An Overview

Introduction

Hi, this is Scott Allen, and this module is an introduction to my AngularJS Playbook. This course does assume that you have worked with Angular and you are comfortable with JavaScript, and you are looking for tips and strategies to handle more advanced scenarios with Angular. I'm going to cover some specific areas I've encountered over the last couple years building applications with Angular, and in this module I'll give you a preview of what's in this course, as well as just a few details on the server implementation.

The Big Overview

In the very next module of this course I want to show you some strategies for a common question that I receive about Angular, which is how to work with tokens, specifically JSON web tokens from an OAuth server. The scenario is that there is a resource on the server that requires authorization to reach, so when the Angular application detects that the server rejected a request with an unauthorized error, the application will then load a Login view, and if I login successfully, the server will return a token which the application will store, and then take the user back to the view they were trying to reach before that error and re-issue the request with the authorization token. All of this takes some coordination between a few different services and HTTP interceptors, and this module should give you some ideas on how to design the components and the routes needed to manage that. In the third module of this course I want to cover some diagnostics, which includes how to handle errors and rejected promises in an application. I cannot stress enough the importance of having some diagnostics and instrumentation in place for an Angular application, because when an application misbehaves in the client's browser, the situation can be difficult to diagnose. I'll show you how to write service decorators in all the right places to catch important errors, and how to write a custom directive to display these error messages or alerts, so the user knows something is wrong and their work is not being saved correctly. We will even see how to debug a page with data binding errors, as these could be difficult to track down even sitting at a developer machine. And one of the re-occurring themes in these first two modules, by the way, is how you can use services to manage your model objects. I've seen many projects struggle trying to keep their models in their controllers, but the sooner that you can make a mental breakthrough about having services manage models, the easier your architecture will be for certain scenarios. The fourth module in this course turns to UI related topics, and one of the first topics will be how to manage more complex UI with nested states, like this vertically tabbed UI where the user can navigate to different views of a dashboard. We will achieve this using a component named UI Router from the Angular UI project. We will also be looking at different strategies for building a custom directive, the directive for rating an employee, and how to take advantage of a modal dialog service from the Angular UI project. Not only how to use this modal service, but how to properly encapsulate the service to make it easier to use from a controller, a controller that we want to keep simple and easy to work with. The fifth module of this course looks at working with forms and performing validation. I want to show you the data structure

Angular builds, the one that will help you with the form, how to perform validation, write custom validators, write asynchronous validators, and how to reduce the amount of markup that is required to build a form. We'll do this by having a custom directive that dynamically adds HTML to a view, and in order to do that we'll need to learn how to use the $compile service in Angular, and compile dynamic markup. We'll also be using the new NG messages module to add validation messages, and along the way we'll see how to apply Bootstrap classes to add some color to the form. In this case maybe the form has too much color for your taste and it looks like traffic signals on a busy railroad, but you can adjust the styles to suit your needs. Then in the last module of this course I want to look at some integration scenarios, because quite commonly you'll need to use other JavaScript libraries in an Angular application, so I'll give you some general strategies to use for integrating with Angular. One library we will look at is PhysicsJS. This is physics simulator, and working physics into Angular will give us an opportunity to see how we can wrap a library with services and custom directives to make it appear Angular-like in nature. I'll also show you how to work with jQuery and a jQuery plugin, and how you can write a custom directive to wrap that plugin, and again, make it appear natural in an Angular application. If all of this sounds good, feel free to jump into the next module. Before I leave the introduction, I do want to provide a few details about the server implementation. It's not that Angular is tied to the server or to the technology used on the server, but I know a few people might want to build enough server implementation to follow along. Let's look at that next.

Some Notes on the Server

Angular is a technology that is, of course, completely agnostic to the technology being used on the server. That's one of the wonderful benefits of using HTTP messages in your architecture. You can have JavaScript talk to Python code on the server or Ruby code on the server, or C# or Java or JavaScript on the server. For building up the demos in this particular application, I did choose an ASP.NET server using Web API controllers, but if that's not your cup of tea, don't worry, for the most part all your server will need to do is serve static HTML files and static CSS files, mostly the Bootstrap CSS file, and of course we're going to have some JavaScript files. In fact, we'll have quite a few JavaScript files throughout the course. And there's only two sections of this entire course that rely on a dynamic response of any sort from the server. One is a custom username validation. This happens in the form section of the course, and I basically just need an HTTP endpoint that returns true or false. To do that, I'm using a Web API controller to return true if a username is greater than three characters, it does this after a 2 second delay. That allows me some time to demonstrate how to show an animation, but you can certainly build something similar with Java or Node or Django. What does require a bit more setup is the OAuth server. This is used in the next module to generate an authorization token that a user needs to see a secret recipe. In my ASP.NET project, I have a controller protected with the authorize attribute, and obviously if you're not using ASP.NET you'll need to dig into your platform of choice to figure out how to secure an endpoint and how to generate tokens or cookies if you want to authenticate a user. And then in the world of NodeJS, you can just npm install one of several packages that will give you most of the OAuth server functionality that you need. I've used node-oauth2-server myself. I can also give you a couple of tips for ASP.NET, specifically I have a blog post detailing the exact setup that I'm using, which mostly

consists of a few Owin components provided by Microsoft and installed as middleware in the application, as well as some custom classes to use not just any type of token, but a json-web-token, and all of this is detailed in the blog post. I would suggest if you go down this road in a real project, you should take a look at thinktecture's identity server. This is an open source project that provides a lot of flexibility and scalability for your auth needs. It will give you everything that the Microsoft components will give you, plus so many more things. The rest of the application then, it's all HTML and CSS and JavaScript. I'm not going to talk about server technology in the rest of the course, and I'm not going to talk about package managers or minification tools, or any of the thousand other topics in web development. We're going to focus on writing straight-up Angular code and look at some strategies, advice, and implementation tips. So let's go to the next module and get started.

Security Playbook

Introduction

Hi, this is Scott Allen, and in this module we'll look at some common security scenarios you'll encounter when building Angular applications. I want to show you how to work with access tokens to authenticate a user, and how to react to an HTTP status code 401 response from a web server. That's a response telling you that the user needs to authenticate herself. We'll also look at how Angular prevents cross-site scripting attacks, and how you can work with Angular to bind HTML into a browser. Let's start by talking about tokens for authentication.

Cookies Versus Tokens

Historically we have used HTTP cookies in our authentication schemes for users on a web application, but in a world of Web APIs where we transfer data using JSON over HTTP, we prefer tokens over cookies. Cookies present problems today, because the browser sends a cookie on every request, even on requests that don't need a cookie to authenticate the user, and sometimes on requests that have been put together as part of a malicious cross-site request forgery, a CSRF. We have to guard against those when using cookies. Also, cookies don't work very well when we need to call Web APIs that are on a different domain because cookies are limited to a specific domain. Tokens, on the other hand, solve many of the problems that cookies present, but we do have to write some code on the client to obtain tokens, to save tokens, and to send tokens along in the proper requests. So we have more control over tokens, and tokens can cross domains, but more control also means we have more responsibility and we have to write a little more code. Let's take a look at how to communicate with a server and obtain an API access token using Angular.

Architectural Overview

There's a number of different pieces to keep track of in the code I expect to write for this demonstration, so first let me give you a high-level architectural overview of the different components, and how I would break this down. I'm running a web server, and this web server exposes a few endpoints that I can reach from JavaScript, including one endpoint at /API/secretrecipe. If I invoke this endpoint, it will give me back the ingredients for a recipe in JSON format. I can make an HTTP request to this endpoint from Angular to get the data that I need to display on a secretrecipe view, but of course the recipe is not open to the public, it's a secret, so first I'll have to obtain an access token and authenticate the user. For authentication, the server also exposes a /login endpoint where I can post a username and password and receive back a JSON web token if the credentials are correct. This endpoint, by the way, it follows the OAuth 2 specification. That's an open standard, and there are many libraries available for all different types of server frameworks that allow you to plug in an OAuth 2 provider for your NodeJS server or your ASP.NET server, just about anything. Now on the client I expect to separate responsibilities for getting the secret recipe into a number of different components. I'm not going to show all of them here, but these are the major players. First I want an Angular service named oauth. This service will be dedicated and solely responsible for communicating with the login endpoint, so it will be responsible for correctly formatting the users credentials into an HTTP request that an oauth 2 endpoint can understand. That endpoint will send back a token when the user successfully authenticates, and when the oauth service receives the token, it will give that token to another service that I'll call currentUser. It's the responsibility of the currentUser service to know if the user is anonymous or if they're logged in, and once they have an access token, to keep track of the user's access token. A third service, addToken, is going to be a service that knows how to take the currentUser's access token and apply that token to an outgoing HTTP request so that we can reach the secretrecipe endpoint. So lots of small, very focused pieces. Let's build them up and see how they're all going to work together.

Get a Token

Here is the application up and running, and I have some basic functionality in place, although most of the security-related code is not filled out as yet, that's what we're going to do. You can see I have basically a blank template in the code behind the browser, but the opening page of the application has a button here where I can click to see the secret recipe. Unfortunately, when Angular makes that request the response will always be 401 (Unauthorized) because we need to get the user an access token before Angular can get the data for the secret recipe and put that into a model so we can display it in this view. I already have the Login form put together, that's in a template that forms part of my navigation. This template puts the login controller in charge of that section of the UI. Hopefully what I want to achieve eventually is that when the user is logged in, the login controller will display Hello username. Otherwise, when the user is still anonymous it doesn't have an access token, we'll display this login form. When the user clicks the submit button on this login form, we will call a login method that is defined inside of this controller, and all this method is really going to do is take the username and password that were entered into the form, and pass them along to this oauth service, which is not implemented as yet. Currently it just returns an empty object. So if we are given a username and a password to a function

called login, what would that function look like? Login would be a function that takes the username, that takes the password, and ultimately what I have to do is do an http.post to the /login url. We could make that a configurable parameter for this service, but I will just leave it hard-coded for now. And when we do this post we're going to have to pass along some data which will include the username and password, so I need an object that has the username property and a password property. Those are initialized to the incoming parameter values. And then because this is oauth 2, I need to specify a grant_type. Now I don't want to drill into all the intricate details of the oauth 2 specification, but I will tell you there's several different grant_types available, and the grant_type that we want for this scenario, where I'm passing a username and password, I want a grant_type of "password". In all the oauth 2 servers everywhere, if the username and password are correct and I put that data correctly into a post, then the server should respond with an access token for me. However, there's one additional little step that I need to take to put this data into the correct format. The oauth 2 specification wants this data to arrive form encoded. What Angular will do by default, if I just pass it an object here as the data to send along on the post, is it will JSON serialize that object, and that's not going to work for my /login endpoint. I need to formEncode it. Fortunately, it's fairly easy to write the service that will take an object and put it into formEncoded data. I could do that just by walking through each property that's available in the object and building a string of name = values, so name value pairs, all properly encoded, and this service, which is just a function, should be able to take my object and put it in the correct format. All I need to do is inject that service, of course I'll also need to inject http, so inject http, inject formEncode, and then formEncode this data. And I also just need to be very explicit and tell Angular in the configuration parameter for this http post, that I need to set up the headers to include a header that specifies the content type as formEncoded. So that is "application/x-www-form-urlencoded", and I pass along config to my http post, and I might have something that gives me back an access token. So let's flip back over into the application and test this out. What I can do is watch the network tab, do a hard refresh, I can fill in a username, and a password, and click Login, and in the Network tab of the developer tools I can see that I did a post to login and that it returned a 200 successful. That means I should have in my response, an access token. So I sent the correct headers, I sent the correct formEncoded data, the username and that password matched what the server expected, so I now I have an access token. Now what I need to do in the next clip is save this access token, which is a long encrypted value, and I need to send it along on the request as an http header so that I can get to the secret recipe, get that data, put it into a model, and display it on this view.

Save a Token

Now that we have a token, I need to store that token somewhere so that subsequent http requests that go out after the user has logged in, will be able to send that token along in a header. I would like to do that with another service, a service I'm going to call currentUser. I have a template set up here that will go ahead and register currentUser as a service for Angular, and currently that service is just an empty object, I'll need to add a few things here. Now one thing I would like to add is some sort of member to the service that will hold the state of my user. You can think of it as a model that is stored in a service, a model that represents the currentUser. And by storing that in a service, which is a singleton, I'll be able

to use that for different places, for example, over in the login controller I already have this set up to inject the currentUser service and assign a property on that service called profile, to a user property that I expose for my model that I bind to the view. That allows me to do things inside of a view, like check to see if the user is logged in. So this means my currentUser will at least need a profile member, and I think what I can do is have that member point to an object that I'll define in my service, and I'll give it the name profile here also. Profile can be an object that starts off with a username property, we're not going to know the name of the user when they first come to the application, so we'll initialize that as empty. This can also be where we store the token for that user. We'll also start that off as empty. And maybe I'll give it a simple property here, get loggedIn, so that I can have easy expression from the view like user.logged in, which returns true or false, and essentially all I need to do here is just look to see if a token is present. So if the token is truthy, then I know the user is logged in. And let me also give the service a method that I can invoke, let's call it setProfile, that will allow me to set the username and token. So setProfile is a function that when you invoke it you'll pass in the username and the access token for this user, and I'll just say profile.username = that incoming username. And just make sure I have this spelled correctly, profile.token = that incoming token. The next job then would be for my authorization service, once we've received a response from our oauth service, we take the username that the user has given us and the token that has come back from the server, and we hand that off to the currentUser service. So I will also inject currentUser, and now we need to do some processing on this post request. So that returns a promise, and I'll call then, and say let's execute this function on a successful post. I can say currentUser, let's invoke that setProfile method. I'm going to pass in the username that was given to me, and of course what will come into this then method is the object representing the http response from the server. To get to the token that is inside of there I just need to go to response.data.access_token. It will be just that easy because of the deserialization that Angular will automatically apply to that response, and then I could return some arbitrary value here. Let's just return username. But in very few scenarios will the caller actually need a return value from the promise, because most of the work will be taken care of by this oauth service and the currentUser service. We'll grab the token and we'll store. The caller just wants to know if this was all successful or not. And one way I could start to see if this was successful or not, is let me just refresh the application and try to login one more time, and what I'm hoping to see is that the login form will disappear and it's replaced with hello sallen! Now I'm pretty sure that I have correctly stored the token somewhere where I can access it. Now I just need to make sure that that token goes out so that when I try to view the secret recipe I actually get some data back. We'll do that next.

Send a Token

Now we need to add our token to the http request for the secret recipe, and I'm going to do this using an http interceptor. In Angular, http interceptors sit in a message-processing pipeline, and Angular will invoke methods on each interceptor for every outgoing request, for every requestError, for every incoming response, and for every responseError. What I really need is just an interceptor that hooks into each request so that I can take the currentUser's access token and add that as a header. So tokens are much like cookies in that they go along in an http header, but we need to explicitly add that as a header

to every outgoing request, because the browser is not going to do that for me. And I will place this interceptor into a file that I already have set up called addToken.js. Ultimately what this has to do is be factory for a service that has a single method attached to it, one called request, because that will get called for every outgoing request. So if I put a response method there, the response method would get called for every incoming response, but I'm just interested in the request. So request is a function that will take the configuration object associated with this http request. So that's the object that will hold the url that we're going to, the headers, the timeout, the data. All I'm really interested in is modifying the headers. I could look at the url of this request and determine whether or not I need to add a token, perhaps everything going to /API/something will need a header, but other requests don't. Just to keep things simple, though, I'm going to add the authorization token to every request. If the user is logged in, and I can tell if the user is logged in or not because our currentUser service has a property available off the profile member that we created, and it's called loggedIn. And so if the user is logged in, I want to go to config.headers and create an Authorization header. The Authorization header has to come in a specific format. In this case it needs to be the word "Bearer" followed by the token itself. So my web server will be set up to understand "Bearer" tokens, and that will be quite common whenever you're using tokens. I just need to grab the currentUser, go to the profile property, and pull off the tokens. That will create that http header for me, it will say authorization:bearer, space, and my token, and that will go into every request once the user has logged in. And then every good http interceptor, or at least the interceptors that are intercepting every request, should return the config object that is passed in, but wrapped into a promise. So I will also inject the q service and use q.when to create a promise that will just immediately resolve with that value of config. And that should be all the code that I need to add the token to every request. Now I just need to set things up so that this is a proper interceptor. First I'll register my factory method for this service. The factory method for addToken, _____Angular please call this function, which is addToken. And then my module will need a little bit of configuration, so module.config. What we need to do is write a configuration function that takes $httpProvider, because it is the httpProvider where I can register interceptor, so httpProvider.interceptors, so yes you can have multiple interceptors. I'm going to push the value of the string value "addToken" into that array and now when Angular sees that at runtime it will instantiate the service, plug it in as an interceptor, and hopefully be calling my request method for every http request. Let's try it out. Back in the application, let me come to the Home view and do a hard refresh. Let me try to log in as sallen, and now let's go to the Network tab and let me click, See today's secret recipe. And it looks like today's secret recipe is French Toast, wonderful. I can see that request went out down here. It was a 200 response. I can see the data in the response, and in the headers I can see that my access token made it into the http headers thanks to this addToken service. So quite a few components floating around, but ultimately we wanted to divide up responsibility so that we had one service responsible for taking our username and password, formatting it, and posting it to an oauth server, grabbing the response token and giving that to a different service that represents my current logged in user as sort of a global singleton, and then a third service which is registered as an http interceptor that will take that token and add it to every request so I can get to things like the secret recipe, which require an authenticated user.

Logins and Redirects

When an anonymous user tries to access a protected resource, a traditional web application will typically redirect the user to a login page, and once the user has established his credentials, the application redirects that user back to the page he originally requested. Can we do something similar in an Angular application or at least redirect between views in an angular application? Well yes we can. In fact, Angular gives us more user experience options than a complete redirect to a different page. We could switch between views or we could display a modal dialog box, something like the Bootstrap modal widget, but what I'd like to show you is something perhaps a little bit trickier, and that's redirecting the user to a Login view when the server rejects a request, and then once the user logs in successfully, we'll redirect her back to the original view. Here in the application, if I am a user without an access token and I try to see today's secret recipe, I'm just left on a blank page. What I'd like to do is have the application respond to this 401 and direct the user to another view that will make it very clear that they need to log in. There's all sorts of options about what to do here. I could provide some sort of animation around the existing Login form that's in the top of the navigation bar, but I want to show you something a little bit different, which is redirect to another route that is still inside of this application, and I want to do that with another http interceptor, because an interceptor just like the addToken interceptor that we built, can listen for every outgoing request, but more importantly for this scenario, it can listen for every incoming response and specifically what we want to do is catch any responseError. So let me create a function, responseError, and that will become a part of this object which is the loginRedirect service. You can see I already have the infrastructure set up here to register this as a service, and also to configure the service as an http interceptor. So what does responseError look like? That is going to be a function that gets the http response, and as part of the response I can look and see if the status code for that response was a 401. If that's the case, that's where I want to jump to another route, otherwise, what do I do, or really ultimately what do I always want to do with this response? Well, I want to inject the q service and create a promise that will be rejected and deliver this response to anyone else who is listening for this rejection. I want to do that in all cases when there's a responseError, but specifically what do I want to do when there's a 401? This is where I want to jump to another route, which means I can inject the location service and tell the location service I want to go to "/login". So, again, that is another route that is inside of the same application, and just by having this little bit of code in here we should be able to see that view. So let me come out and refresh the application, go to See today's secret recipe, and I'm redirected to /login, here's the login form. Now ideally, once I fill out my username and password and click login, I will then jump back to the area I was trying to get to, which was that view that will show the secret recipe, and that means we'll need to add a few more capabilities into the service. Let's keep track of the last path, in fact, we can just default this to the root of the application, and anytime there is a 401, before we set the path we can remember what that last path was, just by asking location for the current path. And then once a user has successfully logged in, we need some ability to go to that last path. Now that could be another service, perhaps part of an authorization service, or I could just expose last path as a member from the service so anyone can look at what the last path was, but let's actually create another public member for this service, redirectPostLogin. That is a function that anyone can call, and just make sure I have my syntax here correct, and inside of here we can say, dear location service, please go to that lastPath, and then we'll re-initialize lastPath back to just

a slash in case this happens again. So this member, redirectPostLogin, I'll add this as part of a service interface, and now there's all sorts of options on where to call this from. Well let's assume that we're going to leave that up to a controller, like the login controller. It gets to decide what happens after the user logs in. Currently we don't care what happens when this promise for login is successfully resolved, we only care if there is an error, that's why the catch statement is here, and we'll look at this alerting service later in the course. But now let me add a function that will be invoked if the login is successful. Let me also inject the loginRedirect service, and now we can just say on a successful login, let's do loginRedirect, redirect after a login. Save all my files, and let's come out and refresh this application, try to see today's secret recipe. I see the Login page. I'll enter my username, enter my password, and now I'm back on the view that will display that secret recipe to me. And this behavior should be familiar to anyone that has used a web application over the years. It's quite common when you try to go somewhere, find out we can't get there, the application displays a Login form, we login, and then we're taken right back to the protected area that we were trying to get to in the first place. All that required was just a little bit of code and an http intercepter.

Managing Tokens

One significant difference between tokens and cookies is how the browser automatically stores and manages a cookie. Once a website sets a cookie, the browser will automatically store that cookie, and then send that cookie along with every request to that website. Since we are only holding the token in the memory of our JavaScript application, we're going to lose that token if the user does a refresh on the page. That effectively reboots the application. That's probably not the behavior that we want, because it doesn't mimic the behavior of a traditional web application using cookies. I can login to a website, it sends a cookie, I can refresh that website as many times as I want, and the browser still has that cookie. It's going to send it along on every request that I am authenticated. The same just isn't true when we're using tokens, and we need to use JavaScript to store that token and put that token into every request. But fortunately there is an easy solution to this problem, we just have to write a little more code to store this token in HTML5 localStorage or sessionStorage. Let's see how to do that next. In the application, I am a logged in user who has an access token and can see the secret recipe, but if I refresh this page, I lose my token and I need to login again, whereas with cookies, even a session cookie would have still been around for the browser to send to keep me authenticated. And the reason I'm losing that token is because it is stored in memory and inside this application and when I refresh the browser that restarts the Angular application. So what I need to do is find a place to store this token and I have placed another service into this project that's called localStorage, and it's a very simple wrapper around localStorage and the browser. I can get access to localStorage through the Windows service, localStorage has a simple key value, store, and I've given the service the ability to add, retrieve, and remove keys and values from localStorage. There's a lot of other fancier things that you could do with a localStorage service like this. You could, for example, fall back to using cookies if localStorage isn't available on the browser, and you can find some third parties that have already written services for Angular that you can plugin, that use localStorage and sessionStorage that way. What's important to me is just that I can take and inject this localStorage service into my currentUser service where the token is

stored, and now when anyone sets the profile for the user I'm just going to take that entire profile and store it into localStorage. That way I can get the access token and the username back when this service initializes. So first I'll need some sort of key value to store the profile. Let's define USERKEY as just some arbitrary string value, just need that to be consistent, I'll call it "utoken". And when anyone sets the profile for the user I can say dear localStorage, please add this key and serialize this object to the localStorage, the profile. And now when the service starts up I can just add some additional logic here so that we can retrieve the profile if it already exists in localStorage. So I think what I will do is instead of assigning the profile here, I'll say profile is the result of invoking some initialize function, which I can write here. So, initialize is a function, but we'll still place some defaults here for the user, so we'll say user = this object that has username token and get loggedIn, but then we'll also try to retrieve that user from localStorage. So go to localStorage, do a get on that (USERKEY) that I'm using, and if that localUser exists what I'll do is copy the username and the token from localUser, so localUser.username and user.token is localUser.token, and now we can just return user. So if a user was in localStorage, that will now be part of the profile that is exposed throughout the application and gives everyone access to a persistent username, a persistent token. Let's save all the files and try this out. I'll do a hard refresh on the application, go to See our recipe, login. I can see the recipe. Let me try another hard refresh. I can still see the recipe because if I look in the resources of the chrome developer tools I can see the username and access token are stored there. And now that I'm using localStorage, there's probably some additional features that I'd want to put into the application, for instance, a sign out button, and that sign out button would have to purge that token from localStorage. And because the user might close their laptop and come back to this application a couple days later, we might want to test when we've retrieved that token if it's still valid, because tokens generally expire and some access tokens can have a very short life, 20 minutes or less. So when we've retrieved that token we might want to hit a protected endpoint just to see if it's still valid, and if it's not, remove it from localStorage. Those are the types of scenarios that I'd like to leave for you, the viewer as an exercise.

Binding HTML

Now let's turn our attention from authenticating users to a different security topic, the dreaded cross-site scripting attack. If you need to catch up on how a cross-site scripting attack works, then Troy Hunt has a number of great security videos on Pluralsight that you can watch. I'm going to assume that you know what a cross-site scripting attack is, and you just want to know how Angular can help prevent cross-site scripting problems. Now imagine the recipe data that we receive from the server includes HTML markup, and we need to render that data as HTML. Let's see what happens in this scenario with Angular. Back on the server side of this application, I've just made a change to the title of the secret recipe. This change will try to emphasize the word Toast, so the word is inside of an em tag. I've done a build, and let's see what impact this has on the rendering of our secret recipe model in Angular. And the tag in the title renders as plain text, it's not an element in the DOM, and that's because binding expressions in Angular will set the inner text property of an element, not the inner HTML property, and so special characters, like the left and right angle brackets, they are encoded into HTML entities and just appear as plain text in the browser. This is a good default behavior to have because this behavior will

help to mitigate cross-site scripting attacks. If someone makes the title of a recipe editable and we allow different users to edit this title and display that title back to other users, there's always a chance that a malicious user might get in and then try to inject script into our database so that other users will load that page and that script will execute and then the script can do things like steal the other users access tokens. But if we really do want this to display as HTML, we want the word Toast emphasized? Well, over here in the view I just want to show you that the double curly binding expression is really just an instance of the ng-bind directive. So I can place the expression secret.recipe.title into an ng-bind directive, refresh the page, we'll get the same result, but I'm just doing this to demonstrate that there is also an ng-bind-html, and this directive will try to set the inner html of a DOM element. So now let me refresh the page, and I'm also going to come over to the console, and you'll see that now the title is missing entirely. And we have a new error in the console, an error that says we are attempting to use an unsafe value in a safe context. This is because with Angular, if you're going to bind HTML into the DOM from your model, then Angular ultimately has to trust that HTML. There's a few different approaches that we can use to get Angular to trust our HTML and actually render French Toast with the word Toast emphasized. I want to show you two approaches. One approach is implicit, it can just automatically happen and we can continue just using ng-bind-html and it's going to work, and then the second approach is a little more explicit, but it also gives us a little more flexibility. Let's look at the implicit approach first, and that's going to be using the sanitization service.

Sanitization

One additional module that you can use with Angular is the Sanitize module. This module includes a sanitization service. The sanitization service tries to clean up the HTML you are going to use with ng-bind-html, so that that markup only contains safe markup, and then Angular will trust the sanitized version of the markup and render it for us. I'll show you how all of this works in a demo, but I do want to provide a caveat that you probably shouldn't rely 100% on sanitization to clean up data that might come from very determined, very malicious users, just because determined people can find all sorts of loopholes and try to sneak in a cross-site scripting attack. But I would use the approach I'm about to show you if the data is coming from a user that I do trust, perhaps someone in an editor role who works on the website and is adding dynamic content into the database. Let's take a look and I'll show you how this works. What I want to do is install the sanitize service into my application. I want to do this because ng-bind-html will automatically call into the sanitize service to make HTML safe and trust that HTML, and just by having the sanitize service installed, I should be able to then see the title of my recipe, and I should be able to see the word toast emphasized, it just happens implicitly. So the first thing I'll need to do is come into the list of scripts that I have loaded into my page, and I'll need to include the Angular sanitize script that's in a separate module, and therefore in a separate script. I have all the core scripts loaded into this folder, I'm just going to drag over angular-sanitize and fix up the path so that it looks like this. And now pretty much anytime you bring in another angular- script, that script will create a different module and I will need to go into my application where I define the module for my application, and also tell Angular that I have a dependency on that particular module. So in this case the module name is ngSanitize. And now I should be able to just save all my files, come into the application and refresh, and

just by having the sanitize service present I will get my expected output because the directive ng-bind-html, ultimately when it goes to display that HTML, it will pass the HTML through the sanitize service, which will clean out anything that it thinks is malicious. Basically it has a white list of what the legal HTML elements are, and also attributes. Then that HTML is marked as safe, and Angular will happily add it into the DOM. So what are the downsides of the sanitize service? If it's this easy, why might we need some other approach? Let me do this. Let me come into a page that displays the recipe title, and let's add an input so that we can easily modify the title. So I'll add an input type="text", and bind it with an ng-model directive, bind it to "secret.recipe.title", and let's try a few things out. So refresh this page once again, and there I can see my emphasis, so I could remove the emphasis, I could add the emphasis back in, and that all seems to work well, but what if I put, let's say, an onmouseover attribute here. Onmouseover="alert", and we'll just display an alert that says 'hit', and now if I try this out nothing happens. Let's actually look at that element in the chrome debugger, and I can see that the word Toast is inside of emphasis tags, but my onmouseover attribute is missing, and that's the job of the sanitization service is trying to sanitize the HTML to make sure it's safe, and remove any script tags or event handlers that might add executable script into the DOM. But what if I really wanted angular to take whatever I had in the model and place it into the DOM, even if what I have in the model contains script tags or event handlers in the markup. That's when I have to circumvent the sanitization service and be a little more explicit and tell Angular explicitly to trust this HTML. Let's look at that next.

Trust with $sce

In Angular, there is a service named $sce. The sce stands for Strict Contextual Escaping. Essentially the service creates a secure-by-default environment, where any directive that moves model data or templates into the DOM will do so by using sce to trust a given piece of markup or an href value or a template loaded from the server. If Angular doesn't trust a resource and we're doing something dangerous, like when we bind data with ng-bind-html, then we'll have a runtime exception and the operation fails as we saw earlier. We couldn't display our recipe title at all. Once we loaded the sanitize service, the operation worked, because Angular automatically ran our model data through the sanitize service and afterwards it trusted that markup. But now we want Angular to use raw data without sanitization, and fortunately we can use sce to create our own trusted objects. That gives us complete control over putting markup into the DOM just by using binding expressions. But always remember the possibility of cross-site scripting attacks. So only use this approach I'm about to show you with some caution, and when you're sure you can absolutely trust the model data that you have. Over in the application I'm going to copy the markup that I have in this input that has the mouseover attribute so that I can paste this back in later once I've made some modifications to the application. So I want Angular to use this recipe title as is, don't try to sanitize this title, and I'm going to do that using the sce service. Now because I'm binding against this title, a two-way data binding with an ng-model, I don't want ng-bind-html to go directly against that title anymore. I want that to be a property that can be bound with ng-model and someone can manipulate it. So instead of going directly to the title, I'm going to ask the secret controller to get me a trusted version of that title. So this binding expression now is a function in location. Dear secretController, please get me the trusted title, and now if I save that file I

will need to add this to my secretController. I will add that by injecting the sce service, and as you can see, the secretController right now, it's a very simple controller. Basically all it does is use that recipe service to get the secret recipe, and then make that recipe available for binding from the view. Now what I need is a function on this model, and we called this getTrustedTitle. So getTrustedTitle is a function that will return the result of invoking trustAsHtml on the sce service, and passing in model.recipe.title. So you can think of this function as taking my title and wrapping it in something that tells Angular, this is a trusted value. You can use this without sanitization. And once I save this file, that is now the value that we will be using when we bind HTML, and that means we should be able to get a mouseover attribute if we really want one here. So let me refresh the page and paste in the version that I cut out earlier that has an onmouseover attribute, and then move my mouse over the word Toast. You can see I now get the popup, and I'm not telling Angular to trust that model data, don't try to sanitize it. It's trusted, just put it into the DOM as is. That's always dangerous depending on who gave me that model data, but if you're building some sort of dynamic content management system, and you have trusted authenticated users who are entering in markup to be displayed to other users, one approach you can use to get that HTML into the DOM is to use the sce service, and that also makes it very easy to audit your code and look through and see the places where you are trusting model data and putting it into the DOM.

Summary

In this module, we learned how to use the sce service to trust some model data for HTML binding, and we also spent quite a bit of time managing access tokens in our application. We used HTTP interceptors to watch for requests that require an access token and also to add tokens into outgoing requests. We even used localStorage to persist the token across refreshes of the browser. All of that didn't require a lot of code, and that's one of the beauties of Angular. You could build small, focused, testable pieces of code, and plug the pieces into an application to make it work. Now one thing you might have been wondering throughout this module is how to get insight into how components are behaving, even built-in components, for example, the sanitization service that I talked about. All we needed to do was add that into the application and it just started working, but when is it being called, what is it doing with the data. We'll look at techniques that you can use to drill into an Angular application and see what is happening, in the next module.

Debugging and Diagnostics Playbook

Introduction

Hi, this is Scott Allen, and in this module we'll look at diagnostics and debugging for our Angular applications. Diagnostics are important for understanding what is happening inside an application when it runs, and in this module I'll show you how to find errors in your application, even data binding errors which can be notoriously difficult to track down. I'll show you how you can give the user feedback on

operations that have failed, and also show you some tips that you can use to debug Angular applications, including tips for the JavaScript console in your browser's built-in developer tools, as well as custom tools like Batarang and ng-inspector.

An Alerting Overview

When you're writing an Angular application, you'll commonly encounter some areas where you want to let the user know about certain events that are happening inside the application, and you also might want to give yourself the ability to see what is happening in an application. You might want to know, for example, when there is an exception from your code or from the code inside of Angular. You also might want to know when a promise is rejected, because that often signifies some error condition. Or perhaps you just want to know some diagnostic information, like how long did it take this block of code to execute, or when does this specific function execute. And in addition to errors, sometimes you want to know about successes, like when does the user successfully save an important record to the server. And it's not just you that wants to know this information, sometimes you might want to display some information to the user, some sort of alert that will tell them what is happening, and you'll want to do this somewhere other than the console window of the developer tools, because the average user will never look at the output there. So to support these various sources of information in an application, we're going to build an Alerts service. This service will be responsible for collecting alert type information from different places in the application. The service will expose an API that lets us add new types of alerts, error alerts, success alerts, warning alerts, and then it's going to have to manage a model, which is the currentAlerts that are in the application. We can inject the service into other controllers, directives, and services, and we can also use the service to send back information to our home server to record an alert in the database. Now funneling information from different sources of events into the alerting Service requires a few different techniques, depending on the information that you want. I'm going to show you some different techniques to use, techniques like decorating a service, but before we even reach that point let's look at a simpler scenario where a controller just wants to raise an alert for the user to see. This will require us to build out our alertingService that collects the alerts and also have a directive that we can use to display the alerts, so let's get started with the service.

The Alerting Service

In the application I had the shell of an alerting service ready to go. In other words I already have a js file for the service. That js file is included in the web page, and I've already registered a factory function that will return the service, but now I just need to figure to the API for the service. And what I'm thinking is that other components in the system will be able to inject this service and then use an API like addWarning with some descriptive text to tell this service there is new alert, it's a warning type alert, and then it would be up to the service to decide what to do with that. And I think since the service will manage different types of alerts I'll need a more generic method up here, let's just call it addAlert, where you need to pass both a type, for the type of the alert, and the message to associate with that

alert. And ultimately since what this service has to do is manage all the currentAlerts that are in the application, I'll just create a new empty array that I can push objects into. So every time someone wants to add an alert I'll say currentAlerts. Let's push in a new object whose type is set to that type, and whose message includes the textural description. And this function can also be a part of the public API for this service, but I also want convenience methods like addWarning that would be a function that just takes a message and it will call through into addAlert to say, let's add an alert of type "warning," and here are some texts associated with that alert, so this is also a part of the public API. And now I can follow this basic pattern to add additional methods, to add danger type alerts, success type alerts, informational type alerts. So let me just add in those few methods and we'll be right back. Now I've added a few more functions to my service. I have the ability to add "warning," "danger," "info," and "success" type alerts. You might notice these alert types map directly to Bootstrap alert types or the CSS classes that Bootstrap uses to display alerts, and that's no coincidence. I want to use the Bootstrap CSS framework when I display these alerts, but there's no need to be coupled to Bootstrap, I could always use different alert types or different vocabulary for my alerts, and map these alerts to Bootstrap styles. But these functions in the alert service, they are all exposed by these service objects returned from this factory function, and here is one more piece that I want to expose. The array that this service is using to track the currentAlerts in the application, I actually want to expose that as a public member here, and this is one of the big mental breakthroughs that I had with Angular. The currentAlert array is effectively a model, an object full of information that I want to bind into the display, and by placing this model in a service I can share it across different directives, controllers, and other services, so yes, models can live in services, we saw that in the previous videos when we stored a user profile in a service. But now that we have a workable service, let's work on something to display the alerts that will be added to the service.

The Alert Directive

I'd like to encapsulate the display of our alerts into a custom directive. This is an ideal use case for a custom directive. If I come into my shell page, you can see I have a two-column layout here with Bootstrap, so one div is a column 8 and then I have another div, column 4, that'll be on the right-hand side of the page. What I'd like to be able to do is give developers the ability to just write alerts, or perhaps a div with an attribute of alerts, and have all the alerts display on this part of the screen without doing any additional work. So over in my project folders I already have an alerts directive, or the shell of an alerts directive, I just need to tell Angular that here is a function which will return the directive definition object for something called alerts, so we will return the DDO. Let's explicitly restrict this custom directive to being either an attribute or an element. We will need a template to display the alerts. I'll use a template file, so let's specify "/apps/templates/alerts.html". That's a file I already have in place, but it's empty, we'll have to create that view. And very important here, I do not want to use the existing scope object. I either want to create an isolated scope or I want to set scope true to get a new scope object that will inherit from its parent. The reason I want to do this is because we're going to need to manipulate the scope, and if I'm sharing the scope with my parent I can run into trouble because I might overwrite something that the parent already has, and in general just have some unintended consequences. By setting scope to true I'll get a new scope object, and that will allow me to add some

things in the scope without harming others, so I'll need to inject that alerting service, and I'll need to get the currentAlerts from that alerting service and put them into the scope object for this directive. One place I could do that would be in a linking function. The first parameter to the link function is the scope object, and I can say scope.currentAlert = alerting.currentAlerts. Now, because services in Angular are always singletons, I will have a reference to that currentAlerts array and anytime the service pushes something new into that array I will also see that from my currentAlert pointer here on the scope object. And that means I should be able to come into my template file, which is here, but empty, and set up a bit of a UI. So let's create a div that will ng-repeat over "alert in currentAlerts". And to style this with Bootstrap I can say that the class for this div should be alert and then alert-, and interpolate an alert.type. So that will be pulling out each alert from the array and using its type property to help style this div. Inside the div I can have the message associated with that alert, and that should be enough UI where we can start trying this out by using a controller that will add some alerts using the alerting service, let's try that next.

The Error Prone Controller

And now, back to the shell page. You will see I already have some UI set up. There is a controller, the errorProneController which is currently empty, we're going to have to build that, but it is in control of this section of the DOM, and inside of the DOM here, if I flip over to the application you can see the UI I'm building. The idea is to allow the user to type in some Alert text, and then select the type of alert that they want to create, that drop-down list currently not populated, we'll have to work on that too, but then you click the CreateAlert button, and that should send a message to the alerting service. So I'm going to need to bind the text from this text box, the selected alert type from this drop-down list, and wire up a click event with ng-click to this CreateAlert button for everything to work. So let's switch over into the errorProneController, which is currently registered with Angular, but there's nothing inside. So let's say that we will bind the alertType to an alertType property, and of course I'm going to set up model to point to this, because we're using controller as syntax, and I will also have a model.alertMessage which will be initialized to an empty string. Now back in the UI, in order to populate this drop-down list I'm going to need a list of the available alert types. That might be best to manage from the alerting service itself. So back up here in the top of the service, let's create an alert type array, and we can have alerts that are "success" messages, "info" messages, "warning" messages, and "danger" messages, and now I just need to expose alert type, or that should really be alertTypes. I need to expose that also, as something you can get to from the service object. And so in the errorProneController I will inject alerting, and see that the model.alertTypes will be equal to alerting.alertTypes. So let me save this and switch over into the shell and see if we can get this to display. This input is where the user will type the alert text, so let's use ng-model to bind that against "model.alertMessage". And here is our HTML select control. I'm going to say that it binds against "model.alertType", those are the properties that we set up on our model for this binding, and I can also use ng-options, which is a very flexible directive, but I'll use a very simple style here to say, please give me options that will by "type for type in model.alertTypes". And if I just save everything and come back to the UI and refresh, hopefully we'll now have a drop-down list, which we do. We could do something a

little more sophisticated to make sure that these entries don't appear with a starting lowercase letter. It might be nice to see a capital first letter, but let's work with this. The next thing I'll need to do is wire up the Create Alert button. I'll use ng-click and say that this goes to model.createAlert, and that's a method that doesn't exist on my controller as yet, we need to create that. So I'll save the shell file, flip back into the errorProneController, and say that model.createAlert, that is a function that can be invoked. And when it's invoked I just need to go to the alertingService, tell it to add an alert. The type will be model.alertType, the message will be model.alertMessage, and once the user has done that, it might be nice to blank these out again, so model.alertMessage is an empty string, model.alertType, also an empty string. Or we could, now that I think about it, let's have a default alertType. We will just make it alertTypes sub zero, whatever is the first element in that array. So save everything and let's try this out in the UI, do a hard refresh. This will be a success message, so this works, create that alert, and that appears just where I expected it on the screen, that's good. Let's create a warning message, This might not work. That also appears on the screen, and I could continue creating messages, but you might be wondering if these messages will ever go away. Let's explore a couple of different approaches. One approach might be to add some sort of close button here so a user can dismiss an alert. And another approach which we can also implement would be to just have these alerts go away after some specified period of time, maybe 10 seconds, or 30 seconds. We'll look at those two scenarios next.

Remove Alerts

Let's first give the user the ability to remove an alert by clicking on something that is inside the alert, and that would be the responsibility of our directive template. I'll need to add a little bit of additional UI here. I'm going to add a div that has a class of close, that's a Bootstrap class that will effectively push this piece of content out to the right-hand side, and inside of here I could have a button, or let's try something with an icon. So a span with a class of "glyphicon, and glyphicon-remove, that should be an icon that looks like an X, something that implies that if you click it, it should go away. And then we just have to tell Angular what to do when someone clicks on this piece of UI. I'm going to say, let's remove the Alert and pass in the current alert that we are on inside the repeater, and that means I will need to add removeAlert to the scope for this directive, so let me save that UI and come over to alerts.js. Now the question is where do I add removeAlert? I could add that to the scope object inside of the linking function, but of course a directive can also have a controller that is good for logic, and this controller can just be an inline function. This will take a scope object, and this will be $scope because this is an injectable function, and if I want scope, I need to use $scope. That's unlike the link function where the parameters are fixed and it's always scope, element, and attributes. But now I have a controller for this directive, and I can add to the scope object the removeAlert function, I just piece it in that name. And now the next question is, how do I remove an alert from the currentAlert array? I do have access to that array, it's on my scope object, but it would be bad form for this directive to directly manipulate currentAlerts, that's really the responsibility of the alerting service. So let me tell the alerting service to remove an alert, and of course this function will take the alert object to remove, that's what we're passing in with ng-click, and I'm just going to pass that object along to the method call on the alerting service. So now when I save this I will need to go to the alerting service and add a removeAlert function

here. Let's do that down here at the bottom of the file. So removeAlert is a function that takes an alert object, and now there's all sorts of fancy ways to find that object in the array and pull it out. If you're using a library like Lo-Dash or Underscore, this would be a one-liner, but we'll do something a little more low-level since I don't have any of those libraries included. I will loop over currentAlerts using a variable i to go from 0 to the length of the array, and inside of here check if currentAlerts sub i is equal to that alert object. We will essentially be comparing pointers there, and then if this is the alert that I want, I will tell that array to splice out at the index, exactly one element, and then we could break out of this loop because we're done. Now I just need to expose removeAlert as another method here on the service, and let's save everything and try this out. I'll come out to the UI, do a hard refresh, let me just add a couple of quick Alerts to the screen and click on the button and I can see them go away, and they go away in the proper order, there's 1 and 2, I click on 1, it goes away, I click on 2, it goes away, so that's good, but what if I wanted these messages, these alerts that appear on the screen, just to time out after some period of time and disappear? That would actually be relatively simple. What I could do in the alerting service is inject the timeout service, and then every time we add a new alert then that always comes through this addAlert function. I will just set up a timer to remove that alert, let's say, after 5 seconds, which is a very short time, but we want to see it work relatively quickly in this video. I can say removeAlert, this alert that I just created, but of course now we will need to close around that object, so let me create an alert variable that points to the object we need, pass it to push, and then set up our timer. Now technically, a user could remove the alert before the timer expires, and nothing should break, we'll just go off searching for an alert that doesn't exist in this array when the timer does expire. If we wanted to be really picky and finicky about that scenario, what we could do is save the promise the timeout returns, and cancel it if that alert is removed, but let's just forge ahead with a simple case here and see if this works. So back at the UI, do a hard refresh, and let me just add a couple of success messages, and we wait a little bit, and they start disappearing with the timer. And what we have now is a relatively robust alerting service that doesn't require a lot of code, but it allows us to add alerts, it will remove alerts, and we've tied it together with a controller and a directive, and again it's a case in Angular where you can build up some big functionality just using some small manageable components that all work together. And now that we have the service built, let's see how we can use it to catch and display real diagnostic information.

Decorators

Now that we have some working infrastructure in our application, it's time to demonstrate how to show some useful alerts, and one technique I use to grab useful information is the decorator pattern. Angular supports this out-of-the-box. You can decorate any service in Angular, the built-in services, the third party services, even your own services. The decorator design pattern, generally speaking, is used to wrap, change, adapt or extend a component. In Angular we typically decorate a service to wrap that service and provide additional behavior or even change the existing built-in behavior. Here is how it works. In Angular you can identify a service that you want to decorate by name, like perhaps you want to decorate the $log service. So I will tell Angular that I want to decorate the service, and I will give Angular an object that looks just like the original service, but it contains my custom code, and my

custom code might use the original service behind the scenes and might delegate behavior to that original service. And now, any component that comes along and needs that service, Angular will inject my decorator into that component, but the beauty is that the component asking for a service like $log, it doesn't know if it's working with the original service or if it's working with a decorator because it doesn't need to know. This is very much like creating a class in object-oriented programming and taking advantage of polymorphism, but decorating is more of a compositional technique, it's not inheritance, and composition generally gives us more flexibility compared to inheritance, so this is good. Let's take a look at how this works in the code and how decorators can be useful.

Decorating $exceptionHandler

In the application, I already have the UI set up with a button that when I click it, what I want the code to do is throw an unhandled exception. Currently that's not happening, let's wire that up really quick and then observe the results. So on this Throw Exception button, I will use ng-click to tell my model that I want it to create an exception. So I will need to take this expression and go over to the errorProneController and add that as a method to my model here. So model.createException will be just as easy as doing throw new Error, and we could put a message in here like, "Something has gone terribly wrong!", and let me save everything, go back to our user interface, do a hard refresh, open up the developer tools with F12 and make sure I'm on the console, and then click Throw Exception. You'll notice that in the UI that the user typically sees, there's no indication that there was an error, the average user will have no idea that something went wrong, but as a developer with my browser tools open, I can see there was an error raised at model.createException. So the goal of this video clip is to get that exception onto the screen, or at least let the user know that something has gone wrong. And this may be the type of behavior that you don't want active all the time in the application, you might not want every exception to go to the screen, but you do typically want to know about these unhandled exceptions because it means something has gone wrong inside of your application. So what's happening behind the scenes is that any callback that Angular invokes is the result of a directive, so the response to an ng-click or a mouseover, or the response from a $http call. Angular is going to call into your code inside of a try catch handler, so that if your code throws an exception, Angular will catch that exception and it's going to pass that exception off to a service known as $exception handler. So if we want information about what exceptions are occurring inside of an application, one approach we can use is to decorate $exceptionHandler. Now I already have a file in place and loaded into the page called exceptionHandler.js, I just need to write the code inside of here that will register a decorator for the exceptionHandler service. And I want to decorate that service, I don't want to replace it, although I could. I just want to decorate it so that I can allow Angular to handle its exceptions using its normal course of actions, but I also want to know about this exception, so I'm going to provide a wrapper for this service, and any exception that Angular hands off to the exceptionHandler, I will also see that exception. So the way to create a decorator is to create a config function for your module and give Angular a function to execute for the configuration phase, and this function will take $provide. You might know that you can use provide to create services, configurable services, but you can also use provide to create a decorator. Now when you provide a decorator, you provide the name of a service

that you want to decorate. So in this case I want to decorate the built-in exceptionHandler service, but I can decorate any service that Angular knows about. And then I provide a function which will return the decorator. Now the object that I return has to essentially be greater than or equal to the original service. In other words, if I was decorating the $logservice, I would need to return an object that looks just like $log to anyone that's using it. And because the log service has four methods, .info, .warn, .error, log, I would need to provide an object that has at least those four methods. I can add more methods than I use for my own code, but I at least have to provide those four methods for existing clients to be able to use. In the case of exceptionHandler, this service is an object, but it's a function object, so I need to return something that is a function that other pieces of code can invoke. That's what everyone expects when they get an exception handler, that's what it is documented as, it's a function. And the exceptionHandler service is also documented as taking two parameters, so this function takes two parameters. The first one is the exception that was thrown, and the second parameter is the cause of that exception. And so now the question is, what do I want to do with this information. Well I do want to call into the original exceptionHandler service and allow Angular service to do whatever behavior that it does by default, and I'll be able to do that because the first parameter to this decorator function is a parameter we'll call $delegate, and that parameter always represents the original service. So if I was decorating $log, delegate would be the log service. I'm decorating $exceptionHandler, so delegate is the exceptionHandler service. Since that is a function, if I just invoke delegate and pass it the exception and the cause, I've now provided a decorator that just passes information through to the original service, I haven't added any additional behavior. But if I save everything and refresh my UI, hopefully we won't see any errors about that until I throw an exception, I just see the original behavior. So now my goal is to add additional code inside of here so that I can see something on the UI. I want to pass the message associated with that exception to the alerting service so it can put something onto the screen. So let me show you what happens if I inject the alerting service into this decorator function. Now I will do a refresh, and you'll notice we get an error in the console window, it's Circular dependency found: $rootScope <- $timeout <- alerting <- $exceptionHandler <- $rootScope. This is an error that you might commonly see when you're decorating low-level services from Angular. It's basically the injectors way of telling you that it cannot instantiate this particular chain of dependencies because there's a circular dependency. Alerting depends on $timeout, which depends on exceptionHandler, and it just goes around in a loop. So there's a number of different ways to solve or break that circular dependency, but one surefire approach to do that is to not inject the service that creates the problem, and instead inject the injector itself, that is the IoC container that's inside of Angular, it's the service that does all the dependency injection. But I can use that component to get to my alerting service. I can just ask the $injector to please get me the service called "alerting". But notice I'm doing that inside of the decorator function, so that piece of code is going to execute later than this function that creates the decorator, that's what's going to allow this to work. So I will use the injector to get to the alerting service, and then I will tell the alerting service to add a dangerous alert using exception.message, which you might have to play around with a little bit, there's no guarantee that message is going to be there, because JavaScript code can throw anything. You can throw a string, you can throw a number. The best practices will say to throw an error, but you might be using a library that doesn't do that, so you might need to do a little introspection on this exception before you try to dig out some sort of message or reasonable explanation for it. Now that I have those two lines of code in here, let's save this and come back out to

the UI and do a refresh, and now I want to throw an exception, and I can see that appear in my UI. I can also still see it in the console window, but that's only because I called into the original exceptionHandler. So hopefully this shows you some of the value in decorating a service. I'm sure you can think of other uses for the information that you get inside of this particular decorator. You could, for example, send this information back to your home server to collect information, and this is just one example of a decorator, a decorator for exceptionHandler. Let me show you another interesting decorator. The scenario for our next service decorator is this.

Decorating $interpolate

I want to show the user how many characters they have typed into the input for the alert message, which sounds easy enough to do. All I need is a data binding expression that looks at the model.alertMessag.length. But oops, I have a typographical error, I forgot the e here, but back in the UI there's no visual indication of an error, there's not even an error logged in to the console of the developer tools. And if you've worked with Angular for any length of time, you'll already know that Angular is very forgiving about data binding expressions, and that has a good side and a bad side. The good side is that I might be waiting for this piece of information to come back from an API call that I made to my server, so the data will be there eventually, but until it shows up I don't want to have to use all sorts of expressions in the view to check if something is null or undefined, I just want Angular to ignore the fact that I'm dereferencing something on an undefined property, and just keep the application working. But the downside to that behavior is that if I do have a true error, like a typographical error, or I re-factored the model and I renamed this property, then there's no indication anywhere in the browser that something is wrong. What I would like to do is build a component that will show me in the console what every data binding expression on this page is producing. Now on this page we only have one, but the reason I don't want this information to go into the Alerts area that we have, is you can certainly have pages that have dozens of data binding expressions, you don't want all that information overwhelming the display. I just want to log this information into the console so I can visually inspect it and see, oh, here's a data binding expression that didn't produce a value, maybe I should investigate that. To get this behavior, I'm going to build another decorator. So I want to use $provide and tell it that I'm going to create a decorator for the $interpolate service, because every binding directive that you have in a view, whether it's an ng-bind directive or it's a binding expression between two curly braces, Angular will go to the interpolate service and ask the interpolate service to parse that expression and produce a function that Angular will invoke every time it needs to update the DOM with a new value. So for every binding expression on the page, Angular will call into interpolate and interpolate will give back a function. That function gets invoked every time the data binding expression is re-evaluated. So I'm going to need to decorate this service and also decorate the functions that it returns, so I can inspect the values that are being produced. But first let's decorate the service. I'll need to take $delegate once again, and I'm also going to take $log, because we're going to be writing into the console and not handing these things off to the alerting service. Now interpolate, much like exceptionHandler is just a function. So ultimately what I need to do is I need to write a function that will be a wrapper for interpolate and then return that function so it effectively becomes the interpolate

service. And there's one additional little step here, which is that because JavaScript functions are objects, functions can have properties and methods attached to them, in fact, every function will, but Angular adds some special properties to interpolate, and what I'm going to need to do to have my function look just like interpolate is use angular.extend to copy into my serviceWrapper all those things that come with $delegate, so now serviceWrapper will look just like the interpolate service. Now unlike exceptionHandler, where I created a function that took parameters because I knew exactly how many parameters were coming in, and I needed to inspect those parameters, what I'm going to do here is just invoke the original delegate function using JavaScript's apply. So apply will invoke the original interpolate service, and I will pass in this, my current context, and also arguments, the implicit variable that you have inside of every JavaScript function that represents all of the arguments that were passed to this function. So we don't need to explicitly add those here, I'm just going to pass them through, I don't care what they are. Well, actually I do care about one of them, however, when I invoke the original interpolate service, it will give back a bindingFunction, and what I will need to do to inspect the results that a binding expression yields, is wrap also this bindingFunction. So I'm going to check if this function is a function, and there's a helper method on Angular to do that. And I'm going to check if I have something in arguments sub zero. So what is in arguments sub zero? Well, if you have a binding expression like model.alertMessage.length, that will be argument sub zero to the interpolate service, and I want to make sure that not only is this a function, but also that that expression is there before we build a wrapper for this binding function. Otherwise, I'm just going to return the original binding function. But in this scenario what I want to do is return a wrapper for that binding function, something that will wrap that function and will pass in arguments sub zero, which is the binding expression. So what is bindingWrapper? BindingWrapper will be a function that takes the original bindingFunction and takes the binding expression and it will return a function, because again, we're essentially decorating a function, in this case the function that is used to evaluate every data binding expression. And so when I want to see the result of a data binding expression, I can just call the bindingFunction. Again, we'll use JavaScript's apply and just pass through the arguments that are available, and that will give me the result, the result that ultimately goes on the screen. I can just return that result, but I also want to see it in the console. So in between getting the result and returning the result, let's do some logging. The first thing I will do is trim that result, just because when you have a binding expression, like we do back here in shell.html, the result of this expression will also include all of the white space that we have inside of here, because a binding expression basically sets the text content of an element, and since white space might be significant for some elements, like pre, Angular's binding expression actually takes all of that into account, so I don't want to see all of that in the console. I'm going to trim it, and then I need to decide how I want to log it through the loggingService. Do I just want to use log.info, which produces a normal console message, or do I want to use log.warn, which will show up as a warning in the console? Let me create a variable that will essentially be a pointer to one of those two methods. So if trimmedResult is truthy, so it's a string that contains a value, let's just point to log.info, so it's really just an informational message. Otherwise, point to log.warn, so this will show up as a warning, and then I'm going to call that log method on the log service, and pass in a string that will show me the bindingExpression "=" + the trimmedResult. And of course I will just need to do proper string concatenation here. So this serviceWrapper should provide us some insight into what the data binding expressions are doing on a page, and also when they get evaluated. So let me do a refresh, and I can see

that I'm getting warnings. I can see that model.alertMessage.length with the missing e is not producing a value, and I can also see a lot of white space here in the bindingExpression. So let's make sure that the bindingExpression is also trimmed, and then let me fix my view, let me put an e in here, save everything, let's refresh the browser, and now I can see a 0 appear in the UI, I can see information log to the console that shows me that model.alertMessage.length produced the string 0, and if I start typing in here, I can see I'm up to 8 characters, and that bindingExpression was evaluated twice for every keystroke. So not only can this service decorator give you some insight into when you might have a bad bindingExpression, but it can also show you how many times these expressions are evaluated, and exactly what results they are producing. I have found it to be a very useful debugging tool.

Counting Requests

So far we've been using service decorators quite heavily in this module, but not all the information you need has to come from decorators. One prime example is network requests. We typically want to show the user some sort of animation when there is an outstanding request, because it let's the user know that there is some activity happening, perhaps we are fetching their data or saving their data. Instead of a decorator for this scenario, the ideal component is an http interceptor. We built some interceptors in the last module, so I won't go into detail about how to build one, but I do want to show you a basic requestCounter interceptor and how it could work. In this file, I am building a requestCounter, and just as we saw in the last module, I'm going to register that as an http interceptor, by pushing it into the interceptors collection on HTTP provider, and also as we learned in the last module, an interceptor can have 0 to 4 methods which are request, response, requestError, and responseError. We have all of those here in addition to a function that will return the current number of outstanding requests. So how do we compute the number of outstanding requests? Well, we simply increment that value whenever the request method on this interceptor is called, and we decrement that value whenever there is an error, a response or a responseError. And we also return the appropriate promise using q.reject when there's an error and q.when for a promise that should resolve successfully. Now since this interceptor is also registered as a service, because that was a factory method that we were looking at, I can inject this service into other directives and components and controllers, but in this case I just want to inject it into a directive, a directive called workSpinner. This workSpinner you can use it as an element or an attribute. And essentially what it's going to do is transclude content, and only show that content when there is a true requestCount, so a requestCount other than 0. This directive will know that because it's going to set up a watch that keeps monitoring the requestCount and since that's a primitive value, an integer, we're going to have to invoke a function to get that, and whenever that requestCount changes, the second function will be invoked by Angular and it will pass in the new value and we're simply going to save that into our scope, in some isolated scope in this case. And now how do I use the directive? It's very simple. Somewhere in my notification area I can just have work-spinner, any markup that I put inside of here will be transcluded into that directive, and this is the content that we'll show or hide depending on if there is an outstanding requestCount. Let's try it out real quick. In the errorProneController, I am going to inject the http service, and then somewhere down here at the bottom where I typically include initialization logic, I will do a get to /api/slow. That's a special endpoint

that I've set up that will take 2 seconds to respond. And now I will make sure I have everything saved. Let's refresh this page. You can see there's our outstanding requests, it's going to be around about 2 seconds and then gradually fade away. So http interceptors are another way to collect information. But I do want to examine a very specific scenario here. What happens if there's an error that comes back from the web server? So in this case I'm going to call an API endpoint that doesn't exist, and the web server should return a 404 error to me. Now typically when you are making an http request, you're going to have a .then on the promise, returned by an http operation, and it's in here where you can process a successful response, and then you can pass another function, which is your errorHandler. Now the one problem I've had with this approach is that when you're working on a team and building an Angular application, if everyone is writing their own local errorHandler, then the errorHandlers tend to become inconsistent. One person will write an errorHandler in a controller that when there's an error it will add something to the alerting service, another person might forget to do that, and then yet another person might actually have an error in the success handler which isn't caught by this errorHandler because the promise is either going to invoke the succesHandler or the errorHandler. If there's an error inside the code here, that doesn't go into the errorHandler. So what I'd like to show you in the next clip is how to setup a consistent errorHandler whenever you're dealing with promises.

Catching with Promises

One of the basic rules I have with promises, particularly promises that represent outstanding work on behalf of the user, is that the code I'm looking at either has to return a promise or have a catch handler, because I either want to see you return a promise to someone else who needs to process that promise, or you're going to have a catch handler and make sure you capture all errors. That would include if something goes wrong in the network request and a promise is rejected, or if there's an error somewhere in the then processing. Since this is a controller, it's not going to return a promise, it's at the top of the call chain. I want to see the catch here, so now the only question is, what sort of function do I write inside the catch handler that will consistently process errors? Instead of every developer writing their own catch handler, what I'd like to do is centralize that logic and have a consistent catch handler. So I could put that logic inside of a dedicated service, or what I'm going to do in this case, since we already have an alerting service, I'm going to add that logic here, because ultimately if anything goes wrong with that promise I want to display some sort of message to the user, and I probably don't want to display an http status code. What I want to display is some sort of description that describes what that promise was trying to achieve. So the idea here is that when you write your catch statement you will go to the alerting service and ask it to give you an error handling function by calling errorHandler, and pass in a description of what you're trying to achieve. Perhaps you're trying to retrieve a record or save some data. In this case I'll just say, "Failed to load data!", a somewhat generic error message, but I'm just trying to simulate a scenario here. Hopefully you'll be able to provide a more descriptive error message in a real application. You could say, failed to load account with the number, or Failed to load employee. We can see what my errorHandler has to do is it has to return a function that will be passed into catch, and then when anything goes wrong with this promise I'll be able to display "Failed to load data! as an alert. Let's finish building out this function, and also, before I forget, expose this function as a

public memeber, so errorHandler will point to errorHandler. And let's do something very simple, I'm just going to return a function, and this function can simply forward this description to the addDanger method that is already inside of this service. So let me save everything, and we have our controller calling a bad endpoint, so when I refresh we'll get a 404 response, but we'll put failed to load data on the screen. And now I have a method that I can use for my alerting service that will provide consistent error handling throughout the application.

Debugging Tips

Although diagnostics can give you some great insight into what is happening inside an application, sometimes you need to get even deeper into an app to see what's happening. Of course you can always step through code and inspect variables using the debugger in your favorite browser's developer tools, but I want to show you a few techniques specific to Angular that work at this level. First up are some tools. Angular Batarang is a tool that installs as a chrome extension, and you can use it to inspect the scope objects in an application and see all of your model data. You'll just need to do a search for AngularJS Batarang, that should bring you to chrome.google.com, the web store, and if you haven't installed Batarang as yet, there will be a button in the upper right that you can click to install this. And once it's installed, and you're on a page that's using Angular, just open up your browser's developer tools, the Chrome developer tools, and there will be a new tab, AngularJS. You'll want to enable Batarang and then you can inspect the scopes. Unfortunately, as I'm recording this, a new version of Batarang was just released. It's a version that has been rewritten from the ground up and it's more than just a little bit flaky, but if you want to give this tool a try, you can install it from the Chrome Web Store. Another tool is ng-inspector. This one will plug into Chrome or Safari, just go to ng-inspector.org, scroll down to the bottom, and you'll be able to find your download links. With ng-inspector, you can go to a page using Angular and you can click on the ng-inspector icon that should appear up in your toolbar. And again, the idea is the same, you should be able to inspect the active scopes that are in your application. When you select one, here I can see that the errorProneController is the controller associated with the scope, and it's also associated with the highlighted portion of the DOM. I can drill into it and see my alert types, my current alert message that will update as I type into something that is bound to that piece of the model, and both of these tools are useful for getting some insight into what your Angular application is doing, but I wanted to show you some things that you can do just from the console, some interactive things that are often quite easy to do and can give you some valuable insight. First of all, since the console represents a window into the current state of the runtime, I have access to the same globals that my application would have access to, like Angular. And that means I also have access to Angular.element, which would allow me to go out and wrap things like the document element or go out and search for an element by id or CSS selector, and then start manipulating that element with the API that Angular provides. Now here's a tip for Chrome. If you go out and inspect a specific element inside the list of elements here, that last element that you have selected will be available as $0. So if I create a variable that is angular.element on $0, element now points to my div that errorProneController is in charge of, and it is wrapped with the Angular API, and that means I can do things like ask this element for its current scope, which means I can drill into the model of this associated with this, and I

can inspect the variables here. I can also ask for the controller that is associated with this element, just by saying element.controller, and you'll see those two are the same thing since we're using controller as syntax. And let me save a few things off. I want to say scope = element.scope, and then I want to say element.controller.createException. Now that throws the expected exception, but you might be wondering why that's not showing up on the screen, and there's a couple good reasons for that. Since we have called this method directly, Angular is not in control of the execution of createException, therefore, Angular wasn't able to catch that particular exception. And that's the one thing you have to be careful with in the console. There's a few scenarios where you can actually go in and interactively manipulate your controllers, your models, even your services, which can be very useful for figuring out how they're behaving, but if you want changes to the model to actually be reflected on the screen, you're going to have to do things like scope.apply and execute your bit of code inside of a function. So now I can say element.controller.createException, and I now I get something onto the screen. I can even ask this element for the injector, the IoC container that is in charge of this application, and I can get a reference to any service this way. So I could say that the alerting service is element.injector, please get me the alerting service. And now I can say alerting. Let's add a danger alert or an info alert, and that will be added to the currentAlert, but again, the DOM isn't updating, and that's because Angular is not running the digest phase, it didn't know that this bit of code is executed, it hasn't detected any changes in the model, so once again, an easy solution to that problem would be to execute that same code inside of an apply. So I can say alerting.addinfo("test"), and now that appears on the screen. And this has been a really instructive way for me to learn some things in Angular to be able to interact with the explore services and just work with them from the console to see how they behave.

Summary

In this module, we looked at a variety of techniques to gather information about an Angular application. Service decorators can be your best friend for looking at information going into and out of any service, while HTTP interceptors can show you all the http messages that flow between your application and the server. We also took a quick look at debugging Angular from the console, and how to provide the consistent error handling strategy across all of your code base, which is really important for large teams and large apps to have that consistency.

UI Playbook

Introduction

Hi, this is Scott Allen, and in this module I want to give you some ideas, strategies, tips, and some of what I believe are the best approaches to use in user interface related topics that I've experienced in building Angular applications. Like the thought process for building custom directives, and how to integrate some useful third party libraries like UI-Router and AngularUI, to add capabilities beyond what

the core Angular library provides. Let's get started by looking at the type of application we're going to build in this module.

The Application

Most non-trivial Angular applications are going to use routing, because routing has many benefits, like the ability to break an application into multiple templates, and that's a big benefit for developers, not to mention the user benefits, like bookmarking and browser history support. As you probably know, with Angular's current built-in router you typically define a shell page with your application, and the shell page includes common markup for the entire application, like the head tag, the body tag, and any script files that you need to get started, and then somewhere inside of that shell there is an ng-view directive. The ng-view directive talks to the Angular router, and asks the router to consult the routing rules that you've configured, and figure out what to do given the current url in the browser. So if the current url in the browser ends with #/dashboard, the routing engine might be configured to load the dashboardtemplate for this route. So ng-view will retrieve the template, cache the template, compile the template into the view, then place it into the DOM, and we are off and running until the URL changes and the process can happen again, but perhaps with a different template the next time. And this routing setup, like I say, it's nice because a large application can be built from small pieces that plug into the UI at just the right moment. However, the current Angular router does have some limitations. What if we went to break the dashboardtemplate itself into smaller pieces? That is, what if we want the dashboardtemplate to be like a shell page inside the shell page, where the dashboard template provides some common UI pieces for all the dashboard functionalities, and then there are different areas inside of that dashboard section where the user can navigate. Perhaps the dashboards for restaurant software allow the user to look at an overall dashboard, a food cost dashboard, a customer count dashboard, and an inventory dashboard. So each of these features, it has a route, and it should plug a different piece of UI into the dashboard view. With a built-in Angular router, this could be accomplished using ng-show or ng-switch or ng-hide directives, and also a pile of custom code, but it's very painful. An easier solution is to use a different router, a popular project in the Angular world by the name of UI-Router. Let's get an introduction to that next.

UI Router

Angular-ui.github.io, this is the home page for AngularUI, and AngularUI includes a number of great modules that you can use with Angular. We'll use a couple in this module alone. The first AngularUI module that we will look at is called UI-Router, and you can find this module if you do a search or scroll around here on the home page. Here is UI-Router, and if I go to the site, this opening page demonstrates pretty much everything I need to know to get started. I will need to download the ui-router script and place it into my page after I've included angular.js, and then in the module where I am using ui-router, I'll need to tell Angular that I am taking a dependency on ui-router. And after that setup is complete, using ui-router is actually very similar to using ng-route, conceptually similar, because I still need to

define some routes, and I need to map URL's to instructions to tell Angular what to do when we've reached that URL. I also need to define views and put placeholders in my view so that other views can load into the DOM, and those views that load, of course, are dependent on the routes that I've defined. So let's try to build up the scenario I described earlier. I want an application that has a couple top-level areas, Dashboard and Admin. Each of these areas should load a different view into the application, but then inside of Dashboard I have different sub areas. Like one section of the dashboard will show me how many customers have visited the restaurant on each day of the year, and another section can show me the types of workplace injuries that my employees are experiencing. So let's build this out using ui-router.

Configuring States

In my application I already have some UI set up. I have a shell page with two links on it, Dashboard and Admin, so I need to configure a ui-router to load different views into the browser here, depending on which option the user has selected. So inside of the shell page for my application which is called "ui" for this module, I have already downloaded ui-router, I have included that script after angular.js, and in the angular module definition for this application, I have told Angular that I have a dependency on ui.router. What that will allow me to do in any config function for this module is ask Angular for the component, stateProvider, which is just like routeProvider, but stateProvider wants you to think in terms of state for your application. For instance, when the user clicks on Dashboard I want to enter the dashboard state for my application, and then the admin state for my application. So I actually tell stateProvider about named states in my application. I will have a state called "dashboard", and I will have a state called "admin". So instead of dealing directly with URL's, like we would with a routeProvider, we've sort of raised the level of abstraction just a bit, and what I can do with these named states is I can programmatically go to a state just by providing a name. And we'll see there's also a directive so I can link to a state, provide a link that the user can click on to enter the state, just by providing the name of the state instead of the URL. But there are at least two pieces of information that you have to provide for every state, and you do that with a configuration object, which is the second parameter here, and one piece of the information would be the URL that maps to the state, so it would make sense that if we are at "#/dashboard", that we would enter the dashboard state, and if we are at "#/admin", we would enter the admin state. And notice I don't have to put the # in front of those. And then the other piece of information that I should provide here is the template to load when we are in this particular state. In this case I want to go to the templates folder, and I already have a dashboard.html template written, and now stateProvider will know enough to load that in. I will also specify templateUrl of "templates/admin.html" for the admin state. And now there's many features of ui-router that we just won't have time to cover, but I will tell you if there's a feature in routeProvider that you use, that feature is also going to be in stateProvider. So if you want to specify the controller to load, you can also specify the controller here, but it just so happens in my application I'm using an ng-controller directive inside the template to load the controller. You can also specify controllerAs, and you can specify a resolve property which will deliver promises that give data to the controller. We're not going to do that in this module, but it is possible with stateProvider. So anything you can do with routeProvider you can

do with stateProvider. And now, if I save everything and refresh the page, what I'm really just looking for here is to make sure that I don't have any errors. So if I open up the developer tools I don't see any errors. Of course, I also don't see any UI here, and that's because, just like the Angular router I need to have a directive somewhere in my shell page that will work with stateProvider and say, hey, I need to load a view here, what view do I load? With an Angular router that directive is ng-view, with ui-router that directive is ui-view. So I'll place that directive here inside of the Bootstrap column that's on the left-hand side of the page. We will keep our alerts that we built in the last module on the right-hand side of the page. And now I should be able to come back to the application and refresh, and now I can see the Dashboard view, and here's the Admin view, so this looks good. But now inside of the Dashboard view the user can select, essentially a sub-state, and so when they select customers I want a particular view loaded into this view that loads into the shell view. So this is one of the great capabilities of ui-router, the ability to manage a hierarchy of views and nested views. In fact, it can even go beyond that, it can manage multiple main views inside of a single view. We're not going to use that particular advanced feature, but I do want to show you how to set up child views. We can do this in module.js, and we can do that by providing additional states. So I have a state for overall. That would correspond to the Overall link here inside the Dashboard. And I am going to tell the stateProvider that the parent for this particular state is dashboard. And one of the things that I like to do when I have a nested state like this, is I like to indent it over, so it's immediately obvious when I look at this code that Overall is a child of Dashboard. And then just like my parent states, I can specify a URL and I can specify a templateUrl. So I'm going to need to do this for the four child states that are here. There is "customers", there is "injuries", and there is "costs". And let me just fill out the rest of the information for each of these child states and we'll be right back. Now I have the information filled out for each of my child states. So each of these child states has a parent of dashboard, each of these child states has a url, and it's very important to understand that when you specify the url for a state that has a parent, this url is relative to the parent. So in order to navigate to the "customers" state, I would have to tell the browser to go to #/dashboard/customers, so /customers would be tacked onto the end. And when we enter that state, we will be loading customers.html, and just like we had to use a directive to tell stateProvider where to load the dashboard, or really the state service, now we need to tell that service where to load customers.html in dashboard.html. So if I flip over to dashboard.html, you can see there's nothing specific here to one of these views, like Costs, but I do have a right-hand side column here just waiting for another ui-view directive, and that will tell ui-router where to plug in the child view. So now if I save everything, come back to the browser and do a hard refresh, what I don't have in place yet are any links or href attributes on these links that will actually transition me to one of the proper states, but if I go there manually and specify #/dashboard/costs, that does load that nested view into my dashboard view. So now I can see a pie chart, I can see how the restaurant is doing in terms of costs, I can see that if we could just reduce the number of lawsuits we could probably make a little bit more money. So in the next clip let's continue building out this application. The next thing I want the ability to do is I want the ability to click on one of these links and actually have them work by loading the appropriate view.

Linking States

I mentioned in the last clip that one of the nice benefits of ui-router, is that you can think about states instead of tying yourself to literal URL's all over the place, and one place that becomes evident is when I want to provide a navigation link that will send a user to a new state. So instead of applying an href attribute, I will apply a custom directive from ui-router, a ui-sref, a state reference. And all the state reference has to do is point to the name of the state that I would like to navigate to. So overall would go to overall, customers would go to Customers, injuries, of course, would go to Injuries, and costs would go to Costs. Let me save this, come back to the browser, do a refresh, and now I should be able to click on Overall to get the overall dashboard, click on Customers to see my Customers By Day. It's easy to see that Saturday and Sunday are the busiest days of the restaurant. I can look at workplace injuries and see that Inhalations, Lacerations, Contusions, and Muscle Pulls are all a problem, and I can also get to my Costs. Now as I'm clicking these links, it would be nice to provide a proper highlighting because I'm using Bootstrap, and Bootstrap provides a class that can use active. If I apply that active to these different list elements, they will get a more colorful Bootstrap highlighting, and I can do that with another directive that ui-router provides. This one is ui-sref-active, and what this will do is see that you're on a specific state, a state referenced by the inner anchor tag, and if the application is in that current state, it will apply the class that I provide as a value for this attribute. So let's apply the active class if this associated state is active, and if I save that and refresh one more time, now it's easier to see that I'm in the Overall state or the Injury state. So all this is working very well, but what happens if I come to a URL that is not registered with stateProvider? In that case, nothing is happening, so there's a couple of cases where nothing might happen. One is the URL in the address bar doesn't point to a registered state. The other time that a user might experience nothing happen is when they try to go to a state, and there's an error trying to transition into that state, perhaps because the controller wasn't available or the server is offline and the template couldn't be loaded. Let's look at those scenarios next.

Redirects and State Errors

In our application, what do we do if there's a URL that isn't mapped to a state? Well one thing we might want to do is set up a default url for the application to go to if it sees a URL that it doesn't recognize. With the Angular router, this is easy to achieve because there is a method on the routeProvider, it's the otherwise method, and you can give the routeProvider instructions on what to do if it hasn't found a matching route. With stateProvider, there's something very similar, but I do this with the urlRouterProvider has a number of methods that allow you tell ui-router how to interact with URL's in a very low-level way, and it also has an otherwise method, so I can tell it if you are on a URl that you don't recognize, it doesn't map to any of the known states, then just redirect to this URL, /dashboard/overall. And if I save this and refresh my browser, even though I am on #/d, I should be taken to #/dashboard/overall. So that's one thing I probably always want in place in my application, some sort of default route. And the other thing that I always want to do when I'm using ui-router is to have some visual indication where there is a problem getting to some specific route. I'm going to simulate a problem by actually having the wrong templateUrl for the "costs" state, and if we look at what happens right now if I try to get to "costs", well there's no indication to the user that there's an error, there's just _____still in the overall state, and if I open up the developer tools, that's the only time where I would be

able to see that there was some sort of problem loading the template. So anytime ui-router has a problem transitioning to a state, it will raise an event, and any application I build with ui-router, I always want to catch that event and give the user some visual indication that something has gone wrong. Otherwise they click on buttons and links and nothing happens, and they get very frustrated. Not that when things are obviously wrong they don't get frustrated also, but at least the application is admitting to them that something is wrong, and to handle this error I will register a run function with my module. I'm going to do this in a separate JavaScript file, stateChangeErrors.js. It is included on the shell page, but I just want this functionality somewhere else so it doesn't clutter up my main module.js definition. To listen for this event from ui-router, I'm going to inject rootScope and the alerting service into the run function. I'm injecting rootScope because that is where I can listen for that event by using the on method and saying that I want to listen for stateChangeError is the name of the event, and when ui-router raises this event, it will pass a number of parameters to this function. It will pass the underlying event that caused the stateChange, the stateTransition to happen, it will tell you the state that you're trying to get to, the parameters for that state, because just like angular router you can have parameters in the URL. It will also tell me the state I'm coming from and any from parameters, and then last but not least is the error object itself. In this case where a template is not getting loaded correctly, the error would be the http response message with a status code of a 404. So your application can certainly inspect that error object and figure out something smart to say to the user, but in many cases all you want to do is provide some sort of generic error message, like "Could not load", and I'm going to throw in the name of the state that we're trying to get to. Now if I come back to the application, let me refresh. Let's try to go to the costs, and now at least the user can see on the screen that something has gone wrong, we couldn't get to this template. And they'd be able to contact me and give me that error message. I'd be able to at least get a head start on trying to diagnose what the problem is.

UI Bootstrap

Let's turn our attention from ui-router to another project under the AngularUI umbrella, which is UI-Bootstrap. Just like the UI-Router, you can find a link to the documentation and code for UI Bootstrap on the angular-ui.github.io website. And as the name implies, this project provides a wrapper for the Bootstrap CSS framework. One of the nice benefits of using this library is that you can use interactive Bootstrap components like the tabs and carousels, without including the bootstrap.js script or jQuery, because this library has everything you need using pure Angular code. All you need is the UI Bootstrap script. If I go to the site for UI Bootstrap, there is a link here to download the script that you need. For this module when I downloaded UI Bootstrap, I used the version of UI Bootstrap with the default templates included. That means that all the custom directives implemented by UI Bootstrap will have their templates preloaded by the one and only script file that I need to include in the application. There are no additional files needed, other than the core Angular library, and of course course, bootstrap.css itself. And once you've loaded this script and have taken a dependency on the UI Bootstrap module, you can use components like Alert. Typically if you're using Bootstrap and you want to display some alert text on the page, you'll need to create a div and add several classes to it. With UI Bootstrap, there's a custom directive for an alert, and there's attributes so I can specify the type of alert, what should

happen when the user wants to close the alert, and I can specify content that should appear inside of the alert. There is also a component like Carousel, which is a slide show. You can see here in the markup, once again, there's a custom directive, carousel, inside of a carousel there can be a slide, and so I have found that using UI Bootstrap often leads to more readable html code than using just bootstrap.css itself. Plus there's two fewer script files that I need to include on a page. When I'm using UI Bootstrap I don't need to include bootstrap.js or jQuery. And you'll find custom directives in UI Bootstrap for anything that you can do in Bootstrap. Now the one piece I want to focus on in the next video clip is actually not a custom directive, but a service, the modal service. What this service will do is give you the ability to launch a modal Bootstrap dialog, and the user is going to have to do something with this dialog before we can continue in the application. And if I scroll down a bit, we'll see the code that is involved in launching that application. It's done through a service, $modal, and I need to specify all sorts of settings whenever I open the dialog. I have to specify the template, the controller, and how to get data into that controller. What I want to show you in the next video clip is how to not only use this modal service, but how to build a better abstraction for modal, an abstraction that will give you more meaningful code in your application.

Using $modal

The place in the application where I would like to use the modal dialog is in the Admin section, because it is from here where you can promote an employee into management and once an employee has been promoted into management, they don't do any useful work so they disappear from the screen. It might be nice to confirm that we really want to promote someone before they get removed from the list. So over in shell.html, I have already included the ui-bootstrap script that comes in after Angular, and in module.js, the module definition for this application, I have a dependency on ui-bootstrap, and that means what I could do in my adminController is ask Angular to inject the modal service for me. And down here at the bottom of the controller I have the event handler for the ng-click directive for that promote button. You can see it's very simple code, just remove the employee. In this case it's just going to remove an employee from an array. You can imagine that calls back to a web server and ultimately ends up in a database. With the modal service, one way I could approach this is do a modal.open, and ultimately I can get back a promise from modal.open that tells me if the user confirmed this promotion or not. It's a wonderful use of promises, because this is very similar to making a network call, it's an asynchronous action. I need to wait for something to happen after I open the modal, but the problem is when I open a modal I need to specify all sorts of options like the template for the dialog, a controller for the dialog. There's all sorts of code that goes in here, and I really sort of lose the essence of what I'm trying to do, which is simply confirm that we want to remove the employee, and then if we have confirmed, remove the employee. So one of the approaches I take in software design quite often is to write the code that I would like to see and then see if I can make that code work. Inside of this promote functionality, it might be nice to invoke a function, confirmPromotion for this employee, expect that to return a promise, so I can say then(removeEmployee). And that's nice, simple code that's very readable and doesn't have all this noise about dialog options and template URL's and controllers. Let's see if we can make this work. I would expect confirmPromotion to be another function here inside the controller,

or even a service that I inject into the controller. So all this code about opening the modal, setting up the templateUrl, it's all going to be in a different file in a different component in a service, confirmPromotion. I already have a file set up for confirmPromotion, it's included in the page. ConfirmPromotion will be a service, and this is the factory method that I am registering for this service. Now typically your factory methods will return an object, and that object has an API, and that API really defines your service, but in this case I don't want to return an object, I want the service to be a function. So when someone injects this service they get a function, and the only thing they really need to do with that function is invoke it, in this case, passing in the employee that they want to confirm. This service will have a dependency on the modal service. We are pushing that responsibility down into the service. It's from here where I will call modal.open, and I need to pass in some options that describe what this dialog will look like and how it should behave. One such option would be the template to load that forms the url for this modal dialog. I already have one written, it's in the templates folder, it's called confirmPromotion.html, and let's take a quick look at that template. When you're building a dialog with Bootstrap, you have classes that you can use to define the header for the dialog, the body of the dialog, and the footer of the dialog. It's inside of the footer where I will have buttons where the user can click Yes I'm sure or No let's cancel this operation, and currently this template is very generic. It's saying, You are about to promote someone. Are you sure? It would be nice if we actually use the employee's name. So let's use model.employee.firstName, and pretend that we'll get some binding eventually so that I can say model.employee.firstName, and model.employee.lastName, and what this will force me to do is have a property on the scope that is in effect for this view, a property called model, and attached to that property will be an employee property. And it turns out this is very simple to do. Inside of the options for my dialog, I can put a controller in charge of this dialog, and I could name a controller here, or I could provide an inline function, and since this is a very simple function, I'm just going to use an anonymous function. That's also nice because what I can do inside of this function is capture a reference to the employee that was passed into the service function. I'll capture that and attach it to my controller instance, and then if if expose this controller as "model", I will now have a model property on my scope that will point to my controller instance, and my controller instance will have an employee property. Let me save everything, and refresh this page, and see what this looks like. I'll click to promote an employee, and you can see there's a couple of things that happen. Yes, the dialog launches, it looks good, it displays the employee name, but I'm also getting an error, Cannot read property 'then' of undefined, and this is because I really haven't set up anything to process the result of this dialog, as yet. So far I've just been focused on getting the right template and getting some data into the controller so that we can display the employee name. That specific error I was getting that I cannot call a method called then on an undefined object, is because currently confirmPromotion is not returning a promise that I can invoke then on. So what I'm going to need to do from this service is make sure that I return modal.open, and it is .result that will deliver that promise back to the caller. And now once the promise goes back to the caller, I'm saying call removeEmployee if the user has accepted this dialog, and this removeEmployee function expects an employee to be passed into it. So what we're going to need to do is make sure that if the user clicks on the Yes button, we resolve this promise successfully and pass in an employee object. If anything else happens, if the user closes that dialog without clicking yes, we need to reject that promise. And I can actually do this from my template, confirmPromotion.html, because inside a modal dialog I'll have my own scope object that the modal service sets up, which is good, since

I'm adding things to the scope, but also on this scope will be two $ methods directly on the scope object. One is called dismiss, if I just invoke dismiss, that is a way of cancelling this dialog and rejecting the promise. And there's also $close, which will resolve the promise, and when I invoke close I can pass in an object that will be delivered to the success handler for the promise. And that means ultimately, since I am returning a promise from confirmPromotion, and resolving the promise and passing in an employee when the user clicked yes, over here in the adminController if I say confirm the promotion and then remove an employee, I should get my employee object inside of here. So let's save everything and give this a try. Refresh, let's start promoting employees, let's confirm this one, that one seemed to work, let's cancel this one, that was not removed from the list, which is good, and now I have a nice confirmation dialog that I can use on this page, and that dialog is nicely encapsulated inside of a service, so that all this infrastructure code about templates, and controllers, and setting up the model, that's all nicely encapsulated in a function that I pass around as a service, and it keeps my controller code nice and simple and readable.

Custom Directives

When you talk about UI-related topics in Angular, the conversation usually turns to custom directives, so let's look at building some custom directives in Angular. To me, custom directives always fall into one of four general categories. The first category of custom directives are the directives that I write that are just fancy includes. That is, instead of using the ng-include directive to place an html template into the DOM, I'll write a custom directive to load that template, and use the directive in my markup to place the template into the DOM. These types of directives are the easiest to create, and I'll show you an example in just a minute. The second category is what I call a declarative directive. This is a directive that requires some interaction. It will contain elements that respond to mouse-clicks, for example, and it usually offers some sort of API for customization, but most of the work can be done by using existing directives in the template file itself, and this kind of directive is also relatively easy to build. I'll show you an example in this module for this also. The third category is what I call the imperative directive. This directive mostly uses imperative code in the link function or the directive controller, and it's the kind of directive I think is the hardest to write, just because usually when you're starting to write a lot of imperative code in a directive, you're also going to use a lot of angular API's that just aren't used in day-to-day Angular programming, API's that compile the DOM, they set up watches, they kick off a digest, those types of API's. I'll show you an example of this kind of directive in this module also. And just to let you know, many directives can be a blend of declarative and imperative, although I try to lean towards declarative for reasons we'll see in just a bit. And the last category of directive is the form-related directive. These types of directives typically need to interact with the built-in ng-model controller or ng-form controller, and they handle user input and validation. We'll look at these types of directives in the next module when we focus on forms, but let's kick things off for now by looking at a templated directive, what I call the fancy include.

Fancy Includes

Inside the application, the top-level navigation with the brand and the menu links for Dashboard and Admin, that is included into the shell page using ng-include. And it might be nice, instead of having an ng-include here, to have a custom element. Let's call it navbar, because this is what Angular is all about. Although the div is a very useful element, it's also a primitive element for this markup language, and I think it's nicer to have elements that have a real meaning for the application, something that encapsulates your solution, something that encapsulates the user experience that you're trying to deliver, and replacing an ng-include with a custom element is very easy. Since I already have the markup that I need inside of nav-html, all I really need to do is build a simple directive that will load this template into the DOM wherever we have navbar. And it just so happens that this particular template, it doesn't need customization, it doesn't need any data binding, it doesn't need to handle any click events, it simply needs to display the brand and the links. So over in navbar.js, which is a file that's already included into the shell page, I have everything set up to create a custom directive called navbar. Here's the function that returns the directive definition object that's currently empty, but that function is registered as the directive function for something called navbar, and more on naming conventions in just a bit. The only thing I really need to specify here is the templateUrl to bring into the DOM whenever this directive appears in the markup, and this will be under "/apps/ui/templates/", and it is nav.html. So now if I save everything, the application should look and behave the same, but now we've seen just how easy it is to create the custom directive that allows me to have more expressive html. And we've built a directive similar to this before. We built a directive called alerts, and alerts even had to display some data. So I got a hold of the model, which is the currentAlerts in the system, by injecting a service that manages that model, and then used an ng-repeat to display all the currentAlerts. We built that in the last module. So now instead of this simple directive, which we used to kick things off, let's build something that requires some data from the outside, and requires some user interaction.

Imperative Directives

On the Admin page of the application, we're displaying the employee rating, presumably a 1 is the worst rating and a 5 is the highest rating, but what I'd like to do here is display a series of stars like you would see on a some reviews website, instead of displaying a number. And I'd also like to give the user the ability to click here to increase the rating of an employee, click until we reach the maximum rating, and then loop around back to the worst rating. And I don't want to implement this by adding that capability to this admin template in my adminController, I'd much rather encapsulate this inside of a custom directive. And the way I usually start writing a custom directive is just like I write any other class or function, I typically try to write what I would like to see, and then see if I can make that work. So I would like a custom directive rating. I'd like to specify some sort of expression, like employee.rating to set the value for that rating, is it a 1, is it a 4, is it a 5, and it might also be necessary to specify what the minimum and maximum rating will be. So that's what I would like to see. Unfortunately, AngularUI already has a directive called rating, so this actually isn't something that you would need to build yourself, but we're still going to do it because it's an interesting exercise, but it does mean that my

directive rating would conflict with AngularUI's directive named rating, and that's why quite commonly you'll see people namespace prefix their directives. So I'm going to call this otf-rating, and that just means when I declare my rating, the function that returns the directive definition object, I'm going to call this otfRating, but instead of using a dash in the name I will have a capital letter, and that's the naming convention for Angular directives, replace a capital letter with a lowercase letter, but also put a dash in front. And from inside this directive, what I need to do is I need the ability to evaluate this expression. I need to know what the employee's rating is, and one of the easiest ways to do that is to set up an isolated scope. And in this case, my isolated scope, I'm going to say that I want a property value, and I want it bound using two-way binding, using the expression that is in the attribute of the same name of this property. So my property is named value. In the HTML there is an attribute value, there's an expression here. Angular is going to keep employee.rating in sync with the value property on my scope object. Now this is the imperative approach to writing this directive. I think this is the hard way to do it, but I'm doing this to demonstrate some common tips and tricks I've learned for writing linking functions. We all know that linking functions take the scope object, the directive element, and the attributes that are on that element. And when you first get started writing directives, it's quite common to want to jump in here in the linking function and start doing things with the element right away. So eventually what I want to do is I want to do is I want to start appending HTML to my element, some sort of HTML like the following that I'm going to paste in, which will place a button into the element. That button has a span inside that basically has an icon or a star, and we would want to append 1 star for each rating point that someone has. So if the value is 4, we would want 4 stars. So we might put this inside of a loop, but this is a common error I see. What you really want to do is you want to wrap capabilities like that inside of a watch. Because scope.value has this, essentially two-way binding between your scope and the expression that your consumer has provided, and the value can change, so we want to watch that value and react when it changes, and we can do that with a watch, which will tell Angular to call this function we've provided anytime the value changes, and we will get the new value here. And it's inside of here where we can make sure we empty out our element, just in case we have previously placed stars in the element and now it's changing. And then we could set up a loop. So, for i is 0 to less than newValue, and increment i each time, let me paste in the code that I just had, which is appending a button with a star inside. And let's take a quick peek at how this looks. If I save everything, refresh my UI, I now have stars for each rating point. These aren't the prettiest stars, we'll improve this later, but I do have the stars. And now if I click, I want the ability to increase that rating, increase it to the maximum value and then wrap around, and that means I will need to know the minimum and maximum values. What's the best way to get these values? Well, I could set up bindings on my isolated scope, but presumably these values don't change while the application is running, they're sort of business logic constants. So what I might do instead is just pick them out of elements, and I'm going to need to parse them, because everything provided through attributes will be provided as a string. So I'm going to need to say, attributes.min, and perhaps provide a default value here. If you didn't specify a minimum value, we'll specify a minimum value of 1, and I also need to parseInt on attributes.max, and let's make the default value there "10". So now I know the minimum and maximum, now I just need to know when the user clicks on my element or a child of my element, and that's easy enough to do using element.on. That allows me to wire up event handlers, so anytime this element is clicked on, we want to execute the following code. Now I'm going to put a common flaw into this code, and we'll see what the

flaw is in just a moment. Inside of here I just basically need the logic to say, if scope.value is < the maximum value for the rating, then let's increment scope.value. And because I have a binding expression here, Angular will keep that in sync with my parent. Otherwise, I want to wraparound and say scope.value = the minimum available value. Now let's take a look at what happens when I run this in the UI. I'm going to do a refresh, I'm going to click on myself a couple times, and I notice the rating is not increasing. But just for the fun of it, let's go out and promote somebody. And now notice my rating has increased, and that's because of something I forgot to do inside of element.on. I've seen this happen many times, but whenever you say, I'm changing a model vale, but the UI isn't updating, that usually means you're changing a model value behind Angular's back. You're doing that because you are wiring up to a native DOM event, or you are in some way executing code that Angular doesn't know that you're executing that code, therefore, it doesn't run it's digest phase and try to detect changes on the model. The proper way to tell Angular that you are about to execute code that's going to change the model, is to use scope.$apply, and pass in a function which represents your logic. So I just want to take what I had before and wrap it inside of an apply. I need to do that because I'm going directly to this click event. I'm wiring up a native DOM event, I'm not doing it through ng-click, which would handle this for me, but now if I save everything and come back to the UI and refresh, I now have stars that represent the rating value and I should now be able to click to increase the number of stars, and that should wrap around the 1 once we reach the maximum. So that's all good and working, but I've really done this, I think, the hard way, by writing most of my code inside of a linking function. And I did that primarily to give you the two tips of using scope.apply when you are reacting to an event and using scope.watch to make sure you are modifying the DOM when a model value changes. But in the next clip let's look at an easier approach to building this, a more declarative approach.

Declarative Directives

Whenever I start finding myself writing a lot of imperative code in a directive in the linking function, particularly code that watches a model value or handles click events, or any event where there's already a built-in Angular directive, I start to ask myself why I'm not doing this with a template instead, because I template, I've found, often makes this code much easier to write, much easier to read, and often provides a nicer user experience because you can have your HTML and CSS designers come in and work on the file, and it's much easier for them to do that there than it is inside of JavaScript code. So let's define a template. It will be rating.html, and quite often when I have a template I'm also going to have a controller, so let's call this the otfRatingController. And I would generally put this controller into a different file, but to keep things simple, and everything on one screen, I'm just going to add it to the same file. What I'm hoping to do is get rid of my scope.watch, I will use binding expressions in the template instead, and also get rid of this click event handler. I will use ng-click instead, and simply call something on the scope that is set up by my controller. So let's just define that controller real quick. This will be a function, I will expect it to take a scope object for reasons that I'll tell you about in just a bit, but just so you know, that scope object is going to be the same scope object that the linking function gets, the isolated scope that is set up by this directive, and I just need to make sure that I register this, which I can do with module.controller. So there's two things I want to talk about. First of all, why am I taking a

scope in this controller, and that's because I really believe the controller for a directive it serves two purposes. One is the traditional role of a controller, which is to set up the model for the view, and I will do that by attaching things to scope. For instance, I know I'm going to have some sort of click event from my template, and that will need to call something here in the model, so I will have it call the click method that's on my scope object. However, the other purpose of a directive, a controller in a directive, is quite often you want this controller to expose an API, and this API can be available to other directives, and even to the same directive. So what I do is, the API for this controller. that is going to be used from other JavaScript code inside of directives or other controllers, that API I'm going to attach to the controller instance itself, so I will add this initialize method to this, and then there's the model which I will attach to scope, and that keeps these two nicely separated. And then where will I use this controller? Well that's the second thing I wanted to talk about. This linking function, I think its real job is still to link together the scope and the DOM, and so what I want the linking function to do is to dig out attribute values and just generally set things up in the environment so that the controller can then take over. And the controller can take over if I can just get a reference to it, and call that initialize method. So the question is, how does the linking function get a reference to this controller? It does that by adding a require to the directive definition. I'm going to require otfRatingController. Actually I'm just going to require otfRating, so that's asking for the directive otfRating, and Angular will dig out the controller for that directive, which is myself, and pass it in as the parameter after attributes. You can require multiple controllers here, but we just need one, the controller for our self. Angular will grab that controller, pass it in here, I can call controller.initialize, and I can actually tell it what these min and max values are. So now I have a controller in place, let's go over and work on the template a bit. I will still want to display a series of stars, and I will do that by having spans with classes of glyphicon, glyphicon-star, but I want to change the behavior a little bit from what we just had. The behavior I want now is that I want a series of stars that range from the minimum values to the maximum value. So if the rating goes from 1 to 5, I want a series of 5 stars, but if max is 10, I want a series of 10 stars, and then only some of those stars will be filled in. We'll fill in the number of stars up to the value of the rating that we are displaying. And all this means is that instead of having individual span elements here, I'm going to need to ng-repeat on something, or basically from min to max, and the problem with ng-repeat is there is no good way in Angular to tell ng-repeat to just go from 0 to 5 or from 1 to 10. You really have to repeat over a collection. And so what I could do is inside of the controller when we initialize it with min and max values, is we could create a collection. So I could add a stars property to my scope, and just create an array that goes from max - min + 1, so that will create an array of n elements, but there's nothing really in that array. And the next problem I would face with Angular is that when you use an ng-repeat and you repeat over something like "star in stars" collection from my scope, Angular really wants to see distinct values inside of there, so distinct object references, or distinct number values, or distinct strings. We just have an array with empty elements inside, and in order for that to work with ng-repeat, I need to tell Angular to track these elements by index, instead of the values that are inside of the elements. So this will create that series of spans for me, and then I need to figure out if they're going to be empty stars or filled-in stars. What I can do is use the ng-class directive to tell Angular to call a method on my model called styles, I'll pass in the index, and that method will tell you what styles to apply. We'll look at that in just a second, we'll build that. I'm also going to go ahead and add my ng-click directive. I can just call click and pass in the index. And then to make this a bit fancier, let's add mouseover and mouseout

events to color the stars differently if a mouse is over a particular star. So ng-mouseover and ng-mouseout, both of those will need an index, and now I believe my template is complete. I just need to fill in some details in my controller for the click, the styles, the mouseover, and the mouseout. So first let's handle click. I know that gets an index. And one simple bit of logic that I could put in here is to simply say scope.value = that index that you gave me, and I'll add 1. Now if someone sets a minimum value other than 1, we might need some additional logic in here, but I'll leave that as an exercise for you the viewer. I'm just going to try to get something basic working. Let's also set up scope.mouseover. That will be a function that also takes &index. And now what I want to do with mouseover and mouseout is basically remember the state of where the user's mouse is. So let me set up another variable on my scope during initialize. I'll call it preview, and I will initialize it to -1, which means the user is not highlighting anything, but when there's a mouseover I'll just say scope.preview = that &index. Now my scope object knows where the mouse is, and of course, if someone does a mouseout, all they really need to do is when you mouseout of a star, I'll set the preview back to -1, and of course the user might mouseout of one star and mouseover right into the next star, but this logic should work. The last bit I'm going to need is that piece that computes the styles for a given index. Now the way that ng-style directive works is that you can basically return an object, a hash, and Angular will inspect every property on that object. Property is truthy , Angular will take that property name and add it to the class attribute for that element. So by saying glyphicon true, I can be sure that glyphicon will always be a class on each span. Now I might also want glyphicon-star to be present, but I only want that to happen if the current index is less than scope.value, and yes I have a real problem spelling glyphicon. But remember, this scope is the same scope that we're using in the linking function, it's our isolated scope, and this value is bound to an expression that the outside view has given us, so we can assume that will point to the employee rating in the scenario. So this will fill in the stars up to the rating value. I also want glyphicon-star-empty, for stars that are above the rating, so that would be where index is greater than or equal to scope.value. And then to provide the little preview effect, I already have a class defined in site.css called starpreview. What this will do is set a slightly transparent yellow color for any DOM element where that class name appears. So let's set up starpreview to be there anytime index is less than or equal to scope.preview, which is the property that we're using to track mouseover and mouseout. So with all that in place, let me save everything, come back to the application, do a refresh, and now I can see that everyone has 5 stars, but not everyone has five stars filled in, because not everyone has a rating of 5, but let's give someone a rating of 5, and we can see that our mouseover effect also works, and now I just need to click on a star to change a rating. And the code that I have right now I think is a lot simpler than the code that I had before. The code that I had before had to worry about using scope.apply, it had to worry about watching model values. All of that has really been stripped away now, it's handled by binding expressions. So anytime the value changes, our classes will be reevaluated, and anytime the user clicks, we'll just call a simple method on our controller. And this is how I would prefer to implement this type of directive.

Summary

In this module, we looked at a variety of UI-related topics, including how to use UI-Router, UI Bootstrap, and my take on building a few types of custom directives. However, we really aren't finished with the user interface or custom directive topics, because in the next module we'll focus on a specific type of UI that we come across in most web applications, the UI that uses a form to accept and validate user input.

Forms Playbook

Introduction

Hi, this is Scott Allen, and in this module we'll look at using forms in Angular, including how to work with a form in ngModelControllers, how to dynamically compile markup, how to use Bootstrap validation classes with Angular validation, and also have a general review of how to program forms with Angular, just as a refresher. Along the way, I want to evaluate some interesting trade-offs that I often see in Angular applications, trade-offs related to the complexity of forms. Do you want that complexity to live in your forms, or do you want to centralize complexity into some directives and apply some simple conventions that everyone on the team must follow. We'll look at topics like these and more in this module on forms.

Forms Are Hard

When I tell people that I think forms are complex, they look at me funny, because what's so hard about a input type = text and a submit button? Which I agree, forms always start off simple, but then the business wants to add rounded corners and cute icons to make the forms look friendly and inviting, and then the real complexity starts when the business starts adding validation rules to a form. Not because validations are hard or complex, but because everyone wants validation to behave differently. Some people want wrong things to turn red and right things to turn green, but other people only want wrong things to turn red, and then only turn red after the user has had a chance to enter information into the form or click a button, and only the error message should turn red, the input itself and the label should appear normal. So over the years I've seen a wide range of opinions about the best user experience for forms and validation, and I'm sure some of you might find what I do in this module offensive from a user experience standpoint, but I don't want you to forget the over-arching goal of this module, which is how to work with forms in Angular. And I'm not trying to give you directions on the perfect user experience, because everyone things differently about how that should work, and it really depends on your target audience. What I want to show you is how you can use the features that are built into Angular to achieve what you want by giving you some examples of implementing different behaviors, like making messages appear, and labels turn red when validation rules are failing. We want to do that in a consistent manner to keep our markup simple, and to do this we need a good understanding of how Angular works with forms. Let's have a refresher next.

Angular and Forms

In the application for this module, I ultimately want to build a form that allows a user to fill out information about themselves. So I want them to be able to tell me their favorite color, their preferred username, some biographical information, and some of this information will have validation rules that we want to apply. Right now there's just a very simple form with a single input so far. It's a place where the user can enter their preferred username. And back here in my HTML you can see I already have a form in place, I have an input type="text", and this is a good starting point for us to review some of the things about how Angular works. Notice this view has a controller in charge, it's the editProfileController. It's exposing itself as model, and that means, if I take a quick look at the editProfileController, that anything that is attached to the controller instance is available for binding. So the first thing that is attached to the editProfileController is this user model. This is the object that I want to contain all the information that is in the form, the favorite color, the biographical information, the username, and there's also a submit method. When the user clicks on the submit button for the form, this is the method that I want to invoke. Typically this is where you would take your model and send it back to the web server and then save it in a database. But to keep things very simple for this application, when the user submits the form, I'm simply going to use the location service to jump over to the results route, and the results route is going to take the same user information and just dump it out into a page, and we can look at that information and be sure that what the user entered into the form is actually making it into the model object. How can the editProfileController and the controller that is in charge of results both see the same user information? That's very simple, because I have that user model defined as a service. This is the factory method for that user service. It's just going to return the model object that I need. It has properties available for me to bind the username and the email and the website and the favorite color, and so forth. So I just need to inject user anywhere that I need it, and I'm getting a single object because services in Angular are singletons, and then I just need to set up to bind the inputs of my form against properties of this user model. And I can do that using directives that you're probably familiar within Angular, like ng-model. I can say that this input should bind against model.user.username, and that establishes the two-way data binding that Angular is famous for. Anytime the user types into this input, those changes will be pushed into that expression, so it will set the username. And if for some reason the JavaScript code changes username behind the scenes, those changes will be propagated into this input. I can also use another directive, which is ng-submit to say that when the user submits this form, please call that submit method that we saw on the controller instance, and that will send me over to the results page. Let's try that out real quick. If I save this file, do a refresh, type in some username, and submit, I end up on my results view where I'm just dumping that user object into JSON, but I can see that I did successfully set the username there. And those are some of the basics that everyone pretty much knows about Angular. You can use ng-submit and ng-model. But what's not immediately obvious to a lot of people, particularly if you're just starting with Angular, is that Angular has a directive register that will tap into the form tag in HTML. So anytime you have a form on the page, there's a directive with a linking function that attaches some behavior to that form. And specifically one of the things Angular will do when you have a form, is that Angular will build a data structure that contains information about that form. For instance, is it dirty, is it valid, what errors are inside of the form, and if I want to see that data structure, all I need to do is give this form a name. And

what Angular will do is take that data structure about this form and attach it to the scope using this name. If I wanted to attach it to my controller instance, I could say model.profileForm, and Angular would attach that to my controller, which is aliased as model. But in most cases, I don't want the controller to know anything about that data structure about the form, I just want it to be available. And let's come down here in a column that will appear to the right of all my form inputs. I just want to take this profile form and dump it out as json so you can see it. So let me save everything and refresh this page, and now I can see that data structure. It has no errors, its name was profileForm, it's not dirty, it's pristine, it's valid, not invalid, and the user hasn't submitted this form as yet. If I start typing information into this text box you'll see some of those values change, specifically, this form is now dirty and not pristine. Now although this data structure is useful, it's even more useful if I can attach information about each of my inputs into that data structure, which I can do. Any input that has an ng-model directive, if I also give it a name, Angular will also give me information about this input as a username property on that data structure. So let me save this file again and refresh, and now I can see that in addition to information about my form, I now also have information about the username input. I can see that it has not been touched, that it's pristine, that it's not dirty, and if I type into it, I can see things start to change, like the view value and the model value. And all the information that is inside of this data structure, ultimately what we want to do is take advantage of this information, to drive the user experience, to show validation warnings, to hide validation warnings, but before we get there, let me show you what the whole form looks like. If I uncomment some HTML that I have here, then I just want to show you that I already have inputs set up and bound with ng-model, to gather the user's email address, the web citing, their favorite color, and all sorts of information. Let's take a quick look at this in the page. And I can see the form is quite difficult to use. Let me try to fill out some basic information, maybe even pick my favorite color, say that I'm a professional and I accept these terms, and submit this. All along you can see information in the data structure changing, but this all works, I can bind each of those inputs to something that is in my model. Unfortunately, this doesn't look good or behave well. If I take out my email address, currently I'm getting the native browser validation messages here, Please fill out this field, that's from Chrome. So let's just try to address the looks of this form first. I'm not particularly happy about the way it appears. We'll try to fix that in the next clip.

Custom Styling Directive

To change the appearance of our form, that generally requires some CSS, and fortunately Bootstrap already provides some CSS classes to make form inputs appear better. If I go to every div that wraps an input and I give it the class form-group, that's going to provide some nice spacing. If I apply the class "control-label" to every input label and I apply the class "form-control" to every input, let's come back and see what that looks like. I've only done that for the first input, but you can see it makes it a bigger, friendlier input. There's a little more margin and spacing in my form. And what I want to do is have this look for all of my input. So that means I have to go through everything on this form and add these three classes. And that adds a little bit of noise and extraneous markup. I would really like to just stick with the essence of what I want, which is, here's the label, here's the input, and not get caught up with so many low-level details inside of my form by adding all these classes. And we really haven't added all the

markup that we need yet, because we don't even display our own custom validation messages as yet. So instead of going through and adding classes to every div that wraps my label and input and class to every label, and a class to every form-control, what I would like to do instead is use something like a custom directive, let's call it forminput, and have that make these changes for me. I also want it to apply validation styles, I want it to show validation error messages, I want it to do all sorts of interesting things, so I can simply say div forminput, put in my label, put in my input or even my custom directive like the otfRating, and just have these things work, so let's try that. I already have forminput on every div that contains some sort of user input control. And I already have a file added to my shell page that will register a function that returns a directive definition object. This is called forminput. Let's restrict this particular directive to just being an attribute. And all I want this directive to do so far is just to add those three classes, which I could certainly do from a compile function, because in a compile function it's very easy to manipulate the DOM. But ultimately I know that this form input needs to work with validation messages, and to do that it needs access to the data structure that represents the form that it is inside of, and to get access to that data structure I need access to the current scope object, so I'm going to write a linking function instead of a compile function. And instead of writing an anonymous function as part of the directive definition object, I'm going to write a linking function up here and it takes scope, it takes element, and just to break things up further, I'm going to have a dedicated method to set up the DOM for a forminput. So setupDom is a function that takes the element wrapped by Angular's jQLite implementation. You might have used that before. That allows you to do things like add class and remove class, and search for child elements. Unfortunately, the CSS selector syntax with jQLite is not very robust, and it's not nearly as robust as what comes with the native browser API's these days, API's like querySelector. So I'm actually going to dereference that element, use element sub 0, and pass in the raw DOM element object. That will allow me to do things like search for inputs by using element.querySelector, and here I could have a relatively fancy selector. I want to look for all inputs, as well as textareas, as well as select elements, and even my custom otf-rating element. Because once I find the input that is inside, I want to go and add to the class list, the class form-control to that element, and unfortunately it's not quite that simple, just because this Bootstrap class, form-control, should not be applied to inputs of type checkbox and radio, it just makes them look strange. There's different classes that you can apply there. So I really need to find the type of this input, and again, there's a DOM API for that. I can ask to get the attribute type, and then I can have an if statement, if(type !== "checkbox" && type !== "radio", only then do we apply this particular class, and you could certainly have else branches here to apply different styling for your checkbox and radio. So that takes care of form-control. I also want to find a label associated with this forminput. So let me go to element and querySelector for a "label" that is inside. We'll tell the "label" to add to its classList, "control-label", that Bootstrap class. And finally the element itself, it's a class of form-group, and that's it for this directive. There's really no manipulation of scope, everything so far is just adding CSS classes, and if I come back to my view I can see everything now is styled a little bit nicer. It may not be the preferred look that you want, but again, we're not trying to build the perfect form, I'm just trying to show you techniques that you can use to manage complexity and keep your html a little bit cleaner. If you wanted this to look different, then you can certainly define your own classes, your own colors. But now that we have this in place, what I'd like to do next is start addressing validation errors. For instance, email is required, how

can I give the user something additional beyond just a red border or my input. We'll see how that happens, as well as make that a little bit better, in the next clip.

Custom Message Directive

If you've done anything with form validation in AngularJS, you probably know that this data structure that Angular attaches to the scope, and that has information both about the form and all of the inputs inside, this data structure becomes very important, because for example, currently I don't have any validations on username. And if I go out and search for username in this data structure, there is a property attached to the data structure that represents the input, and I can look at the $error property, which is a hash object, and I can see it's empty, which means there's currently no errors associated with that username input. But I want to start adding validations using html attributes like required, min, max, pattern, and so forth, and I want to be able to handle the display of error messages on my own. In other words, I want to tell the browser not to use its built-in validations on this form, that's the purpose of the novalidate attribute. That way Chrome will not pop up validation messages and instead it will be my responsibility to pop up the error messages. Let me refresh the page with these changes and do another search for username. And now I can see in the property that username does have an error. The current validation that is failing is the required validation. I know that because the property is there and it has the value of true. So how can I tell the user that the field they're ignoring is currently required? Well one thing I can do is add some color to the input. You can see the border turn red, both the username and email addresses, they are required. And this will happen just because Angular will add some specific CSS classes to input that will describe the state of that input. So the classes associated with the username will tell me that it's currently invalid, and it's invalid because it's required, meaning the user hasn't typed anything here as yet, but if I at least put one character in there I can now see that this input is touched, it's dirty, but it's also valid. And I can use those classes to display this red border, that's something that I already had setup from a previous module. It was set up in my style sheet. If I open up site.css, one of the first rules I have here is that if you have an input that is touched, that is invalid, then please display a 2px solid red border, and that has a CSS class that will add the red here, but it won't add that red until after the user has interacted or touched that particular input. But it would also be nice to have a textural description of what is wrong, and for that one of the things I could use is the new ng-messages feature. This was introduced with Angular 1.3. I'm going to add a div here with a class of "help-block". That's another Bootstrap class commonly associated with forms. This is a help-block for this particular forminput, and it's inside of here where I will use ng-messages, and I have to tell ng-messages what to look at to determine when there is an error. So for this particular input, I would want it to look at profileForm.username.$error. That is that property that we were just looking at where Angular will add a property to that object, require true, whenever this input is failing a required validation. And this is true for all the other HTML5 validation attributes too. If I wanted input type number, and I specify a min and a max, those will be added to that error object, and now what I can have are divs that are a specific message for one of the validation properties. So div ng-message=required is a way of saying, please show this div when this property appears on this object. And now if I save everything and come back to the page and refresh, that text is going to appear right from the start. Now I know a lot of people don't

want to show any error messages until after the user has had a chance to interact with this form, and that's something that you could do with _____ star rules, specifically if you use CSS and use classes that Angular adds to elements, like ng-touched and ng-submitted, that will give you the ability to control DOM elements and when they appear. So you might only want an error message to appear after the user tries to submit the form or after they have tapped out or touched a particular input. What we're going to do here is something a little bit simpler, which is just to always show these messages, and we'll worry about coloring them in just a bit. But again, we are in a situation where I would have to take a div like this with ng-message inside and copy it for every form input that I have that requires some validation. And fortunately, angular makes this a little bit easier. Instead of specifying each individual message inside of ng-messages, I can use another attribute value here, ng-messages-include, and tell Angular about a template that contains all of the messages that I might want to display. Let me open up a template that I already have built. It is inside of here where I already have messages defined to tell Angular what to display if we are filling a required validation or a min validation, or a max validation, or if an input type = email doesn't look like it has a valid email address as yet. And I can just reuse all these messages by saying please include "templates/messages.html". But I'm still not looking forward to having this markup inside of every single div that has a form element inside. So, once again, I want to turn to that custom directive forminput and see if it just cannot add an ng-messages block for me. And fortunately, this isn't too difficult to do. Inside of forminput, after I have set up the DOM, I want to add messages to this particular element, but in order to do so I'm going to need some additional information. Essentially what I need to do is dynamically build this div that has the ng-messages directive, and in order to do that properly I'm going to need to know the name of the form, the name of this input, and then build a string that has form.inputname.$error at the end. So getting the name of the input is relatively easy, I already have access to the DOM element, I just need to ask it for its name attribute. Figuring out the form name can be a little bit trickier, but fortunately that can be easy if I just require the form controller. So remember, require is a list of the controllers that you want passed into your linking function, so now I should be able to get scope, element, attributes, and the form controller. So when I add messages I'm going to pass along this form, the element that I'm working with. I also want the name of the input. Now since setupDom has to find the input, it might be easiest for it to get the name of the input and return that to me. If I just say var name = input, please get me the attribute name, I can return that value from this function. So let's build out addMessages. That will be a function that takes the form, the element, the input name, and what it needs to do is essentially build this html string. Instead of me trying to type that out, I am going to piece that in. So the markup that I want, is I want a div with the class of help-block, with an attribute ng-messages =. I can get the name of the form from the form controller, it has a $name property, then do a "." and the name of the input, and then ".$error". Now the tricky part is that I cannot simply do element.append on these messages. I could do that if this was just plain simple html, but because this markup contains directives, I'm going to need to use the compile service and have it compile this html, associate a scope with this html, and then it's the compile service that will give me something back that I can append to the element. But now we're going to need access to the compile service. The easiest way to do this, well the only way to do it in this scenario, is to inject the compile service to my directive function. I'm going to need to pass it to the linking function, and I'll do that by invoking the linking function. What I will have this do now is, instead of being a function that I execute immediately, I will now have it take compile and return the function

that we had defined before, so the function that takes scope, element, attributes, and form, that's the real linking function here. What I'm doing with this outer function is really building that linking function and forming a closure around $compile so I have access to that. Let me also pass along $compile. And when I compile I have to compile against a specific scope, so that just means addMessages will now take compile and scope, and what I will append is the return value of compiling messages and invoking the function that compile gives me back against the current scope. So if you've never used compile before, it can be a little bit crazy, but essentially I need to compile this markup because it has directives inside. If I just append that markup into the DOM, those directives aren't going to be active, and the way you use compile is to compile a particular piece of markup that will give you back a function, the linking function that you need to invoke with the scope object. And once you have associated your compiled html with a specific scope, then it's time to append it into the DOM. So with all this in place, let me save everything, and come back and refresh our page, and now all I need is a forminput directive and I will automatically get validation messages, or all the different validation scenarios that I have inside of that template. However, they don't really stand out. So let's see if we can apply some Bootstrap validation classes to make messages appear in green or red. We'll do that in the next clip.

Custom Coloring Directive

Bootstrap.css includes many classes that you can use to style forms. Two of those classes are named hasSuccess and hasError. The idea behind these classes is that you can apply them to a DOM element, and inside the element those classes will change the appearance of label text and help text to signify if something has an error or if it has success. With the theme that I'm using, things should turn green when we have success, and things should turn red when I have an error. And I would like my forminput directive to apply the correct class for a given DOM element depending on the state of the form input that is inside. And in order to do that, my directive is going to have to look into the data structure associated with the form, specifically, it's going to have to look at one of the properties associated with an input, like username, and look at a flag like invalid to see if this is true or false, and if $invalid ever changes, my directive is going to have to know about that and make changes to the DOM. So, typically, when you're inside of a directive and you need to know if something changed so that you can update the DOM, you use scope.watch. That's what we can do here. Inside the linking function, I will tell the scope to watch something, and ultimately what I want to watch is form sub name.$invalid, But instead of specifying that inside of an anonymous function here in scope.watch, I went to call a function, I'll call it watcherfor, and this function will return a function that I could give to scope that will establish the watch. So I want a watcherFor, this form, and specifically this name inside the form. So this is another way to use scope.watch. Quite commonly you'll see people use scope.watch and they specify an expression as a string for the first parameter, but you can also specify a function with the first parameter, and whenever the return value from that function changes, Angular will invoke the second function, which typically applies the DOM updates or exercises whatever logic you need to do when something changes. We will get to that one next. For right now, let me build watcherFor. That is a function that takes a form and a name, and it has to return a function that I give to Angular. So inside this function, let's make sure that we have a name, and that this form that I can index into it by name, or

that the form has a property with that name, and if it does, I will return form with that name and the invalid flag. Otherwise, just return undefined, that's fine. All I really care about is if the return value changes, because then Angular will call my second function, which I'm going to use a similar technique and say, build me an updater this element, and now updaterFor has a similar pattern. It's going to return a function that I pass into watch. So updaterFor is a function that takes an element wrapped with jQLite, and returns a function. There will be a value passed to this function. Angular will pass whatever value the watcher returns into this function whenever it changes, so essentially I will be getting a flag that tells me if the input has an error. So if we hasError, I will need to add the appropriate Bootstrap classes to indicate an error, otherwise, I need to indicate success. So on the element let's first make sure we remove any hasSucces class here, and then addClass that indicates there is an error. Otherwise, we just do the reverse, I need to remove any hasError class and add a hasSuccess class. So to summarize that bit of code, what we've done is we have set up a scope.watch. This watch is going to look at the invalid flag for a given forminput, and whenever it changes we will call the updater function which will apply one of the appropriate Bootstrap classes. Let's save everything, come back and refresh our page, and I can see right off the bat that the label and help text for username is red, because it's a required field and there's nothing there. Same with email address, but if I start typing inside of here, not only does the message go away, but my label text turns green, and that's a good indication to the user that things are working. So even though we used Bootstrap, this is an approach that you can use to apply your specific classes to the DOM elements that you want to style depending on success or failure of validation rules. Now there is one area of our form that doesn't behave correctly. We'll look at that in the next clip and also fix the problem.

Using ngModelController

If you come into this application and start playing around with the different input controls and watch the data structure for the form on the right-hand side, you'll start to realize that one of these inputs is not like the other. Specifically the rating directive that we built in a previous module doesn't integrate with the form at all. If I look through the data structure right now I would not find an entry for rating. If I change the rating on a pristine form, the dirty flag doesn't change. And all of this is because when we implemented this rating control we really didn't have a form in mind, but quite often this might be the type of control or type of directive that you do want to have integrated into a form. You might want it to work with validation rules, you might want it to turn the form dirty, and so what I'd like to do is show you how to update our rating directive so that it works with a form and more specifically, with ng-model controller, because if you want a directive to integrate with a form, you'll either need to talk to the form controller, but more likely you'll need to talk to an ng-model controller, because it's really ng-model that's responsible for registering the different inputs that are inside of here, like username. So let's swing over into our code, and the first thing I'm actually going to change is the markup, because currently we are using an attribute value to establish the two-way data binding between what we're displaying and what's going to be in the model, but it would be more appropriate on a form to use the same directive that all the other input controls use, which is ng-model. And now what my rating directive will have to do is talk to the ng-model controller, and that just means down here in my require

statement, in addition to otfRating, I'm also going to need ng-model. This should be on this same DOM element. And I'm going to rely on ng-model to get and set values so I no longer need an isolated scope with a value inside, I still want an isolated scope, because I'm going to be adding things to it, I don't want that to interfere with the parent, so I'll still have an isolated scope, just no automatic two-way binding between myself and value. Instead I'm going to rely on ng-model controller to do that. And now since I have two controllers listed in my require, I no longer get a single controller to my linking function, I get an array of controllers and I will have to dig these out. So the ratingController will be controller sub 0, because that is the first controller that I ask for and require, and the ngModelController will be the second controller in that array, so sub 1, and now anytime you have an otfRating it will require ngModel. And down here, where I used to say controller.setRange, this is really ratingController. And I need to get ngModelController into my ratingController, so it might make more sense to call this something like initialize instead of setRange, and pass in ngModelController. So now let's make changes to the code above based on those changes. Instead of setRange I have an init. This will take ngModelController, and so that I have access to that throughout all the other methods that I have defined here inside of the constructor function for the controller, let me just declare local variable ngModel, and I will save that, ngModel = ngModelController. So what do I have to do with ngModel? Well, one thing I need to do is I need to set up a render method. I actually need to attach that to ngModelController, and I'm going to point it to a function that is a member of my controller. So this.render is a function, and the whole idea behind render is that Angular is going to call render at the appropriate times when you need to take a value from ngModel and make sure that it is properly rendering in the DOM. Now quite often that means DOM manipulation, but we have everything set up in this directive to work off scope.value. So it's scope.value that determines what stars are lit up, what stars are empty, and so all I need to do inside of a render method is simply say that scope.value = ngModel.$viewValue. So ngModel will hold the value that needs to appear in the view, I just need to ask for it by going to $viewValue and putting it into my scope or manipulating the DOM so that it appears on the screen. All I have to do here to light up the stars is just set scope.value. And on the other end I need to update that viewValue when the user changes something, when they click on something. So instead of setting my own scope.value, that's going to happen during rendering, I'm going to tell the ngModelController to set a new viewValue that is equal to $index + 1, and I can tell it that this has been touched. And in this case I can also force ngModel to render, and push that into my own scope. So let's save all this and come back to our form. I do just want to point out that if I do a search for rating, it doesn't appear in the data structure anywhere. I only have one match, and that is the label here, but now if I refresh the application I now have rating, I can set that rating. If I go looking in the data structure, I will now see that rating appears here. I can see it has the proper viewValue, the proper modelValue, I can see it has been touched, it's no longer pristine, it's dirty, and that all happens just because I'm interacting with ngModelController in the appropriate fashion. If I have validation rules associated with the rating, I could also have properties up here on $error here, and things would turn valid and invalid. But since we're talking about validation rules, let's take a quick look at building a custom validation directive next.

Custom Validation Directive

AngularJS provides built-in directives to handle all the HTML5 validation attributes, attributes like required and pattern, as well as input types like number and email. I'm not going to cover those in detail in this course, because there's at least a half dozen other videos that cover them, but I do want to throw in a quick custom directive, because it is easy, and also uses the new approaches of Angular 1.3, and we're using 1.3 here. Let's say as an example that we want our username input to start with a specific character. This can be as easy as coming into the input for username and adding an attribute which will be a directive, starts-with, and let's say I want the username, it has to start with a capital "S". Now I can use this attribute anywhere I want once I write the directive to implement this attribute, and that's one of the advantages of putting some validation logic into custom directives. That validation logic becomes reusable. Now I already have a startsWith.js file, it's included in the shell page, and inside of here I'm going to use module.directive to register a new custom directive, startsWith, and this will return our directive definition object. Since I expect this to be used as a validation attribute, I'm going to restrict this directive to being just an attribute, and because I need to do validation against a modelValue, I'm going to require, again, the ngModelController. And that means I can write a linking function that takes scope, element, attributes, and the ngModel controller. And one of the first things I'll need to do is discover what is the value of this attribute? Is it an A, is it a B, is it a C, and that's easy enough to dig out of the attributes. So I will ask for attributes.startsWith, and now the way to write a custom validator is to walk up to the ngModelController and go to the validators property and add a new method to that object that matches the name of your validation attribute. So ngModel.$validators.startsWith is a function, and because this is registered as a validator, Angular will automatically pass in a modelValue to me every time the validation rule needs to be applied. And inside of here I can simply check, do I have a value and does modelValue _____ sub 0 so the first character, if it's not equal to that value, I will return false, otherwise, return true. And that's all you need to do for a simple, straightforward, synchronous validation, just return true or false. You can grab parameters from other attributes that are on that DOM element, and then just add your function to $validators. The other thing you'll probably want to do if you're using ngMessages, is to add an error message that is associated with this validation rule. So I can say ng-message for "startsWith", and I have to come up with a somewhat generic error message here since I don't know exactly what the parameter is. I'll say invalid starting character, and what would probably be useful for that input is if I have some additional help text that only appears when there is an error, that describes to the user the constraints that are on that particular input, so it might tell the user that it has to start with an S. Well let's give this a try. I've saved everything, let me refresh, and of course the first error that's going to come up is that this field is required. Now if I start with a capital T I now see that error message Invalid starting character, lowercase s, same thing, uppercase S, that works. And that's just how quick and easy a custom validation rule can be, but what if you need to do something a little more complicated, something that requires server-side logic, for instance, making sure this username is unique. That's when you can write an asynchronous validator and make calls back to your server to determine if the given input is good or bad. Let's look at that next.

Asynch Validation

In the application, let's write a custom directive that will take the value in the username input and send it back to the web server to see if the web server thinks the username is valid. I already have an endpoint set up on the web server, it is at /api/namevalidation. All you need to do is pass the name as a query string parameter, and after a 2 second delay, the API will tell me if that value is valid or not, it will just return a true or a false in JSON. And currently, to keep things simple, if you pass a username that is less than three characters, the service will return false, but if you pass something that is more than three characters it will return a true. And the way I'd like to perform this validation is by writing a custom directive, let's call it username. Any input that that is attached to, it will be taking the value and sending it off to the web server. So I already have the shell of a username directive ready to go. All we need to do is set up the directive definition object, and just like our startsWith attribute I'm going to restrict this to being only an attribute, and I'm still going to require the ngModelController, and that means I will have a linking function that takes scope, element, attributes, and ngModel. However, unlike the last directive that we wrote, instead of attaching something to $validators, which are always synchronous, I'm going to attach something to $asyncValidators, but once again, it will be a property that matches my directive name, and it's going to point to a function. In this case, instead of writing it as an anonymous function, let's come out and write a named function, validateUsername. Just like with asynchronous method, Angular will call this at the appropriate times, and it will pass in the value that needs to be validated. But what's going to be a little bit different about this function is that instead of returning true or false, is this valid or invalid, I need to return a promise, because we're not going to know the validation result until sometime in the future. And in this case I need to call the server to perform the validation, so I'm going to inject http and return a promise that is based on http.get, plus a little bit of processing. So I now need to go to "/api/namevalidation", and I need to pass the name as a query string parameter, so it might be good to use encodeURI on our value. And then I'll do a little bit of processing on the response, because instead of returning true or false, is this valid or invalid, I need to return a promise. And if I resolve the promise, I'm telling Angular that the value is valid. If I reject the promise, I'm telling Angular that the value is invalid. So that might be a little bit different than what you typically do with an HTTP call, because even if the HTTP call is successful, if it returns a false, I need to turn that false value into a rejected promise to make sure that this model value is marked as invalid. So basically if I look at the response.data property, if it is true I want to make sure I resolve this promise. So let me bring in the q service and explicitly return a q.when with true. So that's a way of saying, give me a promise that will automatically resolve and deliver the value true, otherwise, I want to use q to return a promise that will be rejected and deliver the value false. Let me save that file and just come into our messages template and add one more error message here. So this will be an ng-message for username, and we can have a generic message here like Invalid username. Let's save all this and try it out. I'll refresh the page, type in an S. You can see there's a pending http call, and I get back Invalid username, but if I type Scott, that works, and that is successful. Now other than our generic indicator for an outstanding request, is there another way to give the user some feedback that we are, in fact, doing something with this username? And the answer is yes. If I come over into our page, then let me just write a little bit of HTML here. What I want to do is have Checking username appear, but only if we are actually actively performing a validation on a username, and fortunately that is easy to do with Angular. All we need to do is look at profileForm.username, and there will be a $pending property on that object, if there is currently an outstanding asynchronous validation being done on this field. So if I save this and

come back, let's refresh one more time, I'm going to type in Sc, you can see there's Checking username that appears, and then it goes away. It's invalid. Let's fill it out the rest of the way. We're going to check it, and now it's complete. So just like I dynamically added ng-messages into my view, I could also dynamically add this ng-if that checks the pending flag, and I could do that wherever the username attribute appears, let's do that. Down here in the directive definition, let's have a setupDom function, and what I'm going to need is I'm going to need a reference to the input element, a reference to the form, because we're going to need the form name so I can build an expression like profileForm.username.pending. And I'm going to need the scope object because we're going to do some dynamic compilation just like we did earlier in this module when we dynamically added ng-messages, so let's inject the compile service. And inside of here I could say the input name is this input element, let's get the attribute name. The formName is very simple. If you give me a form controller, I can simply do form.$name. We've also done that previously in this module. And then I need my html that I'm going to append into the page. So you don't have to watch me type that out, because that can be painful sometimes. I'm going to paste something in, but it's basically the same HTML that we just looked at in home.html I'm just building it dynamically now. So it's a div with an ng-if that looks at the formName.inputName.$pending. And now I just need to compile this and append it to the input parent. So let's go to the input, get its parent element, and do an append on $compile against this pending markup, and you have to compile against a scope, or rather execute the function that $compile returns against a specific scope. So now I need to call setupDom, and I'm going to have to change a few things around. In addition to the ngModel controller, I'm also going to need the parent form controller, and that means my linking function no longer gets just ngModel, it's going to get an array of controllers, but ngModel will be controllers sub 0, the ngForm controller will be controllers sub 1, and I can now call setupDom and pass in my element, which is the input, pass in the ngForm controller, and pass in the current scope. Let's give it a try. I'll save all files, let's refresh, type in an Sc. I can see there's Checking name, it's Invalid, I'll fill out the rest of the username, and now we're valid. And once again, we're taking advantage of directives as heavily as possible to take care of all the boilerplate code that usually appears in a form. I have all sorts of validation messages and text that appears when there's an asynchronous validation, and all of this happens without me explicitly typing in all of that HTML for every forminput. I just have forminput directives, username directives, they're going to dynamically add things into the DOM when needed. And although this form might not look like the exact form that you want, hopefully this module has given you some ideas about how you can write directives to build the forms that you need for your application.

Summary

In this module, we looked at forms and form validation, and the key takeaways were how you can work with ngModelController and ngFormController. These controllers expose an API that allow you to add validation rules, async validation rules, move model values around, and overall integrate tightly with Angular to make forms work the way you want them to work. The other takeaway was how to use directives to customize your forms and keep them consistent in a project. It doesn't matter what colors you choose or what your validation messages say, you can always use directives to achieve the behavior

that you need and then make that behavior reusable, both inside of a form and across all the forms in an application.

Integration Playbook

Introduction

Hi, this is Scott Allen, and in this module we'll be looking at strategies for integrating other libraries into an Angular application, including jQuery plugins and Physics simulators. Once again, I want to highlight some general strategies and tips, but this time for making two code bases work together. So join me for this module where I answer questions like, how should you approach integration problems, and what should you watch out for?

PhysicsJS

PhysicsJS is a fun script library that gives you a Physics engine in the browser, and that includes acceleration, vectors, polygons, and of course, collision detection. I wanted to show you how I'd go about getting PhysicsJS to work with Angular, because first of all, it's one of those libraries that is outside the neighborhood of where AngularJS typically lives, that's the neighborhood of business applications and forms and inputs. But also, because PhysicsJS has some pieces that once you know Angular well, you'll see exactly how to componentize and adapt an API to make it work well from Angular. I don't want to spend too much time focusing on exactly what PhysicsJS can do, but let's try to replicate one of the tutorials in Angular, specifically, there is a tutorial on how to create a scene of colliding polygons. And once we have it working, we can extend it a bit and make it interesting. Here in just the first two steps of the tutorial a couple things jumped out at me. PhysicsJS exposes a global symbol by the name of Physics. This is the gateway to many pieces of the Physics API, just like Angular is the global gateway to Angular, and $ is the global gateway into jQuery. Quite often when a library exposes a global, like PhysicsJS exposes this Physics global, you'll want to wrap that global into a service that you inject into other components. Although you could use the global symbol directly, creating a service for that global gives you the flexibility of injection, which can make testing easier, and in general it just feels like a better abstraction because of the flexibility involved and because of the way Angular works. And anyone who has been working with Angular for any length of time should be accustomed to asking for anything that they need access to by providing it as a function parameter. The second piece that jumps out is how you can associate PhysicsJS with a canvas. That's what this tutorial is doing. And anytime I see a library working with an HTML element, I think of directives, because directives are typically the type of component you want to build when an HTML element is involved. But before we get ahead of ourselves, let's build a service to wrap that global physics symbol.

Wrapping Global Symbols

Looking through the tutorial, it becomes obvious that the physics object provided by PhysicsJS is important. This is how I set up a render so that the Physics engine can draw something into the screen. It's how I add bodies to the scene, it's how I create a clock to set things in motion, all sorts of important APIs hanging off of Physics, so I will probably need access to Physics from many different places. And instead of using the global Physics symbol directly, I want to register this symbol as an Angular service so that I can inject it into other components. And that is easy enough to do, I can just tell Angular to register a service with the value method. The service name will be Physics, and the object to use for this will be the global symbol Physics. This will be available because I am sure to include the scripts needed for PhysicsJS in my shell page, before I include Angular. Now this value method, this is an underappreciated method compared to .service and .factory. But those two methods force Angular to do something to create the service, specifically, invoke some function. With .value, we are telling Angular it doesn't need to perform any action. The object that I'm giving to Angular is the object it should pass along and inject. So dear Angular, please take this Physics object, and if anyone wants to use it, if they're asking for the name Physics, just pass it along directly, don't try to invoke it or do anything with it before you pass it in. And now that this is in place, I can inject Physics into Controllers and directives and other services. Let's see how that works in the next clip when we build a directive to wrap the canvas element for PhysicsJS.

Touch the DOM with Directives

Now that we have a Physics service available in the application, it's time to take a look at what we can do with it. According to the tutorial, there's a few things we need to do with that Physics service. We'll need that service to create a world. We do that by invoking Physics and passing in a function where PhysicsJS will create a world and pass it into the function, and then we need to take that world and give it somewhere to render. What the tutorial is doing is setting up a canvas so that we can create a renderer and tell the world to render into that canvas. And since we're working with an HTML element here, the first thing that comes to mind to me is that I want to create a custom directive. To me the ideal custom directive for this would look something like this. I want a Physics canvas, I want to be able to specify the width and height, and then I just want the directive behind this to be able to set all of this up for me. And then ultimately what I want to do is be able to somehow specify shapes that I can attach to this canvas, so it's almost turning into SVG, but before we can get there, we have to create this canvas. I already have a file included in the shell page, it's going to be the file that returns the directive definition object for physicsCanvas. And I think physicsCanvas, since I ultimately plan on having some child elements like rectangles or some sort of physics shape, I want to restrict this to being used only as an element, and I'll probably at some point need to trasnclude in any content that was included in the physicsCanvas. We'll see that in a bit when we start adding shapes. I'm going to use a templateUrl for this directive. I want it to load "templates/physicsCanvas.html". That is actually a file that I already have created. Currently all it has is canvas, and it's going to set in the width and the height, and I expect those to be on the scope, that's why I have the data binding expressions there. And since I'm also specifying that I want to do some transclusion, I should also, inside of here, have a div with the ng-transclude directive on it. We did some transclusion earlier in this course when we built a custom alerts directive.

But now I know that I want width and height to be members of my scope object, so let's set up an isolated scope. I want width to come from an attribute named width, I want height to come from an attribute named height, and now let me write a link function. So because Physics is now a service, I should now be able to inject that service into my directive and I should be able to use it in my linking function where I take scope and an element. What I will have to do is find the canvas that is in my templateUrl, so let me use querySelector on the element that represents my directive, and ask for a "canvas". And then let me use some of the code that is right here in the tutorial to create a renderer. So a renderer is a Physics.renderer. I'm going to tell it that it's going to render into a canvas and then I provide some options that say, this, in fact, is the canvas element you're going to use. You will be using scope.width for the width, and for height you'll be using scope.height, and that will complete my renderer. Now I just need to set up the world. So if I invoke Physics and pass it a function that takes a world parameter, then I should be able to say, world, please add this renderer. And so far this is all we need to do with our PhysicsCanvas, but let's look at declaratively adding in some shapes onto that canvas in the next clip.

Working with Child Directives

In the PhysicsJS tutorial, they show a couple examples of adding bodies into the world to render. And the code looks something like this, you say Physics.body('rectangle'), here's that starting x and y position, here's the width and height. Then we tell the world to render, and PhysicsJS should put that on the screen. Now even though there's no DOM manipulation in this particular code, and I can certainly inject Physics into something like a controller and find a way to get access to world. What I'm thinking is that I could also do this easier if I just had some custom directives to represent Physics body. So in shell.html, if I had the ability to write code like this, I want a physics-body of type "rectangle", and here are the starting options, it's going to start at this x position and this y position, have a velocity on the y axis of -0.3, and the width and height, I think that would be even easier than writing this code inside of my controller. But if my controller does need access to this body that's being created, perhaps I can have an expression that says, let's take what PhysicsJS creates to represent this body and bind it against the current controller objects. We'll bind it as a property called box1. That will give me access to box1 inside of my controller. I think this will be a pretty good solution. Now there's a couple of things that have to happen though. As soon as you start having a parent child relationship in your directives, you have to be careful about a couple of things. This physics-body is going to create some sort of box, and ultimately it wants to add that box to the physics-canvas, but in order to be able to do that, a physics-canvas has to be initialized first, it has to have the ability to set up the world, set up the render, and then it's in a state to accept objects. But by default, the way we have written physicsCanvas, we have written it with a linking function, and technically when you do this with Angular, this is what's known as a post linking function. And with post linking functions, they run in the child element first and then in the parent element. That means the way we have things set up right now, physics-body would try to create some sort of object and place it into the world before the world has been created. But there's an easy fix for this. Instead of returning a post linking function to Angular, I need to give Angular what's known as a pre linking function, and the only way to get a pre linking function with Angular is to write a compile

function, and what this compile function has to do is return an object with the pre and the post link. So I need to return something, and in this case I just need it to have a pre member. I need this to go to a pre linking function. Let me take this function out of the directive definition object, because it's going to get a little bit uglier and I want to make sure that I cut all of the code inside of it, and I'll remove this link member entirely, because now we have a compile member. Angular will call that, it will return an object that has a pointer to the pre linking function. Let me place that here, preLink = a function that still takes scope and element and it's going to go out and find the canvas, set up the render, and we're good to go. Now the key part here is that when you build a new object you need to add it to a world, and that means my child directive is going to need access to the world created by this canvas, and that's something I can do by having a controller for this particular directive. So in addition to the compile member that we just added, let's add a controller, another small anonymous function. What I want to do is expose an API from this controller, so I'm going to add methods to the controller instance, and I'm going to build a method called add, because that is the API that I would expect on a world. So I want this canvas directive controller to represent a world. And this.add is a function that takes different types of things, so I'll just call it (thing). I just need to get this into the world. How am I going to get it into the world? Well, when we create the world up here in the pre linking function, what I could do is add that world to my isolated scope object, and down here in the controller step where we inject scope, now I should be able to say, $scope.world, please add this (thing). And with that code in place, it's now time to take a look at how to build a physics-body. Remember we want to be able to specify attributes like type, body, and options. So, once again, I have a directive definition object ready to go. I'm going to restrict a physicsBody to being just an element, and I'm going to require, as a parent, the physicsCanvas. That should give us access to that API and have the ability to call add on this controller. I'm going to set up an isolated scope for this one, so that I can have binding against that options object, and binding against the body, those are going to be model binding expressions, and then type is just going to be an attribute binding. I don't expect anyone to change the type of an object once it's been created. And all the linking function needs to do is ultimately call add on the physics canvas. So the linking function takes scope, element, attributes, and then the canvas. Unfortunately we use almost none of those, but we can say, scope.body = Physics.body, and this means we have to inject Physics. And Physics.body is going to take (scope.type, scope.options), and that should pretty well simulate the code that we see here in the tutorial. Now we just need to add that body into the canvas, so canvas.add, passing in (scope.body). What that ultimately should do is go to my physicsCanvas controller and add this thing that's been created, into the world. Now there's one last little bit that we have to do before we can see this render, and that is to set up a timer and a stepping function for the world. we'll look at that in the next clip.

Using Pre and Post Link

One of the things you'll discover just a little bit later in the tutorial is that nothing really happens until you start a ticker and tell the world to render on every timer tick. So what's interesting about this code is that it's not really obvious where to place this. In fact, on my first iteration of the integration I decided to create a ticker service, but now I'm deciding to throw that out. What this code really represents is post initialization code for our physicsCanvas. Every time I have a physicsCanvas, yes I want to set up a world,

I want to wire it to a canvas element, but I also just want this ticker and world rendering to be kicked off automatically. It's kind of interesting, because there's not many places where you can find a good example of using pre and post linking functions in Angular, but setting up this timer is a pretty good example. I want to have a pre linking function on my directive to make sure this parent is ready to go for any child directives, it has the world created, and it has the ability to allow bodies to be added to this world. But then once all of that is done, I want some initialization code to run and set up this timer tick, and that's almost ideally what post linking is for, because with a postLink the parent runs last. Now in addition to pre, I'm also going to have a postLink function, and instead of using an anonymous function here let's come up and write it here, postLink is a function, and just a like a pre linking function this can take scope, element, attributes, and controller, but all I really need here is just scope, because scope will give me access to the world, and that's all I need to do to set up this timer. Well, that and the Physics service, because I need to tell the Physics service to set up an event handler on the ticker so that every time it ticks it will call this function and pass in the time, and I will tell my world to step to that time, basically go to the next frame, and again, pretty much just copying this initialization code verbatim with the exception of adding scope. The next piece the tutorial shows is wiring up a step event on the world, and what you do on every step is tell the world to render. And then finally, we'll do a Physics.util.ticker.start, and that should set the world in motion, because remember, when we created these two little blocks here, I gave them some initial velocities, maybe I can decrease that just a little bit. I will suspect that they will float upward since they have a negative y velocity. I saved everything, let's come back to the browser, this is where the application loads. Let's refresh this, and there I can see the floating blocks. So before we review what we've done, let's add a couple more features to simulate gravity and boundaries.

Declarative Physics

In reading through the rest of the PhysicsJS tutorial, it becomes obvious that once you get beyond adding circles and polygons to a world, most of the features that you might want in a simulation are achieved by adding behaviors to the world. For example, adding the behavior of constant acceleration. By default what that will do is simulate gravity and pull objects to the bottom of the screen. You can also add edge-collision-detection. And when you combine collision-detection with body-impulse-response, what you'll have is shapes that bounce when they hit the edges of the canvas. Now once again, it isn't immediately obvious how to package this functionality, but if you take a step back and follow your intuition, you might say, well, since all of these instructions are adding features to the world object, and the world object is encapsulated in a custom directive, the physicsCanvas, well then I might make sense to make custom directives to add these behaviors, and I think that's a good fit. Over here in the web page, I've gone ahead and added some directives, physics-behavior, that will add the same behaviors that the tutorial is adding, so constant-acceleration, body-impulse-response, collision-detection. And although edge-detection is also a behavior, it takes a few more parameters typically, so I wrote a different custom directive for that. Now all of these custom directives follow the same basic pattern that physics-body did, which is to require the physics-canvas controller and inject the Physics service, use that Physics service to create something, and then add it to that canvas. This is what physicsBehavior

could look like. A very simple directive, it's restricted to be an element. It requires the physicsCanvas controller. All the user needs to specify is the name of the behavior that they want to add, and now we can create that behavior and add it to the canvas. The directive for edge-detection involves just a little more code, because we need to take these values and parse them out. I'm allowing the user to specify the minX, minY, maxX, and maxY, although really those could be obtained from the physicsCanvas if it just exposed them in an API and that would make this much easier. There's also the restitution value here, for how efficiently these objects should bounce, and then the one thing to keep in mind that whenever you're dealing with attribute binding is that you always receive attribute values as strings. And so if you have some sort of API that you want to call, like this Physics API to create a bounding box, this API expects to receive members, so I want to make sure that I use parseInt or parseFloat to convert those string values into actual member objects. But, once again, that's just a case of taking those parameters, creating a behavior, using the Physics service, and then adding that behavior to the canvas. Once I put all this together and save everything, I can come out into the application and refresh, and now I should have some bouncing boxes. So bouncing boxes are interesting, but let's see if we can do something with a controller and a button to make them bounce more.

Binding from Directive to Model

Back when we wrote the physicsBody directive, we provided the ability to bind the body from this isolated scope into a controller scope, and remember the body is that object that's been created using the Physics service. So I want to be able to see if I can interact with that body somehow from my controller. Over here in the shell page, I do just want to point out that there is the movingObjectController that is in charge of this section of the DOM. We haven't needed to use it yet, but now let's come down and add a button. I want a button that I can click on that will give a kick to these objects and make them move faster, move up the screen. So this will be a button, I'm going to style it with Bootstrap. It's a button, let's make it a primary button. It will have the text, Kick, and when a user clicks on this I'll use the ng-click directive to say, "object.kick". Let me save that file and open up the controller, and let's implement a kick method. So inside of movingObjectController, I haven't set up a model as yet, I haven't done anything inside of here, but I do expect that box1 and box2 will be populated by my directives, and that's a little backwards than what you typically have in Angular. Typically in Angular your controller sets up a model, and those model values get pushed into directives. In this case, it is my directive that's going to be creating the data and then binding it into the controller scope, and I think that's fine, but it is here where I will implement my kick method, and this is where I can start taking advantage of some of the PhysicsJS API's. Box1 and box2 will be Physics bodies, and if you look through the PhysicsJS API, you'll see that you can do things with these objects like apply a force. I want to apply a force for box1 in the x direction of 0.1, and in the y direction I want it to move upward, so let's go negative, let's do a -0.2. And I want to do almost the same thing for box2, but let's have it move in a slightly different x direction, and save all this, and let's see what this looks and behaves like. There's my boxes, I'm going to kick them, and now I can push this and watch boxes fly all over the screen. So in the next clip, let's take a review of what we've done with PhysicsJS and some of the general strategies and techniques that you can take away from this.

Integration Summary

I want to take a moment to summarize what we've seen so far. When you want to use another library with Angular, you might want to use that library directly without writing any additional Angular components, but it's also helpful, I believe, to look for opportunities to embrace the library and integrate with the library, and integrate the library with Angular by looking for opportunities to wrap entry points with services. Not only does this improve the testability of your application, but it also gives you the ability to adapt and customize a library by having code, like configuration code, tuck it away into a central place in a service, and build an adapter layer around that library. That's going to make it easier to use from your application. Also look for opportunities to write custom directives to expose library features. Any library that revolves around the DOM will probably be a library that you can use from inside of directives. But not every library will integrate with directives. You can certainly have algorithmic type libraries that do encoding or hashing or integrate with local storage, and these things have nothing to do with the DOM. For those types of libraries, just having services will probably suffice. And in the end what you want to do is you just want to think the "Angular" way, preserve your separation of concerns and set up things so that you don't use a library for DOM manipulation inside of a controller. Don't ruin your testability and always look for opportunities to use two-way data binding between directives and controllers, and use the ability to communicate between controllers inside of directives. These are techniques that you can use to "Angularize" another library into your code. And speaking of other libraries, let's take a look at integrating a simple plugin for one of the world's most popular libraries, the jQuery library.

Using jQuery Plugins

Of course, the world of jQuery has thousands of plugins available, and many people have a good working knowledge of some of these plugins and use them in their applications already. So when working with Angular, you might want to carry your jQuery knowledge and your jQuery plugins forward into the Angular world, which is great. The only caveat I'd give you is to make sure you really need jQuery in those plugins, because in many cases things are easier with Angular, and sometimes you can replicate the behavior of a plugin just using a little bit of code in Angular. And, of course, many people have already written native Angular components and made them available for free, and these components replace what we used to use as a jQuery plugin. A good example is AngularUI, which we looked at in a previous module. Using AngularUI, you can use all the Bootstrap components, and that usually requires jQuery and bootstrap.js but AngularUI rewrote everything using Angular, so those are two less script libraries that you would need to include. Plus, the components have been Angularized, so they just work as custom directives that make sense in an Angular application. Now I found a nifty little jQuery plugin that I want to use, it's jQuery Knob, and with jQuery Knob I can turn an input into a knob, into a dial. And I can spin these touch-sensitive dials to change some input value, and I think it would be neat to control the size of a box in my Physics demo using one of these knobs. Now the general

approach for including a jQuery plugin in an Angular app, is to write a custom directive that will encapsulate that plugin, because as we can see in the examples for jQuery Knob in the GitHub repository for this plugin, what you'll do is just have your normal input type="text", and then you use jQuery to go out and select the inputs that you want to turn into knobs, and then just call the .knob method on the jQuery wrap set, and off it goes. I can already see a custom directive named Knob. It has a link function that will take care of this code for me, but before we get there, just a word on setup. Here are all the scripts that I'm including on the shell page for the application we're building with the physics demo, and you'll notice the very first script that I am including now is jQuery, and I include jquery.knob, which is that plugin. But if you're going to be using jQuery, you want to include it before you include the angular.js script, and that's because angular.js can actually take advantage of jQuery, and if you're going to have jQuery on the page, you might as well use it to its full potential. What will happen is that if AngularJS sees that jQuery is already defined, it will use jQuery when it produces those wrapped elements that you use inside of a Link function or you use angular.element to create an element or wrap an element. So instead of jQLite, which is what Angular provides out-of-the-box, angular will be handing you elements that are wrapped with the full power of jQuery. And that means if I have a plugin, like jquery.knob, if I walk up to an Angular element and I invoke that .knob method on it, if it's the right type of element that knob works with, I will turn that element into a knob, into a dial. So what I'm thinking on my shell page, is in addition to the button that will kick my boxes and make them fly around a little bit more, I want to have an input. Let's make it type="number", and I want to be able to specify the minimum number for this knob, for this input, and the maximum number, and the width, because reading through the documentation for knob I can see that I can set a width. And I will give this the initial value of "15". And you might be wondering, don't you also want an ng-model directive so you can bind that value directly to some box coordinate inside of the controller, and the answer is actually, no, for a couple of reasons, which I'll explain. Let's see if we can create a custom directive knob, and actually turn this input into one of those dials. So I already have a knob.js file, and one of the first things I'll need to do is go out and grab those attributes, the minimum value the maximum value. This jQuery plugin wants to know about all of those when I create the knob. So let's set up an isolated scope, actually, that will grab the min attribute, the max attribute, the width attribute. We will write a linking function that takes scope, and element, I think those are the only two I'm going to need, and what I can say is element.knob. I can do that because element will be wrapped by jQuery, and any jQuery plugin that I have included on the page, the methods that it adds to the wrapped set, to the jQuery API, they'll be available. So I should be able to say element.knob, and give it a configuration object that says, yes, your min is scope.min, your max is scope.max, your width is scope.width, and that should be good enough just to see if we can get something on the screen. Let me save everything, come out to the application and refresh, and there I can see I have my dial, we just don't have it hooked up to control the size of the box. We'll do that next. I just want to point out one important design decision that was made here. One way to program jQuery knob is just with data-attributes. So if you put things like data-width="150" directly on the element that you call .knob on, this plugin will find that attribute, take that value, and then accept the width. And so one of the things that you might want to debate when you're writing a custom directive for this plugin is, do I just want to go ahead and use data- attributes or do I want to define my own attributes and pass them in explicitly like this. And really, the correct answer to that question depends on the application and how you're going to use this plugin. I usually prefer

actually having an isolated scope and explicitly showing what those attributes are going to be, because any developer who's used to Angular and used to using custom directives, will be looking at this file to see what configuration options are available. And if they look in here and explicitly see that, oh, I can set min, and max, and width, that's very beneficial. But now that we have this in place, and now that we have this on the screen, let's look at how we can now actually use it to influence the size of a box.

Function Binding from Isolated Scope

Now what we want to do is have our dial influence the size of one of the boxes on our screen. And as I mentioned before, we're not going to do that by adding an ng-model directive to this input. That will work, and would make sense in many scenarios, but unfortunately with PhysicsJS, this is a library that just doesn't expect objects to be dynamically updated, or at least not updated just by setting a value like a width or a height. It requires a little more code to change the shape or the size of an object that's been created with the Physics library, and for that reason a simple ng-model binding is not going to work. I'm going to need to know when this value changes so I can write some imperative code to update my objects. And one way to do this would be to use a feature of the jQuery Knob plugin, which is the ability to wire up a function that will respond to a change event. And this is one way to do it, it's not the only way to do this, but it will allow us to explore a feature of scope, of isolated scope, that we haven't used yet, and that is function binding. I want to have change be a method that I can invoke on my scope that will ultimately go to some expression with the same name on this attribute, and execute that expression. So I want to say knob change="objects.resize, and pass in some value that is given to me. That adds an extra complication to this scenario. Now before we implement that in the directive, I actually want to go ahead and implement this resize inside of my controller, and that will be a method that is attached to model, that's going to be a function. I expect it to take some value, and this is what I need to do with PhysicsJS to get things to change their shape. I need to go to the geometry of my physicsBody, and I can set the width to this value, and I can take this same code and change the height of this physicsBody, and I'm just going to change box1. And then one of the things that I have discovered the hard way with PhysicsJS is that after you change one of these attributes, you have to go out and null out the view and then tell box1 to recalc. And this is why I don't want to use a simple ng-model binding, I want to know explicitly when the value has changed so I can do all of this bookkeeping and make things work. And there's probably a nicer way to do this, but what I really want to focus on here in this scenario is getting this expression invoked when a change in the knob happens, and getting a value parameter through. So in the configuration object that I pass to element.knob, I can wire up a change function and have it call scope.change. Now this change function, when the knob invokes it, it's going to pass in a value, and you might think, oh, well this is just as easy as forwarding that value to scope.change, but if we were to run this application we would discover that that doesn't work. So you might think, oh, I know what I forgot, Scott told me about this. Anytime I go around changing model values, I should do it inside of an apply block, and that's very true, I do want to have scope.apply here. The irony is that with PhysicsJS it won't matter, because what we're doing is changing a model value that only PhysicsJS cares about. Angular has nothing to do with the width, and the height, and the display of one of these boxes on the canvas. But it would be important to have it apply here, because if I ever do use Knob with

Angular models, then I just have to remember I am wiring up to a native event here, I'd be making model changes outside of Angular code, and now the digest doesn't run, and I don't detect any changes, and I don't update things that are in the display. So I do want to have the scope.apply here, but even with that the application still wouldn't work, and here's what's happening. Whenever you use the & binding, you're essentially creating a proxy function on your isolated scope. And when you invoke this proxy function, Angular goes out and looks at the expression here and it figures out how to invoke that expression to get it to work. But you have to understand that Angular actually understands expressions like this at a very deep level. If you look through the Angular source code, you'll see lots of regular expressions to parse things out, and Angular even knows the name of this parameter that you want to pass to the resize method on objects. I've actually aliased controller as objects, so that would be another reason this wouldn't work. But even if I had objects here, this is still not going to work. Angular understands that it needs a value to pass into this function, and that's the key bit. When I invoke one of these proxy functions, I don't want to just pass parameters along, I want to pass along, essentially, a hash that tells Angular in the expression being used, if there is something with this name, value, then pass along what is inside of this variable for that function argument. So this is the syntax that you want to use, this object literal syntax, to get a parameter into an expression that is bound to a proxy function on your isolated scope. So with all of that in place, I can save everything and now I can hope that this will work. So let's come back to the application and let's change the size of that box. That looks like it works, I can still kick it, make it smaller, make it larger, kick it around a few times.

Summary

In this last part of the module we looked at integrating a jQuery plugin, and 99% of the plugins you'll want to use with Angular will be DOM related, so you'll almost always want to write a custom directive to wrap the plugin. Once again, this gives you the opportunity to adapt the plugin to your application, perhaps specifying some common defaults so the code outside the directive doesn't have to. Remember to include jQuery.js before including Angular.js. This gives you access to all the jQuery features from Angular.element. Also, don't forget to use scope.apply. If you are handling native events with jQuery or custom events from a plugin, you'll need to place the event handling code inside of a scope.apply to make sure that Angular jumps in and performs some change detection. Finally, I showed you a tip for & binding in the cases where you want to pass a parameter to an expression in the view. To be honest, I've never been happy with that aspect of Angular, I think it's a bit confusing for the consumer of a directive, they will need to dig through the code often to figure out what's going on, but in some scenarios it's necessary, so I wanted to show you how to achieve this. And that concludes my Angular Playbook. I hope you enjoyed the videos. They are based on a couple years of working on a half dozen real projects with Angular, that was some hard-won experience, I hope you can benefit from it.