Introduction

Introduction

Angular is great for building frontend client-side applications to interact with the user, but most full-feature web applications also need backend services to handle data management, user authentication, and other processing. Welcome to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this course focuses on the touch points between an Angular frontend client application and a backend service built with ASP.NET Web API. A touch point in this context is any feature that requires a frontend Angular application to touch a backend service. The Angular application reaches out to the cloud or in-house server with a request, and the service reaches down with a response. The key touch points we will cover in this module include retrieving data. Most web applications need data, and if that data is coming from a database or other data source on a backend server, that data is often exposed with a backend service. We'll look at how Angular issues a request for data and how the backend service responds to that request. I'll relate a touch point as filtering, shaping, and querying data. Oftentimes, the web application does not need all of the data provided by a particular resource. Imagine going up to Amazon and requesting a list of all products. Yikes. Instead, the client application can request a filtered list, like only gardening tools, for example. The client application could request specific data elements, thereby shaping the data, for example, providing only the product name and price. The client application could even use a structured query language to query using very specific and powerful criteria, providing the list of garden tools that range in price from $5.00-$10.00 and are currently in stock. We'll cover how to use query strings, URL parameters, and OData to filter, shape, and query data. Many web applications allow the user to update existing data or create new information. Those changes need to be saved to the database on the backend server. We'll look at how to pass new or updated data from the Angular application to the backend service for saving. And what if something goes wrong? What if a requested product is not found on the server? What if the service generates an exception? What if invalid data is passed to the service? We'll look at the touch points around handling error conditions. And, of course, we need to talk about security. The Angular application will collect a user ID and password from the user and pass that information to the backend service for authentication. We'll detail how to set up for authentication, both in the client-side Angular application and in the backend service. Once the user is authenticated, we can implement user-authorization techniques. In this course, we'll cover each of these touch points from front to back, but first, a few things to help you along the way. Thoughts, comments or question as you watch this course? Feel free to use the discussion tab to leave your feedback. Though you may not see all of these options shown here, you can find the link to the discussion on the right side of the Pluralsight page for this course. I check the course discussion page often. Want to code along? I have set up a Github repository for this course. It's called AngularF2B, for front to back, WebAPI. It contains the start files for this course to help you code along with the demos. I'll also add notes and reference materials as needed. With that, we're ready to look at the course prerequisites.

Prerequisites

This course assumes you have some experience with Angular and related technologies, such as JavaScript HTML, and CSS or cascading style sheets. You don't need to be an expert in any of these technologies, but have a working knowledge of each. If you are new to Angular, you may want to first check out an introductory Angular course, such as "AngularJS: Get Started", in the Pluralsight library. Or the "Shaping up with AngularJS" course available from the Angular website. The Shaping up course is very understandable, and fun, and it includes an interactive platform that gives you hands-on practice after each topic so you can learn it by doing it. If you have passed out of the getting-started stage and want a course that walks you through building a full-featured Angular application, check out "AngularJS Line of Business Applications". The "Line of Business Applications" course takes you step by step through building a Line of Business application. It includes accessing data, routing, building data entry forms, validation, exception handling, and more. With its focus only on the Angular side of things, the Line of Business course used mocking to retrieve and save data. The sample Angular application used and extended in this course is similar to the one built in the Line of Business course, without the mocking, of course. So if the Angular demos in this course seem too fast, check out the Line of Business course first and then come back. With the prerequisites defined, let's look at the outline for this course.

Course Outline

The focus of this course is on the touch points between the frontend client application and the backend server. Before we can examine the touch points, we need to look at the architecture and anatomy of both the Angular app and the ASP.NET Web API service. Next, we'll cover the most common touch point, retrieving data. We'll look at how to build a simple ASP.NET Web API service for retrieving data, and the Angular code required to call that service and process the result. Before the Angular application can successfully communicate with the ASP.NET Web API, we need to look at two additional concepts, CORS, or cross-origin request sharing and Web API serialization formatters. Next we'll take a look at passing parameters. We'll cover several options including query strings and URL parameters. We then cover the awesomeness that is OData. We'll define what it is and why you might want to leverage the power of OData queries. Up to this point we've focused on retrieving data, but most applications also need to save data. We'll look at how to pass data from the Angular application to the Web API Service, and how to process and respond to save requests. Next, we'll look at what to do if we get an invalid request, missing or invalid data, or an exception. We'll cover action results, error handling, and validation in the Web API, and how the Angular code can recognize and take action based on both success and error responses from the Web API. One common touch point between the front and the back is user authentication. This is such a big topic that it is divided here into two parts. Part 1 looks at user authentication from the point of view of the Web API. We'll discover how to use the Web API to authenticate a user. Part 2 covers user authentication from the Angular app's point of view. We'll demonstrate how to build a login form, register a new user, and login an existing user. Next we'll look at user authorization. We'll see how to restrict access to the Web API methods and how to grant access to protected Web API methods. The final module provides some final words, a brief review, and additional resources. Now, let's get started.

Getting Started: Angular & Web API

Introduction

Like chocolate and peanut butter, Angular and Web API are two technologies that are great together. Welcome again to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module gets us started building both the client-side Angular app and the server-side ASP.NET Web API service. In this module, we will review web application architecture. That will provide a context for building the sample application. We will create both the Angular app and the Web API service projects. We'll add code to these projects throughout this course. With the basic code in place, we'll examine the anatomy of the Angular application. We'll walk through how the Angular app is structured, and how the pieces fit together, and we'll examine the anatomy of the Web API service. We'll look at the code that is generated for us by the ASP.NET Web API templates. When we are finished with this module, we will have a working sample Angular application, and a start to our ASP.NET Web API service. The sample application used in this course is called Acme Product Management, or APM. The first iteration of this application displays a set of products. The Angular application requests the list of products from the Web API. The Web API Service retrieves the products from a data store and returns them to the Angular application, and the Angular application displays them. As we progress through the course we'll add features to this application, and later we'll build an edit form. Let's begin.

Web Application Architecture

A common application architecture for a web application involves using Angular in the frontend and ASP.NET Web API as the backend service. It all starts in the browser. A user navigates to a page of your application, and the request is routed to a server. That server holds all of the HTML and JavaScript files for the Angular client application. The server then serves up the index.html file and any other resources that the index.html file defines, such as HTML, CSS, and JavaScript files. The Angular application then executes within the context of the user's browser. Since it is running inside the browser on the client, Angular applications don't have direct access to corporate resources, such as a database. Instead, the Angular application needs to obtain everything it requires from a remote web service. The Angular application communicates with that web service using a meaningful set of URLs, so when the Angular application needs data, user authentication or other server-side processing, the Angular application submits a request. That request is routed to a server and processed by the ASP.NET Web API service running on that server. If the request is for data, the Web API retrieves that data, potentially from a database, and returns it to the Angular application, which then displays or processes the returned data. That data is often provided in JSON or JavaScript object notation format. This course focuses on these touch points, submitting requests from a frontend Angular application to a backend service for data, user authentication or other operations, and returning an appropriate response from the service. Why these technologies? Why use Angular for the frontend client application? Why ASP.NET Web API as the backend web service? Let's answer the why Angular question first. Angular is currently the framework for building responsive client-side web applications. It's neat integration with HTML results in an expressive syntax that make the HTML pages easy to read. Its powerful data binding and validation features make display and editing of data a snap. Its modularity emphasizes clean programming techniques, resulting in code that is easy to read and maintain, and it's easy-to-use services make complex tasks, such as retrieving data, a breeze. Bottom line, Angular is often the perfect choice for developing those full-feature client-side applications on the web. Angular can work with many different backend services, including Rails, NodeJS, and ASP.NET Web API. Why pick ASP.NET Web API over other web service technologies? Web API is built to work with HTTP. It is an ideal platform for building RESTful services. It provides actions that support the HTTP verbs, such as GET, POST, PUT, and DELETE. It supports a broad range of clients, including desktops, tablets, and pones. It is built upon the .NET platform, making it easy to work with for anyone already using Visual Studio and the .NET framework. It provides built-in features for routing, action results, and more, and it's clean patterns and samples make ASP.NET Web API relatively easy to learn. If you already use ASP.NET MVC, ASP.NET Web API is similar, and you can apply much of what you already know to Web API. Even without prior MVC experience, anyone can quickly learn how to build a basic web service with ASP.NET Web API. There are several approaches to building an ASP.NET Web API service. You can build it as part of an ASP.NET MVC application, you can build it as part of an ASP.NET WebForms application, or you can build a standalone backend service, which is what we will do in this course. As a standalone service, it can readily be used by any client, including ASP.NET MVC, ASP.NET WebForms, Windows desktop applications, tablet or phone apps, and of course, Angular apps. So what code editor should we use? We can build an Angular application with any text editor, including NotePad, or we can use a more code-focused tool such as WebStorm or Visual Studio. The preferred tool for building an ASP.NET Web API service is Visual Studio. That's because of its built-in templates, IntelliSense, and debugging support. In this course, we will use Visual Studio for both the ASP.NET Web API service and the Angular app. Using one tool for all of the code will simplify code management and debugging. Now that we have looked at the architecture, technologies, and tools, let's build the projects.

Creating the Projects: Web API

Creating the ASP.NET Web API service project is easy with Visual Studio. Simply launch Visual Studio, select File, New, Project from the menu, and select ASP.NET as the template. Let's launch Visual Studio and walk through this process. Here we are in Visual Studio. to create the ASP.NET Web API project, select File, New, Project. You can use the tree view on the left to select the desired language. This course uses C#, but you can build a Web API project with Visual Basic using all of the same techniques that you will see here. Select Web, and we are given a single template called ASP.NET Web Application. That's the main project template for creating ASP.NET applications. We can give the project a name. We'll call it APM.WebAPI and specify a solution name. We'll name this solution just APM for Acme Product Management. Click OK. Another dialog is displayed that allows us to select how Visual Studio will generate the code for this project. We are building a Web API service, so we'll select the Web API from the list of templates. Down here we can select to have Visual Studio add a unit testing project for us. Unit testing is not covered in this course. If you are interested in more information on unit testing your C# code, check out the Defensive Coding in C# Pluralsight course. It has an entire module devoted to unit testing. For the sample application, we'll leave the unit testing unchecked, but you may want to check it when building your real applications. Over on the right side, notice that the template provides for generating code for user authentication. We'll use the default, Individual User Accounts, so that Visual Studio will generate the first cut of our user authentication code for us. We'll go through the authentication options provided by Web API, and talk much more about user authentication and authorization later in this course. We also have the option to host the Web API service in the cloud. In this course, we will run the client and the service through Visual Studio, so no need for hosting at this point, but you can deploy the resulting service to Azure or any other web server you desire. So we're finished here and we can click OK. Visual Studio then generates code based on our selections. When it is finished, it displays a Readme file, providing links to Help topics. Notice all of the folders and files that Visual Studio creates. We'll walk through this code later in this module. Now that we have the Web API project created, where do we put the Angular code?

Creating the Projects: Angular

When creating the Angular app, there are several approaches we could take. One option is to add the Angular code to the same ASP.NET project as the Web API code. This option assumes that the angular code and associated HTML templates will reside in the same location on the web server as the Web API service. Putting the Angular code into the Web API project can make it difficult to see which code is for the client and which is for the service. Another option is to add a separate ASP.NET project and put the Angular code in that project. However, running an ASP.NET web project compiles the project and builds a DLL that has no purpose. If you are combining Angular with ASP.NET MVC, then this is a good option for you. The third choice is to add a new website. A website allows you to define a set of files, such as JavaScript HTML, and CSS, and deploy them to a web server separate from the ASP.NET Web API. This is the option we will use. Let's add the website to our Visual Studio solution now. When we created our Web API project we defined a solution called APM. We can add the website for our Angular application to the solution as a separate website. Right-click on the solution and select add, New Website. We can use the tree view on the left to select the desired language. It won't matter which language we select, since the website will be developed using Angular, JavaScript, HTML, and CSS. Select the ASP.NET Empty Web Site template and select a folder for the Web Site. Let's put it in a folder named APM.Client, click OK, and Visual Studio creates the Web Site. Notice that since we selected Empty Web Site, Visual Studio did not create any code for us, only the config file, so we'll need to add the required JavaScript libraries manually. Before we start writing Angular code, the client application needs some JavaScript libraries, such as Angular, and any style templates we may want, such as Bootstrap. Since we are using Visual Studio, the easiest way to get these components is using the Visual Studio NuGet Package Manager. The NuGet Package Manager automates the process of downloading and installing packages for a website or Visual Studio project. We'll want to install the packages for AngularJS Core, which provides the core functionality for Angular, and Bootstrap, which provides some styling for the application. We will use Bootstrap in this course, but you don't have to know Bootstrap to follow along. If you want more information about Bootstrap, check out one of the Bootstrap courses in the Pluralsight library. Now let's jump into Visual Studio and get these packages. Here we are back in Visual Studio. To add packages to our website, select it in Solution Explorer. Right-click and select Manage NuGet Packages. Ensure you have the Online node selected on the left, and use the Search box in the upper right corner to search for Angular. We'll select AngularJS Core to download only the core functionality we need. Click Install to install it. Notice that it adds a Scripts folder and some files to the website in Solution Explorer. Next use the search box to locate Bootstrap. We'll select the first one here and click to install it. You can see it adds more files to the website in Solution Explorer. With that, we can click on Close. We now have the files we need to get started. Our Visual Studio solution includes a website for the Angular application, we called it APM.Client, and an ASP.NET project for the Web API Service. We called that APM.WebAPI. Now we are ready for some code.

Anatomy of an Angular Application

To ensure we are all on the same page, let's do a quick review of the anatomy of an Angular application. An Angular application is comprised of one or more modules. In our sample application, the main module is named productManagement, and is represented in this diagram with the tan rounded rectangle. In many cases, an Angular application has one HTML page. By convention, this page is called index.html, but it could be called anything. The index.html page normally contains the links to the CSS, or cascading style sheet files that define the styles for the application. These are normally defined in the head tag and are not shown here. Also in the index.html file are the script tags for the JavaScript files required by the application. These are often defined in the bottom of the body tag and are also not shown here, and of course, the index.html file contains any html that is displayed in all of the views, such as a header, menu or footer. The Angular ng-app directive is normally specified on one of the tags here in the index.html file. The ng-app directive specifies the name of the main module, in this case, productManagement. The HTML required for the other views are in HTML template files, such as the productListView shown here. In this example, we are pulling this template into index.html using the Angular ng-include directive, but in most Angular applications, templates are defined as part of the routing, and use an ng-view tag here instead. Since routing is not directly related to the touch points, we'll use this ng-include directive to display our View template. Template files often have an associated controller that provides the data and logic for the view, such as the productListCtrl shown here. The controller manages the data model for the view, and contains the methods required by the view, such as processing button clicks. If these concepts do not seem familiar, or the upcoming demo feels like it is going too fast for you, consider going through the AngularJS Line of Business Applications Pluralsight course first. The Line of Business Applications course takes you slowly, step-by-step through building an Angular application similar to the sample application we are using here, and it includes details such as routing and linking. Now let's take a look at each of these files. Here we are back in Visual Studio. Since the last demo, I have added the code for the first page of the sample Angular application. If you want to code along, you can get these startup files from my Github account, DeborahK/AngularF2B, front to back, WebAPI. Look in the folder called startFiles. Here in our solution is the index.html file containing the HTML page for the application. I added an app folder, and in that folder is the file for the main module, app.js. Within the app folder I added a products folder. That folder contains the view, our productListView, and the controller, productListCtrl. Before we look at the details of this code, let's run it. Set the APM.Client website as the startup project, right-click, Set as Startup Project, then Run. You can see that our sample application is currently a simple display of products in a list. Now let's take a look at the code. We'll start with the index.html page. This is the main HTML page for the application. It contains the links to the CSS here in the head tag. This sample application uses Bootstrap for styling. The JavaScript files required by the application itself are here at the bottom of the body tag. The scripts include Angular, the main module for the application, that's in app.js, and the first controller that is located in productListCtrl.js. In this example, this file contains the HTML for a navbar and an include directive to display the first view. Moving on the app.js file, this file contains the definition of the main module of this application. This main module is called productManagement. Currently this module does not have any dependencies, as is indicated by the empty array as the last element here. Next, let's look at the productListView.html file. This is a template, and as such, it does not contain the standard HTML head and body tags. It is meant to be displayed within another HTML page using ng-include like in the sample application, or when using Angular's routing features. The main div element contains an attribute defining the controller associated with this view. The as vm syntax makes use of the controllerAs feature of Angular that was introduced in version 1.2. vm is the alias name. It can be any name, but often vm is used as a shortcut for ViewModel. This controller as syntax simplifies working with the $scope and accessing the data in the page. In the view, the model and methods are referenced using that alias, such as vm.products here. The html uses the ng-repeat and Angular's data binding to display each product in the list. And lastly, let's look at the controller. This code first registers itself with the productManagement main module we defined in the app.js file. It then defines the controller function, again taking advantage of the controllerAs syntax. Using this syntax, we don't need to pass $scope into the controller. The models and methods are defined directly on the controller itself through this. Here we are hard-coding the set of products to display in the view. Notice that the products are defined in JavaScript object notation or JSON. In the next module, we will work on the first touch point with the Web API Service, and retrieve this data from the service instead of using this hard-coded data here in the client, but for now we are hard-coding the data so we can run the sample application prior to working on the Web API project. Again, if this code does not look familiar or it feels like this demo is going too fast for you, consider going through the AngularJS Line of Business Applications Pluralsight course first. That course takes you step-by-step through building an application similar to the sample application we are using here. So with that, next up is the backend service.

Anatomy of an ASP.NET Web API Service

Now let's take a simplified look at the anatomy of an ASP.NET Web API Service. Our request comes in, and it's routed to the appropriate code based on the request URL and action verb, GET, POST, etc. The URL is normally of the form api/some controller name/an optional identifier. For example, api/values/5 identifies that values is the name of the controller, and 5 is the identifier. The route or routes are normally configured in a file within the Web API project called WebApiConfig. We'll look at that file in a few moments. The routing looks for a controller, which is a class that inherits from APIController and is named with a defined controller name. In this example, the controller name is Values, and by convention is suffixed with Controller. So using standard routing, the api/values/5 URL will route to the valuesController class. Alternatively, the routes could be defined using attributes on the Controller class. To keep things simple, we won't use attribute routing in this course. The routing uses the action verb from the request, GET, POST, PUT, or whatever, and any parameters to locate the appropriate method in the controller. For example, a GET request with a product ID on the URL will route to this GET method. Alternatively, you can use attributes on methods to associate the method with an appropriate action verb, but in this course we'll stick with the basic routing by convention. The action method then returns a value, which is included in the response returned from the Web API service. Earlier in this module, we created the ASP.NET Web API Project. In doing so, Visual Studio created a whole bunch of code for us, including a sample route, controller, and set of actions. Let's take a look at what it produced and how it matches with this diagram. Here we are back in Visual Studio with the Web API project opened. Unlike the Angular project, I have not added any code to this project. All of the code you see here was generated automatically when we created the project earlier in this module. Let's start by looking for the WebApiConfig.cs file. It's located in the App\_Start folder WebApiConfig. Here we can see that it configures a default route, api/controller name/id, and it defines that the identifier is optional. Notice that code here also sets up the authentication. Let's comment that out for now until we get to the Authentication module later in this course. That way we'll be able to run and see our API working. Now let's look at the Controllers folder. We can see that there are three controllers that were generated for us. One is an AccountController for the user account required by the user authentication features. We'll look at that code in more detail later in this course. One is a HomeController. Let's open that one. You'll notice that it inherits from Controller and not ApiController. This is an MVC controller used to display a home page. We'll see that page in a moment when we run this application. In most cases, a Web API service does not need a user interface, and you could delete this controller and associated home page. The controller we are interested in is the ValuesController. Let's look at the code for this controller in detail. First, you can see that the controller inherits from ApiController so it is accessible as part of our Web API. With the routing that is currently defined in the WebApiConfig, a client application can use api/values to access the methods in this controller. The methods in this controller map to the action verbs. Get to get a set of values, Get with an id to get a single value, Post to receive a value within the body of the request, this currently has no code, Put to update a value with a defined ID, and Delete to delete the defined value. The controllers that we create for our API will look similar to this ValuesController. Notice at the top that there is an Authorize attribute here. Let's delete that for now. We'll get back to that when we look at the authorization later in this course. Since Visual Studio generated so much code for us, there's enough functionality here to actually run the Web API project. Set the APM.WebAPI project as the startup project, then run the application. The generated Web API application displays a default, hard-coded home page. Normally a Web API does not have a user interface, but one was generated here to provide links to help developers build ASP.NET web applications. A Web API may have a help page that demonstrates how to use the API, and in this case, Visual Studio generated the code for the Web API help page. To display the help, simply click API here in the tab. Cool, this page displays the current API that is exposed by the generated ASP.NET Web API application. This first set of APIs is for Account. We'll look at the user account information later in this course when we cover user authentication. For now we'll focus on the Values APIs, which are shown here at the bottom. The Values API provides the sample GET, GET with ID, POST, PUT, and DELETE. These map to the methods in the ValuesController class. Click on one of the API examples to see more information. For example, clicking on this GET api/Values/(id) displays a page showing the expected parameters and response information. Let's give this a try. In our address bar we'll type api/values/ some id, let's just type in 5. We can see down here that we get the option to open the returned JSON file, so we can click to open it, and there is the value. Where did that value come from? Well let's peek back here in the code. We can see that the Get with an id simply returns the word value. So Visual Studio creates an example Web API controller for us, along with associated help pages and a default user interface for learning more about Web API. To review, routing defines how requests are routed to the controllers in the Web API, based on the URL and action verb. Each controller is a class that inherits from ApiController. Using the default routing, the class name is the name in the URL with controller as the suffix. Methods in the controller perform operations based on the action verb and parameters. That's the basic anatomy of an ASP.NET Web API Service.

Summary

Let's review what we covered in this module. First we reviewed basic web application architecture. This provided a context for building the sample application. Then we looked at how best to set up the projects. We decided to use the ASP.NET Web API project template so it could build the first cut of the code for our Web API service, and we added a new website for the Angular client application. Next we looked at the anatomy of an Angular application. We saw that an Angular application is comprised of modules, views, and controllers. The sample view simply displays a list of products that are currently hard-coded in the controller. Our task in the next module is to replace this hard-coded list with a call to the backend Web API service. Lastly, we looked at the anatomy of an ASP.NET Web API service. We saw that a Web API service is comprised of Controller classes with methods that map to routes defined with URLs and action verbs. Visual Studio generated sample code to give us a starting place with the API. Our task, then, is to build a product API with a Get method to return a set of products, which we will do in the next module. So that's the start of our sample application from front to back. Now let's examine our first touch point, retrieving data.

Retrieving Data

Introduction

The most basic touch point between an Angular frontend client application and a Web API backend service is the retrieval of data. An Angular application wants to retrieve data for display and the Web API can provide that data. Welcome to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module walks you through building a Web API service that provides data. We'll then modify the Angular app to request that data from the Web API service. At the beginning of this course, we identified the key touch points between an Angular frontend application and a backend service. In this module, we'll kick things off with a look at retrieving data, specifically, how an Angular application requests data, how a Web API service processes that request and returns the data in a response, and how the Angular application processes that response. Imagine that this is our Angular application running within the browser on the user's computer, tablet or phone, and this is the ASP.NET Web API running on a web server. The Angular application wants to display some data, such as a list of products. The basic process of retrieving data goes like this. The Angular application asks for the data by sending a GET request to the backend service. This request specifies the URL of the desired resource, in this case, the list of products. The Web API service routes that request to the appropriate controller and method. We know from the prior module that if we use the default routing features, this request will route to a class named ProductsController because of the products text in the URL. And since this is a GET request, it will route to a method named Get. Since there are no other parameters defined on the URL, the routing will locate a Get method with no parameters. The Get method then processes that request, retrieving the set of products from a productRepository. It then returns a response with the requested data, often in JavaScript object notation or JSON format as shown here. The Angular application receives and displays that data. This process gets more complex, of course, as we add parameters, queries, and security. Right now we'll stick to the basics. We'll cover more complex scenarios as we progress through this course. In this module we will start with the Web API service. First we'll build the model. The purpose of a server-side model is to define the shape of the data passed out of the Web API. We'll walk through the process of building a model for the product data. Then we'll build a repository. A repository separates out the logic that retrieves the data, keeping it out of the model and out of any business logic that acts on that model. Lastly, we'll build a Web API controller. We've already walked through the ValuesController that was generated by Visual Studio. We'll use similar techniques for building the ProductsController. With the backend service in place, we'll move to the frontend. Angular provides two different built-in services for communication with the backend service, $http and $resource. We'll compare these services and select one for communication with the backend service. Then we'll build a custom Angular service to call the Web API service. With that, we'll have the entire retrieve process in place from front to back. Let's get started.

Building the Web API Model

We'll start writing the code for the retrieve process by building the Web API model, but first let's answer three basic questions. What is a model in this context? Why do we need a model in the Web API? And what does that model look like? Let's take these questions one at a time. The purpose of a model in the Web API service is similar to the purpose of a model in an Angular application. It models the data. A Web API model models the data exposed by the Web API. It specifies the shape and elements of that data. For example, a model for a Web API that exposes product information could include product ID, product name, product code, and so on. A key use of a Web API model is to define the return type for the GET operations. Web API provides formatters that automatically serialize a model object to either XML or JSON for transport to the client. The model defines the properties to include in that serialized data. You may recall from the last module when we looked at the anatomy of the generated ASP.NET Web API service that Visual Studio didn't create a model for the ValuesController. If it didn't need one, why do we need a model? Well, the ValuesController GET operation generated by Visual Studio and shown here, returns an array of strings. If we wanted our ProductsController to simply return an array of product names, we wouldn't need a model either, but we want a set of product properties, such as an ID, name, and description, so we need a model to define the properties returned from the GET request. Now that we know we need to create one, what does a model look like? A model is simply a class. Properties in the class define each data item included in the return value of the GET operation. The name of each property here, such as Description and Price, are the names of the properties in the JSON returned to the Angular application. These are the names that must be used in the Angular data binding expressions. Now that we have the answers to our questions, let's create the model for our products in the Web API project. Here we are back in Visual Studio with the Web API project node open. Notice that Visual Studio already created a Models folder for us. In this folder are the models Visual Studio generated for the user account. We'll ignore those for now. We'll look at them in detail later in this course. For now, let's create a new model, right-click on the Models folder and select Add, Class. We'll name the class Product. We use the singular name of the class since the model represents the data elements for a single product. Next we'll define the appropriate properties for the Product class. We can use a snippet for this, prompt tab, type in the return type, string, tab, type in the property name, Description, enter, and we're ready for the next one. I'll just paste the rest. There we have it, our model is complete. Our next step then is to build the repository.

Building the Repository

Before we build the repository, we again have three questions. What is a repository? How is data obtained? And what does a repository look like? Let's answer these one at a time. What is a repository? A repository is a class that contains the logic required to retrieve and save data from one or more data sources. On a retrieve operation, the repository retrieves the data from the data store, it uses that data to populate objects with properties defined by the model. The result is a list of model objects. The repository provides that set of objects to the controller. The controller can then serialize the objects and return them in the Web API service response. By using a repository, the data access logic is separated out from the model and from the Controller. Since the repository contains the logic to retrieve and save the data, what that code looks like depends on how the data is obtained. If the data is in a relational database, such as SQL Server, Oracle or MySQL, the data could be obtained using ADO.NET, the data access framework provided by .NET. Or it could be obtained using Entity Framework, which is an object relational mapping, or ORM framework, for ADO.NET. The data could come from a text file such as a JSON or XML file, or the data could come from any other data source, such as a NoSql database. The nice thing about building a repository for the data access logic is that the controller does not need to know or care how the data is actually obtained. In this course we'll define a simple repository class that uses a text file as the data source. Using a text file as the data source makes it easier to implement the repository and try out these features without the need for a database. You can replace the code in the sample repository with ADO.NET or Entity Framework code later if you'd like. What does a repository look like? What the code in the repository looks like will really depend on your data access strategy. Minimally, it should provide a retrieve style method that retrieves the data, populates the set of objects from that data using the model, and returns those objects. And it should have a save style method that takes in a new or updated item, and updates the associated data source. So let's take a look at this in Visual Studio. Since our last demo, I've added two files to the project here. Product.json is a JSON file we will use as our data source. ProductRepository.cs contains our repository class. If you want to code along, you can get these two startup files from Github. My Github is DeborahK/AngularF2B, for front to back, WebAPI. Look in the folder called startFiles. Because the logic in the repository depends on your data access strategy, and here we are just using a text file, I'm not going to spend a lot of time walking through all of this code, but let's take a quick look. First I added a JSON file to the App\_Data folder. The data here may look familiar because it is similar tot eh data we hard-coded in the Angular application in the last module. This file is included in the startFiles for this course provided in Github, so you don't have to type all of this data in. Next I added a ProductRepository file to the Models folder. This file is also in the startFiles for this course provided in Github. The code in the ProductRepository class has several methods. The Create method creates a new product and sets server-defined defaults. In this case, we're setting the ReleaseDate to the current date. The Retrieve method defines the location of the json file, reads the data from the file, and deserializes that data into a list of products. The Save method has two overloads. The first overload is for saving a newly-created item. The second overload is for updating an existing item. And there is a writeData method that writes all of the data back to the JSON file. In a real application, your repository code will most likely use ADO.NET or Entity Framework to access the data in a database, but this JSON file technique will work for our purposes, and it's easy to get up and running without the need to build a database. Now that we have the repository in place, let's build the controller.

Building the Web API Controller

With the model and the repository completed, we can build the Web API Controller. As we saw in the last module, the Web API Controller inherits from ApiController. Its primary purpose is to define the methods associated with the action verbs, such as Get and PUT. For example, here is the generated ValuesController we looked at in the prior module. It inherits from ApiController and defines the methods associated with the action verbs such as GET and PUT. The method shown here defines the code for the GET verb, which is what we want in our Controller. That looks easy enough, let's give it a try. Here we are in Visual Studio again. First we'll find the Controllers folder that Visual Studio generated for us. Next we can right-click and select Add, Controller. We then have the option to add a Scaffold. This will automatically generate the first cut of the Controller code for us. We have many options when generating code for a Web API Controller. We can select an empty controller and build out all of the code ourselves. If we were using Entity Framework, we could select a controller with actions and Visual Studio will automatically hook it up to Entity Framework. We can select a basic controller with read/write actions. That's the one we'll pick. The last two have to do with OData, which we will talk about later in this course. So it already picked the one that we want here, we can just click Add. Next we define the controller name. By convention, the controller is a noun. It is often defined as a plural value and is suffixed with Controller, so we'll name ours ProductsController, click add, and we can see that Visual Studio generated a controller for us. Looking more closely at this code, we can see that it's basically a copy of the ValuesController content. We'll need to change this code to return our list of products. In this module we'll focus on the GET method. We'll look at these other methods later in this course. We want this GET method to return the entire set of products. We don't want to return an innumerable list of strings, so let's change that first. Change string here to Product. The error tooltip here gives us the option to add the using statement for the model's namespace. Select that tip, and the using statement is added for us. Next, we don't want to return this set of strings, we instead want to return the set of products. Where are we going to get that set of products? If you just said the repository, you are right. We just need to create an instance of the repository. Then we need to return the retrieved products. Done. Having our repository in place makes this code clean and easy. Since we don't yet have the code in the Angular application to call this service, we can use the help page to try it out. Ensure that the APM.WebAPI project is the startup project, ours is bolded here so it's set as the startup project, then run the application. Click API from the menu and scroll down. Ah, our new Product API is now at the bottom of this page, cool. We've only built a code for the first GET action, so let's try that. Type in /api/products, and you'll be given the option to open or save the returned json file. Let's open it, and there it is. It's not formatted for readability, but the data is all here. Cool, the first cut of our Web API back in service is now ready for consumption. Next up, let's see how we can call this now from Angular.

Angular Services for Calling a Web Service

So how does Angular communicate with a backend web service? Well, Angular provides an extensive set of built-in services, two of which can be used to call a backend web service, $http and $resource. Let's look at each of these built-in services in detail. The $http service is part of the core Angular services and does not require any additional downloads or components. It facilitates direct communication with the backend service and provides convenience methods for performing http requests, such as GET, POST, PUT, and so on. This example uses the GET method and defines the URL of the desired resource to get, in this case, a list of products. A call using the $http service is asynchronous, so the call does not immediately return the data. Instead, the call returns a JavaScript promise object. Use the then method, as shown here, to define a callback function. When the request is complete, the callback function is executed. The response object from the backend service is passed into this callback function. The data can then be obtained from the response object and assigned as part of the Angular controllers model. So how does this $http service compare with the $resource service? Before looking at $resource, let's talk for a moment about REST. REST stands for REpresentational State Transfer. REST is an architectural style for building scalable web services. In very simple terms, REST implies that requests and responses involve the transfer of resources. Those resources can be HTML pages, images or sets of data. Resources are identified with a URL. For example, the resource identified by this URL is a set of product data. Requests utilize a standard set of HTTP verbs, GET, POST, PUT, DELETE, and so on. There is a lot more to REST than just this. It has a detailed definition and set of constraints that are beyond the scope of this course. For more information on REST, check out the "REST Fundamentals" course on Pluralsight. The important point about REST for our purposes is that ASP.NET Web API builds RESTful services, so we can use the Angular $resource service to communicate with Web API. So what is $resource, and how do we use that service? $resource requires downloading and referencing an additional Angular component called angular-resource. $resource is an abstraction on top of $http for calling RESTful services, so it abstracts away much of the complexity of communication with a backend RESTful service. It requires less code. The function shown here uses the $resource service. The parameter to $resource describes the URLs that communicate with the RESTful backend web service. The part of the URL prefixed with a colon defines an optional parameter, so this resource defines a URL for products, and if the optional id is included in the URL it refers to a single product. This one productResource function handles multiple verbs such as GET and POST. The resource object returned from this function provides a built-in set of methods that perform a default set of actions. That includes a query method, as shown here, to retrieve an array of data, such as a list of products. There is also a built-in get() method to get a single resource, such as a single product, and a built-in save() method to save a specific resource. We'll see those two methods later in this course. Because our Web API service is RESTful, we'll use $resource in our sample application. So next up, let's add this $resource code to our Angular application.

Calling the Web API from Angular

So what do we need to do to call the Web API from Angular? In the sample Angular application that we built earlier, it is the productListView that displays the product data. The elements in the productListView are bound to product properties provided by the model that is defined in the productListCtrl. Both the productListView and productListCtrl are associated with the main module that we called productManagement. This part of our Angular application is already in place, and now we also have a Web API backend service that is ready to call. We could add code to the controller to call the Web API directly, but that wouldn't be very reusable, and it would not keep our controller logic separate from the data access, so let's create a separate Angular module and call it common.services. We'll register all of our common code in this new module, and we'll build a reusable custom Angular service called productResource. That custom Angular service will use the built-in $resource service to call the Web API. So then when the view is displayed, the associated controller is loaded, the controller uses the productResource service and calls the query method of a $resource object. The productResource service issues an asynchronous GET request to the Web API service. The Web API service routes the request, retrieves the data, and sends a response back to the productResource service. When the productResource service receives the response, it returns the response to the controller. The callback function in the controller is then executed, assigning the returned data to the controllers model. The data bindings in the View catch the change to the model and populate themselves using the model data. Now that we know what we need to build, let's get started. Here we are back in Visual Studio with the Angular project node open. First we need to download the JavaScript library containing $resource. Right-click on the website and select Manage Nuget Packages from the context menu. Use the search box in the upper right to search for Angular, and here is the module that we need for $resource. Click Install to install it. That's it, so click Close. And before we forget, let's add that file to index.html. The NuGet package put the component in the Scripts folder, so here is angular-resource. So to create the script tag, we can simply drag that and drop it in. Just be sure to add it after the reference to Angular itself. Next we will create the new module. Let's start by creating a new folder. Right-click on the APM.Client node and select Add, New Folder. We'll name it Common. Then we'll add a new JavaScript file to that folder for our new module. So we'll right-click, Add, New JavaScript File, and we'll call this file common.services. We now have another new file that we need to define in index.html. In this new file we will create the module using code similar to the code we used when we created the main module. Here we defined a new module called "common.services" and set a dependency on "ngResource". ngResource is the name of the module containing the $resource service that we just downloaded from NuGet. Let's add one more thing here. Let's define a constant for the path to our server where the Web API is hosted. This code defines a constant called "appSettings". This constant has a property named serverPath, which is the path to the hosting server for the ASP.NET Web API. Currently it is set to the hosting server used here in my development environment. You will need to replace the port number here with a port number appropriate for your ASP.NET Web API. How do you determine that? Go to the APM.WebAPI project and double-click on the Properties node. Pick the Web tab, and you will see your port setting for your Web API here. What is defined here in the project URL is what you should use for your serverPath. That's the URL for accessing your Web API in the development environment, so the number that you have here should be the same number that is defined here. Before you deploy your Web API service, you'll replace this path with the URL of your production server. Now we are ready to define the custom service that communicates with the Web API service. Why did we define a custom service here? If you said reusability and/or encapsulation, you are right. We'll put the service in the same Common folder, right-click, Add, JavaScript File, and we'll name it productResource. Now that we have yet another new file, we need to define it in index.html. In this new file we'll insert some code. This code first looks up the common.services module and registers the new custom factory service with that module, and here is the factory function. The Angular $resource service is injected in as a parameter to this function. We also need the appSettings constant so we can access the defined serverPath. The function calls the $resource function, passing it the URL to request the products. The URL is the serverPath + /api/products. The :id here denotes an optional parameter. We'll use this parameter later in this course. And this function returns the $resource object, that's it. Simply by returning the $resource object with the appropriate URL defined, we have set up for communication with the Web API service. The last step then is to use this new data service from our productsController and display the resulting products on the page. The first thing we need to do is to go to app.js. Here we need to define a dependency on the new module we just created, so we're going to define common.services as a dependency here. Next we can modify the productListCtrl. We want it to use the productResource service to retrieve the set of products, so we want to delete all of these hard-coded products from here. To use the productResource service, we need to have Angular inject in a reference to the productResource service, so we do that by passing it as a parameter. And since we now have a parameter here, we need to change the controller method to pass in a min-safe array. And the first parameter of that min-safe array needs to be the productResource. Now we can add the call to the query method of the productResource, so we'll call .query. The query method expects to receive an array from the Web API. We want a list of products, so we expect to receive an array, and we'll assign that array to our products property. That's it, now our controller will call the query method of the resource object, which in turn will send a Get request to the URL we defined. It will contact our Web API and return a JSON array containing the list of products. When we assign that list here, it should appear in our UI. Cool, let's run it. Spoiler alert, our code won't work quite yet, but let's see what happens. First we want to now run both of these projects. We need the Web API service running to receive requests, and we need to run the Angular client app. Luckily, Visual Studio can do this. First we need to adjust the APM.WebAPI properties, so we'll go back to the Properties tab. We'll set the start action to Don't open a page. This will start up the Web API without displaying that generated Readme style start page, then right-click on the solution and select Set Startup Projects. Here we can set multiple startup projects, so we can start both the client and the Web API. With that in place, we can now click Run. By default this is using Internet Explorer. Hmm, it looks like the code displays the rows, but the rows have no data, so our Angular code must be getting something back. Don't worry, we'll fix this. And if you don't get anything at all, we'll fix that too. What if we try it again using Chrome? Well, here we don't get anything. Let's press the F12 tools, ah, we're getting an error, No 'Access-Control-Allow-Origin' header is present on the requested resource. It looks like we have two issues here. The issue that we're seeing in Chrome is a problem with something called CORS or cross-origin request sharing. Chrome won't allow us to call our Web API, because the origin of the request, basically our client application, is on 52436, and the Web API is on 52293, so the origins don't match. The problem with Internet Explorer, however, doesn't seem to be the same cross-origin request issue. That's because if only the port number is different, Internet Explorer does not see that as a cross-origin request issue. Internet Explorer makes it easier to work with Web API in the development environment, but we're still not seeing the data here. I'm betting it's a problem with the data serialization formatting. We'll dive into both of these issues, both the CORS and the data serialization, in the next module, and we'll get our retrieve working.

Summary

This module focused on the most basic touch point between an Angular frontend client application and a Web API backend service, retrieving data. But should we really check this off at this point? We don't yet have a working application. Hmm, so let's wait to check this until it actually works. Even though the application does not yet work, we accomplished a lot in this module. First we implemented the Web API Get functionality. We built a model that defined the shape of the data passed out of the Web API. In this example, the model defined the properties for a product, such as product ID and product name. We built a repository to encapsulate the data access logic. Often the code in the repository involves ADO.NET or Entity Framework code, but in our example we just accessed a text file containing the product data and JSON or JavaScript object notation format, and we built a Web API Controller with a Get method for retrieving a set of products. Then we shifted focus to the Angular client application. First we compared the $http and $resource surfaces for accessing a Web API from Angular. Because our Web API service is RESTful, we selected to use $resource. Then we built a new custom Angular service that uses $resource to call the Web API, and modified the productListCtrl to use that service to retrieve product data. When we tried to run the application, we either couldn't see the data or received a cross-origin request error. In the next module, we'll cover cross-origin request sharing, or CORS, and serialization formatters. With both of those things in place, our application will run successfully.

CORS and Formatters

Introduction

In the last module, we built retrieve functionality into our Web API service and modified the Angular code to call that service, but the application does not run successfully. There's more work to be done. Welcome once again to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module covers Cross-Origin Request Sharing or CORS, and Web API serialization formatters. Once we understand both of these concepts, we can add the required code to the Web API and get our application running. Recall from the last module that when we run our sample application with Internet Explorer we get this. We can see rows, so we must be getting data, but something is wrong because we don't see that data. When we run Chrome, we don't see anything. If we press F12 to bring up the developer tools, we see an error, No 'Access-Control-Allow-Origin' header is present on the requested resource. So what does that error mean, and how do we fix it? We'll answer those questions in this module. The primary objectives for this module are understanding CORS, enabling CORS in a Web API service, understanding serialization formatters, and configuring the JSON formatter in a Web API service. Let's get started.

Understanding CORS

As defined on Wikipedia, CORS or cross-origin resource sharing, allows many resources on a web page to be requested from another domain outside the domain from which the resource originated. In other words, CORS defines a way that a browser application can interact safely with a service that originates on a different domain, or even a different server. Hmm, let's look at that with a diagram. In this first diagram at the top, the browser is on the left and the server on the right. In this scenario, all of the files for the Angular application and the web service are deployed to the same server and in the same domain. When the browser requests a web page, the HTML, JavaScript, CSS, and any other page resources, are downloaded from the server to the browser. If the Angular code issues a request from the service, the web service processes that request and returns a response. In this case, the origin of the client application and the origin of the requested resource are the same. The second diagram is similar, however, the files for the Angular application are deployed on a different server or different domain on the same server than the web service. In this case, when the browser requests a web page, the HTML, JavaScript, CSS, and any other page resources, are downloaded from the first server to the client, but if the Angular code issues a request to the web service the request may be declined, because the client application did not originate from the same domain. This restriction is called the same-origin policy. The URL to the web service must have the same origin as the URL to the client application. Same-origin in this context means same scheme, same domain, and same port. Note that Internet Explorer does not consider the port when comparing origins. In our sample application, only the port number is different, so we did not see the same-origin restriction when running the sample application with Internet Explorer. But if we deployed the sample application to different domains, we would see this restriction with Internet Explorer as well. The purpose of this restriction is to prevent a malicious site from accessing our Web API. But sometimes we want to allow other sites to call our Web API, especially if we want to deploy client applications to other domains or if we want our Web API to be publicly available. One option for bypassing the same-origin restriction is with JSONP. Browsers don't enforce the same origin policy on HTML script tags. That allows client applications to request scripts from any domain and any server. JSONP takes advantage of this and uses a script tag, passing a special parameter to a JSONP-enabled server to provide callback-based cross-domain requests. But this wasn't a clean or complete solution. JSONP is now only used when targeting older browsers that don't support newer techniques such as CORS. Cross-origin resource sharing, or CORS, is a newer standard that makes this second scenario possible. CORS is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a Web API service can explicitly allow some cross-origin requests and reject others. There's no need for any special code in the client application, no script tag, no special parameters, CORS is handled in the service. Since the second scenario matches the way we have our application with the Angular code and website separate from the ASP.NET Web API project, we need to implement CORS. Luckily for us, ASP.NET Web API provides support for CORS. Let's see how to enable CORS in our Web API service.

Enabling CORS in a Web API Service

Enabling CORS in a Web API service is straightforward. Download the CORS package using the NuGet Package Manager. Once it is downloaded, call the EnableCors method as part of the configure setup for the Web API, and then set the EnableCorsAttribute where appropriate. Where can we set the EnableCorsAttribute? We can set it on a Controller class. So we could set this attribute on our ProductsController class. Alternatively, we can set it on a method. This enables CORS for the specific action only, so we could set this attribute on our Get method to allow access only to the Get action. Our third option is that we can set it globally by passing it as an attribute to the EnableCors method. Let's talk for a moment about the EnableCorsAttribute arguments. The first argument is the valid origins. Here we can provide a comma-separated list of origins that are allowed access to the resource. If we only want to access the Web API from our Angular client application, we could put the URL where the client application will be deployed. How did we figure out that port number? When we run the application, we can see the port number in the address bar of the browser. If we want our API to be public and accessible to any site, we can set the valid origin to an "\*". The second argument is the valid headers. Here we could define a list of headers that are supported by the resource. We are using "\*" to allow all. The third argument is the valid methods. Here we could provide a comma-separated list of methods that are supported by the resource. We are again using "\*" to allow all, but if we only wanted to allow read access, we could set this to Get. That's it. Let's give it a try. We are back in Visual Studio with the Web API project node open. Before we enable CORS, let's take a closer look at what is happening. Let's run the application using Chrome. Here we can see the URL used to access the client application. If you are coding along, you'll need this URL later in this demo. Press F12 to display the developer tools, and we see the error. Notice the two different port numbers. Here is the URL of the Web API service request and here is the URL of the origin used to access the client application. Notice that this is the same URL defined here in the address bar. Let's use the developer tools to look at the request, ensure that the Network tab is selected here, and click refresh. Now we can see each network request. Here is the request for the productListCtrl JavaScript file and the request for the productListView.html template file. Here is the call to the Web API to get products. Let's look at that one. We can use the Headers tab to view header information. If you scroll down you can see the Request Header, so notice that the origin is provided as part of that header. And here is the Response Header, but if we click on Response, we see that there is no response data available. Now let's enable CORS. We'll close the browser, and the first thing we need to do is get the NuGet package. So we'll select the Web API project in Solution Explorer, right-click, and select Manage NuGet Packages. Ensure you have the online node selected on the left, and use the search box in the upper right corner to search for CORS. Select Microsoft ASP.NET Web API 2.2 Cross-Origin Support, that's this one, and click Install. Accept the license, and it's finished, so we can click Close. We'll call the EnableCors method as part of the registration configuration, open the App\_Start folder in Solution Explorer, and open the WebApiConfig.cs file. Then we'll add the line to enable CORS. Now that we have that in place we can add the attribute. We want to enable the attribute for our ProductsController, so let's open that file. Controllers, ProductsController, and we could put the attribute on our Get method or on the entire class, and we'll put it on the class. Use the error tooltip to add the appropriate using. As discussed on the slide, the EnableCors constructor has three arguments. The first argument is the valid origins. We are defining our localhost with a port number here to only allow access by our Angular application. Be sure to change the port number here to the port number for your origin. The second argument is the valid headers. We are using an "\*" to allow all. And the third argument is the valid set of methods. We are, again, using "\*" to allow all. Now let's give this a try again with Chrome. Ah, we no longer see an Access-Control error, so now we have the same issue as with Internet Explorer, there are rows, but no data. We have enabled cross-origin resource sharing, but there is still something wrong. Let's use the F12 tools again and see what's going on. Press F12, ensure the Network tab is selected, refresh the page. Here again is our request for products. Click on that request and let's look at the request Header. So here is the Request Header, here is the Origin specified, and if we look at the Response Header we can see Access-Control-Allow-Origin, and our Origin here. So this number matches this number here, so this shows that our CORS is implemented appropriately. Now let's click on the Response Preview. Oh, we have our data, so all of our data is coming out. So if we're getting our data in the response, why isn't it showing up in our UI? Well let's look closely at the property names. They are PascalCase, capital D Description, capital P Price, capital P ProductCode. Now let's look at our Angular code. We can open our template file, our productListView, and here we see that the property names are camelCase, little p productName, little p productCode. Angular is case sensitive so our property names don't match. Let's look again at our model, and we can put these windows side-by-side. So here in our model we see that it's PascalCase, capital D Description, capital P Price, capital P ProductCode, and here in our Angular code they're camelCase, little p productName, little p productCode, and so on. So we could change the C# code here in our model to use camelCase, but C# best practices recommend defining properties with PascalCase, so we could instead change the Angular code to use PascalCase, but Angular best practices define properties to be camelCase, so we shouldn't change that either. Luckily ASP.NET Web API helps us deal with this mismatch using something called serialization formatters. Let's look at those next.

Understanding Serialization Formatters

From Wikipedia, "Serialization is the process of translating data structures or object state into a format that can be stored ... or transmitted across a network connection..." ASP.NET Web API provides a serialization formatter that automatically translates a list of business objects into Json or XML for transmission to the client application. Let's see what that looks like in a diagram. Here is the Web API service. In the code we wrote in the prior module, the productsController Get method retrieves the list of product business objects and returns them. The ASP.NET framework automatically processes that list of business objects through a JSON or XML serialization formatter. The client can request either JSON or XML in the Accept Header of the request. The appropriate serializer then translates the list of business objects into JSON or XML for transmission. The JSON or XML is then returned to the client application. For our client application, we want JSON, as shown here. The problem is that the serializer is defining the properties as they are defined in the business objects, in PascalCase. Luckily for us, ASP.NET gives us some control over the formatting of the serialization so we can instead make it look like this, and our Angular code can then use these camel case property names. Let's see how to configure serialization formatters.

Configuring the JSON Formatter in a Web API Service

Web API provides serialization formatters for both JSON and XML. Since most Angular applications use JSON, this course only looks at the JSON formatter. The JSON formatter provides several configuration settings. Let's look at a few of them. First there's the JsonIgnore attribute. Add this attribute to any property in the model to exclude it from the JSON. This may be useful if you have some internal state that you don't want sent as part of the response. DateTimeZoneHandling is another configuration setting. By default, dates are written in ISO 8601 format as shown here. Use DateTimeZoneHandling to define how dates should be handled. For example, you can select that all dates will be formatted in local time or in UTC. There is a formatting setting. That allows you to control how the JSON is indented, and CamelCasePropertyNamesContractResolver, wow, that's quite a name. This is the one we are interested in. This setting allows writing JSON property names in camelCase, without modifying the model properties. To configure the serialization formatter for camelCase, reference the JsonFormatter, and set the serialization settings.ContractResolver to the CamelCasePropertyNamesContractResolver. Yes this is a really long line of code here, but it works. Let's give it a try. Here we are again in Visual Studio with the Web API project node open. We want to modify the WebApiConfig.cs file. That's in the App\_Start folder. Here it is here. All we need to do is paste in the line of code from the slide, that's it. So let's give the app another try with Chrome. Success, we now see our list of products. Let's try again with Internet Explorer. Ah, that works too. We are now handling a retrieve request from front to back, but is it practical to display all of the products here. In a real application, how many products would we have to display? Hundreds? Thousands? More? Would a user really want to scroll through hundreds of rows to find the product that they want? This page really needs a search filter to find only those products that match specific criteria, so we need a way to pass search parameters back to the Web API service. We'll look at passing parameters in the next module.

Summary

This module focused on two issues we encountered when retrieving data. We began with a look at cross-origin resource sharing, or CORS. If the Angular client application is in a different domain than the Web API service, requests from the client may be declined by the server due to the same-origin policy of the browser. We saw this error when we ran the Angular application using Chrome. To accept requests from the Angular client application, we enabled CORS in the Web API service. This involved calling an EnableCors method during registration of the Web API service, and setting the EnableCorsAttribute on the Controller class or method as appropriate. In the sample application, we set the EnableCorsAttribute on the ProductsController class, and specified that only requests from the Angular client should be accepted. If you are building a public API, you can enable CORS to accept clients from any domain. Next we looked at serialization formatters. We realized that the second problem we had with retrieving data was a casing issue. The properties defined in our Web API model are PascalCase, so the Web API serialized those properties as PascalCase in the return JSON, but the Angular bindings are in camelCase, so we then looked at configuring the JSON Formatter. We set up the ContractResolver serialization setting to transform the PascalCase property names from the C# code to camelCase names for the Angular code. By enabling CORS and configuring JSON serialization settings, we have our sample Angular client application and ASP.NET Web API service working. Yay! Now that we are successfully retrieving data, we can check off that touch point. In the next module, we'll look at techniques for passing parameters from the client to the service for filtering, shaping, and querying data.

Passing Parameters

Introduction

Rarely do client applications want to retrieve all of the data. Instead of getting all products, for example, the client app only wants products of a particular type or those that match a specific search string, so the client application passes the type or search information to the Web API service as a parameter. Welcome once again to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module details how to pass parameters from an Angular client application to the Web API service, and how to recognize those parameters in the Web API service and respond accordingly. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. So far we have successfully retrieved data. This module covers filtering, shaping, and querying data by passing parameters from the Angular application as part of the URL. The Web API service evaluates these parameters, filtering, shaping or querying the data explicitly, and returning the result. In the next module, we'll look at using OData queries for a more flexible approach. There are several techniques available for passing parameters from an Angular application to a Web API service. One technique is using query strings. We've all seen these query strings. They're the strings in the URL that start with a question mark and often contain lots of ampersands. For example, this is an Amazon URL. The base URL is here, the ? denotes the beginning of the query string, then the parameters are defined as name value pairs with ampersands between them. So pf\_rd\_m is a parameter set to a value of ATV. We can use query strings to pass anything we need from the Angular client application to the Web API service. For example, here is our base URL. This is the URL we use to retrieve all of the products. We can add a query string onto this URL. the ? denotes the start of the query string. In this example we have only one parameter, search, and it's set to GDN. We could add other parameters here, separated with ampersands. Another option for passing a parameter between an Angular client-side application and the Web API service is to simply extend the URL path. In this case there is no ?. The parameters are defined by extending the URL with more slashes. So this may be the base URL, and here is the first parameter, and the second. We can use this same technique and simply add our parameter after another slash. Here, again, is our base URL, and the search string is simply added after the slash. These are the two techniques for passing parameters that we will cover in this module. The primary objectives for this module are defining query strings. We'll look at what query strings are and how to use them to pass parameters. Then we'll move on to the second technique, extending the URL Path. Lastly, we'll compare and contrast the two techniques, query strings or URL Path. Let's look at query strings first.

Defining Query Strings

According to Wikipedia, "A query string is the part of the Uniform Resource Locator, or URL, containing data that does not fit conveniently into a hierarchal path structure." But what does hierarchal path structure mean here? Look at this URL. The first part of the URL specifies that it is part of our API. The second part drills down into our API hierarchy to access products. Drilling down further, it references the product with an ID of 5. Drilling down further still, it references all of the transactions for product 5, so the URL navigates down a hierarchy to locate the set of transactions generated for the product with an ID of 5. This last piece of the URL is the query string. It is searching all of the transactions for any that were returns. It is no longer drilling down the hierarchy of resources, rather it is filtering at this point in the hierarchy. As we saw earlier, a query string is identified with a question mark. Each parameter is defined with a name value pair, and each name value pair is separated with an ampersand. This first example is one you may see when using Amazon. The second example is the one we are going to use. It has a single name value pair that defines a search string. Let's walk through how to pass a parameter using a query string such as this. Here is our Angular application on the left, and our Web API service on the right. This is the ProductsController we created earlier in this course, and the Get method we defined to retrieve all of the products. The second Get is a new method with a single parameter for the search string. We can modify the Angular client application to generate this URL and use it for the Get request sent to the Web API service. The Web API service then uses the base URL to route the request to the ProductsController. It then uses the request verb, Get in this case, to route to the appropriate method in that controller, and it uses the parameter name to locate the matching method signature. It looks for an overload of the method with a parameter name that matches the query parameter name. So in this case it looks for a Get method with a parameter named search. If it does not find a match, it ignores the parameter, so if the query string was named srch, no matching overload would be found, the query parameter would be ignored, and the Get method with no parameters would be executed. So we've just defined two steps for implementing query strings. First, the Angular client application needs to generate the appropriate query string. And second, the Web API service needs to have a method overload with parameters that match the exact parameter names from the query string. Let's start with the first step. Client-side in the Angular code passing parameters using query strings is easy. Here is the existing code in our productResource services that we created earlier in this course. It defines the path to retrieve products. Notice that it has no query string and no parameters. Any information that is passed into a $resource method and not declared in this signature is automatically formatted as a query string, so we don't have to change this code. Here is the code we currently have that calls the $resource query method. Notice that its only argument is the function to execute when the query returns. If we simply add an argument to this method, Angular will automatically generate a query string and append the name value pair onto the URL. Notice that the argument is an object. Each object property is defined as a name value pair. Here is the name, a colon, and the value. Each object property maps to one query string parameter. To define multiple query string parameters, define multiple name value pairs here within the braces, separated by commas, so all we have to do to the Angular code is pass the query parameters in as name value pairs here. Let's give this a try. Here we are in Visual Studio again with the Angular website node open. Let's start by running the application, and here is our list of products. Let's imagine that we have hundreds or thousands of products. To help the user locate the desired set of products we could add a product code search feature, only those products with the product code matching the search criteria will be displayed. Notice that the product codes contain abbreviations for the type of product. If we search on GDN, we'll find all of the garden supplies. For now we'll hard-code in the search criteria so we can focus on the touch point between the frontend client and the backend service. You are welcome to add a user interface for this as desired. So let's get started. We'll navigate to the Angular productController. In most cases the user will specify the search string, so it would be in a scope property here. We'll hard-code this value for now by specifying GDN here. We'll be filtering the list of products to those with a product code of GDN. You could later bind this property to a text box in the UI to allow user entry of the search string. We'll then need to pass in an object with a search property set to this variable, and we want it to be an object. We want to specify our parameter as search, and set it equal to our search string. If we had additional query string parameters we could type them here. But that's all we need for this example, now let's run it. Notice that all of the data still appears. Well, of course it does, we haven't changed the backend service yet to actually use those query string parameters. Since it did not find an overload with the parameter, the Web API just ignored the query string and executed the Get method that does not have any parameters, which is currently the only one we have. So how do we know then if we are generating the correct query string? Well, we can open the developer tools by pressing F12. Click to monitor network traffic and refresh the page. We can scroll down, and we can see here that Angular did indeed build a correctly-formatted query string for us. Now we just need to modify the Web API service to use that query string. In the Web API service, processing query string involves specifying an overload with a parameter name that exactly matches the query string name. Recall that our Get request URL looks like this, so our parameter name is search. We need a Get method with a parameter named search, like this. Code in this method, again, uses the repository we created earlier in this course, and it uses language-integrated query or LINQ, to filter the products to those with the product code containing the defined string. If you are not familiar with LINQ in C#, check out my "Practical LINQ" course in the Pluralsight library. Now let's see this code in action. We are back in Visual Studio, this time with the Web API project node open. In the Web API service, we want to modify the ProductsController to process the query parameter. We'll first navigate to the ProductsController, so we'll add another Get method that takes the query string as a parameter. The important thing here is that the parameter name must match the query string parameter. We used search as the query string, so we will use that parameter name here. In this code we are retrieving all of the products using the repository, then using language-integrated query, or LINQ, to filter to only those that contain the search string in the product code. Feel free to implement the code in this method as it is best for your needs. So let's give this a try. If it works, we should only see those products with GDN in the product code, and there they are. So to use a query string we follow two steps. First, in the controller we added an argument defining what the query string parameters are. We're defining them with name value pairs. Second, we define an overloaded method in the Web API controller whose parameters exactly match the query string parameters. Next up, passing parameters by extending the URL path.

Extending the URL path

As an alternative to using query strings, we can extend the URL Path to pass parameters to the Web API service. As we saw earlier, the parameters are defined by extending the URL with more slashes. So this may be the base URL, and here is the first parameter, and the second. We can use the same technique and simply add our parameter after another slash. Here is our base URL, and the search string is simply added after a slash. Notice that using this technique the parameters have no names, only values. Let's walk through how to pass a parameter using the URL Path. Here is our Angular application on the left, and our Web API service on the right. We can modify the Angular client application to generate this URl and use it for the Get request sent to the Web API service. Supporting this style of URL in the Web API service code is a little more complex. As you may recall, the Web API uses the URL for its routing. Now that there is no ? separating the base URL from the parameters, the Web API routing must figure out how to route the request. Here is the current code defining the routes. This code was generated for us when we created the Web API project earlier in this course. We are not currently using the id parameter, so we could change the id to a search string here and here. Then, similar to the prior example, we need a method overload with a matching parameter. In this case, however, the matching is done with the name from the route, so there are again two steps to this process, but each is more complex. First, the Angular client application needs to generate the appropriate extended URL, which requires a change to the $resource path. And 2, the Web API service needs to have a route and a method overload with matching parameters. Let's start with the first step. In the Angular client application, the first change is to the $resource path. To display the search parameter as part of the URL Path, it needs to be defined in the URL here. Prefix the parameter with a colon. The second change is the same as the query string's example. Add the parameter as the first argument of the $resource method. Note that these names need to match. Let's give this a try. Here we are in Visual Studio with the Angular website node open. We want to modify the signature of the URL, so we'll open the productResource.js file. Parameters can be added to the route directly here. We currently have an id defined here, and we're not currently using it, so we can simply change the id here to search. The next step is to modify the query method call, but we already did that for the query string's case. Let's look at both of this code together. If a parameter is passed in the $resource method, and that parameter is not defined in the $resource path, the parameter will be automatically added to the URL as a query string. If a parameter is passed in the $resource method, again, like the query in this example, and that parameter is defined in the $resource path, the parameter is added as an extension to the URL, so a minor programmatic difference, but a big change to the end result. Now let's give this a try. Notice that none of the data now appears. This is because the routing in the service does not have a match to the route we are now providing. If we open the developer tools by pressing F12, click to monitor resource traffic, and refresh the page, we can see that Angular did indeed build an appropriate URL for us. Now we just need to update the route in the Web API service. In the Web API service, processing parameters as part of a URL requires modifying the route template parameters. Here we will change the id parameter that was provided by default. We'll change it to a search parameter that we are naming, simply, search, so the route template now matches the URL. Next we define an appropriate overload with parameters that match the route. The parameter name must match the name defined in the route template. Let's see all this code in action. We are back in Visual Studio with the Web API project node open. In the Web API service, we want to start by modifying the route. Navigate to the App\_Start folder and open the WebApiConfig.cs file. This is where the routing is defined. Recall that Visual Studio generated a route for us with an optional parameter for an id. We'll change the id to our search string. This code will then route to an action method with a search parameter. If we go to our productsController, we see that we already have one, we created it in the last clip, and the parameter is already named search. So let's give this a try. In there we have it, a list of our products, filtered to only those with the product code containing GDN. To summarize, in the Web API code we need to modify the routing path and define a method overload with the appropriate parameter. The parameter name needs to match the name used in the routing path. So when should you use a query string versus extending the URL Path? We'll talk about that next.

Query Strings or URL Path

Now that we have tried out both query strings and extending the URL Path, how do we decide which one to use? Let's start with a look at their differences. As we have seen in the code examples in this module, the Angular code that is needed for query strings or the URL Path is similar. In either case, we pass the parameters as an object to the $resource query method. If the parameter is defined on the $resource URL, as shown here, Angular adds it to the URL Path. Otherwise, if there is no matching parameter on the $resource Path, Angular adds the parameter as a query string. The Web API code required to process the parameters from the query string or the extended URL Path is also similar. In either case we define an action method overload in the controller and write the appropriate code to process the parameter. In this case, we are filtering the list of products to those that match the define search string. When using query strings, the name of the method parameter must match the name of the query string parameter. If the parameter is defined on the URL Path, we also need to update the route template to match, and the action method parameter name must match the route template parameter name. So when should we use a query string versus a URL Path? Here are some general guidelines. Use a query string for filters, anytime we want to search, sort or narrow down a set of results. So our search feature is a good example of when to use query strings. Use a URL Path for locators when locating a specific resource, by navigating down the resource hierarchy, such as api/products/5. This retrieves the specific product with an id of 5, or api/products/5/sizes, this retrieves the set of available sizes for the specific product with an id of 5. And you can combine both of these techniques, as long as the query string is at the end. This URL locates the set of transactions for the product with an id of 5, and then filters those transactions for any product returns.

Summary

In this module, we began with a look at passing parameters in the URL as a query string. We modified the Angular code to pass the parameter in as an argument to the $resource service method. The query string was then generated automatically with the defined parameter. On the Web API side, we simply created a method overload with parameters that exactly match the query string parameters. Then we looked at passing parameters as part of the URL Path. The code here was a bit more complex. In addition to the changes for the query string, we also had to change the $resource path to include the parameter in the URL, and we had to change the Web API route template. When comparing these two approaches, we found that query strings were best for filters or miscellaneous parameters, basically, for any cases where the parameter does not fit into the hierarchal path structure. And URL Paths were best for locators, for finding a specific resource or unfiltered set of resources. These techniques can be combined as required for your application needs. This module looked at passing parameters from the Angular frontend to the Web API service on the backend. We saw how to pass parameters to filter data to a specific subset, in this case garden products. Though we didn't look at an example, we could also pass parameters to shape the data, adding more or fewer properties, and we could pass parameters to perform basic data queries, but we'll see a much easier way to query data in the next module. So maybe only have a checkmark at this point. The next module covers OData, more specifically, using OData queries to filter, shape, and query information to our Web API service. This gives the client application more control over the data it retrieves from the service.

Using OData Queries

Introduction

More data is generated now than ever before, and to help our users decipher that data we need to manipulate and display it in a user-friendly way. That means filtering, sorting, shaping, and querying that data. Welcome once again to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module describes what OData is, and demonstrates how to use OData queries to filter, sort, shape, and query data, so you can display just the data the user wants and how they want it. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. So far we have successfully retrieved data, and in the last module we saw how to filter that data using query strings and URL parameters, which only earned a half of a checkmark here. This module details how to use OData queries for more advanced filtering, shaping, and query scenarios. That should give us the other half of a checkmark here, and once you see the power of OData queries, it may become your go-to solution for all of your data manipulation needs. Let's begin by defining what OData is. OData is short for Open Data Protocol. Its purpose is to provide a standard way to access data on the web using basic HTTP. OData defines a protocol that allows any client application to access information exposed by any data source that implements OData. The OData protocol largely follows the conventions defined by REST, and uses the standard HTTP verbs, such as GET, PUT, and POST. For more detailed information, search for OData in the Pluralsight library. In this module, our focus is on OData queries. OData defines a series of query options that allow us to construct complex data queries on the client and execute those queries in the backend service. We can use queries to filter, shape, sort, and select just the data we want from any web service that supports OData queries. For example, in our sample application we were originally retrieving all of the products. In the last module, we saw how to use a parameter to retrieve only garden products. We could accomplish this same task using OData with a filter. This filters the list of products to only those that contain the string 'GDN' anywhere within the ProductCode property. But what if we wanted something more complex, such as only those garden products with a price less than $10.00 or with a price between $5.00 and $20.00, and sorted by price? Cool, OData provides a syntax for defining more complex queries to filter, shape, sort, and select the data. Hmm, but how much code do we need to write in the Web API Controller to support all of these options? Well, that's the benefit of using an OData query with Web API. Web API supports OData queries. OData defines a generalized syntax for specifying these types of data queries, and the Web API automatically handles processing of these queries. We don't have to write any code. Yep, you heard that right. We don't have to write any code in the Web API to understand or process any OData queries. It just works. The first time I saw how easy it is to use OData queries with Web API, I wanted to share it with everyone I knew. So now I'm sharing it with you. This module covers the following topics, enabling OData queries in a Web API service, learning OData query options, using OData queries from Angular, understanding OData expression syntax, and we can't forget the OData query security considerations. OData queries are pretty awesome stuff, so let's get started.

Enabling OData Queries in a Web API Service

Before we can begin using OData queries in our Web API service, we need to enable it. Here is our current Web API Get method that retrieves all of the products. Look familiar? Enabling OData requires three steps. First, we add the EnableQuery attribute to the Get method. We can add it to our product Get like this. This attribute enables querying using the OData query syntax. Second, we need to change the IEnumerable return type to IQueryable. Currently this method returns a method of IEnumerable<Product>. We'll change it to return an IQueryable<Product>. This provides the functionality to evaluate queries. Third, we need to add AsQueryable() to convert the IEnumerable return from the repository to an IQueryable. We'll add that here. That's it. That's all we need to do. No need for specialized methods or method overloads. No need to change the routing template configuration. These are the only code changes we need to make, and our code will automatically support basic OData queries. Cool, let's make these changes to our Web API. Here we are in Visual Studio with the Web API ProductsController file open. The first thing we need to do is install the NuGet package for Web API and OData. Right-click on the Web API project and select Manage NuGet Packages. Search for OData, we want this one, click Install, and that's it. As we saw in the slides, enabling OData queries requires three steps. First we'll add the EnableQuery attribute to our Get method. This enables querying using the OData query syntax. Use the error tooltip to add the appropriate using. Second, we'll change the IEnumerable return type to IQueryable. This provides the functionality to evaluate queries. Notice that we now have a syntax error because we're still returning an IEnumerable. So third, we'll add a call to AsQueryable. This converts the IEnumerable to an IQueryable, and ensures that we are returning the required type. That's it. That's all of the Web API code we need. So how does Web API handle the OData queries if we don't have any code here for them? The EnableQuery attribute here is an action filter that parses, validates, and applies the query. The filter converts the query options from the OData query into a LINQ or language-integrated query expression. When the controller returns an IQueryable type, it processes the LINQ expression using an appropriate LINQ provider. Performance, therefore, depends on the LINQ provider that is used, and on the characteristics of the data model or schema. In our example we are not using ADO.NET or Entity Framework. The Retrieve here is returning a set of product objects, so the LINQ provider is LINQ to Objects. If you use Entity Framework here, Web API will use LINQ to entities instead. We are now ready to start processing OData queries, but before we can start writing some OData queries, we need to learn a few of the query options.

Learning OData Query Options

OData queries are submitted from a client application to the Web API on the URL as a query string. As we saw in the last module, a query string in the URL starts with a question mark. Each query string parameter consists of a name, $filter in this case, an equal sign, and a value, which is this query expression. Each parameter is separated with an ampersand, so here is the second query string parameter. When submitted to the Web API, this query string filters the products where the price is greater than 5, and sorts the result by the Price property. OData calls these parameters, like the $filter and $orderby, query options. A few things to note here, the OData query options are prefixed with a $. Property names and the query expressions, such as the Price property shown here, are cased as defined in the Web API. This often means that the property names are Pascal cased. The operators are lowercase text, such as the gt here for greater than. Lowercase text is used, because most of the standard comparison operators, such as the equals sign or greater than sign, already have a different meaning in the URL, so the OData specification defines all of the operators as lowercase text. Let's look further at the query options that OData provides and then try them out in our Angular code. The basic set of OData query options include these shown here in this table. The $top query option returns the first n entries from the set, where n is defined as the query option value, 3 in this example. User $top to limit the results returned from the query. This is useful to prevent excessive data downloaded to the client, and for setting up paging, whereby the user can page through the results. The $skip query option skips the first n entries in the set, where n is defined as the query option value, 1 in this example. $skip is often used when paging to skip the entries already displayed. The $orderby query option sorts the results in ascending or descending order by the value of one or more properties. In this example, the products are ordered by the ProductName and are in descending order. The default is ascending order. Note that the field name used in this example is PascalCase to match the casing of the Web API property. Note also that the orderby expression is enclosed in quotes. The $filter query option is one of the most powerful options, because it can filter data using many different criteria. The value of the $filter option is an expression. Each item in the set that matches the expression is included in the return data. This expression returns all products that have a price greater than $5.00. Note that the field name is in PascalCase, matching the casing of the Web API property, and notice that the greater than operator is simply lowercase text. The $select query option defines the properties to include in the response so if the Web API could return a bunch of properties, and we only wanted two of them, we can use $select to limit the downloaded data to only the properties we need. Now let's see how to define these query options in our Angular code.

Using OData Queries from Angular

Now that we know the basic OData query options, let's look at how to use them from Angular. Just like with the query string parameters in the prior module, the queries get passed in as the first argument to the $resource query method. Recall that the Angular code uses the $resource query method to call the Web API and return an array of data, passing an object in as the first argument to this function defines the set of parameters. And if the $resource object does not define these parameters in the URL template path, then the defined parameters are automatically added as query strings. In this example we're using two query options, the $filter option will locate all products with a ProductCode that contains 'GDN' and a price that is less than $10.00. The $orderby option will sort the resulting set by price descending. Since we want these query options to appear in the URL as query strings, we don't need to change the productResource URL template path, so this is the only code change we need to make. Can't wait to try this out? Me either, let's do it now. Here we are back in Visual Studio with the Angular productListCtrl file open. Let's start by removing the search parameter that we added in the last module. Then let's run the application and take a close look at the data that is returned. So here is our list of products. Notice that our first product is Leaf Rake, and the second is Garden Cart. Okay, so let's stop and add our first OData query. Let's start with a simple one. We put the query options in as the first argument, just like we did with our query strings. We want an object here, and in that object we'll specify our query options, so we'll use $skip:1, $top:3. Okay, that's it. So what this should do is skip our first product, which was the leaf rake, and pick the top 3 from our list. So let's run it and give it a try. Yeah, that worked. It skipped the leaf rake, and it displayed the next three products, and remember that we didn't have to create a special Web API method, we didn't have to define a route, we just marked the Web API Get method with an attribute to enable querying, and returned an IQueryable. Web API did everything else for us. Oh, let's try another one. Let's filter to only the garden products. For that we'll use the $filter OData query option and an OData query string function named contains. Note that contains is new in OData version 4. If you are using an older version of OData, use substring of instead. We'll talk more about the set of valid string functions after this demo. Let's run. Yep, that worked also. What if we want multiple criteria in the filter? Let's add a Price Check onto this filter. So inside the quotes we'll add the Price Check. Let's run again, and now we have a much smaller list of products, only those within the price range we specified. But what if we also want to sort? We can separate multiple query options with a comma here in our object, and we'll add an $orderby. Let's run again. Now we have the list of products in order from highest price to lowest price. Oh, this is way too much fun, but we've done enough examples to give you the basic idea. In the next clip we'll focus on additional expression syntax, including the valid operators and some of the key filter functions. Before moving on, let's talk about the user experience with regard to data filtering, shaping, and querying. Currently we hard-coded the OData queries. This may not be practical in a real application, so how do you expose these options to the user, or should you? Depending on your needs, you could hard-code some of the query options and allow the user to select others. For example, what about sorting? We could change these column headers to be links and adjust the $orderby clause based on the user selection. And for filtering, we could add a search box here that would map to a variable in our filter clause. Let's see what that code would look like. Here at the top of the Angular controller code we identify three variables. These could be bound or set based on user selections. We then use these variables when defining a filter, and $orderby OData queries. This clause locates any entered search criteria in either the ProductCode or ProductName. We could add more clauses here if we also wanted to search the product description. This clause sorts the data by the define sortProperty and in the define sortDirection. By limiting the user's ability to filter, shape, and query the data to a set of options that we control, we can offer some of the richness that OData queries provide, yet limit our security and performance risks. We'll talk more about those risks later in this module. For now, let's take a closer look at the $filter expression syntax, specifically, the operators and functions.

Understanding OData Expression Syntax

To get the most from our OData queries, it's important to understand the OData expression syntax. For example, here is the code from the prior clip. The $filter OData option here is using one operator, or, but there are lots of other operators available. The $filter is also using a string function, contains. Let's look further at both the $filter operators and available functions. The $filter query option has a set of operators for creating the filter expression. The key operators are shown here in this table along with a description and example. These are the same basic operators used in most computer languages, equal, not equal, and so on. The important thing to note here, however, is that these operators are defined with lowercase text. That is because symbols, such as the equals sign, greater than sign, ampersand, and pipe characters, all have other meanings in the URL. Once you know that the operators are all lowercase text, the operators are somewhat intuitive. Feel free to pause here if you want more time to look this over. The $filter query option also supports basic functions. Some of the more common string functions are shown in this table with examples. You can check the beginning of a string. Here we are looking for the string 'GDN' at the beginning of the ProductCode property value. You can check the end of a string or use contains to determine if the property contains the string anywhere within the property value. There are also functions to alter the case for comparison. These are especially useful if you're collecting values from the user and you want to ensure they are of the appropriate case for comparison. The $filter query option has many, many more operators, functions, and other features. They are shown in type categories here. For a complete list, along with the details and examples, check out the OData website. This is www.odata.org. Here on the website select Developers, Documentation, Part 2: URL Conventions. If you scroll down to the Table of Contents, Chapter 5 has all of the Query Options. They have the Operators, Canonical Functions, and so on. Use this documentation to get the most from OData queries.

OData Query Security Considerations

Using OData queries is all great fun, but let's not forget to consider the security and performance implications. First off, we probably don't want to allow our users to just type OData queries directly into the URL. Depending on our data, the user could request more information than they should be downloading. This could cause performance issues, and if the user is running the app from a phone, data charges could be expensive. They could also potentially select a set of filters, and/or sorts, that make the query very non-performant. That could have a major performance impact on the web service. Instead, we want to control their filters and sorts, as discussed earlier in this module, and use cross-browser resource sharing, or CORS, to limit API access only to our application, as we showed earlier in this course. There are some actions that we can take in the Web API code to limit our risk. First, we can use the JsonIgnore attribute on any property of our Web API model that we don't want to expose. We discussed this attribute earlier in this course, in the CORS and Formatters module. There may be system or other properties in the model that should not be publicly exposed. By using JsonIgnore, the property won't appear in the JSON, even if it is requested with a $select query operator. Use the EnableQuery attribute sparingly. Only add the EnableQuery attribute on the methods that you want to allow for querying. Consider limiting the PageSize. The query attribute has a parameter, whereby we can limit the number of rows returned. In this example we're setting the PageSize to 50 so it returns, at most, 50 rows. So if the user selects a set of options that returns too many rows, we'll limit it to 50. Consider limiting the available query options. We can specify which query options are allowed, by again using the EnableQuery attribute parameters. For example, setting AllowedQueryOptions to AllowedQueryOptionSkip or AllowedQueryOptionsTop, sets the options to only the skip and top query options. Any other query options are ignored. There are other EnableQuery attribute parameters that you can use to disable query options. Disable query functions, such as any or all, or restrict the available set of orderby properties. And if you want very fine-tuned control, consider writing a vaildator function in the Web API component. A validator function validates a query before executing it. That allows for detailed control over the query contents before the query is executed. The code required to build a validator function is beyond the scope of this course. Search the web for overriding the ValidateQuery method if you are interested in more information.

Summary

Okay, this was some pretty cool stuff. In this module we covered OData, specifically, OData queries. After a brief introduction to OData, we saw how easy it is to enable OData queries in our Web API. We just installed a NuGet package, enabled querying with the EnableQuery option, and set IQueryable as the return type of our methods. Then we looked at the basic OData query options. We could use $top and $skip to implement paging, $orderby sorts the data, $filter provides an extensive set of operations and functions to filter the data, and $select lets us limit the properties returned by the service. This module demonstrated how to build OData queries in the Angular controller, and we tried out several different query options and expressions. Then we took a closer look at the OData $filter expression syntax. Because normal symbols for operation, such as equals and greater than are already used in a URL, we saw that OData expressions use lowercase text, such as eq for equals, and we saw that OData has an extensive list of functions for handling strings, dates, numbers, and much more. Lastly, we covered OData queries security and performance considerations. We looked at several actions we can take to minimize our risk, including limiting access to the OData query options and limiting the page size of returned data. This module demonstrated how easy it is to implement OData queries for extensive filtering, shaping, and querying of data. With that, we can definitely change the half checkmark to a full mark. Next up, the touch points around saving data.

Saving Data

Introduction

Many Angular applications collect data, and that data needs to be saved, often to a backend database. The Angular application provides client-side data entry, then submits the data to a backend service for server-side storage, basically saving the data to a desired data store. Welcome once again to Angular Front to Back with Web API, from Pluralsight. My name is Deborah Kurata, and this module demonstrates how to submit data from an Angular client application to a backend Web API service for storage. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. We've already covered retrieving a set of data such as a list of products, and how to filter, shape, and query data using query strings, URL parameters, or my favorite, OData. This module looks at how to save data. Often saving data is divided into two separate activities, saving updates to existing data and saving newly entered data. Let's look at Updates first. Updating data requires several touch points. First, the Angular code retrieves the data for the existing item from the Web API using a GET Request. Wait, what? Why do we need to retrieve the data? Don't we already have the data displayed on the list page? Well, we might have some of the data, and it might have all the current values. When displaying data for a list page, we may only have key data fields, such as the id and name. When editing data, we want more of the fields, including the description and detail information. Plus, the list page may be cached. When editing the data, we want to ensure that the user sees the current values of each field. So the first touch point for an update operation is a Retrieve of the specific item to edit. In this example, we are sending a GET Request for the product with an id of 5, and the response contains the data for the requested product. The Angular application then displays that data for the user to edit. The user updates the data as desired. The Angular application often includes field-level validation to help the user enter correct and complete data. The second touch point occurs when the user has finished making the changes. The Angular view often provides a Save or Submit button. The user clicks that button to submit all of the data back to the Web API service. The Web API service processes the request, and if the data is valid it is saved to the appropriate data store. What about creating an item? There are two basic ways to handle creating new items. One option is to simply display a blank data entry page from the Angular application, and if default values are desired, they are hard-coded into the Angular client application. Another way to create a new item is with a default item from the backend service. This technique is similar, then, to an update, whereby the Angular code starts with a GET Request. Notice that the request is for the product with an id of 0. When it sees an id of 0, the Web API service creates a new item, setting default values where appropriate, and returns it to the Angular application. Notice in this example that a default ReleaseDate was set. The Angular app then displays that new item for the user to edit. The user enters the data, the Angular app again provides client-side validation. When data entry is complete, the user clicks the provided Save or Submit button to submit the entered data back to the Web API service. The Web API service processes the request, and if the data is valid, it is saved to the appropriate data store. We'll use the second create technique, whereby a default item is created by the Web API. So for both the Create and the Update cases, there are always two touch points. The Angular code starts with a GET Request to the Web API service for either an existing item or a new default item, and the Web API returns the requested data in the response. When the user selects to Save, Angular submits a second request to the Web API service. The data to save is sent in the body of that request, and the Web API saves the item. With the touch points defined, let's look at what we will cover in this module. We'll start with the Web API service. We will implement a Get method in the Web API Controller that retrieves the data for one item. We will then implement both PUT and POST methods to save the data. With the backend service methods in place, we'll move to the frontend code. We'll need to add an Edit Page to the application so we can create or edit data. Then we'll add the code to call the Web API service from Angular. With that, we'll have an operational Save feature in place from front to back. Let's get started.

Building the Web API Service Methods

As we just saw, a Save feature requires three Web API service methods. We need a GET method that will retrieve a single new or existing item for update, we need a POST method, and we need a PUT method. We'll talk about when to use POST versus PUT shortly. First, let's jump right in and build the GET method in our Web API. Here we are in Visual Studio with the Web API ProductsController open. We have several Get methods here. This first Get method retrieves the entire list of products, unless of course we filter that list with OData as we saw in the last module. This second Get method was used to retrieve products that match a specific search string when using a query string or URL parameters. With OData we don't need this method anymore, so we can delete it. The third Get method was generated by the template when we first created this controller. It retrieves one item by id. This is the method that we want, we just need to modify it. First, we want to change the return type from string to product. This Get method will now return the product in the response body in JSON format. Then we need to replace the code in the method body. Where will we get the product? If you said repository, you are right. I'll paste in the code, and we can walk through it. This code creates an instance of the product repository that we created earlier in this course. If the GET Request is for a product with an id greater than 0, this code retrieves all of the products from the repository and uses LINQ, or language-integrated query, to locate the one product with the defined id. The LINQ FirstOrDefault method locates and returns the first product in the product list with an id matching the passed-in id. If you are not familiar with LINQ, check out my Pluralsight course entitled, Practical LINQ. Note that the code used here to locate the product is not very efficient. It calls the Retrieve method on the repository to get all of the products, and then filters to the desired product. In a real application with a real database, this code should instead get the one product from the database. Continuing on with this code, if the request is for a new product, this code calls the Create method of the repository to create a new product with appropriate default values. Now the client application can request a single product. The requested product is returned in the response body. We'll look at error handling for this method in the next module. There is one more change we need to make. Recall that earlier in this course we modified the WebApiConfig file, changing the id parameter to the search string. Now we need to change it back. And we're ready to give it a try. Let's run just the Web API project. To do that, we need to set the Web API project as the startup project, and we need to modify the start action back to CurrentPage. That's defined in the project properties. Now we can start the application. Type in api/products/5, the product is downloaded, we can click to open it. And we see the JSON containing the single product that we had requested. Now let's try it again. This time we'll type in api/product/0, open. This time we see JSON containing a new product. Note that the only value defined here is the date that we set for a default value. Our new Get method seems to be working, so let's move on. So we are now ready to write the code for the methods to save the data, POST and PUT. But what's the difference? POST posts data for a resource or set of resources without specifying an id, as shown in the URL here. It is used to create a new resource when the server assigns the id, or update a set of resources. PUT replaces data for a specific resource using a provided id, as shown in the URL here. It is used to create a new resource when the client assigns the id, or update a resource with the defined id. If using a database, we often want the server to assign the unique id's, and not the client. The URL of a POST Request identifies the resource that will handle the item contained in the body of the request. That resource might be a factory for creating new resources, a process that accepts data, or a gateway to some other protocol. This URL is basically saying, route this request to the ProductsController and pass it all of the data contained in the request body. The URL of a PUT Request identifies the item contained in the body of the request, so this URL is basically saying, I have a product with an id of 5, replace that product if it exists, otherwise, create it and use the data contained in the body of the request. As a side note, there is also a Patch method that allows for partial updates, but we aren't covering that method in this course. One other important differentiator between POST and PUT is that PUT is defined to be idempotent. That means that multiple PUT requests are expected to have no additional effect if called multiple times with the same parameters. For example, multiple PUT requests to update a product should have no additional effect than a single PUT request. POST is not idempotent. Issuing multiple POST requests are not expected to have the same effect as one request. For example, multiple POST requests to create a new product could result in multiple new products. Bottom line, we will use POST when creating a new product, because we want the Web API service to have the responsibility of assigning appropriate id's. We will use PUT when updating a product. We will explicitly set the id of the product to update, and the existing resource will be replaced with the updated resource. Let's look at the code for the POST first. Here we are back in Visual Studio and looking at the Web API ProductsController. When we first created the Controller, the Web API template automatically generated this POST method signature for us. First we need to change the method parameter from a string to a Product object. We are going to use this POST to save newly-created products, so we want to pass in the data for the new product. What is this FromBody attribute for here? The FromBody attribute defines to the Web API that the parameter value should come from the body of the request. Otherwise, the parameter is assumed to be defined on the URL, as we saw with the GET methods. The next step is to write the code for this method. Remember that Product repository from earlier in this course? We just need to create an instance of the product repository and pass the new product to the Save method. We already have all of the code in the repository that actually saves the data back to the data store. What about data validation? What about exception handling? We'll cover all of that in the next module. For now, we're done with the Post method. Moving on to the Put method. Here is the Put method signature that the Web API template created for us. The first parameter is the id of the product to update. It will be provided on the URL. The second parameter is the product data defined in the body of the request. So we, again, need to change the method parameter from a string to a product object. This parameter uses the FromBody attribute so that the method knows that the product object should come from the body of the request. The next step is to write the code for this method. We'll again use the ProductRepository, and we'll pass the Product id and Product object to the Save method. Once again, we already have all of the code in the repository that actually saves the data back to the data store. We are ignoring data validation and exception handling here as well. For now, we're done with the Put method. With the first iteration of the Web API methods in place, let's build the client code.

Building the Angular Edit Page

If we were building an Edit feature in a real application, we would modify our List page to provide an Edit button and/or links to an Edit page for each product. Clicking the Edit button would take the user to a full-featured set of Edit pages to edit the selected product. But building these features is a course unto itself. The application you see here was built step-by-step in the AngularJS Line of Business Applications Pluralsight course. Our focus in this course and in this module is on the touch points between the Angular Edit page and the Web API backend service, so we won't take the time to repeat what is already covered in the AngularJS Line of Business Applications course. Rather, we'll use a simplified version of this Edit page and modify our index.html page to display the Edit page directly. To implement our simplified Edit page, we need to create a new HTML template. This template contains the HTML to display an Edit form. We need to build the associated controller. The controller populates the Edit form from the data retrieved by the GET Request and saves the data with a PUT or POST Request. Lastly, we need to modify the index.html file. We need to include the script tag for the new controller, and we need to modify the ng-include directive to display the Edit page instead of the List page. When we are finished, we'll have a simple Edit page that displays the product to edit. Notice that each data entry element has a label and a data entry field. The nice style of this page comes from Bootstrap, the JavaScript library for layout and styling of responsive client-side applications. I've done each of these steps, let's take a look. Here we are back in Visual Studio with the Angular website node open. Since the last demo, I have added the sample Edit page and associated controller to our Angular application. If you want to code along, you can get these startup files from my Github account, DeborahK/AngularF2B, front to back, WebAPI. Look in the folder called startFiles. Now let's start with a look at the new HTML template. In this new productEditView.html file, I added a form. In the form tag I defined the controller using the controllerAs syntax, similar to how we defined a controller in the productListView.html file that we created earlier in this course. Within the form is a field set, and within the field set are div tags defined using a Bootstrap form-group class. As discussed earlier in this course, Bootstrap is a library for adding styles to our HTML templates. Let's look at the first form-group. Each form-group has a label, ProductName, for example. The form-group has an input element, such as the text element defined here for the ProductName. Notice the client-side validation that is defined here. The ProductName is a required field that must be at least 4 characters, and no longer than 12 characters. The form-group also has a set of span tags for validation messages. If all of this looks strange to you, consider watching the AngularJS Line of Business Applications course first. In the Line of Business Applications course, I go through building a very similar form, step-by-step. In this Edit form, there are multiple form-groups. We just looked at the form-group for the ProductName. There is also a form-group for the ProductCode, for the AvailabilityDate, for the Description, and of course we have Save and Cancel buttons. Notice that the Save button calls a submit method, and the Cancel button calls a cancel method. Lastly, we have a message area so we can display messages to the user. Now let's look at the controller. We defined it here as productEditCtrl. At the top is the code to register the controller with our main module. Next we set up the model using the controllerAs syntax. Then we get the product to be updated. Because we will be including this page directly and not linking from the List page, we are hard-coding in retrieving the product with an id of 5. When the data is retrieved, that data is assigned to the product model. We are also making a copy of the data so that we can implement a cancel operation. Next we are setting the title of the page, based on whether this is a new or existing product. Since we are retrieving an existing product, the title should appear with the Edit text. We have not yet implemented the submit option. We'll do that later in this module. We have implemented the cancel feature. When the user clicks the Cancel button, we revert to the saved copy of the product. Next we need to modify index.html. We need to change the ng-include directive to display the Edit page, and we need to include the script tag for the productEditCtrl. Before we can give this a try, we need to reset the start project and we need to modify the Web API project properties. We, again, want to set, don't open a page. With those two changes in place, we are ready to give this a try. There it is, our Edit page comes up. It displays the data for the product with an id of 5, which is the Hammer. We can update the fields and we see that we have client-side validation in place. Click the Cancel button to undo all of the changes. Now let's try entry of a new product. Because we are simply using an ng-include and not linking to this page, we need to change our hard-coded value. So we'll go in the productEditCtrl and change the id here from 5 to 0. Then we can run again. We get an empty form with only the availability date set. Again, we can see the validation working and use Cancel to reset the form. Oh, and one more thing. Recall that we modified the ProductResource service to pass a search string on the URL. We need to change that back to pass an id. Now we are ready to call the Web API to save the data.

Calling the Web API to Save the Data

The Angular Edit page we just created has a Save button that the user can click to save changes. Let's look at how that process will work. Here is what we have so far. This box represents the main Angular module called productManagement. We have our new View called productEditView, and our new controller called productEditCtrl. The code for the button in the productEditView has a click event that calls a submit method in the controller. This box represents the common.services module called common.services. This module includes our custom Angular service called productResource that we created earlier in this course when we implemented our first Retrieve method. We'll reuse that productResource service to save the data. Recall that our custom Angular service uses the built-in Angular service called $resource to communicate with the Web API. Clicking the Save button calls the submit method in the controller. The submit method will call an appropriate $resource method defined in the productResource service. The Angular $resource service sends a PUT or a POST Request to the Web API service, passing the data to save in the body of the request. The Web API service then processes the request and saves the data. It sounds easy enough. We've already built the code for the PUT and POST action methods in the Web API service. We have the View and Angular controller in place, but we haven't yet created the code for the submit method in the Angular controller. We also need to ensure that the $resource object provided by the productResource service supports all of the actions that we need. Hmm, let's start there and look at the actions provided by the $resource object in the $resource service. The built-in $resource methods include Query. Calling the Query method issues a GET Request, and expects the response to return an array of objects. Earlier in this course, we used the Query method in the productListCtrl to send a GET Request that returned an array of products. We then used that array of products to populate the productListView. There is also a Get method. Calling the Get method issues a GET Request for an individual resource, and expects the response to return that single resource. Earlier in this module, we used the Get method in the productEditCtrl to send a GET Request that returned a single new or existing product. We used that product to populate the productEditView. There is also a built-in Save method. Calling the Save method issues a POST Request and sends the associated data in the body of the request. We can call the Save method in the productEditCtrl to post the data for a new product. The $resource service does not have a built-in method to send a PUT Request, so we can either use POST for both the Create and the Update operations, or we can add a custom $resource service method. Let's go with the second approach and see how to add a custom $resource method. Here is the return value from the productResource service we created earlier. The first argument passed to the $resource method is the URL template. It uses the server path constant we defined earlier. To identify the path to our Web API service, we need to add two more arguments here to define our custom $resource method, like this. The second argument is for parameter defaults. We could set a default value for the id parameter here if desired. We don't want to set a default value, so we are passing in null. The third argument is a set of custom actions. Here is where we define any custom actions needed by our Angular application. We are just defining one custom action. We can call that custom action anything. We selected to call it 'update', and we want it to send a PUT Request, so we defined the associated action method as a PUT. That's it, that's all we need to do to add a custom $resource method that sends a PUT Request to our backend service. Let's add this code now. Here we are back in Visual Studio again with the Angular productResource file open. We'll add the second argument of null here, since we don't want to set any default values for our id parameter. And we'll add an object for the third parameter. In this object, we'll add the update action to send a PUT Request. That's it, we now have our custom update action in place. We now have all of the $resource methods that we need. Query to issue a GET Request and retrieve a set of products, Get to issue a GET Request and retrieve a single product, Save to issue a POST Request to save a new product, and our custom Update to issue a PUT Request and save changes to an existing product. We are now ready to implement the submit method in the productEditCtrl. Here we are back in Visual Studio with the Angular productEditCtrl file open. We want to implement the submit method. I'll paste in the code and we can walk through it. First we clear any text from our message. Recall that we defined this property to contain any messages that we want to display to the user. If the product has an id, we assume it is an Update, so we call our custom update method, passing in the id of the product as the parameter. This will issue a PUT Request, defining the id of the product to save in the URL, and pass the product data in the body of the Request. The second argument of the Update function is the success callback. If the update is successful, we'll notify the user with a message. If there is no product id, we assume we are saving a new product, so we call the built-in Save method. This will send a POST Request, and pass the entered product data in the body of the request. The argument of the Save function is the success callback. If the Update is successful, we'll notify the user with a message. We'll also update the original product property with the product returned in the response. Recall from earlier in this module, that we are using the original product variable for our Cancel feature. It retains the original version of the product data, so if the user has unsaved changes and then clicks the Cancel button, we can return to that original version without having to re-retrieve the product form the server. We need to update this variable here, because we are assigning the product id on the server, and we want to ensure that the updated product with the new product id is stored in the original product variable. Before we can see this in action, recall that we are hard-coding in the id of the product to retrieve for update. We'll change the Get method id to 5, and try out the update operation. Now we can run the application. As before, the Hammer data appears. We can make a change and click the Save button. Our Save Complete message appears. To confirm that the update worked we can click Refresh. Yes the product was changed. Now that we've seen the Update work, let's try out the Create. We'll stop the application and change the id back to 0. And we can run again. The form now appears empty, except for the availability date that we set as a default. Let's fill out the form and click Save. Our Save Complete message appears. To confirm that it worked correctly, let's actually take a look at our data store. We can navigate to the Web API project, open the App\_Data folder, and here is our product.json file. We can scroll down to the product with an id of 5. Here's our hammer. Notice that our update is here in this file, and down at the bottom is our new "Macbook air" entry. We now have the Save process working from front to back.

Summary

This module was all about saving data. We started by building the Web API service methods that we needed. The first touch point was retrieving a single item to update, so we built a Get method in the Web API that returned a single new or existing product. Then we looked at the Post and Put methods. We decided to use POST to save new items, and PUT to update existing items. In each of these methods, we used the repository we created earlier in this course to retrieve or save the data. Next, we reviewed the code required to build an Edit form in Angular. The user enters or edits data on the Edit form. We saw that Angular provides for client-side validation to help ensure that the user enters complete and correct information. The form has a Cancel button to reset the data and a Save button to save changes. We then covered how to call the Web API to save the data. We found that the $resource object provides methods to query a set of items with a GET Request. Get a single item with a GET Request and save a new item with a POST Request. It did not provide a default method for a PUT Request, so we created a custom $resource method that we called Update. We then added code to the Angular controller to call the appropriate $resource method and save the data. This module demonstrated how to save data passed from the Angular frontend to the backend service. One more touch point is complete. You may have noticed, however, that we have no error handling in our Web API code. What if the requested product is not found? What if the passed-in product data is not valid? What if our Web API code throws an exception? Next up, we'll cover error handling, and we'll look at the responses returned to the Angular application from the Web API, including responses for success, failures, validation errors, and exceptions.

Action Results, Error Handling & Validation

Introduction

The Web API methods we have built so far retrieve and save data, but they have no code to handle invalid requests, they have no code to deal with missing or invalid data, and they have no code to catch coding or system exceptions. Welcome once again to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module looks at action results, error handling, and validation in the Web API. It then covers how the Angular code can recognize and take action based on error responses from the Web API. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. We've covered retrieving data, such as a list of products, how to filter, shape, and query data using query strings, URL parameters or OData, and how to save new data or update existing data. This module looks at error handling. We'll see how to catch invalid requests, missing or invalid data, and exceptions in the Web API methods. We'll learn how to use action results to return error information in the response, and in the Angular app we'll add code to extract the error information from that response. Let's look at this with some graphics. Here is our Angular app on the left and our Web API service on the right. We currently have code in place to issue a GET Request from the Angular application to the backend service, and the service responds with the requested product in the body of that response, but what if that requested product is not found? What if the database is offline and the repository generates an exception? We need code in the Web API to return appropriate responses for these cases, and we need code in the Angular app to interpret error responses. What about saving data? We currently have code in place to issue a POST or a PUT Request from the Angular application to the backend service and pass the data to save in the body of that request. Code in the Web API service then saves that data, but what if the product to update is not found? What if the body of the request is missing the data? What if the data is not valid? What if the database is offline, and the repository generates an exception? we need code in the Web API to return appropriate responses for these cases as well, and code in the Angular app to interpret these error responses. This module will set up error handling from back to front. We'll start with a look at Action Result and how to return appropriate responses from the Web API when the request is invalid or the requested resource is not found. And we'll look at how to modify the Angular code to understand and react to these action results. Then we'll look at exception handling. We'll look at how to catch exceptions in the Web API methods and return an appropriate response, and we'll look at the Angular code required to extract exception detail from that response. Lastly, we'll cover validation. We've already seen client-side validation in the Angular form, but the Web API service can't assume that everything is properly validated, so it needs to perform its own server-side validation. We'll look at basic validation in the Web API and how to return validation messages in the response. We'll see the Angular code required to extract validation messages from that response. When we are finished, we will have retrieve and save functionality that can handle error conditions. Let's get started.

Action Results - Server

So what is an Action Result, and why would we want to use it? A Web API controller action method, such as GET, PUT, and POST, can return void, meaning it has no data to return. When using void, the Web API returns an empty HttpResponse with a status code of 204, which is no content. Currently, both our Put and Post methods return void. A Web API controller action method could return some specific <entity type>. Our Get method currently returns IQueryable of product to return a set of products, and our Get method with an id returns a single product object. The defined data is returned in the HttpResponse body. Web API controller action methods could instead return HttpResponseMessage. We would then have control over the exact response message, because the HttpResponseMessage is converted directly into the HttpResponse. In Web API too, we can now use IHttpActionResult. This simplifies the code and makes its intent more clear. The ApiController class that our Web API controller inherits from contains a set of helper methods that return a basic set of built-in action results. What do I mean by helper methods? Here are some of the action result helper methods provided by the ApiController base class. Notice that they cover many of the common types of responses, Ok, BadRequest, NotFound, and so on. Let's refactor our code to return IHttpActionResult. Then we can use some of these helper methods to handle both valid and invalid requests. Here we are in Visual Studio and looking at the Web API ProductsController class. Notice again that it inherits from ApiController. This base class provides methods for returning an action result. Let's start with a Get method. We can change it to instead return an IHttpActionResult. We can then use the Ok helper method to return Ok and pass in the queryable list of customer objects. That way our OData techniques will still work. So we've now refactored our Get method to return an IHttpActionResult. Let's do the same thing to the GetById. We can change the method to return an IHttpActionResult, and use the Ok helper method, passing in the product, but let's take another look at this method. What types of things can go wrong here? Well, the link statement could return null if the requested product is not found. We should check for that here and use another helper method, NotFound. This code now better handles the case when a passed-in ProductId is not found in the list of products. Moving down to the Post method, let's change the void to IHttpActionResult. If the new product was saved successfully, we want to send a created response, so let's add the following code. The Created response is generic, so we can specify the type of item we created. In this case we are creating a product. The first argument to the Created method defines the location of the created resource. It is basically the URL that defines the link back to the new resource. The second argument is the content of the response body, and includes the new product. This is required, because the new product can have values defined by the service. In our case, the service assigns the ProductId, so we are sending the data for the newly-created product back in the response. If you recall, our Angular code is already expecting that product to be returned, but let's take a more thorough look at this method. What types of things can go wrong here? Well, first, the request could be empty, so let's check for a null product. Here we're using the BadRequest helper method to set up the appropriate action result. Looking further down the code, the repository could return a null product if the save was not successful, so let's check for that as well. Here we are returning a Conflict action result. This code now better handles receiving a request body that is null, and handles errors in the save. Lastly, we'll update the Put method. Let's change the void to IHttpActionResult, and return Ok. Now what types of things can go wrong with this method? As with the Post, the request could be empty, so let's check for a null product. We'll, again, use the BadRequest helper method. In this case, as well, the repository could return a null product if the save was not successful. In this code, we're assuming that it was not successful because the product was not found. We could expand this code for more conditions as needed. We now have a set of Web API methods that all return IHttpActionResult. Let's see how to process these responses in the Angular client code.

Action Results - Client

Recall this diagram from earlier in this course? Here is our Angular custom product resource on the left, and our Web API service on the right. In this product resource service we are using the built-in Angular $resource service to communicate with the Web API. We call the Query method of the $resource service to issue a GET Request, and expect to get back an array of products. We call the Get method of the $resource service to issue a GET Request with an id, and expect to get back the requested product. We call the Save method of the $resource service to issue a POST Request, passing the new data to save in the request body. And if you recall, our code was expecting to get the new product back, and we added a custom Update method to the $resource service to issue a PUT Request with an id, passing the data to save in the request body. In this case, we don't expect to get anything back, so how does the Angular code handle action results passed back from the Web API. Let's take a closer look at the method signatures for the $resource methods. Since $resource is a built-in Angular service, we can find the method signatures in the Angular documentation. This text is taken directly from that Angular documentation. Let's break it down. When using Get actions, such as the Query and Get methods of $resource, we call the method using the Resource class. For example, here is the call to the Get method. Notice that the method is called on the productResource class. The first argument is an optional list of parameters. In this case, we are setting an id of 5. The second argument is an optional success callback option. We use this function to process the data returned in the response, or perform any other actions on successful completion of the method. The third argument not used in this example, is an optional error callback function. This is the function we'll define to better handle any error conditions. When using non-GET actions, such as the Save in our custom update method, we call the method using the instance of the item. For example, here is a call to the update method. Notice that the method is called on the vm.product model object instance. also notice that the method is prefixed with a $ in this case. The first argument is, again, an optional list of parameters. In this case we are defining an id of 5. The second argument is an optional success callback function. We use this function to process the data returned in the response, or perform any other actions on successful completion of the method. The third argument, not shown in this example, is an optional error callback function. We want to add this error callback function to each of our methods. Let's add the error callback functions to our $resource method calls now. We are back in Visual Studio with the Angular productEditCtrl file open. We want to add the error callback function first to the Get method as the third argument. In this error callback function, we assign the statusText property of the response to the message property Because the message property is found in the view, the message will appear to the user. Now let's add that same error callback function to the update method and to the Save method. Now, if we get any error information back from the Web API response, the Angular controller will process that response and display error information to the user. Let's give it a try. Let's change our productResource.get to get product with an id of 55. We know that id does not exist, so let's run the application, and we see our Not Found message. You may want to modify your Angular code to display a more user-friendly message. We could check the response.status for a 404, and set vm.message to a more user-friendly message, but what we have sets up the basics for us, so we are now processing the action results from the Web API service. But what if our Web API froze an exception? Then what?

Exception Handling

Code in the Web API project, such as our repository code, could throw an exception. How do we track for that and pass along appropriate information in the response? Since the Web API is C# code, we can use a try catch block to catch any errors that occur, either in the Web API code or in any code that it calls, such as the product repository. We put the Web API code in the try block, and if we catch any exception we use the InternalServerError helper method and pass in the exception information. Just a side note here, notice the ResponseType attribute on this method. I added this attribute to ensure that the Web API help documentation continues to provide detailed help. Otherwise, now that we changed the return type to IHttpActionResult, the help won't be very helpful. Not sure what I'm talking about? I'm talking about these. When we run our Web API project and get this documentation, we can click on the API and here, again, is our product API. When we click on one of them, such as this Get, we see detailed documentation on the Request and on the Response. When we change the return type to IHttpActionResult, this documentation would no longer appear appropriate, unless of course we add the ResponseType attribute. We are back in Visual Studio and looking at the Web API ProductsController class. Notice that I already added the ResponseType attribute. Let's add some exception handling around each method. First, the Get method, so this code tries to perform the Get. If there is any error, it catches it, and it returns an InternalServerError. Let's repeat this code in each of our methods. Here is the Get with an id, here is our Post, and here is our Put. Now we need to add code in the Angular ProductEditCtrl to process any errors returned from the backend service. We already have code in place to display the statusText, but that will simply display InternalServerError. We need a little more code if we want to display the actual exception message. The exception information is returned as part of the response.data property, so we add the exceptionMessage, if any, to the message displayed to the user. We can add this same code to the save and update. In order to test this new code, we need the Web API controller to throw an exception, so let's add a line of code to do just that. Let's go up to the Get method and we'll throw an exception. Let's make sure we're getting a valid product, and run. And we can see our detailed message. Again, we could make these messages more user-friendly, but you get the general idea. Before we continue, let's not forget to delete that throw statement. At this point, if we get any error information back from the Web API response, the Angular controller will process the response and display error information to the user. There is only one thing we are missing yet, validation. That's up next.

Validation

If our application saves data, we want that data to be complete and correct. That's the purpose of data validation. In a web application there are two places to define validation, client-side in the Angular application and server-side in the Web API service. Client-side validation involves checking user-entered values before passing those values to the backend service. Why define validation in the client application? It's immediate. As the user enters data or leaves a field, the user can be immediately notified that there is a data entry issue. They don't have to wait until after they submit the data. It provides a better user experience. Angular has many built-in validation features that allow display of user-friendly validation messages. Plus the messages can be more specific because the client-side application has more information about what the user is doing at any point in time, and it performs better. Because client-side validation runs on the client, it does not require accessing the backend service. Server-side validation involves checking the passed-in values before they are saved to a data store. If the client application already validated the data, why repeat the validation in the backend service? Because the backend service can't and shouldn't count on every client application to provide valid data, and it needs to protect against a malicious user who could potentially bypass the client application. Let's take a look at validation in our application. Here we are again in Visual Studio with the Angular productEditView.html template file open. Recall that this template file already has client-side validation defined. For example, the Product Name is required, has a minimum length of 4, and a maximum length of 12. The productCode is also required. Down here in our button bar we disable the Save button until the form fields are valid, so if all goes well, the data should be valid before the user can click Save. Let's try out the existing validation. If we clear the product field, we see an error. As soon as we type in the fourth character, the error disappears. If we type in too many characters the error message appears again, and notice that our Save button is disabled. As soon as our validation error is gone, our Save button is enabled again, so the client-side validation is working nicely. But for security and data quality control, we want to repeat the validation in the backend service. For field-level validation, ASP.NET Web API provides model validation. Model validation is defined using data annotations. If you have ever used Entity Framework or WPF or MVC, you may already know how to use data annotations. Data annotations are a mature and easy-to-use framework for adding validation attributes to your model. Data annotations are added to model properties like this. Here is the ProductName property from the product model class in the Web API. We defined a required attribute, meaning that the property must contain a value. We defined a MinLength attribute to specify the minimum length of the value, and a MaxLength attribute to to define the maximum length of the value. There are many other attributes available, such as Range, CreditCard, EmailAddress, Phone, and so on. There is also a custom attribute that allows you to build custom validation methods. Let's add these validation attributes to our model in the Web API project. We are back in Visual Studio with the Web API Product Model class open. The first step is to add a using for System.ComponentModel.DataAnnotations. With that in place, we can attribute the model. For the product name we'll set Required, MinLength, and MaxLength. Notice that we have set the server-side validation to the same restrictions as the client-side validation. In a real application this is, of course, what we would want, but in this case we want to be able to test our server-side validation, so we need a different set of restrictions here. That way we can pass the client-side validation and catch the validation in our Web API code. So let's set the minimum to 5 here so it is larger than the length caught by the client-side validation, and we'll set the maximum to 11 so it is smaller than the length caught by the client validation. If we want to define a custom error message to pass to the client application, we can use a named argument here called ErrorMessage. Let's do that here. We can add similar attributes for any of the other properties. Here are the attributes for the ProductCode, for example. I'll space them out and make them a little easier to read. Just setting the Product Model annotations is not enough. We need to add code to check the model state and return an appropriate response if the model is not valid. We'll add that code to the Web API ProductsController. We only need to validate in the Post and Put methods, so before we perform any other processing, we'll check the state of the Model. ModelState is a property in the base ApiController class that we can use to check the state of the Model and determine whether it is valid. BadRequest is one of those helper methods that returns the ModelState as part of the IHttpActionResult. The ModelState is a set of key value pairs that define the validation issues. The key in this case is the name of the property and the value is the validation message. These are serialized in the response to the Angular client application. And we'll put the same code in the PUT Request. So lastly, we need to update the Angular ProductEditCtrl code. In this code we'll process validation errors from the Web API. We only have validation errors in the update and save, so that's where we'll add the code. Recall that the model state is a set of key value pairs that define the validation issues. The key is the name of the property, and the value is the validation message. This code walks through the set of key value pairs and appends the validation messages to our Message property. Now let's give it a try. Let's put in four characters for the Product Name. It will pass the client-side validation, but get caught in the server-side validation where we defined a minimum of 5 characters, click Save, and here's our validation message. Product Name min length is 5 characters. So, we have now implemented action results, error handling, and validation.

Summary

This module was all about handling error responses from the Web API. First we looked at action results. Our methods were returning void or objects. This meant that there was no easy way to return other responses, such as Not Found, so we modified each Web API method to return an IHttpActionResult and used helper methods to set the return values. We then modified the Angular code to add error handling callback methods. Then we looked at exception handling. We added try catch blocks around the Web API code and modified the Angular code to extract exception detail from the response. Lastly we looked at validation. We already had the client-side validation defined in the Angular HTML template, but we didn't have any server-side validation. We used attributes on the Web API Model class to define field-level validation, and then added a check for a valid model object before processing the PUT or POST Requests. Lastly, we added code to the Angular controller to set validation messages returned from the server. This module demonstrated how to use action results to return appropriate responses when an error condition occurs, whether that error was because of an invalid request or a request for a resource that does not exist, or an exception in the Web API code or from validation rules defined on the Web API Model. That completes another touch point. Next up, the touch points around logging in and user authentication.

User Authentication in Web API

Introduction

Most web applications require user authentication. The application wants to verify who the user is so it can provide appropriate information and features based on that user so when you log in to your airline website you'll see your reservation information. Welcome once again to Angular Front to Back with Web API, from Pluralsight. My name is Deborah Kurata, and this module is the first of a two-part series on user authentication. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. We've covered retrieving data, how to filter, shape, and query data using query strings, URL parameters or OData, how to save data, and how to return useful error responses from the Web API service. This module is part 1 of our look at user authentication. We'll cover user authentication in the backend Web API. In part 2, we'll build the Angular login form and call the Web API from Angular to register and authenticate the user, completing this touch point. Before we begin, let's look at what user authentication is and how it is different from user authorization. User authentication is all about identity. Who is the user? Before a user can be authenticated, the user needs to be registered. That is, the user needs to be known somewhere. Maybe that somewhere is our own local membership database where we track all of the users of our application. The authentication process then verifies that the user-entered username is in our database, and the user-entered password matches the password stored in our database for that user. Maybe the user is, instead, registered with a third party, such as Facebook or Twitter or Microsoft. The authentication process then verifies that the user is known to that third party. Our application then does not need to track usernames and passwords. User authorization is all about permissions. What is the user allowed to do and what resources can the user access? Authorization makes use of the authenticated user. Understanding user authentication and authorization can be difficult. Why is that? Well, it has its own vocabulary. Read anything about authentication and authorization, and you'll quickly see that it has its own terminology, identities, claims, certificates, tokens, providers, and so on. There's a lot to learn just to understand the vocabulary. There are many different techniques. There is no one way to handle authentication and authorization. For a public-facing site we could use Facebook or Twitter for user authentication, for a line-of-business application we could authenticate against active Directory or Office 365, or we could store the usernames and passwords in our own membership database on the server. And there is a lack of detailed documentation. ASP.NET Web API generates a set of code to handle both authentication and authorization, but it is difficult to find complete documentation on how to use the generated code. This module provides a detailed walkthrough of user authentication in ASP.NET Web API. First we'll look at what token-based authentication is and how the data flows. Then we'll look at the myriad of authentication options provided by the ASP.NET Web API project template. We'll take a pragmatic approach and implement just one of these authentication options. That allows us to cover our selected option in detail. Lastly, we'll look at the Web API code that is generated when we select an authentication option, and learn how to use the Web API to register a user and login a registered user. The next module completes the authentication story and implements registration and user login in the Angular app. Let's get started with the server side of things.

Token-based Authentication

Token-based authentication is a common way to implement user authentication and authorization and is the technique we will use. Token-based authentication basically works like a wristband for rides at a carnival. You show your identification and pay your money at the ticket booth. If you are authenticated, you are given a token, often in the form of a wristband. This is the authentication process. Every time you make a request for service, such as riding a ride at the carnival, you show your token, your wristband, to prove you have been authenticated. This is the authorization process. You prove you are authorized by presenting the wristband you received when the ticket booth authenticated you. Token-based authentication follows the same basic concept. A user enters a username and password and clicks Login. That username and password is submitted in a POST Request to an authorization server, which acts as a Token Service. That authorization server authenticates the user. The authorization server could be provided by a third party, such as Facebook, Twitter, Google or Microsoft. In the sample application we are building, the user registration information is stored in a local membership database on the server, so the authorization server authenticates that the username exists in the membership database and that the entered password matches the password associated with the username. If the user is authenticated successfully, the Token Service returns a token in the body of the response, but instead of a wristband, the Token Service provides a string of characters. The string is really long, so it's truncated here in the diagram. By default, these access tokens are bearer tokens. The idea of a bearer token is that whomever bears, or basically has the token, is considered to be authorized. When the user requests a resource, such as a product to edit, the GET Request includes that token, proving that the user has been authenticated. If that token is valid, the requested product is returned in the response body and displayed. So this process is the authentication, posting the username and password and getting back an access token, and this process is authorization. The access token is included on future Web API requests to identify the user. The user is then granted access as appropriate to requested resources or operations. So how do we go about building this authorization server to provide tokens? Well, remember at the beginning of this course when we created the Web API project? You may recall that we selected the Individual User Accounts default authentication option when we created the Web API project. By selecting that option, the ASP.NET Web API template automatically created an authorization server that validates the user's credentials and issues tokens. It even creates and maintains the membership database so we don't have to write any Web API code to authenticate the users, we just have to figure out how to call the code that was generated for us. What about third-party authentication? We would not then need a membership database if we used a third-party service. The authorization server could be provided by a third-party service, such as Facebook, Twitter, Google or Microsoft. You may have seen websites that allow you to login using Facebook or Twitter, like the Reuters site shown here. With ASP.NET Web API, we could provide similar functionality in our Angular app. To cover the authentication process in detail, yet keep things simple, we'll use the built-in authorization server. That means we need a Membership database. At Login, the user-entered username and password are authenticated against the user rows in the Membership database, and if the user is authenticated, the authorization server returns an access token. But how do we get the user data into the membership database? One option is to pre-populate the database with a list of users. This is a good option if we control the users that can access our application. Another option is to add a register feature. Users can then register themselves. The registration process adds a row to our membership database for each registered user. For the registration process we need another API. The Account API can handle user account operations such as registration, changing a password, and logging out. The account API, then, inserts updates rows in the membership database. Luckily for us, the ASP.NET Web API template generates the code for this as well, so we get all of this functionality without writing any server-side code. Because it is so important to the topic of authentication, let's back up for a moment and look at the ASP.NET Web API template again and the many authentication options it provides.

Authentication Options

User authentication is such a common requirement, that ASP.NET Web API has it built-in as part of the Web API template provided by Visual Studio. At the beginning of this course we created our Web API project. At that time, we saw this dialog. We selected the Web API as the desired template for our project. Here on the right is the button to configure user authentication. Click this button to select the desired type of user authentication from a list of provided choices. When you click the Change Authentication button, this dialog appears. Let's define the scenarios appropriate for each choice. No Authentication is just that, no generated authentication code. We could use this option if the application is open to all and does not require any user identification. For example, an application for a restaurant may want to provide open access to all users without any type of login. We could also use this option if we don't want the Web API template to generate any code for us. We would then need to write all of the authentication code ourselves. Individual user Accounts is the best option if you want to provide for a local login or a social login. With a local login, we manage user profiles in a database on the server. We could pre-populate that database with a set of valid users or provide a registration feature so users can register themselves. A social login allows the user to login by way of an existing account on Facebook, Twitter, Google, Microsoft or some other provider. Using an existing social membership is nice, in that we don't then need to store the users credentials so we have less information that we need to keep secure. Both the local login and social login use something called OAuth2 to authenticate requests. OAuth is an open standard for user login authorization. OAuth2 is the next evolution of the OAuth protocol, and focuses on simplified and improved authorization workflows. Select the Organizational Accounts option in situations where the users sign in with Azure Active Directory, Office 365, or on-premise Active Directory credentials. The code generated by the Web API template is then specific to the option selected on this dialog. Windows authentication is for intranet-only scenarios. It makes use of the users Windows login for authentication. Our sample application uses the Individual User Account option with a local login. This option was chosen for our sample application because it provides an example of user authentication without the need for Azure or Office 365. At the beginning of this course, when we created the Web API project with the Individual User Accounts authentication option selected, the ASP.NET Web API template generated a significant amount of code for us. Now let's look at that code and learn how to call it from our client application.

Web API Authentication Code

When we selected Individual User Accounts in the ASP.NET Web API project template, the template added an Authorization Server based on OAuth2, it added an Account API for managing user accounts, including registration operations, and it defined an Entity Framework model for generating the database used to store user account information. Note that you don't have to know anything about Entity Framework to use this code, and you don't have to use Entity Framework in the rest of your code. Our sample application doesn't use Entity Framework. Entity Framework is just what is used behind the scenes to automatically build our Membership Database. Let's take a look at the code. Here we are in Visual Studio. Before we look at the code, let's look again at the API documentation. Remember how to bring that up? We set the Web API project as the StartUp Project and set the Web API project properties to start on the current page, then we run, and select API. And here is the API for the Account controller. This controller is part of the scaffolding that the Web API template generated for us when we selected to use the Individual User Accounts authentication option. It provides features such as logging out. We could tie this to a logout button in our Angular code. Notice that there is no login feature here. We'll look at how to log the user in, later in this module. Scrolling down we can see that there is a method to register a new user. We'll use this registration feature to add authorized users for our application. Looking at this page gives you an idea of the features that the generated Web API code provides. But recall how I mentioned the lack of detailed documentation? Let's click on the Register API for more detailed information. So here is the detail for the Register API. It does tell us that it expects an Email, Password, and ConfirmPassword. We'll need to know these parameter names when we call this API from our Angular code, but to really understand what it's doing, let's look at the code. Before we start looking through this code, I want to clarify that I did not write any of this authentication and authorization code that we are about to see. It was all generated when we created the ASP.NET Web API project because we selected the Individual User Accounts authentication option, so I am not going into detail on how to write this code since you don't need to write it. Rather, we are going through it so that we understand the basics of how it works and where we may want to customize this generated code and, of course, how we're going to call it from our Angular code. So with that, let's start with the controller. Under the Controllers folder is the AccountController. This is the controller we will use for our registration process. Let's jump to the Register method. We'll use this Register method to register a new user. This is the method we looked at in the API documentation. The Register method is expecting a model that contains the user information. The API documentation provides detail on what to pass to this Register method, or we can just look in the code, right-click on the RegisterBindingModel, and select Peek Definition. Here we can see that the code expects this passed-in object to include three properties. These are the same three properties we saw in the API documentation, and these are the properties that our Angular code will need to pass to this method. Notice that the generated code uses the attributes we discussed in the prior module to set up validation for the three fields. Let's close this window and look further at the Register method. First, the code checks the ModelState to ensure that the property values are valid. Recall that we used this same technique in the last module to validate our product properties. This code then sets the username and email address fields to the passed-in email address, so it assumes that the entered username is an email address. And the code calls a method to create the user. Let's Peek definition on UserManager. Let's Peek definition on ApplicationUserManager. The ApplicationUserManager is defined in the IdentityConfig.cs file. ApplicationManager inherits from UserManager, which is part of the built-in ASP.NET identity features. Let's promote this file to a tab and look at it further. The IdentityConfig.cs file contains the configuration information for the UserManager. Here is where you can specify the validation logic for a username. In this example, it has AllowOnlyAlphanumericUserNames set to false, and RequireUniqueEmail set to true. This is also where you can specify the validation logic for the password. This code defines a minimum length of 6. The password must contain a non-letter or digit, the password must contain a digit, a lowercase letter, and an uppercase letter. If these validation requirements are too restrictive for your application, change these values as needed. Now let's Peek Definition on ApplicationUser, and Peek Definition on IdentityUser, and Peek once again on the generic IdentityUser. Here is some of the Entity Framework code that defines our membership database so you can see the list of fields that will be generated. Again, we don't need to know anything about Entity Framework to use it for our membership database, it is all built-in and hidden behind the scenes. Look how far we needed to drill down to even find it. Before we move on, let's look at the startup configuration for authentication and authorization. We'll open the App\_Start folder. The Startup.Auth.cs file contains the code to configure the OAuth2 authorization server. If we look at the bottom of this file, we can see that this is also where we could enable logging in with a third-party login, such as a Microsoft account, a Twitter account, Facebook or Google. We'll stick to the built-in Token Service. This unassuming code is the magic that provides user login authorization for our Web API. Our Angular code will send a request to this /token API. This is the URL that our app will use to get bearer tokens. Where is that /token API set up? Let's Peek Definition on the ApplicationOAuthProvider. This generated code automatically sets up an OAuth2 authentication server within our website. This authentication server is often called a token service because it authenticates the user and returns an access token. That access token is a bearer token, meaning that the bearer of the token is assumed to be authenticated. Let's close this for now. Looking at some of the other code here, notice also that we are allowing insecure HTTP. That allows us to easily test this code and view the requests and responses. Before going to production, we would want to change this to false to ensure that HTTPS is used for secure HTTP, otherwise, someone could obtain the user's token, and as the bearer of the token could impersonate the user. To summarize, here are the main application classes that were generated to implement authentication and authorization features in the Web API service. Let's look at this with our picture again. First, let's walk through the registration process. The user enters their username, which in this case is an email address and a password. When the user clicks Register, we issue a POST Request to the Register method of the Account API. The Register method validates the entered data, and if it is valid, creates an entry for the user in the Membership database. Once the user is registered, they can login at any time. The user enters their username and password. When the user clicks on Login, we issue a POST Request to the Authorization Server. The authorization server checks the membership database for the entered username and verifies the password. If the user is authentic, the authorization server returns a bearer token. The Angular code will store that token and use it when requesting protected resources. So we already have everything we need in the backend, and now know what we need to call from the frontend. Let's hook it up.

Summary

This module was all about token-based user authentication in ASP.NET Web API. We started with a diagram detailing the basic flow of token-based authentication and compared it to getting a wristband at a carnival. The user enters a username and password. That username and password is submitted to a token service. The token service authenticates the user and returns a token. The token is then included each time the user requests a resource, to prove that the user was authenticated. We saw how to configure the Web API project for user authentication. We looked at the four options that Visual Studio provides for user authentication. We selected the Individual User Accounts option for our sample project. The ASP.NET Web API template then generated the server-side code that we needed. Lastly, we walked through some of that generated code and learned how to use the Web API to register a new user and login a registered user. Here again are the touch points that we originally identified for communication between an Angular frontend application and a Web API backend service. This module covered user authentication, but only the backend side of things, so that only gives it a half checkmark. Now that we have learned how to use the generated Web API code, we are ready to register and login a user. That's coming up next.

User Registration and Login

Introduction

If an application requires user authentication, then it often needs a login feature, and if the application tracks its own users, then it also needs a register feature to sign up new users. Welcome, once again, to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module is the second of a two-part series on user authentication. At the start of this course we identified the key touch points between an Angular frontend application and a backend service. We've covered retrieving data, how to filter, shape, and query data, how to save data, and how to return useful error responses from the Web API service. The last module was part 1 of our look at user authentication. We covered user authentication features in the backend Web API service. That gave it a half of a checkmark. This module is part 2. We'll build a login form and call the Web API from Angular to register and authenticate the user, completing this touch point. This module covers the following topics. We'll build a login form for entry of the user's id and password. The form will provide both user registration and login operations. We'll also create the associated Angular controller. Next we'll implement a user registration feature. We'll build an Angular service for submitting a registration request to the Web API and call that service from the Angular controller. Lastly, we'll implement the login feature. We'll enhance the Angular service to submit a login request to a backend token service, and we'll get back an access token that we can use for authorization. We cover user authorization in the next module. When we finish this module, we'll have user authentication in place from front to back. Before we get started with the client-side code, let's review what we learned in the last module about calling authentication features in the Web API. First let's walk through the registration process. The user enters their username and password. When the user clicks on the Register button, we issue a POST Request to the Account API. Look at this URL. This request is routed to the Register method in the AccountController class. We looked at this method in a prior module. The Register method validates the entered data, and if it's valid, creates an entry for the user in the membership database. Once the user is registered, they can login at any time. When a registered user accesses the application, the user starts by entering their username and password, then clicks on Login. We issue a POST Request to the authorization server. Look at this URL. This Request is routed to the token service that was created for us by the ASP.NET Web API template. We looked at the code that configured this service in the last module. The authorization server checks the membership database for the entered username, and verifies the password. If the user is authenticated, the authorization server returns an access token. The Angular code can store that token and use it when requesting protected resources. All of the code for these features is in place. We covered that in the last module. In this module we will build all of this. Now that we know what we need to do, let's get started.

Building a Login Form

Let's start the client-side coding by adding a login form at the top of our main page. The form has data entry fields for the username, which in our case is the email address and a password. The form has buttons for login and for registration. To login, a registered user can enter their username and password and click Login. To register, a new user can enter their username and password and click Register. In a real application, the Register button should take the user to a separate data entry form. That way there is room to collect more information, such as the user's name or mailing address. Then the form could also provide a text box for the user to confirm entry of the email address, but this form is good enough for our purposes, so let's build this form in Angular. Here we are back in Visual Studio, and we're looking at the Angular application. Let's open index.html. We're going to add our login form directly to this file. First we'll add ng-controller to define the Angular controller for this page. This is the controller that will handle the registration and login operations. Notice that we are using the controller as syntax to define the controller. We'll put the login form here in the navbar. You can put it somewhere else if you would prefer. I'm going to paste the code and then we can walk through it. Here is the form. We use ng-hide to hide the form once the user has successfully logged in. We'll need to remember to add the isLoggedIn property to our controller. Here is the textbox for the username, and since the username in this example is the email address, we'll bind it to the email address property of the controller. We are ignoring client-side validation, but you could add it here for a better user experience. Here is the textbox for the password. Again, you would want client-side validation here. We have two buttons on this form, Login to log an existing registered user to the system and Register to register a new user. Note that for line-of-business applications the users may already be pre-defined in the database and a Register button may not be needed here. Lastly, we have an area at the bottom of the form to display any messages to the user. We'll use this to display validation and other messages. Let's make one more little change. Let's ensure that the EditView is only displayed when the user is logged in, so we'll add an ng-if here. Now the productEditView won't appear unless the user is logged in. Now let's create the associated controller. We'll right-click on the app folder, define a new JavaScript file, and call it mainCtrl.js. So we don't forget, let's add the script tag for this file to index.html. I'll paste in the basic code for this controller and we can talk through it. As with all of our controllers, this code starts with an IIFE or immediately-invoked function expression. Then we get a reference to the module and register this new controller with that module. This controller will use a service to handle the userAccount functions, such as the registration function. We'll call that service userAccount. This service is similar to the productResource service we created earlier in this course, and defines the $resource object needed to communicate with the Web API service. We'll create that service in the next clip. Here we'll inject it in as a dependency. At the top of this function we defined the basic properties for the model. isLoggedIn will define whether or not the user is logged in, message will define any messages to display to the user, userData defines the data to submit to the Web API as part of the registration process. How do we know what fields we need here? Recall from the last module that we can run the Web API project to view the project documentation. From that we saw that the Register operation requires email, password and confirmPassword properties, so those are the properties we define here, and we added username as well since we'll need that for the login feature later. Recall that our Register button called a registerUser function, so that's the function here, and our Login button called a login function, so we're defining that function here. We'll complete this code throughout the rest of this module, starting with this registerUser function. Let's add that code next.

Registering the User

In the last clip we added a login form with the Register button, but why do we need a registration feature? Before the user can be authenticated, the system needs to know about the user. If we are using a third-party token service, then we would not need a registration feature because the service, such as Facebook or Twitter, already has the user registered. But if we were using our own membership database for user authentication, then we'd need a way to get the user information into that database. If we wanted complete control over the users that can access the application, we could pre-populate the membership database. Then we also would not need a user registration feature, but by providing a registration feature we can allow any user to register themselves. We then store their data in our membership database. That's what's going on in this sample application, so you can register yourself and gain access to the application. Let's walk through the registration process in detail, then we'll write the code. The user enters their username, which in this case is an email address and a password, and the user clicks on Register. We then POST a Request to the Register method of the Account API. As we saw in the last module, the ASP.NET Web API template generates this AccountController code for us. We just need to POST the Request. Recall that the POST Request expects a user object with email, password, and confirmPassword properties. The Register method in the AccountController validates the entered data. In the last module, we looked at the server-side validation for these properties. If the user-entered data is valid, the Register method creates an entry for the user in the membership database. So where is the membership database coming from? Well, you'll see that when we call this method the database is created automatically. So what do we need to build exactly? We created the login form as part of our main page and index.html. The Register button called the registerUser method in the controller. We then created an Angular controller and called it mainCtrl. We defined the needed properties and the registerUser method, though that method is currently empty. The next thing we need is an Angular service to communicate userAccount requests, such as registration, to the Web API service. We'll call that service userAccount and it will use the built-in Angular $resource service, similar to how we built the productResource service earlier in this course. Let's build this userAccount service now. We are back in Visual Studio looking at the Angular project. In the Common folder let's add a new JavaScript file called userAccount.js, and so we don't forget, let's add the script tag for it to index.html. Next we'll add the code for the userAccount service. Again we start with an IIFE, an immediately-invoked function expression. We get a reference to the common.services module that we created earlier in this course and define a new service factory called userAccount. We inject both $resource and appSettings into this service. We'll need the built-in Angular $resource service to communicate with the Web API, and we'll need the appSettings constant we defined earlier in this course, because that defines our server Path. The code in this factory defines a custom $resource method called registerUser. The technique we're using here is similar to the custom update method we defined in the productResource service earlier in this course. This registerUser method will post a request to api/account/Register. Now let's go back to the mainCtrl, and we're ready to add the code to call this userAccount service method. Recall that we need to pass three fields to the Register method of the Account API, email, password, and confirmPassword. Since we didn't provide a textbox for confirmation of the password, we'll just set the confirmation password to the user-entered password. That completely defeats the point of the confirmedPassword, I know, but you can enhance this as needed if you add a full registration form to the application. Next we'll call the userAccount.registerUser method from our new service. We'll pass in the userData, which contains the properties that are required by the Register method. The second argument to the registerUser method is the callback function for a success. In that function we'll set a success message. We'll clear the confirmPassword property since we don't need it anymore, and if we want the user automatically logged in when they are successfully registered, we can call the login function here. Let's use what we learned earlier in this course when we talked about error handling to provide messages for exceptions or validation issues. We'll do that by adding the optional third argument to this method so our registerUser method is complete. Before we can give this a try, let's ensure that the Startup Projects are defined for the solution and the Web API project properties have the Start action set to don't open a page. And let's keep an eye on our App\_Data folder and see when the membership database is created. Click on the APM.WebAPI project and click the Show All Files button. We can see that the only file in this folder right now is the JSON file we are using for our product data store. Now let's give it a try. And we see our new Login form. Let's type in some data. So the username it wants an email address, so I'll put in something very creative here, and for password I'm just going to type abc. Do we think that's going to work? Let's click Register and find out. And we see that we have validation errors. We didn't add any client-side validation, so these errors are coming from the Web API service. Let's look at the request details using the developer tools. We'll click to capture network traffic, click Register again. Here we can see that we're issuing a POST Request to Account/Register. Double-click to view the details, and here we can look at the Request body. Notice that the Request body contains the email, password, and confirmation password in plain text, so anyone could sniff the network and view this information. Hence the reason we should ensure that the site uses HTTPS before deploying this application, but it is very helpful to see this plain text when debugging. Now let's look at the response. We can see that the Response body is passing us back information on what happened. The request is invalid, it's saying that the password must be at least 6 characters long. This is coming from our ModelState. Recall from the last module that we have an extensive set of requirements for our password that needed an uppercase letter, a lowercase letter, a non-letter and digit, a digit, and so on. Before we try it again, let's peek at Solution Explorer. Notice that we don't have a database yet in our App\_Data folder, so now let's try it again with a better password. This time I'm typing in Abc\_123. That should meet all of our validation criteria. So let's go back to the summary and click Register again. You'll notice that the first time the registration is pending for a significant amount of time, but the registration was ultimately successful, and if we look at Solution Explorer we can see that we now have an mdf file in our App\_Data folder. That's our membership database. So let's close our browser and take a look at our database. To look at the mdf file, we can say View, Server Explorer. Under Data Connections here is our database. We can drill down and we can see under our Tables that Visual Studio automatically created a set of tables for us. If we drill down to the ASP.NET users, you can see all of the columns provided in the user table. To see the user we added, we can right-click and select Show Table Data, and here is the user we just added. Here is the email address we entered, and of course, it hashed our password, amazing. So the generated code did a tremendous amount of processing for us. First it validated the user's entry, so we saw if we put in a password that didn't match the criteria that it gave us back and error message. Once we did put in valid information, Visual Studio created a database with a set of tables, and it wrote that data to that database, and then it returned a success response to our Angular app. And recall from the last module, we didn't write any code to make all of that happen, amazing. So if we wanted to pre-define additional users, we could simply add them to this table, and now that we have a registered user, we can log that user in.

Logging the User In

Logging the user in is an important feature in many web applications. Let's walk through the Login process in detail, and then we'll right the code. A registered user enters their username and password. When the user clicks on Login, we issue a POST Request to an authorization server. Notice the URL here. Recall that the Account API did not provide a login function. Instead, the login process is provided by an authorization server. The authorization server was created as part of the ASP.NET Web API template. It provides token-based authentication. The Angular code will call the authorization server using the URL, /Token. The authorization server checks the membership database for the entered username and verifies the password. If the user is authenticated, the authorization server returns an access\_token. This token is a bearer token, meaning whomever bears or has the token is assumed to be authenticated. The Angular code will store this token and use it when requesting protected resources. User authorization is covered in more detail in the next module. The authorization server expects a very specific POST Request. Here you can see the body of that request. The first thing you may notice is that the request is not in JSON format, rather it is URL encoded. Look at how it changed the @ sign to %40, and each property is separated by ampersands. By default, the Angular $resource service that we're using in our code will convert the user data object to JSON format, so we'll have to write some extra code to encode that data and format it as appropriate. Let's look at the properties here. First is the userName. We already defined that property as part of the user data. Second is the password. We have that property already as well. The third property is grant\_type. The grant\_type defines how the authentication will be performed. We'll specify a grant\_type of password, meaning that we'll pass in a username and password, and the authentication is performed using that information. To submit the appropriate Request body, the POST Request Header must define the appropriate Content-Type. As we saw in the prior slide, the body of the Request is urlencoded, so we'll need to specify in the header that the content-Type is urlencoded. Let's add this POST Request to our code. We are back in Visual Studio looking at the Angular code. Let's start by changing the userAccount service to include the resource for the login. We can't just add a second method on here. This won't work because the login process uses a completely different URL, it's not using /Account/register. So we need to change the userAccount service to return two different properties, one for the registration and one for the login. So the code, instead, needs to look like this. Instead of simply returning a $resource with a registerUser method, we are now returning two properties. The registration property returns a $resource with a registerUser method that posts to api/Account/Register, and we have a login property that returns a $resource with a loginUser method that posts to /Token. But this code is not enough. Recall from the slides that we need to change the header of the request to specify a Content-Type of urlencoded. We do that by adding a Headers property to the loginUser method defined here. This headers property sets the Content-Type in the header as shown in the prior slide. But we're not quite finished. Recall also that by default the built-in $resource will convert our Request body to JSON. We instead need to urlencode it so we can transform that Request body here using the transformRequest property. The transformRequest property provides a function that can transform the data any way we want it before adding it to the Request. We added code to this function to urlencode the properties and separate them with ampersands. Now that we've changed this userAccount service, we need to change the mainCtrl function that calls this service. Instead of simply calling registerUser, we now need to call registration.registerUser, and we'll need to add a call to our login.loginUser to log the user in. Let's open the mainCtrl and look at the registerUser function. So now, instead of calling userAccount.registerUser, we now need to call userAccount.registration.registerUser. Again, this change is needed because of the way we changed the userAccount factory service. Now we're ready to add the code for the login function. I'll paste the code and then we can talk through it. This first line of code sets up the grant\_type. We talked about the grant\_type in the slides. This defines how the authentication will be performed. We're defining a type of password, meaning that we will pass in a username and password. Next we are defining the username. During registration, we bound the username textbox to an email address, but the password style grant\_type requires a userName property, so we set the userName here to the user-entered email address. Then we called the loginUser method. Recall that we need to call it using userAccount.login.loginUser. The second argument of the loginUser method is the success function. This is the code that will be executed after successful login. We, of course, set isLoggedIn to true, clear any messageText, clear the password since we shouldn't need it anymore, and most importantly, we store the access\_token obtained from the Response body. We'll see how to use this token for user authorization in the next module. The third argument to the loginUser method is the function to execute if an error is returned. We use similar error handling techniques to those we implemented in the module on error handling and display any messages in the messages area of the form. Now we are ready to give it a try. Run the application, press F12 to bring up the developer tools, and click to monitor network traffic. Login with the credentials used to register the user, click the Login button, and it works, the Product Edit page now appears. Let's look at the requests and responses. So here is our request to the token service. You notice it's a POST Request and it got a 200 result. Double-click, you can see that it was a POST Request and the Content-Type is urlencoded. Now let's look at the Request body. We can see that the Request contains our encoded and ampersand-delimited data. Now let's look at the Response. Here is the Response header, we received a 200 OK, and looking at the Response body, this very long string of text is our access\_token. Notice that it says it's of token\_type, bearer, and so on. Cool, let's try this again with the Chrome browser. And we're not getting anything. Let's press F12, and we got an error, Bad Request, No 'Access-Control-Allow-Origin' header is present. Didn't we fix this earlier in this course? Well, yeah, we fixed it for our ProductController, but not for the token service that was generated by Web API. So how do we fix this? Well, we need the token service to add the 'Access-Control-Allow-Origin' header when the provider issues the token. So let's find that code. It's in the Web API project in the Providers folder, ApplicationOAuthProvider.cs. We're going to look at the method called GrantResourceOwnerCredentials, and here is the magic line of code. This line of code adds a Response.Header of Access-Control-Allow-Origin, and specifies a URL. where did this URL come from? It is the port where our Angular client application is running. It is the same port we granted access to in our ProductController. See this URL here, this is how we enabled CORS in our ProductsController. For the OAuthProvider that is our token service, we do it with this code here, but use the same port number. Now let's try running again. We can login, click the Login button, and here's our Product page. Our login was successful. Now we have user registration and user login working in both IE and Chrome, Yay!

Summary

This module detailed how to implement user registration and login. We started by building the login form. The form provides both user registration and login operations. We also created the associated Angular controller. We then implemented a user registration feature. We wanted to allow the user to register themselves. They're entered data was stored in our membership database on the server. For our sample application, we built an Angular service for submitting a registration request to the Web API, and called that service from the Angular controller. Lastly, we built a user login feature. We enhanced the Angular service to submit a login request to a backend token service. This required a little extra work because of the specific requirements of the request required by the token service. We then retrieved the provided access token from the returned response. Now we can use that access token for authorization, which is coming up in the next module. Here again are the touch points that we originally identified for communication between an Angular frontend application and a Web API backend service. This module completed our discussion of user authentication, so another touch point is now complete. With that, we have an authenticated user. Let's look at how to use the resulting access token for user authorization.

User Authorization

Introduction

In the prior module, we looked at how to log a user in and obtain an access token using token authentication. Now we can use that token to control which data and features that the user can access. Welcome, once again, to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and this module demonstrates how to implement user authorization to limit access to our backend resources. At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. We've covered retrieving data, how to filter, shape, and query data, how to save data, how to return useful error responses from the Web API service, and how to implement user authentication. In this module, we'll use the access token returned from the user authentication to authorize access to specific resources and features. We'll start this module by first looking at how to protect a resource. In other words, we'll look at how to restrict access to the method in our controller. We can define no restrictions. This is useful for any methods that should be publicly available, features such as view the list of stores near you. We can restrict a method to logged-in users. The method is unavailable to any user that has successfully logged in. We can restrict a method to a specific user role. The method is then only available to users with that role. For example, some features may only be available to administrators. We could even restrict a method to specific users. Once we have our resources protected, we look at how to access protected resources. As you might guess, we'll use the access token received when the user logged in. Let's get started.

Protecting a Resource with the Authorize Attribute

We can protect a resource such as our product data by using the Authorize attribute. Authorize is an attribute that can be applied to a controller or action method in the Web API. In this example, our Get method with an id has the Authorize attribute set. When a controller or action method has the Authorize attribute set, all requests to that controller or action must be authenticated. In other words, each request must include the access token that was returned by the authorization service when the user logged in. Otherwise, authorization to the controller or action method is denied and the Web API returns a 401 unauthorized error. The Authorize attribute has several optional arguments. The Roles argument allows us to set the user roles that are authorized to access the controller or action method. For example, a method could be open only to administrators and support personnel. The Users argument allows us to set the specific users that are authorized to access the controller or action method. In this example, only these two users are authorized. In most cases, it would be best to use Roles and not tie the source code to specific users, but that option is available if needed. These options give us a high degree of control over who is authorized to access the resources exposed by our Web API. So let's add the Authorize attribute to our Web API project. Here we are in Visual Studio with the Web API project node open. Let's look at the AccountController first. This controller was generated by the ASP.NET Web API template and provides user account actions such as logout and the Register method we're using to register a new user. Here we can see that the Authorize attribute is defined on the AccountController class itself. That means that none of these methods can be accessed unless the user is logged in. But how then could we call the Register method? Here is the Register method. Notice that it uses the AllowAnonymous attribute to allow anyone to access this specific method. So if we want to protect the majority of our Web API methods, we could set the Authorize attribute at the class level and use the AllowAnonymous attribute to open up access to specific methods as needed. Now let's open our ProductsController. This is the controller we created earlier in this course for our products. Let's set the Authorize attribute on the Get method that has the id. Once this attribute is set, the user of our Angular application won't be able to access this method unless the user has been authenticated. And before we try to run it, there's one more thing. If you have been working along with this sample application since the beginning of this course, you may recall that we commented out a few lines of code from the WebApiConfig file. That file was in the App\_Start folder. Up until this point we wanted to access our Web API methods with no authentication, so we commented out these two lines. Now that we are ready to limit access to our methods, we need to uncomment out these lines. Okay, now let's run it and see what happens. We'll log in so we are authenticated. When the Product Edit form comes up, we see the Unauthorized error. Why is that? We've logged in. Well, the ProductController method doesn't know we logged in, because we are not yet passing the access token as part of the GET Request, so the ProductsController Get method doesn't know that we have been authenticated. Let's look at how to pass the access token we receive from the authorization service to the ProductController Get method.

Accessing a Resource Using an Authorization Header

As we saw in the prior clip, we can set the Authorize attribute to restrict access to a controller class or action method of our Web API in the backend service. If we restrict access to our Web API, then only authenticated users can access the Web API method from the Angular client application. How do we tell the backend service whether the user is authenticated? We need to pass the access token we received from the authorization server during login to that Web API method. To pass the access token in, we place that token in the Request Header. The token indicates that the user has been authenticated, and if the token is valid, the user is authorized to access the restricted Web API method. Let's look at that in a diagram. We've seen this diagram before. Now let's use it to look at the flow required if the Get method of the Web API is marked with the Authorize attribute. The user enters the username and password and clicks Login. As detailed in the prior module, the Login process submits a POST Request to the Authorization Server. Since we are managing our own users, the authorization server checks the membership database looking for the defined id and matching password. If the data is valid, the user is authenticated and an access token is returned in the response. The Angular code then stores that token for later use. Our Product Edit page is then displayed. The ProductEditController issues a GET Request with an id to the Web API to retrieve the product to edit. The stored "access\_token" must be included in the header of the Request. If the Get method has the Authorized attribute set, such as in our case, the Web API looks in the Request Header for that access token. If the token is not found, we get an Unauthorized error. That is why we saw the Unauthorized error in the last demo even though we had logged in. We did not yet include the token in the Request Header. If the token is found in the header, the user is authorized and is given access to the Web API method. The data is returned in the response and displayed on the form. So to let the Web API know that the user is authorized, we need to pass the access token in every request to the Web API. To include the access token in every request, we need to do some refactoring of our Angular code. Currently we are storing the access token as a property of our mainCtrl. We want to make that token more accessible to other parts of the application by storing it, instead, as a property of a new Angular service. Recall that all Angular services are singletons. This means that if we store user data as properties in a service, those properties are available to any code that injects the service as a dependency, so we'll create this new custom Angular service called currentUser. This new service will retain basic user information such as the username for display on the page as desired, and the access token for use in our Web API service requests. Then we'll need to modify the mainCtrl to use this new service. Lastly, we need to modify the productResource service to pass the access token in the header of the request. Let's take these one at a time. Here we are back in Visual Studio with the Angular website node open. We'll add a new JavaScript file to the common folder and call it currentUser.js. And, so we don't forget, we'll add the script tag for it to our index.html file. In this new currentUser service, we'll define key customer data and methods to get and set that data. I'll paste the code in and then we can talk through it. Here at the top we get the reference to the common.services module and register our new factory service. We then define the properties for our user profile. We want a flag that defines whether or not the user has logged in. We want to store the username in case we want to display it somewhere. And, of course, we want to store that access token that we receive when the user successfully logs in. The setProfile method sets the username, token, and isLoggedIn property values. The getProfile method returns the profile data. That's all there is to this service. Any Angular code that injects this service can then get or set the user profile information. Next, let's refactor our mainCtrl. We'll start by injecting this service as a dependency into our mainCtrl. Then let's look at our login method. Upon successful login, we'll set the username and access token using our new service. So the mainCtrl no longer needs to store the access token. We also want to change the hard-coded isLoggedIn property to access the user profile from our new service, so we no longer need to set it here or here, and we'll change the reference here to reference the service. So now that isLoggedIn is a function, we need to change the binding for it in the index.html file. We need to change it here and in our ng-if here. The last piece of code we need to refactor is our productResource service. Here is where we need to define the header of our requests to include the access token. First, we'll inject the new currentUser service as a dependency into our productResource service. That will allow this service to use the retained access token from the currentUser service. Alternatively, we could set up an HttpInterceptor for this, but we'll explicitly define the headers here for clarity. We want to set the headers for the get, save, and update. Though currently we only have the Authorize attribute on the GET, we would want to add it to POST and PUT as well. So it's best to set the headers for all of them. We don't have the definition for the Get and Save methods here, because they are provided by default, but now that we need to set the headers, we need to specify them here. For each of our methods, get, save, and update, we are now including the access token in the request header. We set the Authorization property in the header to 'Bearer', and the token string we receive from the authorization service when the user logged in. So, we built a new custom Angular service called current User to retain keys or information, we refactored the mainCtrl to use that new service, and we modified the productResource service to pass the token in when we call the get, save or update methods of $resource. Notice that we are not passing the token for the query method that returns all of the products. You could add that method here if desired. Now let's run the application. Let's login, and we see the Product Edit page successfully. Let's try the same thing in Chrome. Before we login, let's press F12 to display the developer tools, then let's login. You can see our page is successfully displayed here as well. Let's take a look at our HTTP Requests. The first Request was to our Token service. You can see the Header of the Request has a Content-Type form-urlencoded, and here is the response. See our access\_token here, DqkZ? The second HTTP Request was to display the productEditView. The next Request we're interested in is this one here for the products. If we go back to the Header, you'll notice that in the Request Header we have our Authorization token, and it's set to Bearer, and our long access token string. If we look at the preview of the response, we can see that our data is returned here. So, we have now successfully implemented basic user authorization.

Summary

This module was all about user authorization. We saw how to protect Web API controller action methods. All we needed was the Authorize attribute. We can fine-tune the authorization by specifying arguments in the Authorize attribute. We can authorize to specific users or specific roles. With the protection in place, we locked ourselves out of the Product Edit page and saw the Unauthorized response. Then we looked at how to access protected resources. We already had the user authentication in place from the last module, and we are receiving an access token from the authorization service. In this module, we refactored our Angular code to store currentUser information, including that access token, in a new currentUser service. Since a service is a singleton, we can access the user information from any code that injects the service in. We then modified the mainCtrl to use this new service, and we modified the productResource service to pass the access token in the header of our requests. Here, again, are those touch points that we originally identified for communication between an Angular frontend application and a Web API backend service. This module covered user authorization, so our last touch point is complete. Wow, we've covered a lot of ground here. One more short module, and we are finished.

Final Words

Introduction

As you have seen throughout this course, Angular and ASP.NET Web API play very well together. You can now leverage what you've learned in this course to masterfully implement all of the touch points between your client-side Angular application and your backend Web API service. Welcome, once again, to Angular Front to Back with Web API from Pluralsight. My name is Deborah Kurata, and the final words in this course include a brief summary and pointers to additional information. Let's jump right in to this short module.

Touch Points

At the start of this course, we identified the key touch points between an Angular frontend application and a backend service. You may have seen this slide before. We started with a simple touch point, retrieving data. For the backend, we built the Web API service and implemented a Get method. In that Get method we retrieved a set of products. For the frontend we used the built-in Angular $resource service to submit a GET Request. That GET Request would be routed to the Get method in the Web API and return the set of products to our Angular app, but it didn't work. In Chrome, we were missing an Access-Control-Allow-Origin header, so we implemented cross-origin resource sharing or CORS. And we had a problem with the properties. The model in the Web API uses Pascal case, so the property names were capitalized. The Angular code uses camel case, so the bindings were not capitalized. With this mismatch, we could not see the list of products on our Angular page. We were able to fix this problem by configuring the data serialization formatting. We got the retrieve working, and our first touch point was complete. We then looked at how to filter, shape, and query data. In our first implementation, we passed parameters from the Angular app to the Web API using query strings, or by extending the URL. This allowed us to filter the list of products by passing a search string to the Web API. But then we embraced the power of OData queries. We found it was exceptionally easy to add OData query support to our Web API Get method. We could then send OData queries to that Get method from our Angular app. This allowed us to define much more complex queries. With that, our second touch point was complete. Saving data was next. For the backend we implemented two methods in the Web API. We defined a Post method to save new items and a Put method to replace existing items. For the frontend, we again used the built-in Angular $resource service. We used the built-in Save method of $resource to send a POST Request, and created a custom Update method to send a PUT Request. With that, we completed our Save touch point. But we noticed that something was missing. We weren't handling error conditions, so we changed our Web API methods to return ActionResult. This allowed us to provide more detailed information in the response. Then we looked at exception handling. Since Web API is just C# or VB.NET code, it supports try catch blocks, so we put try catch blocks around our Web API code, and if an exception occurred, returned an appropriate response. And we covered validation. We already had client-side validation in place using the powerful Angular validation features. For server-side validation we set attributes on the model properties and checked the ModelState in the Put and Post methods. That provided basic error handling for our application and another touch point was complete. Next we turned our attention to user authentication. The ASP.NET Web API template generated an extensive amount of code for us. It created an AccountController class that provides features such as user registration and logout. It generated an authorization server that provides token-based authentication and returns an access token we can use for user authorization. The Web API template even provided a membership database for us on the server. For the frontend we added code to call the AccountController Register method to register a new user. We also added code to call the authorization server Token service to login a registered user. On successful login, the Token service provides an access token in the response. One more touch point off the list. Lastly, we covered user authorization. We saw how to protect our Web API methods using the Authorize attribute, and how to access those protected methods from the Angular code by passing the access token received from the login process into the Request Header. That completed our last touch point. In this course we covered each of these touch points from front to back.

For More Information

For many of the topics we covered in this course, such as OData and security features like user authentication and authorization, there is much more to know than what we could cover, so here are some additional Pluralsight courses that may be of interest to you. Feel free to pause if you want to add any of these to your playlist now. And for additional information I have set up a Github repository for this course. It is called AngularF2B, for front to back, Web API. It contains the start files for this course to help you code along with the demos. I'll also add notes and reference materials as needed.

Final Words

Congratulations! You have completed the journey from front to back with Angular and Web API. Thoughts or comments? Please use the Discussion tab on the Pluralsight page for the course to leave your feedback. I check there often. Thanks for listening.

Course author

Deborah Kurata

Deborah Kurata is an independent consultant who specializes in designing and developing successful Microsoft .NET applications as well as mentoring software developers. She has authored several te...

Course info

LevelIntermediate

Rating

(1031)

My rating

Duration5h 15m

Released16 Apr 2015

Bookmark

Add to playlist

Share course