine will not try to process a request that going to reach a real file on the file system. That's why we can have a request for something like slash content, slash site dot css and the routing engine won't interfere with that request. The server can simply send back that css file in response, we don't need this request routed to a controller, the file already exists we just need to pick it up off the disk and send it back to the browser and that's what will happen because the routing engine will see that, that's going to reach something on the disk and it doesn't interfere and try and route things around. But there are some files served up by asp dot net that end with a axt extension and these files don't actually exist on the file system they're virtual files but asp dot net knows how to serve them up. If you've ever worked with asp dot net tracing you might be familiar with trace dot exd. We'll take a look at a situation where we have an axd end point later in the course but for now just know this is a way to tell a routing engine if you see something like trace dot axd in the request just ignore it and don't try to route it, someone else will take care of that even though the file doesn't actually exist on the disk. Now that we have our new route to find let me run the application by pressing control F 5 and let's try to come into the cuisine controller and look for Swedish food. The response I get back from my application is an acdp 404 error, this is the error code that is recognized throughout the noon universe as meaning I cannot find what you're looking for which is exactly what I would expect because I told the mcv framework to go looking for a cuisine controller but I do not have a cuisine controller in the application. So let's fix this problem, I'm going to right click on the controller's folder and select add controller, this will bring up a dialogue box where I can give my new controller a name, I'll call it the cuisine controller. There's a number of other options inside of this dialogue which we'll talk about later in the course, for right now I'll just leave the empty NBS controller template selected and click add, this will add a controller to my controller's folder. If it isn't quote empty it does have an index method inside, it is derived from that base controller class and if I build this application and run it again, refresh the browser we'll still have a 404 error message because we also told the mvc framework inside of val config that the action should be the search action and this controller does not have a search action. There's also no way to specify the action in the URL we didn't provide for that flexibility inside of this route. So we really need to call the index action, the search action, we also don't have a view as yet but that's okay I'm going to do the simplest possible thing inside of an action which is just to return content. When you return content you just send a string back down the browser, there's no html involved unless you put html in the string, there's no view involved, I could just say hello, just to prove that we're reaching the search action properly and if I refresh again now we have a result, we have reached that method inside the controller. Now let's see if we can pull out that name parameter out of the URL but before I do that let me just oint out that actions are nothing more than public methods inside of a controller class. Anytime you add a public method to this class or any controller class you have to think about it as something that will be URL addressable depending on your routing configuration. If someone types the right thing into their browser address bar they might be able to invoke a public method inside of a controller so don't add any methods inside of here that you wouldn't expect to be called via a URL. So just keep that in mind as you move forward. Right now we're going to try to pull out the name value from the URL and I've already demonstrated that we could get to that through route data, I could ask for route data dot value sub name but it turns out asp dot net mvc makes this even easier because if you add a parameter to an action what the mvc framework will do is go out and try to find something that matches that parameter name and then just give it to you. It will do everything it can to populate that parameter, it will look all around the request it will look in routing data, so things that were picked out of the URL. It will also look in the query string and it will look in posted form values. In the case of coming to slash cuisine slash Swedish the mvc framework will see I need a parameter called name and something called name was extracted from the URL and it will automatically pass that into me. What I want to do is just echo that fact when I return this content but before I do that what I want to do is make sure I properly encode this value, so I'll say message equals server dot html encode name server is a property that I inherit on my controller that I can use to get to server type utilities and variable including html encode which ill make sure if a user snuck through some sort of malicious script tag or something like that it will render as text and it will prevent a cross-site scripting attack. The razor view engine will do this automatically for me but since I'm using a content result I have to be a little bit more careful about how I manage user input. Let's just take that encoded message and return it as the content of our message, now I will do a build, I will come back and refresh the browser and you can see we get the name of the cuisine even though it is misspelled, we get the name of the cuisine back out and we can show it in the browser. Now currently the way our route is defined I can come to the cuisine controller and not pass in a cuisine name and it renders just fine that's because in our routes we said that if you do not find a name just use and empty string. If I take this default out of here though something else will happen, let me do a quick build and refresh now we're back to a 404 error message because essentially what has happened is that we've told the mvc framework that an order to match this route you have to have a name it's not optional, we didn't provide a default value for it, therefore we didn't match the cuisine route we went to the default route, we probably went to the index action of the cuisine controller which doesn't exist and that's why we have this problem. So let me put this back in and instead of specifying an empty string I'm going to specify URL parameter dot optional, that's just telling the mvc framework it's okay for this to not exist inside of the URL and now if I do a build and refresh we're getting to the cuisine controller again there's just no value present for the name. If I wanted to provide a default value I could say name equals French or name equals Swedish, I could also do that using c sharp and saying that this has a default value of say French and now if I do another build and refresh the french value comes through even though I didn't specify that in the URL and just to show you that the mvc framework is looking around in different places for something called name let me enter a query string so I'll go to the cuisine controller but say query string name equals a Italian and we get Italian comes through so even though the routing engine didn't find a name in the URL the mvc framework found something called name in the query string and it just passed it into the action force automatically. So you almost never inside of a controller action directly inspect route data or directly inspect the query string which you could do by going to request dot query string inside of a controller method, all of those things are taken care of for us by the mvc framework, it's just going to find things in the request and pass them into us. And of course it should still work if I say slash cuisine slash Swedish and actually spell it correctly this time that value of course was found in the URL. As we move further along in this course we'll see more advanced scenarios involving multiple action parameters and even complex types here with multiple properties and see how the mvc framework behaves with those.

Action Results

At this point we've used routing data and query string values as inputs to our cuisine controller search action and that will prove to be useful knowledge later when we start doing data access and need to use those incoming values but for now I want to talk about the output of an action. We just used the content result to return a string literal from our action and the producing method which is a factory type method that produces and action result, its called content. So that's simply the method that you need to invoke to create a content result. There's many other types of action results available the name column represents the type that is derived from action result, the type of the object that you'll create and produce in return from an action. So all of the class names that you see here they all derive from action result and the producing method is the factory method you can invoke to construct a particular result, it puts things together for you. Previously we've also used the view method to produce a view result and we saw in the introduction how this will render a view that is somewhere inside of our views folder, there are also results we can use to return a file back to the client, return data in java script object notation for java script to consume and also a result to tell the browser to redirect to a different URL. Let's see how some of these different action results behave by switching back into visual studio. We've seen how the content and view results work, let's try a different result, let's look at one of the redirect results, you can see just in the intelliSense window there's many types of redirects, for instance there is a redirect permanent, this returns an http 3 or 2 status code which would basically say if someone came to this cuisine controller it would say go over the Microsoft dot come and never look back this is a permanent redirect, you never have to come here again. There's also redirects that you can use to go to specific actions in the same controller or an action on a different controller that is in this application so I could say when you come to the cuisine controller what I want to do is redirect you to the index action of the home controller and I can even pass along values with this. This is a temporary redirect and I'll show you later in the course why these types of redirects are very common for some scenarios. What's interesting about this redirect to action is that it actually coordinates with the routing engine behind the scenes, it walks up toe the routing engine and says if I want to go to a URL that would reach the index action of the home controller what would that URL look like and based on the way we've configured the routing engine, the routing engine will reply with the proper URL and you can even pass along additional parameters, if the parameters are known to the route they're actually in the URL, the routing engine will produce a URL that has the value in the URL but I can also say let's go to the index action of the home controller and pass along the name so the way you do this is to pass an anonymously typed object in as the third parameter, mvc framework will pick that apart and decide that okay you need to get a parameter called name over here. So what would this look like, let me do a build, we're currently at cuisine slash Swedish so the cuisine is Swedish but after I've built the application if I refresh this we should end up back on the home page and notice the URL is just slash query string name equals Swedish because there is no name in the URL when we go to that default route to reach the home controller and so the framework decided to build this URL with the name parameter in the query string. We know now that it's very easy to pull that value out of the query string if we need it and this was a redirect result so if you're not familiar with that the browser of requested slash cuisine slash Swedish and the server responded with an http code and a location basically saying no you don't want this URL you want to go over here so the browser had to issue another get request and come back here to the home controller, that's how we ended up here. You can also if you need to redirect to a route by name so here I could say redirect to route and say please redirect to the default route and now I could plug in values if I wanted to, I could say please pass along so that we reach the home controller and the about action, so with redirect of the route you don't pass the controller and action name was parameters you have to pass them in an anonymously typed object but those values will match up with however the route was configured. Let's try a couple other different type of routes, just for kicks let's return a file result, so I can return file and here I can pass in the file contents if I already have them loaded into a byte array, there's different overloads of this, just pass in a stream if I already have a stream open or pass in a file name the mvc framework will figure out how to pick that up off disk and pass it back. I'm going to pick the overload here where I just need to pass a file name and a content type. So this will be a little bit weird but let's just say when the user comes to the search action of the cuisine controller we want to return our site style sheet, site dot css it's in the content directory. So I'll need to come up with the physical path to that file, I can do that...do that server property here it is one more time, this time I'm going to use the map path method on that and from here what I can do is pass in a virtual path, the path that's in my web application slash content slash site dot css and server dot map path will turn that into a physical path that the operating system understands and I can say that the content type of this is text slash css. Let me do a build and let's see what happens when I come back out to the browser and refresh the cuisine controller this time, now what I'm looking at is essentially our style sheet, site dot css, we returned that as the action result. Let's try one more, let's come back and return a Json method and then pass in an object. Behind the scenes what will happen is the framework will run a java script serialize over this object to turn it into java script object notation. And this object can be pretty much anything, it can be one of your models, it can be just an anonymously typed object so I could say give me an object with message equal to that current message that we computed and I'll just throw in something extra just for the fun of it so I don't have a name property called Scott and I need to add one more parameter here to allow this to work when I get request, I need to explicitly say allow a get request to retrieve this Json and now do a build. Let's come back to the browser and refresh and there you can see Json appears in the browser window, message is French name of Scott. Later in the course we'll see just how easy it is to take that and consume it from java script then turn it into anything we want, turn it into an html display.

Action Selectors

When the mvc framework is selecting one of your controllers public methods to invoke as an action it will use any action selector attributes that might be present to find the correct action to invoke. One such selector is the action name selector, when you apply this attribute to a controller action it specifies the action name for the method or the edit method that we see here we can no longer reach this method as an action named edit we have to reach it as an action named modify, action name is an attribute you can use when you went to alias, the name of your action methods. Accept verbs on the other hand will specify the http verb that is allowed to reach a particular action, you can say that an action method is only reachable with an http get request or an http post request or delete or put or some combination of those verbs. The accept attribute will become extremely important later when we start modifying data, we'll be using both get and post then but right now we're just using get but I'm introducing this topic now so you're ahead of the game. Let's see how this works in visual studio. Inside of our search action let's change it back to returning a simple content result with a message that we've computed but I will make one more change I will add an attribute here that says that you can only invoke this during an http post message and now if I do a build and refresh the browser on slash cuisine I'll get the 404 error message, essentially the mvc framework could not find an action it could invoke in response to this get request. A get request is what a browser sends off by default if you just type some something into the address bar and hit enter. On the other hand if I had specified http get and now do a build and a refresh everything should work just fine because now the mvc framework can find an action to invoke and we see French again because that's the default value for this parameter. Let me give you a preview of why this is extremely useful. What happens if we had two actions both with the name of search, the first problem here is that the c sharp compiler will be unhappy because these methods have the exact same signature, the same name or the same number of parameters we wouldn't be able to get this to build but let's take the parameter off and instead of returning content with the message, let me just return content that says search and now do a build and come out and refresh the browser. We'll get a different sort of error, the mvc framework sees that it should invoke a search action but it's not going to try to distinguish between the two search actions that we have, one that takes a string, one that doesn't have any parameters it's just going to throw up its hands and throw an exception. There's some scenarios later in this course where it's going to be very useful to have two different versions of an action and have one respond to an http get request only and have the other one respond only to an http post request and now by placing these attributes here the mvc framework can clearly see that if it's a get request for search it should go to this action and display research, if it's a post request for search it will come over to this one. So I just did a build and if I refresh now we should get rid of the error and we get a search text. Again that's because the mcv framework chose this one, it was an http get request, it saw the search action...it saw that this first one was only legal during a post so it selected the second one. Just remember this discussion when we come back to modifying data later.

Action Filters

In addition to the action selector attributes we just looked at there are also a number of action filter attributes you can use. Action filters apply pre and post processing logic to a controller action and its result. Action filters are the components that you want to use to apply cross-cutting logic that's logic that has to execute across multiple controller actions but you don't want to duplicate code inside of individual controllers. One example is the output cash attribute this tells the run time that it's allowed to cash the final output of some action and to use that cashed result to service future requests. When you apply this attribute in the right places you can dramatically increase the through put and scalability of an application, we'll take a look at cashing later in this course. Another example is the authorize attribute, authorize allows you to ensure a user is logged in and perhaps a in a specific role like the admin. role before the action is allowed to process that user's request. We'll be looking at most of these filters during the duration of this course but right now I just want to show you how to apply a filter and also take a quick look at building your own custom action filter. Here inside of visual studio let's get rid of our second search action result and go back just to a single search action result, we'll have it respond to both a get and a post and I'm going to use the authorize attribute. You can just use the authorize attribute without any parameters, you can also specify something like roles equals admin., that means the user has to be logged in and they have to be in the admin role before we can invoke this function, you can also specify user names inside of here, I'm just going to use the authorize attribute without any parameters, what that tells the mvc framework is the user has to be logged in in order to be able to use the search action. Now if I do a build and I come back to refresh the application and what has happened is we've been redirected to the log on page because I've tried to go somewhere that requires me to be authorized, it requires me to be authenticated and the run time will see authorized attribute and it's smart enough to redirect me to the log on page where if I were to register on this site I could create an account and the log in. After I log in I'd be redirected back to the cuisine controller where I originally tried to get to, back to the original URL that I tried. We'll be looking at authentication and authorization and other security topics in a later module but right now I just want to focus on action filters and I want to demonstrate that you can place them not only on an individual action but you can also place them on a controller and when they're at the controller level they apply to all the actions inside of that controller so again if I tried to get to the cuisine controller now that we've rebuilt I'll still be redirected to the log on page, I need to log in before I can get past that authorized gate keeper. In addition to applying action filters at the action level and at the controller level you can also have what's known as global filters. Global filters are typically registered during application start up which means we register them during the application start event to let me return to the magical global dot asax dot cx file and there's the application start event one of the calls that you can see in here is a call to filter config dot registered global filters, I'll put the cursor right on that method, press F12 and we'll jump over to that class which by the way it's also inside of the app start folder. Here you can see that we're registering a single global filter, it's the handle error attribute. A global filter like handle error attribute will be in effect for every single request that is processed by any controller inside of your application. The purpose of the handle error attribute is to display a friendly error page to users when something goes wrong. Let's see how this works, let me return to the cuisine controller, we've commented out the authorized attributes so we should be able to get inside of this action without logging in and we'll pretend that something terrible has happened during the execution of our program and we'll throw an exception that's going to escape the action. That means if I come back to the cuisine controller, let's go to cuisine, what we'll see is affectionately known as the asp dot net yellow screen of death. It's the screen that shows you there was an un-handled exception on the server and it gives you a stack trace and some additional information that you as a developer probably will find useful because it can help you track down a problem but your users don't want to see this in fact it can be dangerous because it can give away information that someone who is up to no good can use to try to attack your server. But in actuality your users will probably never see this page because by default this level of detailed error information is only shown when the request originates from the same machine that the application is on so on the local host. If we want to see what the user is actually going to see we just have to configure our application a little bit differently. Configuration is another topic that we'll cover later in this course but for now let me introduce you to the web dot config files that is inside of this project, this controls configuration settings for the entire application and one of the things I can configure in here is how to handle errors, I want custom errors and I want the mode to be on by default it is that third option there, remote only which means it only shows a pretty error page to remote users. I'll just type in that xml element that's inside of the system dot web section, save the web dot config file and now let me come back and refresh this page, the application will restart any time that you twiddle with the web dot config file and this is the pretty error page that the users will see. Of course pretty is in the eye of the beholder but the reason we end up here is primarily because of that handle error attribute, it is providing post processing logic on an action and when it sees that an exception has escaped from an action it will display an error view instead of the yellow screen of death. The error view is actually in your application by default you can find it if you go into the solution explorer window and look under views shared, views that are in this folder are available to controllers throughout the application, there's an error dot see html view here so the error view is just a razor view like all the other views we are using and it's inside of here where you can provide some additional information or additional messages or give someone the number to the help desk so that they can call them in the middle of the night. Let me just change that view and if I refresh we'll be able to see that change reflected in the browser. Let me also demonstrate that if that if we were to go into our app start filter config and comment on this line it places the handle error attribute into the global filter collection then after I do a build and refresh we get a slightly different error, well actually it's the same error it's just that asp dot net is displaying a different page it's not giving out a level of detail in this error because we have custom errors mode equals on we're going to see exactly what the user sees here, they're not going to see a stack trace they're just going to be aware that something is wrong on the server. What I should do is come back and put this filter into place so that we have friendly errors and also go into web dot config and change this mode to remote only, that means our end users will see custom error pages but when we're developing here on local host we'll still get to see stack traces which can be very useful for debugging. Now before we finish this module I also want to demonstrate that it's very easy to build your own custom action filters. I'm going to go to the filters folder that's already in this application and click add class, what we're going to build is a logging attribute so I will call this log attribute, that will create a class, log attribute, I'm going to derive it from action filter attribute that is in the system dot web dot mvc name space and once I derive from the base filter attribute class what I can do is override one of four important methods. On action executing is a method you can override to look at the request before an action even executes where as on action executed is a method they you can override and you get to poke around inside of what happened at a point in time after the action executes. You can also catch before the result is executed and after the result is executed so for instance before a view vendors and a view vendors and these action filters are very powerful, you can change the environment, you can change results you can change parameters, let me just go ahead and override these methods. Let's override on action executing and override on action executed, override on result executing and override on result executed and then I will come in and press F9 at the top of each of these methods to set a break point and now let's apply this attribute to our cuisine controller and see how it behaves. All we need to do that is place it on an action or at the controller level, this log attribute is in a different name space so I'll need to bring in the name space ode to food dot filters, but at this point I should be able to run with the debugger and see where we break. Let me go to slash cuisine and you can see we get inside of on action executing even before we get inside the action itself and one of the things I could look at inside of here is the filter context. From filter context you can find out all sorts of things, you can find out what parameters are being passed to an action, you can add parameters, you can modify parameters, if I continue on and press F5 you can see now we're in the action, we're throwing and exception, something terrible has happened, press F5 to continue again and now we're in one action executed, this also has a filter context, this filter context can have more information on it because now the action has executed. You might have a result that you want to inspect or you might have an exception that you want to inspect so here we can see that an un-handled exception escaped from that. If I was really writing a log attribute I would probably record that exception somewhere, I'd probably also be recording what time the controller started executing and what time it finished and I could do all sorts of interesting and useful things with attributes. I'd probably also install this log attribute as a global filter attribute so it would execute for everything inside the application but for now I just want you to know that these are available when you see attributes on actions and controllers you'll have an idea of what they can do and you can see how easy it is if you ever need to write your own attribute, how easy that is to do.

Summary

This module was a stepping stone to give us some of the foundational knowledge that we need to know about how mvc works before we can move on and build out our application. I demonstrated how to use the routing api's to route requests to controllers and how to receive parameters in my action methods that arrive in the URL and query string, we also looked at action filters and different types of action results, these action results will come in useful in future modules.

Razor Views

Introduction

Hi. This is Scott Allen and in this module we're going to build views with the Razor view engine in ASP.NET MVC. The topics we'll cover in this module include the Razor syntax and how to transition between c sharp code and mark up inside of a Razor view. We'll also see how to use HTML helpers to keep our view simple and how to work with a layout view that defines the structure of our user interface. No module on views would be complete without a security discussion so we'll also talk about how to avoid cross-site scripting attacks and how to use HTML encoding. Finally I'll also show you how to use partial views and when they can be helpful.

Razor Basics

The Razor view engine allows us to use Razor templates to produce HTML. Razor views if you remember from previous modules, those are the files with the CHTML extension. Think of them as templates that consist of mark up which is your HTML and C sharp code expressions that are evaluated to place data into the mark up. So if I have a view and my model is a RestaurantReview I'll take properties of that review like the restaurant name and address and place those property values into specific locations in the HTML. When it all comes together I render a full HTML page of the client's browser that gives them all the information that they want to see. Let's look at this in an application to make the concept all the more concrete. Imagine that in the application we want to have a place where you can come in and see the latest reviews in the data base, we'll call that slash reviews. We won't be working with the data base just yet, we're going to focus on views but we cannot even get to a view yet because we do not have a reviews controller and as we saw in the last module, that leads to a 404 error message. So let me right click on the controller's folder and add a new controller. We will call this the reviews controller and instead of using the empty template I'm going to pick the MVC controller with empty read write actions. This will give me an index method the default action for a controller but it will also give me other actions that will ultimately allow me to edit a review. Create a review, delete a review, none of the code will be there as yet but we can fill it in as we go along. It's the index action where you typically would display a list of everything. But we don't even have anything that represents a RestaurantReview yet, so let me also go and add a class called RestaurantReview and add this to the project. We'll give it just a few simple properties. Ultimately one day we'll have a data base behind this and we'll need some sort of property to hold a primary key so I will add an integer ID property to the RestaurantReview. We'll also give this a place to store the name of the restaurant, the city that the restaurant is in, the country, and finally some sort of rating. And that's all I need for a RestaurantReview. One day, this will be in the data base. But for right now in the reviews controller, what I'm going to do is come down to the very bottom and just give us some in memory data to work with, so we don't focus on data access yet, we can focus on views. So I'm going to paste in some code that is just a static private list of RestaurantReview. I need to bring the name space in the scope using OdeToFood.Models and now this is legal C sharp code and this is not something I suggest that you usually do which is keep around a static list of in memory data in your controller that never works. This is just so that we don't have to get a data base involved just yet. What this will allow me to do is to come back into my index action and say that my model for this action is going to be from R in reviews, we'll do a little link query here, let's order by R dot country and select every review. And that will be my model for the view that doesn't exist just yet. But even though it doesn't exist it's still educational to build this application and come back to a browser and refresh to see how it behaves. And what we're looking at now is not the 404 error that we have before but we still do have an error. We've told the MVC runtime that we want to render a view and the runtime has gone looking for what we call the conventional view. Since we're inside the index action of the reviews controller the MVC runtime went to the reviews folder and looked for something called index, that has the same name as the action. And it looks for a couple of different types of index files because there's actually two view engines that are registered by default in the MVC runtime. One view engine is the web forms view engine and web form views have the extension ASPX or ASCX. The other view engine is the Razor view engine with the CSHTML extension, and since both of these view engines are around by default, the runtime is looking for all these different types of files. And it looks for these files in the Views slash reviews folder since we are in the reviews controller. The convention is to have a views folder with the same name as your controller, but the runtime also went looking in the Shared folder because views on the Shared folder are as you might have guessed from the name, shared among all of the controllers. But the runtime didn't find that view anywhere, we need to create that view. An easy way to add a view is just to right click inside of a controller action like this Index action and select the Add View. You can also hit control M control V. This will launch a dialog box and the dialog is rather smart, it knows on inside the index action. So it wants to create an index view which is perfect. I do want to use the Razor view engine and I want to create a strongly-typed view. Remember in the introduction, we talked about strongly-typed views. This is where the view knows the exact type of model object that you are passing into it. I'm going to tell it that my model is a RestaurantReview. In fact, I'm going to have a list of RestaurantReviews. And I'm also going to pick a scaffolding template of list. What scaffolding in Visual Studio will do is reflect upon your model object, figure out what properties are interesting and then build out a basic UI for you. It will create a view and it's something that you can go in and modify and customize. But it gives you a starting point and the list scaffold template assumes that you have multiple model objects that you need to display. There's also a scaffolding to create, and edit, and delete, we'll be looking at those in this course too. I'm going to leave all the other settings in place and just click Add, and you'll see what happens as we now have a Reviews folder in the views folder. That reviews folder has an index view. The index view itself is strongly-typed against the IEnumerable of RestaurantReview so it expects a collection of RestaurantReviews. And it's going to build a table to display those multiple reviews. The table will have header cells with the name the City, the Country, the Rating, and then for each item in the model, we'll be writing out a table row with cells that show the name, the City, the Country, the Rating and then a whole bunch of links. We're going to come back and talk about the syntax and what this HTML thing is. For now, let me just save everything, do a build to make sure everything is up to date and I should be able to refresh my view. And now, we have a list of reviews on the screen.

Code Expressions

Inside of a Razor view like this index view for RestaurantReviews our responsibility is to take the model object given to this by the controller and present the model in a user interface. And the user interface in the web application is generally built using HTML. As we talked about earlier, you can think of Razor as a template, where you combine literal text with pieces of data from your model that you pull out using sharp C sharp code. The literal text is like the h2 and Index text too in this view. If I want a view to display a more descriptive header, I can change this text. Let's say that this will display, the Latest Reviews. I can just save a view, refresh the browser and I can see that literal text appears here. Razor just output that without changing anything, the text gets sent to the browser where the h2 element becomes a level two header, so it gives this a bit of a title effect here to announce that these are the Latest Reviews. When you see an at sign in Razor, chances are then you are not dealing with literal text because an at sign can introduce C sharp code and the C sharp code can contain an expression for Razor to evaluate. So I can use C sharp code to reach into my model and pull out data. I could have something like we are showing the latest at Model.Count reviews. Save this, put parenthesis because count is a link operator, it needs to be invoked and now refresh the view. And we get-- we are showing the latest 3 reviews. So we didn't output at Model.Count, instead Razor saw that that was C sharp code, it evaluated it, it produced the number 3, so it sent that number 3 down to the browser. And technically, you don't need to only access the model in the view. You could also access ViewBag. In fact, you could access anything. Let me just put out a property that is inherited by a view. The views VirtualPath and save this. And when I do that I can see that the VirtualPath property contains the path of the view in this application. So I don't need to stick to just the model property. I can go anywhere to pick up data but remember there is no data access or calculation logic in the view at least not heavy calculation logic. Because data access in calculation logic, that's what controllers and models do. And while I'm looking at it, a tilde is something we've seen before but I'm not sure I explained it. A tilde represents the root of the application so tilde slash Views would mean go to the view folder from the root of this application. It's common to use a tilde everywhere you specify a path because without the tilde you need to be careful of your app that gets deployed into a subdirectory. You have to know the name of the directory where the app is deployed in order to hand out correct links. And that gets very messy. So anytime you need to point to something in the application whether it's a JavaScript file or an image or another view, using the tilde is a safe and easy approach. Let's go back to the code we have before. I'll hit Control Z a couple times and we're back to at Model.Count. You can think of the at sign here a bit like response dot right if you've ever done that in ASP.NET or classic ASP, the result of the C Sharp expression Model.Count is sent down to the client essentially with a response dot right but there is a significant difference between this code and response dot right because Razor will automatically HTML encode any output sent through the at sign to help prevent cross-site scripting attacks. Let me demonstrate this. If we switch over into the ReviewsController and come down to our data that we're displaying, what we could do is change the city for the Marrakesh restaurant from Washington D.C. to something like script alert XSS which stands for cross-site scripting attack slash script. So what this represents is what would happen if somehow, someway a malicious user was able to get something into your database that was a malicious script. And if I build the application and refresh the browser, what we'll see is that Marrakesh, the city is now script with an alert box. Now an alert box is just an annoyance. But if someone actually did get a malicious script into the database and Razor was not HTML encoding this, the browser would actually see a script tag and execute that JavaScript. Cross-site scripting attacks can do terrible things. They can steal cookies from users, they can display fake password dialogs and steal user's passwords. Cross-site scripting attacks are very dangerous and unfortunately they are the number one vulnerability on the web. So you have to be very careful when displaying data that the user has given to you and make sure you HTML encode that data which again fortunately Razor does that by default. We can see that if we do a View Source here and I do a quick search for XSS, instead of actually outputting a less than character which would allow the browser to see that this is a script tag and start executing JavaScript, instead Razor converted that less than character into ampersand LT colon which is a way to tell the browser I want to display a less than character. And so now what we have is a script that's showing up in the city which is a lot better than actually having executable script. And just to show you what happens if you really want to display something without HTML encoding, you can do that in Razor. What we'll do is we'll come down to this line of code that is displaying the city and we'll talk about what HTML.Display for does in just a bit. But the way to do this in Razor is to use HTML.Raw, that's a way of saying take this expression, take the string, do not HTML encode it. So if I just put item.City in here, save everything and refresh the browser. Now we have actual executable script on the page which in this case is an annoyance if that really was a malicious script, terrible things could happen. So let me quickly get rid of that script. I'm into the ReviewsController, just set Control Z to get back to Washington D.C. And now we can move forward again. And now, let's look at item.Rating. I'm going to comment out this line of code because we're not quite ready to talk about what display for does. I just want to show you that I can also output the item rating juts by saying at item.Rating. And if I save everything and rebuild, we've gotten rid of our script but we should still be showing the rating as a 10 which is correct. And the at item.Rating is what we will call an implicit code expression in Razor. We'd let Razor figure out what a C Sharp code and what is mark up. It's smart enough to know that at item.Rating is a C Sharp expression whereas less than slash TD are going to back into mark up. But what if we have some text displaying right after this rating? So the rating is on the scale of 1 to 10 but if we wanted to let someone know that by saying 10 out of 10 or 3 out of 10. I just put a slash 10 here. There's two ways to interpret this. You could say take the rating and divide it by 10 because this is all a C Sharp expression or show slash 10 as literal text. And by default what Razor will do is figure that that is text, not part of the C Sharp expression, so we get 10 out of 10. But what if we really did want to take the rating and divide it by 10? Then we would need what's known as an explicit code expression where you use parenthesis and the at sign to explicitly denote what your C Sharp code is. And if I'll save this version of the view, all the ratings turn into 1 because we're taking 10 and dividing it by 10. But for the most part, Razor is really good at figuring out what is C Sharp code and what is not C Sharp code. If I turn this back into an implicit expression, let me just show you a couple other edge cases. Imagine I wanted to prefix the rating with an R. So if I put the letter R here and we want to have an R10 or an R3 and I save this and refresh, what we see is at item.Rating is now output as literal text. That's because Razor makes a special case for e-mail addresses and when it sees something at something dot something, it assumes you might want to display an e-mail address so it doesn't evaluate this as C Sharp code. The nice thing about this is if you actually have an e-mail address and you put it there something like scottallen@Pluralsight.com, you don't need to worry about escaping anything. But in this scenario where I really do want to say something like R1 or R10, I need to go back to using an explicit code expression and surrounding my C Sharp there with parenthesis. Now that will be treated as C Sharp code and I get the R10 out. So Razor is smart enough to figure out e-mail addresses because of the presence of the at sign. Well what happens if I want to put out something that has an at sign in it like a Twitter handle? Now this Razor handle this. Well, it thinks at OdeToCode should be a C Sharp expression. It goes off looking for something called OdeToCode but it doesn't find one. This would a case where I just need to escape the at sign by using a double at sign. And now if I save the view, now we're outputting at OdeToCode literally which is what we intended to do for a Twitter handle.

Code Blocks

In addition to code expressions, both implicit and explicit, Razor also supports code blocks. We can see an example of this here at the top of the view. You can see a block of code that starts with the at sign and has an opening curly brace and a closing curly brace. In between those curly braces, we can place as much C Sharp code as we want inside of a view. But remember, keep the view simple. So a lot of code is usually a bad sign. The C Sharp code that is inside of this block is going to execute. It's not going to have its expressions evaluated and written to the client. It simply executes or produces side effects. We can set things on the ViewBag like you can see us setting the ViewBag.Title property. We'll see the significance of that line of code in just a little bit when we talk about layout views with Razor. I can even declare variables and use them inside of the code block so I could say let's create a variable called firstReview that is equal to Model.First. So it literally gave me the first item in the collection that is my model. And inside of the code block, you have to observe the correct usage of semicolons for C Sharp codes. We'll terminate that with a semicolon. Now we can use firstReview anywhere else inside of this view. So at the top of the view, I could now say firstReview.Name, we'll up the name of the restaurant. And you can see we get the name of the restaurant, the first restaurant and all of the reviews here at the top. Another example of a code block is the foreach statement here in the middle of the view. This foreach statement allows us to iterate through the model pulling out one item at a time. And you can see that Razor allows us to transition between C Sharp code before each statement and back into HTML without making any explicit marks or using any sort of delimiters to tell it when we're transitioning from C Sharp code to HTML. We simply have the opening curly brace and then we have a TR to start the table row. It works in the other direction too. So at the bottom of the foreach, we have a closing curly brace. Razor is smart enough to figure out that the slash TR at the bottom is HTML but that closing curly brace is part of my C Sharp code that terminates the foreach statement. Let me demonstrate a few other features of code blocks by removing this table and we'll rewrite this a different way. I'll remove the table. I'm also going to remove the bit about a firstReview to clean up this view a little bit. And now here above the create link, let's write a new each statement. So for each review that we have in the model, let's-- instead of writing out a table, let's write out a div for each review. And in the div we'll have an h4 that has the name of the restaurant. We'll include a span that has the rating associated with that restaurant. And then perhaps just a paragraph tag that has the city-- And the country. And then, for one last touch, let me go in to the div and give it a class of review and I'll come in to our style sheet for the Site.css. And down here at the bottom, I'm just going to paste in some code that will pretty up our review. Give it a bottom border, add some color to the h4 tag that we've added just to make it appear a little bit better. I'll save the style sheet. If I'll go save everything and build and let's refresh, we're still displaying our reviews now, we're just not using a table. We're using a series of divs. And now I'd like to show you a couple things about code blocks. For the most part, when I wrote that code block, I didn't have to worry about when I was writing C Sharp code and when I was writing HTML. I know I have to use an at sign when I want to write a C Sharp expression and I just do that. But you'll notice that a comma became part of my literal text that's outputted in the view. Again, no explicit delimiters there. But again, there's a couple special edge cases here. For instance, what if I wanted the text review to appear before this div and not inside of its own HTML element? So if I just put review here and save the view and refresh, Razor gets a little bit unhappy and essentially that's because it didn't make the transition from C Sharp code back into literal text. It thinks that review is part of my C Sharp code. The easiest way to fix this, if you really don't want review inside of some sort of HTML element, is just to use at colon that tells Razor that this is literal text. And now if I refresh, we'll have the word review in front of every div. The same thing could be used if we, for some obscure reason, wanted to have a closing curly brace underneath of every div and have that actually display in the view. Fairly, if I refresh that doesn't display. Razor thinks this is part of C Sharp code and it thinks this second closing curly brace is literal text so that appears at the bottom of the view. I wanted this curly brace to be part of my text. Again just use at colon and now we have the closing curly brace appearing where we wanted. But those are edge cases you probably won't run into this very often, for the most part, just write what you feel in Razor, it will figure out the proper thing to do.

Layout Views

Next, let's turn our attention to what was not in our index view or restaurant reviews. What was not in our index view was the navigational menu that you see in the application, at home, about and contact links or the Ode to Food logo or the copyright at the bottom of the page. All the code and mark up in our index view was there only to output the review information into the middle of the page. As I alluded to in the introduction, there is another type of view at work here to handle these common things that appear everywhere. Every application has some common UI structure. You need navigation links and headers on every page and maybe a footer. To provide that structure in Razor, we use a layout view. If you're familiar with ASP.NET web forms, then you have probably used master pages. Razor layout views are similar to master pages but really, they're much, much better and easier to understand. In this layout views, that's typically where you have your doc type, your head tag and all the common markup that you need. And then a layout view has two special methods that I can call render body and render section. These methods allow the content views like the index view that we wrote to plug in their pieces of the UI into specific locations of the layout. Let's look at an example. Inside of the index view that we've been working on, let me remove some of the experiments that we've done with code blocks and save the view and refresh. And just point out again that the only thing we're really rendering here is the middle section, the latest reviews. All the other pieces like the logo and the menu, those are handled by the layout view. You can find the layout view if you go into the Solution Explorer, go into view Shared folder and here is underscore layout.csHTML. There's nothing special about that underscore. Many developers use underscore just as a convention to identify views that are not primary content views like the index view that we wrote. It's not a view you would typically return as a view result from a controller. But the layout view itself is really nothing special. For the most part, it's just another Razor view. It has literal text that can have C Sharp code expressions, it can have code blocks. It's where you typically have your doc type. It's where you want to have your head tag and you typically have a definition for the body element. And any changes that I make here will be reflected across the entire application because in this application, this is the only layout view that we're using. You can have more than one but we may now have only one. I can demonstrate that if I collapse this header section and then let me just use Control K, Control C to comment that out. I'll save the view and refresh. And you can see not only did our restaurant review is changed but also if I was able to navigate to the homepage, it's also missing that section now too. To undo commenting that out and scroll down here a little bit, and here is where you can see the call to a RenderBody. When a layout view calls RenderBody, that is on the content view, our Index.csHTML that is one it will render and output it to HTML and that HTML will be inserted right here where RenderBody is. One obvious question at this point might be, how does the MVC runtime know to use this layout view? We're not saying anything inside of the layout view that would be particular to telling everyone that this is the layout view for the application. We're certainly not doing anything inside of Index.csHTML that indicates that that's the layout view we're using. And the answer to this question actually comes from a small magical file called underscore ViewStart.csHTML. You'll find it in the root of your Views folder. When I open that file up you can see there's a code block inside of here that sets a layout property equal to tilde slash Views slash Shared slash underscore Layout.csHTML. So the secret here is that underscore ViewStart.csHTML has the ability to execute this code before my view starts rendering and it sets this property. And this is just a convention with the Razor view engine. If you have a file called underscore ViewStart.csHTML, anything you put in that code block at the top will be able to execute before the view does. And this works in a hierarchy. So if ViewStart.csHTML is in the root of the Views folder, it applies to all the views that are inside of here. But if I make a copy of this file and put it inside of Reviews, let's open up this copy underscore ViewStart.csHTML and change this over to Layout2. Now I'll save everything and refresh the home page. We have our header section back and this works normally. But if I go to reviews, we'll get a runtime error because we've specified that the Views and the Reviews folder should use a different layout file, underscore Layout.csHTML and that doesn't exist. So in order for this to work I don't have to have underscore Layout.csHTML in the shared folder. So let me delete this file. First, let me make a copy of this. And also I'll show you that you can overwrite this on a per view basis. So if I open up our index view that's showing the latest reviews, I can also specify the layout file here and let's just change it to Layout3.csHTML so we can see a difference. Save everything and refresh. Now it's looking for Layout3. I can also turn off a layout view here if I wanted to. I could just say layout. I don't want to use a layout page. I'll set that equal to null. And now it's working but we're also not using a layout page so we do not have any style sheets applied and the view looks very plain and simple. Let's go back to using the default layout view. Now inside of a layout view, in addition to RenderBody which is a required call somewhere inside of a layout view you must call RenderBody. You can also call another method called RenderSection. This is optional. You can have one or more sections. And what RenderSection will do is provide a content view like Index.csHTML, the ability to plug in content into other sections of the page. So this RenderSection is rendering a section called featured. There's also another RenderSection down here at the bottom of the layout view that will allow a content page to inject scripts at the bottom of the page. We'll come back to that later when we talk about Ajax. But right now let's talk about this RenderSection which is currently not a required section that means a content view can have this section that cannot. If I switch this over to be a required section, what will happen when we render this Reviews view is that we get a runtime error saying you did not have this section called featured. I want to change this back to being an optional section but we're also going to implement that section inside of Index.csHTML. Let's just take this bit of markup that we already have and put it inside of a section. We can define these sections anywhere. You just have to give them the name. That name has to match what you specified in RenderSection. And let's put that we are showing the latest X number of reviews inside of the featured section and refresh. And you can see that appears at the top of the view. But inside of layout, if for some reason I wanted to show a featured content at the bottom of the page, let's say in the footer, I could also work. So let me move that section back up to the top. And now that I'm looking at it, what would be really nice is that when I'm on the home page of the application, I'd have a link here to go to that ReviewsController instead of typing reviews into the URL. That's easy enough. This is the navigational menu that the layout page specifies. Let me make a copy of that last action link and create a new action link. We've looked at this before. This is the text I want to display, Reviews. This is the action that I want to reach from some controller, the Index action. And this is the name of the controller that I want to reach, the ReviewsController. And now if I save everything and refresh, we'll have a Reviews link that will appear throughout the application. I can click on it and we end up on the ReviewsController.

HTML Helpers

The action link helper that we just used is one of many HTML helpers available inside of the view. The purpose of an HTML helper is to make it easy to create small blocks of HTML. There are helpers to create links like action link but there's also helpers to create inputs, validation messages, labels and more. Many of these helpers are quite intelligent. For example, in these codes on the screen, we're calling HTML.EditorFor and passing in an expression that references a property on the model, the FirstName property. EditorFor can walk up to that property and examine its type, determined that the FirstName property is a string and therefore what it needs to do is emit an input type equals text. If instead we were using EditorFor against the Boolean property, then EditorFor could produce a checkbox input. There's actually a number of other things that EditorFor will do. We'll explore this as we move throughout the rest of the course. But all of these methods like EditorFor, LabelFor, ValidationSummary, they're available from an HTML property that a view inherits. That's another reason we call them HTML helpers and it's also the reason we access them using HTML dot. Let's use a few of these in our application. Inside of our Index view, lets' give ourselves the ability to edit any particular restaurant review because the edit scenario is going to involve a lot more HTML helpers than what we've seen so far. So let me add a span inside of a div for each review, a span that's going to be aligned to the right. And inside of here we will use HTML.ActionLink to display a link that displays the text edit and we'll go to the Edit action on this current controller. I also need to pass along some information so that the Edit action knows which review I'm trying to edit and I can do that by passing along the ID of the review. So I just need to say item.Id. The ID property then is wrapped up in an anonymous object. And you can put as many additional values as you want inside that anonymous object, that get passed along to the routing engine and essentially say here's some additional information that needs to go in the URL. The routing engine is going to look at ID and say, oh, I can see where I can put ID actually into the URL, so we should come out with something like slash reviews, slash edit, slash three to edit the review with an ID of three. And let's test it out, I'll save the view, refresh the browser. There's our Edit link. It looks correct. Slash review, slash edit, slash three. And when I click on that, we already have an edit controller action because when we scaffold it out this controller, we said that we wanted a controller with the ability to update reviews. But what we don't have yet is a view itself. But that's okay because we also need to add a little bit more logic here into the edit action. This is the edit action that responds to that Get request. The ID that's in the URL will automatically be extracted by the runtime and passed into our action. All I need to do is go out to our data source and find that review that we want to edit. So I can say that the review, it goes reviews give me the single review. Or I'm given a review R dot ID equals that incoming ID. So just a little link query to say give me single object out of this collection that matches this criteria, its ID property has to match this incoming ID parameter and that is the model for our view. Now we actually need an edit views but we don't have this error screen over here. I can right click inside the edit action say please add a view for me. We do want it to be called the edit view, we do want to use Razor. It's going to be strongly-typed against the RestaurantReview. All of this is correct. But I do want to scaffold out this time an edit view. Now when I click Add, I'll have a brand new edit view. And what I should be able to do now is just do a quick build and refresh slash review slash edit slash three. And now I have a form that will let me edit all these values. So let's take a step back and see what some of these HTML helpers are doing. And one easy way to do that is to go in and view the source for this and compare it to what is inside of our view. Let me scroll down a bit. And now we can start to match some things up. So HTML.BeginForm writes out an opening form tag. Without passing any additional parameters into begin form, what it will do is emit that form tag, it will set the action of that form tag to go to the same URL that we came from slash Review slash Edit slash 3 but to do a post back when the user clicks on the button to submit this form and save data. Begin form is useful because you can pass in additional parameters to change that URL, change the method. It will take care of emitting the correct HTML. A little further down we have a HTML.HiddenFor. What this helper will do is it will emit an input type equals hidden into the form so we're storing the ID value in the form. We're just not giving the user the ability to edit that value, it's a hidden input. And down a little bit further we have HTML.LabelFor. Labels are great for accessibility. And this is simply building a label attribute where the for attribute is equal to the name of the property. So the name is name-- or the name of the city, in that case will build label for city. And then we have editor for HTML.EditorFor is literally saying I want an editor for this property. Now you go out and figure out what the best type of editor for. Maybe that editor will be an input type equals text or an input type equals number or a checkbox or radio button. In fact if we scroll down a little bit further, we can see that the input for a rating is type equals number. That's a new input field for HTML 5 on browsers that are up-to-date with the standards, they'll know that the user is supposed to input a number here not just some random string like A B C. There's also a lot of data dash attributes in here like data dash val, data dash val dash number. Those are providing data for client side validation. We'll talk about that later in this course. But for now, I just want to point out that this is what HTML helpers are all about. We could certainly for the rating come in and all by ourselves say that we just want an input type equals number and set the value to at Model.Rating but EditorFor will do that for us. And also populate the ID and the name attributes so that the editor of the input matches that property name of our model. And if you remember back in our discussion in the last module, we talked about the way that the MVC has a way of matching up things that we need by name. So if we see that we need an ID parameter as a parameter to our action method, it will go out and look for something called ID and it can look in the routing data, it can look in the query string. It can also look in these posted form values. So if I need a rating for my model something that the user has typed into the screen and then click save, the MVC framework can automatically find that rating and move it into a property for me just based on a naming convention. We'll see that here in just a second. But let me close the source code view and implement what should happen when the user clicks on the save button. In that case that's going to be an HTTP post message. This is also a topic we talked about in the last module. That will end up not at this edit action because this responds to a get request. It will end up at this edit action that explicitly advertises itself as basically only replying to a post. I'm going to change around the code that was given to us automatically just a little bit. And the idea here is that I want to take values that the users typed in. I want them-- and I want to move them into a review so I can save that. First, let's grab the review that we need to edit. And then to move the values in, I'm going to call a method called TryUpdateModel. What TryUpdateModel will do is go through a process known as model binding. In fact, model binding happens anytime you even have a parameter in an action method. It's what ASP.NET MVC does when it goes out and it looks around in the request try to find things to move into an object for you. So when I have a parameter called ID on the edit action, the model binder in ASP.NET MVC will find that ID, move it into that for me. When I say TryUpdateModel on review, the model binder will go out and look at review, see that it has a rating property, and then go out and try to find something called rating. But fortunately, there should be a posted form input named rating. The MVC runtime will find that and just move it into my review. If anything fails, if any validation errors occur, TryUpdateModel will return false and I don't want to save that review. We'll talk more about validation later. And if that happens, I can return the view with that same review to try to let the user fix whatever problem they have. But if TryUpdateModel works, this is the point where I would save that into the database. We're not working with the database yet. And for right now, what I will do is return a redirect action. Let's go back to the index action. I want to redirect here instead of just letting the users sit on the saved values on that posted form field. It's very common that after an HTTP post where the user has saved something you redirect them back to a page where they can view the changed results. That way, they don't hit refresh on the result of this post operation and accidentally submit something twice. But with this logic in place, let's try to do a build and let me come out and do a get on this page slash review slash edit slash 3 so we can start everything over. So you can come up again with the house of Elliot let's bump the rating down to a 9 and click save and you can see we made that change. Let's edit this again change this back to a 10, perhaps using alternate spelling for Gent and click save, those change is also reflected here. And one of the reasons this was so easy to build is because we're using TryUpdateModel which just relies on the names of properties to match up data with what needs to get pushed into the model. And all of that matches up because our edit view is using things like HTML.EditorFor which will automatically build the correct inputs with the right names and the IDs and the validation and everything else associated with it. There are many other HTML helpers that are available than what you see here, you can see them in the IntelliSense window. There's ways to build check boxes. There's ways to display things, there's ways to generate names, route links and validation messages. You can also build your own custom HTML helpers. And I would suggest that if you find yourself typing a lot of C sharp code into a Razor view to build an HTML element, I would suggest you go out and look at how to build a custom helper. They're very easy to build and then they're very easy to use and a custom helper can really help to get logic out of your view and keep the view as simple as possible. That's really the ultimate goal of the HTML helpers, keeps views simple. ( Pause )

Partial Views

Another type of view that you'll find useful when building MVC applications is the partial view. A partial view allows you to allow you to put HTML on C sharp code into a file that you can reuse across multiple other views. We'll see some used cases for partial views in this course but for now, let's just imagine that the div we have here to display a review is something that we need in several different places in the application. We show the reviews in a couple different places and we'd like the display to be consistent but we do not want to duplicate this code every time we display a review. We want to reuse it that's why I'm going to select the code and cut it out of this view. Then I'll come over to our views folder, the reviews folder in the views folder and say add a new view and I'll call this underscore review. Again the underscore is just a convention that means this is a special type of view in this case it's a partial view. It's still a Razor view. It can still be strongly typed against the restaurant review but this time I'll leave the Scaffold template empty because I already have the mark up that I want to put in here. And I will select the check box that says create as a partial view. What that really means is that when Visual Studio creates this, it won't automatically add a code block at the top of the view to set the page title because we don't want to set the page title from a partial view that's the responsibility of the content view. This is just rendering a portion of the page. And now I can click add and I will paste in the code that I cut out from the index view and hit control K, control D to do some formatting. In this case, it just bumps it up against the left margin and aligns some things. But in the mean time you have some HTML or C sharp code that doesn't quite fit right, it's not formatted properly. If you hit control K, control D that will do some formatting for you. There is one other change I have to make in this view. It used to be inside of a foreach loop that had a local variable called item. Now, the model for this view is actually going to be a single restaurant review. So I need to change item to model. And one quick and easy way to do that is with quick replace. We'll just change item to model, replace all and now our view should be ready to go. And you can see that the partial view is just another Razor view. It has the same csHTML extension that can still be strongly typed. It still has a model that it can work against. It's just the intention is that we're going to reuse this from other views in the way you render a partial view, is to use another HTML helper HTML.Partial. When you call HTML partial, you give it the name or the view that you want to render in this case, Review.csHTML. And you don't need to specify the file extension when you ask the MVC runtime to render a partial but I can pass in the model that it needs to render. The model in this case is each individual review each item. And with all this in place, I can just save everything and refresh the application. The user won't see any visual difference in the page. It looks exactly the same as it did before. We've just made a structural change in how the page displays. We've created this partial view and we might have done that because we want to reuse that across several other views. Another good reason to use a partial view is just to simplify a view. Now the foreach statement is a lot easier to read because I can clearly see when it starts and when it ends and I can see that all we really do in here is render each review using a partial view. I also want to point out that since we put underscore review in the reviews folder, it's only available from other views that are in the same folder. If instead, we had placed this view in the shared folder, it would be available anywhere in the application from any view to say render partial this review. I also want to show you another scenario where partial views can be useful. So one way to render a partial is to use this HTML.PartialHelper. Typically, this is the approach you take when you want to reuse a piece of the HTML or you want to simplify a view because you have a really complex model. But regardless of why you're using HTML.Partial, when you do use it the only data or model information that you can pass to HTML.Partial is model information that you already have available inside of your view. So you can pass the model item or you can pass a subset of the model item like here, we just passed a single review. But you can also come across a situation where you want to render something that is not part of your model. This frequently comes up with the layout view. Because the layout view is executing for every page in the application, and so it's very difficult to tie it to any particular model, there's ways to do that but it involves a lot of messy inheritance. So the scenario I'm talking about is imagine in the layout view if we want the best review in the database to always appear somewhere on the page. If I went to do that there's no way for me to go into the layout view and specify a model directive here that says this view will always have a restaurant review as a model because this layout view is used again throughout the entire application. It's used on pages that display restaurant reviews but it will also be used on pages that display restaurants or allow a user to register and log in. And all of those other scenarios, they do not involve restaurant reviews. So I cannot strongly type this layout view to a particular model. But there is another helper available that can allow me to solve that situation. And it involves partial views quite frequently. This other helper is called HTML.Action. So imagine we do want to show the best review on every page in the application. We'll show it down here just above the footer. What I can do is invoke HTML.Action and what HTML.Action allows you to do is set up a sub request inside of this primary MVC request. So it's a subrequest that can go out and call another controller action that builds its own model and dependent of what the main controller is doing and render its own view, a partial view that will be inserted right here where the call to HTML.Action is. So let's say that we need to build an action called BestReview. And this is always going to be on the Reviews controller. Just like HTML.Action link, you can also pass in additional parameters here that will get passed along to this other controller action, but we don't need to pass any parameters. And this is basically saying go out and call this action on the reviews controller and whatever it omits, whatever outputs whatever the result is, place it here where I have HTML.Action. And I do want to point out that this is not a separate HTTP request. This doesn't require the browser to come back to the server and call this other controller action into controller action that's called from server code it's like a subrequest. So let's go over to the ReviewsController we need to implement BestReview and I'll just put it as the top action here. This can still be an action that returns an action result. Actually it's going to be returning a PartialViewResult and it's called BestReview. We'll need to find the best review and I realized that all of our reviews right now have a rating of 10, so we'd need to find a way to reconcile a tie or let's say from R in reviews. Now we orderby R dot rating in descending order. Now, we have our reviews in sorted order and now we're going to return not a view this time, but a PartialView. And we could create a brand new PartialView for this action to render. It could be called BestReview. We could also render that other PartialView that we just created, the one that is called underscore Review.csHTML. Again, no extension needed here and we will pass in the first review that we picked up based on putting the reviews in the sorted order. This is by the way how you would overwrite what view and action is going to render, you can always explicitly specify a view by name here without the file extension. But I can do that for the IndexView too. If I didn't want it to return the IndexView for some reason I could always specify a view name by string here. But now I should be able to do a bill and let's come out to the home page of the application. And what we should see here at the bottom of the home page is there's our best review. We should probably put a better title around it so we know why that information is appearing here. But that should now be on every page of the application, because it's in the lay out view. And we did that by calling an action that got to build its own model, render its own view so it's completely independent from everything else, it didn't really complicate our layout view to do those. But one thing to keep in mind when you do build an action like this BestReview, as I said before, your public methods are available to the browser. So if I come in to the browser and I go to slash Reviews slash BestReview I will be able to get that result. Because like all public methods on the controller, if the user enters the right URL into the browser, they're going to be able to invoke the controller action. But in this case, if you do not want that to happen, there's an attribute that you can apply to this action. It's called ChildActionOnly. And if I do that and do a build it's no longer legal to call that directly using an HTTP request. You can only access this by calling HTML.Action as a child request, so it would still work in the layout view. I just can't call it directly. And now you know a couple of different ways to use partial views. There's HTML.Partial which allows you to simplify a view and reuse HTML. And there's also HTML.Action which allows you to set up a completely independent subrequest that builds its own model and renders its own PartialView. ( Pause )

Summary

In this module, we covered a number of different topics related to building views in an MVC application. We look at the Razor syntax, and learned about implicit and explicit code expressions as well as code blocks. We talked about HTML Helpers and I showed you how helpers like EditorFor will help you build form inputs. But I haven't told you the whole story there yet. We'll see more when we start validating incoming data. We did look at layout views and how they can define the UI structure for our application and I demonstrated how Razor does HTML encoding for us to help avoid cross-site scripting attacks. Finally, I showed you a couple of scenarios where PartialViews can be your friend. You can use partials to render a portion of your model in a reasonable way and you can also use partials and child requests with HTML.Action to delegate work to another controller.

Working With Data (Part I)

Introduction

Hi! This is Scott Allen and this module of the course is devoted to Data. In this first Data module, we want to start using a real database with our application. So the focus of the first module is to bring SQL Server into the picture. In the next Data module, we'll look at other data-related features in the MVC framework, features like model validation. What I'll demonstrate in this first module is how to use a technology known as the Entity Framework and combine the Entity Framework with the plain simple C Sharp class definitions we've been working with to build an application that uses SQL Server to persist data. We will see how to query data using link in the Entity Framework, and also use the Entity Framework Migrations to manage our database schema from C Sharp and Visual Studio.

The Entity Framework

The first stop we'll take in this module is to get up and running with the Entity Framework. The Entity Framework is a part of the .NET framework and you'll already have a reference to the Entity Framework in the new MVC internet application. The Entity Framework or EF as I sometimes call it for short, allows me to access a relational database using strongly typed C Sharp code, or Visual Basic code, or really any .NET language. When I'm working with classes to talk to the Entity Framework, I don't have to worry about SQL connections, SQL commands, SQL parameters, SQL data readers, none of the low level abstractions you might have programmed with in the past with .NET. And if you don't know them, that's good for you because you won't have to learn them. With the Entity Framework, I work with my C Sharp code and I can issue queries to the database using a language feature called Language Integrated Query also known as LINQ and this is a feature that's in both the C Sharp and Visual Basic languages. There's a few different ways to get started with the Entity Framework. One approach is called the Schema First approach. In this approach, you open up a graphical designer in Visual Studio, you point it to an existing database, and it can import the database schema and generate all the classes you need to query and update that database. Another approach is a Model First approach, this is where I use the same graphical designer in Visual Studio to draw a conceptual model for my applications, so what classes do I want. And then I have the Entity Framework generate both my class definitions and my database schema. Finally, there's a Code First approach you can take with EF. In this approach, I just write C Sharp classes and the Entity Framework and use those class definitions to create a database for me. And it will do that either using conventions like naming conventions, very similar to the naming conventions we've seen with the MVC framework and that is where things just magically work if they had the right name. I can also provide explicit mappings or change the mappings if I don't like any of these conventions or defaults that the Entity Framework uses. In this module, we're going to use a Code First approach, so we won't see a visual design surface or any XML configuration files. I hope that makes you happy. And when I say that the Entity Framework supports relational databases, then the Entity Framework supports quite a few. You can point the Entity Framework towards a full-blown SQL Server installation on a remote server that has hundreds of gigabytes of RAM and network storage. Or you can point it to SQL Server Compact edition which is running in process and accessing a database file on the local file system. You can also use it with usher in the clown (phonetic), you can use it with Oracle, you can use it with DB2. But my code really doesn't have to know what's on the back end. The Entity Framework knows how to work with all these different databases. I can build an application without worrying so much about the database details. In this module, we'll be working with SQL Server, specifically the local DB version of SQL Server.

Building Entities

So far in the application, we've been using some in-memory data and this Restaurant Review class to get some things done and experiment with razor and controllers. But this restaurant review really combines a restaurant, the restaurant name and a review which is the rating, I really need to separate those out and form a relationship if we're going to move forward. So there should be a restaurant class which I can create right now. And that restaurant class will have multiple reviews associated with it. We'll give a couple of properties to the restaurant, there's going to be a primary key fields so we'll give it an ID property. We'll give it a name. We'll also store the city that the restaurant is in and the country. And then every restaurant can have some associated reviews. We'll put that in ICollection, a collection of restaurant reviews. And this seems like a good definition for a restaurant. Let's go over into the review class and change some things around. A restaurant review will still have an ID, the name, city and country, that's really restaurant information. We don't need that. We'll have the rating. Perhaps we want the body of the review, so if a person wants to type out some text, what they really thought about the restaurant, they can put that in here. And although it's not entirely necessary, it does make some scenarios easier if I add a restaurant ID field. And this restaurant ID will point back to the restaurant that this review is associated with. And now restaurant review and restaurant are my two entities, these are going to be objects that I expect to instantiate and save in a database and retrieve from a database. But right now, I'm leaving their definition in the models folder. But if you continue down this path on larger applications, the model's folder becomes way too big. There's too many classes inside of it. You might consider having a separate folder or even a separate project where you define entities like this. But this is good enough for our purposes to get started with. Now let me also add a class that will use the Entity Framework to persist and retrieve this data. We'll call it OdeToFoodDb. OdeToFoodDb needs to derive from an Entity Framework class known as the DbContext. To use DbContext, I'll need to bring the System.Data.Entity name space and the scope, so I'll add that using. And then a DbContext, you can have properties of type DbSet that represent the entities that you want to query and persist. So we'll have a DbSet of restaurant called Restaurants. And walking up to that DbSet and asking for all the restaurants would essentially do a select star from a restaurant table in SQL Server. We'll have Restaurants and we'll have a DbSet of RestaurantReview. RestaurantReviews is not something that I absolutely must have, I can always get to reviews through a restaurant but I'm going to add it as a DbSet here. Again, it makes some things easier. To see how easy this is to use, let's say that we want the home page of the application now to display a list of restaurants that we have in the database. So let me swing over into the home controller and we're going to change around the index action and also the view for this controller. We don't really want to show the controller and the action and the ID anymore, instead, we want to show restaurants. So here I can instantiate OdeToFoodDb and I'll also point out the sense this is a disposable resource, I should override dispose. And if this is not null, tell it to dispose itself. To dispose API and .NET is just a way to clean up resources that might be open. Anything that implements the IDisposable interface or has a dispose method, you should call it as soon as possible to make sure everything is cleaned up as early as possible. But with this bit of code in place, let's come into the home index action and say that our model is equal to db.Restaurants.ToList. What the Entity Framework will do when I execute that line of code is go into SQL Server, find where it stored all the restaurants, retrieve all of them and put them into a list. Eventually, if this application grows, we will need to provide some search facilities, we'll need to provide some paging facilities but this should be enough to build a screen that has a list of all our restaurants. And now let's swing over into the index action for the home controller. And let's clean some things up. Let's get rid of the feature section and also the information about what we suggest because we really just want to show restaurants on this page. And because this view is already created, I'll add a model directive here manually to say that this is strongly typed against IEnumerable of restaurant. And I need to fully name space qualify that as OdeToFood.Models.Restaurant. Again, just control period will give you that drop down menu and you can have Visual Studio add that for you. And now let's write those restaurants out. So for each item in my model or each item as a restaurant, let's write out a div. It has an h4 inside with the restaurant name. And we can also write out the city and the country. And then perhaps just put in a little horizontal line to distinguish one restaurant from the other. Now at this point, we don't have any data in a database because we don't even have a database but we're very close to doing that. Just by having a DbContext derived class, if I instantiate this class and ask it for data, it will go out and see if a database exist. And if that database doesn't exist, it will be created for me. So I should be able to do a build. And, of course, we'll have some compiler areas because we already had some code inside of the reviews controller that was depending on a different definition of restaurant review. So for right now, I'm going to hit Control A to highlight all the code inside of our reviews controller and then hit Control K, control C to just comment it out. We'll come back to the reviews controller later and rebuild it so that it actually displays reviews that are associated with our restaurant instead of using this in-memory data that we had here before. There's another change that I want to make because that's commented out. We're also going to have to change the layout view because if you remember, caught into the reviews controller to list the best review but we don't really need that. I'm going to delete that bit of code from the layout view. And since the definition of our restaurant review has changed so dramatically, I'm going to go ahead into the Reviews' views folder and just delete our existing views from inside of here. And this is all okay. We're just taking in an iterative approach here to building an application and learning MVC at the same time. But I should have done enough now to be able to do a build and run the application. And on the home page, it will try to list our restaurants but of course we have no restaurants. But the question here is, we didn't get an error so the Entity Framework talked to some database, where is that database? And to answer that question, let's go to view Database Explorer and add a connection. This is going to be a SQL Server connection. And I'm going to specify LocalDB v11.0 as the server I want to connect to. This is a special connection string for SQL Server 2012 Express, the LocalDB edition. So if you haven't heard of SQL Server LocalDB, it's a new edition of SQL Server that's primarily targeted towards developers. The big difference between this and SQL Server Express which you might have used in the past during development is that LocalDB runs with your identity and it doesn't run as a service. It's launched on-demand when you need to get to it. Other than that, it's very much like SQL Server. And now let's pull down this drop down list to see what databases are available there. And I can see that one of the databases is OdeToFood.Models.OdeToFoodDb. I know this is the database that the Entity Framework just created because the name of that database exactly matches the name of the OdeToFoodDb class that we created. So by default, when you instantiate one of these and try to query data, if the Entity Framework doesn't find a database and you have no other configuration in place that tells the Entity Framework where to go, it will just look at LocalDB and create a database that has the same name as your DBContext class. And if we come in here, indeed we can see Restaurants. We can see the RestaurantReviews. We can see that RestaurantReviews have a primary key. That's the ID column. We can see that Restaurants also have an ID column. And just to prove that this is working, if I go in and I say, show table data, what I should be able to do here is add some new restaurant into the database manually. Save that in the database. And if I come up to the home page of the application again and refresh, that restaurant is displaying on the home page. So we are querying SQL Server and we've created our first database. All this was taken care of by the Entity Framework just using the definitions of our C Sharp classes, Restaurant and RestaurantReview.

Database Migrations

You might be wondering if it's possible to influence where the Entity Framework creates database and also if you can influence how the Entity Framework creates a database. Can you change the names of tables and add indexes and things like that? Let's look at the first question, the where question. You can configure your data context with a specific database connection string. The way you do this is to call into the base class constructor. And the syntax for that in C Sharp is to use the keyword base. And right here, I could pass a connection string. Explicitly specify the server, the database, and the credentials. But we generally don't hard code connection strings into an application because when we want to point the application to a different database that means we have to change the code and recompile. And since we usually write applications that used different databases, at least, there's one database during development and one for deployment, and perhaps one for testing. Well, it's a good idea not to hard code connection stings. Instead, with ASP.NET, we can store connection strings in our web.config file. I'm going to open up the web.config file that's in the root of this application. And we can see there already is a connection string here in place for us, it's called default connection. The project template put this in place when we created this new application. And we can change this connection string to point anywhere. I'm just going to tweak it a little bit. I still want it to point to LocalDB but I want the initial catalog not to have the word ASP.NET and all this other stuff in it. I just want it to be OdeToFoodDb Integrated Security equals SSPI. That basically means connect with the process credentials, which is fine. We do want it to attach to a file that's in the DataDirectory. And again, I'm going to call this OdeToFoodDb.mdf so it's very easy to identify there. And now, if I want the Entity Framework to use this connection string, what I do when I call into the base class constructor is say, name equals, and the name of my connection string. And that tells it to go out into the web.config and use whatever that connection string value is when it connects to the database. Now, I never need to recompile the application when I want to point somewhere else, I just need to go into the config file and I should be able to run the application again. And we should be back to a blank page, there it is, because we're now pointing to a new database. It doesn't have that restaurant in there that I added manually. But I should still be able to use the Database Explorer to poke around in that database. In fact, default connection is already here in my data connections. It just detected that when I loaded this project. And there's Restaurants and RestaurantReviews. I could add data in there manually again. And also, I should be able to see this database now if I come into the Solution Explorer and look in the App Data folder. What I'll have to do is allow Solution Explorer to show all files with this toolbar button. And there I can see OdeToFoodDb.mdf. There are some other databases here that I should clean out but we'll worry about that later. Let's go back to our second question at the beginning which is, how can I influence how the Entity Framework creates my schema? And can I have it initially populates some data for me into that database? And the answer to both of these questions is yes. All we need to do is use Entity Framework Migrations. Migrations are a feature of the Entity Framework that allow you to configure database schemas with C Sharp code, seed your databases with C Sharp code and Migrations can then track changes that you're making in your entity classes. Those classes like Restaurant and RestaurantReview. And it can keep the database schema in sync with the changes that you make in your C Sharp code. The way to get started with Migrations is to open up the Package Manager Console. There's two ways to do this. You can go to the View menu and go to Other Windows and find the Package Manager Console or you can use this Quick Launch toolbox that's up here in the upper right of Visual Studio and just type, Package or Package Manager Console, it'll find all the commands in the menus everywhere throughout Visual Studio really quickly and then you can just click on it. The Package Manager Console will open up here inside of Visual Studio as another window. It's essentially a PowerShell command line. If you know how to write scripts in PowerShell then you can use any PowerShell command in here that you want and list directories and manage services. Inside of here if you want to get what directory you're in, you can type GET-D and then hit tab. You can see it has some tab completion. Let's try and Get-Date instead, that executes, that's a PowerShell command. But we're here actually for Migrations. And what I want to do is enable Migration. So, enable dash with an M, it knows I want Enable Migrations. And I'm going to have to specify a ContextTypeName. This is the name of my DBContext that I want to enable Migrations for. In this case, just typing in OdeToFoodDb is sufficient. And now when I press enter, this is going to kick off some logic that is part of the Code First Migrations of the Entity Framework that goes out and pokes around the environment and looks like at my classes, looks at any existing database, and it will add a new folder with a new file into my solution. The new folder is called Migrations. And inside of Migrations, right now there's two files, Configuration.cs and basically a schema change script that's written in C Sharp. Let's talk about configuration first. The configuration class is really about controlling Code First Migrations. How do you want it to perform? When should it run? The one setting in here that is most notable is the Automatic Migrations Enabled setting. By default, it will be false. And that means, basically, that the Entity Framework won't make any changes in your database unless you explicitly tell the Entity Framework that you want it to make changes. I'm going to set this to true because quite often, when you are initially starting a project, you just want to be able to make changes in your C Sharp code wherever those classes are and just have the database be ready for you in the application to run. If you're working in a more controlled environment and the product is a little more mature, you might want to set this to false and be very explicit about when changes in the database happen. But I'll show the impact of setting this to true a little bit later. The second nice part of the configuration class is the seed method. This is basically where you can tell the Entity Framework to populate the database with some initial data. And now, every time that it goes to update the database, it's going to invoke the seed method. And this gives you the opportunity to populate those tables that always need data, things like lookup data. You need a list of countries. You need a list of postal codes, all of those types of things can go into seed, and that's usually static data. But I'm going to do something a little bit different because we do not have a screen yet that will allow us to create a restaurant or a review, I'm going to paste in some code that will create restaurants for us. I'm pasting it in so you don't have to watch my typing skills, I know it's painful. I do need to bring in some name spaces, OdeTofood.models and system.collection.generic. But now that this code is legal, let me explain a little about what it does. This is walking up to the RestaurantsDb set on our OdeToFoodDb context and it's saying add or update the following restaurants. So literally go into the database and see if any of these restaurants exist by name, it's looking them up by name. So what we'll see if there's a Sabatino's in the Restaurants table. And if there is, it will update that record with the information I have here. If that name doesn't exist in the Restaurants table, it will add this new restaurant as a row into that table. It will do that for all three of these, in fact, it's even going to add a review for this last restaurant here. And now, every time I update the database, the seed method will run. And when I'm talking about updates, I'm talking about Migration updates. You typically run a database update when you want to migrate the schema. And that's why it's important that we're not just doing an add because we might update this database multiple times. As we're making schema changes, we're doing an add or update so this information isn't duplicated inside of here. So how do you update the database? There's a couple of different ways to do it. You can configure the application to automatically apply updates, that's going to be very useful much later in the course when we deploy this and it will just create a database for us in the Cloud. We can also do it explicitly through the Package Manager Console. It's a command update-database which I'm going to run with the flag-verbose, so update database-verbose. Verbose isn't going to add much information to this update because we don't have any other changes to do. And the two important pieces of output here are number one, there were no pending code-based Migrations which is good. That means essentially nothing in our C Sharp class has changed so it didn't have to change the database. But it did run the seed method that would that put that information into the database. I'd be able to see that if I went into the Database Explorer. I should also be able to see it if I just refresh the application. And there you can see, I pulled those records out of the database. So they are always there. And every time I do the update database, it will check to make sure that they're there and update them if they already are. The other piece of output about no pending code-based Migrations, again, that's because the Entity Framework didn't detect any changes. But the other thing I can do when I run update database is have the database synchronize with my model. So for instance, if at this point, I realize that a review should really also include the name of the person that was reviewing this, I would probably come into this class and give this a new property, let's call it ReviewerName and do a Build. And at this point, if I come back to the Package Manager Console, I could do one of two things. I could tell the Entity Framework that I explicitly need a Migration script to move from the database schema as it currently exists into a new database schema that can store ReviewerName or I can just tell it to update the database and let it figure that out without being very explicit about it. I'd only be able to do that because Automatic Migrations are enabled. So what I will want to do is just to update the database. But before I do that, let me show you what a Migration script would look like. Because when we enabled Migrations in this project, the Entity Framework created both the configuration.csfile but also this, and this will create .CS file. And if we look in here, it's essentially a schema change in C Sharp code. In order to initially create the database, what we need to do is create a table called Restaurants, give it the following columns, ID, name, city, and country, and set the ID to be primary key. By the way, ID is also an identity column. That means in SQL Server, you'll have that value automatically populated when you insert a new row. We also need to create a table called RestaurantReviews with the following columns. It has a primary key. It also has foreign key that references another table. And we want to apply an index to RestaurantId because it will probably be important to query the reviews given a specific Restaurant ID so we can find all the reviews for a given Restaurant. I'll just quickly point out that there's many other commands that you can run in here. For instance, if you just need it to execute some row of SQL during a Migration, maybe to update some data, you could put an update statement in here, execute it through that SQL method. And when you update the database, when the Entity Framework applies the script, that statement will execute. So these Migrations, I keep calling them scripts even though they're not really SQL scripts, they're C Sharp code. But you can generate SQL scripts from them. The Entity Framework keeps track of which ones have been applied to a database, which new ones need to be applied, and then what order they need to be applied. It takes care of all that for you to a table that's hidden inside of database called underscore Migration History. It will be a system table in your database. But at this point, what has happened is that we have changed the definition of review. We really need to update the database to get that in sync. So let me run Update-Database-Verbose again and you'll see in the output that the Entity Framework automatically applied an alter table RestaurantReviews to add ReviewerName to the table. So that's what I could've generated as an explicit Migration that would come out in the C Sharp code style. But since I have the Automatic Migrations enabled, I just let the Entity Framework take care of that. And if I wanted to actually have a name in there for this particular review, I could always say ReviewerName equal Scott. Run update database again and it will run that seed method, that value Scott should be put into the database. And now, one more thing to know about, you might have noticed that when we added ReviewerName, it was added as in nvarchar(max) field. That may not be what you want. You might want to restrict that field to a specific lane. But I'm going to defer that discussion till the next module when we start talking about data annotations because we can change that through a data annotation. In fact, we can change nearly everything that the Entity Framework is doing to the database in some fashion either through attributes or through some explicit code. But for our application, we're just going to move ahead. We've seen just enough to know how to use Entity Framework, how to use Entity Framework Migrations. And we have an application that's up and running that's displaying restaurants that are coming from a table in SQL Server.

Using LINQ

We now have a working database in our project and we will be using LINQ statements to query the data in that database. LINQ stands for Language Integrated Query. And I want you to know about two different styles of queries that you can write with LINQ. The first style looks almost like a query you would write with structured query language against SQL Server. One difference you'll notice that might jump out immediately though is that select appears at the end of the statement, unlike in Structured Query Language where you begin the query with a select. These LINQ queries are going to be strongly typed. That is the Compiler and Visual Studio, they know about the types of objects you're querying against. So if you change something in a class, you can end up with compiler areas in your queries because the compiler will know exactly what is inside of those objects. This particular query is what we call the Comprehension Query Syntax. It starts with a from keyword to specify where the data is coming from and it introduces a range variable. In this case, the name of the range variable is r. You can use that throughout the rest of the query when you want to do filtering, grouping, joining, and projecting. There are keywords in C Sharp to let you do the filtering, grouping, joining, and projecting. Keywords like where, and orderby, and select. That's just a subset of the available keywords. In this example, we're taking our restaurants, filtering out so we only have the restaurants that have a country of USA. We're going to order them by their names and that's an ascending order by default and then select the entire Restaurant. We're not doing a projection into some smaller set of data. The job of the Entity Framework then is to take this query and translate it into an efficient query for the database, then execute that query and bring you back objects, actual restaurant objects. But you can also express this query a second way. This query is using extension method and lambda expressions. The first part of this query would produce the exact same results as the query above. We want restaurants whose country is equal to a specific value, and we want them ordered by name. But when you use this syntax, there are some additional operators that you can use. For instance, Skip and Take. They're quite commonly used when you want to do paging operations. I want to skip the first ten results because they're on the first page. I want to take the next ten results because I'm going to display the second page of results to users. There's no C Sharp keywords to express those operators so you'll always see those invoked as extension methods. There are dozens of LINQ operators available. If you want to explore the syntax and the possibilities, then you can look for 101 LINQ Samples on MSDN. These samples are available on both C Sharp and Visual Basic. You can also download a free utility called LINQPad. It comes with hundreds of samples and you can write LINQ queries in an interactive manner and try them out immediately. Also, if you're a Pluralsight subscriber, I have a few courses devoted to LINQ. Well let's try out a few different queries inside of our application. Inside the application, we're displaying a list of restaurants now on the home page but they're obviously not in alphabetical order. So let's try to fix that by changing the query around in the index action of the HomeController. I'm going to switch over to using the comprehension query syntax for a bit and say from r, where r is my range variable, it represents a restaurant because I'm going to say from r in db.restaurants. Let's OrderBy r. Name. By default, this will be ascending, you can explicitly specify that using the keyword. You could also specify descending but we'll stick with ascending for right now and just select the entire restaurant. Just by putting that change in the code, if I refresh, we should now be getting Great Lakes on top of the list which we do, so that's good. But we could do some more fancier ordering. Let me try to arrange these windows so we can see everything. And now let's say that we want to display the restaurants but perhaps we want to display the most popular restaurants at the top. And perhaps one way to measure the popularity of a restaurant is to see how many times it's been reviewed. So we'll order by r.Reviews.Count descending, so the one with the most review should be here at the top and that's going to be our restaurant in Sweden, if I refresh. And now that appears to work too. And behind the scenes, the Entity Framework is taking this query and translating it into a query for SQL Server then it executes that query and brings back a result set from SQL Server but transforms that result set into real objects, into real restaurants that I can use in C Sharp code. And you can see the query we have now is a bit of a mix. It is in the comprehension query syntax but there's a little bit of extension method in here too because we need to count the reviews. And there's no keyword in C Sharp to say count. Now maybe someone discovers that the restaurants coming to the top are restaurants that have a lot of reviews but they're all negative reviews. So what we'd really like to do is order by the average review and find out what the best restaurants are. So there I could say r.Reviews, please give me an average, compute an average. And for the average extension method to work, I need to pass in a lambda expression that tells the Entity Framework which property I want to average. So given a review, please average Review.rating, and we'll still specify this in descending order. And if I do a build and refresh, you can see now we have a relatively complicated query going on, still going to produce the same result though because we only have one restaurant with reviews in the database. And actually this query is quite mild in complexity compared to some of the LINQ queries I've worked with in the past. But let me show you one more thing. Suppose that in addition to displaying the restaurant name, city and country, we also want to display how many reviews that we have for that restaurant. There's a couple different ways to do that but I want to show you one of the most performant way and also a popular approach to this. And that is if the view needs to know all this information including the number of reviews and perhaps the model that we build for should include the number of reviews. So I'm going to do a projection and not just hand back a list of restaurants to view, it could use that list of restaurants to compute the number of reviews for each restaurant. But I'm going to do a transformation. Let's say give me a new object that includes r.Id, r.Name, r.City, r.Country, and also a new property NumberOfReviews equals r.Reviews.Count and just put my missing comma in here. This is creating a new anonymous type that has properties named ID, Name, City and Country. The C Sharp compiler will just give it those property names if you don't specify something equals. And we're also adding in this NumberOfReviews equals r.Reviews.Count. I could take this and still pass it to my View. But being an anonymously typed object, we have the problem that we don't know its name, it's anonymous. What would I put as a strongly typed model directive here? Well, what I could do instead is create a new model. I'll call it a View Model because it's dedicated to this View. So, let's create a class called RestaurantListViewModel and I'm going to give it those properties that we need. I'll just paste in that code to give it the properties that I need: ID, Name, City, Country, CountOfReviews. And now, I'll use this class when I'm building a model. Select new RestaurantListViewModel, ID equals r.Id, Name equals r.Name, City equals r.City, Country equals r.Country and CountOfReviews will be equal to r.Reviews.Count. So this is an entity that I've created, something that I'll add as a DbSet to my DbContext to save in the database. It's what I call a View Model, it's something that is going to carry the information along that the View needs to display and respond to this request. So let me take this class and we will open up the Index View for the HomeController and we'll change the model from being Restaurant to being a RestaurantListViewModel. And now one of the pieces of information that we can add here would include item.CountOfReviews. Let me save everything and do a Build and refresh our home page, and now we're getting a little more insight into each restaurant, a little more information out here. View models, by the way, are a topic that we will return to talk about again. They're quite useful because, very often, a page that you need to display requires more information that one of your entities can have. So you need to aggregate some information from a couple different places, a couple different sources, wrap it all up into one object to hand over to the view so it can do its job very efficiently. And it's very easy to build a View Model. You can do a projection with the select operator. You can use a framework like AutoMapper. AutoMapper is very good at copying data from one object into another object. But what I want to focus on is just to show you that in addition to using this pretty comprehension query syntax, it's entirely legal and we can see it works well. We could rate the same query using extension methods. So I could say model equals db.Restaurants. and let's do the OrderBy. Now I need to pass a lambda expression to specify how to OrderBy. So I'll say given a restaurant r, r goes to r.Reviews.average, I need another lambda expression here that says given a review called review, make you sure you average review.rating. And in this case, to make this descending, what I should really invoke is not OrderBy, I should invoke OrderByDescending into different extension method, different LINQ operator that's available. And now we just need to select. I could say select, given a restaurant r, r goes to and let me try to save myself some typing, I'm going to take this bit of code, new RestaurantListViewModel and try to paste it into my Select statement here and highlight this, Control K, Control U to uncomment and then close things off with a parenthesis and semicolon. Control K, Control D to do some formatting which didn't really work in this case. Sometimes it's not quite smart enough to know how you wanted something formatted. I'll just select everything, tab this over. But that is the equivalent query in the comprehension syntax. There is no reason to choose these based on performance. It's really which syntax do you like better and what operators do you need because again some operators like Take and Skip, you can only invoke them using this extension method syntax. So I could say just give me the first ten restaurants that are in the database. And if I do a Build right now and refresh, we should get the same result because we have the exact same query that we had before. And now, the one scenario we haven't looked at quite yet is filtering. Let's look at that next.

A Search Filter

Let's adds some filtering to our query with a where clause. But instead of using a where operator, in a hard coded value like I did in the slide that introduced you to LINQ, let me add a parameter because a user might come to the home page searching for a specific restaurant. So we will allow them to enter a searchTerm and I'll provide a default value of this for null, which is slightly redundant because if the MVC runtime doesn't find a searchTerm in the request, it will pass along a null for the string parameter. But there is a unit testing project in this solution that is invoking the index action and the unit test and it's not passing any parameters. So I'm just going to provide a default value here so that the compiler will be happy with that unit test. And now we can use that with a Where operator. I can say Where, given a restaurant r, r goes to and then we can check a searchTerm equals null. If searchTerm equals null, everything will be true, we want all the restaurants, we're not searching for any specific restaurant or r.Name starts with whatever searchTerm was entered. And now if I do a Build and refresh the Home application, we'll have the same result so far because we didn't enter a searchTerm but I just want to verify that it still works and it looks like it does. And now I can come up here and say searchTerm equals S, and now we just get the restaurants that starts with an S. So that looks good but of course the user might not know to enter something in the query string. How can we provide a bit of a user interface that allows them to do some filtering? That will require a modification to the index view. So lets' come in here and up above the list of cities, let's add a form. Now, I could add this form using HTML.begin form but there's no real special parameters that I need this form to have, so I'm just going to type in a form tag. I do want to change one thing about the default though, the default method. I want that to issue a get request but I don't need to specify a parameter for action. If there's no action there, it's just going to go back to the URL it came from which is perfect because we can show the index view, the user can type something in, hit the submit button and will go right back to where we came from which is the index action of the HomeController. I'll explain why I used the get method in just a second. But let's have an input type equals submit, value equals Search By Name. That will be the button that the user clicks when they want to do the search and then we'll have an input type equals search and that is where they can type in to perform the search. And, of course, we'll have to give this a name and remember ASP.NET MVC matches things up in the request by name. Because we have a searchTerm parameter on the index action, I would want the name of this to match, so the name of this should be searchTerm. So let's save everything. Do a build just to be safe and refresh the application to see what this looks like. And it's not the optimum user interface but it should be functional. If I go in and type a G and click Search By Name then you'll notice we get the right result, we get Great Lake. You can see in the query string, we have searchTerm equals G. And the reason that is in the query string is because our form method equals get. When you click on a submit button for a form that does a get request, it has to wrap up all these input values and put them in the query string. If instead we had used post and done a submit, all these values would be inside that it should be message body, you wouldn't see them in the query string. It's nice to use a get request here because by putting these things in the query string, someone can copy this and paste it into an e-mail or they can bookmark it, and they can always come back to this page and see all of the restaurants that start with a G or an S or whatever they're looking for. You do have to be careful about when you use get because you wouldn't want to use a get request for a form that is submitting information to the server to save in the database, so something that is processing a credit card or creating a restaurant or editing a review, those are all post operations. Post operations are write operations. A search is a read operation. It's perfectly safe to issue a get request or a search request.

Summary

In this module, we took our first steps with the Entity Framework. We let it take our C Sharp classes and not only save objects in the database but we also let it create the database for us. This was all done using the DbSet and DbContext classes of the Entity Framework. We also used EF Migrations to seed the database with some initial restaurants and manage the schema. Finally, I showed you a few different LINQ queries to give you an idea of what is possible when querying a database with the Entity Framework. In the next module, we're going to build on what we learned here to look at editing and inserting data along with all the features you need in those scenarios, features like validation and from the error messages.

Working With Data (Part II)

Introduction

Hi. This is Scott Allen and this is the second module in this course focusing on working with data and models in ASP.NET MVC. In this module we'll continue working with the data base and the Entity Framework and we'll build some views to edit and create data. We'll also see how to validate incoming information and uncover more details about the model binding and html helpers in ASP.NET MVC.

A Restaurant Controller

Thus far in the application we've been accessing the data base and using different queries to show and search for restaurants on the home page of the application. I think the next step would be to add the ability to create update, edit and delete restaurants but I don't think we want this functionality on the home page, so let's create a new section of the application to do this. We need a restaurant area that will let us manage restaurants and what I'll do is come into the layout view of the application and let's replace the link that's currently pointing towards reviews and instead is going to say Restaurant is going to go to the index action of the restaurant controller which means we need a restaurant controller, we don't have a restaurant controller as yet. So I'm going to right click the controllers folder, say add a new controller this will be the RestaurantController and I'm going to select a new template, I'm going to select the template MVC controller with read/write actions and views using Entity Framework. Essentially, the visual studio can create your controller and all the views that you need if you tell-- first of all the model class which in this case will be restaurant, I can just type the first few words to this and find it in the drop down list and the data context class which in our case is going to be OdeToFoodDb and now I'll let visual studio work its magic by just clicking ad and it will go out it will create that controller, populate all the actions with all the necessary code to save things and query things from the data base, also give me all of the views that I need in my views folder. And when it's done it will open up all these new files in the editor. So now I can see that I have a restaurant controller. Let's build the application and try it out. The build was successful now if I refresh the home page we should have a restaurant link and if I click that I'll go to the restaurant controller. And there we can still see all of our restaurants now we're back in the pre-generated view that shows the restaurants in a table but I should now be able to read a new restaurant just by typing in a name and I'll give this restaurant a City and click create, and we now have a new restaurant in the data base. Let's see if we can edit this because it's actually Yellow Brick Bank Restaurant and that seems to work. Can we delete the restaurant? Yes we can delete the restaurant too. This was all thanks to the scaffolding that I selected when I created this controller, it gave me a controller that instantiated an OdeToFoodDb, give me an index action that will show all the restaurants. If we go down here to the bottom it also provided the dispose method to make sure things were cleaned up. And in between were a number of actions like delete and edit that will save and create and delete restaurants. The idea again behind scaffolding is that it gives you a good starting point. You cannot necessarily write an entire application just by scaffolding everything out. There's very few applications that are that simple but it does give you a code base to start working with and shows you some of the basic MVC patterns that we usually follow. You can certainly go in and change the controller code, you can go in and change the view code but I'm not going to drill into each of these methods just now because what's ultimately going to happen is we're going to impellent many of the same methods in another controller, the reviews controller so that we can create and edit reviews. But I will take a minute just to walk you through one controller action the details action. The details action is what is invoked when I click on the details link, it just shows all the details oft the restaurant, not the prettiest UI so certainly it's something that you'll look at changing but the way the controller works is to accept an incoming ID parameter that's the restaurant that we have to look up and then using the restaurants DbSet on my OdeToFoodDb it will invoke the find method passing in that ID. Find is one of those convenient methods that the Entity Framework provides when it's EPI, it's basically saying I'll go out and find something by it's primary key value all you have to do is give me the primary key value, if it does find that object, great, and return to a restaurant. No link queries or link operators or lambda expressions here required. If it doesn't find that restaurant it will return null, so one of the things we could do here is check for null, if we didn't find the null or return HttpNotFound which is another method that will produce a result, that essentially return to 404 to the browser. If we were to come in to this application in look for slash restaurant slash 22 where there shouldn't be a restaurant with a 22 in here, so we get the 404 error message which may or may not be what you want. If it is uncommon that a user should click a link for a restaurant it doesn't exist anymore then you might just go ahead and return a 404 but for some reason there's a lot of links out there and you delete a lot of restaurants, you might consider returning some sort of custom view. So I could write a view called NotFound.chhtml put into the reviews folder and basically put a little message there that says, "Sorry, we couldn't find that restaurant" and maybe provide a link that you can click on to go back and search for other restaurants, but for now just for turning a NotFound the area is fine, if we do find the restaurant and of course we'll pass that into the view and it will display as much restaurant information as possible. But one thing the details are missing is a list of any reviews that the restaurant might have. That would be easy enough to add to this details view, I could certainly come down here somewhere and have a for each statement and look through Model.Reviews and print out some information about each review or list how many reviews it has or its average rating. But I thought we could do something that would be a little bit of the beat and path. I'd like to be able to click on a link here that says reviews and go to the reviews for the restaurant and then have the ability to add reviews to that restaurant or edit the existing reviews for that restaurant. This will be a little bit off the beat and path because the built in scaffolding cannot create that for me, it can be close but what we'll do is write most of it from scratch so that we can actually see how some of these data modification actions need to be implemented. And along the way we'll be talking about model binding, model validation and learn some more about how to work with the MVC framework.

Listing Reviews

Let's implement a listing of reviews for each restaurant. Since we're going to not use this details view anymore I'm going to delete that from my views folder and I'll also delete the details action for my restaurant controller. And since we don't have a details action I need to change the index view that's displaying all the restaurant to not have a link to details, instead we will say here's where you can go to see the reviews for this restaurant and that will go to the index action on the reviews controller. And now there's a little set of bug here in the way that I've invoked to action link because I've picked the wrong overload. I expect that ID equals item.ID will go as the route values parameter. Because it did before I just added one more thing here, the name of the controller to call, why wouldn't it go to route values. And just because of the way that the action link overloads or design you'll notice I'm not passing this as route values, I'm passing this as htmlAttributes. And the string reviews is being passed as route values and I see this happen all the time I want to be able to show you how to spot this particular bug. If I save this view and refresh the browser and I hover over one of the reviews links that's about to appear I'll see in the little chrome pop window that previous the url that I do not have an ID in the url, instead I have restaurant query string length equal seven. Anytime you see length equal seven you probably picked the wrong over load of action link unless you really did mean to pass length in the query string. The only thing I need to do to fix this is to pass an additional parameter so that reviews becomes the controller name, ID equals item.ID goes to route values and this last parameter goes to htmlAttributes. htmlAttributes by the way, that's a way to add additional attributes to that anchor tag that is being produced. So if you wanted to give that anchor tag a special class or title attribute you could specify another anonymously typed object here something like new title equals foo and that will be emitted into the html. The passing null here will suffice if I don't want to set any htmlAttributes that should also fix the link if I come back out here and refresh the view. Now, I can see that this looks like what I need, slash review slash index slash one or slash two, so the ID is now in the url, very good. But if I click that link we'll have an exception because we comment it out our reviews controller in an earlier module. What I want to do now is select everything with control A uncomment with control KU and delete code that we don't need anymore, for instance we don't need this in memory data. Really, the only thing we need is we'll just start back with the index view that's right here at the very top of the controller. And in fact we don't need this best review anymore. We deleted that from the layout action in the previous module. What we do need is a data source. So let me go ahead and add an OdeToFoodDb private variable and instantiate up here when the controller is created. Now, we'll also go ahead and override dispose. You can just type override, have this lists pop up. Pick the method that you want to override and then type code into it. This looks good. And now, what we'll do is change the index action around. So, the index action will receive an Id parameter. And what can be a little bit confusing here is that this Id parameter is not the Id for overview. Typically, when you're working with an Id inside of the controller it's an Id for the entity that that controller is centered around. Instead this is an ID that represents a restaurant Id. That's a little bit confusing. I could in the index view say instead of Id equals item.Id I could say restaurant Id equals item.Id and pass restaurant Id along in the query string. Remember the routing engine doesn't know anything about restaurant Id in the URL so we'd have to go in the query string. I could add a new route if I wanted a restaurant Id to appear in the URL. But I want to show you another option. I'm just trying to change the name of this from Id to restaurant Id and instead of changing the routes or changing where this parameter is passed, I'm putting it into the query string, I'm just going to alias. I can do that with the bind attribute. But bind allows you to do is tell the MVC Model Binder that when it goes to look for the restaurant Id parameter value look for something called Id which is not very obvious syntax I know, but that's essentially what this attribute is going to do. When it looks for restaurant Id it will look for with the name of Id. And now I can paste on some code that will go out and find that restaurant by its restaurant Id. If it's not equal to null we will render a view with that restaurant. Otherwise we'll return a four for error. And at this point we don't have any views for the reviews controller. So, we're going to need to add a view. I'll add an index view. It's going to use razor. I wanted to be strongly typed against a restaurant review. Even though we're passing in a restaurant I'm going to strongly type it to restaurant review because I want the scaffolding to create as much of the HTML forming as possible. We're going to need to go in and fix this up because we're actually passing in a restaurant not a restaurant review. But that's okay. This is going to help me. When I click add I'm going to do a couple things. First of all let me copy this model directive because it's going to come in useful here just in a minute and change the model to say, no, you don't need a restaurant review. What you'll be receiving is a restaurant. And that means I could change this H2 tag to say something like reviews for at Model.Name and that will give me the restaurant name. And now let me take that model directive actually and paste it right here and take all the rest of those code out because this code that the scaffolding generated is here towards iterating through a collection of reviews and writing them out. But the model for this view is a restaurant. What I want to do is create a partial view that will take care of rendering this table for me and the model for that partial view will be an IEnumerable of restaurant review. Inside of the reviews folder I'll create a new partial view. Let's call it Reviews. Using a Razor I strongly type to restaurant review and we'll create it as a partial view of course. I'll click Add. Actually, delete the model directive that's here and paste this in. This gives me everything that I need. Our model is going to be IEnumerable of restaurant review. I wanted to create a table of these reviews. I don't necessarily want it to display the restaurant Id because that's meaningless to an end user so let's delete that column. I'll also delete that down here where it's displaying the value and I think we'll just go with an edit link. So, delete the details and the delete link and save everything. Let me do a build and see if this works. Oh, it won't work yet because back in the index view we need to render that partial. I'll do that ahead of the create link. So, html.partial, please render the reviews view for me and pass in Model.Reviews. I'll take the reviews that are associated with this restaurant and pass it in to that partial view as the strongly type model. It will take care or rendering the table for me. And now, let me do another build just to make sure everything is out there. And let's refresh this page. And let me go to the reviews for this first restaurant that I know there is a review in there because we see that the database with a review. But when I click that link I'll get a very strange air. This is something that you could spend a long time debugging if you don't know what's going on. Because the error is a little bit misleading it's telling you that you passed in a restaurant to this partial view. But the partial view needs an IEnumerable of Restaurant Review. And then you'll go back and look at your code, and you'll say, but I'm not passing a restaurant to this partial view. I'm passing in at Model.Reviews which should be a collection of restaurant review. What is debug? What's broken about the runtime here? But the problem is not that you're passing the wrong thing. The problem is that these reviews property is going to be null by default. And this is something that you run into with the entity framework. There's several different ways to solve this problem. So, just to make it clear let me open up the Restaurant class and tell you that in the controller when we do Restaurants.Find in passing an Id what the entity framework will do is happily load up these properties, Id, Name, City and Country. It won't touch reviews because that's information that's in a different table. There's a one to many relationship between restaurant and reviews. And the entity framework just doesn't load up this associated child collections automatically. At least not in this scenario the way I have things to find here. There's at least six different ways to solve this problem. I'll show you the easiest one which is just to add the virtual keyword here. Now, what the entity framework will do when you spell virtual correctly is that it will at runtime create a little wrapper for this restaurant class and intercept recalls to the reviews property. So that when you go to get to the reviews for a restaurant it will make sure that it loads them up for you. And that requires a second query to the database. In other words Db.Restaurants.Find will just pull in the restaurant information. Later in the view when we say Model.Reviews behind the scenes the entity framework will issue a second query to pull in those reviews. And you might say, but that's data access from a view and I'll say that's fine because it's implicit. I'm not writing explicit code data access code to load something. It's just happening behind the scenes for me. But some people will worry that there are two queries instead of one to get all this information. And for now, I'll just point you to some documentation that if you read through it will explain how to eagerly load related information at the time you do a query. I can get you down to just a single query to pull in a restaurant and a bunch of reviews. As I said there's a number of different ways to solve this. But just having the virtual keyword there is one of the easiest solutions if you're not too worried about performance. So, with the virtual keyword in place on a Restaurant Reviews, now I've done a build. Let me go in and refresh this. And this area should go away. So a very non intuitive error what is going now because we have some reviews here and it should also work when there are on reviews because behind the scenes the entity framework is making sure it gives you some collection even if it's an empty collection. Now, again, our display isn't the prettiest display in the world. It would be nice not to have ReviewerName, all this one word. We'll come back and fix that later when we start looking at some annotations that you can add to a model. For right now I want to focus on creating a new review.

Creating a Review

Let's turn our attention to creating a review. When I click on this Created New link to create a review I'm going to need to know the restaurant to associate the review with. So, in the index action I will add a parameter here that will tell the routing engine, please pass along. A parameter called RestaurantId set it to the Restaurants Id property. And I didn't pick the correct overload of action link here. I can see that's going into route values. So the routing engine will say, restaurant Id, I don't see that in the URL. So, I'll place it in the query string. This then should give us the proper link. Now, what I need in the reviews controller is an action that will respond to a Get request for create. This is a very simple action. It just needs to return a view that contains everything the user needs to type in to create a review. An early response to a get request and even though the restaurant Id will be in the query string and I can take it as a parameter here I don't actually use it in this create action. But I'm going to leave the parameter there just so we know that it's available in the query string and in the future if we create perhaps some sort of view model that will provide some defaults for some of the RestaurantReview fields. We can always populate it with that restaurant ID. But really, this is all we need in the Create Action for this simple scenario. I do need to create view so I will add a view called Create using Razor, strongly typed against the RestaurantReview and this time I will select the Create Scaffold template and we'll get a view that is strongly typed to RestaurantReview. It's using the BeginForm helper that will post back this form when the user clicks Save. It'll post back to the same URL it came from. So if we are on /Reviews/Create?restaurantId=3, we'll post back to that same location. We'll still have a restaurant ID in the query string and here's a little trick about BeginForm. What Html.BeginForm will do is return an object that implements iDisposable and that means that we can use it with a C# using statement. Typically, iDisposable objects are objects that hold on to precious resources like database connections and you want to dispose them, and an easy way to do that is to wrap something that's iDisposable with a using statement and create a code block. At the bottom of the code block, the C# compiler will emit the proper code to automatically call Dispose on that object. In this case, BeginForm isn't really hanging on to any precious connections but when you call Dispose on that object, that's when it will emit the closing form tag. So that's just some information on how BeginForm works and why we're placing it inside of a using statement. Inside of the form, we'll have some validation messages. We'll see those in this module. We also have some literal text that we could make a little bit prettier instead of RestaurantReview all one word, I could give this a legend of New Review. Then we have labels and editors for everything the user needs to fill in, the rating, the body, the Reviewer Name and oops, here's restaurant ID. So sometimes the scaffolding is a little eager and emits things that you don't necessarily want these or to edit. We're going to delete that. Sometimes it is not so eager and it doesn't emit things that you want it to, but in this case we just had to delete something. That should be enough so that if I do a build and make sure everything is saved, if I refresh this page, to get the new Create New link that has a restaurant ID in it. I should be able to click this and here we're on a page where I can enter a rating, I can enter a body, and I could enter a Reviewer Name and click Create. That'll do a post back. So we need a controller action on the Reviews controller that will respond to that post request. That action will look like this. We respond to a post request and we take a RestaurantReview as a parameter. That's where the model binder in MVC will jump in and say, "Oh, you need a RestaurantReview." I'll instantiate an instance of that type and then look through all the properties to see if I can match them up with something in the request and it will find a rating, it will find a body, it will push those values into it. It will also find the restaurant ID so it will populate the restaurant ID property of a Review. So it's associated with the proper restaurant and then inside of that Create Action, the first thing you typically do is check to see if ModelState is valid and if this flag is valid, returns false, that means something went wrong with the model binding. Something that was required isn't there. Something is too long or something was too short. It runs through all the validation rules associated with an entity and tells you if everything worked or not. We'll add some additional validation rules in this module but for now, just know that if that returns false, what we want to do is not save the thing in the database, something was invalid, instead, we'll return the Create View again and let the user try to fix any problems that might have occurred. Those Html.validation message for helpers will show validation messages to the user. We'll see that here in just a second. But if ModelState is valid, that means everything was proper. So we will tell the Entity Framework to please add this review to its reviews collection and then the Entity Framework doesn't actually save anything to the database until you call SaveChanges. At that point, it will issue Insert or Update or Delete Statements or all three, depending on how many objects you changed to reconcile the changes that you made in memory with what's in the database. In this case, it should just issue a single Insert statement to insert a new review into the Reviews Table and if that works, we will redirect the user back to the index action. We don't want to let them sit there on the Create View that was the result of a post because they might hit Refresh and create another review accidentally or redirect them back to another page where they can see that review. And now I should have everything that I need if I do a Build where I can come to this page and click Create and there I can see my new review in the list. Let's try it with a different restaurant. Let's see if we can add a review for a restaurant that doesn't have any reviews yet. And that seems to work. And now before we start talking about validation and how to add additional validation rules, let's finish off the edit scenario.

Editing a Review

For editing a review, we already have a link specified here that will take us to an edit action and pass along the ID of the review to edit. All we need to do to respond politely to that request is to implement an Edit Action that responds to an HttpGet. All we need to do here is take that ID parameter, lookup a review using that primary key and then return a view that will allow the user to edit that. We need the view so I will right click and add a view called Edit using Razor, strongly type to a RestaurantReview. This time the Scaffold template will be Edit. This is going to look very similar to the Create View. It still has a BeginForm. It still has validation summaries. It still has a legend that we don't quite like so we can say Edit Review. It does have something a little bit different here. It has a hidden input for the model. That hidden input will allow us to track the ID of the review that's being edited and it's hidden because the user doesn't need to see it. We just need to preserve that piece of state between the requests, and then we have the labels and the editors for Rating, Body, Reviewer Name and ah, here again is restaurant ID. This time I'm going to delete it again but we also need to keep track of this restaurant ID because when we update this review, we're going to need to properly populate the restaurant ID field. What I can do is just add another Html.HiddenFor that says given a model called Model, please give me a hidden input for restaurant ID. There are some other approaches that we could use for this but this makes it easy and now when the user clicks on the Submit button, we will have to respond to that post request. So back in the Reviews Controller, I'll implement an edit action that responds to a post that's very similar to create but it has one little difference. It's still going to take a RestaurantReview. The MVC model binder is still going to populate that object with things that it finds in the request. So again, we don't have to look in the query strings or post of form values ourselves. The model binder will take care of all that for us. We'll still check if ModelState is valid. Return to Edit View if ModelState is not valid because that will allow the user to fix any problems that they have but if it is valid, we're going to come in and use a slightly different API and I do need to bring in a namespace here, System.Data, for this to be legal C# code. But now that it's legal, I can tell you about this entry API on the db context. The entry API is a way to tell the Entity Framework, "Here is a review that I want you to start tracking so you can make changes in the database for this review but it's not a new review. I don't want it inserted into the database. Instead, this is a review that's already in the database. I just want you to take ownership of this object and treat it as if it had some new data inside." That's the purpose of EntityState.Modified. Now, when I call SavesChanges, you will update that record in the database, the record that has the same ID. In the Create Action, we add a review and EF will insert the record. In the Edit Action, we attach an existing review with that entry, set the state to modified and EF will update the record, and once that is successful, again, we'll redirect the user back to the index action where they should be able to see the effect of their edit. So after a build, let me come in and edit this review and because I'm really excited about this restaurant, I will add extra exclamation points and click Save and I can see that the edit worked. And we're not going to go into validation just yet. Before we do that, I want to tell you about some serious security implications of model binding, some things that you need to know about.

Mass Assignment

You do need to be careful with model binding in MVC. The model binder has a brute force algorithm that will try to move anything and everything into your model that it can possibly move. This means you might values you didn't expect in a model which can lead to security related problems you might not have thought about. Let's pretend that once a review is created, no one should be allowed to edit the Reviewer Name of a review. If you didn't watch this video, you would think that would be very easy. All I need to do is take Reviewer Name out of my form. If there's no more editor for Reviewer Name and I don't see it on the form, how can I possibly change the Reviewer Name? Well unfortunately, there's all sorts of tools out there that you can use to modify data in a request or just create a request with arbitrary data and send it off to a server. People use these tools all the time to break into servers and send people information they didn't expect. In fact, we don't really need a fancy tool here, all I need to do to get something into reviewer name is to add a parameter to the query string. I'll add reviewer name equals hack. We usually get request for this form and now has that value in the query string and I don't need to save anything here, all I want to do is get that reviewer name in there. If I click save, we now have a new reviewer name. And the problem here is that the model binder didn't know about my forum, about my view code, it didn't know that I didn't want a reviewer name to be edited. All it knows is that the restaurant review has a reviewer name property and that the request had a reviewer name value so it moved the value into the property. This problem is often referred to by one of two names. Some people call it over posting that is we created more values than you expected in the request but it also goes by the name mass assignment because the model binder just assigns all the values it can without regard to how you wrote your view code. If someone knows the right syntax, they can move anything into your model. There's a number of different solutions to this problem. I'll show you one quick and easy solution probably not the best solution but if you want to see more alternatives, you can read the post on my blog where I cover this topic in more detail. For now, let me show you one quick easy solution which is to use that bind attribute again. We use this before when we aliased a parameter name. This time, I'll show you some of the other things that you can pass in the bind, they include a list of things to exclude and a list of things to include. You typically pick one or the other. Exclude is a black list. I can say please exclude the following fields, reviewer name. And if I had additional field here, I could just separate them by commas but that's the one thing I want to be excluded when you do the model binding. You can also use include and that's a white list, I just list all the things that you do want to be bound. But exclude works for me well here because I only need to exclude one thing. Another common approach to this problem is to define a view model. This time, an input view model and the view model would only have the properties you expect to receive from the user, nothing else. You'll use the view model as the parameter to your action and then you'll need to copy the values into a real entity to update the database. But by only having the values you expect from the user in the view model, you can be sure you don't pick up anything unexpected from one of your shadier customers. Also, related to security, make sure to watch the security module and learn about cross site request forgeries. Cross site request forgeries are a dangerous settle attack you should worry about when posting form data and we'll look at the attack and how to avoid a cross site request forgery in the security module later in the course.

Validation Annotations

Currently, when we edit a review, we could run into a couple problems. There's nothing that constrains the rating to any sort of scale so if someone wanted to save a rating of 1000 into the database, we would let them. There's also nothing that requires them to enter a body or restricts them from entering a body with 10,000 exclamation points in it. These are all types of validations that we can do very easily with the MVC framework. Before we look at them, let me just add a couple of things back into the project. Let's remove the bind attribute for right now, I actually want people to be able to edit the reviewer name. I'll hit control Z in the edit view to bring back the editor for the reviewer name. And now, let's go over and look at restaurant review and apply some data annotations. So, one annotation that you can apply is the range annotation. I can say that this rating has to be between one and ten and now, when we do model binding in ASP.NET MVC, the model binder after it moves property values into this properties, it also exercises any validation logic that is applied through this attributes. So it defines that the rating is 100, it will tell you that the model state is not valid and then you can let the user try this again. There's also a required attribute although putting this on an integer property is a little bit redundant because to the model binder, an integer property is required by default because integers and C sharp are value types. That means they cannot be null. And when the model binder comes across a value type like int or date time, long and decimal, it automatically requires a value to be present in the request for that. We can also come in and tell the body-- tell the model binder that the body should be required and we can also give it a maximum string length, let's say 1,024 characters. In addition to this attributes, there's also attributes to apply a regular expression validation, there's attributes to compare two properties and there's an attribute to do a remote validation. Actually, call back to the server and check your property value as the user is typing into a forum. You can read all about these validation annotations on MSDN and MVC documentation. So just having this attributes in place and doing a build, we'll make something different happen in my application and that something different might not be something that you expect right away but that something different is this. The entity framework is always checking the model that is in effect against the model that you use to create the database including any migrations that have been put into the database. And at this point in time, it is detected that there is something different about the model and if you think about it, some of these attributes really do influence the database schema. For instance, saying that Body is required is like saying I need a not null column. Previously, this was a null column, you can insert null values into it. Specifying a string length gives this a definite string length. Previously, it was nvarchar max, now it has a maximum length of 1,024 characters. If I want it to get pass this error, there's a couple different ways to do it. There is a way to just tell the entity framework trust me, I know what I'm doing, just work with me and move ahead. Or, you can actually apply the migrations to get the database schema and sync with what is in your C sharp code and that's what we'll do. Remember, we have automatic migrations enabled so what I am able to do is just come in and say let's update the database and I'll throw in the Verbose flag. If I do this, it's going to fail the first time because migrations is smart enough to realize that we're taking one column, the body column or choose to be nvarchar max and we're making it smaller that could result in data loss. But here, I can say basically, I know what I'm doing and I want to force this through and now, we can see things like alter table RestaurantReviews, alter the column body to make it nvarchar 1024, not null. And then we rerun the seed method, that mens all my data should still be in here. We didn't loose anything. And now, after I do a build one more time just to be safe, I should be able to refresh the home page and we'll have a working application again. So let's go into restaurants and look at one that has reviews and see if we can edit this review and put the value of 1,000 in for the rating. No, as soon as I tab off of that, it will give me a message that the field rating must be between one and ten. Notice, I didn't even click the Save button and that's because one of the nice things that the editor for HTML helper will do for you is emit some Metadata into the input tag that client-side JavaScript can pick up and exercise the validation logic for you on the client-side. So before the user even tries to submit these to the server, we can tell them that they've done something wrong. Same thing should happen here with Great, if I blank that out and try to click Save this time, now, it can come back and tell me the body field is required. That request actually didn't go to the server. That was triggered on the client side by JavaScript. Some of these validations require a little more than just filling something in and tabbing out of the input. But as soon as I start typing in here again, the validation message goes away. And as a user, I feel happy and I can also fix the rating here. And just to be clear, these validations are run on a client side, they're also run on the server side so even if the user has JavaScript disabled for some reason and a bad value comes into your server, the model binder will run the same validation rules after it's done its model binding for this restaurant review. And if any one of those rules failed, it will set model state is valid equals False. And then when you re-render the view, things like the validation message four will pick up those error messages and display them in the view. In addition to data annotations for validation, there's also a data annotations that can influence the display. So if you remember, we had HTML helpers like display for and display name for that we would use to write out things like the reviewer name here. And the problem is that reviewer name as all one word isn't very pretty. That's being written out by LabelFor ReviewerName. So let me introduce you to just a couple of the display annotations here with the reviewer name. One of the things we could use here is the display annotation and one of the parameters we can pass for the display annotation is the name that should be used when this is shown in the UI. So I could just say this is perhaps user name as two words. There's also a display format attribute. With this attribute, you can specify things like the data format string that should be applied that's quite useful when you want to format numbers and dates for display. For this one, what I could do is say that the NullDisplayText perhaps is anonymous. We don't know who the user is and it always helps when I spell that correctly. And now, let me do a build and come back out and refresh this page. And we can see that Reviewer Name has changed to User Name. And if I blank out the User Name to get rid of this ugly value that some malicious user put in my database, I can see that User Name displays as anonymous when it has no value. When I go into Edit it, I see the real value that's there which is empty, it's just when we use the display for helper, it's going to use that null value display text in place of a null string. Again, there's many more Display attributes they can use, if you just want to poke around inside of the System.ComponentModel.DataAnnotations name space, I think you'll find a number of useful attributes in there.

Custom Validations

I'm sure you can think of many validation scenarios that are not covered by the validation attributes provided with the framework. This is when you can implement custom validation logic. And there's a couple different approaches you can take that will plug right into the framework. One popular approach is to write a custom validation attribute. Use this approach when you have validation logic that you want to apply to multiple models. An example might look like this. I'm going to temporarily paste some code inside of here, that's not something we're going to keep in the project. So I'm just going to keep it in the same file as a Restaurant Review and we'll remove it after I explain what it is. This is a maxWords attribute. What I can do with this is restrict the number of words that a user can enter into a string property. It does this just by using a simple check and looking at the number of space characters. So it's not an advanced validation. I'm just really showing you how to go about implementing this approach. And before I explain what it does, let me just show you how you could use it. Let's say that we wanted to restrict Reviewer Name to be a single word. And I could say, maxWords and you need to pass in a parameter which is the maximum number of words. So let's say just a single word for Reviewer Name. And now, if I do a build and come out into the application, I'm going to try to edit this User Name and put in Scott Allen and click Save, User Name has too many words. But if I go with just Scott, everything is fine. So my validation attribute is working. Let me remove it from here and I'll show you what it does. First you derive from a base class ValidationAttribute. The MVC runtime and the model binder, they respect ValidationAttibute. So that automatically plugs you into the framework. And when you derive from ValidationAttribute, you can override a method is valid. When the model binders performing validation on your model, it will pass in the value of the property where you place this attribute. So here, it would pass in the Reviewer Name when it's validating a Restaurant Review. And then it's up to me to inspect that value and see if it meets the proper criteria. We're just doing a String.Split to create an array and it's going to split up that string wherever it sees a space character then look at the Length and see if that is created with maxWords. So again, a very simple brute force validation. I'm sure you can think of all sorts of edge cases where this would fail but this is really more just about showing you how to implement custom validation logic. You could have anything inside of here. If the model passes validation, you can just return ValidationResult.Success. But if there is a problem, you want to return a new ValidationResult and associate an errorMessage with it. And the error message you can create using another method that you inherit from the base class FormatErrorMessage, we actually set the error to display string to use for this attribute and the constructor. And then back in is valid. We'll build that string using Display Name. And that's what ultimately produce the string, User Name has too many words. It also allows other people to set a custom error message if they want. You can do this actually on any attribute even the built in attributes. Anyone of these attributes, I could walk up to them and say, errorMessage equals and have a custom error message here that will display when the user is outside of that range. And I could do that for maxWords too just because we're using FormatErrorString and deriving from a validation attribute base class. Now, this type of validation only happens on the server-side. You can implement custom JavaScript code to plug in to the client-side validation framework. You'll find many examples of this on the web, just remember you don't need to duplicate all the validation logic on the client. Some validations are difficult on the client. If you at least have the required and length attributes and simple text like that, you can give quick feedback for common errors that's going to help your users out quite a bit. In addition to this validation attribute approach, you can also have a model object implement and IValidatable Object Interface. This is a good approach to use if you want to do a deep inspection of a model because what I can do with this interface is use control period to tell Visual Studio that I want to implement this interface and it will give me that validate method. That's the method that you implement to provide the custom validation logic. And it's inside of here where you have access to the entire model. When you write a validation attribute, you only have easy access to a single property, that's the property value where the attribute I was used and that's the property value that the model binder will pass in to you. You can get to other properties on the model but it's a bit messy. With Validate, I'm inside my model object so it would be really easy for me to exercise validation logic that looks at multiple properties, for instance, I could say something like, if Rating is less than two and ReviewerName ToLower, startsWith Scott and maybe we just have a trouble maker on the site with the User Name of Scott. And it gives everyone a bad and low rating. We just want to prevent him from entering messages. And what I could do here is use yield return and create a new ValidationResult with an error message. And otherwise, if everything succeeds, I don't need to yield return anything here, that'll essentially produce an empty IEnumerable and the MVC runtime will know there's no problems. And now, if I build the application again and we come into edit a review and I have a User Name of Scott and I tried to put in a Rating of one. I'll save the error message. Notice that this error message appears above any properties that's because we didn't associate this validation error with a specific property like User Name or Rating. And in that case, it just gets associated with the model and it will display in the Validation Summary. If I open up the Edit view for a Review real quick, that error message is appearing here in the Validation Summary which by default only shows errors that aren't associated with a specific field in the model. That's what the true parameter is telling the Validation Summary to do. If you change that value over a false value, the Validation Summary will display all errors on the models for all properties and also the model of all errors.

Summary

In this module, we added the ability to create, delete and edit restaurants and also added the ability to create and edit reviews. We looked at Model Binding, some Entity Framework APIs for data updates and saw how to use data annotations to influence the display of the model information and also to validate model information. We saw two different approaches to custom validation that work with the Model Binder and the MVC framework to automatically set model state and populate our error messages. In the next module, we'll be turning out attention to JavaScript and adding some Ajax features to the application.

AJAX and ASP.NET MVC

Introduction

Hi, this is Scott Allen. And in this module, we will look at adding AJAX features to an MVC application. Specifically, we'll look at the AJAX helpers offered by the MVC framework. We'll also spend time using jQuery and jQuery UI to work directly in script code. I'll show you how client validation works, how to minify and bundle your scripts, how to produce JSON from your application, and what to do with JSON once you have it on the client. I will do all these things while adding features to the application.

The Scripts

The letter J in AJAX stands for JavaScript which means we'll need to send JavaScript to the client. And I think it will be good to look at what Script Libraries are already available in the project and how to manage those scripts to get them to the client which will demonstrate some new features in the framework. So inside of the Scripts folder in Solution Explorer, you can see the JavaScript files are available when you create a new project. We'll start at the top with \_references.js. This file works with Visual Studio 2012 to give Studio a list of the files you commonly use so that you can get better IntelliSense when writing JavaScript. If we open up the file, we'll see it contains references inside of JavaScript comments. These references look suspiciously like XML. But these references are telling Visual Studio what files you commonly use. We can see the list includes jquery, jquery-ui and more. And now, Visual Studio will go out and find those files and parse them and try to provide better IntelliSense for you. It should know about method names and attributes of the objects that you work with from these libraries. If you were commonly working with additional libraries in the application, you probably want to list those files here too. Underneath of references is the jQuery library. There's three versions of the core jQuery library file. The first one has IntelliSense in the name. You'll never use this file. You'll never send it to the client. This is just another aid for Visual Studio to generate intelligent prompts with IntelliSense. When the references file says to reference jQuery, Visual Studio will automatically find this file which has some additional comments inside to make Visual Studio look smarter. The jquery-1.7.1.js file, that's the core jQuery library that you can send to the client. If you aren't familiar with the jQuery, then it is a JavaScript library that gives us an API we can use on the client to select and manipulate DOM elements, make asynchronous call to the server, and lots more. jQuery supports all the modern browsers including Internet Explorer, Chromes, Safari, Firefox, Opera. And if you're new to jQuery, you'll see a few tips on this module. This version of the jQuery file is very readable. If you open it up, you can find lots of white space, well-formatted code, and comments. The jQuery file with .min in the name is minified. Minification is the process of going through a JavaScript file and making it small as possible by removing unneeded white space, unneeded comments, and even shortening the names of symbols and variables that are exposed to the outside world. If you open that up, you'll see it's not very readable. But it is smaller, and by making the file smaller, the browser will need less time to download the file, which means, your page can load faster and users like fast pages. However, we don't really need to touch this file because there's feature in ASP.NET MVC 4 that will automatically minify files for us or pick up minified files as we'll see in just a bit. In the Scripts folder, we also have jquery-ui which is a plug-in or extension for jQuery that gives you UI widgets like dialog and accordion that you can use. There's a minified and unminified version. We also have jquery.validate, another plug-in for jQuery. This one provides client side validation features. The MVC framework actually relies on this library. We also have knockout.js which you can use on the client to apply a model view-- view model design pattern in client script. It provides features like declarative data by mean. And then, there is modernizr, which as I mentioned early in the course, is a library to detect and enable HTML5 features on a browser. The files I skipped over all have unobtrusive in the mean, and these files are authored by the MVC team and they serve as a bridge between ASP.NET MVC and jQuery. In order for some MVC features to work like client side validation, you need both the jQuery validation library and jQuery, and the jquery.validate.unobtrusive script file because it takes metadata that's omitted by HTML helpers like Editor 4 and feeds that data into jQuery validations so it can apply the validation rules on the client. We'll see that metadata in this module. And now that we know what's here, let's look at how we send these script files to the client in an efficient manner.

Managing Scripts

Libraries like jQuery and Modernizr are libraries that you typically use everywhere in an application, which means you need a script reference on every page. And as we've learned, it is the Layout views that are the place to go when you have some common markup that you want on every page. This application only has one Layout view. It's in the Shared folder. It's called \_Layout. And at the very top of the Layout view, you can see a call to Script.Render. And this somehow involves Modernizr. Before we talk about the details of Scripts.Render, let me just assure you, this will put a script tag referencing the Modernizr JavaScript library into the top of the page here, inside of the head tag. It is important to include some JavaScript libraries at the top of a page. This is true for Modernizr. Modernizr needs to start executing before the rest of your HTML goes to the client. As we talked about in the first module of the course, Modernizr will make sure older browser like IE6 are prepared to work with a new HTML5 elements like nav and section which appear in this page. And to do this, Modernizr has to start executing before those elements appear. Other libraries and scripts don't need to appear at the top of every page. You can instead include them at the bottom of a page. And if we come down to the bottom of the Layout view, we can see another call to Script.Render. This one somehow involves jQuery. When possible, you should add scripts to the bottom of the page because it can make a page perform better, or at least, it looks like the page is performing better because the browser doesn't have to block as it downloads the script and executes the code inside. Web browsers tend to bring everything else to a halt when they encounter a script tag. But by placing scripts at the bottom, we get more content to the client as the HTML images will start to display and then finally, the browsers sees the scripts and loads them at the end. And now, let's go to the question of what is Scripts.Render doing? And for that matter, what is Styles.Render doing? Because it turns out, these two calls are closely related. They both rely on new features and ASP.NET that can bundle and minify files for you. Scripts.Render can give you a minified job script bundle. Style.Render can give you a minified bundle of style sheets. What's a bundle, you might ask. Well, another way to boost the performance of a webpage is to combine files together, so a browser doesn't need to download as many resources. For example, if you have three style sheets to include in a page, you might want to bundle them together into a single style sheet so the browser only downloads one file instead of three. But, during development, it's often nice to have three separate files. They're probably creative for different purposes and it's easy to find and maintain the code when it's nicely separated. That provides a bit of tension because we went three files for development, but a single file for download. And this is where the bundling feature in ASP.NET steps in because it can dynamically bundle files together at runtime. To see this, let's open up the Global.asax.cs file, look in our friend, the application start event where we can see a call in here to BundleConfig.RegisterBundles. Up at the cursor there, and strike the F12 key to open that file up. And here we arrive inside of a class that is building various style bundles and script bundles. There are two parts to a bundle. First, you give the bundle a virtual path to reach the bundle. In other words, if I open up a browser and come to this URL, I'll receive all the files that are included in the bundle. This is also the path or the identifier you use when you reference a bundle and want to render it using Scripts.Render or Styles.Render. The files that are included in the bundle are listed with calls to include and you might want to include one file. You can certainly have a one file bundle. But you can also include multiple files. You can specify a full name here. You can also specify a place holder for a version number and you can use wild cards. For instance, the first script bundle here that is basically including jQuery is going to have a placeholder for the version. That means if I upgrade jQuery from -1.7.1 to -1.8.0, I don't need to come in and change my C# code. The script bundle will automatically pick up the latest version from the file system, whatever is in my Scripts folder, and it's smart enough to distinguish between IntelliSense files, minified files, and unminified files. So even though we have three versions of that core jQuery library, the IntelliSense, the minified and the regular file, it's only going to pick the one that it needs and it won't include jQuery more than once. You can also use wild cards. So down here in the jQuery validation bundle, you can see a call to jquery.unobtrusive, that means go out and find all the files that start with jquery.unobtrusive. But again, it's going to be smart enough not to include both the minified and the unminified file, it can distinguish between those two. So the first bundle, the jQuery bundle, that's a bundle of one file. And that's okay because in bundling will add some other features to this, it can minify files that aren't minified. In the jQuery validation bundle, that's going to include at least two files, jquery.unobtrusive and jquery.validate. Once you've created a bundle which includes all of the files that you need, you render it in markup using Scripts.Render or Styles.Render as we saw on the Layout view. And you can see, it is using that virtual path, that identifier that you selected for the bundle. And I call it a virtual path because there's not really a file at that location. But the MVC runtime is smart enough to intercept that request and route it through the bundling and minification pictures here which will served up the files you need. So once you've created a bundle, you render Scripts.Render or Styles.Render. And these methods will behave differently depending on your application configuration. So for example, back in Bundle.Config, there was a StyleBundle to find for all of the jQuery UI style sheets. You can see those quite a few of them because they factor them out based on features. If you wand to use just a dialog, then all you need to do is include the core style sheet and the dialog style sheet. But if you use everything, you'd want to include all these style sheets. I'm going to take this bundle and render it from the Layout view because eventually we will be using jQuery UI. So I just need to add a call to Styles.Render and pass that path in. And now, let's run the application. And I'll do a View Source. And you can see, with my single call to Styles.Render, I omitted a link for each style sheet into the page, almost 12 links or exactly 12 links. So it added a style sheet link for every file in the bundle. You might think that's not much of a bundling feature if we have individual links, or certainly not combining things into a single download. But this is because we are running the application in debug mode. And in debug mode, you'll find it easier to debug scripts and styles when they are not bundled up into a single download. However, if I come over into the Web.config file, we can change how we're running. I just need to find the debug setting which is right here at the top. We'll change it from true to false so that we're running in release mode. And I'll run the application again, and do a View Source. And now, you can see there is a single link tag admitted since we're in a release mode. And if I were to take this URL and place it into the browser to download this directly, it will all get back as a css file, let's open it in Notepad. And that's all of the css from all 12 of those files combined into a single file. It's also been minified, so it's been made as small as possible. There's no white space or comments to make this a larger download. This bundling and minification features then can really decrease the amount of time needed to load a page, and they're easy to use. We'll be using them in this application as we send more scripts to the client just by defining the new bundles and using Scripts.Render to put those scripts in the client. ( Pause )

AJAX Helpers

Now that we know how to get scripts down to the client, let's add some AJAXy features to the application. We'll start with a homepage. We currently have a search here which is working just fine, but let's make things a little bit harder. Instead of three restaurants, let's use 1,000 restaurants. And the easiest way to do this is to see the database with a thousand restaurants. I'll swing over into our Migrations folder and open up Configuration.cs. Inside of here, you remember, is our Seed method. And what I can do inside of here is just add a loop from 0 to less than 1,000. And for each iteration of the loop, we'll just add another restaurant that just has a name like 1, 2, 3, and 4. All these restaurants are in no where USA. Just having that code in place, if I open up the Package Manager Console, I should be able to update the database which will execute that Seed method and will have a lot more restaurant in the database. Now, the Seed method is complete, so let's swing back into the application. And now that we have all these restaurants, we're currently only taking the top ten. It becomes much more important to be able to search. We'll also need some paging, but we'll worry about that later. Let's say I scroll halfway down the page and I search for restaurants. Let's start with 77. And the search still works, I can find restaurant 77 and 770. But you'll notice what happens when I do the search, when I scrolled down into the page, is I completely lose my scroll position. And that's because we're doing a full request back to the server withdrawing the entire page and it starts fresh. It would be nice if I could search for restaurants and not lose my scroll position. In other words, I just type in a search term, click Search, and just this portion of the page refreshes. You could think of this is a partial page update very similar to what update panel could do in ASP.NET. Let's add that feature, but first, I want to fix somethings up the way that scripts are managed in this application right now. What I want to do to simplify things is have a single script bundle that's rendered at the bottom of every page from the Layout view. And in order to do that, I'm going to need to fix up a few things that I've been scaffolded into the data modification views like the view to edit a restaurant. When we've been scaffolding up these views, we've been allowing the scaffolder to include Scripts.Render at the bottom of this views or inside of the scripts section. And it's rendering the scripts for everything that's needed for jQuery validation and the clients side validation to work. Basically, I want to take those scripts out of here. If I just do a Search and Find in Files for section Scripts, I'll find all the files where this is in place. And what I'm going to do is coming to each one, and I'll leave the script section in place, but I want to delete that Scripts.Render from each one. And now that I've deleted eight instances of that script bundle, let's close out all the documents that I opened, and come in to out App\_Start folder. It's inside of here where we will define a new bundle in BundleConfig.cs that will take care of all our script needs. I'll paste the code in here at the top of the class. We're essentially going to create bundles under ~/bundles/otf or OdeToFood. And we'll include jquery, jquery-ui and all of the scripts we need for unobtrusive JavaScript and jQuery validation. And now, I can come in to the Layout view. And at the bottom of the page, instead of rendering just jQuery, we will render that script bundle. And now, the application should behave the same. We're just going to have more scripts inside of every page. It's going to simplify things for us in this application. And now, we can work on this feature in the Index view for the HomeController. Inside of here, let me comment out the simple form that we have in place and replace it with the call to Ajax.BeginForm. Ajax.BeginForm is very similar to Html.BeginForm. It writes out to form tag and you can tell what controller and action and route values to use. But Ajax.BeginForm will make an asynchronous request to the server. Html.BeginForm makes a synchronous request to the server that withdraws the entire screen. Ajax.BeginForm makes an async request and it can withdraw just a portion of the screen. All you need to do is tell the helper what to update on the page, and you could do this with the AjaxOptions object that we pass it. We tell the helper to make a get request and whatever comes back should replace the content in the page that's currently identified by this UpdateTargetId. We don't have an HTML element by that name, so I need to add one. This element will delimit essentially the area that we want updated. So I want to update all of the restaurant information and all of that is now inside of a div with an id of restaurantList. And with just this code in place, let me press F5 to run this application with the debugger. That will prompt me to change by Web.config back. If you remember, we went into Web.config and set debug equals false. This is going to set that back to true which makes things a little bit easier to debug if something goes wrong. I'll just click Okay and let it make that change. And now, in the application, let's search for 23. And you can see this kind of works. Asynchronous form made a request back to the server, got back a response, and then we're after that into the page where the restaurant list used to be. So we now we have a page inside of a page that has a restaurant list and I could continue doing that infinitely. So the problem in here is that the client side stuff all worked, we need this server to respond differently when we do a search. We need it to return just updated restaurants that should be displayed on the page. And that means this restaurantList needs to display in two ways. First of all, it needs to display on the homepage when the user first comes to the homepage. We need the restaurant list there. And then, when we do a search, we need the restaurant list to display and just the restaurant list. And since we're going to need to this in two places, that means we'll be using a partial view. Let me cut the restaurantList out of the Index view and come into the views for the HomeController and add a view called \_Restaurants. This will be a Razor view. It's going to be a partial view. And I'll just click Add and paste in our HTML. We can also make this strongly typed. It's going to have the same model is the Index view, so I'll just copy the model directive out of the Index view, paste it into \_Restaurants, that's our partial view. That's part of what we have to do. Now, we can render this when someone renders the Index view, so that we get a full page with the restaurants and the header and the login and everything. So we'll render Restaurants and we'll send the model along to be rendered. And now, it's really going to be up to the controller to decide how to render a response to the indexed action because this is all happening through the index action, the homepage, the search results. Let's open up the HomeController and I'll show you how easy it is to make this decision inside of the HomeController. What I'll do is stop debugging, shift F5. And then paste in a little bit of code that can determine if a request arriving to the server is in an asynchronous request. It does this behind the scenes by looking into the HTTP headers. There'll be a special flag there that you can find out easily just by asking Request.IsAjaxRequest. So if it is an asynchronous request, we'll return just that partial view with the restaurant information, otherwise, we'll return the full view. Use the same model object for both views. But now, if I run the application again, we should have the result we're looking for. I'll scroll down the page just a little bit and search for 24. And you can see that bottom portion of the screen just updates instantly. We don't lose our scroll position. In fact, it's sometimes hard to tell that the screen changed. We might need to add a little animation or fade-in, fade-out effect, just to draw the user's attention to the fact that the screen has in fact updated.

An Async Search

We've just seen how Ajax.BeginForm works. Although, we haven't really seen how it works behind the scenes. I think, it's interesting to take a look at what is happening there because we'll see some interesting techniques we can apply to our own JavaScript code. There are basically three features MVC provides out of the box that we will consider Ajax related. There's Ajax.BeginForm, which we just used, and there's also an Ajax.ActionLink. Just like Ajax.BeginForm, Ajax.ActionLink creates a link to make an asynchronous request and update the screen instead of navigating the browser to a new page. The third MVC feature is Client Side Validation. All of these features use an approach to JavaScript programming that is known as Unobtrusive JavaScript. Unobtrusive JavaScript means, we don't have JavaScript littered throughout the view on click events. That's obtrusive to other developers who try to read the code. And it's also obtrusive to users because it usually means the content is only available if they have JavaScript enabled. If I were to remove the scripts from this page or turn off JavaScript in IE, the homepage would continue to work and the search would continue to work, it just wouldn't be nice and asynchronous. But it would continue to work and this is because of the way the scripts interact with the page. If we look at the source code, we'll get an idea of how BeginForm works. The AjaxBeginForm helper admits data-attributes into the form tag. Data dash attributes are part of the HTML5 specification and you are allowed to invent as many different data dash attributes as you want. They're basically private data for the application to consume, the browser ignores them. Then in scripts that you load on the page, there will be some JavaScript that will go and interpret those data dash attributes and start attaching events or doing whatever is necessary to add AJAX features, whatever the data dash AJAX instructions tell them to do. Client side validation works the same way. Let's close this view and go out to a Review Edit page and look at the source code to this. And here, you can see data dash attributes that specify all the validation rules for rating. They're on the input for the rating value. They include things like the minimum value, the maximum value, and the error message to show it something is out of range. So there's JavaScript code coming through, finding these data dash attributes, interpreting the values inside and then adding behavior to the page. It also works if JavaScript disabled, we won't have client validation, but nothing will break. The data dash attributes will go unused, but the form will still post and we'll have server side validation. Let me show you how easy it would be to come in to the Index controller. And instead of using Ajax.BeginForm, we'll go back to our simpler form and implement this using just our own JavaScript code with some jQuery. I will add some additional information to this opening form tag. So it's still method equals get. Now, I'm going to explicitly specify an action to make sure this form has an action attribute. I'm using the URL helper to generate a URL to a controller action. In this case, the HomeController index action, and then two data dash attributes. One to identify that this form should be AJAX, (inaudible) to behave asynchronously, and one to ID the DOM element that needs to be updated when data comes back from the form submission. Very similar to what we had in the AjaxOptions that we have to pass into begin form helper. And I'm using a bit of a prefix in here, otf, just so my data dash attributes don't conflict with any data dash attributes defined by the MVC framework. And now, we will need a script to interpret these data dash attributes. Let me come in to the Scripts folder and add a new item. We'll search the installed templates for script, find JavaScript File, and let's just call it otf.js. This will be a JavaScript file that we use throughout the application, so not just on the homepage. And I can have these asynchronous forms anywhere. For this to work everywhere, we will have to add it to our bundle that is rendered on the Layout page. So I'll just add this new script down here at the bottom. It's going to depend on jQuery, so it has to come after jQuery. But right there should be fine. It will now be everywhere in the application. And now, let's add some code to the script. The first thing we'll do is use jQuery to hook up to the DOM ready event. If you haven't used jQuery before, this is when you just invoke the jQuery function which is the dollar sign, and then pass in a function that tells jQuery that you want to execute some code when the DOM is ready. That's the point when all the HTML has been received, has been parsed by the browser, it's been put into memory in the document object model or the DOM. So all the elements are in memory, they're already to be scripted. Then we can use a jQuery selector to go out and find all the forms that have this data attribute present and set to the value true. Another great feature of jQuery is that it lets use css selector syntax like this to select elements on a page. This would select one form element or zero form elements or multiple form elements just as-- however, many have the data-otf.ajax attributes set the true inside them. And then the third feature of jQuery is that it's very easy to wire up events. So once I have selected zero or more form elements, I want to wire up to submit events, so that when the user clicks a Save button or a Submit button, instead of that posting back to the server, instead of that going back to the server, it will come into my JavaScript code. In this case, call a function ajaxFormSubmit. We haven't written that function yet, but it's just a normal function that you can write in JavaScript. Inside of this function, it's going to be our responsibilities to handle this form submission. So we'll have to collect all the parameters, send them off to the server, get the result back from the server, and then graph it into the page somewhere. So the first thing I'll do is grab a reference to the form that is being submitted. You can do that because it will be the this reference inside of the event handler. And then, I'm just going to wrap it inside of jQuery so that I can use jQuery functions on that element. I want to use those functions to build an options object that will contain the URL that we're going to go to. I can get that by looking at the action attribute that is on that form. The type of request to make, that would be a get or a post and we can get that from the method attribute on the form, and finally the data to send along to the server. Whatever inputs are in that form, we need to collect them all up and to name value pairs and post them. In the case of a search, that will just contained the search term, but there could be additional data in there too. And then once we have the options together, it's time to make the asynchronous call. There's a number of different ways to make asynchronous calls back to the server, the jQuery. One is with $.ajax. That's the one that gives you the most flexibility in the most options. Here, I can just pass the options object in that will tell jQuery where to call the URL, and also the data to pass along, and whether to do or get or a post. And then when it is done, this is a call back function when that request is complete and successful. This function will be invoked and the response from the server will be in this data object. What I need to do inside of this call back is go out and find the target that is what is the DOM element on the page that I wanted to update with this data, so we will go out and find that by digging the identifier out of the data dash attribute. And then using the jQuery HTML method or rather what we could do is use the replace method, we could say replace that target with this hunk of HTML that we got back from the server. That will update the page. And then, one last touch inside of ajaxFormSubmit, we need to prevent the browser from doing its default action which is navigating away and going to the server by itself and redrawing the page. I can do that just by returning false down here. And with all these code in place, I should be able to do a build and run the application. Now, we'll be performing the same action as Ajax.BeginForm, but we'll be doing it with our code. So it's a little more open to customization and flexibility. I should be able to search and I get a JavaScript error which happens occasionally. I picked the wrong method to update screen with. Let's try replaceWith to replace this element with what is inside of this data. Save everything again, let's refresh the page. And scroll down a little bit and try the search again. And now, we're working just fine. And now, we have some code to build on that we can customize for our application using data dash attributes and we could this code, make a loading message visible or log errors into all sorts of interesting things that the application needs. ( Pause )

Autocompletion

Now that we have this nice asynchronous search feature, it would be nice if we could also help our users find a restaurant they are looking for by providing some suggestions when they start typing in the search box. In other words, if I type the letter S, I'd like to see a list of possible restaurants that start with the letter S as I'm typing. Fortunately, we have jQuery UI in the project and jQuery UI includes an autocomplete widget that will help us do this. The way autocomplete works is that when the user starts typing, we can configure autocomplete to make a request to the server and find possible matches. Of course, we'll be calling a controller action and the controller action will need to return JSON. That's one of the formats that autocomplete can work with. And then, once it has the list of possible restaurants, it will take care of the UI part and put that list on the screen. Since we're going to need some JSON data, let's start with a controller. And since we are on the Index view of the HomeController, I'll put this action that autocomplete will call in the HomeController. This action is called autocomplete, but you could call it quick search or restaurant search or anything that you want because ultimately, we'll tell autocomplete what URL to call and we'll have control over what action it we call. But I do expect this action to receive a parameter called term. How do I know that, because if you look on jQueryUI.com, they have lots of wonderful examples and demonstrations of how these widgets work. And the documentation for autocomplete says when the user is typing, it will send a request to the server and include a parameter in the request called term and that will represent what the user has typed so far. So I know if it sends something named term, I can get that as an action parameter. I just need to take that term and query the database with it. We'll say please give us the restaurants where the name of the restaurant starts with whatever term that autocomplete has given me. We will limit the result set to the first ten restaurants that we find and then we'll do a projection with the select operator. We'll turn every restaurant into an object that has a label property and that label property will be equal to the restaurant name. Why am I creating something with a label property? Because, again, the jQuery UI documentation for autocomplete says the JSON you return, the object should have a label property, or a value property, or a label and a value that will use any combination of that. Just giving it a label will be enough to have it put something on the screen and when the user selects it, it'll put it into the input box for us. All I need to do is to put that into JSON format which as we saw early in this course is very easy to do, just invoke the JSON method, it will serialize my model into JSON format. We're going to allow this to happen during a get request. And I should be able to test up this controller action from the browser. If I just do a build, I can come and go to that controller action from the browser and see the JSON that it's going to produce. We'll go to home, autocomplete and let's pass a search term of 1 along. And I can see the browser is telling me that some JSON came back the server. We'll open that up. And there I can see my JSON array. It looks like all the objects have label properties. Looks like all the objects have the label property set correctly. So I think this is going to work with autocomplete. Now, we just need to implement the client side of this. The way autocomplete works from the client site is you have to wire it up against the input that needs the autocomplete behavior. And the way I'll identify the inputs that need an autocomplete behavior is with, guess what, a data dash attribute, data-otf-autocomplete. And the value for this attribute will be the URL that autocomplete needs to talk to. So again, I'm using the URL helper to generate a URL that will point to the HomeController's autocomplete action with this and that value will be embedded in the data dash attribute. That's the only change I need to make in the view, everything else will be JavaScript. What I need to do using jQuery is go out and find those inputs that have that data dash attribute. So again, I'll use a selector here. Find me all the inputs that have data-otf-autocomplete. And for each one, call this function createAutocomplete. In this case, we're not really wiring up an event like we did with the form where we wired up the submitted end. In this case, we need to go up to each object and give it the autocomplete widget. So createAutocomplete will be a function that we need to write. And when jQuery invokes this function, it will pass along the input as the this parameter. So for each input that it finds with that data dash attribute, it will invoke this function, pass along that single input as the this reference. This code is just wrapping that input in jQuery using the dollar sign function so I can use the jQuery API against it. And very similar to last time, I'll also create an options object. Now, there's all sorts of options that you can specify for the autocomplete. You can tell it the minimum number of characters that the user has to type before it will send a request to the server. You can tell it how many milliseconds it has to wait until after the user has stopped typing before it sends off that request. But the one option that you must specify is the source option that tells the autocomplete widget where to get the data. So in this case, we'll just take the URL that was embedded in that data dash attribute. We'll pull it out using the attribute method of jQuery and put it in the source property on the options object. And then, all I need to do to wire up autocomplete is to walk up to the input, invoke the autocomplete method and pass in that options object. This is how a lot of jQuery plug-ins work. They extend jQuery by adding additional methods on the jQuery objects so you can just invoke them. jQuery UI adds methods to jQuery like autocomplete and dialog and button and tabs and accordion, all sorts of user interface type things. And at this point, I think I have everything correct where if I run the application, I should be able to see autocomplete work. So let's type in the letter S and there we can see the two restaurants that start with S. So this looks promising. Let's type in the number 11. There, I can see all sorts of restaurants that start with 11. And after I select one from the list, if I click Search, then it appears. So what I need to do is first type something, select it, it'll be placed into the input box for me by jQuery autocomplete and then I need to do a Search. So a lot of people look at that and say can't make it so that one someone selects one of these items, it just automatically submits that search for them so that the results are updated. And the answer is of course we can. Over here in the options that we passed to autocomplete, when it creates itself, we can pass in an additional option which is the function to invoke when the select even occurs. So when the user selects something, please call a function called submit autocomplete form. That is another function that we'll have to write. And once again, the this reference will be set up to point to the DOM element that we're interacting with. In this case, it's the input again. And I'll wrap that DOM element with jQuery so that we can do some interesting things with it. For instance, one of the things that we'll have to do is set its value because even though jQuery autocomplete will automatically populate the input with the item that the user has selected, sometimes that doesn't happen before this select event is raised. So you can get into submit autocomplete form and still have an old input value there, not the new one that the user selected. We'll circumvent that problem just by setting the input value ourselves. So this is the jQuery API to set the value of an input. We're going to set it to you ui.item.label. UI is a parameter that autocomplete passes in. This is something also that you'll find in the documentation. That UI object will have all sorts o interesting information about the state of autocomplete and the parameters of autocomplete including the label of the item that the user clicked on. So once the input is set, we just need to submit the form. But we have to find the form first. Fortunately, that's pretty easy with jQuery too. What I can do is say, dear input, go and look through the parents above you, all the DOM elements above you, and find a form. In fact, find the first form. That's just in case you're inside of a form that's inside of a form. That would seem a little bit weird, but we'll just take the first form that you find as you look up through your ancestors. And once I find the form, there's a jQuery API where I can tell the form to submit itself. That will raise the submit event and that will ultimately come back into our AJAX form submit function that we wrote earlier. And now, if I save this script, the only change I made was in script, so I should just be able to save the file, refresh the browser and see this new behavior. Let's try looking for 11 and selecting something. And there, I can see the form was automatically submitted for me. Try it again with the letter S. That seems to work too, and 11 too, all this seem to work. And the one thing that still bothers me is that it's hard to tell when the bottom half of the screen is updated. This is automatically updating for me now. I almost don't notice it. It would be nice if I could draw the user's attention to the bottom half of the screen somehow. This is another place where jQuery UI can help. It includes a number of effects that you can apply to DOM elements. Effects that make things pulsate or bounce or grow or shrink. They're very easy to use. Let me show you an example. Up here, when we take some HTML back from the server and we replace what's in the page for the restaurant list with that new data, we can make it highlight itself. I'll place some code in here that will take the HTML that's coming from the server and wrap it in jQuery so we can manipulate it with the jQuery UI. And then, after we have replaced the restaurant list with this new HTML, I'll call the jQuery UI Effect method and tell it to run the highlight effect. This is where you can pass the parameter to tell it if it should explode or pulsate or highlight. We'll choose highlight. Just, again, save the script. Let's refresh the page to get that new script in the browser. And then I'll search for 11. And there you can see, it's just a very brief flash. Hopefully, you can see it in the video. But it should be just enough to draw my attention to that area and reassure me that, yes, the results here actually have changed for me automatically. ( Pause )

Paging Results

Finally, there's one more feature I'd like to add. I'd like the ability to pageList lists of restaurants. The last search we did was to search for restaurants that start with a 20. And I can see 20 and 201, but it only goes to 208 and there is a restaurant 209 but I cannot see it here. There's no indication that there's more results. That's because we only take the first ten records from the database that we get. So let's have paging. To do the paging, I'm going to rely on a third party component and one of the best ways to install a third party component is by right clicking on the References folder and selecting Manage NuGet Packages. NuGet is a package manager for .NET. You can download software packages which might be JavaScript libraries or .NET assemblies or something that has both of those and more. Now, you don't have to find the project and download the zip and extract the files and add them to your project. All you need to do is come to that References and open up the NuGet Package manager. From here, I can see a list of all the packages that are installed in my application. That includes things like the EntityFramework and jQuery UI. I can also go online and I want to search for paged list. I'll search from the text box up here in the top right of the screen and that will find me PagedList.Mvc and PagedList. Two great projects by a fellow named Troy Goode. I'm going to tell NuGet to install PagedList.Mvc. NuGet understands package dependencies too and it can see that this package depends on PagedList, so it downloads them both and adds them to my project. I can click Close and come in to the assembly references and see PagedList and PagedList.Mvc. Now, I can start making changes to the application. We'll start with the HomeController. The index action of the HomeController will no longer be able to just return just ten records. Instead, we need to return the proper page of records. And what the PagedList package will give me is, first of all, an extension method that I can use in the query to get the proper page of results and also some HTML helpers that I can use in a view to display a pager widget, something with more and next links and page numbers in it that a user can click on to navigate around through a large result. The first thing we need to use though is the Extension method in a query. So let me add a using statement for PagedList. And now, I can come down to the index action and instead of having that take operator in there, we will end this query using a call ToPagedList. ToPagedList, I need to pass it the page number that I want. Where do I get that from? We'll take it as a parameter, because when the user interacts with the pager that we're going to produce, we'll make sure there's something in the request called page. It might not always be there though, so we'll have a default of 1. That way, if you come to this page and you don't specify what page number you want, you get the first page of results. And so, ToPagedList will get whatever page has been passed in. And also, we need to specify the page size. Let's just say ten restaurants per page is good. And these are all the changes that we need to make to the controller. Everything else is going to be in the UI. The first change we'll make in the UI is to tell it about the new model type because this is now a model that is of type IPagedList. That's going to allow me to use the pager helpers to build a paging control. Let's come in to the index action and instead of being a model that is IEnumerable of RestaurantListViewModel, it's now going to be a model of IPagedLisr of Restaurant View Model. Right now, I have a red squiggly here because razor doesn't understand what an IPagedLIst is. I could add a using statement in here to tell it what namespace that is in, or I can just add that as a default namespace for my entire application. In fact, I could add OdeToFood.Models also as a default namespace. This just cleans up the code and makes it a little easier to work with. So the way that you do this is by opening up the Web.config file that is in the Views folder. Not the Web.config file that is in the root of the application, but the Web.config file that's in the Views folder because it is this web config that controls the razor configuration. This is the list of default namespaces that razor uses. I want to add three additional namespaces, PagedList, PagedList.Mvc, and also my OdeToFood.Models. Now, quite often, when you change this section of Web.config, Visual Studio and that razor parser, they do not pickup the changes here until you restart Visual Studio. And because this always happens to me, I'm just going to go ahead and I've maid these changes and save everything and close Visual Studio. We will reopen it right away. And hopefully, all the red squigglies are gone from Index.cshtml. And they are, so we're in good shape. Now, from the Index view, we're also rendering a partial view with the same model. I need to come into this partial view and also change the model directive there to let me open up \_restaurants and we'll change this from in IEnumerable to also be an IPagedList. And now, we can get rid of the OdeToFood.Models namespace qualifier too. It is inside of here, in fact, inside of our restaurant list too and I will place a pager, a call to Html.PagedListPager that's provided by PagedList.Mvc. This will build the pager for me. All I need to do is give it the model. It will figure out how many total things are in there and also a lambda expression that given a page will be able to return a URL to go to that page. The PagedListPager will pass me the page. I'll use Url.Action to generate the URL to the HomeController Index action passing page along in the request. And we know now, that will get into the action and into that to PagedList call. There's all sorts of options that you can specify for the PagedList pager. I'm going to select MinimalWithItemCountText. That will just give me previous and next links and the total number of items that are in my model that you can play around with here with the different options. You can have individual page numbers. You can do all sorts of crazy things with the PagedList pager and make it work the way you want it to work. But this should be enough that if I run the application, we should have a pager on the screen now that tells me how many restaurants are in the database. And indeed, there's the pager with the next link and it tells me I'm seeing ten restaurants out a possible 1,003, that's the rate number. So this seems to be working. And I can click on Next to go to the next list of results. So this is all very good, but there are two problems. One is that the pager, it's not aesthetically pleasing at the moment. That's an easy fix. And also, you'll notice that we're redrawing the entire page when I click on the Next link. We're going off and just redrawing the whole page. I'd like that to be able to redraw just this portion of the page just like we do when we do a search on this page. So let's fix the easy one first and that is to improve the user interface here. One of the files that the PagedList package will install into my application is a PagedList.css file. So a style sheet to make that pager look a little bit better. I just need to make sure those styles are included with my styles, or put them in a bundle to download with my style sheet bundles and that's the path I'll take. This is the bundle that includes my site.css. I'll also just include PagedList.css along with that. And just doing a build and refreshing the application, we should see a better implementation of the pager. And indeed, that has helped quite a bit. Now, I have the Previous and Next links, and those still work. Now, let's try to make this asynchronous which will be a little bit more challenging. What I need to do, just like how previously we intercepted this form submission, when I click on this, we catch that event and send off the request ourselves and redraw the screen, now, I have to intercept click events on these anchor tags that are here. Intercept those requests, go off and get the page data myself and redraw the screen. To intercept those requests, of course, we'll still be using jQuery. So let's go back to our script file, but we'll do this a little bit differently. What I'm going to do is not go out and look for those anchor tags and wear out Dot Click event. That would be easy enough to do with jQuery. I'm not going to do that because we destroy those anchor tags every time we repaint the screen. We go off and get a fresh batch of HTML and then we put that HTML in the page and destroy the anchor tags that were there and replace them with new ones. I'd have to rewire the event every time we redrew the screen like that. Instead, what I'm going to do is walk up to something called main-content. main-content is in the Layout view. You can see it right here. A section with a class of main-content. I'm going to wire up a click even on this section because it contains all of the HTML that our view renders. And this section isn't destroyed. It's always going to be on the page. So this syntax is saying go out and find the element or elements with the class of main-content and wire up the Click even. This is using the On method with jQuery. What this allows you to do is wire up an event on a DOM element that's higher up in the tree and then specify how to filter those events. This is saying, yes, I want click events, but I only want click events that originated from an anchor tag that was inside of an element with a class of pagedList. This way, I don't pickup just any anchor tag that might be on the page. I'm picking up just the anchor tags that are inside of this div with a class of pagedList. Now, we're intercepting those events. We can call a method called getPage. getPage is a function that we'll write. And inside of getPage, this reference will point to the anchor tag that the user clicked on. We'll get a reference to that and wrap it with jQuery so that we can extract some values from it. We will extract values like the Href attribute that's on that anchor tag where it's pointing to, because that contains the information that we need to use to get to the correct page number. It will be something like /home/index query string page equals 5. That's the Href value that we need. We'll take that value. Put it into a URL property on this options object. Just like we did earlier with a form, we're going to build an options object that tells jQuery how to contact the server. This is the URL we want to go to. This is the type of request that we want to make and get request. And the rest of the code would look like this. Given that options object that we just built, go out and do $.ajax and make a request to get request to that URL. And when it's done and you have the new data, call this function. We'll go out and find the target, the thing that we need to update which will be restaurant list, but I'm trying to rate this code in a generic fashion so it could be used anywhere with any page list in the application. All you need to specify after the user has clicked on one of the pager links, what DOM element that you want to update. I put that here in a data dash attribute inside of the div with a class of pagedList. So we're just looking for this value to extract restaurant list. And once we have extracted restaurant list, we'll go out and select that element with jQuery and replace it with the fresh data from the server. Let's cross our fingers and run the application and see if we can get this to work. I'll scroll down a bit. That's always a good test to see if you're doing a partial page refresh, or refreshing the whole screen. And it looks like this is not working. I'm able to go to every page just by clicking Next, and we're not refreshing the entire screen. Does it work with the search? Let's search for 20 again. And there's my 20 results. If I go to Next, that seems to be broken and that's because when we go off to fetch the next page of result in here, we're not taking into account the search term that user has entered on the page. All we're passing along is the page number that we got out of the Href for the anchor tag. But this is actually a very easy problem to solve. What I can do inside of our AjaxOptions is say please add some additional data to this request. Essentially, go out and get the form that's on this page and serialize it and send those values along in the request too. And now, if I save the script, and let's do a hard refresh in Internet Explore. Now, let me search for 20 and page through the results, and there's only two pages of results and that looks good. Now, I may have a list of restaurants on the homepage that can search through and page through, and it all happens asynchronously to provide a nice slick user interface. ( Pause )

Summary

In this module, we added some AJAX features to our application using jQuery, jQuery UI, and our own custom JavaScript. We brought controller actions to return JSON and implemented paging and searching with async behavior to provide an updated page without disrupting the user experience. Finally, we saw quite a few data dash attributes in our HTML. We saw how we could work with those attributes from our script and also have the MVC framework uses those attributes to add AJAX functionality through AJAX helpers and also to support client validation.

Security and ASP.NET MVC 4

Introduction

Hi. This is Scott Allen. In this module is we'll look at the security features of ASP.NET MVC. In this module I want to show you how to work with users and roles and also how to use OpenID and OAuth so your customers can sign in to your website using a third party like Microsoft, Google, or Facebook. I also want to show you some of the attacks that you'll face if you deploy an application on a web server. So we'll look at cross-site request forgeries and how to make your application resist such an attack. We'll start by looking at authentication and see how we can use Windows Integrated Authentication for internet style apps and forms-based authentication for web applications. ( Pause )

Authentication

When you authenticate a user, you are verifying the identity of the user. You might need to know a user's identity because you're building an application that only specific users should access, like a payroll system. You cannot let just anyone poke around in the salary information. Or maybe you're building a website where users post their picture collections. You might want to know the user's identity just so you can track their photographs and only allow users to delete their own photographs and not the photographs of others. So the first step would be identifying the user and making sure you know who they are. There are three ways to do this with ASP.NET. The first mechanism is called Forms Authentication. With forms-based authentication, the website is responsible for providing a page with an input form where the user can enter their user name and enter their password. And when they click a log on button, something inside our application is responsible for making sure they entered the right password. We'll see how MVC provides a lot of the infrastructure support for Forms Authentication so it's not as hard as it sounds. Another way to authenticate a user is to use OpenID or OAuth. These are open standards for authentication and authorization respectively and using OpenID or OAuth means your users don't need to create or share a password with your site and you don't have to store or validate a user's password. Instead, you rely on a third party like Twitter or Microsoft to authenticate the user and then tell you who they are. We'll see this in a bit. The third approach is Windows Authentication. Windows Authentication is typically used for internet applications because it uses components and services provided by the Windows Operating System. Microsoft also refers to Windows Auth as "Integrated Auth", because once a user is logged in to a domain, Windows can automatically authenticate them to My Application if My Application uses Windows Auth. Although Windows Authentication does work across a variety of browsers and not just Internet Explorer, it's still best to use for an internet application that runs inside your company's Firewall and where all of your users are logged into a Windows domain and you have an active directory server somewhere. Your users can have a single sign on experience, because once they logged on to their desktops, they use that identity for your application as well as network shares and network printers and other internet apps. If you're building a public website then you probably want to avoid Windows Auth because users might not be on the same domain. In fact, they might not even be running on a device that supports Windows Authentication. And that's where Forms Authentication comes in, because any web browser can let a user sign into a site using Forms Authentication. It's highly customizable because you build the log on page and you control the log on experience. You can customize how the form looks and how strong of a password that you require. Forms Authentication relies on cookies by default. So once a user is signed in to an application, the runtime will store a cookie in the client's browser so they don't need to sign in again during the same browsing session. One word of warning with Forms Authentication, is that SSL is required to make Forms Authentication secure. Unless I'm logging into a site using an HTTPS address instead of regular HTTP, then I'm passing in my user name and password across the network in plain text and someone who is watching network traffic can see my password and steal it. OpenID and OAuth are also designed to work with internet applications. When you need to validate a user's identity, you redirect their browser to a third party website, like a website run by Google or Microsoft, and the user proves their identity to this third party. Then the third party will redirect the user back to you with a message that proves they successfully authenticated themselves. This all works securely, thanks to the magic of cryptography. We'll see how to work with these three different approaches to authentication in this module. And we'll get started by looking at Windows Authentication. ( Pause )

Windows Authentication

The OdeToFood application we've been working on was created with the internet application project template. You might remember that from the first module. Internet applications use Forms-Based Authentication by default, but I want to demonstrate Windows Authentication, and changing this application over to use Windows Authentication is a bit messy. It's possible but it's messy. So instead of getting messy I'll launch a second instance of Visual Studio and we'll create a new temporary project just to look at how Windows Authentication works. So a new project using MVC 4, I don't really care about the name because we won't be spending anytime with this. So we don't need a unit test project either. I just want to make sure instead of creating an internet application, I create an intranet application and then I'll click okay and be back in a second once Visual Studio has the setup. Now the project is ready to go and Visual Studio will open up a readme.text file by default when you create an intranet application. It'll list a couple manual steps that you have to go through in order for this to work because many web servers today ship with features turned off by default. And one of the features that is off by default is the ability to do Windows Authentication. To fix that, I need to come into to the MvcApplication and open up the property window. And I do that by going to View, Properties Window. In this case I cannot right-click on the Project and go to Properties. That takes me to the wrong place. I need to open up the Properties Window by clicking this or pressing F4. And that allows me to set some properties on the development server. Remember the development server is IIS Express. That's a little application running down here in the system tray. We've hardly looked at it at all. It just sits there and runs in the background to process our request. We need to come in for IIS Express and tell it to enable Windows Authentication. And that's all the configuration we really need right now. Everything in an intranet application requires authentication by default. So it will not let a user in until it knows who they are. And that means if I run the application, the first thing Internet Explorer will do is pop up a log in box. And you might wonder, well, what's so integrated about Windows Authentication if Internet Explorer is always going to ask me to log in? But this behavior is peculiar to Internet Explorer because Internet Explorer will only automatically log you into a site if it thinks that site is on your local network. And by default, Internet Explorer doesn't consider local host to be on the local network. If I want to change that experience I would have to go into the Tools menu in IE, so Alt T will open up the menu, and then I can go to Internet Options. Go to the Security tab. Click on Local Internet and then go to sites. Go to Advanced, and finally I reach a dialog where I can tell Internet Explorer that local host should be considered an intranet site. And the reason this is important is because if we look at the custom settings for a local internet, essentially what are all the settings that are applied in this soon, and I explore all the way down, you'll see right here that Internet Explorer will only automatically log on for a website that's in the intranet zone. And it doesn't matter if you've created an internet site or an internet site with MVC, this is completely based on IE's heuristics on determining what's an internet site and what's not an internet site. But now, local host should be considered in the internet zone and if I close all these dialog boxes, and close the browser and press Ctrl F5 to run this again, Internet Explorer will automatically log me in using my current credentials for this machine or my domain if I have a domain controller. The machine is WIN8VPC, my user name is Scott and that can automatically display it because I've been authenticated, even here on the Homepage. And no log in box was required this time. Where does this text come from? That's in the layout view for this application. If we go to Views, Shared, Layout, we'll find there's a section in here using User.Identity.Name. So the user property and User.Identity, they're both available for Windows Authentication and also Forms-based Authentication. You can do things like find out the user's name. You can also use identity to find out if the user has been authenticated or not and how they were authenticated. And we'll see User.Identity again when we come back to Forms-Based Authentication. But for now, this is how Windows Authentication works. And the one other primary difference between this application and the OdeToFood application, if I come into web.config, inside of here there will be a section called "Authentication", and you can see the mode is equal to Windows. Inside the OdeToFood application, we have that same section, authentication section, but the mode is equal to forms. And that's essentially what's telling ASP.NET if it's using Windows or Forms-Based Authentication. ( Pause )

Forms Authentication

Now we can look at Forms Authentication in the application we've been building, OdeToFood. But before we get into the code, I feel we should take a step back and look at the big picture of how Forms Authentication works in general and how ASP.NET implements Forms Authentication in MVC 4. First, imagine the user tries to go to a members only area of the site, an area that requires user authentication. We'll see how to enforce this requirement. But MVC will know when you require authentication. And when it sees an anonymous user trying to get to such a place, it will redirect the user's browser to a log in page. You can configure where this log in page lives and in an MVC application this will not be a page of course, it will be a controller action. But to the user it's a page and it's a page where they enter the user name and password. When ASP.NET does this redirect, it will also save the URL, the user originally wanted to access. So if they successfully authenticate, ASP.NET will automatically redirect them back to where they wanted to go. It does this by storing the original URL in the query string with the name ReturnUrl and it will use that value. So once a user logs in, they don't have to click around the site trying to find that members only area again. They'll be redirected right back to it. If the user cannot get logged in, they'll never get to the members area. But if they do give the user name and password, ASP.NET will redirect them back and they'll get to see the content that you're protecting. ASP.NET also gives the browser a cookie for the application. A cookie is like a piece of data that the browser will hold on to. And once a browser holds a cookie for a site, it will continue to send that cookie along in every request that it makes to the site or at least as long as the browsing session lasts, possibly longer if you and the user allow it. Inside of that cookie is some encrypted information letting ASP.NET know that the user has already authenticated themselves and this means, as the user is browsing through that members only area or other sections of the application that require authentication, ASP.NET will see that cookie, know the user has been authenticated, it's not an anonymous user, and ASP.NET will allow them to succeed with that request. It won't force them to reauthenticate or sign in again. What I'm describing about Forms Authentication isn't specific to ASP.NET, many web frameworks take a similar approach to Forms Authentication. They use log in pages, they use cookies, just the names of the pages and the names of the cookies and query string values change. Let's talk about how this works in ASP.NET MVC 4 specifically. First, when you create a new application using the internet project template, the project template will include everything you need to register a user, allow them to log in, allow them to change their password, all the basics that you need for Forms Authentication. All these basics are provided by an AccountController, you'll find that in your Controllers folder as well as a set of views in the Views Account folder and there're also some models. You'll find them in the Models folder. If you peek at the code inside the AccountController, you'll see it's using a class named WebSecurity. And WebSecurity is from a Microsoft library named WebMatrix. Perhaps, you've heard of WebMatrix. It's another tool from Microsoft, you can use to create websites and ASP.NET MVC is borrowing some of that technology from WebMatrix to implement Forms Authentication. It's the WebSecurity class you can talk to from your code in the AccountController to tell it to validate a user's password. WebSecurity in turn talks to a component known as this SimpleMembershipProvider. And although there are more layers that what I'm showing here. Ultimately, you can think of the membership provider as doing the hard work of storing information in the database. Earlier I mentioned that Forms Authentication is customizable. And you can certainly go into the AccountController and the AccountController views and make as many changes as you'd like to have the log in form look different. You can also customize the application by taking control over the underlying SQL Storage. You can customize what you want to store about a user. For instance, you could customize the Register screen by asking the user to name their favorite restaurant, and then store that in a table in SQLServer along with the rest of the user information that you need. I'll show you how all this works together and some steps you want to make to customize the application. ( Pause )

Taking Control of Membership

Before we get in to registering and logging in and seeing how to require authentication in the application, let me show you a few steps you can take to have more control over what's happening behind the scenes, particularly at the database layer and particularly if you want to customize the information that's being stored about a user. The first step I would recommend is coming into the project and finding the Filters folder. Inside of here you'll find the file, Initialize SimpleMembershipAttribute. If we open this up, there's a lot of complex code in here because when you start a new MVC 4 application with this project template that we used, the internet project template, it's not 100 percent sure that you want to use Forms Authentication. So there's a lot of complex code here to make sure that Forms Authentication is initialized in a lazy manner. Perhaps you're not even going to use a database in this website. And in that case, none of this code would have to execute. But if you've already decided that you are going to use Forms Authentication, there's no reason to have all of this code. What you really want to get to is the essence inside of here which is WebSecurity dot InitializeDatabaseConnection. This is a call to initialize all the membership infrastructure and tell it, a name of the connection string that you're going to use to connect to the database, which is DefaultConnection, the same database we're using to store a restaurant information is also specifies the name of the table that will contain user information, its UserProfile, a column that contains the primary key value for a user, so it can look a user up given their ID value and also the column that contains the UserName. You can have as many other columns in this table as you want when you store user profiles, but you at least need to give it a primary key and the user name. If we know we're using Forms Authentication, we might as well take this line of code and cut it out of here, Ctrl X, and run it during application startup. So I will come into Global.asax.cs and inside the application start method I will paste that code inside of here. We'll need to bring in a namespace, so control period and select that first entry, this WebSecurity class is in the WebMatrix.WebDataNameSpace. Now that namespace is added and we have legal C# code. If you want to be consistent, you could wrap up this call into some static method on a class so everything looks like something .Register. But right now we'll keep things simple and just leave initialize database connection right here in the application start. Now that that is here, there's no use to have this filter. So we'll come into this filter and delete it. And yes, the next time you build you'll get compiler errors. But we're going to fix this. Before I fix this, there's one more step I want to take. Inside of the Models folder, you will find an AccountModel.cs file. Inside of here, there's a class that derives from DbContext, you recognize that now as the entity framework DbContext. This is giving the AccountController access to that user's table through a DbContext derived class. But we want to control this. We want to have this as part of our DbContext. So I'm just going to delete this DbContext. We don't need it. We do need something to represent a user profile. So I might as well start with this class. It has a UserID and a UserName. I could add additional properties here. For instance if wanted to know a user's favorite restaurant then I can include that as another property in the user profile. But just so I know that this is my code, I want to own this code, it's not part of the generated code that came from this template really anymore. I've customize it. I'm going to cut it out of here and add it to its own class file. So let's add a class called UserProfile and I will delete what comes out of here by default and just paste in that UserProfile that we just had. I'll need to include a couple of namespaces here, too. So Ctrl period and bring in System.Component Model.DataAnnotations.Schema. This attribute is one way to tell the entity framework that when you need to store user profile information, when you need to store objects of this type, put it in a table with this name, UserProfile. We also need to bring in a namespace for the key attribute that's in the System.Component Model.DataAnnotations. We've used that before when we added some model validations. This particular attribute is a way to tell the entity framework that this is the primary key value for a user. And by the way, it's also an identity column, meaning the database will manage the values that get placed into here when you insert a record. It'll automatically be generated by SQL Server. And now that we have our own user profile class that can store our favorite restaurant, I need to define a way to get to that in my own DbContext class. So I will open this one up, OdeToFoodDb and let's add a property of type DbSet of UserProfile and we could call this UserProfiles. And now if you've been watching the previous modules, you know we've made a change to what we're storing in the database. We're going to need to do a schema migration. But before we can do that we have to get the project to build now that we've deleted some classes. Let me do a quick build and that would give me a list to work from. So first of all, if I double-click one of these errors, this Initialize SimpleMembershipAttribute doesn't exist any longer. That's okay. We're initializing things explicitly during application start, so I can just delete that. Then I'll double click another error. This one is complaining about UsersContext, we deleted that. That was what the AccountController was using to get to that UserProfile table. Now I can get to it through OdeToFoodDb. So let me just change this to use the var keyword and instead of a new UsersContext we'll instantiate a new OdeToFoodDb. Let me try another build, Shift Ctrl B. Now we've successfully built. Let's quickly migrate the database so we can store our custom information. I'll open up the Package Manager Console. We have implicit migrations enabled, so I should just be able to say, update database and I'll throw in the Verbose flag just so we can see exactly what it's going to do. And it finished the migrations. It's running the Seed method. But you can see where it went out and created a table, UserProfile, that has UserID, UserName, and FavoriteRestaurant. We can probably tweak the annotations on UserProfile so that User Name was an nvarchar max. It would be much better for performance if that had a reasonable size like nvarchar 255 or nvarchar 80. It will be simple enough to do that just using a string length attribute on UserName, and the same for FavoriteRestaurant. But at this point, we're good. We have more control over what's going to be happening in the database. And now, we can turn our attention to looking at how to register, how to log in, how to require authentication. ( Pause )

Forms Authentication In Action

Let's see how Forms Authentication works inside of the application now. As a user, I can come in and click the Register link. And this will give me a page where I can fill out the bare amount of information to create an account, just a User name, a Password, and then confirm my password. I can click Register and I have an account and I'm automatically logged into the site. My name appears at the top of the page and that's done using User.Identity.Name, just like we saw with Windows Authentication in the Layout view. At this point, I could click on my name to manage my account and change my password. I could also log off of the site and then come in and log in again. And all of this functionality is provided out of the box when we create a new internet application. The register page for example is given to us by adding a view to the account folder called Register.cshtml. And it's inside of here where you can see textboxes to input the user name and password boxes to input the password and confirm the password. This is one of the places you'd have to come in and modify if you wanted to force a user to provide you additional information when they register like the name of a favorite restaurant. And this view is rendered from an AccountController which we were also given. Of course that's in the Controllers folder. And it includes methods like Register. And based on what you know about the MVC framework so far, you can probably figure out what most of the code inside of here is doing. We have a register action that responds to an HTTP GET request that gives the user a form to fill out their information. And we have a register action that response to an HTTP POST request and takes information from the user to create a record in the database. We also have ModelState and we have a RegisterModel which is like a view model containing only the information about a user that we need in the view to get them registered. These are all concepts we've looked at so far. So let's focus on what we haven't seen which includes the call to WebSecurity. WebSecurity is basically a wrapper around membership functionality. It takes care of data access, cryptography, and all the other code that you need for Forms-Based Authentication. Here we can see when I successfully register, there's a call to CreateUserAndAccount, that's what actually puts me into the database. So what database do we go to? Well, if you remember, we placed a call to WebSecurity.InitializeDatabase in the application start event. And we passed the parameter specifying the database connection name to use which was DefaultConnection. The same connection string that we used for storing restaurant information and that means if I go to the Database Explorer, and look at our DefaultConnection and open this up to see the tables, I will see not only Restaurants and RestaurantReviews which we had before, but also the UserProfile table which we've just added in this module, that's a table that we're in control of. And then, there's a number of tables that are used by WebSecurity, they all have a webpages\_prefix. And if we open up some of these tables to poke around, you can see the UserProfile table has an ID, UserName, and FavoriteRestaurant columns. That's exactly what would we expect based on our definition for UserProfile. And if I right-click on this and select Show Data, we'll see there's exactly one record in here, the sallen user that I just created has a UserID but no FavoriteRestaurant because we didn't forced anyone to fill that information out. We don't have a place yet to fill that information out. And now if I look into webpages\_Membership, what I'll find is it there is a corresponding record in here that has the same UserID so we can join these two tables together to get the whole picture for a user. And this one includes information like a hash password for sallen. WebSecurity will automatically hash password which is good because then we're not storing the user's password in plain text in the database. That's something you almost never, ever, ever want to do, store plain text password in your database. And there are some additional columns in here, too. But overall, the database is now a combination of tables that we used to manage restaurants and reviews and user profiles, and tables used by WebSecurity and its friends to manage membership and roles. And this database was created through a combination of WebSecurity and our entity framework migrations. And now that we know where the data is stored, let's go back to the AccountController and take a look at the second WebSecurity call, the call to log in. You'll also see this in the log in action which is where we come after the user has clicked the log on link and they filled in the user name and their password, they click Log in, we'll end up in this action that simply needs to take that user name and password and pass it to WebSecurity for validation. If the model says it is valid and WebSecurity says that we can log in, what we'll do is redirect back to the ReturnUrl. We'll see how that works in just a second. You can think about what WebSecurity.Login has to do. It has to, first of all, compare the hashed version of this password against the hash that's in the database and see if they match. And if they do, they can issue that cookie that we talked about. Let's actually take a look and see if we can see that. Here in Internet Explorer, on the log in page, let me press F12 to open the Developer Tools and let's go to the Network tab and click Start Capturing. This will capture network request. And I'll now try to log in with my user name and password, I can click Remember Me if I want the cookie to stick around between browsing sessions and click log in. ( Pause ) And here we can see what happened. I posted my user name and password to /Account/Login. It determine that I logged in successfully so it returned a 302 result which is a redirect, which says, "Dear web browser, please go somewhere else." So it redirected me back to that root of the application, the homepage as a logged in user. And if we open up this response from the web server and go to the Response headers, then what I want you to pay attention to is the Set-Cookie header. This is how you tell a browser to accept a cookie. The name of this cookie is .ASPXAUTH. When decrypted will tell ASP.NET MVC that this user has successfully authenticated to the application, there's no need to make them sign in again and now the browser is going to send that cookie along on every subsequent request. We can see that if we go back to the summary view and look at this request that comes back into the homepage. If I double-click on that, and now we look at the Request headers, so what the browser sends to the server when it's requesting this page. Down here in the Cookie section, if I double-click that, you can see that we send along the ASPXAUTH cookie. And that will actually go out on every subsequent request now until I close the browser or actually it should stick around longer because I clicked on the Remember Me check box. But the other thing I want to show you here if we go back to the summary view and look at Account Logon, double-click on that to go into the details and look at the request body. If I scroll over here, you can also see my UserName and my Password. And so can anyone who might be sniffing network traffic between me and the web server. So this is obviously not secure and this is why pretty much any reputable website that needs to force its users to log in is going to use SSL. They'll acquire an SSL certificate and install it on the web server and then force the log in process to go through HTTPS. If you go down that path, that's always a good idea to come into methods like this log in method and add an attribute RequireHttps. This is an action filter. We've talked about action filters in this course but this is an action filter that will make sure that the request that is arriving is coming over a secure encrypted connection. And that would make sure that we're not passing any passwords in clear text. No one should be able to see those traveling over the network. And now that we know a little bit about how Forms Authentication works, let's put on our developer hat and see how we can apply it to force users to authenticate before they reach specific sections of the application. ( Pause )

Authorize

Now let's look at forcing a user to authenticate. First, I'm going to remove the RequireHttps attribute that we put on the log on action because I do not have an SSL certificate set up. And instead, we'll talk about a different attribute, the Authorize attribute. Let's imagine that for some reason, you have some secret members only information inside of the About page for the application. In that case, what I would want to do is use an Authorize attribute here. Authorize sounds like it's more about authorization than authentication. But when you use the Authorize attribute like this with no other parameters, essentially, you're specifying the type of authorization that says, only authenticated users are authorize to invoke this controller action. So before we get into the About action, a user has to be logged in. And just by having that attribute there, if I do a build and we come back out to the application, let me go to the homepage and close the Developer Tools. And from the homepage I'm currently logged in. So I'll also log off and now try to go to the About page. And ASP.NET MVC has detected that Authorize attribute. It knows I have to be at least logged in into this application before it can view that. So it redirects me to the log in page where I can log in. And notice that the ReturnUrl includes /Home/About, it's been URL encoded with percent 2f. But it's /Home/About is the ReturnUrl, that way once I log in MVC will be able to redirect me back to where I was trying to get to which is the About page. Now if you're wondering how MVC knew to send me to /Account/Login when it detected that I needed to authenticate, that is something that you can configure in an application. If you look at the root web.config in the project, at the authentication section, the mode is equal to forms, and we have a loginUrl specified in the forms element underneath that authentication element. So the loginUrl tells the MVC framework where to go to authenticate a user. The timeout value here specifies how long the user will be authenticated after logging in. And this number is specified in minutes, so 2,880 minutes. But coming back to our Authorize attribute, like many of these attributes, you can apply them at the action level. You can also take something like the Authorize attribute and apply it at the controller level. So I'll remove it from the About action and let's just apply it to the entire HomeController. Now the user would need to be authenticated to do anything with the HomeController. If there were just a couple of things that you wanted a user to be able to do, an anonymous user to be able to do, there is an AllowAnonymous attribute. I can place that-- let's say on the index action. That will at least allow a user to get to the homepage of the application without logging in. So let's try this real quick. Authorize is at the controller level. AllowAnonymous is on the index action. So if I log off then I come back out here to the homepage and it's viewing the homepage just fine. And if I try to go to About or Contact, now I need to log in. But before I log in, let me show you a couple of other things that you can do with Authorize. You can be a little more specific about who is allowed into the HomeController. You can specify users. So I could say Users equals sallen, plall. So you can have a comma separated list of users. And having that in place, if I login as sallen, I should be able to get somewhere. So sallen and my password, and now I can get to the Contact page. But it's not very frequently that you can specify specific users except for perhaps an admin user or a super user. It's more common often to include roles instead of specific users. And then you could say roles equal something like administrators and sales. These users and roles parameters, by the way, I should point out that they also work with Windows Authentication. When you specify users, you'd be specifying users in active directory. When you specify roles, you'd be specifying groups in Windows. But now that I've added roles to my Authorize attribute, I've hit a bit of a stumbling block because I do not have any roles defined. We saw in the Database Explorer that there is a table where you can store roles, but I don't have any user interface available for me to create roles. We'll look at a way to populate some of these membership tables next. ( Pause )

Seeding Membership

What I'd like to do now is show you how to automatically populate your database with some users and roles. Having a database with users and roles predefined is nice to have during development because if you're building an application that restricts access to specific pages based on the user's role membership then needing to set up those roles when you start with a fresh project only slows you down. I'm a big believer in making it easy to get started working on a project. There're several techniques that you could use to populate the database. You could, just write SQL statements that insert data into the membership tables we looked at earlier. But I want to show you how to do this with the membership API that is given to us by the SimpleMembershipProvider. Remember that's one of the components that working behind the layers of other components in our MVC 4 application. And since this provider owns the schema, I think it would be more robust to populate the database using this provider APIs. The question then is when and where to populate these roles that we need? We could during the application start up, make sure that the roles exist in the database. But since we're using entity framework migrations, we can also use the Seed method of our entity framework migration class. Remember, that's in the Migrations folder. It's called Configuration.cs. This is where we added a thousand restaurants into the database so we could do some testing in our AJAX module. So inside of the Seed method, I'll add a call to a new method that we're going to write called SeedMembership. I can use Visual Studio to create this method for me if I just hit Ctrl period and select that option in the drop down. That will generate a method stub for me down here that I can now fill in. The first thing we'll do here is use a call to WebSecurity dot InitializeDatabaseConnection just to make sure that everything is set up and the schema is in place for the SimpleMembershipProvider. I do need to bring in a namespace for this, it's WebMatrix.WebData. We're using the same parameters that we're using during application start up when we call this method. It probably would be a good idea to take this two lines of code and put them in a static method in a class somewhere so that I can call them from both SeedMembership and also the Global.asax.cs application start event and not have this hard coded string (inaudible) duplicated everywhere. But I'll leave that as an exercise for the viewer. So once the database is set up, I can go out and get access to the current role provider and the current membership provider by walking up to a property called Provider on the roles class and the membership class. These are static properties. They give you the current provider that's in effect. I do need to bring a namespace for this to work, System.Web.Security. But now I have access to two objects that I can use to create users or see if a user exists, see what roles the user is in, all the good things that you need to do with membership. So there's not just a single membership provider that does both users and roles. Those are actually a little more fine-grained than that. There's a role provider and there's a membership provider. I just think of them both really as the SimpleMembershipProvider. And here is the API that we can use with them. We can walk up to the role provider and say, "Is there a role called admin., if not create the role." And then, "Is there a user with user name sallen, if not create that user and here's the password." And finally, check to see if sallen is in the admin role, and if not, add that sallen user to the admin role. We always want to check if things exist first before we just add them because remember the Seed method runs every time that we execute update database in the Package Manager Console. We'll be doing that many times as we change the schema and change our models through development, and we wouldn't want this Seed membership method to try to insert duplicate data that will actually generate an exception. But with that code in placed, let me come in to the Package Manager Console and run update database, that will execute the Seed method and it will have a disappointing result. And the disappointing result is an exception that has been thrown saying that the role manager feature has not been enabled. It took me a bit of time and debugging to figure this out. But I finally realized that I needed to add some explicit configuration to the web.config file for this to work. So the SimpleMembership stuff just works inside of your MVC application. That's a real web application. But here when we're running update database, part of the entity framework migrations. This isn't a web application and it seems to require a little bit of configuration to get this to work. So the configuration goes into web.config because this is the configuration that migrations will use. It has all the settings that are in place when that migrations process executes. And the bit of configuration I have to put in here is just to explicitly configure a role manager which is the simple role provider and set it to enabled equals true. And also explicitly configure the SimpleMembershipProvider as the membership provider. And just having that bit of config in placed and saving the file, I should be able to update database again. And now, we'll have users and roles in the database for sure. So that worked. And don't take my word for it that it just works behind the scenes. Let's actually test it out. I'm going to remove the Authorize attribute from the HomeController and we'll place it somewhere where we might actually want some authorization. So I'll remove that attribute and while I'm at it, I'll remove this AllowAnonymous attribute from the index method. It makes no sense if nothing in here is-- needs authorization. And instead, let's pretend that we only want to allow administrators to be able to create restaurants. That means making a change in the restaurant controller. Specifically, we'll add an Authorize attribute to the create action and say that this can only be done by users in the admin role. And putting this on the create action that responds to a post, we'll make sure that no one can create a restaurant unless they're an admin. But I will also put it on the create action that responds to a get request, just to make sure that someone who is not authorized doesn't come in and try to create restaurant and fill out all the information and they don't have the proper permissions that would be a disappointment. Speaking of which, it might also make sense to go into the Index view for restaurant and here with the Create New restaurant link appears, we might want to make sure that we only display this link if the user is an admin. Again, that mean standard users won't click that and get confused when they have to login. So there's an API for this, just User.IsInRole, the same user property where we can see User.Identity.Name. There's also an IsInRole check, you just pass in the name of the role and that will return false if the user is not an admin and we won't show that link. Let's try it out. I'll run the application and we'll come in to look at the list of restaurant. And since I'm not logged in, I do not see it create new link. So let me log in and now my database has an admin role. We've seeded that into the database. And I put sallen in that role. So once I logged in and come in to the list of restaurants. Now, I can see the Create New button. And if I were to go directly to this Create action as a regular user, so I'll log off and paste that URL in here, the MVC framework would decide that it needs to authenticate me in order to see what role I'm in and that's where I could log in. And I can now get to that Create page. And I feel like I can sleep well at night now, knowing that only administrators can create restaurants in the application. My application is error tight and a malicious user should never be able to get through this authorization checks. Or can they? ( Pause )

A cross site request forgery

One type of attack, you will potentially face in a web application is a Cross-Site Request Forgery. I want to make you aware of this attack because it's subtle and potentially dangerous and it's also easy to prevent. A Cross-Site Request Forgery or CSRF is an attack a malicious user can execute using unsuspecting users who are logged in to your site. A CSRF attack might do something a user will find annoying like log them out of the site or change the profile name. But this attack can also be destructive and steal or destroy data. Let me give you an example. Let's imagine I'm a bad person who wants to create a restaurant on OdeToFood, but I don't have administrator access. So what I will do is go out and create my own web application. I'll start a new wen application here with Visual Studio, again, it'll just be a throw away application. I'll create an MVC 4 application but it doesn't need to be an MVC application. It could just be an HTML page that I upload somewhere on my website. I'll use an empty project template. And now that that's ready, all I'm really trying to do here is build an application that I can run in IIS Express and point out that it's a different application than OdeToFood, so it's someone else's website. And then inside of here, I will add a new item, just a regular HTML page that can be served up from my site, let me call it, funnypictures.html. And my goal here is to execute a CSRF attack against someone who is logged in to OdeToFood. In order to do it, I usually need to know a little bit about what HTML is on the site that I want to attack, but that's usually not too hard to get, and the HTML that I'm particularly interested in is the form that is used to create a new restaurant. So let me copy it out of the source view of this page and paste that into my HTML page. And then just clean it up a bit because we really only need a form tag and the inputs for name, the city, and the country, and hit control K, control D to do some formatting, that looks better. And now I'll add the values that I want to get into OdeToFood through an unsuspecting user, my values and this could represent trying to transfer money or change someone's profile, anything on a site that only an authenticated user can get to, that's what I want to get to and place my own values into the application. And I'll do just a couple of other things to dress this up. Let's give the form an ID. I'll just call it the form. Let me add in inline style. So nothing inside this form actually displays to the user. The goal is not to have the user fill this out, the goal is for me to trick the user into taking this HTML and submitting it to OdeToFood application behind their backs. And that means the action, we're not going to be posting back to my website. My website doesn't have anything to do with restaurants. It's just full of malicious pages. I'm going to change that to postback to the real website which is over here on a different port number on this machine. And I don't even need the user to click a button because I'm going to add some script here that will just automatically submit this form after a 2,000 millisecond delay. So what I've done essentially is craft a form to look just like the form that OdeToFood expects. And now if I run this application or just view this in a browser, as me the bad person, the malicious user that form gets submitted to OdeToFood which redirects me to the login page because I need to login but unfortunately I'm not an administrator. However, if I take this URL and somehow get it to another user who is an administrator, maybe I'll send a link in the e-mail, use some social engineering, get them to click on something else that's on another page, thinking they're going to download a free song or something. If I get that person to do it, someone who is logged in to OdeToFood all they need to do is click a link and come to this page and then I get a value into the database. It will be down here at the bottom. And of course there's all sorts of things I could do to make this less obvious to the user that they just posted something to OdeToFood. I could use jQuery to do the submission in the background so they never even see the OdetoFood website. All they see is that page of line where I put a funny picture on there. Behind the scene something is happening on OdeToFood that they don't know about. And me the bad person has used someone who is an administrator on OdeToFood as a proxy and channeled my malicious data through them to get it into the database. That's a Cross Site Request Forgery. It works because that other user is logged in to the application. And because they're logged in to the site, their browser has a cookie. And because their browser has a cookie, the browser sends that cookie along on every request that it makes to OdeToFood even if the user didn't intent for that request to go to OdeToFood, even if me, the malicious person tricked them into sending a request to OdeToFood that carried along something dangerous. So when you are building an application that uses forms authentication and you have areas with form posts that you want to protect with authorization rules, then it's not enough to just authenticate a user, and authorize them based on their role. You have to make sure that the form they're submitting to you is a form that you gave them and not a form that a malicious user tricked them into sending. It should be your restaurant create form not someone else's form from another website. And fortunately, this is easy to do with ASP.NET MVC. It has the concept of an anti-forgery token to ensure that the form post that is coming in when you create a restaurant is something that you gave the user. The way we use an anti-forgery token is to go to the restaurant controller and on this create action, the destination for a post put an attribute to say validate anti-forgery token. This is part of the solution. And just having that attribute in place, if I build the application come back out as a user who is logged in to the site and try to create something new with any name, any city, any country, I'll get an exemption that there was no request verification token that I submitted. So the way this anti-forgery token works is that we need to go into the form that will post this controller action and use an HTML helper that will add a verification token into the form. The verification token is a hidden input that holds a cryptographically significant value. So that is basically, added to every form that the user is going to submit into an authorized area. That cryptographically significant value is also placed into a cookie in the user's browser. So when the user submits the form, the form value has to match the cookie value before MVC will allow that request through. The idea is that even if a malicious user gives one of your members some evil HTML, they will not be able to set that cookie in your user's browser because browsers don't allow one site to set cookies for a different site and that is how an anti-forgery token will prevent a CSRF attack. So the evil application that I was building over here, it wouldn't be enough just to have these inputs anymore. I would also need to figure out what the user's anti-forgery token is and set a cookie in their browser, that's just impossible for me to do, so I would give up. But we need to get this working in the real application and I have validate anti-forgery token that attribute is on my create action. I also need to go into the create view for a restaurant and for this to work inside of Form, I'll just use Html.AntiForgeryToken. That is all that it's needed and now if I come into the browser let me refresh this page. And I should be able to create a restaurant with a name of two and a city, and a country of two just to see if this works. And down here at the bottom on my list I can see now, I can successfully add that. So one more quick summary if you're building something that uses forms authentication, you have form post coming in to authorized areas add an anti-forgery token to the form and validate that anti-forgery token in the action that is a destination for the post. Taking these two steps will help to prevent a Cross Site Request Forgery.

OpenID and OAuth

Summary

ASP.NET MVC 4 Infrastructure

Introduction

Hi, this is Scott Allen and in this module I'll show you how to use some of the infrastructure features of ASP.net inside an MVC application. In this module, I'll show you how to use the underlying ASP.net caching engine to improve the performance of an application. We'll also see how to work with ASP.net resource files to globalize and localize some application to support multiple cultures. Finally I'll show you some techniques you can use to enable diagnostics and logging inside of an MVC application. ( Silence )

Caching

Output caching allows you to store the output of a particular controller action in memory. ASP.net can then respond to future requests for the same action just by giving back the cached result and that avoids executing any of the code inside the action. Because you're executing less code and doing less work, caching can lead to some pretty dramatic performance gains but you do have to be able to careful when using the cache to avoid erroneous responses. I'll show you such a scenario in our demo. Caching itself is easy to enable using the output cache action filter. You can cache actions that produce few results and jayson (phonetic) results or really any type of data oriented result. You'd primarily want to cache the actions that get called the most or the ones that are expensive because they execute a slow database query. Those are some of those places where you would get your biggest win with the output cache attribute. The attribute will also work on child (phonetic) actions as we'll see and there's a variety of options you can specify on the attribute itself like the duration that the cache items should leave in memory, that's specified in seconds and what parameters should bury the cache. This is best understood through an example. ( Silence ) Inside the home controller for the application, let's add an output cache attribute here. We'll add it to the index action. The index action represents the home page of the application and chances are it might get the most traffic of the entire site. And I say might because I don't really know. One of the tips to implementing a successful caching strategy is you really do have to know where your traffic is going. You have to know what are the most expensive operations in your software. You have to take some measurements and do some logging in order for caching to really work effectively but this is just a demonstration of how the output cache attribute works. So I'll place the attribute there. Set a break point at the beginning of this action and then run with the debugger. ( Silence ) And what we should see here is on this first request, we will hit that break point so we're inside of the index action and you'll notice the one parameter I have to specify with the output cache attribute is the duration. How long do I cache the response? It's specified in seconds and this is 60 seconds. So now if I press F5 to continue with the debugger, I can now see the home page and now I can refresh this as many times as I want and the caching logic inside of ASP.net is looking at the request, seeing that there's a cached response for this request. So it doesn't even go into my code at all. It just returns that cached response. We don't execute code inside of the index action. We don't render any views. Everything is just a cached response that's being served back from memory and not executing code is the best optimization of all. It's the best way to increase performance and scalability. Now this also works with child actions. So let's stop debugging and let me add another action here. It's going to be a child action. It's called say hello. It's also going to be cached and we'll say for 60 seconds. And I went to change the duration of the output cache for the index action to just 5 seconds and now inside of the view for this index action, we will need to call and render this child action. So let me go into index.cshtml and do a quick html.action. Please render say hello. Call into that controller action. Save everything and let's run with the debugger again after I set a break point here, inside of say hello. So F5 and on this first request, we should see that we come into the index action which we do and we come into the say hello child action. So I'll continue debugging and now if I refresh, more than 5 seconds elapsed. So we're back inside the index action and we need to re-execute all of that code but we don't need to execute say hello again. So the MVC framework is able to cache that small part of the response independently and when I refresh the page, yes it has to rebuild everything else and execute the database inquiry inside the index action but it doesn't to call back in to say hello. If I had something that was particularly expensive to produce and it was relatively static so I could aggressively cache it, that's a good place to use html.action just to cache the result of something expensive.

Cache Settings

( Silence ) In addition to duration, the output cache attribute supports a number of different settings. These settings are all available from the underlying ASP.net cache engine. It's the same engine that web forms will use which is why I call caching part of the ASP.net infrastructure. It's not something that's specific to just the MVC framework but one of these settings is the vary by (phonetic) parameter. The default setting for this is star or asterisk which means vary by every parameter possible and that is usually the setting that you want. That's the default because you normally do not want to return the same cached response for different parameters and by parameters think of things like query string parameters. A query string parameter can contain a search term and you wouldn't want to return the same cached response when someone is searching for a and b. Those would be two different results for two different search strings but if I do want to take control over this, I could say vary by param equals none which means don't vary by any parameter. Always return the same response regardless of what you see in the query string or I can say vary by and specify the name of a parameter that I want to vary by. And you can have multiple names in there separated by a semicolon. So if I were to say vary by a param search term and someone searches and passes along the parameter value of a. That would be one cached response that ASP.net saves. A little bit someone comes along and passes a search term of z. That would be a second cached result that the ASP.net saves and it would only use that in response to another request that came in that says search term equals z. We'll see that in demo in just a bit. You can also set the cache location. The default value here is anywhere meaning it will cache on the server and the client can also cache the result. You can be very specific and say that the result should only be cached on the server or on the client or on proxy servers in between. There's also a vary by header setting. That allows you to vary the cache based on a specific HTTB (phonetic) header like except language. We wouldn't want to return a response that included English text when someone needs German text. We'll see how except language works in this module. And if all fails, you can always do vary by custom. When you do vary by custom you'll need to override a method inside the global.asax.cs file. Inside of that method, you can build your own custom caching string telling ASP.net what to cache and how to categorize it. You could look at anything in the request including headers or quickie values. There's some examples on MSDN that will show you how to do that. There's also examples that will show you how to setup a sequel dependency. A sequel dependency will cache a response until data inside of a sequel server table changes. It sounds great but it's not widely used just because there are a lot of restrictions on the type of sequel query that can be used. ( Silence ) In Visual Studio, now that we know how the output cache attribute works with a child action, I'm going to remove this child action because it's not really providing any value. It was just here for demonstration purposes. (Silence) And I'll also need to remove the call to html.action that was rendering that child action. And now that we've made that change, let's bump up the duration on the output cache attribute for our index action and run this with the debugger to see how it behaves with paging. And as expected that first request comes into index action, I press F5. We now have a cached response for the home page and I can come up and pound on the refresh button as much as I like. The ASP.net run time is always returning a cached response. Now let me try to go to the next page. We'll come back into the index action because the output cache directive will vary by every parameter and previously, we didn't have a page parameter. Now we do have a page parameter. The page number is equal to 2. So it needs to pick up a new response and cache that. Now we have cached response for page 2. I go to the next page, same thing. We come into the index action, produce that response for page 3, and now I have cached responses for pages 2 and 3. I can toggle between the two. It's not going to go back into the index action. So does this also work for search? Let's search for a 20. We come into the index action. That changed the search term parameter. We get the response and it appears on the screen correctly. What if I go to 21? Again into the index action, come back to 20. That just serves up a cached response. Let me show you a problem. What if I come directly to the application through a bookmark and I'm looking for search term equals 20? Well now the browser issued a get request not asynchronous request but a get request and it's displaying a cached response that was only a portion of the page. So we're missing the layout view. We're missing our style sheets. We're missing our scripts and here's the problem. The only difference between a full request which is what I just did that needs the entire page full of html and the AJAX request that we were using previously, the only difference between those two is an HTTB header known as x requested with. So the ASP.net caching layer, there's no change in the parameters. So it will happily return the partial view that it cached to fulfill a previous request but that's not what we want in this scenario and searching by going directly to the page with the value in the query string, if that's a supported scenario we need to fix this. Actually there's two things that we need to do. First, I'll stop debugging. ( Silence ) And we'll come into our output cache attribute and also tell it vary by a header. The x requested with header because this will be present on an AJAX request but not on a whole page request and now ASP.net will be able to tell the difference between the two and cached different responses. One with a full page of html, one with just the portion of the page that needs to update. Unfortunately this often isn't enough. We're also allowing the browser to cache the response with this output cache attribute and some browsers won't be smart enough to detect the difference between those two requests either. So I am also going to add a location parameter. To do this I need to bring in a name space for this numerated value, system.web.ui but now we're only going to allow caching on the server. And ASP.net will send instructions to the browser and the response so the response isn't cached. The browser always has to come back to the server and check. On the server, ASP.net can just serve up the response from memory if it has a cached response. Of course another solution to this problem would be to call a different action for AJAX requests and that would make a lot of sense. I combine the two into a single action here because it made it easy for a demo but it would be easy to change the url for an AJAX request in our JAVA script and have it call a different action. And then we wouldn't have this check for request.AJAXrequest. We would only return a full response. That is AJAX request method, by the way also looks at the (inaudible) requested with header to determine if a request is an asynch request. Now we'll run the application again and just to avoid any strange things. I'm going to go into the developer tools by pressing F12 and just telling Internet Explorer, clear out anything that you might have previously cached. This way we'll make sure that we're starting with a fresh slate when I test out this functionality with the new parameter values in place. So let's try the search for 20 and we hit the index action. That was good. Try a search for 21. Again, hitting the index action. Try an asynchronous search for 20. That was a cached response which was good and now let's go to the query string and enter a full request for search term equals 20 and we do, indeed, come into the index action and the page renders normally. And that's all thanks to doing a vary by header and only caching on the server. If we wanted the client browser to be able to cache too, we really would need to split this index action into two pieces. One that only responds to an AJAX request and one that responds for a full page.

Cache Profiles

( Silence ) In reality, it is almost impossible to pick the correct cache duration setting in a development environment. You need to put the application into a production like environment. Put it under load, take some measurements, and then make some adjustments. You might found out that you're using too much memory and caching requests that just aren't being used or you might find that you're not caching enough. For these reasons, ASP.net allows you to specify a cache profile in the output cache attribute. A cache profile then is something that's stored in your web.config file and can specify a duration. It's underneath a section named caching and it's inside of here where you can put duration values. As this example demonstrates, you can have multiple cash profiles and you reference the cache profile by name in the output cache attribute. I generally recommend using cache profiles instead of hard code and cache settings because first of all, it avoids repetition in the cache attributes. If you want to double the duration inside of all of your cache attributes, you don't have to do a search and replace throughout the code. You just go to one location in your web.config file. It's also easier to change the cache settings once the application is deployed because now all you do is edit the config file and then you avoid changing the settings in code, recompiling, and redeploying. And typically when you're doing performance tuning on an application, you need to run it while taking some measurements. Then you tweak the cache settings and then you run it again to make sure you did really get better results and it's not getting worse. And you'll do this over and over again until you reach some goal like reduced memory usage or reduced CPU or an increase in how many requests per second your application handles. ( Silence ) Inside our application if I wanted to use a cache profile instead of this hard code of duration, all I would need to do is come into the root level web.config and somewhere inside of system.web, create a caching element. We're going to create output cache settings. Notice you get intel sense (phonetic) the entire way through and we're going to create multiple output cache profiles. So let's add one called long that has a duration of 300 seconds and add another one with a duration of 3 seconds called short. And of course, you could have as many of these as you want. Now I just need to reference one of these from the home controller. So instead of having a duration, have cache profile equals long. And the application should behave just like it did before. Just now I have my numbers inside of a config file instead of hard coded inside of the controller. ( Silence )

Localization

The next topic we want to talk about it is how to localize an application. There are two important settings for every thread of execution that determine how an application behaves under different cultures. The first setting is exposed by the static current culture property of the thread class. This property tells the run time how to format strings. For instance, if we have a dot 2 (phonetic) string on a date time or a dot 2 on a currency, then how should the date time be formatted and what currency symbol should be used? Is it going to be a euro, a dollar, a rupee or a yen? Another important setting is exposed from the static current UI culture property. This setting impacts how the run time performs resource loading which we're about to look at. You can set these properties in your own code. For example, if you have one of those sites where you let the user select what language they want to see and that's in a drop down list or they click on a flag icon, that would be a scenario where you would set these properties manually. Or you can let ASP.net set these properties for you. ASP.net will set the properties based on the accept language HTTP header that the client's browser sends. In order for ASP.net to do this automatically though, you do have to add a globalization section to your web.doc config file and set the culture and UI culture attributes to auto. Let's see how these settings make our application behave. ( Silence ) Inside of the index view for the home controller, let's do a little experiment with the globalization settings. So first I'm going to add a code block that defines two variables, amount which is a decimal. That's what the m suffix is and sum date which is a date time representing July 9th of the year 2002. And now I will add some code to output these two variables by calling two string. Amount will call two string and pass in a C parameter which means please format this as a currency and with sum date, we'll just call two short date string. And before we run the application, I will come into the web config and inside of system.web, I will add that globalization section that we talked about. We need to set culture equals auto and we need to set UI culture equals auto. And now let's see how this behaves. I will run the application with Internet Explorer and since my default culture on this machine is United States English, I see the currency value is formatted with a dollar sign and the date is formatted as month, day, year. But now let me come into the Internet Explorer options. So ALT T will open the tools menu. Come into internet options. Under appearance, go to languages and here I can set language preferences. It's from here where I can add a language and in this list, let me select French and not the Canadian French but the real Francois from France and let me take that language and move it up in the list. So this is now essentially my preferred language and I will close this out and refresh the page. ( Silence ) And now my currency is formatted with a comma and a euro sign and the date format has also changed. It is now day month year. So what we've done so far is to see how globalization works and this is just a starting point for localization. We've changed how data amounts were formatted but we would also need to go through the application and take care of string literals like search by name if we really want to localize those to a specific culture because all the literal texts in here are still in English. Let's see how we can change that.

Resources

( Silence ) To localize the text, I went to display in this application. I'm going to rely on resource files. A resource file is an XML with a resx extension, r e s x and it stores localized strings for a culture. It can also contain binary assets like images but we'll just be using text in this application. To support different languages, I'll need at least one resource file for each language that I explicitly support. The first resource file I create will be the base resource file and this might have a name like strings.resx. If I want to produce a version of strings.resx that works with Spanish, I'd also create a strings.es.resx. So you can see the language the resx supports, it's embedded in the file name and it follows a naming convention to describe to the dot.net runtime what language and what culture a particularly resx file supports. You can also be very specific about the culture. So es dash mx would be Spanish Mexico. In dot.net there's a resource manager that will load the appropriate resource file for the current UI culture that's set but we generally do not have to interact with this resource manager directly because Visual Studio will automatically compile these resources into our project and generate code to access the resources with a strongly typed class. If you go to the properties window for a resx file, you'll see Visual Studio has the build action set to embedded resource. We'll talk about some of the other settings here when we get to the demonstration but to use a resource is very simple. In a view, I can get to a resource with a greeting property inside by going to the generated class and asking for the greeting property. This might return the string hello for English but it could also return hola if the current UI culture is set to Spanish and a Spanish resource exists. If a resource doesn't exist for the current UI settings, the resource manager will fall back to the base resource. Let's go into Visual Studio and see how all of this works. Inside the index view for the home controller, let me add a greeting to this view just above the other culturally sensitive information that we're displaying. And now what I want to do is localize this so it's not a hard coded string in English. We could display English or French or any other language that we create a resource file for. So the first step is to create that resource file. And one of the decisions you'll have to make when creating a resource file is where to create that because you can have multiple resource files and these resource files could live in the web project, in this ode to food (phonetic) project. You could also put them in a separate class library if that was more maintainable for you and you also need to select what folder these resource files are going to go into. To keep things simple, I think I might need a resource file just for the views for the home controller. And so what I might do in that situation is come into the home views folder and add a new item. I'll search for res and that should bring up resources file. I will call this just resources.resx and this will be my base resource file. It opens up in a Visual Studio resx file editor. One of the changes that I have to make sure I do inside of here is to change this access modifier. It's currently internal. I'm going to change that over to be public. This is just required because of the compilation model. It turns out that razor views are compiled into a different assembly, a different dll file than our ode to food assembly and without having these resource files be generated as public classes, we wouldn't be able to see them from our razor views but just changing that access modifier will set that for you. Behind the scenes what's happening is there is a class being generated for you. You can see it if you expand a resources.resx, there's a generated .cs file behind here and we'll take a look at that in just a second. First, let me come in and add a name. So this is how I want to reach a specific resource. I want to have the greeting resource and this is where I would give it a value like hello. And now if I save this resources file and open up the generated code, you'll see there's a class that's generated in the odetofood.views.home name space. It's called resources and all the way down here at the bottom -- ( Silence ) -- there will be a greeting property that I can get to and all the other code that is inside of here is just code that is required to have the .net resource manager load the appropriate resource file. And if I wanted to use this code from an index view, all I need to do now is replace my hard coded text with the name spaces, the class name, and the name of the resource that I want to display. In this case, greeting and at this point, if anyone comes to the site specifying any language, we're always going to display hello because this our base resource file. We don't have a resource file specified for any other languages so everyone gets English. If I wanted to customize this for say, French, that's when I could come in and add a second resource file. So let's add a new item -- ( Silence ) -- with the same name, resources.fr.resx and inside of here I would duplicate the name value pairs. You can obviously have as many name value pairs as you want but we just have one. The name is always greeting but this would be the French translation, bonjour. Notice there is no code generation for this resource file. Code generation only needs to happen on the base resource file, resources.resx. All of these files get embedded into my assembly by Visual Studio when it builds. That's why the build action is set to embedded resource. So you can think of all this information getting compiled right alongside with your C sharp code that goes with your application wherever you copy the dll file to. Well, let's run this and see how it behaves. ( Silence ) On the home page of the application, I'm still configured to prefer the French language. So I see bonjour. I see Euros. Now let me go into tools, internet options, languages, set language preferences, and move English back up to the top. Then close all these windows and refresh Internet Explorer and refresh Internet Explorer and nothing happens. I'm still seeing bonjour. Is this a problem with the way I created my resource files? Is this a problem with Internet Explorer? Well no, none of the above. It turns out, once again that we have to be very careful with something that we added earlier which is the output cache attribute. In this case, what is probably happening now is I'm making this request but all the MVC run time caching cares about is that all the parameters are the same, the x requested with hasn't changed. It's going to return a cached response that was built for the French language. I need to tell ASP.net that there's another header that it's need to vary the cache with and that is the accept language header. And we actually don't separate these with commas. We separate these with semicolons. And now if I build the application, let's try this again. I'll refresh this page and now we're displaying hello. That seems good. And let me go into tools, internet options, languages, set language preferences, move French back to the top, close one window. Close two windows. Close three windows and refresh and now we display bonjour. So caching isn't interfering with our localization anymore. And while we're here at the home controller there's something else you should know about resource files. There's really nothing that restricts you to using them only from a view. So I could have a greeting string defined here that is set to odetofood.views.home.resources.greeting and that would work just as well. The correct string will be loaded on the UI culture setting and in fact, you can also use these resources if you want to build custom error messages for your data annotations. So let's look at the restaurant review model that we built and we had things like required. And if the user doesn't fill something out here, they'll get an error message like the body field is required. Let's say you wanted to customize that and also customize it for different languages. In that case, you could come in and you wouldn't use error message. Error message would just be hard coded text on what you wanted the error message to be. We want this to be localized so I would specify error message resource type and I would set this to the type of my resource class, the generated class that was given to me and I would also specify the error message resource name. That is what to look up inside of that resource file. This is where I would provide one of the names that's inside of my resource file. Greeting, obviously, wouldn't make any sense as an error message but it's the only one I have in there now so I'm adding it here. You'd obviously have to create some error messages. Maybe they would be in a different resource file somewhere in your project but you could just specify the type of the resource and the name to pull out of that resource file and then you can have localized error messages too. ( Silence )

Diagnostics

Our last infrastructure topic is diagnostics. Diagnostics help us answer the question just what is our application doing? When did it start? When it restart? Are there any unhandled exceptions? Are there any authorization failures? To get this type of information, there are a number of options available. First, ASP.net does have a health monitoring system you can configure to report on everything from application start to application end and all the events in between, exceptions, failures, compilation events. There are also many third party options for logging and monitoring. Logfornet is a popular open source library you can reference to build your own logging infrastructure. And the patterns and practices team at Microsoft also has an application logging block. One really popular open source library is ELMAH, e l m a h. That stands for Error Logging Handles and Modules. ELMAH can capture events about your application and store them in a variety of destinations, xml files, text files, databases. It could even post them to Twitter and it has some prebuilt UI to show you the events it has recorded. Let's take a look.

Health Monitoring and ELMAH

( Silence ) The default configuration for ASP.net health monitoring lives in a web.config file but it doesn't live in the web.config file that's in the root of your project. Instead it lives in a machine level web.config file that applies configuration to every ASP.net application that runs on this computer. You can find that if you go into your Windows directory. Go to Microsfoft.net. Go to framework and then pick the right version number. We're using Visual Studio 2012 with .net 4 point 5. So of course we go to the version 4 directory because version 4 point 5 was an in place update to version 4. Dot.net version numbers have always been confusing and dot.net 4 point 5 continues that historical tradition. We go into that directory. There'll be a config directory. It's inside of here where you will find a machine level web.config that applies configuration everywhere. I can drag that into Visual Studio and we'll do a search, control I in here to find health monitoring. And the two most important sections in here are -- well actually there's 3 important sections. One is providers. Providers include destinations for where you can publish events. You might want to save them in the event log. In that case, you'd use the event log provider or maybe you want to store them in sequel server or maybe you want to use WOMI (phonetic) to publish these events. So those are the built-in providers. You can always add more providers if you write one. Another important section is event mappings. This categorizes all the possible events into buckets with friendly names. So we have all events which is every event that could happen in the health monitoring system, lifetime events, request processing events, and all errors. It is the rules section that will define where to send something like all errors. What provider do we send these to? So by default because this is in this machine level web config, we will send all errors to the event log. Any unhandled exception that occurs in the ASP.net application should appear in the Windows event log. Also, any failure audits should appear in the event log. You can add additional rules. The best thing to do would be to take this health monitoring section and mimic in your own web.config file if you want to reconfigure any of these rules but the one thing you need to be careful about is not to do something like send all request processing events to the Windows error log. It's not designed to handle a high volume of events being pumped into it. That would be something that you could do with sequel server though. But because we have these rules in effect, all errors go to the event log, that means if I have an unhandled exception, I should be able to see that in the Windows event log and in fact, we still do have a controller, the cuisine controller, that we wrote a long time ago that has an unhandled exception. It intentionally has an unhandled exception. So let's run the application and go there once. And now something terrible has happened. Let's see if I can find this in the Windows event log by going to the gigantic start menu and typing -- ( Silence ) -- event and under settings, there's view event logs. ( Silence ) ASP.net errors would be recorded under Windows logs, under application and here at the very top we can see there's been an ASP.net problem. It's listed as a warning but it's inside of here where you will find the event message which is that there was an unhandled exception. You can find the path to the application where this occurred, the machine name. You can find out who the web server was running as. Since we're using IS Express, it's running under my account and there's a full stack trace that we could use to try to track down exactly where this error came from. So all of that information's recorded in the event log for you automatically but sometimes the event log can be a little cumbersome to get to. Sometimes you want to store things in a way that you can get to them through the web interface or perhaps, you want to be able to email errors out to the people and that's when you can turn to ELMAH. Remember this stands for Error Logging Modules and Handlers. The easy way to install it is through newget (phonetic). We'll just do a search for ELMAH and the package I want to install is ELMAH.mvc because this will allow me to reach the ELMAH error logs in a nice MVC friendly way just by going to slash ELMAH. And now if I close this, it's been added to the project. I'm just going to do a build to make sure all the right files get copied to the right places. Otherwise, there's no code changes I need to make. We'll go back to the cuisine controller. I have a few exceptions pumped in here by pounding on the refresh key and now let me go to slash ELMAH. And I can see all the exceptions that it just recorded. You can drill into the details and see the stack trace. You can also configure ELMAH to send these via email or store them in different places. All that configuration can take place in your web.config file. In fact, ELMAH did add some configuration to my web.config file and it added some of these app settings. For instance, who is allowed to see the event log? Since we now have rules in the application, I could say well only admins. You can also configure the route that you use to get to see those error messages. So by default it's ELMAH and just go to slash ELMAH to see the errors, that's very simple. And one more thing, since we are going to require rules here. I'm going to explicitly say this requires authentication before you can view it. You can read more details about ELMAH and ELMAH configuration on the ELMAH website. ( Silence )

Summary

In this module, we spent some time with the ASP.net output cache attribute. We saw how to use the attribute and vary by location and vary by header to hopefully increase the performance of the application. I say hopefully because I never took any performance measurements before and after caching and the only way to know for sure that you've helped is to start taking hard measurements. So I encourage you to always take measurements before you start applying cache attributes. We also looked at the localization features of ASP.net and saw how we can let resource files and ASP.net take care of most of the hard work. We saw, for example, how the globalization settings in web.config can automatically give us different formatting behaviors for different languages depending on the client's accept language header. Finally, we took a look at diagnostics and we now have ELMAH added to the application and watching for errors and giving us an easy way to see the unhandled exceptions that occur. This was just a quick look at all the functionality that ELMAH can provide. ( Silence )

Unit Testing with ASP.NET MVC 4

Introduction

Hi, this is Scott Allen and this module looks at test-driven development and unit testing with the "ASP.NET MVC Framework". In this module, I want to show you how to use test-driven development to drive the design of a feature. We'll look at the TDD cycle, which is red, green, re-factor. First, you write a failing test which gives you a red result in the "Test Runner". Then you make the test pass, which gives you a green result. And then you re-factor the code to improve the design. I'm also going to demonstrate how to unit test controllers, and how to build test doubles to control the environment the controller executes inside of when we test them. ( Silence )

Test Driven Development

To give you some idea of the test-driven development flow, we're going to step away from the web project for a bit and focus on the test project. If you remember, in the first module of this course when we created the application, we also created a unit testing project. We can run tests inside of this project using the "Test Menu" or by using the "Control-R A" shortcut key. "Visual Studio" will find all of the test classes inside this project and run the test methods. We currently have a failing test, and if you look at the problem here by clicking on the test, and down here at the bottom of the test explorer, it will show you the exception that happened. You'll see there was an exception because the "Home Controller" tries to talk to the database, but it doesn't find the connection string that it's looking for, and this test is envoking the "Home Controller Index Action". We have that connection string defined in the web application, so the web application works just fine when we view the app with a browser, but the tests are failing because we haven't configured this properly yet. We're going to defer this problem and come back later. And I'll show you a couple approaches you can use for testing controllers that need data. For right now, I just want to focus on what it would look like to try test-driven development. I encourage you to try test-driven development, and write unit tests because I personally have found them to be the most beneficial practice I have ever followed in my career. And the one secret to this is that tests are just as much about design as they are about insuring quality. Let me show you what I mean. To get ready to implement this feature, I've already added a "Features" folder into my "Unit Tests" project. All I need to do is right-click this and say "Add a Unit Test". It's called "Unit Test One.cs". We can change the name later. For now, I'm going to paste in some comments that I jotted down as I was talking to the business people about this feature. It turns out that they want to try at least two different approaches and see which one works the best. One approach to calculating the overall rating for a restaurant would be to just take the simple average of the rating property for the last N number of reviews where N is something an administrator can configure, so it might be the last 10 reviews or the last 100 reviews. Another approach would be to compute a weighted mean where the most recent reviews are more heavily weighted. That would benefit a restaurant that is improving after a bad start and getting better reviews. It might also benefit a customer who is thinking about going to a restaurant that used to have great quality, but is now on the decline. I like starting with some comments inside of my test file, just so I can refer to them and think about them. They usually describe the business scenario that I'm trying to solve. I don't use the TDD approach for all of the code I write, but this represents the type of scenario where I probably would use TDD. I'm not sure how to design this feature, and test-driven development is a great design tool. I can delete the comments when I'm done, but for right now it provides me something concrete that I can look at and remember the goal, and I can start thinking about a solution and thinking about how I'd implement this feature. "Visual Studio" has already given me a test class. It has a method inside of it, "Test Method One". The first thing you might try to do is give this class and method a more descriptive name, but don't get too caught up on naming right away. My idea is that I want to use "Visual Studio" like I used to use a whiteboard before I started test-driven development. Someone would give me a software requirement, and I'd go to the whiteboard and start trying to diagram out some classes and methods that I think I would need. I'm going to do that in "Visual Studio" instead because "Visual Studio" gives me a compiler, and I can use tests to validate that the code works. And since I'm not sure what direction I'm going just yet, there's no need to spend time thinking of names. Once I've figured that out, we can rename things later. I do know I'm going to be working with some data, some restaurant data, and we already have that restaurant class defined. I just need to pull in the name space "Ode to Food.models", and my restaurant will have some reviews associated with it, and I need to bring in the name space for "List of T", which is "system.collections.generic". And once I have that, I'm ready to add some reviews to this restaurant. I'll just add a single review, and I think the only thing that we're going to care about for the purposes of these tests is that a review has a rating. In this case, we'll just have a single review, keep things very simple; the rating is a four. That's the data I want to evaluate. And now I can start thinking about the API that I want to use to compute the restaurant rating. This would be the point where I could experiment with different class names and method names until I found something that looks like it might work. And let's say I reach the point where I've decided that I think a restaurant rater might work, so I will create a restaurant rater and pass the data into it. And, of course, this code doesn't compile, but I don't expect it to yet. Remember, this is just a whiteboard. In fact, one of the first steps that you'll take many times with test-driven development is writing some code that doesn't even compile yet. The goal is to go and do the simplest possible thing to get it to compile once you're relatively happy with the names and the design. Next, I might say that this rater -- I can tell it to compute a rating given some number that it will use to determine how many reviews to use, so let's say 10 reviews, and it will give me back some sort of result. Again, this is the time where I am exploring how I want to invoke these API's, and what these API's should look like. What parameters do I want to pass? What do I want to name things? And forging ahead with what I have so far, I'm thinking that if I tell this to compute the result for a restaurant with a single review, and we're doing just the simple averaging for right now, I should be able to write an assert at this point, and assert that four would be the "result.rating". At this point, if I tried to do a build or run the test, it's going to fail miserably because we don't compile, but I can use "Visual Studio" to help me create the classes. Just put the cursor on "Restaurant Rater", open the drop-down list with "Control-Period", and let's generate a class from "Restaurant Rater" that will create a new ".cs" file in this project. Ultimately, it's going to need to be in the "Ode to Food" project because it's going to be business logic that we need to exercise from the web application, but leaving it in here is fine for right now. I haven't solidified anything yet; things might change. At some point, the design will start to firm up and I can move these ".cs" files into the right project, and then adjust the name spaces accordingly. My next compiler error is that the "Restaurant Rater" needs a constructor that takes some restaurant information. Again, I can generate this from "Visual Studio", just "Control-Period", and select the entry "Generate Constructor Stub". That should be there now. It's going to need a method called "Compute Rating". I will also generate that. And let's look at what we've developed so far in side of "Restaurant Rater". We have a "Public Constructor" that takes a restaurant that seems good. Let's get rid of the spurious name space here by deleting that and bringing in a using statement for "Ode to Food.models", and then I can delete that here also, just trying to clean up the code already. I should really wait until I have a passing test, and then come in and do changes like this. That would be the re-factoring step. This "Compute Rating" method, I believe, will ultimately return some sort of rating result. Let's create that, generate a class for "Rating Result". So it's not going to return "Object"; it's going to return a rating result, and we don't need this exception here. What I could do, the simplest possible thing to make this work, would be to just return a new rating result, not even do any calculations just yet. And are we ready to build? Not quite yet. It looks like my "Rating Result" needs a rating property, and my "Compute Rating" method needs to be public. So let's come in and change this. This is a public method, and "Rating Result" will ultimately be a public class, and it's going to have a "Property of Type Integer" that is the rating. I know this is an integer because the business specifically asked me to make ratings an integer value so we don't have ratings like seven point six six six that scare users; that would just be a seven or an eight. And at this point, I think, I can do a build. Build succeeded. I'll run all the tests with "Control-R A", and of course, our new test method fails. We told it that it should be expecting a four result; the actual result it got back from "Result.rating" was a zero because we haven't done any calculations. And this is the first step in TDD. I've designed some classes, designed some methods. I now have a failing test. My next goal is to do the simplest possible thing that will make that test pass, and the absolute simplest possible thing would be to say -- let's say that the result equals a new rating result, "'Result.rating' equals four". "Return Result", run the tests, now "Test Method One" passes. And you might look at that and say, "It's completely ridiculous. You're hard-coding computations just to make the test pass." However, stick with me on this. One of the tenets of doing test-driven development is to do the simplest possible thing to make a test pass, and then you just keep adding tests that will test more conditions. You're going to have to come in and change this "Compute Rating" code so it actually behaves correctly. But along the way, as you're making these changes, you're going to be writing tests that validate scenarios that are going to prove very valuable later on. They're going to make sure that you're making the right changes, that as you're adding things and adding features and reconstructing the code, that you're not breaking anything that used to work. That's one of the real values of having tests. And once you have a passing test, one of the first things you'll want to do is come in and re-factor, clean code up, make sure things are named properly. I don't really like the "To-Do" statement here. In fact, I don't really like having a data field in here when this is really a restaurant. And I can use "Visual Studio", "Control-Period", to say, "Rename data to restaurant everywhere," And that fixes things up for me. I also want this to be called "Restaurant". Same trick here, "Control-Period", it will rename the local use of that restaurant variable, also, this "P parameter", no idea what that means, so let's just change it right away to "Number of Reviews". That's what it's supposed to be, ultimately. And now I'll hit "Control-R", "Control-A" again. Test method is still passing. My changes haven't broken anything. Those are very simple changes, but you can imagine, as things build up in complexity, you change things in one place it breaks something somewhere else. That's a sign that the tests are helping you, and also that you might have a design issue. Maybe something is too coupled to another class. But now that we've seen a basic test in action, I'm going to accelerate things a bit, and work through the rest of the design and implementation of this feature. ( Silence )

Test Driven Design

I've added a second test to the "Test Fixture". This one passes in two "Restaurant Reviews" to the "Restaurant Rater". Since the code is starting to take some form, I've also given the tests some names, "Computes Result for One Review", and "Computes Results for Two Reviews". I've also changed the implementation inside of the "Restaurant Rater". The simplest possible thing I've found to work is a very simple link statement. It averages the reviews. The good news is both of the tests now pass and I'm in a re-factoring stage. I think the "Restaurant Rater" is in good shape so far. It's actually the tests that are bothering me. One thing you'll learn about unit testing in general is that the tests are just as important as the application code. And it bothers me that I have so much setup work inside of each test to make a restaurant and make the reviews for that restaurant, so in this re-factoring phase, I'm going to re-factor the tests to make the setup a little bit easier. ( Silence ) What I've done to re-factor the tests is to add a method, a private method, named "Build Restaurants and Reviews". This is not a test method that the "Test Runner" will invoke to see if something passes or fails. It doesn't have a "Test Method Attribute". This is a helper method I can use for my other tests, to keep them simple. When you look at a test, it's a little easier to see that "Arrange, Act, Assert" pattern that we talked about in the introduction to this course. This new method takes a variable number of integer parameters, and uses some link operators to transform the integers into a list of reviews. Those reviews are attached to the restaurant. We just return the "Restaurant" then. I can use this method in my tests by either creating an array of integers and passing them into this method, as we did here with the values four and eight, or I can use the params ability of "C Sharp" to just pass them in one by one. And now I'm feeling a bit better about these tests. They look a little bit cleaner, and both the tests are still passing, which is good. For the next step, I might want to start to test boundary conditions. What happens when we rate a restaurant with zero reviews, or if we rate a restaurant that has a negative review? Should we throw exceptions? How do we deal with odd numbers? Those would all be good tests to write, but I'm sure I can come up with an implementation to satisfy all those odd scenarios. They don't feel like they'd be high risk or difficult to implement. And at this early point, I'm really trying to focus on design issues first because, remember, TDD is primarily about design, and if I look at these requirements again, they say we have to support at least two different types of averaging. There's the simple average and the weighted average, and the business people also hinted that there might be more in the future. They want to try, and experiment, and find out what will work the best. I want to see what happens if I introduce the second form of averaging, and see how it impacts my design. ( Silence ) I have now added a new test, the "Test the Weighted Average of Two Reviews". Remember, the requirements also said I need to weight the most recent reviews twice as heavily as the oldest reviews. And we haven't worked with review dates. We haven't sorted anything yet, but in the future, again, I know I can write some tests to make sure that the reviews are properly sorted before I compute the rating. I can do that later. Sorting is another implementation detail I know I can figure out, and I'm not finished writing tests yet but I am still focused on design. In this weighting test, the rating for a restaurant with review ratings of three and nine is five because the value three is more heavily weighted than the value nine. It comes before the value nine in order. I've put together a quick and dirty implementation that will pass the test, and, of course, I did this after I watched the test fail. And this is what the implementation looks like. I'm not in the re-factoring phase again. I just want to show you some of the things that tests will make you think about, for instance, rounding. Right now, I'm truncating the average rating, so the link statement computes the average rating and casts the result to an int. The weighted average also performs division between two integer values, which will result in truncation. So the question is, is truncating the right behavior? This is one of those scenarios that tests will make you think about, and many times you'll need to go back to the business people and ask for clarification. All I know so far is that they told me that the rating should be a whole number like six or seven so I need to clarify the exact behavior. Do I need to round? Do I need to truncate? Once I have an answer, I can write the tests that make sure that the rounding or the truncating happens correctly. But for right now, I still want to think about high level design issues, and come back to handling simple averages versus weighted averages. One thing that bothers me about this "Restaurant Rater" right now is how it is responsible for computing a simple rating, which I would probably rename if it was going to stay here, and computing a weighted rate. But I know, based on my notes, that this is an area where I can anticipate change. The business is going to want to add new algorithms, and change algorithms all the time in search for something that works the best. If we had to go into the "Restaurant Rater" and add a new method every time the business came up with a new algorithm, I think that would make things difficult to change. I want to make things easy to change, make them as easy as possible. Let me do a bit more re-factoring, and I'll see if I can make it easy to change algorithms while still having my tests pass. It's wonderful having these passing tests now because I can rip code apart, rebuild everything, and my tests will tell me if I did anything wrong. ( Silence ) I've done a bit more re-factoring, and I've decided that the "Restaurant Rater" should not be responsible for computing the actual result. You'll notice the class is much smaller now. Instead of computing the result directly, the rater is going to rely on an algorithm that is passed in and abstracted behind an interface definition. The interface definition is "I-Rating algorithm", and it looks like this. Any object that implements this interface you can call "Compute on" and pass in a list of reviews. It will then return the proper new rating result. I've made two concrete implementations of this interface. They're both in the same file. That's something that I'd want to change later on, move these classes to their own files perhaps, but we can worry about that later. The first implementation performs a "Simple Averaging of Reviews". The second implementation performs a "Weighted Averaging of the Reviews". You might have noticed that the code inside the algorithms is the same code I had before inside of the "Restaurant Rater" itself, so I just moved code around. I took code out of the "Restaurant Rater" and pasted it into these algorithm classes. Is that good? I think so, because it's assigning specific responsibilities to different classes now. I now have algorithms that are dedicated to and focused on computing a rating result, and I have a "Restaurant Rater" that is now going to be focused on coordinating all of the pieces together needed to produce that result. And now when a client needs to determine a rating, like if we looked inside of our tests now, we don't need to figure out which method to call on the "Restaurant Rater". We always call "Compute Result", but we pass in the algorithm that's required to perform the computation. This is the Strategy Design pattern for you pattern fans out there. What's important about this pattern is how I can now introduce new algorithms without changing any of the code inside the "Restaurant Rater" or inside of any of the existing algorithms. I've made the system much easier to extend, and we should be able to keep up with all the changes that the business wants to introduce in this area as they experiment with different ways to rate restaurants. All I need to do is write a class that implements "I-Rating algorithm". The good news is, after I've ripped all this code apart, all of the tests still pass, so I know I haven't broken anything. And you might be wondering, if all I did was move code around, then why couldn't I just call this algorithm directly instead of going through this rater class? If I'm a client, like our unit tests are clients for this code, why do I instantiate a "Restaurant Rater" and call "Compute Rate"? Why don't I just instantiate an algorithm and call "Compute Rate"? And the answer is that we haven't given the "Restaurant Rater" all of the responsibilities it needs to take care of just yet. Let me write another test and come right back. I've written a new test, this one to make sure that the "Restaurant Rater" only uses the top N number of reviews in the calculation. To test this scenario, I've built a restaurant with six reviews, one, one, one followed by 10, 10, 10, and I expect if I tell the rater to only use the first three reviews, it should produce an average of one. If I run the tests with this new test in place, I'll discover that this test fails because I haven't implemented the logic for this to work yet. Let me go into the "Restaurant Rater", and get a list of filtered reviews. And I think what I can do is just take "Restaurant.reviews", and take the first number of reviews to use, and that should satisfy the logic. This should make the test pass, but let me do a "Filtered Reviews.to List", run the test again, and I can see that that test now passes. And now you can see that the "Restaurant Rater" is going to have a purpose in life. It's going to massage the data, and set everything up properly so the algorithm can work. It's still the algorithm that does the computations and leaves us open to extending the software with new algorithms. And at this point, I'm sure we could think of many more tests to write. We could write tests that check for rounding errors and truncating errors, tests for restaurants with no reviews, tests to make sure the reviews are sorted properly, and the list goes on. I'm not going to write those tests for you in this video. I'll leave that as an exercise for you. Instead, we're going to move on and look at more MVC-specific issues around unit testing. But the goal of this was that I hope I've shown you some of the benefits of test-first development and how to apply TDD. It has a real learning curve. You need to keep trying, and then learning, and then trying again. You might think this example was too simplistic, but I assure you that after many years of doing this if you can make your code easy to test, like in this example, and there's always a way to make code easier to test, then you're going to have success, not only with test-driven development but also in writing maintainable code and building software. ( Silence )

Home Controller Tests

When you are unit testing an "ASP.NET MVC" application, you'll probably want to unit test your controllers. And one of the first decisions you'll need to make is if you want to isolate your controllers from infrastructure-related services that they use, like web services or mail servers, or, in the case of our application, a database. There are two ways we could go. We could write unit tests that hit the controllers, and the controllers go against the database. This is perfectly reasonable for some people. Other people think it is an atrocity. I just want to let you know there is some debate on this topic. The problem with writing unit tests against your controllers when your controllers need to access a database is that the unit tests can run a little bit slower, and in unit testing you generally want the test to run as fast as possible. But there's actually a bigger problem, and that's usually setup. Because you are writing tests, you want the controllers to behave predictably, and that means you need predictable data. So you need to make sure the database is set up, it has the right data in it, there's nothing new there that could break a test. In the long run, this can actually end up being a lot of work. I would suggest if you go down this path, you might look at an embedded database like "Sequel Server Compact". With the "Entity Framework", you can point the connection string to a Sequel compact database, and it can be a little bit easier to work with compared to a real Sequel server. But "Sequel Compact" doesn't support all the queries, and features that regular "Sequel Server" supports, so you can still run into a little trouble, in which case you might stick with a real Sequel server, but your tests are going to be slower. I want to show you how to isolate your controllers from the database using a simple approach, the simplest approach I can come up with. Right now, a test for the home controller is failing because the entity framework is running inside, and it's looking for a database connection string. It's the index test, and the exception that's being thrown is "no connection string named default connection can be found in the application config file." If I want this test to pass, I can set up a real database for my unit test to use, or I can remove the database from the application when it's under test. And removing the database usually involves some sort of interface definition. Let me show you how this works. First, we'll have an interface "IO Defu DB" (phonetic). This will represent all the operations that I want to do against a real database. Not everything is there yet. All we have so far is just a "Query of T" method. I want to be able to query different objects, restaurants and reviews. The "Ode to Food DB" class that we've been using all along now implements "IO Defu DB". It implements that query method explicitly, meaning you can only get to this query method through an "IO Defu DB" reference. All this method needs to do is turn around and call into the DB context that it derives from. There's a set method on that DB context that would essentially be asking for the queryable set of entities. So if someone wants a query of "Restaurant", all we need to do is turn around and return a set of "Restaurant". Then inside the "Home Controller", I can have it work against an "IO Defu DB" reference instead of a real "Ode to Food DB". At runtime, when it's actually running on a web server, we'll give it a real "Ode to Food DB", so when it says "DB.query Restaurants", that'll actually be something that goes to "Sequel Server". But when it's under test, the beautiful thing about an interface is the "Home Controller" really doesn't know what it's talking to, and under test we can have "IO Defu DB" point to something that just contains some in-memory data. We do just have to make a couple changes to the "Home Controller". One is that we can no longer say, "DB.restaurants". Now we have to say, "I want to query restaurant," and we'll need to do that in two places, once in the "Auto-Complete" and once in the "Index". The other change that we'll have to make is that when the controller is constructed, we'll need to initialize this to point to something. When we're in a test, we want it to point to something fake that the test passes in. When we're not in a test, we want it to use a real "Ode to Food DB". For right now, let me just define two constructors, one default constructor that will initialize this to the "Ode to Food DB" that we've always been using, and another constructor where you will pass in an "IO Defu DB", and we'll just assign it. That'll give a unit test a chance to pass in something for the "Home Controller" to use that looks like a real database but it isn't. And with that in place, what I can do inside of the test now when we're arranging this controller, I can set up a fake database. So I could say, "DB equals New, Fake Ode Defu DB". This is a class that I've already written. What it does is simply implement the "IO Defu DB" interface. And behind the scenes there's just in-memory data. It's not the prettiest code in the world with all the generics, but essentially behind the scenes there's a dictionary of type. So when you want to query "Restaurant", we'll pull out a bunch of "Restaurant" out of that dictionary, something that's I-queryable, and hand it over to you. And there's a method here called "Add Set", which allows you to program the restaurants that you want inside of this "Fake Ode Defu" database. In fact, I've already defined a class, "Test Data" that has some restaurants that it just conjures up in-memory. So 100 restaurants, every restaurant has a review with a rating of four. And inside the "Home Controller" tests, what I can do is initialize the database with that, to add a set from "Test Data.restaurants", and I can use that "Test Data.restaurant" over and over again for all the controllers that need to query restaurant information. And now when I construct the "Home Controller", I don't want it to use the real database. I want it to use that fake in-memory database. Just with those changes, let's run the test again and see what happens. And now the index method is still failing, but it's a different exception now. The exception now is because of our call to "Is Ajacks (phonetic) Request". We've run into trouble with this before, but I've been leaving it in the "Home Controller" just to make a point, and that is there's nowhere else in the application where we have actually gone up to an "HCDP Context Object" or messed with the query string or cookies or anything like that. And for the most part, with "ASP.NET MVC", you don't need to do any of those things. But when you do, you can almost be certain that it will make the tests a little bit harder to write. To this "Call to Request.is Ajacks Request", what's happening behind the scenes is it's looking at the HCDP request. Well, there isn't one; it sees a NULL reference somewhere, and an exception gets thrown. Fortunately, with "ASP.net" MVC, there is a way to work around this. And I have another class defined, if we come over here into the test project. It is a "Fake Controller Context". A "Fake Controller Context" is something we'll assign to the controller and give it a "Fake HCDP Context". A "Fake HCTP Context", then, is going to have a "Fake Request Object", and this "Fake Request Object", again, really doesn't do anything, just returns nulls, just returns empty collections. All of this is just here so that we don't get that exception. Very simple implementation, but it's a lot of code just to have that one call inside of the index method to check if it's an "Ajacks Request". And using these classes, what I can do is come back into the test and ALSO TELL the controller that its controller context is going to be a "Fake Controller Context". And now let's run this test one more time. It's still going to fail, but at least we've gotten to the "Assert" statement. The "Assert" is failing because we no longer put a message in the VIEW BAG that says, "Modify this template to jump-start your MVC application." We've changed it and gotten away from that starter template, so this "Assert" really isn't valid anymore. But now what could we assert. We just fed the "Home Controller" a fake data source that has 100 restaurants in it. We only want 10 on the home page, so maybe what I should do is look at the model that's in this "View Result" and assert that we have 10 restaurants inside of it. Remember, our "Home Controller" should be returning a sequence of "Restaurant List View Model". So what I could say here is that I expect to have an I-numerable of "Restaurant List View Model", that's my model, and it's equal to "Resolve.model", which is typed as "Object" because "View Result" really doesn't know what type of model you're going to have, so if I just do some type coercion here, then my code should be happy and, now I could write an assert like, "Assert that R equal 10 is the number of those items in the model." And finally, run the test one more time, and finally, we have a passing test that's actually testing something that we want. And this is the kind of path that you'll go down if you want to isolate your controllers from actual infrastructure things, program to interfaces, have fake or stub or mock implementations of those interfaces for your tests. And we can build a whole course around the best testing strategies, and fakes versus mocks. In fact, you'll find those courses on "Pluralsight" if you're a "Pluralsight" subscriber. I'm just trying to give you a sample of what this would look like. ( Silence )

Create Action Tests

Let's see what it would look like to test the "Create" action of the "Restaurant Controller". Here are two tests with less than 10 lines of code for each test. The first test wants to make sure a controller saves a restaurant when the restaurant is valid. The second test wants to make sure we do not save a restaurant when the model state is invalid. Both of these tests use our fake database to make it easy to test. In the scenario where there is an error, we set up "Model State" to look like validation failed. I can do that by adding a model error to "Model State". You can call "Add Model Error" to simulate an error condition. You can also call "Add Model Error" inside of "Controller" if you wanted a real error condition that would display to the user. But I will point out that you need to be careful. If you're calling "Add Model Error" a lot in your controllers, it might mean that your controllers have too many validation responsibilities, and validation is generally a responsibility you can assign elsewhere. In this application, we're using validation attributes, but you could also have a model validate itself. Once we call the "Create Action", we'll make sure nothing was added to the "Added" collection of our fake database. And this demonstrates how fakes have a different purpose than real objects. Fakes should make things easy to test, perhaps even adding additional properties and methods to figure out how the object was used by the code under test. In the "Create Scenario", where we want the "Create" to work, we can make sure our restaurant was added to the database by asserting that "Added.count is equal to one", and also check the saved property of the fake database to ensure the controller called "Save Changes", which will save everything. Obviously, the fake database has some additional features now for the "Create" scenario to be testable like this, so let's see what they are. We'll start with the interface definition. This interface definition now contains everything that we currently need to do against the database. We can query objects, add an object, update an object, remove an object, and call "Save Changes" to flush everything to the database. For the real "Ode to Food DB" that implements this interface, we just need to forward these calls to the "Entity Framework" DB Context API, calling methods like "Add" or "Remove" on a DB set, or in the case of "Update" we call the "Entry" API. Remember, "Entry" was a way of attaching an existing object to the context so the entity framework would track the object, and it will issue an update statement because we tell the framework the object has been modified. The fake database takes a different approach. When you call "Add", it keeps the object in memory in a collection that remembers all the added objects. So during a test, it would be easy to see what's been added, what's been updated, what's been removed. We also have a flag telling us when someone calls "Save Changes", so we can write asserts and make sure that changes were committed to the database. And the end result is that we can write relatively simple tests, tests that will execute very fast because all the data is in memory. We can also control the data in this fake database, and we don't have to set up a schema or database file or make sure we're pointed to the right database server. And this represents, as I said, just one approach to testing controllers. Some people will test against the database; some people will test against in-memory fakes. You have to find what you are comfortable with, and what works for your team and for your application. And the tests here might be tests you want to write, or you can certainly have different tests or additional tests. You might want to test that the controller returns the proper "Action Result", like a "Redirect" result when it successfully saves or updates and object. And that would also be a valid and easy test to write. ( Silence )

Summary

In this module, we looked at test-driven development and unit testing with "ASP dot" and "MVC". I demonstrated the test-first development cycle of red-green re-factor to implement a new feature. We saw how TDD is more about design than quality control, but a wonderful side effect of TDD is how we have unit tests in place when we're finished with the implementation, and that allows us to make changes in the future to the code. We also looked at unit testing controllers. "MVC" makes controllers easy to test. For the most part, you just need to instantiate them and pass in parameters, and assert on the result. We just have to make sure the code we write inside the controller is testable. And we looked at using fake testables to give us complete control over the execution environment for a controller, and avoid using a database in our unit tests. If you're new to unit testing, I really hope this module has given you some inspiration to try it out.

Deployment and Configuration

Introduction

Hi, this is Scott Allan. And in this module I'll be focusing on configuration and deployment of an ASP.NET MVC application. I'll talk about the various configuration files that you'll find on the machine, and what those configuration files mean to your application. I'll also be transforming my development machine into a webserver by installing internet information services -- or IIS -- and building a web deployment package to install the application into IIS. Then finally, we'll take a look at deploying to the cloud using Microsoft's Windows Azure websites. ( Silence )

Configuration Files

Configuration files and .NET are XML files that control everything about the environment your code will execute inside of. For web applications, this includes the authentication settings as we saw in the security module of this course, but config files also control the compilation of use, database connection strings, cryptography settings, custom errors settings, and much, much more. The configuration system in .NET is also extensible, so if you want to build your own custom configuration section, all you need to do is write some classes to support that. I'm not going to show you that specific scenario in this module, but I will demonstrate an easy way for you to store custom settings in a configuration file. Inside of Visual Studio I'm going to open up the web.config file that's in the root of this project; the one we've been using repeatedly throughout this course. This is where we specify the connection string for the database that we want to use. We've also looked at the globalization settings in here, we've created some cash profiles, we've configured custom errors, and we've looked at authentication. So there are various components of ASP.NET and ASP.NET MVC, and also of the webserver itself that look in this configuration file for settings. And so, the obvious question might be what if I want to keep my own settings inside of web.config? One of the easiest things to do if you just have a simple value that you want to store is to add it to the app settings section. Right here I can add a new setting with a key of mail server and give it a value. And now, I want to be able to retrieve that value inside of my MVC application; let's say, somewhere inside of the home controller. Let's just read that value out, and put it into a view. The easiest way to do that would be to just put it into viewbag. And the way I can read that setting is to use configuration manager that's in a namespace system.configuration. Configuration manager will give me access to connection strings, to app settings; really to anything that's in a config file. But here I'm explicitly asking for the mail server app setting. We'll grab that value. We'll put it in viewbag, and in the index view that that action renders. Let's delete some of the culture globalization stuff that we were fiddling with and instead write out viewbag.mailserver to see if we've retrieved this correctly. I'll do a build and refresh the homepage of the application, and there you can see we get out mail.server.com. So that worked. And configuration manager makes it very easy to grab app settings, and really any setting that's in a web.config file. And this is the place where you want to put things that you don't necessarily want to hardcode. So connection strings, server names, file share names; all of those types of things can go into your configuration file. That way, you can change them just by changing the config file and you don't have to recompile and redeploy all your binaries. ( Silence )

Configuration Hierarchy

Configuration files for .NET are hierarchical, meaning the configuration for your application is controlled not only by the web.config in your project, but also by configuration files at a higher level. When you're running the MVC application, the configuration starts with a machine level configuration file. That file controls basic settings for all .NET applications that run on the same computer; desktop applications, windows services, and even web applications -- everything. We'll take a look at where this file lives and what's inside in just a minute. There's also a machine level web.config file. We looked at that when we were poking around in the health monitoring settings. It puts in place all the default settings for every ASP.NET application on the machine. And before we get to the web.config that's in our application, we might also have a parent web.config file that we inherit settings from, and that would happen if our application is deployed underneath another ASP.NET web application. These configuration settings then are inherited downwards in the configuration in effect for your application is a combination of all the configuration files. Inside my web.config file I can generally override any settings that might be in place from the machine level config or the machine level web.config file, although administrators do have the option of locking down settings and preventing me from changing a value. Let's take a look at what these configuration files look like. Using Windows Explorer, let's find the machine level configuration file. I need to look on the C drive under windows, under Microsoft.net, framework. And as I mentioned in an earlier module, even though we're running .NET 4.5 we need to look in the version 4 config folder. It's inside of here where we will find the machine level web.config. I can open that up. It's about the most boring configuration file you'll ever see. Most of it is just sections describing the types to use for other configuration sections when they're loaded. But there are some things in here, and there are some things in here that actually apply to ASP.NET. For instance, all the way at the bottom, here's the default membership provider that's configured for all ASP.NET applications. It's the ASPNET sql membership provider, but we're not using this in our application. We have changed the setting in our web.config -- overridden it. Down here at the bottom -- remember we added our own membership section that is using the simple membership provider. So that's an example of changing the defaults that are specified at a higher level. We can also find the root level web.config file here. This is the file that would apply to all ASP.NET web applications that are running version 4 or version 4.5 of .NET. It's inside of here where you can find things like the default authorization rule; will allow all users, even anonymous users into every site unless you tell us otherwise. It's inside of this file where you can also find the default health monitoring rules -- we looked at those previously -- the default HTTP handlers; that is like a file with and ASHX extension. What is the handler for that? What class does the .NET runtime need to instantiate to handle that request? And we can also see inside of here some default modules. Modules provide services like implementing our forms authentication, doing checks against the cookie that arrives to see if the user has been authenticated. Also, the routing engine is an HTTP module. It runs in the ASP.NET pipeline and gets to look at every request. That's what an HTTP module can do. So many settings in this root level web.config file, but eventually the settings inside of our own web.config file get to take effect. And these can override the settings like for membership. And we can even inside of our own application have web.config files that override settings for a specific directory. For instance, there's a web.config file in the views folder. This is the web.config file that controls razor configuration. These are the namespaces that are in effect when a razor template is parts and compiled. We added a few custom namespaces here during the course. And there's other interesting things in here too. Like if I scroll down here a little bit, there is an HTTP handler registered essentially saying, if a request arrives here looking for any type of file using any type of HTTP verb -- doesn't matter if it's a get or a post -- the component that will handle that is the HTTP not found handler. The HTTP not found handler essentially returns a 404 error to the client. Why are we doing that inside of the views folder? We're doing that because if someone launches a browser and request and file that is in this views folder directly -- like if they typed in localhost/views/home/index.cshtml, we want to return a 404 error. A user shouldn't be able to go directly to a view using a browser. They should go through a controller. It's the controller that gets to select a view. Therefore, we'll just return a 404 if anyone requests a file inside of here directly. And this file is here by default when you start a new MVC application. ( Silence )

Hosting ASP.NET MVC

Now that we understand a little bit about ASP.NET configuration, let's see how to deploy an application and how to change configurations on a deployed website. First, understand that when you build an MVC application, you're producing a .DLL file with the application logic inside. That's going to sit in the bin directory of your project. DLL's don't execute by themselves. They need some sort of host process to load them into memory. And for websites that host processes, also then responsible for delivering HTTP requests to the logic inside the DLL. We've been using IIS Express as the host for our application so far. IIS Express makes things very easy for development because it runs with our identity, and we can start and stop the webserver whenever we need to. But if you're deploying an application for real on the internet or intranet, chances are you'll be using the full version of internet information services. You can install IIS on nearly any version of Windows, but it is off by default on nearly all versions of Windows. So you might need to go into Windows features or use the web platform installer to get IIS onto a machine. Let me show you how that works. What I want to do now is transform my development environment into a server type environment by installing IIS. And one of the easiest ways to do that is to use the web platform installer. You might remember that we used this in the introduction to this course to install Visual Studio. Now, I'm going to launch it, and install IIS as well as a few other tools. So up here in the top right where I can search, I'm going to search for IIS and ASP.NET. That should find me this entry; IIS with ASP.NET 4.5. Let's add that. I also want to be able to manage IIS. So let me search for IIS, and select the IIS management console. And I'm also going to install sql server -- a sql server express edition because even though we already have local DB installed, it turns out that using local DB from IIS is a bit of a pain. It can be done, but it requires a lot of configuration and it's configuration that you'd never have to do, because when you really do deploy to production, chances are you'll have a real instance of sql server and a real license, and everything will be set up for you. I'm just going to install sql server express just to get this running under IIS, and let you see what it would look like. I'm also going to install sql management tools because chances are we're going to have to log into that database and tweak some permissions. So I will install the sql server management studio. Four components to install just click the install button. I will need to fill out some information about the SA password. SA stands for system administrator. That's the all-powerful account inside of sql server. I'm just giving it an initial password. I can click continue and accept, and we'll come back after all of this has finished downloading and installing. Now the web platform installer has finished. I should be able to exit out. And I should be able to open up internet explorer and go to local host. And I can see that IIS 8 is running on this machine. That's a good sign. This is exactly where we want to deploy our application. So if I come to local host I won't see the IIS banner page. I'll instead see Ode To Food. Let's check to make sure the other components installed too. I'm going to go and search for sql management studio, and once this has initialized I will try to connect to the sql express instance. There's two ways to get to a local sql express instance. You can specify a server name/sqlexpress. You can also use a period here to specify that you want to go to the local sql express instance. I should be able to click connect. And that looks good, there's no databases there yet. That's where we're going to put our Ode To Food database. You can also use this management tool by the way, to connect to local DB. A connection string is just like you see in your web.config file. It's local DB inside a parenthesis, slash V11.0. And when I connect to that I can see our Ode To Food DB as well as some other temporary databases that Visual Studio has created for things like storing the unit test results. And finally, there was one more tool that we installed. That was the IIS management console. It wants me to launch the web platform installer. We don't need to do that right now. I just want to see that IIS is running on this machine. Inside of this manager you can go through and modify any of the settings for this server. A lot of this just maps down to the same XML files that we've been looking at -- configuration files. Inside of a server I can see two nodes: there's application pools. An application pool is a process that a web application will run inside of. We'll be using the default application pool. I can already see there's one application running there. That's the website that we went to that just has the IIS banner. If you go into the task manager for Windows -- and in Windows 8 the task manager is a little bit fancier than the previous versions of Windows. So let me go to more details, and go to the details tab. Inside of here I should be able to find a W3WP.EXE. That's a world wide web worker process .EXE. That will be the process that will host our MVC application default app pool. And also, inside the manager if I drill into sites I can see there's one site in here now. It's the default website. That's where we will deploy our application. You can have additional websites, and you can have web applications listed under that website. We want to deploy our application to the root of this website. So that's what we'll get ready to do next. ( Silence )

Preparing For Deployment

There's just a couple housekeeping tasks that I want to take care of to get my application ready for deployment. These mostly revolve around the database. The first thing I'd like to do is re-do our database migrations. We've been a little loose with our migrations. We've been running with automatic migrations enabled, which means, we don't get explicit migration scripts; or migration files in the migrations folder. Instead when we go to the package manager and go to update database, it just makes whatever changes in the database that it sees fit to do. Whatever is needed to sink up the schema with our models. But for my first deployment, I'd like to baseline the database and get us into a state where we will not be using automatic migrations anymore. We'll be a little more careful because after we deploy, we'll get live data in the database. We need to start taking care of things, and making sure we're managing the schema properly. At least that's how I feel. I essentially want to restart the migrations. There's nothing in the database that I need right now. All the data comes from our seed method down here, which while I'm in here, I'm also going to remove the 1000 restaurants that we create just to test paging and sorting because I don't want to deploy those either. And I will also delete our initial database migration script. I'm just going to recreate this from scratch using a brand new database. And the definition of the schema will be based on what is in our model classes right now; what's a restaurant, what's a review. And the best way to do this with entity framework migrations is to come in and also delete your database, so it doesn't see anything there and knows that you want to start from scratch. And I can do this through the sql server management studio just by highlighting Ode To Food and right-clicking and saying delete; or hitting the delete key. I'm going to tell sql management studio to close any existing connections to make sure we can drop this database. And it looks like it's gone. And now reopen the package manager console, and let's create our initial migration. Remember, we're not using automatic migrations anymore. We won't be able to just say update database and have it create a database for us. We need to have a migration script there first. So add migration, initial create. And I need to make sure that this executes us under the right project; not OdeToFood.test, but OdeToFood. That's where my entity framework DB context class is. And now we will have an initial create that includes everything that we've been using, the user profile class, restaurants, restaurant reviews. And this is the point where I would also go through this file and make sure all the column sizes are reasonable. Make sure indexes are applied. But for this application we can forge ahead. Now, the other thing I'm going to want to do with this application is that it when deploys, I want it to run the migrations automatically. There's a way to do that through configuration, but we're actually going to do it through code. And the other thing I want to do before I forget is make sure that web security does not get initialized more than once because if you call initialized database more than once, this line of code will throw an exception and stop everything. It's very easy to check if web security is initialized. If web security initialized -- and put a not in front of here -- so if it's not initialized, we will initialize the database. And the reason I'm putting that check there is because now we're going to be running migrations when the web application runs. So this initialized database connection can run when the web application is running. And we have another call to initialize the database connection here during application start. I'm also going to put a guard around this. Although I'm fairly certain it's always going to run before we reach this point because I want to run the migrations before we get into this point in application start. I'm still going to put that check there just to make sure it doesn't throw and exception. And it is during the application start event when I want to run the migrations, and I can do that using a class called DB migrator. It is in the system.data.entity.migrations namespace. So I'm just going to bring that in. And we give it our configuration class that is in our project in a migrations folder. So also bring that namespace in. And then just tell it to update. That will run any schema changes that need applied. That will run the seed method to make sure the database is populated. And the reason I'm doing this in code instead of in my web.config file, which is possible is because I want to make very sure about when these migrations run. I want them to run at this point before web security tries to initialize the database. If you're not using forms authentication or if you are using forms authentication, but not using web security, you wouldn't have to use this approach. You could just put some configuration in place that would run the migrations automatically. And the tools as we'll see in just a bit, will automatically add that configuration for us. I'll point that out when we get to it. But for now, I think we've reached the point where we are ready to deploy.

Deploying to IIS

When you're ready to deploy an application, you can right click on your web project and select publish. Visual Studio will figure out all of the files that need to be deployed to a web server, collect them all together, and push them there. All you need to do is tell it how to do that. The first thing you do is select a publishing profile. I'll create a new profile. I'll give it the name release because this will be my release mode build that I want to publish. And on the next screen, I can select a publish method. There's several different methods here. If you need to FTP the files to the server, Visual Studio can do that. If you need to just push something onto a file system, you already know the directory where the webserver will serve this application from; you can also do that. You can select web deploy, and just enter in a server name and the site that you want and it will contact the server and push your files over. You can do this with shared hosting providers on the internet. You can also do it on intranet servers if you're deploying with inside your company. The one catch to web deploy here is I would want to push to local host. And the problem is that in order to do this, I need to be an administrator. So I could re-launch Visual Studio using renams ( assumed spelling ) administrator and then right click and publish and be successful, but I'm going to show you a different approach. We're going to build a web deploy package. A web deploy package is ultimately just a zip file, and you can take the zip file and put it on a USB key or email it to a administrator at your company, and they will be able to take the web deploy package and copy it to a server, and install it for you. In this case, we'll be doing the installing. Let me create a package -- let me create a directory under dev. We'll call it Ode To Food release. And I need to include .zip in the name so release.zip. And actually, let me put the zip file inside of a folder called release. And then I pick the site where I want this to be deployed. So it could be something like default website/OdeToFood, but I actually want this in the root directory of the website. So if someone just goes to local host, they will see this application running. So the site I wanted to deploy it to on this IIS server is default website. And that's actually just a parameter that you can change when you deploy it. But now, if I click next, the next screen will go through what sort of configuration do I want it to build in; release or to bug. We will deploy in release mode and I can also pick a connection string to use for default connection. Remember, right now default connection is the connection string that's in the web.config that's pointing to local DB. When I deploy this to IIS, I want it to point to that sql express instance that I just installed. So I will say that the default connection should actually point here. The server name will be dot/sqlexpress. I will use Windows authentication to connect. We'll see how that creates a slight hiccup, but we'll work around that, and the database that we will use; I will call it OTF production just to make this dramatically different. And select okay. We can say yes, go ahead create this database for me. And when this application is deployed, that connection string will be substituted into the default connection, and replace whatever we are using there for development. And then this checkbox here, execute first code migrations. That's where I was telling you if you select this checkbox, Visual Studio will also add a bit of configuration to your web.config file that will automatically run into the framework code first migrations. But we're actually going to explicitly run the migrations in our code, so I'm going to leave that unselected. Then there's a preview screen, which will tell you, you are about to create release.zip. And let me just click publish. All the files will be packaged up together. And what I can do now is open up a command prompt; make sure I right click it and say run as administrator because I will need administrator privileges to actually get this installed into IIS. And then come into that directory where I placed the zip file, and inside of here I'll find the zip file. I'll also find a command script that I can run that will do the installation for me. And if I run it without any parameters, it will pop up a little help and notepad that will tell me there's two ways to run this. You can pass the /T to test the deployment and simulate what would happen, and it will show you any errors that might occur. Or, you can do a /Y to just go ahead and do the full deployment. You can also point it to a remote computer, pass in a username and password; all sorts of additional flags you could use. Let's see what happens if we run this with a /T. I don't see any error messages, so this just might go through if I run it with a /Y. And now, if I come back into internet explorer, where we used to see an IIS 8 banner page, if I refresh, we'll see our application trying to make a connection to the database, but it can't quite get there. It doesn't have permission to do it. And this is because the IIS worker process that we looked at earlier W3WP.EXE runs under an identity known as IIS app pool default app pool. And this user account doesn't have any permissions in the database or on the server at all. When we're using local DB for development, we're administrators for local DB and IIS express is running under our identity, so we can do anything with local DB. But to get this application running with a production database, I need to give some permissions to that default app pool account. So I'll open up sql server management studio, and under security there will be logins. What I want to do is create a new login for that account. So the account is IIS app pools/default app pool. And I'll come over into user mapping, and say please give this login the ability to get into OTF production as a user, and give this login the ability to read data, to write data, and to execute DDL. Essentially execute things like create table. The entity framework migrations will need to modify the schema. We'll need to be DDL admin here. And now click okay. And now come back and refresh the browser and our application is now running under IIS 8. We have our database in place; we have our C data in the database. I should be able to log in, create restaurants, create reviews, and use all the features that we've built into this application.

A Second Deployment

Now once you've completed all this work and you're done with the single deployment, subsequent deployments are much smoother. For instance, I'm looking at the home page. I'm seeing it's displaying mail.server.com on the home page. I don't necessarily want that there. That was just an experiment we were doing to make sure that we could read the configuration file that was in the home index view. Let me open up and remove that, save the view, come back out to the project, and say publish. It has already selected by release package that we created earlier. So let me go ahead and click publish. That will rebuild the zip file. I can come out here to the command line, re-execute to the deploy. Notice that it's smart enough to understand what's on the server; what's new, what's changed. It only copied over the files that needed change. And now if I refresh the application, then that text is gone and we just made a small tweak to the deployed version of the application. And let's just see what's happening behind the scenes. If I open up the IIS manager, I'll be able to come into the default website, and switch over into content view. That actually shows me the files that are sitting out here. It does need to be refreshed. And now I can see my views folder, my scripts folder, my content folder with site.css inside of it. You might be wondering what happened to the controllers folder. This all looks like exactly like it did in my solution except there's some things missing. Well the controllers folder is all C sharp code. Visual Studio compiles all that code into a DLL. Remember I said that when you build a web application you produce a DLL, and that goes into this bin folder. Here we can see OdeToFood.DLL that will have all the controller code compiled inside of that assembly. We also have a lot of other assemblies. These are all things that we reference and that we need to run. Assemblies like DotNetOpenAuth, which as I described in the security module is what helps users log in using OpenAuth and OpenId. And when a request arise or the default website -- aka local host -- IIS will spin up that W3WP.EXE process, load our assemblies into it, and start sending us requests to process. That's when MVC takes over, calls into our controller, which renders a view. That's how the application all works.

Deploying to Windows Azure

Now that we've deployed to IIS on a local server, let's deploy to IIS running in the cloud with Windows Azure. To get started, I'll to go the Windows Azure website; WindowsAzure.com. And if you haven't heard of Azure, it is Microsoft's Cloud platform where you can host websites, databases and virtual machines. You can run Windows or Linux, and you can start small and scale up really big just by adding more resources for Microsoft's global data centers. If I want to put a new website into Azure, I would first log into the Azure portal. This is currently in preview, but by the time you watch this video it might be live. I'll first need to sign in with my live ID. ( Silence ) And the first screen that is about to come up will be a screen where I can see everything I have deployed in Azure under this account. It's my portal where I can add new databases, new websites, new virtual machines, new media services. It's also where I can manage my existing services, and where I can create new services, which is what I'll want to do when this is finished loading. We want to add a new website and give the website a database for storage. And you can see I have a few things out there already; couple websites, couple databases, a virtual machine, a storage account. But I want to create a new website with a database to host Ode To Food. First I'll need to give my website a URL. What will happen is that my website will be OdeToFood.AzureWebsites.net. I can also get a DNS and point this to my website in Azure, so it could just be OdeToFood.com; but we'll leave it as AzureWebsites.net for right now. I get to pick a region for the data center. I'll pick the eastern United States. I want to create a new sequel database. My connection string name -- let's call it OdeToFoodDB. And in the next step I'll need to specify my database settings. This is the name that I want to use. The server will be a sql server that I already have in Azure; I just need to log in properly. And I can show you what the advanced database settings look like. This is where I can select my addition wherever business The maximum database size one gigabyte will be good enough to start, and my collation. And with all that in place I can select okay, and Azure will go off and start creating this website. As soon as it's done creating that I'll have a website on the internet that I can go to. It'll have just the blank Azure template. And what we'll need to do is publish our website now, and set it to local IIS to this website. There are a number of publishing options available with Azure websites. You can see them in the list here. You can use GIT, you can use TFS, you can use Visual Studio or WebMatrix. We're going to be using Visual Studio; it's very simple. First I'll go to this website. This brings up the dashboard where I can look at things like how much CPU has been used by the website over the last 24 hours or over the last week. But it's also where I can download a publishing profile. Remember when we were publishing to IIS we created a publishing profile. This is going to download one for me that I could just plug into Visual Studio, and it will have all the settings that it needs to upload my website here. And we just downloaded it into my downloads folder. And now, let me go into Visual Studio, and let's publish this website again. I'm going to go to the profile tab, because I want to import this new profile. And there it is; the publish settings file. If you open that up, it's just a big scary amount of XML inside, but it has everything that I need to get this to the right server. So it already includes the URL, the site name, the user name, passwords. All you need to do is -- I can validate the connection; make sure this is going to work. And that looks good. So let's go to the next phase where you can see we're going to again deploy a release mode build. It is already understood that my default connections string will probably want to point to that sql server in the cloud. It will have populated the server name and the user name and the password, and everything I need. We're not going to run the entity code first migrations. We're taking explicit control of our migrations in the code. I'll just go to next. This allows me to preview what it is going to push out. But I'm feeling lucky, so I'm just going to click publish. Since this is going over the internet, it might take a little bit of time. The first time you need to publish it needs to push out a lot of assemblies, a lot of data going over the network. We'll come back in a second when it's finished pushing everything up into the cloud. And the publishing operation has just succeeded. I should now be able to go back into the browser. Let's refresh this website. And I'm now running on AzureWebsites.net, very simple to deploy. All my functionality should be there. The database is working. We see the database. That means I should be able to log into this application. ( Silence ) That was successful. I should also be able to go and create new restaurant at this point. And since I just launched, I don't expect a lot of traffic. The wonderful thing about ager is that this is now in someone else's data center. They make sure the electricity stays on. They're making sure they take backups at all the right time. And as this grows in popularity, I can come in and scale this thing out. I can reserve my own machine, crank up the number of instances, so I have more instances available to serve up all the requests that are coming in. And of course, you'll pay for the resources that you use, but hopefully you'll be making money from all of the customers that come to the website, and that will pay for everything that you need in Azure. ( Silence )

Good Bye, and Good Luck

This concludes this module on configuration and deployment where we looked at all the configuration files that are in effect when running a web application, and we also deployed to IIS both locally and in Windows Azure. This also concludes my ASP.NET MVC 4 course. We've seen how to work with routing, controllers, and razor views. We added javascript and used jQuery and jQuery UI to add Ajax features to the application. And we also saw how to work with foreign authentication and the membership provider that comes with our MVC application. I hope you enjoyed this course and were able to use the videos to either jumpstart your MVC development or solidify your existing knowledge of the framework, just a reminder that all the code that we've been working on is published on codeplex. If you look at the history of the check-ins you'll see I did one check-in for each module. So if you want to go back in time to a specific point and work along with the video, you should be able to do that. And now that we are finished I'd just like to say thanks for listening, and good luck with your software development endeavors.