Introduction

Introduction

AngularJS is the framework for building responsive client-side web applications. With its neat integration with HTML, its emphasis on clean programming techniques, and its easy-to-use services, it is the perfect choice for developing those full-featured business applications on the web. Welcome to AngularJS Line of Business Applications. My name is Deborah Kurata, and this Pluralsight course takes you step-by-step through the process of building line of business web applications using Angular. This introduction answers three basic questions. Why use Angular? What are the key factors in building a line of business application? And the key question is How? What are the steps required to build an Angular line of business application? Let's start with the why.

Why Use Angular?

So management decides that applications for key business operations will now be on the web or in the cloud. That will allow company personnel to use the application on a pad device while walking around the plant or warehouse, on a phone when out of the office, on a laptop from home, and of course from the computer on their desk, and it's your job to make it all happen. If you have ever looked at web development with trepidation, especially when building the types of interactive business applications that users expect, you may have wished that there was a better way. With Angular, your wish is granted. Angular can help you love web development. AngularJS is a client-side JavaScript framework for building interactive applications for the web. Angular brings simple and clean back to the development of complex web applications. Angular was originally developed by Google and is now open source. You can download it at the web address shown here. If you already have an ASP.NET or other web application, angular plays well with others, so you can use Angular to augment your current web applications. So let's answer the question Why Angular? Here are the top reasons why Angular should be the tool of choice for building line of business applications on the web. Angular helps make your HTML expressive. As developers, we often spend much more time reading code than writing it, so making the code more clean and readable improves our overall productivity. Angular extends HTML giving it simple logic like If statements and repeat loops. This makes the HTML expressive and easier to read and leads to less errors and easier maintenance over time. Angular is modular. Angular guides developers into a modular approach to building web applications. Even complex business applications can be divided into manageable units so that no matter how large the application gets over time, the code stays clean, controllable, and testable. Angular provides rule-based navigation. Business applications often include rich navigation for moving between the pages of the application. With Angular, you simply configure the rules, and Angular handles navigating to the correct page and loading the associated code logic, all without server-side postbacks. Angular has powerful data binding. You define a model with your data fields and any HTML element can be bound to that model, and that binding is two way, meaning as the user changes the data in the UI, the data fields are updated automatically. No more writing code to populate the controls in a form and no more code required to read updated data from those controls. It's all automatic with Angular binding. Angular is testable. Angular was designed from the ground up to be testable, and with its focus on separation of concerns, the logic is separate from the user interface making it easier to test that logic. And Angular is popular. Popularity in and of itself is not a reason to use a framework, but there are many side effects to Angular's popularity. For example, its easy to find help. With so many developers now using Angular, there are lots of resources available to you. There are blog posts and training videos on Angular, including several Pluralsight courses. There are many independent consultants with Angular experience, like me, that can help with your application. There is an active stack overflow forum to help with issues that you may run into. These are all great reasons to use Angular, but what is unique about building line of business applications? We'll see that next.

Key Factors in Building a Line of Business Application

Building a line of business application often requires different considerations and techniques than building general purpose web applications. One critical difference is that in business applications, data is an asset. Data is an important part of most applications, even general purpose applications. A weather application stores your location, and a ToDo list application stores your ToDo list. But for line of business applications, data is often more than that. Data is a critical corporate asset. The application may store your list of customers and their orders or manage employees and payroll or track important or company proprietary research or define the products that your company sells. In many cases, the amount of data is significant. Whereas a weather application may track your favorite five locations or the ToDo application may track a dozen items on your list, corporate data can include hundreds or thousands of rows or more. So you need techniques for helping the user filter that data, and even with filtering, you may need to display hundreds of rows of data, so you need techniques for paging through the data only bringing down a reasonable number of rows at a time. Often a line of business application has significantly more data entry fields. The weather application may only require entry of the zip code for each location, and the ToDo application may require a task name and due date. But a corporate data entry form may include dozen of fields. You've probably seen desktop applications that have six or eight or ten tabs worth of data that the user must enter. What is the best way to display that much data in a web application? How about web-friendly tabs? And unlike a desktop application, with a web-based application, you have to worry about the user closing the browser before completing and saving. You can imagine the scenario. Say I am a user entering data on tab five of eight. Then, Oh look! I have five new tweets! I open another browser tab to look at them, get distracted, and ten minutes later, I close the browser and think to myself, Now what was I doing? I'm going to be pretty unhappy if I then realize I just lost all of my work, so you need techniques for saving partially completed data. In a line of business application, data integrity is critical. You enter the wrong zip code into a weather application, and you end up with the weather for Kalamazoo. No big deal. You enter invalid data into your ToDo list, and your list may not appear correctly. But invalid payroll information? Or incorrect product data displayed on your company's website? These can be catastrophic. They can result in corporate fines or tax consequences or expensive lawsuits or people losing their jobs. Validation, then, becomes a key requirement, as are helpful data entry controls such as datepickers and masked edit fields that guide the user to enter valid data. And of course data visualization. For companies to succeed, they need thorough data analysis. Good data analytics help guide a company to achieve its goals. So in addition to good web design, line of business applications need to focus on managing, manipulating, validating, and displaying the data. Note that all of the screen shots shown in this clip are from the application that we will be building throughout this course. That leads us directly to the last question, How? How do we build a line of business application with Angular?

How to Build a Line of Business Application With Angular

Once you decide to use Angular for your application, the next question is how. What are the tasks required to build an Angular line of business application. As with any software project, the first step is to collect the business requirements. Those requirements define the set of application features. This process considers design issues. This is also the time to define the architecture for the application. How will the Angular application be structured? Where is the data stored? What type of back-end service is required? A big part of every web application is the user interface layout. This task defines the look and feel of the application. It includes defining the page layout called the View and adding some style. Luckily the Bootstrap framework from Twitter is available to help style the application. And since most line of business applications have multiple pages, there needs to be a mechanism for navigating between those pages. In Angular, navigation is defined through a process called Routing. Data is a key part of most line of business applications. This task involves writing the code to retrieve and save application data, and since most client-side web applications don't have direct access to a data source, this task includes calls to a server-side web service that fetches and updates the data. Most line of business applications require data entry forms. This task involves defining the layout for the forms, validating the form data, and submitting that form data to a server-side web service for processing and storage to a data store such as a database. Business applications often have specialized business logic such as calculating tax or a profit margin. This task is to build custom Angular services to encapsulate the business logic for the application. That business logic can then easily be used or reused throughout the application. Management often wants features in the application to help them make operational decisions. These features are often called data visualization or data analytics. This task involves building charts to show important business analysis, such as profitability. These are the tasks we will cover in this course. We will also look at a few cross-cutting concerns that cut across these tasks. Concerns such as exception handling, unit testing, and security. You'll continue to see this diagram as we work through all of these tasks throughout this course. Now we know the answers to our three questions. Why use Angular? Well, we defined many reasons including its expressiveness and modularity that lead to clean and maintainable code. What are the key factors in building a line of business application? Data is a key business resource. A line of business application needs to focus on managing, manipulating, validating, and displaying that data. And how do you build a line of business application with Angular? We identified a set of tasks required for most line of business applications. Let's conclude this introduction with a look at the prerequisites for the course and the course outline.

Course Outline

This course expects that you already know some basis HTML syntax including these tags plus other common tags such as table, div, anchor, and image. You should know some basic cascading style sheets or CSS techniques, including use of the style attribute and the purpose and use of the style class. But you will be able to follow along without much CSS knowledge. The course expects that you have some basic JavaScript knowledge, specifically JavaScript syntax, such as this code. But you only need to know the basics. When intermediate level JavaScript techniques are used in this course, they will be explained in detail before being used in the sample application. And, finally, this course assumes you have had some exposure to Angular. Specifically, Angular directives like the one shown here, and Angular binding expressions. While this course does expect that you have been exposed to Angular, it reviews Angular's syntax and techniques along the way to reinforce and strengthen those concepts and then applies them to the sample line of business application. Now on to the course outline. This course takes you through the process of building a web-based line of business application with Angular. The course begins with this introductory module. The next module walks you step by step through building the first page of the application. The demonstrations build the View, Model, and Controller for the page. Along the way, we review basic Angular concepts, such as directives, binding expressions, and modules. We also review basic JavaScript concepts, such as immediately invoked function expressions or IIFEs. You'll also see how to give the page some style with Bootstrap. Business applications are all about data, so next we'll look at accessing data with Angular. The demonstrations show how to use an Angular service to call a server-side web service to retrieve data. Since we are only building the client-side application, we'll then mock the web service so we can execute the application without the web service in place. That will give us an operational Angular application that displays one page. But web applications are often many pages, so the course then introduces Routing and why you want to use it for navigation. The sample application is extended to add a menu and additional pages. And Routing is added to move between the pages of the application. Line of business applications often use forms for entry and edit of data. The demonstrations build a data entry form using Angular's two-way data binding. It also demonstrates more complex HTML elements such as a masked edit control and a datepicker. Before submitting a form to the server, the form data should be validated. The demonstrations here validate the data in the form on the client side before submitting the form data to the server. The course then demonstrates how to build a custom Angular service for the application's business logic. Using a service for business logic keeps it encapsulated and separate from the forms, and a service is easily accessible from the controllers making it easy to use and reuse the business logic anywhere in the application. To make good operational decisions, management often wants to see the data. The demonstrations in this module show you how to represent application data using charts. It would be nice if the code never had runtime exceptions and if the server-side web services were always available and operational, but that is not the case. So this module walks through adding exception handling to the application. The final module provides a brief review and looks at some additional cross-cutting concerns such as unit testing and security. Let's get started.

Building the First Page

Introduction

Every great journey begins with that first step, and every great web application begins with that first page. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module walks you step by step through building the first page of the application. Along the way, we review Angular concepts and syntax to reinforce and strengthen your Angular knowledge. We expand on those concepts throughout the remainder of this course. The primary objectives of this module are to turn the business requirements into a view, model, and controller, organize the project structure for the application, build the view using appropriate directives, define the main module for the application, build the controller, define the model, implement the methods for the actions on the page, and style the view with Bootstrap. A secondary objective of this module is to ensure that the basic Angular concepts are clear and well understood as they will be used throughout this course. Let's get started.

Turning Business Requirements Into Application Features

Let's imagine for a moment that we are working on a real project at our company, Acme inc. We sell products to the cartoon industry. Our first step is to collect the requirements and use those requirements to detail appropriate user stories. During the requirements-gathering step of this project, we identified a Product List View feature for the product managers. This Product List View was selected as our first iteration and will be developed in this module. Here is a simplified version of the user stories for this feature. As a product manager, I can browse the list of products to review key product information. I can optionally see the product images. I can select a product from the list to view and validate the product details. And I can select a product from the list to edit and update the product details. From these user stories and from further discussions, we mocked up a user interface design. Notice that this design meets the requirements of the user stories. The user can browse the list of products to review key product information. The user can optionally see the product images. The user can click on the product name to view the product details. And the user can click the Edit button to edit the product details. This defines the view for the first page of our application. We can see from the view that the application requires a specific set of product data fields such as an image, a product name, a code, availability date and price. In an Angular application, the data is often defined in JavaScript object notation or JSON. The JSON structure for this page defines an array of products. The array is denoted with square braces. Each product in the array is defined within curly braces--{}--and separated by commas. I only have one product defined here. That's all that would fit on the screen. The dot dot dot indicates where I would have added a second product. The data for each product is defined with name value pairs separated with a colon. First is the product ID with a numeric value of 1, then the product name with a value of Leaf Rake. Then there's the product code, the release date, a description, the cost and price, which are decimal values, category, then the tags. The tags are defined in an array of strings. These are the search tags for the product. Lastly is the image URL. This links to the location of the image. So this is the model for the first page of our application. So now we have defined the view and the model. Next we define the controller. An Angular controller defines the data and the code for the actions in the view. So a controller has two primary roles. The controller defines the model. The primary purpose of most Angular controllers is to define the model required by the view. So the controller is said to contain the view's model. And, hence, an Angular controller is sometimes referred to as a View Model. The controller also implements the methods for any actions that the view can perform. So the controller contains the code for the user interface actions. For the product list controller, the model is the list of products. To define the model, the controller needs to retrieve this set of products. In a web application, this is most often achieved by calling a back-end web service. We'll talk more about retrieving data in the next module. The product list controller implements the method required to hide and show the images. The Project Team had voiced concerns about the size of the images, both the size that the image takes up on the screen and the physical file size. When hiding the images, the products are displayed in a smaller amount of vertical space. Hiding the images can also improve performance because the image files don't need to be downloaded. We have now defined the controller for the first page of our application. Let's summarize this with a picture. At this point, we defined an Angular application with a view, a model, and a controller. The view contains the HTML and basic Angular directives. In this case, the ng-app directive defines the application as an Angular application. The ng-controller directive identifies the controller associated with this view. If you are not yet familiar with directives, no worries. We'll look at them in detail shortly. The controller is shown on the right. It defines the data model that the view will use, and the methods for any actions that the view can perform. We now have defined the view, the model, and the controller for the first page of the application. The first step for building this application is to create the project's structure. Let's do that next.

Organizing the Project Structure

If you are building a simple little application, it does not matter much how you set up your project directories. But most line of business applications are neither simple nor little. Logically, organizing the directories for your web application project has many advantages. Every member of the team knows where everything goes. This provides consistency and leads into the next point. Each person can quickly find and edit files as needed. By following the logical structure, everyone knows where everything is. Over time, the structure can readily be extended as the project and code base grow. For a more in-depth look at organizing your project structure, see the Pluralsight course entitled AngularJS Patterns: Clean Code. This course uses the bi-feature approach, meaning that the directories are organized around the application features. Here is a high-level view of the directory structure for the Product Management application that we are building throughout this course. The app directory contains the code specific to this application. The app directory has subdirectories for each primary feature of the application. For this Product Management application, we have two primary features--prices and products. For larger applications, these directories can be broken down further as needed. The common directory contains code common to all parts of the application. The services subdirectory defines the location for common services, such as data services or business logic services. The CSS directory contains all of the cascading style sheets for the application. The fonts and images directories are, well, self-explanatory. The .js directory is only for external JavaScript files, such as Angular or other third-party libraries. Notice that Angular.js is already included in this folder. Index.html is the only file created for the application so far. It contains all of the references to the CSS and JavaScript files, and it's the launching point for the Product Management application. Let's launch an editor and take a look at the index.html file. The editor you're seeing here is called WebStorm, which is the editor that I will be using throughout this course, but you can use any editor, even Notepad. But using a more advanced interactive development environment or IDE, such as WebStorm or Microsoft's Visual Studio, can aid productivity by providing coding assistance and a full-featured debugging experience. Notice here that I have the index.html file open. This is the first code file that will execute when running the web application. In the head tag, I have given the application a title, Acme (that's our fictitious company name) Product Management. I have also added a reference to Angular.js, which resides in the js folder. I had previously downloaded and put the copy of Angular.js into that .js folder. In the body tag, I have identified this application as an Angular application by simply adding the ng-app directive. A directive is a special Angular marker on an HTML tag that tells Angular to run or reference some JavaScript code. They apply special behavior to attributes or elements in the HTML. All of the built-in Angular directives begin with ng-. Directives are a major part of the power and beauty of Angular. They are simple to define in the HTML yet provide an extensive set of features and capabilities. It is the set of directives that make the HTML so expressive and easy to read. We'll see that more as we go along. Setting the ng-app directive on an element declares that everything inside that element belongs to the Angular application. So in our application, everything in the body tag is part of this Angular application. We can actually run this application, and we can see that it doesn't display anything, but our title is indeed appearing here in the tab. If we open the Developer Tools, we'll be able to watch the startup process in detail. So let's refresh the page. The network tab of the Developer Tools shows us the network traffic. When the application starts, the index.html is downloaded from the web server and parsed by the browser. We can see that here. As part of the startup process, the browser fetches each script referenced in the index.html file. In our case, we reference the Angular.js script, so the browser retrieves and executes that script. When the browser finishes parsing the index.html page, Angular looks for the ng-app directive. If Angular finds the ng-app directive, Angular initializes the application. But since we don't yet have any other visual elements or directives, the application does not do anything at this point. So let's create our view.

Building the View

In general, view refers to the user interface of the application. It is often used to describe both the HTML that contains the user interface markup and the page generated from that HTML. But you'll find that some AngularJS documentation makes a distinction. A piece of HTML that contains the user interface markup and Angular directives is called a template. Angular parses and processes the HTML from the template. The loaded transformed and rendered result is then called the view. The view is sometimes referred to as a "live template" because the individual parts of the view are dynamic. Throughout this course, I use the term view in the more general sense referring to both the HTML and the generated page. With that cleared up, let's create our view. Here we are back in WebStorm. We'll add the markup for our view directly into the index.html file. The Product List View uses an HTML table to display a header and the detail product rows. I'm going to paste the HTML in, right inside this body tag. So here's the code. As we walk through this code, you'll get a sense for how Angular helps make the HTML more expressive and easier to read. Let's start at the top. We've already talked about the body tag and the ng-app directive. Inside the body tag is our table, that's a standard HTML table. Inside that is a Thead tag. That's also a standard HTML tag, and we're defining our headers. The Tbody tag defines the body for the table. Inside the body is a TR tag. This tag uses the Angular ng-repeat directive to repeat a row for each product in the set of Products. So let's see how we would read that. So we're building a TR element, and we're going to repeat that element for each product in the set of Products. That's pretty expressive what that's doing. But where is Products defined? Since we don't yet have a controller, we can define the list of products using the Angular ng-init directive. We'll put that directive in as an attribute of the table tag. Any directives or binding expressions within the table will then be able to access the list of products. So I'm going to replace the table tag here with a revised table tag. I'm using the ng-init directive. I'm setting it equal to an array of products. And I'm defining that array of products. Does that code look familiar? It's the product data we defined for our model. So when prototyping or just starting out, you can use the ng-init directive to define the model directly in the HTML. That allows you to run the application with some test data before connecting to a back-end service. So let's go back down and look at the rest of the Tbody tag. So we're repeating a row for every product in that set of products. We have two products in that list, so we should get two rows. Inside that row, we've got our columns, which are defined with a standard TD, or table data tags. Each is bound to a property of the model using the double curly brace--{}--syntax, which is an Angular binding expression. These binding expressions allow you to insert dynamic values into the HTML. A binding expression is a JavaScript-like code snippet that can perform calculations, append strings, or reference model properties. In our case, we're referencing model properties. The binding expression syntax tells Angular to replace the text content of the associated HTML element, in our case the TD element, with the value of the defined expression. In this example, the text content of each TD element is replaced with the value of the defined product property. The first TD tag contains an image. The source property of the image is bound to the image URL property of product. The title of the image is bound to the product name. So that's all the code we need. Let's run it and see what happens. Okay, that worked, but it doesn't look very nice. The images are way too big, and the price should be shown as a currency. Let's fix that now. So we'll fix the image by giving it a style attribute, so that defines a width for our image and gives it a margin, and we can fix the price. Angular provides a set of filters that format or filter the value of an expression for display to the user. We'll use the currency filter that's built into Angular. The filter is defined with a pipe character, the filter name, in our case Currency, and optionally any filter parameters. We don't have any parameters needed for our currency. So let's run again and see the results. Ooh, that looks better. Just to be sure everything worked okay, let's open the Developer Tools. Whoops! What happened here? We're getting Failed to load resource: the server responded with a status of 404 (Not Found) And look at that address. Oh, that's our binding expression. %7B is the left curly brace--{. So it's {{product.imageURL}}. The application is trying to load the image before Angular can evaluate the binding expression. Then when Angular does evaluate the binding expression, the images are correctly retrieved and displayed. How do we stop that intermediate error? Well, let's go change the source attribute here to the np-src directive. That way, the browser won't attempt to find the image until after Angular evaluates the binding expression. Now when we run again and display the Developer Tools, you'll notice that there are no errors. We can refresh the page. We can see that it brought down the two images, but what we have now is more of a prototype than a real application. If we want to build a full-featured Angular application, we need to define that application using a module. We'll do that next.

Defining the Module

A module defines an Angular application. Most Angular applications have one main module, but they can reference any number of other modules. For example, the main Product Management application is defined with the main module. That main module can reference a second module that contains a reusable set of common code. A module tracks all of the application code. This is accomplished through a registration process. As each piece of the application code is built, it can register itself with the module. We'll see how that is done as we add controllers and other code to the application. The module also tracks all of the dependencies on external modules or libraries. This is accomplished when the module is created, as we'll see shortly. Bottom line: As its name infers, the module keeps the application modularized. To create a module, we use the Angular Module Method. The Angular Module Method has two overloads. When defining a module, the Module Method requires two parameters. The first parameter is the module name. Since this is the main module, we set it to the application name. The second parameter is the array of dependencies. A dependency is a reference to any other module or library that this module uses. These dependencies are also known as injectables because Angular injects them into the parts of the application that need them. If there are no dependencies, an empty array must be provided. Our application does not currently have any dependencies, but it will. We'll see how to set up dependencies in the next module. This syntax is sometimes referred to as the Setter Method for the Angular module because it is the syntax used to set or define the module. Use the second overload with one parameter to look up and reference an existing module. The parameter in this case is the name of the module to look up and reference. This is useful when registering other code, such as controllers, as we'll see later. The syntax is called the Getter Method for the Angular module because it is used to get the reference to that module. Let's create our main module now. Here we are back in WebStorm. By convention, the main module is defined in a file called app.js. In our directory structure, that file is defined under the app directory, so we can right-click on app, select New, JavaScript file, and we want to call it app.js. Let's declare the module and assign the result to a variable called app. We'll use the Setter Angular Module syntax. The first parameter of our Module Method is the name. We'll call it Product Management. The second parameter is the set of dependencies. We don't currently have any dependencies. Now that we have a module, we want to modify the ng-app tag to reference the module by name, so we'll go to index.html and where we have our ng-app tag, we're going to set it equal to our module name, and there is it. This ensures that Angular launches this module as the main module. We also need to ensure that the new app.js file is downloaded, so we can add a script tag for it. If we run the application, it runs just as it did before. But let's look at this code again. What have we done here? We've added the app variable to the global name space. We've always heard that global variables are bad, right? How do we fix this? Well, we fix it with something called an IIFE. An IIFE, or immediately-invoked function expression, is a JavaScript pattern that helps prevent global declarations. You may sometimes see it called a self-executing anonymous function. In JavaScript, variables and functions defined with the var keyword are added to the global name space giving them global scope. Global variables make it too easy to accidentally reuse the same variable name. With JavaScript's carefree attitude, it won't notify you when this happens. So you might find weird bugs in your application. If you instead define variables and functions within another function, those variables and functions are local within that function, so the idea with the IIFE pattern is to put all variable and function declarations within another function, that immediately-invoked function expression. This is what it looks like. Let's give the IIFE a try and talk through its syntax in detail. Here we are back in WebStorm with the app.js file open. Let's start by defining a function around the module. Now the app variable is local to this function and not added to the global name space, but by putting the code within another function, the code in this function won't be executed. How do you execute a function? By putting parentheses after the function declaration. The function then becomes self-executing, but notice the syntax error here. The parser then treats the function as a function declaration instead of a function expression and throws a syntax error because a function declaration requires a name. If you are not familiar with the difference between a function declaration and a function expression in JavaScript, I wrote a blog post about it. See the link to my blog at the beginning of every module. The accepted way to tell the parser to expect a function expression is to just wrap it in parentheses. In JavaScript, you can't put declarations within parentheses, so the existence of parentheses implies that the contents must be an expression. So we're going to put one there and there. The syntax errors are now gone. And there it is, an immediately-invoked function expression, or IIFE. You'll see this pattern used throughout this course. Before we move on, let's add one more thing inside our IIFE. Inserting use strict here puts the entire contents of the IIFE in strict mode. In strict mode, JavaScript will catch some common coding mistakes and throw exceptions. For example, using an unassigned variable will cause an exception. Let's try this out. Let's remove the var from in front of app. Notice that app now turns red, and it says Unresolved variable or type app. If we comment out the use strict, the undeclared app variable is no longer in error. So by using strict, you can catch some of the more common errors. So let's put our var back, and there we go. Now we are ready to build the controller for our view.

Building the Controller

The purpose of an Angular controller is to provide support to the view. The code in the controller defines the model required by the view. The controller retrieves data as needed and exposes it for the view to use for binding. In our application, it will provide product data. The controller takes the place of the ng-init attribute that we are currently using for our binding source. Code in the controller also implements the methods for any actions that the view can perform. In our application, the only action currently defined for the view is to hide and show the images. So our controller will provide a method to implement that action. How do you build a controller that can define the model and implement the actions for view? By first defining a controller function. The controller function contains the definition of the model and the methods for any actions that the view can perform. Once the controller function is created, the view identifies its controller function using the ng-controller directive. The ng-controller directive specifies the controller by its logical name, not its physical file name, even though in many cases, those names are the same. For Angular to find the associated controller function, the controller function can be defined in the global name space. This technique is often used during prototyping when the controller function is defined directly in the HTML file. For real applications, the controller function is instead registered with one of the application's modules. The module can then look up the function by its name. Since we are building a real, albeit fictitious application, we will use the registration technique. Using the registration technique, the steps to creating a controller are as follows: Start by defining the immediately-invoked function expression, or IIFE. This function will wrap around the entire contents of the controller and ensure none of the controller code is defined in the global name space. Also, though not shown here, be sure to add use strict, then look up the module. We need to register the controller with the module, so we must first have a reference to that module. We can look up the module using the module Getter syntax. That returns a reference to the module. Then we can register the controller with the module. This is done using the controller method of the module. The first parameter of the controller method is the string name of the controller, in this case, ProductListController. The second parameter is an array. The first elements in the array are the string names of the parameters to the controller function. In this case, there is only one parameter, and it's named $scope. We'll talk more about $scope shortly. The last element of the array is always the controller function itself. The purpose of defining the parameter names in this array is to ensure that if the controller code is ever minified, Angular can understand the parameter names. If you're not familiar with the minification process, it is the process by which JavaScript files are reduced in size by removing white space and reducing the size of variables, including function parameters. Minification will change the function parameters to one-character names, but Angular expects specific parameter names. By defining the parameter names in an array, Angular will correctly handle the controller parameters. If you are certain that the code will never be minified, you can skip the array here and instead pass the controller function in as the second parameter. The last step of course is to write the actual code for the controller. That code will go here. Before jumping into the code for the controller, let's talk more about the controller parameter, $scope. $Scope is a built-in Angular object that provides communication between the view and the controller. In this diagram, the view is shown on the left, and the controller is here on the right. By defining $scope as a parameter to a controller function, Angular injects the built-in $scope object into the controller, something like this. The controller then augments the scope, adding the model for the view binding, and defining methods for any view actions. The directives and expressions in the view can then bind to the model properties or call the controller methods. But there is an alternative to injecting $scope. Angular version 1.2 introduced a way to bind the model and methods to the controller directly instead of using $scope. This technique is often called the controller as syntax. Using this syntax, you don't need to pass $scope into the controller. The models and methods are defined directly on the controller itself. In the view, the model and methods are referenced using an alias that is defined in the ng-controller directive. The $scope still exists, but it's behind the scenes. And you'll see later in this course that it is still needed by some third-party modules. Notice the slight difference in the ng-controller directive syntax. It now has an As clause that defines the alias. Hence, the controller as moniker. Many examples define the alias as vm, for view model, because it's short, and its purpose is clear, but you can use any alias name. For example, in this case, we may want to just use Product. The directives and expressions in the view can then bind to the model properties or call the controller methods using that alias and a dot syntax. When building a controller, there are a few best practices to keep in mind. Define each controller in a separate .js file. This means that each controller script file will be separately downloaded from the server, so each controller needs a script tag in the index.html file. Consider adding that script tag immediately after defining the new .js file. Otherwise, it's easy to forget. By defining each controller in a separate .js file, you keep your concerns separated. Most variables and functions in JavaScript are CamelCase, but controller should be Pascal Case. That means the first letter of each word of the name is capitalized. Suffix the controller with Controller or Ctrl. This recommendation has had its share of controversy. Some developers don't want to use abbreviations. Some developers don't want to use suffixes at all. This course uses Ctrl as a suffix because it's shorter and makes it easier for the code to fit in the screenshots. My recommendation is for the team to decide on a convention and stick to it. The final recommendation is to always wrap controllers in an immediately-invoked function expression, or IIFE, just like we did with the module. In some cases, it may not be needed, but by always having it, if code is later added to the controller, or the controller is re-factored, the IIFE will always be in place. So let's create our controller.

Building the Controller: Demos

Here we are back in WebStorm. The product list controller that we're going to build is part of the product functionality, so let's navigate to app, products, then New, JavaScript File. We'll call it productListController. We don't need to capitalize the file name, just the controller function name. So we don't forget, let's add the script tag for this new JavaScript file now. We'll go to index.html, and we'll add our product controllers, so there's our first one. Let's go back to the controller and as per our steps, we'll start with the IIFE, the immediately-invoked function expression. Note that we added use strict as well. Next, we'll look up the module. We called the module productManagement. So we use the Angular.module and pass in the name that looks up that particular module. Now we can register the controller with the module. So we have a reference to the module, we can just type in .controller and the first parameter is the name of the controller. Now we're going to use the controller as syntax, so we don't need the $scope injected into the controller. Since there are no parameters to the controller function, we don't need an array here. We can just define the function as the second parameter to the controller method, so we can declare our function directly here. Let's make one more little change. Instead of defining the function as the second parameter to the controller method, let's instead just pass in a reference to the function. So instead of the function here, we're going to type ProductListCtrl. That's going to be the reference to our function. Then we'll just create the function separate from the registration. And in this case, we need to give it a name. It needs to match this name here. So the controller method of the module takes the string name of the controller as the first parameter and a reference to the controller as the second parameter. And we can then define the contents of the function separately down here. I think that makes the function easier to read and work with, but it works either way. Now we just need to write the code inside the controller function. There are two things we need to do in this controller. We need to define the model for binding to the view and define the methods for any actions in the view. When using the controller as syntax, the model is defined on the This variable. So let's define a variable that references This. We're going to say var\_vm = this. Defining this in a separate variable at the top of the function ensures that the code references the appropriate This. If you are familiar with This in JavaScript, it can sometimes provide unexpected results, especially within child functions. By assigning a different variable to This upfront, that variable then can be accessed within any child function. As with many examples you will see, I am naming this variable vm for view model, but it could be named anything. We could name it Product here. We then need to augment that variable with the model and any methods needed by the view. Let's start with the model. We need to define an array of products. We already have that array defined in the ng-init of the view, so let's cut that array from the ng-init and paste it here. So we'll go to our index.html file, we'll cut this Products out of here, and we'll go to our controller and define vm.products =, and there it is. We are now augmenting the vm variable with the model, which is our list of products. As you might expect, this code should be replaced with code that actually retrieves the set of products. We look at data access in the next module of the course. So for now, this will do. Now that we have the data defined in our controller, the next step is to let the view know about the controller. To hook this controller to the view, we use the ng-controller directive. So let's go back to index.html, remove the rest of this ng-init tag, and instead add in ng-controller tag. And we'll specify our controller. We're using the controller as syntax, so we're going to add As and an alias. We are using vm here as the alias for the controller, but that could be any name. We also used vm in the controller itself here. But that variable name and this alias do not have to be the same. Now that we have an alias, we need to change products to use the alias. So this is going to be vm.products. We don't need to change the other bindings because they are using the local variable here, product, not a property directly on the model. The application should now run as it did before. Let's give it a try. And there it is. Now that we have the application working with our new controller, let's take the next step. Let's provide a method to turn on and off the images. So I'm going to go back to the controller. First, let's define another variable on the model that tracks whether the images are currently shown. So we'll just add that to the bottom here, vm.showImage = false. Then we can define a function that toggles that variable. We'll call that function Toggle Image, so vm.toggleImage =, and we'll define the function. So all this is going to do is inverse the Boolean, and there it is. So that function will toggle the showImage variable on or off. Now we can use that variable and call that method from the user interface. We can go to index.html, and let's start by changing the header of the image columns, so instead of having a word here, let's make it a button. So, we're using the button tag defining it to be of type-button, and we're defining ng-click. Ng-click is another Angular directive. That directive is executed when the button is clicked. So when the button is clicked, we're calling vm.toggleImage. And since it's a function, we use the parentheses to actually execute the function. Notice the use of the vm alias here. That is needed because this method is part of the controller, and the controller uses the vm as its alias. Now we need to actually hide and show the image. We can do that adding the ng-show directive to the image tag. So here's our image tag, so we're just going to add ng-show. This code shows the image if the variable showImage on the model if that's true. Otherwise, the image is not shown. Notice the right-hand side of the ng-show directive is not specified with the double curly braces. The double curly braces are used on the ng-source. When a directive expects a value, like ng-source, double curly braces are required when assigning it to an expression. If the directive already treats the right-hand side as an expression, then double curly braces are not required. Directives such as ng-show, ng-hide, ng-click, and ng-if all expect expressions and, therefore, don't require double curly braces. So we have enough here now. Let's give it a try. So, click the Image button. There are images. Click it again, they go away. It works. Let's make one more change. It would be clearer to the user if the button said Hide Image or Show Image. So let's add an expression for the button text. So up here in front of Image, we're going to say either Hide or Show. So if vm.showImage is true, that means that the image is currently shown, so we're going to add the word Hide. Otherwise, we're going to add the word Show. So here's an example of using a JavaScript expression inside the double curly braces. Let's give it a try and see if it works. There we are. So if we want to show the image, we click, and if we want to hide the image, we click. And it works. But let's open the Developer Tools and see what is actually happening here. Notice by default, we set Show Image equal to false, so it shouldn't show the image. We're going to click refresh, but notice that the images still came down. If the requirement was to hide the image to improve performance and cut down on data charges, our code here is not meeting the requirements. That's because the ng-hide and ng-show directives only affect the display, not the fetching process. Let's try ng-if instead. So if we change ng-show here to ng-if, and that's the only thing we're changing, ng-show to ng-if, and we're going to run again. Again, we bring up the Developer Tools and refresh the page. And this time, you'll see that it is not bringing down the images. When I click Show Image, only then are the images downloaded and displayed. Excellent. We have all the functionality we want. Are we done? Well, the UI doesn't look all that great. We don't have funding for a designer, but let's see what we can do with Bootstrap.

Styling the View With Bootstrap

Our current user interface looks like this, but we would it to look like this. That's where Bootstrap can help. Bootstrap is a framework for prettifying Web pages. It helps you develop responsive applications on the web. Responsive in this context means that it is responsive to changes in layout. In other words, it scales to multiple form factors--phones, tablets, laptops, and desktops. It also handles browser differences so you don't have to. It was originally developed by Twitter. You can download it on the web address shown here. There is also a larger third-party community where you can find free or inexpensive themes and other designs. One of the key features of Bootstrap is its grid system. Instead of using tables or divs to layout the user interface elements for your application, you can use Bootstrap's grid system. The grid system is divided into 12 equal columns. You can place an element into any number of columns using the syntax shown here. For example, if your first display element should take up four columns, you can specify col-md-4. We'll use this grid system in some parts of the application. Here is the Bootstrap page. In addition to the grid system, Bootstrap provides styling for many other things, such as tables, forms, buttons. There are also components, the style Glyphicons, panels, and as you can see, much more. Let's add some styling. Here we are back in WebStorm. I have already downloaded Bootstrap and added the Bootstrap.css file to the project under the CSS directory. There it is there. I also added the associated fonts under the fonts directory. There they are. We can then add a link to the CSS file in the index.html. And while we are here, let's add an app.css file as well. So we'll say New, File, app.css. This file is currently empty but will contain any additional styles we may want for this application. So we're just going to close it for now, but we'll add a reference under our style sheets. That way, it's all set up and ready for us when we do want to add some styles to that file. Now we're ready to style our page. Let's put the entire table into a panel with a nice header. Much of the Bootstrap styling is done using div tags. So we can add a div tag with an appropriate style class to find by Bootstrap. So let's start with the panel. The first div that we added defines a panel. The second name in the class definition defines the color. Panel-primary is a blue panel. The second div tag adds a panel-heading. We added a style attribute to up the font in the heading. The third div defines the panel-body. The table will then appear in the panel body. Let's see how that looks so far. Ooh, much better. But the white background looks a little harsh. Let's change the background to a light blue. We'll use basic CSS for that adding a body element to the app.css file. So we'll open app.css and we'll add a body tag. This is a light blue color. Now let's go back to index.html and style the table. We'll just add the Bootstrap table class. So we can just say class=table. Let's take a look. Ooh, wow, that made a big difference. Look at that. Let's style our button now. So for our button, we're going to say class=, we're going to style it as a button. And again, we're going to use primary to make it a blue style button. Run one more time. And now our button's blue. That's pretty nice looking. Wow, only a few minutes of work, and our application looks much better. We now have an Angular application with a Product List page that we can be proud of.

Summary

We've covered a lot of territory in this module. Let's review the objectives. We turned the business requirements into a view, model, and controller for the Product List feature of the application. We organized the directory structure for the application using a feature-based structure. We built the view to display the set of products and made that view smart enough to only download the product images on user request. We defined the main module for the Angular application. We used an immediately-invoked function expression, or IIFE, to ensure that we were not defining global variables or function. We built the controller and registered it with the module. In the controller, we defined the model by creating an array of products that are then displayed on the products. We also implemented the method to show or hide the images. Finally, we styled the application with Bootstrap. The result was the first view, model, and controller for our Product Management application. The view contains the HTML for the pain and styling using Bootstrap. We used the Angular ng-app directive to identify the main module for this application. We used the Angular ng-controller directive to identify the controller associated with this view. We used the controller as syntax to access the controller without injecting a $scope object. The controller contains the code for the page. The controller defines the model. In this case, the model contained an array of products and to show image flag that tracked whether the images are shown. The controller implements methods for the actions on the page, in this case the toggle image method toggled the display of the images. At this point, we have covered several of the tasks we identified for building a line of business application. We looked at the business requirements for our first page and turned those requirements into application features. We laid out the view for that first page and used Bootstrap to give it some style. But with only one page, we have not yet defined any navigation. But there is one major problem with our application as it stands right now. We hard-coded in the data. In a real business application, we need to get that data from a data store somewhere. Let's do that next.

Accessing Data

Introduction

In a corporate environment, the data required for a client-side web application is often provided by a back-end web service. So an Angular application must communicate with that web service to retrieve and update data. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module demonstrates how to retrieve data from a back-end web service. And since that web service may not be available when building the Angular application, this module also looks at how to intercept the outgoing web service calls and mock the results. The primary objectives of this module are to understand how a client-side web application retrieves data, evaluate the existing Angular services for calling a web service, build a common module and custom Angular service as a reusable data access component, modify the controller to retrieve data instead of hard-coding it, fake it and retrieve data without the web service. Let's get started.

Understanding How Data Is Retrieved

Angular applications are entirely client side. That means that Angular applications run within the context of whatever browser that the user has open. And it means that Angular applications don't have direct access to corporate resources such as a database. Instead, an Angular application needs to obtain everything it needs from a remote web server. It communicates with that web server using a meaningful set of URLs. Here is a simplified view of how this process works. As you know, a real application is ultimately deployed to a remote web server. It may be a server hosted by a third party or a self-hosted by the company. When a user accesses the application through an appropriate URL, the web server responds by downloading the index.html file and any other resources that the index.html file defines, such as CSS and JavaScript files. We saw this process earlier in this course when we used the Chrome Developer Tools to watch the files get retrieved from the server. When the Angular application needs data, the application can call out using a URL to access a web service on the remote server. A web service is an application on the back-end server that exposes an application programming interface, or API, for access by client-side web applications. That web service can be built using Node.js, Java, PHP, ASP.NET Web API, or any other server-side technology. Any web service that can return JSON data will work with Angular. Notice the syntax of this URL. It is requesting the product with a unique ID of 5. It can request all products by dropping the ID portion of the URL. In either case, the back-end service processes that URL and returns the appropriate data. That data is returned in JavaScript object notation or JSON format just like what we have hard-coded into the application. The JavaScript code in the Angular application can then prepare that data for display in the view. Okay, so the question now is how? How does Angular send out the URL requesting the data from the web service? And how does it process the response?

Evaluating the Built-In Services

Angular provides an extensive set of built-in services, all of which are prefixed with the dollar sign ($). Angular provides two such built-in services for calling a back-end web service--$http and $resource. Angular also provides a service for intercepting the outgoing web service requests. It's $httpBackend. This service is used to mock communication with the web service. We'll cover the top two next and the last one later in this module. The built-in Angular $http service is a core Angular service that facilitates direct communication with a remote web server. It provides convenience methods for performing HTTP requests, such as get, post, put, and so on. This example uses the get and defines the URL. This call will request the list of all 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. You can use the then method to define a callback function. When the request is complete, the callback function is executed, and the response object is passed into that function. The data is obtained from the response object and assigned as part of the model using code similar to this. So we could update our controller to include this code in place of the hard-coded array of products, but there is an easier way. The built-in Angular $resource service is a factory that creates a resource object you can use to interact with a RESTful back-end web service. REST is short for Representational State Transfer. It is a method of intelligently serving data from a back-end web service based on a URL. The $resource service only works with RESTful services. The $resource service abstracts away much of the complexity of communication with the web server, and it is the technique that we will use in this sample application. The function shown here uses the built-in $resource service. The parameter to $resource describes the URLs that communicate with the RESTful back-end web service. The part of the URL prefixed with a colon defines an optional parameter. The $resource provides a built-in set of methods that perform a default set of actions, including a get to retrieve a specific product, query to retrieve a set of products, and save to save modifications to a product. It's hard to see from this slide how much easier it is to use $resource. So let's add it to our sample application.

Building a Reusable Data Access Component

Before we get started building a new component, let's do a quick review of where we are with the sample application. Currently, we have a main module that we call Product Management. We defined that module in the app.js file using this code. The first parameter is the name of the module. The second parameter is the list of dependencies. Currently, there are no dependencies. We have defined one component and registered it with the main module. That component is the ProductListCtrl, and we registered it as a component of the module using this code. We obtained a reference to the main module, then called the controller method to register the controller. So this is where we are now. When we last saw the ProductListCtrl, it contained a hard-coded set of data for the application. While that may be helpful early on, it's not what we really want here. For most line of business applications, the data should instead be obtained from a back-end web service. Our task is to replace that hard-coded data with calls to a web service using $resource. Because several different parts of the application will ultimately need to access data, we want to put the code that retrieves the data into a reusable component. In Angular, the recommended way to define a reusable component is to create a second module, and in that module, create a custom service containing the reusable code. So let's talk through how to do that and then build it. We will define the reusable data access component by first building a separate module that we will call common.services. The code to create this module is similar to the code we used to create the main module. The first parameter is the name of this module common.services. The second parameter is the list of dependencies. Code in this new module will use $resource to interact with the web server. The $resource is defined in the Angular ngResource module. The ngResource module is an optional Angular module that adds support for interacting with RESTful web service. As an optional module, it is not included with Angular by default. During the upcoming demonstration, we'll look at how to download it from the Angular website. Since the module is not provided with Angular, it has to be defined as a dependency as shown here. Once the module is created, we are ready to build the custom service. We can build the service required to use $resource and call the web server. We'll call it Product Resource Service, and we'll need to register that service with the common.services module with code that looks like this. We obtain a reference to the common.services module, then call the factory method of that module to register the Product Resource Service. We'll talk more about the different types of custom Angular services later in this course. The last step here is to add the code to the Product Resource Service that actually uses $resource. After completing these steps, we will have a reusable data access component. Let's do it. Here we are back in WebStorm. First, we need to locate the JavaScript library that is required to use $resource. The Angular Resource.js file is available on the Angular website. Click on the Download button, select Browse additional modules under Extras, locate the Angular-resource.js file, then click to download it. Or you can pickup the minified version if you would prefer. I've already downloaded this file and added it under the .js folder. So I have it here. And before we forget, let's add that file to index.html with a script tag. Be sure to add it after the reference to Angular itself. Next, we will create the new module. It is a common service, so we'll put it under the common services folder. Right-click and select New, JavaScript file. We'll name it common.services. Now we have another new file that we need to define in index.html. Let's do that now. So we're going to go back to index.html, and we're going to add our service. Now we'll go back to the file. And in this new file, we'll create the module using code similar to that we used when we created the main module. Does this look familiar? It is very similar to how we created the Product Management module. We start with an IIFE, we add use strict, then we use the module method of Angular to define the module. The first parameter is the name of the module. We called it common.services. The second parameter is an array that defines the dependencies. In this case, we need a dependency on ngResource, which is the module that contains the $resource service. Now we are ready to define the factory service that communicates with the web server. We could put it into this same file, but we want to keep all of our services separately encapsulated, so we'll put it into a new file. We do want to keep it in the same folder, so we'll right-click on common, services, New, JavaScript file, and this one we'll call productResource.js. So yet another file to add to index.html. We'll put it there. Now we'll go back to the file. In this new file, we'll again start with an IIFE, an immediately-invoked function expression, and define use strict. Then we'll look up the common.services module, and we'll register the new factory with that module using the factory method of the module. The first parameter is the name of the factory service. We're calling it productResources. The second parameter is an array. The first elements in the array are the string names of the parameters that are going to be passed to the function. In this case, we want $resource. Recall that we defined the string names here to ensure that the application will still work if this JavaScript file is minified, so this array is sometimes referred to as a Min-Safe array. The last element of the array is then the factory service function. We're just going to set it to a reference to that function, and we're going to call it productResource. Next, we define the productResource function itself. We want to inject the $resource service in as a parameter to this function. The function then simply returns the $resource object giving it the URL for the products. Simply by returning the $resource object with the appropriate URL defined, we have set up communication with the web server. Now we have two modules--our main module and this common.services module. How do we tell our main module about this new common.services module? If you guessed dependency, you are right. So let's go to app.js, and in here now we need to define a dependency. And that dependency is on our common.services module. The reusable data access component is now complete and accessible anywhere from within our application. Now we just need to use it. We want to delete the hard-coded data from the ProductListCtrl and replace it with a call to this new productResource service.

Modifying the Controller to Retrieve Data

Our new productResource service is ready to go. It returns a resource object that we can use to call the web server to retrieve the data. The resource object provides methods that allow interaction with a back-end web service. By default, the object provides five methods. Get sends a get request to the URL and expects a JSON object in the response. Query also sends a get request to the URL, but it expects a JSON array in the response. Save sends a post request to the URL. Delete sends a Delete request to the URL. And Remove is synonymous with Delete. It is often used in place of Delete because Delete is a reserved word in JavaScript. Notice that there is no put request defined here. Save sends a post request so you can either use the post or you can extend $resource to include a put request. When we add saving functionality to our sample application later in this course, we'll just use the built-in post functionality. Currently, our application needs a list of products, so we will use the query method. The query method sends a get request to the URL we defined in the $resource function. The parameter is the callback function that is called upon receiving a successful HTTP response. The response is the JSON data returned from the query. So the resulting data is simply assigned to products as part of the model. Let's modify the ProductListCtrl to use the productResource service. Here we are back in WebStorm. I have the ProductListCtrl JavaScript file open. The first thing we want to do is get rid of the hard-coded JSON array that defines the list of products. If you are working along with this example and typed in some products, don't delete this array, just comment it out. We'll use it again later. Next, we need to ask Angular to pass a reference to the productResource service to this specific controller function. We do that by adding the productResource as a parameter, so we're going to pass it in here. Since we now have a parameter here, we need to change the controller method to pass in Min-Safe array and provide the string name of the parameter as the first element of that array. So we're going to change this to an array, and we're going to add productResource in quotes as the first element in that array. Next, we can add the call to the query method of the productResource. So here I can say productResource.query, and I can pass it a function. In the function, all I need to do is to assign the returned data to my model, and 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, and return a JSON array containing the list of products. Easy enough. Let's run it. We don't get any products. Let's look at the Developer Tools, and it responded with a 404 (Not Found). It can't find the URL. Well, of course it can't find it. We have not created a RESTful web service to process that URL and return the data. So at this point, we can no longer run the application. So we could wait for the back-end team to build the web service, or we could build that web service ourselves, or keep going without the web service and continue to write code without running it. Yeah, right. But Angular has a better solution. Angular provides a way to fake the web service. Let's give that a try.

Faking the Web Service

Angular provides a $httpBackend service to intercept calls to a web service and fake the response. This fake HTTP back-end implementation is suitable for any application that uses the $http or $resource service. It mocks the calls to the web service, returning predefined static data to the application. There are two implementations of $httpBackend. One is an ngMock and is provided for use during unit testing. The other implementation is an ngMockE2E. That implementation is for end-to-end testing. It can also be used for back-end-less development, which is what we want to do. We want to develop our Angular application without having a back-end web service in place. So we want the implementation of $httpBackend that is implemented in ngMockE2E. Implementing the $httpBackend service requires the following steps. First, download the JavaScript file containing the ngMockE2E module, and of course insert the script tag into the index.html file. Create a new module that depends on ngMockE2E. In that module, set up static data and define the fake responses to the web server calls using that data. Add this new module as a dependency in the main module. Anytime you want to turn off the mocking, simply remove the dependency. Let's give this a try. Here we are back in WebStorm. First, we need to locate the JavaScript library that contains the ngMockE2E module. Luckily, it's in the same place that we found ngResource. Click on Download, pick Browse additional modules, and locate Angular-mocks. I have already downloaded this file, and you can see that it's listed here in the .js folder. Before I forget, let's add that file to index.html with a script tag. Just be sure to add it after the reference to Angular itself. Next, we'll create the new module. We need this mocking throughout the application, so we'll put the new module under the common services folder, New, JavaScript file, and we're going to call it productResourceMock.js. Now we have another new file that we need to define in index.html. Let's do that now. So it's going to be another service, so we can put it here under our services. And we'll go back to our mock. And this new file will create the module using code similar to that used when we created the main module and the common.services module. So we have our IIFE, we have use strict, and we declare a variable where we're holding our reference to our new module. We're defining the module with the module method. The first parameter is the name of the module productResourceMock, and the second parameter is the dependencies. This module will depend upon the ngMockE2E module. By now, I'm sure this syntax looks familiar. We're going to use app.run to perform the initialization. App.run takes the function passed in as a parameter and executes it when the module is loaded. Inside this function, we need to do two things. First, we need to define the default set of data. This is the data that will be our mocking data. And we need to define the fake responses to the web service calls. First, let's define our set of products. If you are coding along and kept the list of products created in the ProductListCtrl, you can copy those here. Note that I added several more than what we had in our sample code. That's so that we have a little bit more to work with here. Next, we need to define the fake responses to the web server calls. We'll start with the URL that we're expecting to intercept. We're expecting to intercept /api/products. Then we'll define what should happen when a get request is sent to this URL. We want to return the full list of products. We'll use the whenGET method of the $httpBackend. This method says where there is a get request on the defined URL, respond with the entire array of products. Now because we put this code in another module, in order to use it in the rest of our application, we need to define it as a dependency in the main module. So we're going to go to app.js, and we're going to add another dependency here. And that dependency is on productResourceMock. Let's try the application again. We'll go to index.html and run the file. There we have data. But what really happened here? Let's look at that code again. Let's start with the controller. The controller is calling the query method of the productResource service. That service is sending a RESTful get request to a web service using the URL that we defined there. In that case, it's /api/products. The httpBackend service is intercepting that request, so when it receives that URL request, it's responding with that entire set of products, and it's taking the data it receives and assigning it to the products part of our model. Since products is already bound as it was before when we had it hard-coded, the data immediately appears. So since we again have the products hard-coded in here, how is this different from just hard-coding the products into the controller? The biggest difference is that the code is now set up to communicate with a web service. Just remove that dependency, so we could go to app.js and simply remove this dependency, and the code will call the web service directly. So it's all prepped and ready to go as soon as the web service is in place.

Faking the Web Service - Part 2

Before we move on, let's add a little more code here to our mocking module. Some parts of the application, such as the ProductDetailView and ProductEditView we will be adding later in this course, retrieve only one product. Recall that our mocking service is only processing /api/products. When retrieving one product, it will need to process calls in the form api/products/n where n is a number. We need to add another get. We could do a set of simple whenGET statements, but that is not very clean, nor is it very extensible. Luckily, the whenGET takes a regular expression, so we can define a regular expression that handles a numeric parameter. So let's first define a regular expression. So you'll notice our regular expression is the productURL plus a slash plus any number of digits. So our whenGET then can use that regex. But we don't want it to respond with the entire set of products. We want it to respond with the one product as defined in the last part of the URL. That requires a bit of JavaScript. I'll paste it in, and then we can talk through it. The Respond method can take a function. We can write any code we want into that function. In this case, the function locates and returns the desired product. The code begins by splitting the URL on the slashes. This puts each part of the URL into an array. The ID will then be in the last array element. The code then loops through the array of products looking for the specific product with the matching ID. If one is found, it returns a response code of 200, which is a success value, and the product. Later in this course, we will also implement save functionality, so we'll want to process the post calls to fake out adding and editing products in the list. Again, I'll paste the code, and then we can talk through it. The $httpBackend also has a whenPOST method. It works similar to the whenGET method. We can pass in a function that is executed when the post occurs. In this case, we want to access the passed in data, so we use Angular's fromJson function to deserialize the passed in JSON string and return the result in the product variable. We can then access the fields in the JSON structure. We can check the product ID to determine if this is a new or existing product. New products won't have been assigned an ID, so we assign the product ID here and push the product onto the array of products. For updated products, we can locate the product in the list and update this. This code then returns a response code of 200, which is a success value, and the updated product. Now when we add a save feature later in this course, the mocking will save changes to the In memory array of products. Cool! Before we finish, let's add one more thing. Using $httpBackend intercepts all URL requests. In the upcoming modules, we will be using the URL to request and download additional HTML files. This mocking framework will intercept all of those calls as well. We don't want to provide mocks for those UI elements. Rather, we want those particular requests to be passed through. All of the HTML files that we create will be in the app directory up here, so we can use that as a generalized URL. Every request to an HTML file or any file in the app folder will then be passed through and ignored by the mocking that we're doing here. We'll look at this further in an upcoming module. But for now, our web service mocking method is complete. Let's run one more time. And there's our list of products.

Summary

This module was all about the data. We described how a client-side Angular application retrieves data from a web server using a URL. We looked at two built-in Angular services--$http and $resource. Both of these services provide communication with a web server from an Angular application. We selected to use $resource as it provides a higher level of abstraction for accessing data. We then built a common.services module and defined a productResource custom service within that module. The productResource service is our reusable data access component. We then modified the controller to call the query method of the productResource service to retrieve the data, but since we did not build a back-end web server to process the request, we were not getting any data. We instead built a productResourceMock module using $httpBackend to intercept the request to the web server and provide some mock data for the application. We now have data even without the web service. When we have a real back-end web service, we can simply remove the productResourceMock dependency, and the application will instead use the back-end web service. So our application is web service ready. Looking back at the set of tasks we identified earlier for building a line of business application, we now have code that retrieves and saves data. Though we still have hard-coded mock data, the application is now set up to access real data from a back-end web service. When the back-end web service is in place, we simply remove the mocking dependency, and the application will the back-end web service. Now that we have data, let's add some additional views of that data and the navigation between those views.

Routing to Multiple Views - Part 1

Introduction

Business applications often display a view for each of the many business operations, including selection, review, and edit. And as soon as you have more than one view, you need a way to navigate between them. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module is Part 1 of a two-part segment that demonstrates how to define, configure, and navigate Angular route states to display multiple views. Let's start with a definition. Routing is a technique for navigating between the views or user interface pages of a web application. Each route represents a specific view. Activating a route navigates to that view. One way to define a route is with a URL. The portion of the URL prefixed with a hash symbol defines a fragment identifier. Angular routing takes advantage of the fact that the fragment identifier is processed by the client and not submitted to the server. So you can assign a unique fragment identifier to each view and then navigate to that fragment to display the view without submitting to the web server. The fragment identifier then uniquely identifies a routing state, not a physical resource. For example, say you have a menu, and from that menu, you want to navigate to a welcome page or to the product list view. You can identify a default route to the welcome page using just a slash. You can identify the route to the product list view using /products, and so on for each view. Angular provides two different routing frameworks, each with their own approach to routing. There is the built-in ngRoute framework and the more full-featured UI router framework. NgRoute is distributed as an additional module that can be downloaded directly from the Angular website. NgRoute is in the file Angular-route.js. NgRoute navigation is based on URL routes using fragment identifiers. Each route has a URL fragment identifier, a view template that is displayed when navigating to that URL fragment identifier, and an optional controller. NgRoute provides an easy-to-use framework when handling simple URL-based navigation between pages of an Angular application. The other Angular routing framework is UI router. It is available from a third-party Angular UI library. It can be downloaded from the URL shown here. It is in the file Angular-ui-router.js. UI router navigation is based on application state. Each state has an optional URL fragment identifier, a view template that is displayed when the defined state is activated, an optional controller, an optional additional attached behavior, such as preloading the data for the state as we'll see in the next module. These routes states can identify named, nested, and parallel views providing powerful features to manage more complex navigation scenarios. We'll see one of these more complex scenarios in the next module. Because of these additional features, this course and the sample application will use UI router. Let's look at what we will cover in this module. The primary module objectives are to define a site map, layout the layout template containing the main page of the application, configure the routing, build a menu that activates the appropriate route state based on the user-selected menu option, and build a view for the default route state. The discussion of routing continues through the next module where the objectives are to use resolve to preload the data for a routing state and define nested routing states to support a multi-view wizard. Nested routing can provide the navigation for a tab-based set of edit pages. The focus throughout these two modules is on the routing. Let's get started.

Defining the Site Map

A site map is a list of the pages or views of a web application. It is normally organized in a hierarchy and shows the navigational paths between the views. We can use the result of our hypothetical requirements analysis to define the views needed for our sample product management application. We'll want a default view of some type. In this example, we'll build a welcome page. That page could include a login feature and display notifications. In the sample application, we'll just display an image. As per the requirements, we need a product list view. The product list view should allow the user to select a product and immediately navigate to the product's detail view. And the product list view should allow the user to select a product and navigate to the product edit view. once we have multiple views, we'll want a menu so the user can navigate to the desired feature. The menu will have an option to display the welcome page. It will have a product list option that displays the product list view, and we'll add an Add Product option that displays the product entry view for entry of a new product. So here is the site map for our Product Management application. With the site map in place, we can define a state for each route. Let's give the route to the default welcome page a state name of Home and the route to the product list a state name of productList, the route to the product detail a state name of productDetail, and the route to the product edit a state name of productEdit. You can call these states anything you want as long as they are unique. But for maintenance, it's best to call them something obvious, as we did here. With the states defined, let's look at how to go about building what we have specified here in the site map.

Laying Out the Layout Template

We'll start the process of building the application defined in the site map by building the layout template. The layout template is the main page of the application, and if you're familiar with the concept of SPA or single page applications, the layout template is the single page of the single page application. All other application views appear within this main page. Let's see how this works with a demonstration. We wanted to find the HTML for the header and menu once and display each page in the area below it. When the user clicks on Acme Product Management, the welcome page should appear in the area below the menu. When the user clicks on Product List, the product list should appear. When the user clicks on Add Product, the product edit view should appear with the edit fields set up for entry of a new product. We achieve these results by first defining the layout template. The layout template includes the elements that are common for all of the views of the application. It can contain a header and a menu as shown here. It can include toolbars, a footer, or anything else that you want displayed on every page. Each view is then defined as a separate view template, each in its own HTML file. And the appropriate view template is inserted into the layout template based on the current application state. For example, when the Product List state is activated, the product list view is inserted into the layout template. For our sample application, the index.html file will define the layout template. It will contain all of the HTML elements that appear on all pages of the application. So it will contain the header and menu as shown here. Currently, the HTML for the product list view is in the index.html file. We need to remove the HTML specific to the product list view. We can extract that out into its own separate HTML file. Let's do that now. Here we are back in WebStorm. Let's start by creating a new HTML file for the product list view template. This view should be in the same location as its controller to make it easier to find and edit related files. So let's navigate to app, products, right-click, New, HTML file, and we'll name the file productListView. We don't need any of the default HTML here. Since these view templates will be inserted into the main layout template, they don't need the HTML head or body tags. So let's just delete them. Then let's open the index.html file and cut out the HTML elements for the product list view. So we'll take everything here, so we can just collapse that, then we can cut it and paste it, and there it is. This is now our product list view template. One more thing. The routes will now specify the controllers associated with each view. So let's remove it here from the template. What do you think will happen when we try to run this? The code as we have it just displays the index.html file, which contains no display elements. Let's configure the routing and get the application working again.

Setting Up the Routing

Setting up the routing using UI router requires the following steps. First, set up UI router. You'll need to download the UI router library, insert the script tag into the index.html file, and set the UI router module as a dependency in the main module. These steps should seem somewhat familiar since we've used similar steps when setting up to retrieve data in the last module. Once UI router is set up, then we need to set up the layout view. Identify where the view should appear within the index.html file, add the ui-view directive to that appropriate location in the layout view. Use data-ui-view instead if your IDE does not recognize ui-view. We are then ready to configure the routes. We'll need to identify the desired route states, which we've already done, and then set up each route state in the code. Let's do the first two steps now, then dive deeper into the route configuration step. Here we are back in WebStorm. The first step for setting up the routing is to set up UI router. We need to locate the JavaScript library that is required to use UI router. UI router is available here. Scrolling down, you can see the Getting Started section of the documentation. There are several ways to obtain UI router. You can click the Download the Release link to download the JavaScript file, or you can pick up the minified version if you prefer. I have already downloaded this file and added it under the .js folder. And before we forget, let's add that file to index.html with a script tag. Be sure to add it after the reference to Angular itself. Next, we'll add UI router as a dependency. So we'll open app.js and add UI router here as a dependency. So now we've set up UI router, and it's ready to be used. Now that UI router is in place, the second step for setting up routing is to set up the layout template. In the sample application, we are defining the layout template in the index.html file. We need to identify where in the layout template we want the view templates to appear. We want them in the body tag. We'll use a div tag with the ui-view directive to identify the location of the view templates. So we'll put them here. Depending on the IDE you are using, directives prefixed with ng- may be recognized as valid HTML attributes. But the UI- may not be recognized. If you're IDE marks ui-view as a syntax error, you can use data- as a prefix. The data- is the standard prefix for custom HTML attributes and is recognized as valid by most IDEs. WebStorm does not mark the ui-view as a syntax error, so we'll leave the data- off. Lastly, let's put the views in a Bootstrap container to improve the look. So that's it. That's all we need to do. We just specify a div tag with the ui-view, and that's where the routing will inject all of the view templates. If we ran the application at this point, it still would not display anything. That's because we have not yet told Angular which view template to put into this container. Let's do that next.

Configuring the Routing

Here again is the site map with our defined route states. The states correspond to a specific place in the application. Each state defines the view, the controller associated with that view, and other state properties. When the defined state is activated, the associated view is displayed within the tag marked with the ui-view directive, and the associated controller is executed. The routes are configured in the application using the UI router's $stateProvider service. The $stateProvider service provides a state method for defining the states. Each state includes a name and an object. Use the object properties to define the parameter for the state. The URL property defines the URL fragment identifier associated with this state. By associating a URL with the state, you get deep linking. Deep linking reflects where the user is in the application. This ensures that the browser's forward and back buttons behave as expected, and the user can bookmark or email links to locations within the application. The template URL property defines the path to the HTML template containing the HTML elements for the view. This template is then displayed in the HTML element that contains the ui-view directive. The controller property is optional. It defines the string name of the controller associated with the defined view. Since the controller is registered with the module, the $stateProvider can locate the controller by its name. Since we are using the controller as syntax for managing scope in the application, as discussed earlier in this course, we are adding the as and an alias here. There are additional object properties available for defining more complex states. Check out the UI router documentation for the full list of properties. Now that we know what a state looks like, let's configure the state for our product list view. Here we are back in WebStorm. We will add the route state configuration for the entire application in the app.js file. Let's start by adding the route state for just the product list view. To set up the states, we'll use the Angular application's main module. We're referencing it here with the app variable, and we'll call the config method. The config method takes one parameter, which is the function that defines the configuration code. We'll want to pass the $stateProvider service into that function. And since we want to make this code safe for minification, we'll actually pass an array into this configuration method. The first element of the array is the string name of each parameter we're going to pass into the function. We're only going to pass $stateProvider. The second element of the array is then the function itself. And we're going to pass to the function the $stateProvider. Angular will then inject this $stateProvider service into this function for us. Now we are ready to use the $stateProvider and call the state method to set up our first state. So I'm just going to paste in the code from the slide. This code defines the state named productList. It sets the URL fragment identifier for this state to /products. It specifies the URL defining the location of the HTML file that contains the view template. When this state is activated, this template is fetched from the web server if it has not already been downloaded. The HTML from the defined template is then inserted into the layout template at the location defined with the ui-view directive. This code also specifies the associated controller. When this state is activated, the defined controller is constructed. Will our application run now? Let's give it a try? Nope. We don't have any code that activates that state. But we associated a URL of /products for this state, so if we add #/products to the URL, bang, the routing found our view, and the mocking framework picked up the data requested providing our mocked data. The application is working again. When we ran the application, we had no initial page displayed. That's because our routing does not define a default route state. A default route state provides a fallback state for use when no other state is activated. We can define a default state using the $urlRouterProvider service. The $urlRouterProvider service watches $location for changes to the URL. When $location changes, it runs through the list of state rules one by one until a match is found. Then it activates that state. It is used behind the scenes whenever you specify a URL in a state configuration. For our purposes, $urlRouterProvider provides an Otherwise method for defining a default URL. In this code, the default URL fragment identifier is /products, which will activate the product list state. Let's set up a default route now. We are back in WebStorm with the app.js file open. We need Angular to also inject the $urlRouterProvider service into this configuration method so we can add it as a function parameter. We'll just put it here. We'll also need to add the parameter to the Min-Safe array. Then we can use the Otherwise method of $urlRouterProvider to identify the default URL. We'll put that as the first line of the function. This line of code says that if an activated state has no entry here in this function, or if there is no active state, then display the state associated with the /products URL fragment identifier. Looking here, that's the productList state. So let's run the application again. Now it automatically navigates to #/products and displays our product list. Cool! We're rolling with some routes here. Now that the routing is configured, let's look at the three basic ways to activate a route and navigate to a view. One option is to navigate to the state's URL. That's how we displayed the product list view. We defined a default URL and UI router navigated to that URL. This activated the state and displayed the associated view. Another option is to activate a state with code using $state.go. $state.go activates the defined state. We'll use $state.go later in this course. Another option is to click a link that contains a ui-sref directive. The ui-sref specifies a state. Clicking the link activates that state. We'll see how to use ui-sref when we create the menu for the application, which is coming up next. Notice that the first technique uses the state URL fragment identifier. The other two techniques use the state name.

Building a Menu With Routing

Most line of business applications have some type of menu or toolbar for the user to access the application features. For the sample application, we'll create a menu. Initially, it will contain three options. Clicking on the application title will activate the home state and display a welcome page. The Product List option will active the product list state and display our product list. And the Add Product option will active the product edit state and display a product edit view for entry of a new product. We have only added the code for one state so far. Before we create the menu, let's add the home and product edit states. Here we are back in WebStorm with the app.js file open. Let's add the route state for the home page first. States can be chained together so we can just add the new state here. The state name is Home. It will have a simple URL of just a slash and a template URL that references an HTML file containing the UI elements for the welcome page. We have not yet created that HTML file. We'll create it later in this module. Now it seems that this home page should really be the default page of the application, so let's change the Otherwise to navigate to just the slash. Now if there isn't a valid state or if there is no state that's been activated, it'll display the home page. Now for the edit page, we can just attach that here. As we defined in the site map, the state name is productEdit. The URL syntax is just a little different here. This URL has a parameter identified with a colon prefix. The parameter is required because we want to display the edit page for one specific product, and the parameter will identify which product. On an add, the product ID will be 0, but when using the state to edit an existing product, the productID parameter will be the ID of the product we wish to edit. The template URL references an HTML file containing the UI elements for the edit view. We have not yet created that HTML file. We'll create it later in this course. The controller property references the controller associated with this view. Because we are using the controller as syntax, we specify the as and the alias here. Now our application is configured with three states, one for each of the three menu options. We are now ready to create the menu. We want the menu to appear as part of every page of the application, so we will define it in the layout template. The layout template is in the index.html file. We'll add the menu within the body tag. Recall that the div tag here with the ui-view directive is where all of the UI templates will appear. So we want to put the menu above that, and we'll use Bootstrap to define the styles and make the menu look nice. Because there is quite a bit of code involved, I'm just going to paste it in here, and then we'll talk through it. This code uses the HTML5 nav element and the Bootstrap navbar classes. This first anchor tag contains the application name. Clicking on this link should display the welcome page. The div class containing the button is for use when the menu is displayed in a smaller form factor. You'll see what I mean in a moment when we run the application. The bottom div class displays the menu option as a list of anchor tags. Notice the I tag here. This puts a plus icon in the menu option name. Let's give this a try. So there's our menu. Look what happens when the browser is resized to a smaller form factor. Bootstrap changes the horizontal list of menu options into a dropdown list suitable for a phone or smaller device. The problem is that the dropdown list doesn't work. That's because it needs the code that is in the Bootstrap.js file. We have not yet included that file in the application. Plus, the Bootstrap.js file requires jQuery, so we need that too. The Bootstrap.js file should have been downloaded when you downloaded Bootstrap. You can download jQuery from here. I have already downloaded both of the required files. They're here in the .js folder. We can then add a script tag for each file in index.html. So we'll go up to the top here. We'll add Bootstrap.js, and we'll add jQuery.js. Note that the jQuery script tag should be before the Angular tag. Now let's try running again. Ah, the dropdown list works. Notice, however, that the menu options don't work. If I resize it back to a bigger size, menu options still don't work. That's because we have not yet defined the code to activate the route states. Let's do that next. Now that the menu is in place and seems to be displaying correctly, let's activate the states. Looking at the menu options, they are all defined within anchor tags. As we discussed earlier in the slides, you can activate a state using the ui-sref directive in an anchor tag. So we'll change each menu option to specify the ui-sref directive. Let's start with the product name. In here, let's activate the state, ui-sref=, and when they click on Acme Product Management, we want to display the welcome page. Its state was called Home. We want to put the state name here, not the URL. Then we can update the productlist option. So we're going to add ui-sref=, and it's state was called productList. Then finally, the Add Product. We can add the ui-sref directive. It should activate the productEdit state and display the product edit view. If you recall, the productEdit state requires a parameter defining which product to edit. We can specify the parameter by passing an object into this state. So here is where we define the object, and in that object, we define the state. Now our state variable was productID, and since we're creating a new one, we want that ID always to be 0 for new products. Where did that productID come from? Well, if we go back to app.js, it was identified right here, so it has to match that name. So we're calling the productEdit state and we're providing the productID. Let's give it a try. Notice that when the application starts, it navigates to slash, which is the URL fragment identifier for the default home state. But since we have not yet created the UI for that page, nothing is displayed underneath the menu. When we click on the product list, the application navigates to /products, which is the URL fragment identifier defined for the product list state. The menu works. Now let's build that default view.

Building a Default View

Often, business applications start with some type of welcome or home page that is used as the default view of the application. That page can include a login feature, display notifications, or provide a dashboard with open tasks and key performance indicators. For now, the welcome page for the sample application just displays the logo of our fictitious company. You can add anything you want to this page. Let's create it now. We are back in WebStorm. The route state definition already has the name and location of the HTML file for the welcome page, so we can navigate to the app directory, right-click, New, HTML file, and we'll call it WelcomeView. We don't need any of that default HTML, so we'll just delete it. These few templates will be inserted into the main layout template, so they don't need the HTML head or body tags. And we don't need to reference this file from index.html. This file won't be downloaded when the application starts. Rather, it will be loaded when its associated state is activated. Then let's define the contents of this view. I'm just going to paste it in. For consistency, this view is defined using a Bootstrap panel class just like the product list view. The panel header displays the application name Acme Product Management. The content of the page is simply an image. You can display an image or whatever other elements you wish to display here. That should be all we need to do since the routing state is already set up. I just need to copy that image to the images directory. Now it's there, and we should be able to run the application. So let's run the index.html file. There's our welcome page. Click Product List to display the product list. Click Acme Product Management to go back to the welcome page. Cool. We now have a menu that works, and when we click on the menu options, they activate the appropriate states.

Summary

This module was all about routing using route states. We began by defining a site map for the sample application and identified a state for each view in the site map. Then we laid out the layout template. The layout template is the main page of the application. For our sample application, it's defined in the index.html file. It only contains the HTML elements that are displayed on each page of the application, such as a header, footer, or in our case a menu. So we extracted the product list view elements from index.html into a separate HTML file. We then configured the routing by downloading and referencing the UI router library. We used the ui-view directive in the layout template to specify where the views should appear and accessed the $stateProvider service of the UI router library to specify the routing states. We built a menu by adding appropriate elements to the layout template and used the ui-sref directive to activate the appropriate route for each menu option. Lastly, we built a default home page view for the sample application. This is the page that appears when the application is first displayed. The next module continues our discussion of routes and demonstrates two advanced features of UI router using resolve to preload data for a routing state and defining nested routing states to support a multi-view wizard.

Routing to Multiple Views - Part 2

Introduction

In the last module, we covered the basics of routing using UI router including how to define, configure, and navigate Angular route states. But wait, there's more. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module demonstrates how to configure routing to support more advanced scenarios, such as preloading data and handling nested views. In the prior module, we covered these routing basics. The primary objectives of this module are to use resolve to preload the data for a routing state and define nested routing states to support a multi-view wizard. Nesting routing can provide the navigation for a tab-based set of edit pages. Let's cover the first one first.

Adding Another Route State

Here are the route states we have identified so far for this application. We have already created the welcome page and its associated route state. We created the product list view and its associated route state. Let's now build the product detail view and its associated route state. Based on the imaginary requirements analysis, the product detail view will look like this. We'll use Bootstrap's grid columns to line everything up nicely. The Back button will active the product list state and return to the product list view, and the Edit button will active the product edit state and display the product edit view to edit this product. Creating a new route state requires several steps, all of which we covered in the last module. Let's do a quick review. Creating a new route state requires building a view template in its own HTML file, optionally creating the controller for the view, configuring the state, and activating the state where appropriate. Now let's do a quick demo to set up the route state for the product detail view. Then we can get to the more advanced scenario, which is adding the code to preload the data for the product detail view route state. Here we are back in WebStorm. The first step is to build a view template in its own HTML file. So let's create a new HTML file for the product detail view. We can navigate to app, products, right-click, New, HTML file, and we'll name the file productDetailView. We don't need any of that default HTML. Since these view templates are inserted into the layout template, they don't need the HTML head or body tags. So let's just delete all that. Then I'm just going to paste in the code for this view template. Let's walk through this code. Just like the product list and welcome page, the product detail view is displayed within a Bootstrap panel. This provides a consistent look for the user interface. Note that the title for the page uses the controller alias vm, so this view expects that the title is provided by the controller. The body of this page is a set of rows and columns for each product property. For example, let's look at the product name. Two Bootstrap columns are allocated to the property Name, such as Name: that you see here. The next six columns contain the contents of the property. Each uses a binding expression to bind to the model property, and each uses the controller alias as a prefix. Notice the vm. here. Notice that some of the binding expressions include filters. For example, the release date uses a date filter. The cost and price use the currency filter. And the margin percent uses a number filter. This code displays the image. It uses the ng-source directive that we saw earlier in this course to ensure that Angular replaces the binding expression with the appropriate image URL before the browser attempts to fetch the image. The panel footer contains two buttons. The back button includes the ui-sref directive to activate the product list action. The edit button includes the ui-sref directive to active the product edit action. Notice that this code passes in the product ID as a parameter. This lets the route state know which product that we wish to edit. So, that's the UI. The next step is to create the controller for the view if the view needs a controller. So let's create a new JavaScript file for the product detail controller. We navigate to app, products, right-click, New, JavaScript file, and we'll call it productDetailCtrl. So we don't forget, let's add the script tag for this new JavaScript file now. And we'll put that under Product Controllers. Now let's go back to the controller, and I'll paste in the code. As with all of our controllers, this code begins with the immediately-invoked function expression, or IIFE, and it includes use strict. Then we look up the module. This controller's going to be registered with the main module called productManagement, and register this new controller. The first parameter of the controller method is the controller name. The second parameter is the reference to the controller function. So then here is our controller function. First, we assign this to a different variable. We set the product property of vm to product. We have not yet defined where product is coming from. That's part of the preloading feature we'll add shortly. So for now, let's hard code this instead. The code then sets the page title, and the tags are defined in the JSON within an array, but we want to display them as a comma-separated list, so we can use the two-string function to convert the array to a string. We'll see the result in a moment when we run the application. So that's it. That's our controller. So we've created our view template, and we've created our controller. The next step is to configure the state. We're configuring all of the states in app.js. So we can just add in another state here. This code defines a state named productDetail. It sets the URL fragment identifier for this state to /products/:productID. The :productID denotes a parameter that will be passed to the state very similar to the product edit state. The template URL property specifies the location of the HTML file containing the view template. And the controller property specifies the associated controller including the controller as and the associated alias. The last step is to activate this state. When the user clicks the name of the property in the product list view, we want to activate the product detail state. So let's open the product list view HTML file, scroll down to the code with the product name, and replace the product name with an anchor tag that includes a ui-sref directive. There we have it. So now if the user clicks on the product name, the product detail state is activated. We're passing an object literal to the product detail state that defines the product ID that it should navigate to. While we're in here, let's add a button that will activate the edit route state. So we'll just add another td here at the end. We're using an anchor tag here but specifying a Bootstrap button class so it looks like a button. And we're using ui-sref to activate the product edit state. In this case, we're also passing in an object literal that includes the product ID. That way, the edit knows which product we wish to edit. With all of the steps complete, we should now have a product detail feature. Let's give it a try. So we can click Product List to see the list of products. Click on a product name like Garden Cart, and you see the details. So it has the name and description, the availability nicely formatted as a date, the cost and price nicely formatted as currencies, and our tags are in a nice, comma-separated list. Success! Well, sort of. If I click on any of the products, I get the garden cart. That is, of course, because we hard coded it. So we are now at the point where we are ready to look at how to preload the data needed by this page as part of the routing. Let's do that next.

Using Resolve to Preload Data

In the prior module, we looked at the $stateProvider service. The $stateProvider service has a state method for defining a state. Each state includes a name and an object literal. The properties of the object provide the state details. We looked at the basic set of object properties: URL for the fragment identifier associated with this state, template URL for a link to the HTML file containing the view template for the state, and an optional controller. Another very useful property is resolve. Resolve is a property of the state configuration object that you can attach to a route. It can provide custom content or data to the defined controller. It identifies a set of dependencies that can be used or injected into the controller. If a dependency is a promise, the promise is resolved before the controller is instantiated and the route is changed. This means that if the code is retrieving data, it waits for the data to return before navigating to the associated route and displaying the view. This simplifies the controller because the controller does not need to fetch the data. These dependencies are defined with key value pairs where the key is the name of the dependency. The value can be a string, which defines the name of a service that can be injected, or the value can be a function. We want to use resolve to retrieve the data for the product detail view. Remember the product resource service that we built earlier in this course. We'll use that again to retrieve the data for the detail view. The product resource service calls a web service on a remote web server and retrieves the product data. Previously, we used the query method of the product resource and the product list controller because we wanted an array of products returned for display in the list. The product resource service also provides a get method for returning a single product. We'll use that to retrieve the one product for the product detail view. Why not use the product that's already in memory from the list? Well, we want to ensure that we have the most recent data in case it was updated by another user. Currently, we don't have a web server. We are currently intercepting the calls to the web service with another service which mocks the data. This allows us to run the application without the web service in place. Here is code that uses the resolve property to retrieve the data for the product detail view. Resolve is a property of the state configuration object. Resolve is defined with an object literal compromised of key value pairs. The first key value pair defines a dependency on the product resource service. As we just saw, the product resource service is the service that provides the data for the application. We built this earlier in the course. We are going to use that service to load the data for the detail page. The key is named productResource. This key could have had any name, PR, A, whatever. I named it as the same name as the service for clarity. The value is the string name for the service. The resolve assumes that any string name provided as a value is an alias for the service. The second key value pair defines a dependency on the result of the defined function. The key is named product because this function returns a product, but it could be named anything. The value is a function that returns a promise. This function depends on the product resource service identified in the first key value pair. Hence, the reason we needed to define that dependency first. This function also depends on the $stateParams service. The $stateParams service is part of the UI router. It is a service that is populated by the current state's parameters. This service is needed because our product detail state has a product ID parameter, and we want to obtain that parameter in order to retrieve the appropriate product. The first line of this function uses $stateParams to read the product ID parameter. The second line of this function retrieves the defined resource using the get method of the product resource service passing in an object with the ID of the product to retrieve. The function then returns the associated promise. You can add any number of key value pairs here if you need to obtain multiple sets of data for this state. When this product detail state is matched, this code first pulls the product ID from the parameters passed to this state. It then calls the get method of the product resource service, which returns a promise, and issues a get request for that product. The product resource mock service intercepts that request, looks up the product with the defined ID in its in-memory list of products, and returns the product data. When the data is returned from the service, the state is activated, the controller is constructed, and the product dependency is injected into the controller. The defined template view is then displayed, and the data in that view is populated from the model defined in the controller. Let's add this code to the sample application and give it a try. Here we are back in WebStorm. The app.js file is open. We will add the resolve property to the product detail state. So we need a comma here and the resolve. So as discussed in the slides, the resolve is defining two dependencies, one on the product resource service defined with the key productResource, and the second on a function that gets the defined product. The key for that dependency is product. Next, we need to modify the product detail controller to take in that injected product. So we're going to go back to the product detail controller and then we can inject in product. So we can replace this hard-coded product with the product that was injected in. Now, since we're now passing a parameter into the function, we need to modify the registration to define the product in a Min-Safe array. So we're going to make this second parameter an array and define product in a string as the first element of that array. Now look at how clean and simple our controller is. Now let's run the application and see if we get the right data. So we go to the Product List, Leaf Rake. Yup, that's the leaf rake. Garden Cart, yup, that's still there. So the resolve is getting the appropriate data for the product detail view. It works. Let's review what's happening here. When we click on a link, the state is changed. Behind the scenes, the state is matched in the list of states. The controller is constructed with the retrieved product passed in. The HTML file defined with the template URL is downloaded if it was not already, and it's displayed. Since the HTML file contains binding expressions, those expressions are evaluated, and the data is populated. Notice that the address bar is changed to /products/10 where 10 is the ID of the product to view. Use the resolve property when defining a state to retrieve the data for that state. That ensures that the associated view is not displayed until the data is retrieved and ready. We have just seen how to use the state configuration resolve property to preload data and provide that data to the controller. Check. Now let's change gears and look at nested routing states.

Defining Nested Routing States

Most line of business applications require data entry and editing operations. You've probably seen desktop applications that have six or eight or ten tabs worth of data that the user must enter. What is the best way to display that much data in a web application? How about web-friendly tabs? With tabs, the user can navigate between multiple edit views. We could add Previous and Next buttons so that the user can work through the data entry using a wizard-like user interface. Let's take a closer look at the views required to display this user interface. The outermost view is the layout template. The layout template provides the menu and is defined in the index.html file. Displayed inside the layout template is the edit view. The edit view contains the edit panel and tabs. This ensures that the layout of these elements remains consistent. The appropriate edit detail view is then displayed within the edit view. When the Basic Information tab is active, the basic product information is displayed. When the Price Details tab is clicked, the price detail information appears in this area. And when the Search Tags tab is clicked, the search tags information appears in this area. This arrangement is called nesting. These detail views are nested within the edit view. UI router has features to handle the routing states for these nested views. Here is our current site map. The edit will require at least three edit detail views: basic information, price details, and search tags. Here are the route states we have defined. The editing has one state, productEdit. To handle the nested views, that editing state needs a sub-state for each view. These sub-states are often called nested states. When using UI router, nested states are defined using a dot syntax. So the state name for the basic info is productEdit, which is the primary state, .info, which is the sub-state. For the price view, the state name is productEdit.price, and for the search tags, the state name is productEdit.tags. These are our nested states. The state configuration then looks like this. The first state is the productEdit state that we had already defined. We have not yet created the template file associated with this state, but we will shortly. We have also not yet created the controller. We'll do that in a moment as well. The next state here is the productEdit.info state. This state is for the view that provides editing of basic product information. Notice that the nested state does not require a controller. The nested child states can share the controller from the parent state. Similarly, the next state is productEdit.price. This state is for the view that provides editing of price information. And there's the productEdit.tag state. This state is for the view that provides for editing of the product search tags. We have not created the template files associated with these nested states, but we will shortly. For now, let's add this routing state information to our sample application. Here we are back in WebStorm with the app.js file open. Notice that we already have the productEdit state in place. We just need to add the sub-states or nested states. So here they are. We have the original productEdit state, the productEdit.info state, productEdit.price state, and productEdit.tags state. There is one more thing that we want to do here. The edit pages will need to retrieve a single product. We can use the resolve as we did earlier with the product detail state to retrieve a single product. The cool thing about nested states is that we can add the resolve to just the parent state yet use the resulting product in each nested child state. So I'm simply going to copy this resolve and add it to the edit, so we need a comma there. So there's our resolve. And that works the same as it does for the product detail. We can't try this out at this point because we have not yet created any of the template files defining the HTML for these views.

Defining an Abstract State

Let's take a moment and talk through these states. The productEdit.info state provides for editing the basic product information. The productEdit.price state allows for defining the cost and price. And the productEdit.tag states defines the search tags. But what about the productEdit state? Would we really ever want to activate the productEdit state without one of its nested states? In many cases, no. UI router provides a property that can help you prevent activation of a parent state if it doesn't make sense for your application in cases where the parent state should never be active without one of the nested states. That property is called abstract. An abstract state is a state that cannot be explicitly activated. Attempts to activate it directly will throw an exception. It is activated implicitly when one of its child states are activated. Abstract is another state configuration property as shown here. By setting the abstract property to true for the product edit state, that state won't be activated directly. Rather, it will be automatically activated when a nested child state is activated. Let's add that to the sample code. Here we are back in WebStorm with the app.js file open. We'll modify the product edit state to be abstract by simply adding abstract here. That it. That's all we have to do. We can't try this out yet because we have not yet created the edit view templates. Let's create those next.

Building the Nested Templates

To try out these nested states, we need to create the associated edit view templates. Each template is specified with the template URL property. We'll create the actual data entry forms in the next module. For now, let's just create the edit view template files with some hard-coded text. That way, we can test out the routing. So let's create each of the three nested view templates next. Here we are back in WebStorm. We'll start by creating the product edit info view template in its own HTML file. So we'll go to app, products, right-click, New, HTML file, and we'll name it productEditInfoView. We don't need any of that default HTML. Since these few templates will be inserted into the main edit template, they don't need the HTML head or body tags, so we're just going to delete them. And we'll just add some simple text just for testing purposes. We'll do the same for the price view template. (Typing) And repeat for the search tags. (Typing) We also need to create the template for the product edit state. This template includes a panel, a panel header, a tab bar, and space for the child templates to be inserted. Let's build it. And we are back in WebStorm. We'll start by creating the product edit view template in its own HTML file. Right-click, New, HTML file, and we'll call this productEditView. Again, we don't need the standard HTML. We'll define a panel so that the edit view has a consistent look with the other pages of the application. Inside that panel, we'll all a header. The body of the page is comprised of the tabs, and a space for the nested child templates will appear. Let's talk through this code. We are styling the tabs using a wizard class. That is not a class provided by Bootstrap. We'll have to build that CSS class ourselves. The anchor tags use the ui-sref directive to activate the appropriate nested state. So the basic info activates .info, pricing activates .price, and search tags activates .tags. Notice that we didn't need to prefix it with the parent's name here. It wasn't productEdit.info. That's because we're in the product edit page, and it knows its parent state. The ui-sref active directive adds the defined CSS class to the element when the related element is active. This will allow us to show which tab is active. At the bottom, we use the ui-view directive to define the location in this view where the nested views should appear. Looking at the app.js file, we can see that the product edit state also has a controller defined, so let's create a new JavaScript file for the product edit controller. App, products, right-click, New, JavaScript file, and we'll call it productEditCtrl. So we don't forget, let's add the script tag for this new JavaScript file now. So we'll go to index.html, and we'll add the script tag for this controller. Now let's go back to the controller and insert the code. As with all of our controllers, it begins with the immediately-invoked function expression, or IIFE, and it includes use strict. Then we look up the module. In this case, we're registering the controller with the main module, and it registers this new controller. The first parameter of the controller method is the controller name. The second parameter is an array. The first element of the array is the parameter for the function. The second element of the array is a reference to the controller function itself. We are retrieving the required product using the resolve feature so the resulting product is injected into this function. Within the controller function, we assign this to a different variable, and we assign the passed in product to the product object on the model. We then define the title to display in the panel. If it has an ID, we're assuming it's an existing product and that we're editing it. If it has no product or no product ID, we change the title to New Product. A few additional changes and we can give this a try. In index.html, we activated the product edit state. Well, that state can no longer be activated. We instead want to activate one of the child states, so we probably want to start with the info state. So we'll put .info there. Same thing is true for the list view. We have an edit button here where we're activating the product edit state. That's no longer a valid state, so we'll activate productEdit.info instead. Let's give it a try. There's our product list. We can click the Edit button, and there's our basic product information. We can click Price Details and Search Tags. So all of our appropriate pages are being displayed. But the tabs don't look much like tabs. Let's fix that next. Earlier in this course, we added an app.css file to the CSS folder of the project. That's where we can add the style classes for the wizard tabs. So I'll paste them in here. Now we have our wizard style class defined. So let's try the application again. Product list; we'll edit the leaf rake. Oh, that looks nicer. We have tabs. But we don't have anything that's highlighting the active tab. Let's do that next. So we'll go back to app.css, and we'll add a .active, and we'll try running again. Go the product list. Ooh, now we've got the active tab highlighted. Sweet. Now we have a set of tabs. Each tab activates a specific nested state. The view template for that active state is then displayed within the parent edit view template. We can add as many tabs and nested states as we need for displaying data for edit.

Summary

This module completed our discussion of route states that began in the last module. In this module, we looked at how to use the resolve property of a state to preload data and how to define nested routing states to build a tab-based edit page. We also saw how to define a state as abstract so that it cannot be activated. Looking back at the set of tasks we identified for building a line of business application, we have now laid out the first set of views for the sample application and added route states to navigate between them. The application can now navigate to each of the first set of application features, including the edit features. But the edit views have no content. In the next module, we'll look at completing the edit views by building forms for the data entry.

Building Data Entry Forms

Introduction

Business runs on data, and where does that data come from? Often, from a person typing into a data entry form. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module demonstrates how to build clear and helpful data entry forms using Angular and Bootstrap. It's not enough for a data entry form just to collect data. It should facilitate entry of relevant, accurate, and valid data. The user may be entering a customer or an order, or updating employee or payroll data, or defining the products that your company sells. So however possible, the form should facilitate entry of accurate and valid data. The data entry form should also help prevent entry of incorrect, invalid, and erroneous data. Incorrect order information, invalid payroll information, or erroneous product data displayed on your company's website can cause major problems. It can result in corporate fines or tax consequences or expensive lawsuits or loss of good will. So however possible, the forms should help prevent entry of incorrect or invalid data. A data entry form can accomplish these things through clarity, helpful data entry controls that guide the user to valid data entry, and, of course, validation. This module covers these first two. Validation is covered in the next module. When laying out data entry forms, clarity is key. Clarity is achieved through defining intent. To ensure that relevant, accurate, and valid data is entered, the intent of each data entry field must be clear. Ever tried to use a health insurance website? Your insurance card has an identification number, group number, and contract code. But, of course, the web application asks for your insurance ID. Is that one of the numbers on the card? A login ID? Something else? Use a clear label that the user will understand, and be sure to use the words from the domain, basically the business terminology, not technical terms. Where feasible, add a placeholder. The placeholder can provide additional information on the value that the field is expecting. The second way to achieve clarity is through organization. Especially if there are many data entry fields, having those fields organized aids the user in entering that data. There are many options for organizing the data entry fields. You can group the data entry fields by common function. Then provide access to each group of fields using a toolbar or menu. Or you can use tabs as we have in the sample application. Panels, group boxes, and simple lines are other options for grouping related data entry fields. Data entry controls can guide the user to valid data entry and minimize the chances of an entry error. This module will cover two of those controls. First, a mask edit control. A mask edit control restricts the text that can be entered to characters matching a specific mask. For example, it can restrict data entry to only numbers or a specific combination of numbers and letters. In addition, the mask can format the entry to match an expected layout. For example, a product code may be three letters, a dash, and then four alphanumeric values. Displaying the data in a format that the user recognizes increases the probability of valid data entry. The second control we will cover is the datepicker. The datepicker assists the user with entry or selection of a date. A good datepicker allows entry of the date in any of the common data entry formats. The primary objects of this module are to create a data entry form, bind the UI elements to the model, style the form with Bootstrap, use the mask edit directive, use the data picker directive, and submit the form to save new or edited data. Let's get started.

Creating a Data Entry Form

The basic process for creating a data entry form in an Angular application is straightforward. If you have ever created an HTML form, there is nothing new here. The form element defines the form and gives it a name. The field set element groups controls that are related. And the legend element displays a caption on a field set. Adding controls to the form is all basic HTML as well. This code adds two fields--a text box for the product name and a text area for the product description. Using a text area allows for multiple lines. In this case, the text area displays three lines for data entry. Each field defines placeholder text that is displayed within the data entry box. Each field also has a label. The for attribute of the label associates the label with its data entry control by using its ID property. This improves usability because clicking on the label activates the associated data entry control. Let's add a data entry form and these controls to the sample application. Here we are back in WebStorm. We have the productEditInfoView.html file open. Recall that we created this HTML file in the last module to demonstrate the tab-based navigation. Let's delete the label we were using for testing. Now let's add the form and the first two controls using the code from the slides. So this defines our form, our field set, our legend, and then inside the field set, we have the product name and the description. Let's run it and see how it looks. Click on Add Product to display the form. It doesn't look great, but I can type information into the controls. Notice that clicking on the label activates the associated data entry control. That's due to the for attribute we added to the label control. But what if I view this form in an editing scenario? We can go to Product List, pick to edit the hammer. The edit page appears and has the correct header, but the form is not populated with the existing data. We are missing the data binding. Let's do that next.

Binding UI Elements to the Model

One of the key benefits of using Angular is its awesome data binding features. In the product list view, we bound UI elements to the model defined in the controller using binding expressions. Binding expressions are denoted with double curly braces. This is one-way binding, meaning that the binding expression displays the data from the model, but a binding expression cannot update the model data. That was fine in the case of the product list because the data was display only. In the productEditInfoView form, we need two-way binding. We need to obtain the value from the model when displaying the data for edit, and we need to update the model with any data entered or changed by the user. For two-way binding, we can use the ng-model directive. The ng-model directive provides two-way data binding by synchronizing the model to the view, as well as the view to the model. Let's add ng-model to each of the data entry elements in the sample application. Here we are back in WebStorm. We can add the ng-model directive to each data entry element. Let's start with the product name. We'll add ng-model, and we'll set it to vm, that's the alias for the controller, .product, that's the variable containing the product retrieved from the web service, .productName, that's the property of the product containing the name of the product. Now let's repeat that for the text area. Ng-model=vm.product.description. Now let's give it a try. We'll go to Product List, click Edit. Wow! We don't even have a controller for this page, but it populated the two controls. How did this work? Let's look back at the app.js file. Recall from the last module that this is where we identified our routing states. We defined a parent state called productEdit and marked it as an abstract state. We then added nested states for each of the three tabs defined for editing the data. If a nested state does not specify a controller, the nested state uses the parent controller, which in this case is ProductEditCtrl. When a nested state is activated, the parent state is also activated. The resolve property for the productEdit state calls the web server and returns the product for edit. The returned product is then injected into the parent controller. The parent controller was ProductEditCtrl, so let's look at that. So you can see here that the returned product is injected into this controller. Code in that controller then assigns the product to the model using vm.product. That's why our ng-model used vm.product and then the name of the product property. Let's run that again. Product List, Edit, garden cart, and you can see it has the data. So it works. But it's not very pretty. Let's add some styling to our form next.

Styling the Form With Bootstrap

Even though our application may only be used by internal personnel, adding some style gives the form clarity and makes it easier to use. That in turn aids entry of appropriate data. Here we are on the Bootstrap website. Bootstrap provides several different form styles, and there is a large third-party market that provides many more form styling options. I'm selecting CSS here to see the standard Bootstrap styles. I'm selecting Forms from the right menu. Bootstrap provides three form styles. Basic, as shown here, displays the label above each data entry element. Inline, as shown here, displays the controls all in one line. And horizontal, as shown here, displays the labels to the left of each data entry element. For our design, we'll use the horizontal form style. Notice the styling classes here. The form has a style class of form-horizontal to display the label to the left of each data entry element. Each label and control combination has a class of form-group. This provides for optimal spacing of the controls within the group. Each label has a class of control-label, and each control has a class of form-control. This ensures that the control takes the full width available to it. The form also uses the call-classes to define the layout. Recall that Bootstrap uses a 12-column layout. Here, the labels fill the first two columns, and the controls fill the last ten columns. Are we ready to give this a try in our sample application? Here we are back in WebStorm. We have the productEditInfoView.html file open. Instead of watching me type all of the Bootstrap classes in, I'll paste them in, and we can talk through them. The form class defines a horizontal-styled form. Each label and control combination is styled within a form group. Each label is defined as two columns wide and given a control label style. Each data entry control is enclosed in a div tag. The textbox is only four columns wide, but the text area is ten columns wide. Both controls use the form-control class styling. Let's see how it looks. Ah, that's better. We can tweak the look by adjusting some application-specific styles in our app.css file. Let's add some styles for our form and form legend. So we'll open our app.css file. Let's add some styles for our form and our form legend. Now let's run it again. Go to product list, edit, looking good. How about the add feature. That looks good too. But now we need to add the product code, and it needs to be in a specific format. Let's see how to do that with a mask.

Using a Mask Edit

In business applications, there are many cases where an input must follow a specific format, for example, a phone number or product code. One way to confirm valid data entry is to let the user know what the valid format is and then use validation to ensure that the user entered the data in the appropriate format. Another option is to use an input mask to provide more controlled guidance for the entry. An input mask is not part of a standard HTML input control, nor is it part of Angular. But there is a ui-mask directive available in the Angular UI.Utils tools. Setting up the ui-mask directive is a process that should look familiar at this point. Start by locating and downloading the required JavaScript file. You can find ui-mask as part of UI-Utils at this URL. Add ui-utils.js to the .js directory of the project. Add a reference to this new .js file in index.html. Add a module dependency on ui.mask, and add the ui-mask directive to the desired input control. In this example, we are adding a product code using the same layout and styling pattern we used previously. The ui-mask directive is defined in the input tag. Set the directive value to the desired mask. In this case, the product code starts with three letters, then a dash, then four alphanumeric values. The mask uses the following key: A capital A accepts any letter. A 9 accepts any number. An asterisk accepts any letter or number. Any other symbol is inserted into the result as is, as shown with the dash in this example. Use a mask for phone numbers, credit card numbers, or any other place that requires strict entry of numeric, alpha, or alpha numeric values. Let's add this product code input field to our sample application, so we can give it a try. Here we are back in WebStorm. I have already downloaded the UI-Utils library and added the ui-utils.js file to the .js folder in the project. We then need to add a reference to this js file and index.html. I'll add it right there. Next, we need to add a dependency on ui.mask to our app.js. So we'll add the dependency here. Then we want to go to our HTML file and add the product code label and control. I'm going to put this beneath the description, so we added a label and an input control. The ui-mask directive defines the mask that we will use for that control, and the ng-model defines the two-way binding to the product code. Let's see it in action. So, we go to Product List. We'll edit the saw, and there it is. You can see the product code. So let's clear this product code. I can type in three letters, AAA, the cursor is automatically moved for me. I can then type in four alphanumeric characters, 1234. If I clear the field and try to type in a number--I'm clicking the 1--you can see that I can't type in a number because I defined the first three characters as requiring an alphabetic character. So using a mask provides more control over the user's data entry. But what about dates? Let's look at those next.

Using a Date Picker

A common requirement in many applications is entry of a date, such as a hire date or due date or effective date. The problem with dates is that there are many valid date formats. Using a datepicker control can make it easier to accept entry of multiple date formats, or the user can simply pick a date using the dropdown calendar. The datepicker is available as part of UI Bootstrap. UI Bootstrap provides a set of Bootstrap components that are written in AngularJS and are available for use in your application. Here we are on the UI Bootstrap site. If you look under Directives, you'll see all of the different components provided here. We are only going to use the datepicker at this point. If you have a chance, check out some of these other cool components. You can find the documentation for the datepicker here. We again have similar steps for setting up this new datepicker component. Start by locating and downloading the required JavaScript file. You can find the datepicker-popup as part of UI Bootstrap at this URL. Add this JavaScript file to the .js directory of the project. Add a reference to this new .js file in index.html. Add a module dependency on ui.bootstrap. And add the datepicker-popup directive to the desired input control. In this example, we are adding a product availability date using the same layout and styling pattern we used previously. The availability date is the date that the product is available in inventory. The datepicker-popup directive is defined in the input tag. Set the directive value to the default format for display of the date. In this example, we're defining the month abbreviation, day, and year. But you can define any valid date format. The span tag adds a button with a calendar icon. The user can click this button to open or close the datepicker. You can add a datepicker anywhere you need to request entry of a date. Let's add this date to our sample application. Once again, we are back in WebStorm. I have already downloaded the UI-Bootstrap component and added it in the .js folder. We then need to add a reference to this .js file in index.html. I'll put it here. And we need to add a dependency on UI-Bootstrap to app.js, so we'll put that here. Now we'll go to our form. The first thing that we want to do is move the HTML elements for the product code. Recall that the description is a wide field. The form will look better if the wide description field is at the bottom of the form. So we're going to move this above the description field. We'll also add the availability date label and control above this description. And we'll paste in the code from the slides. Here we added a form group following the pattern we have used previously. We defined a label and an input box. We set the type of input box to text, not a date. That way the browser won't attempt to handle the date, and we'll let the datepicker handle it instead. We set the datepicker-popup directive to the date format we want to display in the control. Any valid date format will work here. We set the ng-model directive to the product's release date, which is defined in the JSON as a string. The ng-model directive sets up the two-way binding. The button tag defines a button with a calendar icon. The user can click this button to dropdown a calendar and pick a date. Let's see this code in action. We'll select Product List and edit the hammer. And we can see that the date is displayed. We can modify this date. But the datepicker doesn't open. That's because we don't have any code set up for the button's click event. Let's do that now. Let's look back at app.js. Here again are our states. Notice that we don't currently have a controller defined for the productEdit.info state. So we don't currently have a place to put any code for the button. So we have a decision to make. Should we add the required code for the datepicker button to the parent controller or build a controller for this child page? For this sample, we'll add the required code to the parent controller, but in a real application, you may want to add a separate controller for each state. That keeps the code encapsulated. Let's define an open method in the parent controller. So we're going to go to ProductEditCtrl, and we're going to define an open method in the parent controller. I'll paste that code in, and then we can talk through it. So here's the open method. This function takes $event as a parameter. $event represents the original event object, and we'll need to pass that in. Within the function, we first call preventDefault on the event to prevent any default action from being triggered. We only want our action that we're defining within this function to be executed. Then we'll call stopPropagation on the event to prevent the event from being propagated. Lastly, we'll add a variable onto the model. The variable is opened. When the button is clicked, this code reverses the value of the variable, so if the calendar is closed, this code will define that it should be opened. And if it's open, the click will define that it should be closed. So the next thing we need to do is add the ng-click directive onto the button passing in the event. We do that in productInfoView. Here's the calendar button. We need to add the ng-click directive. We want to assign this directive to the appropriate method in the controller. When the user clicks on the button, we want to call the open method and pass in the current event, which is defined with $event. So we're going to call vm.open and pass in $event. Recall that we defined a variable called opened on the model. The opened variable tracks whether or not the calendar is open. We can bind the is-open attribute of the datepicker to that model variable we defined. So let's do that. Here, we'll add an is-open, and we're going to bind that to our opened variable. Now let's try the datepicker again. Go to Product List, select to edit the hammer, and you can see our date is displayed. We can click the calendar button, and the calendar pops down. We can then pick a different value from the popup. If we don't want to change the date, we can simply click the button again, and the popup closes. Cool! We now have a user-friendly datepicker. But we are still missing one key feature here on our form--the save feature. Let's do that next.

Submitting the Form

When a user enters new data or updates existing data, that updated data needs to be posted back to the web server. The web server can then store the updates in the appropriate data store. So what do we need to do to save the data from our form? To save the data from our form, we first need to define the appropriate resource for communication with the web server. Check. We did that already. The single resource we identified for the retrieve will work for the save as well. Next, we need to mock the resource URL so we can run the application without the web server. Check. We've done that too. When we set up the mocking earlier in this course, we added the mocking for the save. We do need some UI element for the user to click to perform the save operation. And we need a submit function to execute when the user clicks the save. What would the submit function look like? If you recall from our data module earlier in this course, We are using a resource service to communicate with the web server. The resource service provides a $save method on the object's returned from the server. So all we need to do is call the $save method on the product to be saved. That's it. Let's do that now. Here we are back in WebStorm once again. The first thing we want to do is add a save button. Let's put it at the button of the productEditInfoView. While we're at it, let's add a cancel button as well. So this code identifies another form group, and in it, it has a button that's identified as the submit button and another button identified as the cancel button. The ng-click directive for the submit button calls the submit method in our controller. The ng-click directive on the cancel button calls a cancel method on the controller. So we'll need to add those two functions next. Notice, however, that we are using the click directives for the submittal. We are not adding any kind of submit onto the form itself. Recall that we don't have a controller for the productEditInfoView. Rather, we're using the parent controller. The parent controller is the ProductEditCtrl, so that's where we'll add our new methods. First, we'll add the submit. The submit function simply calls the $save method off of the current product, which is specified with vm.product. And here's the cancel function. Recall from earlier in this course that we can use $state.go in order to navigate to a different state. On cancel, we're navigating back to the product list. Since we are using $state here, we need to inject it into the controller. So we're going to add $state here. And since we added it as a parameter to the function, we need to add it to the Min-Safe array as well. So we'll add it here. One other thing to point out here, notice that we're not using a copy when we're assigning vm.product to the product that we retrieved from the web server. That is because every operation, the product list operation, the product detail display, and the edit operations always get the latest product data from the web server. This ensures that we see any changes made by any other users and prevents the need then for a copy. Okay, so with the submit and cancel in place, that should be it. Let's try it out. We'll go to the Product List. We can go to the hammer, and we can change the product name. We'll just do a silly edit, 1234, and we'll click Save. Notice we don't yet display a message to the user notifying them that the save completed successfully. We'll add notifications at the end of this module. Now let's go back to the product list. Yes, our product was updated. Next, let's try adding a new product. We can enter all the data entry fields. After we finish data entry, we can click Save. Going back to the product list, we can see our updates there and our new product is there. We were able to update an existing product and add a new product. So we now have an application that retrieves and saves data. But we only have one tab of the edit page complete. Let's do the search tags form next.

Implementing the Search Tags Form

Before we move on, let's reinforce what we've learned in this module by implementing the third tab of the edit page. The search tag form allows for entry of the key words associated with a product. These key words will be used as search tags for the product. Let's build this form now. Here we are back in WebStorm. We have the productEditTagsView.html file open. Recall that we created this HTML file earlier in this course to complete the tab-based navigation. Now let's remove the hard-coded text from here and add the layout for this form. This form uses the form-horizontal style for consistency with the product info form. And it uses the same pattern for the controls on the form, with a form group style, a label, and an input control. The ng-model directive on this input control defines a new model property called newTags. So the newTags property is bound to the data entry box. That data entry box provides a space for the user to enter search tags applicable to this product. Those tags should be separated with commas. We are also adding a button as part of the form group. The user will use the button to add the entered tags. The ng-click directive on this button calls a method on the controller to add the defined tags for the product. We'll need to create this method. Below the form is a set of buttons that display the current set of search tags associated with this product. This code uses the ng-repeat directive to display each tag in its own button. The ng-click directive on each button calls a method on the controller to remove the defined tag if the user clicks on the button. We'll need to create this method as well. At the very bottom of the form are the submit and cancel buttons. These are very similar to the submit and cancel buttons on the product info form. Let's look back at the app.js file. Notice that the productEdit.tag state does not define a controller. We could add a controller for this page and put the code to implement the tags into that controller, or we can simply reuse the parent controller. We'll reuse the parent controller for this demonstration. So let's open the ProductEditCtrl file. In this file, we'll insert two new methods. Here's the new addTags method. We pass the list of tags that the user entered into this function. If the user didn't enter any tags, we display an alert box. Otherwise, we split the list on the comma, and we append it to our list of tags for the product. Then, we clear the bound newTags property so that the user can enter additional new tags. And here is the new removeTag function. Here we're passing in the index of the tag to remove, and we're using the splice method to remove that particular product. Let's give this a try. We'll go to the Product List. We'll edit a product, and we'll go to the Search Tags tab. You can see that there's currently one search tag, and we can add some more. So I separate the items with a comma, and then I can click Add, and it adds the additional search tags. If I click on a tag, the tag is removed. If I then save, notice we've got hardware and utilities now, again we don't have a notification yet that the save was successful. We'll want to add that. We can then go to the Product List, show the hammer details, and we can see that it has the new tags. So the data was saved successfully. Before we move on, let's do one more thing. Notice that when we type in a search tag here and click Add, we get visual feedback that it actually added the tag. But when we click Save, we don't get any feedback. How will the user know whether the save worked. If we don't display something, the user may click the save multiple times to ensure that it actually saved. A quick and easy way to provide notifications is with a toastr-style component. There are several implementations of the JavaScript toastr notification, but we're going to use the one from here. This component displays simple notifications. You can use the Download ZIP button here in the lower right to obtain the files. I've already downloaded the files and put toastr.js under the .js directory. And I've put toastr.css under the CSS directory. Then we need to add the script tag for the .js file to index.html. So we'll put it here. And we also need to add the link for the CSS file. We'll put that here. We can then modify the submit method to display a notification popup using the toastr. So we'll go to the controller, and we'll go to the submit. We can pass a function into the save method that is executed upon a successful save. We'll add that function and display the notification in there. So now let's give it a try. Now notice every time that we come back here, our edits are gone. That's because we've stopped and started the application. Every time we restart the application, we get the default set of mocked data. The mocking retains an in-memory set of data, so while we have the application running, all of the updates are retained. But as soon as we stop the application, all of those updates are gone, and we're back to the original set of mocked date. So let's add another search tag here again, and we'll add it to the list and click Save. Now we have our little popup coming up letting us know that the save was successful, and it automatically goes away. So now the users know whether the save was successful.

Summary

In this module, we experienced some of the elegance and magic that is Angular. We saw that communicating with a web server using $resource makes everything easier when working with forms. Since we set up the data earlier in this course using $resource, we could readily retrieve the data for use in editing and save the data with a simple call to $save. We didn't have to do anything else. Nested route states simplify binding. By allowing child views to bind to the parent controller's model, there's no need to pass data around or manually share view data. Since we set up the routing and the nested route states earlier in this course, we didn't have to write any code to set up a model for the editing feature. We also saw that ng-model is magic. Well, not really, but it certainly reduces the amount of code required to get data onto a form and read the data from that form. Ng-model's two-way binding handles all of that automatically. Just set the ng-model directive to a property of the controller's model and you are set, or in our case, the parent controller's model. Putting these concepts together really show the power and benefits of Angular. We covered a lot in this module. We created the data entry forms for entry of basic product information and for the product search tags. We looked at one-way versus two-way data binding and set up two-way data binding on the forms using the ng-model directive. We styled those forms with Bootstrap. We added the mask edit directive to ensure that the property code matched a specific mask. We added a datepicker directive to provide flexibility in entering or picking a date. Finally, we wrote the very small amount of code required to submit the form data back to the server and used toastr to notify the user that the save was successful. Here again is the set of tasks that we originally identified for building a line of business application. This module was all about the data entry forms. We laid out the forms, styled them with Bootstrap, and we demonstrated how to submit the data from the forms to a web server with very little code. But we don't currently validate any of the data. Let's do validation next.

Validating Forms

Introduction

The last module was all about accepting user input into data entry forms and submitting that data to a web server. But that process is missing a step--client-side data validation. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module demonstrates client-side validation of data entry forms. There are several benefits to performing validation in the client-side Angular application. First, the user can be notified of any validation issues immediately as the data is entered. This provides a better user experience because the user gets instant feedback on the issue, and hopefully information on how to correct it. Second, validating in the client-side application means that the data submitted to the web server is of higher quality, and that means fewer round trips to the server. If the server catches a validation error, the server has to notify the client application of the error. The client application must then notify the user, and the user must correct the error and resubmit the data to the server, resulting in another round trip. So client-side validation results in submittal of higher quality data, which in turn results in fewer round trips to the server. Keep in mind, however, that no matter how good or how thorough the client-side validation is, server-side validation is still required. The server-side web service processing the submitted data should never depend solely on the client-side validation. There are several techniques for implementing validation in an Angular application. You can use one of the special type attributes on an input element, such as email, URL, or TEL for telephone. Angular provides validation for these types. You can use the input element's required attribute, which specifies that the user must provide a value for the input element. Angular performs required entry checking. You can use Angular validation attributes on an input element. Angular provides attributes such as ng-minlength, ng-maxlength, and ng-pattern. Angular performs the specified validation. You can build your own custom Angular directive for validation. This technique is the best option for complex validation scenarios that exceed what is built in. Or you could ignore Angular's built-in validation features entirely and validate in the controller or in other JavaScript code. Though what's the point of using Angular if not to use its power? This module covers these two. The techniques shown in this module also work with these special type attributes. Building your own custom directive is beyond the scope of this course, and I don't recommend this last approach. That brings us to the objectives for this module. The primary objectives of this validation module are to prepare the form for validation, identify required fields using the HTML required attribute, mark invalid fields, display validation messages, use Angular validation attributes, and, of course, prevent submittal of invalid data. Let's get validating.

Preparing the Form for Validation

Before adding validation, there are a few preparatory steps that are required. To successfully use Angular validation, the data entry form must have a name. That name is used to reference the controls on the form from Angular. And it's a good idea to set the novalidate attribute on the form element. This turns off the browser's built-in validation and allows Angular to handle the validation instead. One note here before we move on: Recall in the last module that we used basic HTML tags like the form tag as shown here to create a data entry form in an Angular application. That was a little oversimplified. Angular actually has a set of directives that replace the standard HTML elements such as form and input. This replacement happens automatically, so what looks like standard HTML form and input elements are really Angular directives. Because these HTML elements are transformed into Angular directives, Angular can provide some additional attributes on these elements. Now let's prepare our form for validation. Here we are back in WebStorm. Our productEditInfoview.html page is open. We already gave the form a name--it's just called productForm--all we need to do then is add the novalidate attribute. So now the form's ready. Let's start the validation by identifying the required fields.

Identifying Required Fields

In business applications, there are often data entry elements that the user must enter. In our sample application, the product name and product code must be entered. Without these two fields, it would be difficult for the user to look up the product again. You can mark data entry elements that must contain data as required fields. Once you have defined which fields are required, you need to consider how those fields will be indicated to the user. Clarity is key. An asterisk in the field label is a common technique for denoting required fields. Some applications add an asterisk at the end of the label. Other applications add the asterisk before the label. Another option is to simply add required after the field, or you can add required to the placeholder. The technique you select will depend on your users and the application. Are your users used to seeing an asterisk? Whatever you decide, be consistent and be clear. Here we are back in WebStorm with the productEditInfoView.html page open. First, let's make the product name a required field. We'll start by adding the required attribute. Then we'll add required to the placeholder. Then we'll do the same to the product code to make it a required field. When using a mask, the placeholder does not work very well, but we can instead add required directly to the mask. There we are, we now have two required fields--the product name and the product code. Let's give that a try. We'll select Add Product, and we can see that the placeholder now says required, as does the mask. But if we leave the fields blank, we are not getting any indication of a validation error. Let's add that next.

Marking Invalid Elements

Performing validation checking is not very useful if you don't also notify the user that the value is invalid. Marking invalid input elements in an Angular application relies on three features: the ng-class directive, Bootstrap styles, and Angular validation states. Here's an example. This example uses the Angular ng-class directive to set the has-error Bootstrap style class when the Angular validation state for the product name is invalid. Let's take the syntax step by step. The Angular ng-class allows setting a CSS style class dynamically based on an expression that defines the set of desired classes. That expression can be a string containing space-delimited class names, an array whose elements are strings containing space-delimited class names, or an object literal defining a set of key value pairs. The key is the style class name. In this example, the key is has-error. That's the name of the Bootstrap style class, and the value is an expression that is evaluated. In this example, it is an Angular validation state, which we will cover shortly. If the expression is truthy, the class name defined in the key is applied to the element. Using the ng-class directive allows for logic to define whether or not to apply a specific style or set of styles to an element, which works great for our validation. Bootstrap provides several validation styles. This is a screen shot from the Bootstrap documentation. The has-success style by default displays in green. The has-warning style displays in brown. And the has-error style displays in red. We will use the has-error style to mark any input elements that have validation errors. A valid entry we'll leave as is. We won't use the has-success style, but you could use these other styles as desired. We said that marking invalid input elements in an Angular application relies on three features: the ng-class directive, which sets the style class based on an expression, the Bootstrap style, and Angular validation states. Angular provides a predefined set of validation states. $pristine identifies that the entry has not been changed. $dirty specifies that the entry has been changed. $valid specifies that the entry is valid based on any defined validation for the element. $invalid specifies that the entry is invalid based on the defined validation for the element. And $error gives you the list of all validations on the element and whether they are valid or invalid. It is basically a set of key value pairs where the key is the name of the validation rule applied to the element and the value is a Boolean. That Boolean is false if the validation rule is broken. We'll use $error later in this module. In order to use these Angular validation states in the form, there are some prerequisites. The form must have a name. The input element must have a name. And the input element must be bound using the ng-model directive. The ng-model directive must be set for Angular to define the validation states. So let's show the user the validation errors by marking the elements with the has error style class using this technique. Here we are back in WebStorm with the productEditInfoView.html page open. As we just saw in the slides, we need three things in order to meet the prerequisite for marking the input element when it is invalid. We need a form name. Check. A name on the input elements. We have an ID, but we don't have a name. We'll need to add that. And we need the ng-model directive on the input elements. We already have that defined. So let's add names to the data entry elements that we plan to validate. We'll do the product name first. So we want to give it a name, and that name can be the same name as the ID, so we'll just use that. And we'll do the same thing for the product code. And we'll set it to the same name as the ID. Then we need to use the ng-class directive to set a Bootstrap style class when the data entry element is invalid. If we add this class to the form group div, then it will mark both the label and data entry element as invalid. This draws the user's attention to the invalid field. So we'll put it here on this div. We'll do ng-class, and we'll set it equal to an object literal. The key in the object literal is the name of the Bootstrap style class. We are using the has-error class. We need to put that in single quotes because the whole expression is already in double quotes. Then we need to specify the value of the key value pair. That's the expression that we want to evaluate. We want to evaluate the $invalid state for the input element. So we need to reference the input element. We do that by using the form name, which is productForm., then the input element name, which is inputproductName, and then we can reference $invalid. We want to make the same change to the product code, so we'll go to its form group div, and we'll add a similar ng-class directive. So this line of code adds the has-error style class to this div element when the product form's product name element is invalid. And this one does the same thing for the product form's product code element. Let's give this a try. Select Add Product, and there they are. The input element appears immediately showing us that there is an error. And the label even turns red. If we type something, the style is immediately changed, and the data entry element is no longer marked as invalid. Since the product code is a mask, typing in a single character won't make the field valid. I need to make the mask valid, as well. So now it meets the requirement of the mask, and the product code is now valid. If I delete the value, the field is again marked with a validation error. But how will the user know what's really wrong here? It would be nicer if we also displayed a message telling the user why the input field is red. Let's add that next.

Displaying Validation Messages

In addition to marking fields as invalid, it is helpful to display a validation message providing details on what is wrong with an entry and how to fix it. There are several ways to display validation messages. One option is an alert box. This is not a very good option for several reasons. First, it interrupts the user's data entry flow. The user must stop what they are doing and immediately respond to the alert. And second, when the user dismisses the alert, the messages are gone. So they have to remember the message details. Displaying a non-obtrusive message on the screen is a much better option. You can display all of the validation messages in a message area somewhere on the form, such as the bottom of the form, at the time, or along the right hand side. But my preference is to display the validation messages adjacent to the associated input element, either below or to the right of the element depending on the layout of the form. Like this. The user can readily see which input element has a problem and what the problem is. To display the message to the right of the input box, use a span element. On the span element, use the ng-show directive to only show the message if the validation state is invalid. Notice that the syntax here is very similar to the syntax we used to mark the data entry field with the has-error style. Let's add this to the edit form. Here were are back in WebStorm. The productEditInfoView.html page is open since that's the one we're currently validating. We want to add a span element within the form group div for the product name, so I'll insert the code from the slide. So this is adding a span that will appear immediately following the input element. It's using the Bootstrap classes help-block and has-error. The ng-show directive will ensure that the span only appears if the input product name on the product form is invalid. We'll add a similar block to the product code. Now let's give it a try. We'll try it with the edit form, so we'll go to Product List. We'll edit the hammer, and if we remove the entry here, we can see that the input element shows us that there's an error and displays the message to the right. As soon as we type in something, that message disappears. Now let's look at the add. We can click Add Product. Notice that the edit fields are immediately marked with an error and that error messages appear right away. This is both distracting and somewhat rude. Maybe we should wait to display the validation errors until after the user has had an opportunity to fill out the form and only tell them that there's an error if they actually make an error. We can do that by using the $dirty validation state, in addition to the $invalid state. Let's do that now. So we'll go back to the productEditInfoView, and where it has the ng-class has-error $invalid, we're going to do an "and" (&&), and we're going to "and" it with the productForm.inputProductName. $dirty, and we'll do this same thing for the product code. Alright, let's give that a try. So now when we go into Add Product, the message appears, but it's not shown in red, so it's just providing some basic information. Now when I type in something, the message goes away. If I then remove it, now it shows in red, because now I've made the error. So now we've done some basic validation primarily using the required attribute. But what about other types of validation, such as minimum length, maximum length? We'll look at those next.

Using Angular Validation Attributes

In addition to the required attribute and use of the type attribute, such as for email validation, there are other Angular validation attributes available. Ng-minlength defines the minimum required length for the value in a data entry element. Ng-maxlength defines a maximum length for the value of a data entry element. Ng-pattern validates an entry against a regular expression pattern and can be used for more complex data entry validation than can be handled by a simple mask. Ng-change allows for definition of an expression that is evaluated when the content of the input element changes. This allows for some level of custom validation code. Often times, a single data entry element may require multiple validation rules. For example, the product name. It's required, has a minimum length of four characters, and a maximum length of 12 characters. Let's give the product name a min and max length and see how to provide multiple validation rules on an element. Here we are back in WebStorm. Once again, we're editing the productEditInfoView.html page. We're going to add an ng-minlength to the product name element, so we'll just add that to the end here, and we'll set it equal to 4. We'll also set a maxlength and set that equal to 12. I know that's a little small, but we want to try it out, and we only want to have to put in 12 characters. So let's try this out. We'll go the Product List. We'll go to Edit, and we'll delete the product name, so we immediately see Product name is required. As soon as we start typing in characters, as soon as we hit the fourth character, the message goes away, because now there are more than four characters. If we go up to 12 characters, as soon as we hit the 13th character, we again get an error message. But the message is not right. It says Product name is required. How do we show different messages for different errors? When there are multiple validation rules defined for a single data entry element, we need to display an appropriate validation message based on the error. One really slick way to handle display of multiple validation messages is using the ng-messages directive. Ng-messages is a directive that shows or hides messages based on a key value pair collection. This directive is similar to a switch or case statement. Ng-message is a directive that defines the specific cases. So, the first message is displayed if the $error object contains a key of required with a value of true. The second message is displayed if the $error object contains a key of minlength with a value of true, and so on. Neat? Easy? But the bad news is that these directives are in Angular version 1.3. The sample application is currently on 1.2. So unless your project is using Angular 1.3, you need another solution. So what are our options for displaying multiple messages now? We can instead use the ng-show directive and access each $error object key directly. So if the required key has a value of true, this message is displayed. If the minlength key has a value of true, this message is displayed. And if the maxlength key has a value of true, then this message is displayed. Let's add this code to the sample application. Here we are back in WebStorm once again still with the ProductEditInfoView.html page open. Let's change the message text for the product name to display the appropriate message. So we're going to replace this span tag that we have with the one from the slides. If the error key value collection contains a key of required and the required rule is broken, this message is displayed. If the $key value collection contains a key of minlength, and that rule is broken, this message is displayed. Same with maxlength. Let's try this out. We'll go to Product List. We'll edit the hammer. We'll delete the contents of the element, and it says Product name is required. We put in the first character, and it tells us Product name must be at least 4 characters in length. So as soon as I typed in a character, it reevaluated the validation and displayed the appropriate message. Now as soon as I get to the fourth character, that message goes away. If I keep typing, as soon as I hit that 13th character, I get a new message that the Product name cannot exceed 12 characters in length. It works. But notice that I can still save. I can save this data even though it's not valid. That's not good. We need to prevent the submittal of the form if there are validation errors.

Preventing Form Submittal

If the form data is invalid, we don't want to submit that data to the server. There are several ways you can prevent form submittal if the form is invalid. One option is to disable the save button. Just add the ng-disabled directive to the button and set it to the forms $invalid property. If the form is invalid, the button is disabled. By disabling the button, the user cannot click on the button until the form data is valid. This is a common technique. But depending on your users, it could cause confusion. The user may not understand why the button is disabled or what they have to do to enable it. Instead of disabling the button, another option is to display a message when the button is clicked. This technique involves passing the form's valid state to the submit method. The submit method can then check the state of the form and display an alert message if the form is not valid. Let's try both techniques. Here we are back in WebStorm once again with the ProductEditInfoView.html page open. Let's scroll down to the submit button. Then let's add the ng-disabled directive to that button. So here we're specifying that that button should be disabled if the productForm state is invalid. Let's give it a try. We'll go to Product List. We'll go to Edit, and our save button is active. As soon as we clear out the field, we can see that it's disabled. We can no longer click on it. If we type in the first four letters, our message goes away, the field is valid, and our button is enabled again. So that technique works. It just makes the assumption that the user understands why the button is disabled. If you're using validation messages, as we've shown, it should be obvious why the save button is disabled. But depending upon your users and how you're validating the data, that may not be the case. Now let's remove the ng-disabled directive and change the ng-click directive to pass in whether the form is valid. So we're going to use the $valid state of the form itself. So productForm.$valid. Since the productEditTagsView uses the same submit method, we'll need to do the same thing for that form. So we'll go to productEditTagsView, we'll find its button, and we'll pass in productTagsForm.$valid, and we also need to modify the submit method in the controller. So we're going to go to the productEditCtrl, and now the submit function is taking in a parameter. We'll call it isValid. Then we'll modify the body of the function. If isValid, then we want to perform the save, else, we want to provide an alert. Let's give this a try. We'll do the same demonstration. We'll go to Product list, edit the hammer, and now when we clear out the field, we get the error, but the button isn't disabled. But if we do click the button, we get an error message, Please correct the validation errors first. So this technique works as well. Use whichever approach works best for your users and your application.

Summary

This module was all about validation. We first prepared the form for validation by ensuring that the form had a name and adding the novalidate attribute to ensure that the browser did not validate the form so that Angular could. We then identified the required fields and added the required attribute to the appropriate input elements. We then looked at ways to mark data entry elements on the form that are invalid. We used the ng-class directive, the Bootstrap has-error style class, and Angular validation states, such as $invalid, to set a style on any invalid input elements so the user can readily see which entries need corrections. But marking the data entry elements with a style was not enough. We also demonstrated how to display messages providing more detail on the issue. We used the $dirty validation state to ensure that we did not display messages prematurely. We then looked at the set of Angular validation attributes, such as ng-minlength and ng-maxlength, to provide additional validation. And we updated the message handling to display an appropriate message using $error. And all of this work would be for naught if we allowed the user to submit the form with validation errors. So, lastly, we looked at how to prevent submittal of the form until all of the validation errors are resolved. Here again are the tasks that we originally identified for building a line of business application. This module completed the data entry form's task by adding the validation. The next task is business logic. Most line of business applications require some amount of business logic. Let's look at how and where to add business logic to an Angular application.

Defining Business Logic in an Angular Service

Introduction

Most business applications require business logic. That logic may be in the form of rules, such as orders cannot be placed for items not in stock, or that logic may be calculations, such as computing the profit margin on each product. In either case, you need somewhere to define that logic, and that somewhere should allow you to reuse that logic across multiple views in the application. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this is module shows you how to build a custom Angular service for your business logic. The primary objectives of this module are to build an Angular service for business logic. In the sample application, the business logic will calculate the profit margin for each product. We'll want to use that service. In the sample application, the detail page will display the calculated margin. Then we'll want to reuse that service. The point of reusable business logic is to, well, reuse it, so we'll build a new view and reuse the business logic from this service in that view. Looking at the tasks we identified for building a line of business application, we have already covered many of the required tasks. As we laid out the views and the navigation, we created a product detail view. In this module, we'll update that view to display the profit margin calculated in our business logic. Previously, we created the data entry forms for the application. We created a form for basic product information and for product search tags. In this module, we'll define the data entry form for entry of the product cost and price. In that form, we'll reuse the profit margin calculation defined in our business logic. But first, we'll define that business logic in a custom Angular service. Let's get started.

Building a Custom Angular Service

An Angular service is code that can be used to organize application logic, thereby keeping your controllers clean and focused on defining the model and user interface operations. Services can be used to encapsulate logic that is not directly related to a view or a model. And services allow you to share logic or data across the application. Services can be injected into any controller or any other service, making it easy to use that logic anywhere it is needed. Services are lazily instantiated. That means that Angular won't instantiate a service unless a component depends on it. And services are singletons. That means that each component that depends on the service gets the same instance of the service. So you can keep data, such as look up tables, around for the lifetime of the application and share it across multiple controllers. Angular has a set of built-in services. But it also provides for the creation of custom services. Angular built-in services are prefixed with the $ sign. We have already covered some of these built-in services. The $http facilitates communication with remote web servers. We used $resource, which is a wrapper around $http, for interaction with RESTful back-end web services. And we used the $httpBackend to mock the back-end web service so we can run the application without having a web server in place. You can create your own custom Angular services for your business logic. There are several ways to create your own service. One approach is to use a service method. A service method defines the function that can be constructed. The most common approach for building a custom service is to use a factory method. A factory method defines a function that can be invoked. We'll use the factory method approach. Use the factory method to register the service. Register the service name and the service factory function. The object or function returned by the service factory can be injected into any component that specifies a dependency on the service. Here is the code required to register a service. This code is registering the service with the common.services module we created earlier in this course. It uses the factory method of that module to register the service. The first parameter to the factory method is the service name. We'll call it productService, and it will provide any product-related business logic. The second parameter is the service factory function. In this example, the function is referenced by its name. And here is the service factory function itself. The business logic code is defined within this function. Once the service is registered, Angular can reference it and inject it as a dependency as needed. Creating a custom Angular service requires the following steps. Create a new JavaScript file for the service. It's a good practice to put each service in its own .js file. Insert the script tag for the new JavaScript file into the index.html file. That will ensure it is downloaded to the client. Register the service with a module, as shown on the prior slide. We already created a common.services module for our common services. That is the module we will use for the registration. And the last step is to write the code for this service in the service factory function. Let's do these steps now.

Building a Custom Angular Service - Demo

Here we are back in WebStorm. We already created a common folder, and underneath that, a services folder. We'll put this new service there. Right-click, New, JavaScript file, and we'll call it productService.js. So we don't forget, let's add the script tag for this new JavaScript file now. We'll go to the index.html file, and we'll add it under the Services list. Now we'll go back to the service. As is now our pattern, let's start with the IIFE, and we'll add use strict. Recall that we already created a common services component by defining a common.services module. We'll register our new factory service with that module. So angular.module, and we'll register it with common.services. Now we can register the factory with that module, .factory, and the first parameter to the factory is the name of the service. We're going to call it productService. The second parameter is the function. We're just going to pass in a reference to the function, and we'll define the function separately down here. We are now ready to write the code for this service factory. Now we need to define what we want our service to do. What business logic does our application need? For the sample application, we want to calculate a margin percent. The code looks like this. So this function calculates a margin percent given the price and the cost. If both the price and cost are defined, the margin is calculated. The code subtracts the cost from the price and divides by the price to calculate the margin. We round the result and then return it. A service can provide more than one function, especially if those functions are related. So let's also calculate the margin amount here. That code looks like this. This function is calculateMarginAmount. It also takes a price and a cost. If both the price and cost are defined, we calculate the margin amount. We simply subtract the cost from the price to get that dollar margin amount, and then we return it. We also have a business requirement to calculate a price from a cost and a markup percentage. So we'll define a function for that, as well. This calculatePriceFromPercent function takes in the cost and a markup percentage. If the cost and markup percentage are both provided, this code calculates the markup amount based on the cost and markup percentage. It then adds the result to the provided cost. The price is rounded and then returned. Lastly, we want to calculate a price from a cost and a markup amount. This function is calculatePriceFromAmount. It takes in the cost and the markup amount. If both the cost and markup amount are provided, this code adds the markup amount to the provided cost. It rounds the resulting price and then returns it. We have defined four private functions in this service. To make these functions callable from outside the service, we can define a public API exposed by the service factory function. We do that on the return statement. This return statement returns the service object, which defines four functions. Note that the name of the function and the name of the public API do not have to be the same, like in these last two functions. To try out all of this code, we need to use this service. Let's do that next.

Using a Custom Angular Service

Now that we have built the custom service, let's use it. In the controller that wants to use a custom service, follow these steps. First, inject the service into the controller function. To inject the service, just pass it in as a parameter to the controller function that needs it. Since we have already registered the service with the application, Angular can locate this service and pass in the singleton referenced to that service. You can then use that reference to call the service methods or access its properties, like this. Easy as that. You can use these same steps to access the custom service in another service as well. Here is our current product detail view. Notice that there is a place for a margin percentage that does not display a value. We'll want to use the custom service we created earlier, the product resource service, to calculate that percentage. So in the productDetailCtrl, we can add the code to call our custom service. Let's do that now. Here we are back in WebStorm with the productService.js file open. This is the service we created earlier. We want to use this service in the productDetailCtrl. The first step for using a custom service is to inject the service into the controller function. So we will pass the service in as a parameter. We'll just put it here on the end. Because we have defined a Min-Safe array containing each of the function parameter names, we need to add this parameter to that array as well. So that's up here. So that's it. We can now use this custom service in our controller. Let's look at the productDetailView.html file. Here, we are currently binding to a margin percent variable. We'll need to assign that variable in the controller. So going back to the controller, we want to define that variable on the model. So we'll just put it here. One of the functions in the custom service calculates the margin percent based on the product price and cost. Let's call that function here, so we're going to use the productService reference. This instance is injected into the function by Angular, and we're going to call the calculateMarginPercent. So that shows up on our IntelliSense here. We're going to pass into this function the price and the cost as shown in our tooltip. So the price is vm.product.price, and the cost is vm.product.cost. I'm going to split this onto multiple lines so it's a little easier to see. So all we had to do was pass the service into the function, and then we can call any of the functions from that service. Let's give it a try. So we'll go to Product List. We'll click on one of the products, and that brings up our detail page. And notice our margin percent is now appearing. Let's go back and pick a different one. And its margin percent is appearing. Cool. But where are the cost and price entered? If we go to edit, we can see that the price details tab is still empty. So let's finish the price details next.

Reusing Business Logic

Let's look at what we just did with a summary diagram. We built a custom service called productService. And in it, we defined several functions that express the business logic for the application. We injected that service into the ProductDetailCtrl by passing the service into the controller function. We then called one of the methods in that service to calculate the margin percent and assigned the result to a variable on the model. The binding in the form then displayed the margin percent on the detail page. Now we want to reuse the logic in the product service to build the price details tab of the edit feature. This feature will ask for the cost and allow the user to calculate an appropriate price based on a desired markup percentage or a specific markup dollar amount, or the user can simply type in a price. We'll again display the margin at the bottom for reference. Let's add this to our sample application. Here we are back in WebStorm with the productEditPriceView.html file open. We had previously created the skeleton for this view when we created the route states. Let's replace the simple text with the HTML to build the view as we saw in the prior slide. I'm going to paste in all of the code for this page, and then we'll talk through it. We are creating a new form. We are using Bootstrap to layout that form horizontally for a consistent look with the other data entry forms. We give the form a name so we can implement validation. And we specify novalidate so that the browser won't validate the form allowing Angular to perform the validation. Each data entry element is defined as a form group with an ng-class for validation, as we discussed in the validation module earlier in this course. The form group has a label, an input box, and a span to display the validation messages. The second form group displays two buttons so the user can select to calculate the price based on a markup percentage or a markup amount. These buttons use the btn-radio directive provided by UI-Bootstrap. UI-Bootstrap is the component we used previously for the datepicker. It provides radio buttons that look more like sticky normal buttons. The ng-model directive defines another variable on the model called priceOption. If the user picks the markup percent button, the priceOption variable is set to percent. If the user picks the markup dollar button, the priceOption variable is set to amount. The code will use these values to change the display in the view. The next group uses the ng-if directive, so it is only displayed if the percent radio button was selected, which set the priceOption equal to percent. This group provides for entry of the markup percentage. The next group is similar. It is only displayed if the amount radio button was selected, and the priceOption variable was set to amount. Each of these two groups has a calculate button. The calculate button has an ng-click directive, which calls the calculatePrice method on the model. We have not yet created that method. The next form group is for the price. We can either populate this price with the calculated value, or the user can enter the price. There is also a row for the calculated margin percent. This uses an Angular filter to format the percentage so that it is displayed with no decimal places. Notice that the binding expression here is calling a function. That is needed because the margin percent must recalculate every time the user updates the cost or price. This last group contains the same two buttons as we defined on the other tabs. There's a submit button, which calls the submit function, and a cancel button, which calls the cancel function. Let's run the application and give this a try. So we'll go to Product List. We'll pick garden cart, and we'll pick the Price Details tab. Okay, the display doesn't originally look very nice, but if we click Markup %, we can see that it looks much nicer. When we click Markup %, an additional set of controls are displayed. We can then enter a percentage, say we want to mark up the cost by 10 percent, and click the Calculate button to recalculate the price. But notice the Calculate button isn't currently working. That's because we have not yet written any code for this button. We can also click Markup $. When we click that, the screen changes and allows us to enter a markup amount instead. So we might want to mark it up by $5. But, again, the Calculate button doesn't work. Also notice that the margin is not displayed. So there are several things we need to do here. First, we need to consider setting a default value for the radio buttons so that the initial screen comes out more clearly. We also need to define the code for the calculate button. And we need to ensure that the margin percent value is calculated. Let's do that next. As with the other edit tabs, there is no controller for the productEditPriceView. Rather, all of the edit pages share the productEditCtrl. The first thing we said we need to do was to set a default value for the radio button. So let's do that first. The variable used by the radio buttons was priceOption, and we're going to set that to an initial value of percent. When the screen first comes up, the user can enter the margin percentage to calculate the price from the cost. Next, we want to calculate the margin percentage. In order to do that, we need to inject the product service into the productEditCtrl, just as we did with the productDetailCtrl. So we'll put it here, and we need to add the parameter to the Min-Safe array. So we'll put it here. Now we can call the function, so we'll put it here, vm.marginPercent =. We could assign this variable directly to the service, but instead we want to make it a function. By making it a function, it will recalculate every time the price and the cost changes. So we simply want the function to return the productService.calculateMarginPercent, and we're going to pass in the price, vm.product.price, and we're going to pass in the cost. Next, we'll define the code for the Calculate button. We could define two separate functions, one for calculating with a markup percentage and one for calculating with a markup amount, or we can combine the logic into one function. This function calculates the price based on a markup. It checks the value of the price option variable and performs the appropriate calculation. Let's give it a try. We'll go to Product List. We'll edit the garden cart. Go to Price Details. Notice that the Markup % button is now toggled on by default. And our margin percent is displayed. The current price is $32.99. That's a margin of 39%. Let's enter a markup percent of 50 and click the Calculate button. The price is now 30, and the margin was recalculated to 33%. Since the submit and cancel buttons were already defined in the productEditCtrl, these buttons should just work. Let's try the save. The save was successful. We have now implemented a full-featured Price Details edit page for our application that uses the business logic we defined in the custom Angular service.

Summary

This module demonstrated how and where to define business logic in an Angular application. We built an Angular service for the business logic. In our sample application, the service calculated the margin percent, margin amount, and a price based on a markup. We then used that service to display the margin percent in the product details view, and reused that service in the pricing tab for the edit feature. The pricing tab provides a place for the user to enter the cost and enter or calculate the price. Build a customer service to define business logic or share data across all of the views in your application. In this module, we worked on several tasks. We added the margin percentage to complete the product details view layout. We added another data entry form to complete the pricing details tab of the edit view. And we created a custom Angular service to define the business logic for the application. Next up, data visualization.

Visualizing Data With Charts

Introduction

Data visualization can help users see patterns in an often overwhelming amount of data. One key technique for visualizing data is to display that data in a set of charts. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module shows you how to add charts to an Angular application. Let's say that the product manager's just requested another feature in our fictitious product management application. They need to analyze the profit margin on their products. We could give them a grid. How quickly could they evaluate profit margin performance from this grid? How about if we sort it? That's makes it a little easier. But what if there were hundreds of products? How about a chart instead? Now how quickly could they identify the highest-profit item? This first chart shows the cost, price, and profit margin amount for the highest margin products. The second chart shows the profit margin percentage for the highest-margin products. Charts make it easier to visualize and analyze application data. The primary objectives of this module are to download the charting component, define a new route state to a chart page, define a new menu option that activates that route state, build the view to display the charts, and build the controller to configure and define the data for the charts. When we are finished, we'll have a new menu option that the user can select to display a set of charts. Looking at the tasks we identified for building a line of business application, we are going to go back and add an additional view for the charts and the routing necessary to navigate to that view. We are going to use the data access and business logic that we already have in place, and add the charts for data visualization. Let's get started.

Setting Up the Charting Component

Charting is a common requirement. As such, there are JavaScript libraries available for building charts. D3.js is one such library. D3 is a JavaScript library for manipulating documents based on data. It provides powerful data visualization components and is commonly used to build charts. With all of its power and flexibility comes a steep learning curve. Luckily, you won't have to learn d3 to build charts with Angular. Angular-charts is a wrapper around basic d3.js features for use in Angular. It provides Angular directives for commonly used d3 charts and supports pie, bar, line, point, and area charts. In the controller, you simply configure the chart defining the labels, title, legend, and so on. You define the series, which identifies the meaning of the data, and of course you set the data for the chart. Setting up the charting component is a process that should be somewhat familiar at this point. First, you download the component. In this case, you need to download Angular-charts from this website. The Angular-charts component has a dependency on d3.js, so you need to download that as well from this website. You add the downloaded .js files to the .js directory of the project. Add references to these new .js files in index.html. And add a module dependency on angularCharts. Angular-charts can be downloaded from this website. Just click the GitHub link. That takes you to GitHub where you can download a zip file containing the JavaScript file. As shown here, Angular-charts has a dependency on Angular, which we, of course, already have, and it has a dependency on d3.js. Click the d3 link here. Or you can directly navigate to the d3.js site. D3.js can be downloaded from here. Just click the d3.zip link here at the bottom. Now let's set up the Angular-charts component in our sample application. We are back in WebStorm. I have already downloaded the two components and added them to the .js folder of the project. We then need to add references to these .js files in index.html. We'll put them here under Library Scripts. Next, we need to add a dependency on Angular-charts to app.js, so we'll add it here. That's it. That's all we need to do. But before we can see the chart in action, we need to build the charting feature. Let's start that by defining a new route state.

Defining the Route State

These are the route states we have already defined for the sample project. There is a productList state. That is used by the product list menu option to display the list of products. There is a productEdit state, which is used by the add product menu option to display the edit page and by the product list to edit a product from the list. This state has nested states, which are now shown here. There is a productDetail state, which is used by the product list to view the details for a specific product. We want to add a new route state that displays the price charts. We'll call it priceAnalytics. Let's add that state now. We are back in WebStorm. Recall that the navigation states are defined in the app.js file. We'll add the new state to the bottom of the list of states. So .state. We'll call this new state priceAnalytics. The URL will be /priceAnalytics. We have not yet created the template for the view, but when we do, we'll put it under the prices sub-directory so we can define the expected template URL is app/prices/priceAnalyticsView.html. We'll define a controller. The controller will configure the chart, define the series, and set the chart data. We have not yet created this controller, but when we do, we will call it PriceAnalyticsCtrl. Note that we won't use the controller as syntax for this option. I was not able to get the chart to work unless I explicitly injected in $scope, so we will use the classic controller syntax instead for this state. Lastly, we want to define a resolve object. The resolve object will ensure that all the data is loaded before bringing up the page. We'll use the productResource to obtain the data. We'll call the query function and return a promise. This is similar to the resolve used for the product detail and product edit features. But in this case, we don't need a parameter since we want to retrieve all product data. And we're using query instead of get to get the products in an array. Now that we have the route state, let's define the menu option to activate this state.

Defining the Menu Option

We want to add a menu option so the user can quickly display the charts. The new menu option will look like this. Let's add that option now. We are back in WebStorm once again. To display the menu on every page of the application, we put the menu in index.html. We can add a new menu option by simply adding another list item with an anchor tag. The anchor tag includes the ui-sref directive to activate the defined navigation state. At this point, we can run the application and ensure the menu option is appearing correctly. Here's our new menu option. And if we make the browser smaller, emulating a phone or smaller device, the new option then appears in the dropdown list. Clicking the option doesn't display anything. That's because we don't yet have the view template. Let's create that next.

Building the Chart View

Let's add two charts to the chart view template as shown here. The first chart shows the cost, price, and margin amount for the top products. The second chart shows the margin percent for the top products. You may be surprised to see how little code is required for this view. This is the code required to display a chart using the Angular-charts component. In the view template, add a div tag where each chart should appear. Use the ac-chart directive to identify the type of chart. You can specify a pie, bar, line, point, or area chart. Use the ac-data directive to assign the data for the chart. The dataAmount variable in this example is assigned to the scope in the controller, which we will create shortly. The dataAmount variable must include both the data and the chart series information. Use the ac-config directive to assign the configuration data for the chart. Configuration data includes the chart parameters such as the title and legend information. And we are assigning a CSS style class that we will create. Let's use these directives to add two charts to the chart view template. Here we are back in WebStorm. The priceAnalytics option is a price feature, so let's navigate to app, prices. Then we'll add a new template and call it priceAnalyticsView.html just as we defined in the route state. Right-click, New, HTML file, priceAnalyticsView.html. We don't need any of that default HTML. Since these few templates are inserted into the main layout template, they don't need the HTML head or body tags. So let's just delete them. Then we'll define the contents of this view. I'll paste it in, then we can discuss it. The panel heading is the same as all the other pages for consistency. It simply displays the page title. The panel body contains the two bar charts. The first bar chart is for the cost, price, and profit margin amounts. So the $scope variables, dataAmount, and configAmount are suffixed with amount. That will make it easier to keep track of the variables in the controller. The second chart is for the profit margin percentage, so the $scope variables are suffixed with percent. The only other thing we need is the style class. Notice that we defined a style class on the charts called chart. We need to create that style class. We'll add it to the app.css file. So we'll go under CSS, app.css, and we'll add the chart. Note that the chart will not display unless a height and width are set for the chart. Next, we need to build the controller and set the $scope variables we bound here.

Building the Chart Controller

The controller is where you define the data, series, and configuration for each chart. This requires the following steps. Retrieve the data. This step includes retrieving the appropriate data from the web server and adding any computed fields, such as the calculated margin amount and percentage. Filter the data. In many cases, all of the data cannot be displayed on the chart. The chart may be limited by an amount, such as the top ten products, or by dates, such as the products ordered this month. Format the data for the chart. The chart requires that the data be provided in a very specific format. So some data manipulation is required. Define the variable for the chart's ac-data directive, which includes both the series information and the actual chart data. And, finally, configure the chart using the variable assigned to the chart's ac-config directive. If you have the option, you may want to consider calculating the computed fields, filtering the data, and potentially even formatting that data in the web service on the back-end web server. That would minimize the amount of data retrieved from the server and the amount of processing required on the client. But we're going to perform all of these operations on the client in the sample application. Let's add the code for each of these steps to the controller one by one. We'll do the first two in this clip and the last three in the next clip. Here we are back in WebStorm. Let's create a new JavaScript file for the price analytics controller. We can navigate to app, prices, right-click, New, JavaScript file, and we'll call it priceAnalyticsCtrl. So we don't forget, let's add the script tag for this new JavaScript file to index.html now. Since this is a price controller, we'll give it a separate comment. Going back to the controller, we'll use the same pattern we have been using for controllers. It begins with the immediately-invoked function expression, or IIFE, and it includes use strict. We'll start by looking up the module. Since this is a part of our main application, we will register it with the productManagement module, then we can register the controller. The first parameter to the registration is the controller name. We're going to call it PriceAnalyticsCtrl. The second parameter of the controller method is a Min-Safe array. The array contains each parameter injected into the controller function. The charting directives don't seem to work well with the controller as syntax, so we'll use the classic controller syntax and explicitly inject $scope. We'll also want to inject products as defined in the resolve block of the navigation state configuration. Let's look at that again. Here is our navigation state. We have a resolve, and the resolve is actually retrieving our products using productResource.query. We can then inject products into the PriceAnalyticsCtrl. So we'll go back to the controller, and we can inject products. The last element of the array is the reference to the controller function. We can then define the controller function itself. And we're passing in $scope and products. The first line of code in this controller sets the title of the page, and we'll call it Price Analytics. The data for the first chart includes the cost, price, and margin amount. Cost and price are properties provided in the JSON for each product. Margin amount is a calculated property and is not provided in the JSON. The data for the second chart includes the margin percentage. The margin percentage is also a calculated property and is not provided in the JSON. So the next set of code in the controller must calculate the margin amount and margin percentage. To perform those calculations, we'll want to call the product service we created in the last module. To use the product service, we need to inject it into the controller. So we'll inject it here, and we need to add it to the Min-Safe array. Now that we have the product service injected in, we can loop through the list of products and calculate the margin amount and margin percentage for each product. So this code calculates the margin percent using the price and the cost. And this code calculates the margin amount, also using the price and the cost. Now the margin percentage and margin amount are defined on the model for each product. The next step is to filter the data. Let's do the filtering for the first chart first. The fictitious requirements identified that the top five products should appear in the chart. So we need to do three things here. First, we need to inject $filter into the controller function. $filter is a built-in Angular service for filtering data. We'll use the service to order and limit our list of products. So we'll add $filter here and in the Min-Safe array. Then we can use $filter to order the products by margin amount. So we use the orderBy filter, take our list of products, and order it by the margin amount. Then we want to limit the display to five products. Note that we could pick any number here, we just picked five. So we use $filter again, use the limitTo filter, take our orderedProductsAmount, and limit it to five products. The code to filter the data for the second chart is similar. We use the orderBy filter to order by the margin percent, and then use the limitTo to limit it to five products. The next step is to format the data for the chart and define the chart variables.

Setting Up the Chart Data

The Angular-chart library requires that the data be provided to the chart in a very specific format. So some data manipulation is required. The chart data must be defined as an array of XY coordinate properties. So you have to transform the data to the chart's desired format. This example creates an array of chart data for the amounts chart. It loops through each product in the list of products. It adds the product name to the X property. This is for the X, or horizontal, axis. The Y property is defined with an array containing the cost, price, and margin amount. This defines the Y, or vertical, axis amounts. Let's add the data manipulation for both charts now. We are back in WebStorm with the PriceAnalyticsCtrl open. We'll paste in the code from the slide. This code loops through the filtered products and populates an array containing the X and Y coordinates. The X coordinate is the product name. The Y coordinate is an array containing each series, cost, price, margin amount. Next, we can define the data amount variable used by the ac-data directive in the view. We'll add that variable to the scope. This variable contains an object with two properties. The series property is an array containing the name of each data series. Data is the data object containing the X and Y coordinates. The code for the second chart is similar. The second chart only contains one set of data so has only one series defined. The last thing we need to do is add the configuration information for each chart. Let's do the first chart first. The website for the Angular chart provides the list of configuration options. This code specifies a title, turns on tooltips, turns off labels, does not define any event functions, and specifies a legend. This information is added to the configAmount variable on $scope. The second chart has a similar configuration. The only thing that's different here is the title. Now we have everything that we need here in the controller. Let's give it a try. I'll run the application. Select price Analytics. Voila! The chart even has some cool animation. Let's refresh and watch that again. Cool. Now management can easily see that they need to sell more of those video game controllers. Charting is relatively easy with these libraries. Use charts to help management visualize the data in your application.

Summary

This module was all about data visualization through a set of charts. We downloaded both the Angular-charts component and the library it depends upon, d3.js, and added Angular-charts as a dependency in the application. We then added a new route state called price analytics and defined a new menu option that activated the new route state. That allows the user to quickly navigate to and display the charts. We then built the view using the ac-chart, ac-data, and ac-config directives. The last step was to build the associated controller. In the controller, we called our custom product service to calculate the computed fields. We filtered the data to the top five products and coerced the data into the format required for the chart. We defined the series information, then set up the chart configuration, including the chart title and legend information. The result was a great looking set of charts that management can use to evaluate the profit margin on the most profitable products. You can use the techniques presented here to build charts as required by your application. Here again are the tasks that we originally identified for building a line of business application. In this module, we updated the layout and navigation by adding a price analytics view, adding a routing state to display that view, and adding a menu option to activate that routing state. But the real focus in this module was on building charts for data visualization. We are finished with all of our basic tasks. But there are a few cross-cutting concerns that we have not yet covered, such as exception handling. Let's look at that next.

Exception Handling

Introduction

Wouldn't it be great if everything always just worked? That we never had a runtime error? That the server-side web services were always available and working? But we know that is not the case. Things go wrong. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and this module covers exception handling. The primary objectives of this module are to evaluate the options for exception handling in an Angular application, look at how to catch runtime errors with the try/catch block. But we'll see that using defensive development techniques often provides better results. We'll look at how to use the Angular global exception handler, and implement failure functions for asynchronous operations. By the end of this module, you'll have the basics of exception handling in an Angular application. You may have noticed that exception handling is not shown as a task here. That's because exception handling is part of every task. The business requirements task is a good place to consider the exception handling strategy as part of the design considerations. Are you going to display detailed exception information to the user? Log it on the client? Call a web service to log it on the server? Some combination of these? In each of these other tasks, exception handling should be implemented as the code is written. Think through the kinds of errors that each part of the code could generate and add defensive coding techniques or exception handling as needed. There are several options when implementing exception handling. The first line of defense is to prevent the error entirely. For example, a function retrieves a set of products and then processes each product in the set. If there are no products returned, the code generates an error. Instead of writing code to catch that error, you could instead add an If to ensure that the processing does not occur if no products are returned. We'll see an example of this shortly. Another option is to catch the exception. You can catch an exception around specific code or use Angular's global exception handler. We'll see an example of that as well. Depending on the type and severity of the exception, you may have to notify the user. For example, if the server-side web service is down, you'll want to notify the user so they know why the application is not working. Since this course is about building line of business applications, you can make some assumptions about your users. For example, you may be able to assume that your users have access to a Help Desk so you can display a message to the user requesting they contact the Help Desk for specific issues. And depending on the error, you may want to log information about the exception. This can help when working to resolve the issue. These options for exception handling are basically the same as for any other programming language. The only difference is in the how. The most common ways to catch exceptions in an Angular application are with a JavaScript try/catch block. You can add a try/catch block around code that could fail. That allows you to catch specific errors and handle them appropriately. You can use the Angular global exception handler. Angular has a built-in service called $exceptionHandler that catches exceptions. By decorating this service, you can tailor it to the needs of your application. When working with promises, you can also add failure or catch functions. Let's look at each of these techniques in detail.

try/catch Blocks

One option for catching errors is to use a JavaScript try/catch block. The idea is to try execution of a specific set of code. For example, here is an addTags function. The tags variable passed into this function could be undefined. If so, calling .split will generate a runtime error. So we can put this code into a try block. Then we add a catch block to catch any runtime exceptions that occur in the try block. If the code in the try clock generates a runtime error, the error is caught in the catch block. The catch block won't catch coding syntax errors such as missing parentheses. It only picks up runtime errors, such as an undefined variable. The catch block in this example displays an alert. The parameter to the catch block shown with e in this example contains information on the error. You can use that variable to include more detail and a message to the user or to log information on the error to a log. You can also define a finally block. The finally block is optional and often used for any clean up that must occur on success or failure. Oftentimes, however, good defensive coding techniques prevent the need for try/catch blocks. In this sample code, we simply check the variable tags and proceed only if the tags variable has a value. Defensive development techniques, such as this simple if/else structure, are often better than try/catch blocks. That's because the resulting code is more explicit, plus the try/catch block could potentially mask an unexpected exception. Any error within the try block will cause the catch block to execute. For example, in this sample code, the user could enter tags, but the vm variable could be undefined. The code will then catch that error and display Please enter tags to the user, basically hiding the real problem. So although Angular supports the standard JavaScript try/catch block structure, defensive development techniques, such as simple if checks, are often preferred. But unexpected errors may still occur. That's why Angular provides a global exception handler service. Let's look at that next.

Decorating $exceptionHandler

Angular provides a built-in global exception handling service called $exceptionHandler. Any uncaught exception is delegated to this service. This service only catches exceptions. It does not handle communication errors, such as 404 Not Found, or syntax errors. The default implementation logs the error to the browser console. You can decorate this service to customize its behavior. Say that we want to change the default global exception handler to display a message to the user. We can accomplish this by decorating the $exceptionHandler service. The code for that looks like this. This code configures the application with a decorator. This function needs the Angular build-in $provide service. The $provide service is used internally as part of Angular's component registration process. The decorator method of this service can intercept requests for a service in order to provide different or additional functionality. The decorator method takes two parameters. The first parameter is the string name of the service being decorated. In this case, $exceptionHandler. The second parameter is a Min-Safe array containing the name of each decorator function parameter and then the decorator function itself. The decorator function has a dependency on $delegate. So $delegate is listed in the Min-Safe array and passed as a parameter to the function. $delegate provides the original service to the method. That allows you to call the base implementation of that service. The return function has two parameters--the exception and the cause. In this example, the code in this function updates the exception message to a custom message. Then it uses the $delegate to call the base implementation of this service, passing in the exception and the cause. The base implementation simply logs the error to the console. Lastly, this code adds an alert to notify the user of the issue. That's the custom code we have here extending this functionality. This method then returns a decorated $exceptionHandler service object. You can add anything you want to this function. For example, you may want to add more extensive logging or a more detailed message. Let's add this code to our sample application. Here we are back in WebStorm. Before we add any code, let's look at what happens by default when an error occurs. In order to get an error to occur, we're going to need to force that error. So let's open the productEditCtrl, and let's remove $state. Looking down through the code, $state is used by the cancel button to cancel an edit operation. Let's see what happens now that $state is no longer defined. So we'll run the application, click on Product List to display the list of products, click Edit to edit one of them, and then click the Cancel button. Nothing happens. The user doesn't know why the cancel button is not doing anything. Is that how it's supposed to work? So let's open the Developer Tools. We can see in the Developer Tools that we are getting a message. The message is $state is not defined. So by looking at the Developer Tools, we know exactly what's wrong, but that didn't help the user very much. Since we have a Help Desk and want the users to contact the Help Desk if there is a problem, let's extend the default exception handling and display a message to the users if any exception occurs. Let's close the browser, go to app.js, and paste in the code from the slide. We already talked through this code when we were looking at the slides, so let's just rerun the application and see how it behaves differently. We again click on the Product List, click Edit to edit one of them, and click Cancel. Now the user gets a message, Please contact the Help Desk! Message $state is not defined. We could include the Help Desk phone number here or a link to email the Help Desk, or we could modify this code to automatically log error information. By decorating the $exceptionHandler service with an alert as we did here, we can ensure that the user is aware of any unhandled exceptions in the application. Before we leave this demo, let's be sure to put back the $state dependency. There we go. the Angular global exception handler will catch any unhandled exception, but it won't catch syntax errors, nor will it catch issues with asynchronous operations. We'll need to handle asynchronous operation exceptions differently. That's coming up next.

Failure Functions

A promise provides a way to handle asynchronous operations, such as retrieving or saving data to a back-end web service. And with the number of things that could go wrong when communicating with a web server, adding exception handling is important. Here's a simplified look at how an Angular promise works. A function in an Angular application launches an asynchronous operation. And that operation begins execution. The function immediately returns to the calling code. The return value is a promise. Optionally, you can write a callback function to receive status notifications. The async operation can send out notifications that are sent to the notification callback function. You can use this feature to provide a progress bar or status messages. The operation will eventually complete with a success or a failure. You can optionally write a callback function to process the return data or other response when the operation is successful. You can also write a callback function to process the response when the operation fails. Since we are talking about exception handling in this module, we will focus on the failure callback function. An Angular promise provides a then function that is executed when the promise is completed either with a resolve or a rejection. The then function takes three parameters. The first parameter is the success callback function. It is the function that executes if the operation completes successfully. The second parameter is the failure callback function. It is the function that executes if the operation completes with a failure. The third parameter is the function that executes when a notification is received. When working with promises, consider implementing the failure callback function. In this function, you could retry. You could notify the user of the problem. You could log the issue. You could navigate to an error page, or any other operation. What you do in this failure callback function will depend on the important of the operation to the application and the application requirements. In this sample application, we are using promises to communicate asynchronously with the back-end web service. We are using $resource, which we encapsulated in a factory service called productResource. This sample code is calling the query method of the $resource service to retrieve an array of products. The functions provided on $resource, such as get, query, and save, also allow definition of a success callback function and a failure callback function. Here is the same sample code but with the callback functions defined. Here is the success callback function, and here is the failure callback function. We can use the failure callback to handle issues with the web service. Let's try this out in the sample application. Here we are once again back in WebStorm. First, let's navigate to common, services, productResource.js. Here is the code we are using to communicate with the web server using the defined URL. Recall that we used the productResource in the configuration function for our route states. That's in app.js. Here's the last route state that we added for the charting. We used the productResource in the resolve feature of the navigation state to retrieve the data required for the page before navigating to the new page. This resolve calls the query method of productResource to retrieve an array of products. The ProductAnalyticsCtrol then uses the resulting products variable to sort and filter the data for the charts. We can add a function for success here. It goes in as the first parameter to the query method. We don't need any code in this response, but you could add code here to perform data manipulation or logging. The important part is the failure function. Here is the failure function. It checks the status on the response. If the status is 404 Not Found, it displays an alert box. The alert not only displays a message, but it provides some additional details on the URL that's causing the problem. If the status is not a 404, it just displays an alert box with a status text. So there is our failure function. To try the exception handler, we'll have to do something to cause a 404 error. Let's remove the mocking service. If you recall up here at the top, we're setting a dependency on the mocking service. Let's remove that. Since we don't have the required back-end service in place, the application will now generate an error when it attempts to get data. We'll then be able to see this error handling code in action. So let's look at the message one more time. It should say Error accessing resource and define specific information about the resource it's trying to obtain. So let's run the application. And we'll select Price Analytics since that's the one that we changed. And we get our message Error accessing resource. It defines the method we are using, which is a GET, and the URL, which is /API/products. That's the message from the alert box that we had added. When we click OK, notice that it does not navigate to the page. Instead, it cancels the change of state and leaves us where we are. Before we leave this demo, let's put back the mocking service. And let's run it again to make sure that it's still working. Price Analytics. And there it is. Now let's summarize the three exception handling techniques we looked at in this module.

Summary

This module provided a brief introduction to exception handling in Angular. We looked at the options for exception handling. We compared the try/catch block to defensive development techniques and found that often defensive development techniques, such as using If blocks, are more explicit. We looked at how to decorate the built-in Angular $exceptionHandler service to customize the global exception handling. But this handler only catches runtime errors, not syntax errors and not failures in asynchronous operations. So we also looked at how to define failure functions on promises. Use any combination of these techniques to catch and handle both expected and unexpected exceptions in your Angular applications. For information on more advanced exception handling techniques, check out the Pluralsight course entitled, AngularJS Patterns Clean Code. The next module provides some final thoughts, including information on unit testing and security.

Final Words

Introduction

As you have seen throughout this course, AngularJS is an excellent tool for building full-feature client-side business applications for the web. You can now leverage what you've learned to build well-crafted Angular applications. Welcome back to AngularJS Line of Business Applications from Pluralsight. My name is Deborah Kurata, and the final words in this course include a brief summary and a discussion of additional considerations including unit testing and security. The primary objectives of this module are to review the tasks for building a line of business application, provide an overview of unit testing, and discuss security considerations. Let's jump right into this short module.

Tasks for Building a Line of Business Application

At the beginning of this course, we identified a set of tasks for building a line of business application with Angular. We looked at the business requirements and turned those requirements into application features. We laid out views for the user interface of the application and used Bootstrap to give them some style. We used Angular's $stateProvider service to define routing states for navigation between the pages. We used Angular's $resource service to easily interact with the RESTful back-end web service. When a back-end web service is available, the application can use that service to retrieve and save the data for the application. To run the application without the back-end web service in place, we used the build-in Angular $httpBackend service to intercept the web service calls and provide mocked up responses. We build three data entry forms for entry of product information. We implemented validation and completed the submittal process by saving entered or updated product information. To keep the business logic separate from the controller code, we created a custom Angular service and implemented the business logic in that service. We then used that service in several places throughout the application. And we implemented a charting library to provide some basic data visualization. Lastly, we looked at implementing exception handling across all of these tasks. We've covered a lot of territory, but there are a few things that we have not covered. What about unit testing? Angular was designed from the ground up to support unit testing. But what is unit testing and how is it done? And what about security? Line of business applications often require a login process, and only specific users can access specific features of the application. Plus, there are the back-end services and data that need to be secure. Let's talk about unit testing first.

Unit Testing

Unit testing involves isolating a piece of code and validating its functionality by testing it independent from the rest of the application. So the idea behind unit testing is to break the application into small pieces or units and test them by writing code that validates the operation of each unit. You can think of each controller as a unit that can be unit tested. And you can test each service as an independent unit as well. Unit testing can promote clean and encapsulated code. Since unit testing tests individual units, developing unit tests helps you focus on building small encapsulated units of code. Unit testing reduces the bugs in the application. The purpose of a unit test is to ensure that the code works as required and produces the appropriate result. So the unit tests can reveal bugs early in the process. And unit tests can minimize debugging time. When a bug is found, you can write a unit test to reproduce the incorrect behavior making it easier to fix the bug without a huge amount of debugging time. Unit testing Angular code requires some additional tools. Jasmine is a commonly used unit testing framework. And Karma is a commonly used test runner. Both work well with WebStorm. If you are using Visual Studio, jasmine works there as well. When should you unit test? You can take a code first approach: Write the code, then write the test to test it. Or you can take a test first approach: Write the test to test it first, then write code to pass the test. In either case, each controller and each service should be unit tested as it is developed. This course skipped the unit testing of the controllers and the services. That's because unit testing Angular code is a detailed subject in and of itself, and there are already courses specifically focused on unit testing your Angular and JavaScript code. Here are two Pluralsight courses that focus on unit testing client-side JavaScript code. The first course is Angular focused. The second course covers testing client-side JavaScript. Check out these courses for details on unit testing your Angular and JavaScript code. Next up, security.

Security Considerations

When developing line of business applications, several security considerations come to mind. First, authentication or user login. Authentication is how the application ensures that the user is who they say they are. Most line of business applications are restricted to a specific set of users. Those users have login credentials that they must enter in order to access the application. We didn't specifically cover a login form in this course. But you can use the same techniques we used for the product form to build and validate a login form. Authorization is the process of restricting data and functionality to only those users authorized to access them. Data can be restricted on the server-side. Functionality, such as menu options or specific data fields, can be enabled or disabled on the client-side based on the users' role. Server-side web service security is another consideration. The server-side web service requires security to ensure that no other client-side code accesses the API and that the user accessing the web service is authenticated and authorized to use the service. Security is a large topic and includes many alternative approaches. It requires server-side features, and it is already covered in several other courses in the Pluralsight library. Here are several Pluralsight courses that provide information on security topics. First, there's Building End-to-End Multi-Client Service Oriented Applications - Angular Edition. This course shows you how to build a login form and details server-side security with Microsoft's Web API. There is Node Application Patterns. This course builds registration and authentication modules with Node. Node.js for .NET Developers includes a module on securing Node.js. Building AngularJS and Node.js Apps with the MEAN Stack includes modules on authentication, security and authorization, signup, and user profile. Consult any of these courses for information on authentication, authorization, and other security features for your line of business application.

Closing

This module provided some final words on building line of business applications in Angular. First, we reviewed the tasks required for building a line of business application. We discussed unit testing. We covered what it is, why you would want to do it, and provided some references to other Pluralsight courses. You can use these references for details on how to unit test your Angular controllers and services. And we discussed security considerations with references to other Pluralsight courses for details on how to implement security into your application. Congratulations! You have just completed the journey through building a line of business application with AngularJS. Thoughts or comments? Please use the discussion tab on the Pluralsight page for the course to leave your feedback. 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

(1273)

My rating

Duration5h 36m

Released26 Sep 2014

Bookmark

Add to playlist

Share course