Your First Directive

Course Introduction & Updates

Hello, I'm Joe Eames, and welcome to Pluralsight's course Angular JS Directives Fundamentals. This course will teach you how to go from writing your first directive to becoming an expert in Angular Directives. By the time you are through with this course, you will be able to amaze your friends and neighbors with your encyclopedic knowledge of transclusion, speak intelligently on the difference between isolated and inherited scopes, and ridicule the pitiful directives written by other developers, or at the very least, you will feel really comfortable writing directives and understand not only how, but when and why to write directives. Directives are one of those pieces of Angular JS that is a bit like a rabbit hole. It appears to go down and down, but by watching this course, you will discover that directives really aren't that difficult, and it's quite possible to learn even the corners and recesses of directives in a reasonable amount of time. One of the major goals I had with this course was to do a lot of different demos, so in this course we will build a lot of different directives. Some directives will be small and simple and built in a few minutes, some of them will be big and complex and slowly built up over multiple modules. In doing this, you'll get the opportunity to see each concept taught through more than one demonstration. I highly encourage you to follow along and code up everything I code. It won't be difficult to follow along, but it will require effort to learn the things that will be taught. So typing up the code and making it work and trying things that aren't shown will all enhance what you get out of this course. The first four modules are a gentle introduction to directives. We'll start in module one by creating our first directive. We'll learn all about bindings, templates, and naming directives. Then, in module two, we'll learn about how to react to events, how the different kinds of scope relationships effect our directives, and how to add a controller to our directive. In module three, we'll discuss decorator directives, which modify the display or functionality of an existing node, and in module four, we'll learn all about transclusion, which lets us fill in our own display to an existing directive. This may seem like an advanced topic, but in this module we will see that transclusion is a lot simpler than many people may think. Then we'll move on to the second part of this course, which covers the more advanced parts of directives. We'll start with structural directives, which are directives that modify the structure of the DOM. Then, we'll look at advanced ways to deal with controllers and directives and how to create nested directives. Finally, we'll look at how directives and services interact, and some directives that are better built as services. This course is meant to be a comprehensive and indeed exhausting coverage of directives in Angular, but it will also show that directives are easy to learn and easy to build if you just go step by step. Because Angular changes so frequently and the publishing process for videos can take so long, I have created a course website. This allows me to let you know if the course is currently up to date, and if not, how long it will be before the course is brought up to date, and what to do to work around whatever may have become out of date. The things that we'll do in this course aren't too likely to get out of date, but it still is possible for that to happen, so it's a good idea to go and visit this page right now.

Tools

In this section, we're going to talking about the tools that we'll be using in this course. There are two main tools that we will use. The first is the developer tools in our browser. I'm going to give you a quick look at the developer tools in Chrome, but everything that we will need to do we can do in either Firefox or Internet Explorer. The first thing we need to do is be able to open up the developer tools. On a Windows machine, you do this with Ctrl + Shift + I or with the F12 key. On a Mac we use Command Option I. Once we have the tools open, there's only two tabs that we'll be using in this course. The first is the Console tab, which is right here. This will show us the Console output every time we use the Console.log statement inside of our program, and the second is the Elements tab. This shows us the structure of our HTML. In addition to that, we can look at specific pieces of HTML by either clicking the Inspector tool right here, and then selecting an element, which highlights that element, or we can use a keyboard shortcut to do the same thing. On Windows, that's Ctrl + Shift + C, and on a Mac, that's Command Shift C. Command Option C also works. The other tool that we'll be using in this course is Plunker. This is at the address plnkr.co. From here if you simply click Launch the Editor, you'll get to a Plunker editing window. Plunker is a comprehensive online coding tool. This will be our development environment throughout this course. Of course, you're free to use any development environment you wish, but Plunker makes it very easy to try out new things and save your work. I'm going to give you a very brief introduction to Plunker. This is by no means comprehensive, but it will give you the basics that you need in order to follow along. On the left-hand side over here is our list of files. This link right here allows us to create a new file, which can name whatever we want. This area down here allows us to describe our Plunk. Here in the center is the editing window. This will display the contents of the currently selected file and allow us to edit it. Over here on the right-hand side are several different slide-out windows that we can open. The first one is the Live Preview window. By clicking this, we will see a live preview of the code that we have created. This next one is the external libraries window. This is where we can find Angular JS and add it into our project. When we want to add a library to our project, we have to have an HTML page open. I currently have index.html open, so if I click this angular.js link, it will insert it into my file. You can see that it's added this script tag right here. One thing that is important to note is that we've added the released version of Angular 1.3. You should use the latest version of Angular. You'll see that throughout the course I'm using various release candidates of Angular 1.3, but you can safely ignore that, because everything we do in this course will be compatible with the latest version of Angular. Up here, I can search for packages, so if I want to look for different Angular packages I could type in Angular and hit Enter, or if I want jQuery, I can type in jQuery. And of course if I just click Home, I go back to just seeing all the popular packages. For most of our projects, we're going to use Angular, jQuery, and Bootstrap. I've already added in Angular, so I'll add in jQuery and Bootstrap. It is important to know that jQuery should come before Angular and Bootstrap. jQuery is a requirement for the Bootstrap JavaScript, and if we put jQuery before Angular, then Angular will use the full version of jQuery instead of the mini version, which comes included with Angular. We're going to be creating a lot of different directives; therefore, we're going to be creating a lot of different Plunks. Because of this, it's a great idea to set up a template inside of Plunker so that you can use that every time we create a new project. So let's walk through the steps for creating a suitable template for an Angular project, and then save that template. First I'll go over and collapse my external libraries window so I can see the editing window. Then I'm going to add my ng-app directive, and we'll just use the app name of app. I'm going to go down to the body, and I'm going to add the ng-controller directive, and I'll use a controller named mainCtrl. Those are all the changes I'm going to make to the HTML. You could also add the container class to the body. I do that in several other projects as well, but that's up to you. Let's go down into our script file, and here's a useful tip, if you want to navigate between files, holding down Control and hitting an arrow key will navigate between files inside of Plunker. For our script template, we'll just create a module, and we'll create a controller, which will be that mainCtrl, we'll inject in the scope, and then we'll leave it empty. This is a suitable starting point. Now that I have this looking the way that I want for a template, I'm going to click Save-- you can also hit Ctrl + S-- that will save my Plunk, and now I've got this button, which says Save this Plunk to your list of templates. But before I do that, I do want to give it a description to make it easier to tell among the different templates that I may have. So I'll come down in here in the Description box and type in Simple Angular 1.3. I'll say that again, and now I'll go over and click this button to save this Plunk to my list of templates. Once that's done, if I click over here in my Templates list, you can actually see the template that we just created has shown up right here. So there's a basic rundown of how to use Plunker. There's a lot more you can do with it and I recommend you explore the options in Plunker, but this will give you enough knowledge to follow along in this course.

User Info Card Directive

Let's move right into creating our first directive. This directive will be a user information card that we can drop anywhere on the page that will show information about a specific user. The first question we'll address is, how are we going to specify this directive? There are four different ways you can specify directives with Angular. The first way, which you're probably very comfortable with, is specifying a directive as an attribute. You've already done that several times through the use of directives like the ng-controller directive, which is specified here as an attribute, or the ng-app directive, which is specified up here as an attribute. The second way to specify an attribute is by creating a brand-new custom tag, such as this. These two methods for specifying directives are the most common and the recommended ways to use directives with Angular. There are two other methods that you can use to do directives with Angular, but they are not recommended and typically only used when working with older versions of Internet Explorer. The first of those two methods is to specify the directive as a class, or to specify the directive as a comment. Either of these two methods will work, but as I mentioned, they're not recommended, especially because they're easy to overlook when reading HTML. Of the two methods which are recommended, attributes and elements, there's definitely a difference between the two. Using an attribute typically implies that you're modifying an existing element, whereas using an element typically implies that you're adding some kind of a new object to the page. So for something like the user information card, it would be more appropriate to use a new custom element. Before we do that, though, we're going to start out by creating our directive in the simplest way possible. So let's go back to our script file, and we'll specify a new directive. We do that by calling the directive function on our module and then first giving it a name, and this function will return an object which describes the directive we're creating. At this point, we've created a valid directive in the most minimal way possible. This directive will do absolutely nothing, but it will be handled as a directive by Angular. Let's see that in action. Go back to our index.html page, and instead of creating a new custom element, I'm going to add a div tag with the user-info-card as an attribute. Notice that the attribute is specified differently than the name that I gave the directive in the script file. Back in the script file, you can see that the name was camel case, lower case u, uppercase I, and uppercase C. But in the HTML, I've used dashes in between each of those different words. That's because HTML is not case sensitive. So in order to support the non-case-sensitivity of HTML, the names of directives in HTML are specified by using dashes in between the words. This is called dasherization or snake-casing. There are alternatives to this method as well. You can also use colons like that, or you can use underscores like this. All of these methods are completely equivalent. Also, in order to support valid HTML, you can prefix your directives with an X or with data. You can also mix the use of dashes, colons, and underscores, and Angular will still handle the directive just fine. Of course, the most common way is to just use dashes and to not specify a prefix. Now let's see our directive in action by loading up the preview pane, and we're going to use the developer tools in order to look at this div. And you can see that the div is there with the directive listed on it. Now of course our directive isn't doing anything, so let's start building our directive by giving it something to display. Back into our script.js, and I'll close the developer tools. We're going to add an attribute to this element called template. This attribute is a string that we'll display inside the element that the directive has been added to. So let's just write in User Info Card, and you can see that the text is now displayed. Now let's go back to our HTML and look at our directive again. Our directive is specified as an attribute on this div element. But of course as we discussed before, our user information card is more appropriate as a custom element, because it's the kind of component that you lay down on a page and doesn't necessarily modify an existing element, which is what an attribute implies. If we were to change this into a custom element by doing this, if you're using Angular 1.2 or before, then the text will disappear. If you're using 1.3 or later, then it will still be the same. The reason is that Angular has a default restriction for directives. In 1.2 and earlier, by default a directive can only be an attribute. For 1.3 and later, the default is element or attribute. We can set the restrictions specifically by going back to our directive definition and adding a new attribute to our object. This is the restrict attribute, and that's a string that will specify the format of the directive. If you want the directive to be an attribute, you specify a capital A, but if you want it to be an element, you specify a capital E. The other formats that we could specify are of course classes, which is a capital C, or comments, which is a capital M. Now if you want your directive to be available in multiple formats, say as both a comment and as a class, then you could specify multiple formats together. Of course for our directive, all we want it to be is a custom element, so we'll change this back to just the E. And our directive is now displaying again.

Adding Bindings

Let's improve our directive by adding some bindings to our template. Right now our directive simply displays some static text, the words User Info Card. Let's make it display something more useful. Let's say that up on our scope, we've got a user object. And this user object has a name property. Now I want my user info card to display the name of the user, so I'll go back down into the template, and I'll put the word Name, a colon, and then inside of double curly braces, I'll type in user.name, and you can see that the directive is now displaying the name of our user. When Angular processes this directive's templates, it will see a binding to a user object, so it will look on the current scope, and since this directive is placed inside the HTML on by the mainCtrl, it's going to ask the mainCtrl scope for an object named user, and then access the name property. The template for a directive works exactly like a regular HTML page. You can add in bindings and they will be interpolated by Angular and replaced with the actual values. Let's make this a little more interesting by adding some more data to our user. Now let's go down to our template and add in places for all of that data. I'll add in the user street, a line break, and then their city, one more line break, and then their planet. I've added quite a few bindings to my template and Angular has replaced them all correctly. So not only was I able to add in bindings, I was also able to add in HTML tags, like these br tags. Templates are extremely versatile and can handle anything that you would do with Angular inside of a regular HTML page.

Using Built-In Directives

We've seen how the template for a directive can handle both bindings and HTML. Another thing it can handle is other directives. Let's look at the one of the problems our directive has so far. If I were to go in and remove this address property from my user object, when the user object displays, you can see that the address heading is still there. I'd like that to not be visible if there's no address object for the user. There's a great directive built into Angular called ng-show, which will do this for me. The way that we do that is by wrapping the address inside of an HTML element, so in this case I'll wrap it in a div, and go out to the end of the address and put a closing div tag, and I'll go back to the opening div tag, and I'll add in an ng-show directive. And here I'll specify that it should only show if the user object has an address property. I'll do that with a double exclamation mark, user.address. And now you can see the address heading has disappeared. And if I go back in and add the address back in, the address heading comes back with the address. Let's add one more thing, and let's give our user object some friends. We'll give this user three friends. I'd like to display these three friends on the user card, so go back into my template all the way out to the end, I'll add another div, put the heading of Friends, and now I'll add one div for each friend, and I'll do this by using the ng-repeat directive. Inside of here I'll put a simple binding that will display the name of the friend, and I'll close off the div for the entire friends section, and let's format this just a little bit by adding an extra break right before the friends section, and one more right before the address section. And let's go into our index.html and let's take out this h1 tag, and there our directive is working the way that we want it.

Templates

Now one of the issues that we've seen so far is that working inside of this template string is becoming really unwieldy. As we add line breaks and divs, it's really difficult to tell where we are inside of the string and where we are inside of the nesting of our HTML. Thankfully this is a problem that is easily fixed by using an external template. Let's create a new file, which we will call userinfoCard.html. We'll go back into our script, and let's grab this template string, cut it out of here, put it into this userInfoCard, and then we'll format it. Let's just indent these a little bit, remove this blank line, all right, now that I've got my template formatted the way that I want it, we'll go back to our directive, and we'll change this template to be a templateUrl property. And we can just specify the name of our template. And now our directive is back to working. This is one of the great features of Angular directives. You can author the templates for your directives inside of their most natural environment and HTML page, but Angular will incorporate them into your directive for you. For all but the most trivial of templates, you'll want to use a separate template and specify a templateUrl inside of your Angular directive.

Types of Directives

Directives come in three types. Each of these three types of directives has a different purpose. This is something that is almost never talked about, but it's an important concept since it will play a more central role in future versions of Angular. We've already built the first type of directive, and that is the component. A component is the canonical custom directive. It represents some data and its associated HTML display and any necessary functionality, and is almost always implemented as a custom element. You might also call this type of directive a widget. Our user info card is a component directive. We can put it where we want, and it will add the new functionality in that place on the page. Components are often business case specific, like our user info card, but they can be generic as well, such as a button. The next type of directive is a decorator. This type is actually the most commonly used type of directive. A decorator adds additional functionality to an existing tag or modifies that tag's display. The classic example of this is the ng-click directive, which adds click functionality to an existing tag. Also, the ng-show and ng-high directives are examples of this. Again, these directives are almost always implemented as an attributed and almost never have a template. The final type of directive is a structural or templating directive. A great example of this is the ng-repeat directive. This type of directive manipulates the DOM structurally to produce a display. We will be working with and building all three kinds of directives through this course, and future modules will go in more depth into all three types of directives. Understanding the type of directive you're building, and how it should be implemented is important so that you can understand when you might be mixing directive purposes and use that as a way to determine if you're building too much into your directive.

The Replace Attribute

Now one of the interesting things about directives of templates is how the contents of the template actually end up inside of your HTML. Let's look at the HTML for our directive. As I look inside this HTMl, you can see right here there's actually a user-info-card element. Now, of course, there is no user-info-card element in the HTML 5 spec, but you can see that the contents of the template have been inserted inside of this user-info-card element, and the user-info-card element has been left inside of the document. Now there's a couple of reasons why we might not want that user-info-card element to remain inside of the document. Usually that has to do with HTML validity reasons, but there are some other reasons why you might want to remove that user-info-card element from our HTML. Some of the reasons are fairly complex, so we won't go into them today. We'll simply look at how to remove the user-info-card element. We do this by going back to our directive and adding a new attribute to our directive definition object called replace, and we'll set this to true. This attribute will replace the directive element itself with just the contents of the template. You'll notice that the directive is no longer displaying. That's because our template is not compatible with using the replace attribute. Let's go into our template, and you can see that our template is just a bunch of HTML, but there's no common root element. If you want to replace your directive element with the contents of your template, then you actually have to have a single root HTML element. So let's just wrap this inside of a div, and you can see that the directive is now working again. Let's look at the HTML, and you can see that we no longer have that custom element, we simply now just have a div. Now with current modern-day browsers, there's really no issue with leaving the custom element inside of the HTML. And in general we're going to want to do that. In fact, Angular has deprecated the replace attribute as of Angular 1.3. You can still use it, but they consider it to be best practice to not use the replace attribute. For all of our examples, we won't be using the replace attribute, so I'm going to back out the changes that I've done, and take this back off again, but if you ever do need to take out that custom HTML element, the replace attribute is how you do it.

Naming

Let's quickly discuss naming our directives. When naming directives there are actually four names we need to worry about. The ones that we have been looking at so far are the name and directive in code, which is this name right here, in addition to the name of the directive in HTML, which would be this name right here, and the third name we've been dealing with is the name of our template, which is this name right here. The fourth name that we haven't dealt with yet is the name of the directives JavaScript file. In this course we just put our directives inside of our script file, so we haven't had to worry about the name of our JavaScript file. But in a production application, we're going to want our directives to be inside of their own files. For more comprehensive naming advice, my course on Angular Best Practices has an entire module devoted to how to organize your code, including how to name objects and files, so I highly recommend that you at least watch that module of that course. There is one very specific piece of naming advice that I want to cover here. When naming directives, you should always make sure that your directives have some kind of prefix. Note that in Angular, every directive is prefixed with ng-. When writing our own directives, if we also prefix them, that gives us a couple of very big benefits. First, let's imagine we created a new directive called carousel. If this was our directive, we have two potential problems with this name. One, it's theoretically possible that in the future there is going to be a new HTML 5 element named carousel. Two, for any other developers that are looking at this HTML, it's not obvious that this is a directive. They might think that this is an HTML element they're not familiar with, or they might accidentally just skip over this code. And what if you wanted to create a section directive? Well, you really shouldn't do that because there already is a section HTML element. So it's always best to prefix your directives with some kind of a short prefix, two or three letters is best. You can use some kind of pneumonic, perhaps something for the project or for your company. For example, if you worked for the Empire, you might use the prefix te. This would let any developer easily look through the HTML and see which elements are directives. Then down in our script, we would simply change this to teUserInfoCard, and our directive is back to working. For this course, I won't be using prefixes, I'll simply be using multi-word directive names, but in a production application, you absolutely should use some kind of prefix.

Summary

In this module, we got our first introduction to how to build directives. We talked about the tools we're going to be using in this course and how to leverage them effectively, not only for this course, but also for experimenting with Angular. Then we spent the majority of this module building our first directive. This is a info user card directive, and also what we call a component directive. We'll be enhancing this directive through later parts of this course, but we'll also be building other directives as well. Lastly, we talked about the three different types of directives, how they differ from each other and how they might be used, and we talked a little bit about how in Angular 2.0 these different types are going to play a more important role with directives. This lets us look forward to the future while also giving us a way to organize our directives as we build them today. So we've learned some of the absolute basics of directives and things are only going to get more interesting from here on out.

Events, Scope, and Controllers

Introduction

In this module we will be discussing Events, Scope, and Controllers. This module is the second module that covers the absolute basics of directives. After creating the basic directive that simply displays data, the next step is to start adding some inner activity to your directives through events. Handling events is the job of functions on your scope, which will require a controller. These three pieces play an integral part in making you directives interactive. In this module, we'll learn how to do this, the different kinds of relationships that directive scopes can have with their containing controller, and we'll see how and when to use each of these effectively. By the end of this module, you will be able to create powerful interactive component directives.

Adding a Controller to Handle Events

In this section, we're going to be adding a controller, but before we get into that, I'd like to style this up a little bit better. The display is really less effective, and I want the information to be a little bit more visible. And that's something that can easily be done by including Bootstrap. So even though this course isn't about Bootstrap, I'm going to go ahead and include it anyway and style this up just a little bit so that we're not looking at just such ugly plain HTML. I do that by going to my Find external libraries, and the popular ones that come up Bootstrap is number three, so I'll click that. Remember I have to have that HTML selected, and then when I click Bootstrap it will add it into the HTML page. So I click that, it adds Bootstrap. Let's close this down. Now one of the things I notice is that it put the jQuery script after the Angular script. Whenever we include jQuery and Angular in the same page, we want to put jQuery beforehand, because Angular includes its own mini version of jQuery with limited functionality, and if we include the full version of jQuery, then Angular will use that instead, but in order to make that happen we have to have the jQuery script before the Angular script. So I'll move that up. And now that's fine. Let's open up our preview pane, and we can see that things have been formatted a little bit differently, but I'd like to get them looking a little bit better. So we'll start by adding a class to our body. This would be the container class. And that will add a little bit of left and right padding. I also want some top padding, so I'm just going to manually add that. All right, that looks pretty decent. So I've got the right amount of padding. Now I want to go into my userInfoCard, and I want to format this up just a little bit. So I'll start by putting a div around the entire thing. And this I'm going to make it a panel with the panel-primary class. And notice that Plunker will expand this out into HTML format, it is called Zen coding, so if I hit Tab it turns that into actual HTML, move this end div down to the bottom. I want to take these and indent them all. And I want to add a panel heading, so the name of the user is going to be inside of my panel heading. So that's another div, and inside of here I'll put the user.name, and since that's up at the top, I really don't need to prefix this with some sort of a name label, so I'm just going to remove this whole line there. Now we'll wrap the rest inside of a panel-body div, put that down there, and that's starting to look pretty decent. Let's indent these again, and let's make one more change, we'll make these labels h4s, and I'm going take Friends out of its own div and just wrap it in h4, which means I can get rid of this end div. And instead of having the ng-repeat be divs, I'm going to make this in an ordered list. And these will be lis. Okay, now this is looking a lot better and it's quite a bit easier to read. So let's move onto our functionality. What I want to do is I want to add the ability to take a user and knight them by clicking a simple button down at the bottom of the user card that will give the user the rank of knight. Obviously to do this I'm going to need a button. So let's go add a button, and let's make this a success button, and the text will be Knight Me. Next we need to actually handle the click event. And we do this by adding the ng-click directive to the button. And we'll call a method named knightMe and we'll pass in the user. And so that will pass the current user into the knightMe function. Now the next question is, where are we going to put this knightMe function, and what is it going to do? So let's move over to our script file, and it's obvious that we could just put it on our controller up here, and that's a very simple way to implement this, because the scope of the controller is visible inside of the directive, because this directive lives inside of the main controller, and when we knight a user we're going to give them a rank property and set it equal to knight. Now that we've knighted the user, we've got to display that somehow. Let's go back to our userInfoCard, and right above the button, we'll add in another little section, and this will display the user's rank. Now, of course, I don't want this rank label to show if they don't have a rank, so let's add an ng-show directive, and it will only show if they actually have a rank property. And the same thing with the button, I don't want the button to show up if they have a rank, because there's no reason to give somebody the rank of knight twice. So let's add another ng-show and this time it will just be not user.rank. Okay so now our functionality is going to work, so if I come over here and click the button, we can see that it's been given the rank of knight and the button has now disappeared. That's the functionality we wanted. Let's go back and look at our script. Right here in this directive, notice something very interesting. The knight button lives inside of the directive, but the method that handles it has been placed inside of the controller, not inside of the directive, so we're violating a law of encapsulation here. We actually want to put this knightMe function inside of the directive. And the way that we do that is by giving the directive its own controller. That way the directive can implement its own methods to respond to events. In order to add a controller to a directive is very simple, we just add a new controller property, and then we add a function that is the controller. And this is going to be a dependency injected function just like a regular controller, so we add in a $scope. And from here we can take the knightMe function, pull it out of our controller up here, and put it inside of the directives controller. Now if we go back and click the Knight Me button, we're going to get the same functionality, but the function has been moved inside of the directive and not inside of the controller, so we've maintained our encapsulation. So we're beginning to see that between the templates and controllers, directives are almost like a little mini ecosystem inside of Angular where everything that you can do inside of regular Angular can be done inside of a directive. And that's great, because we can encapsulate a lot of functionality into a single piece and just drop it on a page wherever we want. Again, if we go back to our index.html, we can see that all that we've done is add this user-info-card tag, and now we've added all of this functionality by simply adding a single tag into our HTML. And that, of course, is the power and beauty of directives.

Directive Scopes

In this section we're going to solve a major problem with our directive, and that is the fact that if we were to place two copies of our directive on the same page, they would both display the same data, because they're both simply looking for an object named user. We don't have the capability right now of specifying the data source for each directive. We're going to solve that problem with scopes. In our last section, we added a controller to our directive. One of the interesting things about adding a controller to a directive is that that controller receives a scope object, just like a normal controller in Angular receives a scope object. So our next question may be, what is the difference between the two different scopes? Well, that depends on how you create your directive. There are three different ways that you can set up the relationship between the directive scope and the containing controller scope. The default is for the directive to share the scope with the containing controller. This diagram illustrates this relationship. You can see the parent controller scope here in purple, and we can see our user object that was created on the scope of the parent controller here. And we've created a directive with Shared Scope. That directive, as you can see, lives entirely inside the parent controller scope, and it has access to every object on the Parent Controller Scope, and if the directive were to modify or add new items to the scope, then they would actually belong to the parent controller scope, not the directive. This is very straightforward and easy to demonstrate. Going back into our code, let's add a line to each controller that will logout the actual scope. Now if open up our console, you can see there are two child scopes, and if we open up one of them, you can see this $id indicates a unique ID for the scope. It's 003 for the controller scope, and for the directive scope we have the same thing, 003. Shared scope is by the far the simplest way to deal with scopes and directives. The next option is to have an inherited directive scope. This diagram illustrates this. Again, we have the same situation where the Parent Controller Scope has the User object, and we've created a directive with Inherited Scope. But now if we were to create a new item on the scope inside of our directive, that item would become internal to the directive itself and not visible to the parent scope. In other words, that item would live on the directives scope and not on the parent scope. There are several reasons why we might want to do this, the most common is just to encapsulate data inside of the directive. Creating a directive with inherited scope is simple. Going back to our code, all we have to do is add in a scope property to the directive and set it to true. That will create inherited scope. By the way, if we were to set this to false, that would be shared scope, which is the default. So let's set this to true and we'll go back to our console and look at the two scopes that have been logged out. Again, the controller scope was logged out first that has an ID of 003, and the directive scope was logged out second, which has an ID for 004. So you can see these are actually two different scopes. Now one thing that's interesting to note is that the child scope has a parent property. If we open this up, we can see that that's 003. So the containing controller scope has been set as the parent of the directive scope. You can also see that the data on the parent scope is visible. If you look right here, here is the user object on the parent scope. Now the Angular team did not arbitrarily choose the word inherited. There actually is true JavaScript 00 inheritance going on here. If we close this parent object and simply look at the prototype of the child scope, you can see that it is also that same scope object. There's our 003 and our user object is also visible. And because this is inheritance, this user object is visible to the child scope without doing anything extra. We can see that in action because our directive is still working. Even though we have inherited scope instead of shared scope, the directive scope has a user object that it can access and read exactly the same as it could when the directive shared its scope with the parent controller. Again, the main difference here is not what the child scope can see on the parent scope, but instead what happens when the child scope creates new data. This data is internal to the child scope and not visible on the parent scope. The last option is to have isolated scope. This diagram illustrates this scenario. Again, we have a parent scope that has a User object. Just as before, if the Isolated Scope creates new data on its scope, it's going to be internal and only visible to it and not visible to the parent scope. But we still have the issue that isolated scope can't see data on the parent scope. And often times we want to be able to see data on the parent scope. To solve this problem, you can actually create a special binding between the isolated scope and the parent scope on an object by object basis. This way the isolated scope can't see everything on the parent scope, but it can see specific items that you make visible to the isolated scope. Let's look at how that would work in code. If we go back to our code and we change this true to an object, we've created an isolated scope, and you can see that our directive no longer works. That's because the isolated scope cannot see the user object on the parent scope. Before we address this issue, let's look at the two different scopes in the console. So again the controller scope was logged out first, we'll open that up and we'll see it's still 003, and if we look at the directive scope, you'll see that that's also 004. But notice that this still has a parent object on it, and if we open that up, we can see that's still 003. The containing controller scope is still set as the parent of the child scope, but it's no longer the prototype. If we open up the prototype, we won't see a parent scope, and we won't see a user object anymore. Let's close this up and let's make the user object visible to the directive. We do that by specifying it on the scope object. If I add a property called user and set its value to the string =, I've now told the directive that it should expect an object named user. But I still have to specify what on the parent scope is going to be this user object. I do that through the HTML. So let's go to our index.html, and on the info card I'm going to add an attribute, user, and I'm going to set the value of that property to be the object on the controller scope that I want to be the user object on the child scope, which of course is an item called user. And now you can see that as soon as I set that up, our directive begins working again. Let's make this a little more obvious by going into our script and specify that in our controller, instead of calling this object user, we'll call it user1, and we'll go into our HTML and specify that the user object on the directive should be the user1 object on the controller. And now it's back to working again. Incidentally, this also fixes a major issue that we had with our directive to begin with, and that was that we could only put one user-info-card on any page, because any user-info-card is going to display the value of the user object inside of the containing controller. Now that I've created an isolated scope, I'm effectively able to set the data source for my user-info-card. If we duplicate this and set the second one to user2 and go into our script and create a user2 object, we now have two user info cards, one displaying Luke Skywalker, the other one displaying Han Solo. And now by using scope property on our directive, we're able to create a directive that is truly reusable.

Scope and Multiple Directives

In the last section, we looked at the three different ways that directive scopes can relate to their parent scopes. In this section, we're going to look deeper at directives and scopes. We've already seen that directives that use with inherited or isolated scopes create a new scope object, whereas directives with a shared scope do not. Let's go back to our HTML and consider a few things. Here we've got our two user-info-cards. Each of these creates an isolated scope, which is a child of the scope created by the controller. Our user-info-card directive is a custom element, but we also know that directives can be attributes. Because of this, an element can have more than one directive on it. For example, our user-info-card could also have the ng-click directive. So what happens when two elements on the same directive both ask for isolated scope? We can test this easily by creating a new directive, we'll call it my-directive, we'll go in our script file and create that directive. In this case, it's going to be an attribute, so we don't need a restrict. We're not going to have a template on it; we simply need to put on a scope property. You'll notice over on the right that the bindings are now broken. Let's go and see what's happened. You can see that we're getting an error that multiple directives are asking for isolated scope on that element. So obviously Angular does not allow us to have two directives on the same element that asks for isolated scope. That's very interesting. But what happens if one of the directives is asking for an inherited scope? Well, as of Angular 1.3 we'll get the same error. In Angular 1.2 this works, but it creates this crazy inheritance chain between the different scopes that could be a real problem if you are potentially going to be using scope variables from scopes up the inheritance chain. Now this is a very complex situation, and definitely the kind of situation you're going to want to avoid. So whenever you're creating directives that could be attributes, it's best to try to stick with shared scope if at all possible. And also when using custom element directives, try not to use other directives on the same element that will ask for an inherited scope. Now one more question I want to look at, is what happens if we put another directive on the same element that also asks for an inherited scope? This is easy to test. I've created a new plunk, and inside of here I'll create a div, and I'll give it two directives, inherited1 and inherited2. Let's go down to our script and we'll create those two directives. We'll make these super simple, we'll just set an inherited scope on them, and a link function that will just logout the scope. And our inherited2 directive will look exactly like inherited1 with just the names changed. Now let's run our code and look at the console output. Looking here at inherited2, we can see the scope is ID 3, now let's look at inherited1. And you can see the scope is also ID 3. So in the case of two inherited directives on the same element, they use the same scope. This is pretty straightforward and not a big surprise, but it is important to realize that because they used the same scope, they could easily conflict with each other by trying to write to the same variables on the scope. So this is again one of those situations that is best to avoid if possible, so here's another best practice. Try not to put two inherited directives on the same element. And if you do, make sure you're very careful about the names of the scope variables that your directives write to. Now these best practices about avoiding isolated and inherited scopes on the same element apply in most cases, but when we look at structural directives later on, we'll actually see an exception to this rule. But in general, our takeaway is that component directives should use isolated scopes, but decorator directives should probably be using either inherited or preferably shared scope.

Inherited Scopes

In this section, we're going to take a look at when you want to use inherited scope and the advantages of doing so. First off, I want to be able to collapse the entire user info card whenever I click on the user's name here. I would also like to be able to collapse just the address by clicking on the address label here. So let's start off by removing our console.log statements, and then let's go over to our userInfoCard, and we'll add in a click even on the panel heading. When these are clicks on the panel heading, I'll invoke a method named collapse. I also want to be able to hide the entire panel body, and I'll do that by adding in an ng-hide attribute, and we'll set that to a variable named collapse. And now to implement that I'll go into my script file and let's add a collapse method to our controller, and inside of this method, we'll take our collapsed variable and set it equal to the opposite value of what it currently is. I also want to default this value to false, so at the beginning of my controller I'll set that value to false, and now if we go up and click on the user's name, it collapses the entire panel. Very simple. Now let's make the address collapsible. I want to do this by turning the address into its own directive. So I'll start by creating a new directive, and I'll call it address, and this directive is going to be a custom element, so I'll restrict it to E. I'm going to give it a templateUrl property, and finally I'll give it a controller. Now we'll create that address.html file. I'll go back into my userInfoCard.html and grab the address section, remove this entire thing, and replace it with the address directive. And I'll go up into my address.html file, paste that in, do a little bit of formatting, and to implement this, I'm actually going to create two entirely separate divs, one that shows when the address is expanded, and one that shows when it's collapsed. So this will be what it shows when it's expanded, and we'll create a second div that shows when it's collapsed. We'll use the same heading. And in the collapsed case, we'll just show the user's planet, and then three periods. Now on the same restriction on this div that I want on the top one, and if the user doesn't have an address I don't want it to show up, so I'll add the same ng-show attribute. And in order to collapse and expand the address, I know I'll need to add a click event handler to both divs, so I'll add an ng-click. Clicking on this div will expand the address, so I'll call the method expandAddress, and clicking on this div will collapse it, so I'll call the method collapseAddress. And now I need to set this upper div to hide when the user clicks collapseAddress, so I'm going to adjust this ng-show to only show if the user has an address and the collapsed variable is not true. And I need to do the same thing to this div down here, so I'll adjust its ng-show to show if there's an address and collapsed is true. Now let's go pull up those methods in our script file, and we'll start by defaulting collapsed to a false value, and we'll implement the collapseAddress method, and the expand method. All right, our implementation looks good. We're setting the collapsed variable inside of our address whenever we click in the address, so let's go ahead and test this by actually clicking on our address div. Okay, now something very interesting has happened. Notice that the entire user card has collapsed. If I open it back up again, you can see the address is still there, I click it to collapse it, and it collapses the entire user info card. So you may be wondering what's going on here. Well, because we're using a directive with a shared scope, this collapsed variable is the exact same as this collapsed variable. So collapsing and expanding the entire user card also collapses and expands the address. Now that's not functionality that we want, we want this collapsed variable to be separate from this collapsed variable, which we can fix easily by making this an inherited scope. Now, if we go in and click the address, it simply collapses the address and leaves the entire user info card alone. But we can't set this scope to be isolated, because if we do, now the user is no longer visible to this address directive because the user lives on the parent directive and not on the address directive. We would have to do the same thing that we did to the user info card and actually bring the data in that we want. But in this case, because of the relationship between these two directives, we're okay if they actually share inherited data, but we want this address directive to have its own collapsed variable. So we'll go back to an inherited scope, and now the address directive can see the user object, and therefore the address within the user object, but it can have its own collapsed variable on the scope that is separate from the collapsed variable on the parent scope. It's important to realize that until we assign the collapsed variable here on line 60, there is a collapsed variable in scope for this address directive, but it's the same value as the collapsed variable for the parent. That's because of JavaScript inheritance. For the same reason, if we assign a new user variable, that's going to make the user variable for the address directive this empty object, instead of it being able to see the user object from the parent directive. Let's actually logout the address directive scope object and see this a little more visually. If we open this up, again we can see that we have our own collapsed variable right here, and the user object is visible, and if we go into the prototype we can see that it has its own collapsed variable here, and it has its own user object. Whereas if we come up here and take away this user object, so now we're just inheriting the user object from the parent directive, and we come down in here, we see that we don't have a user object on our own scope, but the user object on a parent scope is still visible, and so we can still access it as $scope.user. So inherited is very powerful, because it lets you see data on your parent scope, but it can be dangerous if you're not careful.

Isolated Scopes - Simple Value Parameters

Directives with isolated scope can take in parameters in three different ways. You've already seen how to take in data when we added the user parameter to our user info card directive. There are two other kinds of parameters that directives can receive, simple value parameters, and functions. In this section we're going to see how to pass simple value parameters into directives that use isolated scope. We'll do this by adding another piece of functionality to our user info card. Now that we can control whether the user info card has collapsed or expanded, I'd like to let the developer decide whether the info card is collapsed or expanded to begin with. Let's go into our index.html, and inside of here I'd like to be able to specify a property just like this, and that way be able to indicate whether or not the card has collapsed or expanded by default. So let's go back down to our script, and inside of here we're going to add another parameter. Just like we added the user parameter, we're also going to add a collapsed parameter. Now in this case, we're not passing in data like we did with the user object. We're actually passing in a simple value. So instead of using the equal sign, we're going to use the at sign. This tells Angular that we're going to be passing in a simple data value. In fact, we're limited to only passing in strings. So even though I've put in the word true, what we're going to get is not going to be a Boolean, it's going to be the string true. Now based on that, I want to be able to set the collapsed variable on my scope. At this point, we've got a conflict. I've got the collapsed variable right here, and I'm initializing it to false. I don't want to do that anymore, but if I leave it uninitialized, it's going to get the value that this parameter specifies, which is the string true, in which case it's going to be a string and not a Boolean value. Inside of my controller, I want to be dealing with a Boolean value, but I'm getting in a string value. So we need to translate this string to a Boolean, but we can't just convert it to a Boolean at the beginning of the function right here. If we were to do that, Angular would continue to override it with a string value. The reason being, we've defined here that collapsed is a string value by giving it the at sign. So what I really want is to be able to take in this string parameter and convert it to a Boolean. But since we told Angular the name of our variable is collapsed, I can no longer use the collapsed variable down here as a Boolean. So let's do a naive implementation and simply create a new variable, and we will call it scope.collapsedState. This will be a Boolean, so I'll set it equal to whether or not the collapsed variable is the string true. I'll also have to change the variable utilized down here in the collapsed function to be collapsedState. I'll also have to go into the userInfoCard and change it from looking at the collapsed variable to looking at the collapsedState variable. At this point, our directive works the way that we want, and you can see it starts out collapsed, if I click on it, it expands, and collapses again. But what's less effective about this is the fact that we had to rename our variable. Let's close the live preview and see how we can go back to our collapsed variable. Let's change this back to collapsed, go up to our script, change this to collapsed. What I would like to do instead is have this parameter have a different name than collapsed. Perhaps it could be initialCollapsed. Now I take this value, put it here, and set this value to scope.collapsed. Now I've got to go back into the HTML and I'll set this to initialCollapsed. And again, I've got to snake-case this, and now we'll be back to a working state, but I'm still unhappy, because I don't like the fact that this is named initialCollapsed. For the developer that's utilizing this directive, collapsed is a much better name to set the initial state of the directive. So what I want to do is be able to name this parameter collapsed, but I want the variable here to be named initialCollapsed. So there's a simple way to do that. I can simply go up in here, and after the at sign, I can specify the name that I want this parameter to have in the HTML, but instead of my controller, it will be this name. So right here I'll write in collapsed, which tells Angular that the attribute's name is going to be collapsed, but when you bring it into my controller, make it the initialCollapsed variable, which is here. And now that I've specified this, my code is back to working. We can see that we're able to expand and collapse our userInfoCard, whether the initial state was collapsed or not. So that's a way to bring in simple string values and also shows you how to rename attributes in directives. We could do the same thing over here with user. Instead of bringing in the value user, we could bring in a property named person, and of course, that breaks our directive, but we just go back into our index.html and change this to person, and our directive is back to working. So that's an easy way to set the name of the attribute to be something that works for the interface to the directive, but inside of the code, be able to use a name that works well for what you're doing in your code.

Isolated Scopes - Function Parameters

We've now seen how to take in two different kinds of parameters into directives with isolated scope. In this section, we're going to look at the third kind of parameter, which is taking in a function. In order to that, we're going to add some more functionality to our user info card. I'd like the ability to remove friends from a user. So what I want is to have a little X show up next to each friend, and if I click it, it'll ask me to confirm or cancel whether or not I'm going to remove that friend. Let's go ahead and implement that directly into this directive. So we'll go down to our directive, and after the friend itself, we're going to insert a new span, gives ourselves a little bit of room, and in this span I'm going to use the glyphicon from Bootstrap for an X, and I also want to make the cursor be a pointer when you're over the span, and when the user clicks this span I want to show the confirm or cancel buttons, so I'll add in an ng-click, and we'll call this startRemove function. And when they do click this X, I want to hide it and just show the confirm or cancel buttons. So I'll add an ng-hide, and we'll use a variable called removing. Then I'll have another span here with an ng-show on removing so that this span only shows up while they are removing a friend, and inside of here we'll have two buttons. They'll need to be extra small. And this one will be a success button with the text of Remove, and this one will be a danger button with the text of Never Mind. And of course, they need to have click events, so I'll add an ng-click handler, and this one we'll call the removeFriend function, and then I'm just going to pass in friend, since the friend object is the current object inside of this ng-repeat. And on my second button, another ng-click, and this one will cancel the Remove. Okay, now let's go up to our script and we'll add those functions. We need to make sure that removing is initialized to false, and startRemove will simply set removing to true, cancelRemove will set removing to false. And then finally we want to remove the friend, so our removeFriend function, which takes in the friend object, and that friend object exists in the scope.user.friends array, so I need to find its index. And if it's a valid index, then I need to splice it out of the array. Now let's go ahead and run our code, and we'll probably need a little bit more room, we'll open this up. And we'll click the X next to Han, but all of a sudden we can see we've got a problem. Next to each friend, the Remove and Never Mind buttons are now visible. When we click Next we don't want all the buttons visible, we just want the one for that particular friend. And if I click Never Mind, it cancels all of the them. Now if I were to remove the friend, that does work correctly, but the fact that all the Remove and Never Mind buttons are showing as a problem. If we go back to our code, we can see that the issue is this removing variable only exists once for the entire directive, yet it controls the showing and hiding of the Remove and Cancel buttons for every friend for that particular user. So what we really need to do is move this into its own directive. So let's start by creating a new directive, and we'll call this the removeFriend directive. Let's go down into our userInfoCard.html and see how we put this. We probably want it to be like this, which means that it's going to need a template. We've already got the template right here, so I'll grab this HTML, cut it out, and create a new file. And we'll just call it removeFriend.html, paste that in and format it, and let's go back to our script. So we know that this is going to be an element, and we know what the templateUrl is, and we know that we'll need a controller to handle the events. We can go up here to our userInfoCard directive and remove all of these methods and put them onto our controller. And now if we run our code, we can see that if we click on a friend, we only see the Remove and Never Mind buttons for that specific friend and not for the others. So we fixed our issue that the buttons are showing for every friend. Let's remove just one to make sure that it still works. And it does remove just that one friend. But there is a problem. If you look at this code here, you can see that the removeFriend method inside of the removeFriend directive is actually reaching up through the scope to the user into the friends object. This removeFriend directive only has to do with one friend at a time. It's really not its job to know about the entire user object and how it works. Its job is simply to know how to show and hide those buttons next to each friend, and once the user clicks Remove, this directive should not be responsible for removing the friend, because removing is an operation that is done on the entire collection, and this directive has nothing to do with the entire collection, it only has to do with one object at a time. So what I'd like to do is let the removeFriend directive notify the userInfoCard directive that somebody has clicked Remove on a friend and which friend they clicked Remove on. Now Angular does support events, so I could use their EventBus in order to notify the parent directive, but here's a much better way to handle something like this. What I really want is in the userInfoCard, when I specify the removeFriend, I'd like to give it a method to call. Just like when we use ng-click and we specify a method that we want to call, I'd like to do the same thing for removeFriend. I'd like to come over here and specify that this is the method you should call when somebody actually clicks to remove a friend, and the name of that method is removeFriend. So with that in mind, let's go back to the code, and let's implement the removeFriend method on the userInfoCard directive, which is a much more appropriate place for this method. And down here in our removeFriend directive, we're going to give it the ability to call the method that's passed in. So to do that, I'm going to create an isolated scope, and I'm going to create a parameter that will represent the method that we use to notify the parent that the user has clicked Confirm. So I'm going to name that parameter notifyParent, and since this is a function instead of the equal sign or the at sign, I'm going to use the ampersand. Now I also in the HTML named that attribute method, so I'm going to specify that name here, and that way the function named by the method attribute is going to become notifyParent on the scope for the removeFriend directive. So now I'll go back into my removeFriend.html, and when the person clicks here to confirm that they are removing the friend, instead of calling the removeFriend method, I'm going to call the confirmRemove method. Now at this point you may notice that there's an issue. I'm passing in the parameter of friend here, but this directive has no idea about the friend object. That's specified in the userInfoCard. Over here in our repeater we have a friend in user.friends, but here in our removeFriend directive, there's no such object as friend. So I actually need to remove the friend parameter from here, go into my userInfoCard, and where I call removeFriend I want to pass in what the parameter is here. That's because this method knows about the friend object, the other method doesn't. Now there's an important point to understand here. When I tell the removeFriend directive that I want it to call removeFriend with this parameter, when I go into the removeFriend directive and I say call confirmRemove, I'm not giving it a parameter. Instead in the code I'm going to implement that method and I'm going to call notifyParent. This method notifyParent is actually going to call removeFriend with the friend parameter that is the current friend for the particular node that we're dealing with. And that's the power of this ampersand right here. This binds the method here, which is removeFriend with the friend parameter, to a method called notifyParent, and notifyParent will call removeFriend with the friend parameter. I need to fix my syntax there by adding a comma here, and if we run our code, you can see that if we click on Remove a friend and click Remove, it actually removes the friend. But now instead of the removeFriend directive having to understand how to remove a friend from a list of friends, it's the userInfoCard that understands how to remove friends from a list of friends. And I was able to put the responsibility for that logic inside of the appropriate place. This is also laying a foundation for us, that instead of having one big directive that does a lot of things, we actually have several smaller directives. In our case, we have our info card, our removeFriend, and our address directive that each do a smaller piece and therefore each piece is encapsulated, and in order to work with each piece, you only have to understand a smaller set of information. This model of building directives becomes very important when building component directives, as they can become extremely large and complex, and it's much better to move pieces of functionality out into smaller components and then compose a larger directive from these smaller directives.

Isolated Scopes - Overriding Function Parameters

In this final section, we're going to look at how to override parameters when calling methods that were passed into a directive. Remember on our removeFriend directive, when we call the method parameter, we're actually calling the removeFriend function on the parent scope, passing in the current friend as the parameter. Well, there's a way that we can override what parameter we pass in when we call the removeFriend on the parent scope. Let's go back into our code, and here where we call the notifyParent method, which is what that method attribute is named inside of my controller, when this calls the parents removeFriend method, I can actually override the parameter that's called by passing in an object here. This object is going to be just a set of keys and values where the key is the name of the parameter I want to override. In this case, if I look up at the parent scope and see the removeFriend is the method that I'm calling here and it has a parameter named friend. So I'm going to go down here and add the key friend, but I'm going to override it with the value of Han. Now if I run our code, if I go in and try to remove Leia, it's actually going to remove Han, even though I told it to remove Leia. Now if there was a second parameter, all I'd have to do to override that second parameter is to add a second property and set its value to whatever I want it to be. So that's how you can override parameters when calling methods that were passed into your directives.

Summary

In this module, we saw how to add a controller to our directives, how the scope of a directive relates to its parent scope, and what happens when multiple directives have competing scope requirements. Then we looked at inherited and isolated scopes and how and when to use them effectively. This module showed us how to make our directives interact with the user in practical ways with minimal coding.

Decorator Directives

Introduction

In this module, we'll be covering Decorator Directives. Decorators are directives that add additional functionality to an existing tag or modify a display of that tag. Most of the built-in directives that we use are decorators. They are almost always implemented as attributes, and pretty much never have a template. We'll look at examples of built-in decorator directives in this module. as well as building several of our own decorators.

Built-In Decorator Directives

The first thing we want to do in order to start getting a handle on what decorator directives are, is to look at some of the built-in decorator directives that come with Angular. We're going to start with the starting point for an Angular application, and that is the ng-app directive. This is a decorator directive. It's adding new capabilities to the HTML tag that lets it run as an Angular application. Corresponding to that, we also have the ng-controller directive. Again, another decorator directive that adds new capabilities, in this case to the body tag, by adding a controller and therefore scope to that body tag. Another decorator directive that we use a lot is the ng-click directive. Again, it adds new capabilities. In this case, the ng-click adds the ability to listen to click events on the elements it's put on. In this example we're looking at a div, but this will work on just about any element. In fact, all of the event handling directives in Angular are decorator directives. For example, on an input tag, we commonly put an ng-model directive. That's a decorator. Also, directives that manipulate the display of a tag are decorators-- ng-show and ng-hide are great examples of this, so also are the ng-class and ng-style directives. All these directives add new capabilities to the tag that they're on. So when thinking about adding new capabilities to a tag, a decorator directive is what you're going to create.

The Link Function

Now that we've seen some examples of decorator directives, let's go ahead and create our own. I'm starting over with a blank angular.js project in Plunker, and I'm going to add in a couple of libraries that I know I'll need. I want Bootstrap, but in this case, I only need bootstrap-css, and I also want jQuery. Now again, we should always put jQuery before Angular, so I'll move that up, and now we can go ahead and get started. In this section, we're going to be talking about the link function on directives, and in order to show that, we're going to create a directive that will add spacebar support to HTML5 videos so that you can use the spacebar to pause or play an HTML5 video. I'll start by getting rid of this ht1 tag. I'll also add the container class to the body and give it some top padding. Now I'll add in a video element with an id of vid. I'm going to ask it to display its own controls, I don't want it to preload, and for the poster image, I'm going to give it the URL of a png image to use as the poster. Now I've got to set the source for this video and I'm going to use an online open source video to be the source for this HTML5 video. You can just copy these URLs if you don't have your own video to use, or you can use your own video if you have one handy. Now that we've got our video element created successfully, let's just see it in action. If I hit play right here, it'll go ahead and play the video, and pause it, and we'll pause the video. Of course, if I hit the spacebar nothing happens. So let's go back into our code and let's create our directive that will give us spacebar support. We'll name it spacebarSupport, and we'll start with just an empty object, and consider what kind of attributes it needs. First off, this directive is not going to be implemented as its own element. As a decorator directive, we're going to be modifying an existing element. In our case, we want to modify the video element and indicate that that video element now has spacebar support. So going back to our HTML, what I'd like to do is come into this video element, and simply be able to add an attribute like this, and have that enable the spacebar on this video. Now since we're going to be dealing with capturing keystrokes, there's something important about HTML that you need to understand. You can only capture keyboard events with an input control or with the entire document, essentially the body tag. The keyboard events will not fire on any other kind of element, so even though we're adding a spacebar-support to our video element, we're actually going to end up having to modify the body. What our directive will have to do is capture the key press event on the body, look for a space, and if so, tell the video element to play or pause. So going back to our code, we know that this is an attribute, so we can restrict it to A. Again, this is the default, we don't have to put this in, but it's nice to specify it explicitly. Now for this directive, we won't need to specify a scope or a controller. Instead what we want to do is actually modify the HTML on the page and register a key press event listener on the body. We do things like that using a link function. So I'll add a new property to my object named link. This will be a function that takes in three parameters, the scope, the element that the attribute is on, and finally, an array of all the attributes that are on that element. In this case, the attributes are going to be the ID, the spacebar-support itself, the controls, the preload, and the poster. We won't need any of these attributes for this directive, but we're going to capture that parameter anyway. Now what I want to do is register a key press event handler on the body. That's very easy using jQuery. I can simply register an event listener, and this will be my callback for that event. Now I'll check to see if the spacebar was pressed, and if so, I want to check if the video is paused. If it is, I want to play it, and if it's not, I want to pause it. So now I need a reference to the video element itself. Remember the second parameter here is the element that the attribute is on, which in our case is the video element. That's the correct element, but it's actually going to be wrapped by the jQuery wrapper. I need the raw HTML object itself. And I can get that by simply calling el, and then an indexer, and then 0. That will give me the actual video HTML DOM object. So instead of doing that here, I'm going to set a reference to the video element outside of this if, because I'm going to use this in a few places. First, I'll ask if the video is paused, and if it is, I'm going to play it. These are methods and properties that are on the video element in HTML5. Next I want my else clause, and in this case I'll pause the video, which I do just by calling pause. At this point, our code is now complete. Let's go ahead and run it, and I'll set focus inside of here, and then hit my spacebar. And you can see that the video is now playing, and if I hit the spacebar again the video pauses. So I was able to add spacebar support to my video element simply by creating an attribute that I stuck on the video element itself. Now one other interesting thing to note, is there's another way to specify a directive like this. In Angular, if instead of returning an object in a directive, you actually return a function, Angular will assume that it's a direct that has all the default properties and that the function that you're returning is the link function. So if we specify our directive like this, then our directive will still work exactly the same as it did before. If we open up our video and hit spacebar, and it's working, and hit spacebar again, and it pauses. And there's a very simple example of a decorator directive, and using the link function in order to get at and modify the underlying HTML of the document.

Demo: Adding Pause Notification to HTML5 Videos

In this section, we're going to look at another example of a decorator directive. In this case, I want a decorator directive that will notify me whenever an HTML5 video is paused. So over here in our HTML, what I'd like to do is add an attribute like this, and then specify a function to call whenever the video's pause event is fired. In this case, I'll just name it handlePause. Let's go back to our code, and we're going to implement the handlePause function on the main controller, and for now we'll just logout to the console that the video is paused. So let's create our directive. And this is an attribute, so we'll restrict it to A. And as we saw in the last module, we can pass in function parameters using the ampersand in an isolated scope, so let's create an isolated scope. And even though the name of our directive is eventPause, the name of the attribute that has the function we want to get a handle to is also eventPause. We can see that here in the HTML the attribute eventPause has the function we want to get a handle to. So in our code, on the scope we've actually got to call it eventPause. And now since we want to listen for the pause event, we're going to have to use a link function. Again, it takes in the scope, the element, and the attributes. The HTML5 video element exposes a pause event, so we'll register a listener for that event, but now we've got a new situation that we haven't seen before. The HTML5 video has fired the pause event, but that's actually happening outside of Angular's knowledge. So if we were to try to do anything with Angular, in Angular nothing happens outside of a \_\_\_\_\_ cycle. So we're going to have to start a new digest cycle, because an external event has happened. And we do that by calling scope.apply, and that function is $apply. We'll just pass in a simple function, and inside of here we'll call the eventPause function, which is on our scope. That will cause the handle pause function to get called on the parent. And now if we run our code, open up the console, if we play the video and pause it, we get the message that the video is paused. And since we have our spacebar support, even when we do this using the spacebar, we still get the same thing. The count here is now 2 because I've hit the spacebar and paused the video. Now let's go back to the code. Now let's look at what happens if we don't wrap this function inside of a scope.apply. If we don't start a digest cycle, then Angular will not update any of its bindings because it doesn't know that something has changed. So rather than logging out to the console, let's notify ourselves we paused the video through a different method. Let's come up here and create a new array called messages, and down here let's push a new object onto the array that has a text property called paused. I'm going to use an object instead of just a simple string, because I'm going to bind this in inside an ng-repeat. An ng-repeat does not support an array whose items are completely identical. Essentially that means that if you compare any two items inside of the array using triple equals, they should not return true. If we just push on strings and each string is the word paused with an exclamation mark, then they will return true when using the triple equals comparison. But any two objects, even if they have identical contents, will not return true because triple equals is an identity comparison. So let's go into our HTML, and let's go down after our video, and we'll create a div, and add an ng-repeat to that, and this will be message in messages, and we'll bind to the message.text property. And go back to the code, let's actually put back in our console.log statement, so we can see that these functions got called even if bindings aren't getting updated. And let's go down here into our link function and comment out the scope.apply and just call scope.eventPause. Now let's run our code, open up the console, and in here we'll hit play, and pause again, and you can see that it's paused, but there's no messages showing up inside of this HTML, no messages are showing up here. Let's go back to our code and we'll comment back in our scope.apply, run it again, and this time when we pause, you can see that the message paused! has shown up. And pause it again, and we get another message. So it's very important whenever you have an event that fires that Angular doesn't know about, such as an HTML elements event, that you start a digest cycle by calling scope.apply. Now let's talk about another problem that we got with this directive. We used an isolated scope. If you remember in our previous module, we saw that whenever we create an isolated scope on an element, that effects the other directives on the same element. We don't want this directive to affect other directives on the same element, and that may cause them to cease working if they rely on data being visible from a parent scope, or if we create two decorator directives that both require isolated scope and then put them on the same element, then the entire system will be broken. That's not how we want to craft our directives. We want our directives to be as non-intrusive as possible. So let's implement this directive without using an isolated scope. We'll start off by removing our scope property from our directive object. At this point, we have to figure out how to get a handle to that function that was specified in the HTML. Again, this is the handlePause function. So remember, this third parameter over here is the attributes on the element that this directive is on. So, if we access attrs and then the name of this directive, which is eventPause, that's going to be this string value right here, handlePause with an open and closed paren. So we're part of the way there, but we still don't actually have a reference to the actual function on the parent scope. In order to get that, we're going to need the parse service from Angular. So I'm going to have my directive bring in the parse service, and now I can come down here and I can create a new variable, and call parse, and pass in this string. At that point, Angular is going to go and find that handlePause function on the parent scope and give it back to me. Now I can no longer call scope.eventPause, because that's not the name of the function anymore. I could call scope.handlePause, but that would create a very tight binding between this directive and the parent directive up here, because we would have to know the name of the method on the parent directive down here inside of this eventPause directive, and I don't want that to be true. So instead, I'm going to call that fn function, and the first parameter I need to pass into that is the current scope. And now I can pass in an object, which has a list of parameters to set when it calls the function. This is just like when we overrode function parameters when using an isolated scope. In this case, I'm going to create a parameter called evt for event, and set it to the event object, which is this event object right here. The object represents the pause event on the video element. Now that I've named it evt, I've got to go back into my HTML and pass in the parameter evt. Now up here, I can receive this parameter using any name I want, and I'm not using this yet, so let's just log it out to the console, and now let's run our code. So I'll go hit play and then hit pause again, and now you can see that out in the console I've gotten the event object logged out to the console, and the message paused. And if I scroll down I'll see that the paused message has shown up, and the bindings are firing and getting updated correctly inside of Angular. So there is how to create a decorator function that accesses a function that's passed in as a property without creating its own isolated scope. This is a very important thing to be able to do so you can create small decorator functions to encapsulate small pieces of functionality and thereby reduce the complexity of your overall system by composing your system from small, simple pieces.

Demo: Recreating ngClick

In this section we're going to learn more about decorator directives by recreating the ng-click directive. I'm going to start with a blank Angular 1.3 project. And I'm going to modify the HTML a little bit. I'm going to leave in the h1 and add in my own h2 with a binding to show a message. And I'm going to add to the body my new directive, which will handle click events, which I'll call my-click. This will work exactly like the ng-click in that I'll pass in a function to call. Now I'll want to make sure that I can pass parameters into my function, so I'm going to pass it a data parameter, and inside my code I'm going to add that data object to the scope on the main controller. This will be an object with a message property, which we will initialize to the string "I have not been clicked." Next, I'll implement the clickHandler function, which is passed to the my-click directive, and that will receive one parameter which is the data we're working with. I'm just going to name this parameter P, so that we know that it's separate from the scope.data object, and in here I'll set its message property to be the string "I have been clicked." So even though in this simplistic example we're just taking the data property off the scope here and we're passing it into the clickHandler here, and we know that it's the same object, we're still going to rename it to prove that we can work with parameters passed to our decorator directive. Now let's implement that directive. And as we saw in the last section, we're going to need the parse service. Then we'll return our directive object, we'll give it a link function, and just like in the last section, we'll create a function by asking Angular to parse out the attribute. Next we'll register an event listener for the click event on the element. Then we need to make sure that the bindings get updated by calling scope.apply. And finally within here we call our function passing in the scope. Let's go ahead and run our code. And if I click anywhere in the body, we can see the message gets changed to I have been clicked. So here you successfully recreated the ng-click directive, and you can see it really didn't take very much code at all in order to do that.

Business-Specific Decorators

In this section, we're going to talk about and create one that's fairly business specific to the case and probably wouldn't be very reusable, but it will show us how to take functionality outside of a bigger component directive and put it in a more business specific decorator directive. So let's start again with the plain Angular 1.3 plunk. And this time I'm going to take out my h1, and I'm going to create a new component directive that's very simple. I'll call this user-tile. And it will need a property that tells it which user to display. This will be very similar to our userInfoCard directive, but a little bit simpler. Let's implement this directive by creating a new directive. And we'll make it very simple. We'll restrict it to just an element. We'll create an isolated scope, passing in the user object, and lastly we'll give it a template. Now let's create that template. And this will be very simple, we'll just create a div that has a class of well, and it has an h1 that displays the user's name. Now let's go back into our script and let's create the user object that will be displayed. And this will be just a simple object that only has a name. And now if we run our code, we can see our user-tile in action. Now the business functionality I'd like to implement is that if a user comes in and clicks on the user-tile, then I want this user to become highlighted by changing the font color of the name to be red instead of black. And if the user clicks again, we'd like to unselect him by turning the red back to black. And to reflect that state change, there should be a selected property on the user that's set from false to true and back to false. So let's start by adding that selected property to our user. And we'll default it to false. Let's go into our style and let's create a class that will set the foreground color to red. Now we'll go into our userTile.html and we'll add in the ng-class directive to add that hilight class if the user's selected property is true. And we can test if this works by going back into our script and changing the selected value to be true. And you can see that part is working, so let's put it back to false, and talk about how we would implement this functionality directly into our userTile controller. While we would probably just go down and add a controller to the directive, and on that controller scope we'll create a function called select, and in here we'll change the user selected property to be the opposite of what it is. So now we just need to go into our userTile and add an ng-click directive that calls the select function. And now we can test that this is working, and we've got our functionality. So let's go back to the code and look again at what we've implemented. In order to get this functionality working, we've only added a little bit of code, although in a real production environment, this actually might be a lot more code. Instead of just a controller and 3 lines of code, it might be more like 5 or 10 or even 15 lines of code, because you might be checking different states and in order to select the user, you might be validating some other things, the code could easily be more complex than this. And if our directive was already pretty large, then adding that additional code could really be pushing the size of this directive and just have it be doing too much. I'm sure you can easily imagine a situation where you end up with a directive that is several hundred lines of code long. So let's talk about how to take this business specific code out of the userTile directive and put it inside of its own business specific decorator directive. First let's go into the userTile partial, and we'll take out the ng-click, and instead we'll add a new directive called user-click-select. This directive will handle the functionality for when the user gets clicked, they also get selected. Now let's go back into our code and we'll create a new directive. And this directive is going to be very simple. We'll just create a link function. And inside of here we'll listen to the click event on the element, and when that happens, we will set the selected property on the user. Again in a more realistic case, this code could be quite a bit longer. And the last thing we'll need to do is call scope.apply, since Angular doesn't know that this event has fired. Now we can go up and remove this select function and the entire controller from our userTile directive, and we'll run our code and validate that it's still working. And indeed we can see that things are working. So if you compare how the code looked when we implemented the functionality inside the userTile directive versus implementing the functionality in a business specific decorator, we don't really see a very compelling case in this scenario. But this is an extremely simplistic scenario. If our userTile was already 80 or 90 lines of code, then adding on another 5 or 10 lines of code would be something we may not want to do, and that's a good case where we would want to put this sort of functionality in a business specific decorator directive instead of just adding it into our main component directive.

Manually Creating Watches

In this section, we're going to build a decorator directive that modifies the display of an element. What we want is a directive which will allow us to control the size of a font being displayed. For example, we might have a div, and inside that div is some text, and when that text displays we want to be able to change the size of that text dynamically based on another value. So we need our directive to manipulate the font size attribute, and instead of controlling the pitch size, we'll use a percentage. So let's call our directive font-scale. And even though in a realistic scenario we would probably bind this to a more interesting value like a slider or a numeric value associated with some data, instead we'll just create an input box that we can set the scale directly. At this point, I want to bind this input box to a value that will control the size of the text in that div that my directive is on. So I'll name that value size and then I'll go up into my font-scale directive and I'll give it the size variable so it knows to bind to that value. And just for reference purposes, let's also add a div that has some unscaled text. Now let's go down into our code and let's create that directive. And of course we start with the typical code for directive. Since this directive is an attribute, we don't need to set to restrict, we can leave it at its default value. We just a link function. Again taking in the standard three parameters. Of course, as I've showed before, I could just be returning the link function itself rather than returning an object, but again, it's often better to just return the object. It doesn't add very many more lines of code, and later on as you maintain this directive, it's easier to add other attributes to the directive itself, and also anybody who's not familiar with the fact that returning a function works for a directive won't be thrown off when they see it. I also want to set up the default value for that size variable, which I'll do up in the controller itself, and we'll default it to 150, which will mean 150%. Now back into our code, we now need to set up the ability to change the size of the font as that size value changes. So in essence what we need to do is create a binding. So as one value changes, Angular will make the other value change correspondingly. Thankfully, Angular gives us this facility, and that's called a watch. Whenever you create a binding inside of HTML using the double curly braces, like this, under the hood is what Angular actually does is set up a watch. The watch will get notified anytime the size value changes, and then any dependent display, which would be this binding right here, for example, will get updated to the new value. And that's exactly the process of how bindings work in Angular. At this point, anytime we change this value, it's updating the rendering for that display, but under the hoods it's using a watch. So let's clean this up, and we'll go back into our code, and we'll create our own watch, which is something we can do with any scope object by calling scope.$watch. At this point we create an expression that we want to watch. The first parameter to this function is the value that we want to watch. In our case, we could just type in size, but that would bind this directive to the implementation up here in the controller, and we don't want that to be true. Remember in our HTML, we actually told it what the variable is we want to bind to by setting it here. So we want to take advantage of that, which we can do by using the attributes parameter that's passed into our link function. So we can ask for the attribute that is our directive itself, so that would be fontScale, and that will return the value of our fontScale property, which again is right here. And our second parameter is a function that will run anytime the watched value changes. This function takes in two parameters, the new value and the old value. Now in our case, I don't care about the old value, so I'm going to remove the old value from the parameter list, and now in here I want to set the font size on my element, which I can do with the css function since the element is a jQuery wrapped object. And here I'll set the font size, and I'll set it to the new value, and I'll append on a percentage sign so that it's set to that percent. And that's all we have to do in order to implement our directive, which will now watch the given value and set the font size correspondingly. Let's run our code, and let's go in and change this 150 to 250, and you can see that the font size has been scaled up correspondingly. And if we change it down to just 50, it's now 50% as big. We'll go back to 150, and we're back where we started. Now there's something you need to watch out for. This function will run anytime the value it's watching changes, but only if the new value is different from the old value. That means it won't necessarily run very often, but if your watch is going to execute very often, you'll want to make sure that the coding inside the watch is very efficient and runs quickly. So, for example, if you bound this value to say the horizontal position of the mouse on the monitor, as you move that mouse around, this watch could get fired hundreds of times per second, if not thousands. If this code was even remotely complex, that could quickly cause a performance problem. So be aware of things like that.

Demo: State Display - Naive Implementation

In this section, we're going to look at a more complex example of a decorator directive. We're going to build it in three stages. First, we'll implement it directly in a component directive without creating a separate directive, then in the next section, we'll pull it out into its own directive and show the benefits of doing that. And finally, we'll adjust it again and make it even more reusable. We'll start in this section with a naïve implementation that we'll simply put inside of our component directive. So let's talk about the functionality that we want. What we want to be able to do is set our user into different states, and we want to reflect the state by changing the background color of the panel body for the user-- this div right here. There's a lot of different reasons in order to do this. For example, in an application you might have users moved through several different states, and you might want to visually indicate which users are in which state. A background color is a great way to do this. Another reason you might use functionality like this is if user objects can go into a state where they need to have something done to them. For example, a warning or error state. So for our example, we'll do something like that. We'll create a hilight level that's either 0, 1, or 2. If it's 0 the background will be white, if it's 1 the background will be yellow, and if it's 2 the background will be red. So let's start by implementing that functionality directly into our user directive. Now in a typical scenario, this would probably be based on some data element in the user itself. For this demo I want to be able to manipulate that value, so I'm going to add a button at the bottom of my component. I'll add a click event, which will set the user into the next state, and the text of this button will be Next State. Now we'll go into our code, and down to our directive, and I'm going to add a link function to this directive. Now you may be wondering why I added a link function rather than just implementing this functionality in the controller for this directive. The reason is, I need to manipulate the HTML itself and change the background color. That's not something that I can do readily within the controller-- even though technically that's possible, it's not a good idea. That's not what controllers are for. That's what the link function is for. The element itself is passed in here with the el parameter, and through that we can manipulate the DOM itself. So the link function is where we want to put this kind of functionality. Now I need a value to bind to, so I'm going to create a level value on the user. For Luke we'll give him a level of 0, and for Han, he'll have a level of 1. Back down in the directive, I'm going to implement the next state function. Now again, typically a function like this, which is an event handler, it's something you'll usually implement down in your controller. In our case, we need to implement it in the link function so that we have access to change the DOM. When the user presses the next state, the first thing we'll do is increment the user's level, and then if it's 3 or over we'll wrap it around using the modulus function, and finally, we'll switch on the level, and if it's 0 we'll set the background color of the panel body div to white. And if it's 1 we'll set it to yellow and 2 we'll set it to red. Finally, we need to add a comma after our link, so now our code should be ready to run. Let's go ahead and run it. And right off the bat we can see that the Han Solo background should be yellow, if we click Next State it goes to red, white, yellow again, and Luke Skywalker works correctly and it's a dependent of Han Solo. So our problem is that we're not setting the state initially. So let's go back into our code, and this we can handle by taking this section and making it its own function, and then we'll just call it here, and we'll also call it initially down here after everything is set up. Open that code again, and we can see that the background is now set correctly and we can still rotate through. So there's a simple implementation to reflect the state of the user in the background color done directly in the component itself. In the next section, we'll move this to its own directive and see the advantages of doing so.

Demo: State Display Directive

In our last section we implemented our desired functionality directly inside of our userInfoCard directive. From a certain standpoint this is fine. The functionality about highlighting the state of the user seems to belong directly to the userInfoCard directive. This could possibly be functionality you would never reuse in the rest of your project, but just because this functionality belongs specifically to the userInfoCard directive, doesn't mean it has to be added to the userInfoCard directive itself. Creating another directive that will be consumed by the userInfoCard has a lot of advantages. A good comparison would be thinking of a classic object oriented language, like Java or C#. If you were to write a class that represents a user, there's no reason to put all the functionality for that user inside of the main class, and in fact, there's many reasons not to. Creating smaller classes that are consumed by the main class is a great way to encapsulate functionality, which will reduce maintenance costs. So let's see about implementing our functionality inside of its own directive. In addition to encapsulation, we also have a coupling between the display logic, right here on these lines, and the business logic around the users level. Right now our implementation is a button that simply rotates the level variable through the states. Now let's implement the functionality as its own directive so we can get an idea of the advantages that gives us. So we'll start by creating a new directive, which we will call stateDisplay, since it displays the state of the user. This directive is going to be an attribute, so I won't set the restrict property, but it does need a link function. And I'll go down into my userInfoCard directive and grab this switch statement here, and remove it, and I'm going to delete this function setState, and I call to it there and there, and I'm going to put that switch statement right up here. At this point, you may be thinking that I want to also grab this next state function and put it up here, but I don't want to do that because the next state function is an event handler on a button that actually belongs directly to the userInfoCard directive itself. The method in which we change the user's level has nothing to do with the directive that displays the state. Right now this is a button, but it could be a dropdown box, or a set of radio buttons, or really anything else. So we're going to leave that implementation on the userInfoCard itself, but what we want to do is watch that variable and change the background color when that variable changes. And as we learned previously in this module, when we want to watch a variable and take an action when it changes, we can use the watch function. But what do we want to watch? Let's go into our userInfoCard.html and write the directive in the way that we want it. Let's put this directive on the panel body div, and again it's called stateDisplay, and here we'll tell it which variable we want to bind to, which is user.level. While I'm in here I'm also going to format our button so it looks a little nicer. And now that I've specified which variable I want to watch in my attribute, I'll tell the watch to watch that variable, which is attrs stateDisplay. I've also have to change this switch, we're no longer switching on the scope.user.level, instead we're switching on the newVal. I also need to change the implementation of actually setting the background color. I'm no longer looking for a sub-element on my element, I'm instead looking for the element itself and setting its background color property. Same thing here and here. And now I can go down into my userInfoCard directive and I can take this nextState function and actually move it into the controller, since it's no longer doing anything to manipulate the DOM, change this to a dollar sign here, here, here, and here, and now I can get rid of my link function. And with that we can run the code, and our directive is working. So let's go back to the code and talk about this just a little bit. Our userInfoCard directive now only has the logic that has to do with manipulating the user's level. Our new directive now has all the logic about changing the display. Our new directive is also now somewhat reusable. We can stick this on any element, and as long as we bind to a value that rotates between 0, 1, and 2, it will change the background color of the element that it's placed on from white to yellow to red and back again. But by far the biggest benefit that we've gained is the fact that the display logic is now encapsulated within its own directive, which simply worries about display. We've kept our userInfoCard directive tight and focused, and we've \_\_\_\_\_ functionality out to its own formatter directive. This is definitely an improvement in maintenance.

Demo: Reusable State Display Directive

In this section, we're going to make our stateDisplay directive more reusable. Right now there's a couple parts about it that would make it very difficult to reuse with anything but the userInfoCard directive. First off, we've limited ourselves exactly three states, 0, 1, and 2. If we decide to add a fourth state, we've actually got to come into this directive and add a fourth case. On top of that, the colors that our backgrounds turn to, white, yellow, and red, are specified here in the directive. There's no way we can use other colors. So if, for example, somewhere we wanted to switch from pink to blue to green, that wouldn't be possible using this directive. So let's see if we can fix those two issues. Going back to our userInfoCard.html, we can see that here in the directive we've already specified the variable that we want to bind to as user.level. Well, why can't we also specify the colors that we want to rotate through, white, yellow, and red, and for that matter, be able to specify any number of them and have it simply rotate through that many states? So long as this never corresponds to how many values the user.level can have, we would then have a much more customizable directive. Let's implement that by going back to the code, and we'll make a few changes here. Since all of the information that I need is now in a space delimited list inside the attribute value to my directive, first thing I want to do is take that space delimited list and turn it into an array. So I'll create a new variable called parms, and that will be an array based on that list, which I can do by grabbing that string, and splitting on space. And now I know out of this list the very first value is going to be the value I'm binding to, so I'll create a new variable to capture that, which I'll call linkVar, and that will be the first value from that array. And now for my scoped out watch, I'm no longer watching on the entire value of that property, I'm just watching on the linkVar. And now instead of doing a switch, I'm just setting the element's background-color to be the matching value from that parms array, which is going to be parms and then the newVal is a 0 based index, and since element 0 was my linkVar, I've got to add one to that to find the matching element inside of the parms array. Now I can remove the rest of this code, and if we run this, we can see that our directive is working. And what's really nice is if we get a fourth state, all we have to do is come down in here and change this to modulus on 4, and go into the userInfoCard and add another color. In this case, after yellow we'll do orange, run a code, and now we have another state inside of our mix. So now I have a much more reusable directive that can rotate between any number of states and change the background color. But can we make this even more reusable? What if instead of setting the background color, we actually rotated between a set of css classes? If you're feeling adventurous, go ahead and take a moment now and try to make this implementation change yourself, and then come back and check against my solution. To solve this problem, we'll start by going into our style sheet and specifying a set of styles. Let's go with a user-ok style, and this we'll just leave blank. We can use the default display to indicate that things are okay. We'll create a user-warning class. And in this case, instead of setting the background color, I want to set the foreground color, but it will be up in the panel heading. We'll set that to yellow, let's create another class called user-problem, which we'll use the color orange, and finally user-error, which will have the color of red. And of course I could do anything here. I'm no longer limited to changing just one css property, I could radically change the display of the userInfoCard based on the class that's assigned to the element. Now let's go down to our userInfoCard, and let's take the stateDisplay off of this element, and instead I'm going to add a span around my user.name, and add the directive to that, put that around user.name, and then instead of white, yellow, orange, and red, we'll have user-ok, user-warning, user-problem, and finally user-error. Now that I've got the HTML the way that I want it, I'll go into my script file, and I want to create one other variable up here, and this will be the list of classes, which is all the elements in the parms array minus the first value, which is the value we're binding to. So in order to get that, I'll create a classes variable and set it equal to the parms array, and I'll slice it and by putting in a 1 that will slice out the very first element and leave me with all the remaining elements. Now I'll go down into my watch, and here instead of setting the background color using the css function, I'm actually going to use the addClass function to add the corresponding class from the classes array. Now the problem here is I'm only adding classes, so if I have the user.ok class and I add the user.warning class, we'll now have both classes on the element. So I want to remove any old classes. And it's easily done by calling el.removeClass, passing in a space delimited list of all the classes I want to remove. In this case, I'll just take the classes array and join on space, and this is a bit of a shotgun approach since it will tell the element to remove all of these classes when it should only have one of those on it, but it's a simple way to get done what we want. So now let's run the code. And now our display shows the user state up here in the panel heading. And by setting it to the Next State, we're rotating through the displays. This final implementation of our directive gives me a lot more power and flexibility. At this point, since I can just specify the classes that I want, and I can specify any number of them that will match the variable that I'm binding to. And what's surprising is the number of lines of code that we required really wasn't that many. So being in the habit of looking for formatting pieces of functionality on your component directives and pulling them out into their own decorator directives is a really great way to encapsulate functionality and potentially create new, reusable directives.

Summary

In this module, we looked at several examples of built-in decorators, and then we went and built our own decorator, learning how the link function works, and saw several examples of this. After that, we recreated the ng-click decorator directive, then we looked at business specific decorators as a way to take functionality out of bigger component directives and move them into a kind of sub-directive. Then we looked at how to manually create watches, and finally, we saw several more examples of how to create decorator directives. Decorators are a great way to create small, reasonable pieces of functionality. Whether they're business specific or more general, they're a great way to componentize your functionality.

Transclusion

Introduction

This module will be all about Transclusion. By far, the hardest part of transclusion is the name. It's such a weird word and it sounds so esoteric that it probably causes more confusion than the actual concept itself. By building some directives with transclusion, we will get a good foundation of what it is and how it works. By the end of this module, you will have no problem understanding and using transclusion in your own directives. Before we get into actual code, it's a good idea to look at what transclusion is from a conceptual standpoint, so that we can lay the groundwork for our later discussion. The best way to think about transclusion is to think of a picture frame. A picture frame has its own design and visual space. When someone chooses a picture frame, they care about how the frame itself looks, but the frame is completely separate from the picture that will go inside of it. We can decide what picture we will put inside a frame and we can change the picture if we need. This is just like transclusion. When it comes to Angular, we have some kind of controller with its scope, and inside of that we will place a directive that supports transclusion. This directive will have its own display and functionality, represented here by the blue area. With a non-transcluded directive, the inner display of the directive is decided by the directive itself, but with transclusion, just like a picture frame, we get to decide what goes inside the transcluded directive. This visualization is useful as a basis for how transclusion works, but there are certainly more details to it. We'll look at those as we get to the code in the following sections.

Basic Transclusion

Let's implement our first directive with transclusion by creating a very simple directive. What I'd like to be able to do is wrap some content in a simple box using the well class from Twitter Bootstrap and have a close icon in the upper right corner that when clicked will hide the element. I'm going to call this directive displayBox, and I'm going to start a little differently by creating the template to begin with. So let's think through what kind of HTML we'd want. We'll start with a div with the class of well, and I'd like to limit it to a specific width, so I'll use a style attribute, and we'll just set it to 350 pixels. Now of course in a production application we'd probably make this is a class, but for convenience sake we're just going to use an inline style. I also need a close icon, so I'll put a little div, and I want this to float to the right. And I'd like it to sit up high inside of the well and out of the well have some padding, so I'm going to set the margin-top to -15 pixels, so it'll move it a little bit up inside of the well. Within that div, I'll have an i tag, which will be my icon, and I use the remove icon. Now when somebody clicks this icon, I want it to close the window, so I'll add an ng-click, and call a method named close, and I also want people to know that they can click this icon, so I'll change the cursor to a pointer, and finally I want the placeholder for whatever content is going to go inside of this directive. So this will be just a simple div. Now we'll come back to this HTML in just a minute, let's go down to our script file. Let's create our new directive, let's call it displayBox, I'm going to restrict it to an element, and we know what our templateUrl is, we need a controller to handle the click event, and let's control the visibility of the directive with a variable called hidden, which will default to false, and let's implement our close method, which will simply set that variable to true. Now in order to get this directive to support transclusion, we have to add the transclude property to our directive definition object. And that's simply transclude, and we'll set it to true. Now we have to tell our directive where it's supposed to be putting in the transcluded HTML, so back up in our template, we'll take this out, and tell it by adding the ng-transclude directive. Now it's customary to have this directive as an attribute on a div, but as of Angular 1.3, it can actually be its own element, like this. The different being, when it's an attribute on a div, then you're going to have a div wrapped around your transcluded content. When it's its own element, there won't be a div wrapped around you transcluded content. This may or may not make a difference to you, and of course, you can also put it on something besides a div. For example, a span or even a section, but it is most common for it to be a directive on a div, and that's how we'll mostly use it in this course. Now the final thing we need to do is control the visibility of our directive, which we'll do with an ng-hide directive, which we'll set to hidden. So again, even though transclude itself seems to be very complex, utilizing it is extremely simple. Just adding the ng-transclude at a certain point will mean that this is where our content will go. Now that we've got our directive set up, we can go into our index and actually add that directive. So it's a displayBox and inside of it let's put in a div that will show a message, and let's put in some other HTML, we'll do an unordered list. Now let's put in a binding for some data, and let's go into our script file, and inside of the containing controller, we'll set that message variable. Now if we ran our directive, we can see that everything is working as we anticipated. We have our static content up here and we have our bound content down here. And if we come in and click the close button, it hides the element. And there we have the implementation of an extremely basic directive using transclusion.

Transclusion and Scopes

Now when creating directives with transclusion, it's interesting to look at how the scopes work. Going back up to our index.html, we could probably guess as to where there are scopes. We know that there's a scope here on the mainCtrl, but is there a scope here on our directive? Does it share scope, does it have inherited scope? Let's try to override the value of the message property and see what happens. Let's go back to our script, and down inside of the controller for the directive, let's set that message property to a different value. Now let's run our directive. Let's see what's happening by logging out the scopes. Now let's run our directive and look at the console output. We can see we logged out two scopes. So looking at the controller, we can see it has an ID of 2. Let's look at the directive, and it has an ID of 2. So they're sharing the same scope. That's not surprising, since our directive doesn't have a scope property on it. So the scope of our transcluded directive operates just the same as the scope on any other directive that we create. This is both good and bad news. The good news is we have the control; the bad news is we might accidentally overwrite data elements that are important, as we've seen here in our demo. So how do we protect against this? Well let's see what happens when we add a scope property to our directive. Let's start out by adding scope true so that we're inheriting scope. And you can see from the results as soon as we've done that, our messages are no longer getting overridden. That's very interesting, because when we look at the HTML, it doesn't seem obvious that that's how this would work. Again we know we have a scope up here on the mainCtrl, and since we added the scope property to our directive, we know that there's a scope right here, but since it's simply inheriting scope, we would assume that these data elements would also be inheriting scope. But that's obviously not what's happening. Let's see if we can figure out what's happening by looking closer at the scopes. I want to look at the scope both on the body and on the displayBox, and a scope that applies to these elements here. Now we're already logging out the scope on the body in the displayBox, but I'd like to also logout the scope for this data here. We can do that by adding another controller inside of our displayBox, and wrapping around all this HTML here. We'll go down to our script and implement that controller, and here we'll simply logout its scope. Now let's run our code and open up our console. Okay, so again let's look at our outer controller scope. It has an ID of 2. We'll look at the directive scope, it has an ID of 3, and its parent has an ID 2. We can see that its parent has the message property, where our scope has an entirely different message property. Now let's look at that inner controller. It has an ID of 5. If we look at its parent, its parent has an ID of 4. That's a new ID that we haven't seen. Notice that the 4 controller doesn't have the message property on it. Let's look at its parent, and its parent is ID 3, which we know is our directive. But if we look at the prototype, we can see that the prototype is ID 2. That means that this additional scope is actually skipping the scope on the controller in its inheritance chain, and that's how the scope inside of our transcluded directive is getting the correct message property. It's completely skipping and ignoring the scope on our controller. So that's really cool. If we ever have a directive that is transcluded and it has a scope property, then we know that the contents inside of that directive have a new scope, and that scope skips the transclude directive in its inheritance chain. That way the transcluded directive cannot overwrite data and cause problems. From this we can create a best practice for whenever we're creating directives that use transclusion. Anytime we create a directive with transclusion, it should at least have inherited scope, if not isolated scope. I hope you found this section to be as fascinating as I did. Seeing how the scopes work on a directive with transclusion has really taught us a lot about the power of scope inheritance in Angular.

Demo: Questionnaire Directive

In this section, we're going to show a fairly typical usage of transclusion that also suddenly makes a little bit more business sense than what we've seen so far. So we'll start off with a fresh Plunker. jQuery and Bootstrap are at the ready. And what I want to do is create a question directive that allows me to really customize how the answer is displayed, so the display of the text of the question can be very standard, but the answer needs to be highly customizable. This is difficult to do in a typical directive, since it's hard to anticipate all the ways that you might want to answer a question. By using transclusion, we can allow the developer to specify the HTML for the answer without having to specify all the rest of the HTML. So let's get a sense for what the HTML should look like so we know how to create our directive. Start out by changing this to Questions, and for the question itself we'll use the standard Bootstrap panel. The header will be the text of the question, and the body will be where we answer the question. So for example, we might have a simple input, or we might do something crazy like radio buttons, or we might get even crazier and mix radio buttons and buttons. So now that we have an idea of what it looks like, let's go down and create the directive itself in script. And your directive will be named myQuestion. I'm not going to name it just question, since then it wouldn't have a hyphen in it and it wouldn't look like a directive in the HTML. So again, this is just like creating your own prefix for naming your directives. This will be an element. Obviously we're going to use transclusion. It'll have a template. And we'll specify an isolated scope. And the one parameter that we want is the text of the question. Next we'll create our template, and let's go into the index and just grab this HTML right here, and we can use that for the template. Do a little formatting. And obviously we're not going to use static text, we're going to want to bind. So we'll bind a variable called questionText, and this panel-body is where we want to input our own HTML so that'll be where we put the transclude directive. And now that we've got that created, we can go into our index and start specifying questions. And remember our questionText comes in and it comes in as a variable named q. And inside of our directive, we put in the HTML for the answer. Now when it comes time to specify our ng-model, this raises a very interesting problem. If I just specify a variable here like say baseLocation, and then I go down to my script, what I want is for that to set a variable on the mainCtrl scope. And indeed if I were to type in a default value here, and run the code, we can see that the default value has indeed been specified, but I have an issue. Let's go back up to our index, and outside of the question I'm going to just put a little binding to baseLocation. While I'm at this, I want to format this a little bit and put a container class on this, so that it looks a little better. Now we can see that they're both Yavin 4, but if I go in here and start typing, it's not updating the binding. The reason that the other binding isn't getting updated is because we've created a brand new scope inside of the my-question directive. So essentially the HTML here is an inherited scope to this area here. And since it's inherited, I'm overwriting the value with a brand new value, but only on the child scope, but not on the parent scope. In order to avoid that situation, we just add a dot. So go back into our code, and we'll change this to answers.baseLocation, and the same thing on our ng-model, and we'll go into our script, and we'll set a default, this is an object. And now that I've changed the binding to the property of an object, if I go in and change the value, then we're not overwriting the value with a new value, instead we're just changing the property on the answers object itself. This is a very important point to understand about Angular and inherited scopes, so if you haven't seen this before, this is definitely something you should make sure you understand. If you need more information, in my course on Angular Best Practices in the module on dealing with scope, I've got an entire section devoted to understanding these problems with inherited scopes. So let's go back and let's continue authoring more questions. And this time we'll use radio buttons. And let's do one more question. And this time we use buttons and we'll really customize the display of them. And of course to make this work so these answers did something, we probably added in some kind of an ng-click, and I'm not going to implement anything right now, because that's not \_\_\_\_\_ to the discussion, but that's pretty straightforward and something you could do on your own. So let's also go up and remove this binding and run our code. And so you can see because we've been using a directive with transclusion, we've allowed ourselves to really customize the look of how each question is answered.

Reusing HTML With Transclusion - Problem

In this section, we're going to look at one of the benefits of transclusion, which is the ability to reuse HTML. We're going to go back to our userInfoCard demo and extend it a little bit. We've discovered through some requirements gathering that our system no longer needs to support just humanoid users, but we also need to support droid users. We'll first look at the problem and implement a naïve solution, then we'll implement the solution using transclusion. Let's start by going into our script, and let's see what the data for a droid might look like. Droids have a name, but unlike users, they don't have an address, instead they have a list of specifications. This includes a manufacturer, a type, and a productLine. Droids have additional information, which we won't specify here, but this might be things like a list of owners and other data. So we can see that the data for a droid is quite different than the data for a humanoid. Because of this, the userInfoCard directive is no longer sufficient, since it won't display the data adequately for both droids and for humanoids. So let's do a little bit renaming in order to reflect the fact that we now have two uniquely different kinds of users. Instead of using the long word humanoid, let's start by renaming these two pieces of data from user1 and user2 to person1 and 2. Let's rename our userInfoCard directive to personInfoCard, and change the templateUrl, and everywhere we use the word user, instead we'll use the word person. We need to rename our template. Inside of our template we need to change user to person. We'll leave our display classes alone since they should work for both kinds of users. I want to go in and fix the data in the index.html. (Typing) We also have to fix the bindings in the address template. (Typing) And we missed one place in our info card directive. We actually need to set this from user to person. And now let's run our directive and make sure that everything works. And our address is showing up. We can collapse it by clicking on it, we can collapse the user card by clicking on it, we can change the state, and we can change the rank. We can also remove their friends. So our directive is working now. Let's see how we might add a droidInfoCard directive. We'll simply start by taking our personInfoCard directive and copying it, and since a droid doesn't have a rank, we won't worry about this function. Since a droid doesn't have friends, we won't worry about this function. Instead of person, we'll use the data point called droid. We'll change the name from person to droid here and here. And now let's create that template. And we'll just copy the personInfoCard.html. And everywhere we use the word person, we use the word droid. And of course droids don't have friends, so we can remove this section. Droids don't have a rank, so we can remove this, and the button as well. And they don't have an address, so we can remove this directive. And let's just replace this with a very simple h4, that says what the specifications are, and then we'll list out the three data points. And let's just double check that everything looks good. Put in a name here. And let's go back to our script and double check that that's all correct, and oh-- we need to use the word droid down here. And now let's go up to our index and let's actually add an instance of this directive. Now let's run the code and see if it's working. And the display is correct, the collapse works, the Next State isn't working, and that's going to be because we did not give the droid a default level. All right. And now if we click Next State, that works. So here is our naïve implementation. Let's look at some of the drawbacks of what we've created. First off, our directive here, this droidInfoCard directive, has actually got a lot of code that is completely identical to the personInfoCard directive. The collapsed, the Next State, in fact, essentially all of this code is completely duplicated between this directive and this directive. Let's also look at the HTML. Our personInfoCard has a lot of HTML that has to do with displaying the heading, hiding, and collapsing, and if we look at our droidInfoCard, they've essentially got the same HTML, right here and right here. So again we've got a lot of duplication between these two directives. In the next section, we'll look at how to solve this by adding a directive that uses transclusion.

Reusing HTML With Transclusion - Solution

In the last section, we created our droidInfoCard directive, and now we've got a problem, because we're duplicating a lot of HTML and functionality. If we go back and forth between the droidInfoCard partial and the personInfoCard partial, you can see that they're duplicating a lot of HTML. If we look at the code for these two directives, you can see a lot of the code is duplicated as well. Basically everything inside the droidInfoCard is completely duplicated here in the personInfoCard. We'd like to remove this duplication, and we're going to do this by introducing a new directive that uses transclusion. This is going to be a really interesting section where I make use of the things we learned about directives and transclusion with Angular. So let's first go ahead and code up what it would like in the HTML. This is an approach I often like to use in creating a directive, because it lets me understand the interface that I want, and from there I can build the directive to match. So going back into the droidInfoCard, between that and the personInfoCard, I'm looking at all this duplicated HTML, and this is what I want to remove. So I'm going to grab all of that HTML, and I'm going to cut it out my droidInfoCard, and I'll create a new file, and I'm going to call this userPanel. We'll build a userPanel directive. Let's paste that HTML in, and then let's just close it up. Going back to my droidInfoCard, what I'd like to be able to do here is specify that directive, and that removes most of the duplicated HTML. We've still got this button, our nextState button. Going back to our userPanel partial, we know that this div right here is where we want the content to go. In other words, that's where we want this HTML to be placed. So, all we need to do is add the ng-transclude directive, and now that tells this template that this is the element that we want to put our content into. Now, of course, we need to go into our script and create that directive, and I'm going to create it above the droidInfoCard directive. We know a few things, that's going to be an element, we're going to be transcluding, and we know it has a template. Now if we run the code as is, things at least are displaying correctly, but they're kind of displaying correctly for the wrong reasons. We haven't specified a scope at all in this userPanel directive. The moment that we do, some things are going to break. Going back into the partial, you can see right here we're using the droid's level to display the state, and over here we're using the droid's name. And of course, we want this to be a reusable directive that will work for either kind of user, either a droid or a humanoid or a person, so we don't want to have to specify droid right there. So we want to change these to be more generic, so let's change this to just be name, and this to just be level, and now if we run our code, things are going to be broken. The droid's name and level doesn't show up anymore. So we go back into our script and we need to specify a scope. And this is where we tell it where we're getting our name and level information from. So we'll specify name, and I'll use the at sign to indicate that this is a one-way binding. I'm putting in the name, and I'm not expecting it to change. In the future this could give us problems if we decide to make name editable, but for now we're not, so let's just leave it as is. We also want the level, and we know that the level changes, so we're going to use an equal sign to indicate a two-way binding. And lastly, I also want to be able to indicate whether or not what starts out in a collapsed state. And remember down in the userPanel, the functionality on collapsing and hiding is all inside of this HTML, so I'm going to set my initialCollapsed, and I'll use the at sign to indicate it's one way, and the attribute itself will be named collapsed. And now that I've got that specified, I can go back in my droidInfoCard, and I can give it that data. So we set the name and that's going to be our droid.name. And notice that I'm using a binding here. The reason I'm using a binding is because name is a one-way binding. Remember we specified that with the at sign. So I've got to put the actual text of the droid's name here in the attribute. We also want to set the collapsed state. Again, we're going to use another binding. And then finally we're going to set our level. Now since level is a two-way binding, we're not going to use the curly braces, we're just going to specify the name of the variable that we're binding to. So there are some differences here that it's worth making note of as to how to set data on a directive. And now that that's set, we can go down to our script, and we can add a controller here. And I'm going to take all the duplicated functionality outside of my droidInfoCard, and move it up into my userPanel directive. Now at this point, our droid has got the new directive and the collapse functionality will be working, including the initial collapsed; I won't demonstrate that right now. So let's go ahead and make these same changes to the personInfoCard directive. So again I'll start by taking out the duplicated HTML, specify the user-panel, specify the name, this time it's person.name, the collapsed is initialCollapsed, and finally the level. And now if we simply run this, again we'll see that the collapsed initialCollapsed is working correctly. But what's not working correctly right now is our Next State. Now we can go into our script and remove that duplicated code from our personInfoCard. So we can get rid of collapsed and nextState, and get rid of the collapse function. At this point, we're left with one piece that's not working. And that is the Next State button. This is now broken. There's an interesting problem with fixing this, and that is that the button for the Next State actually belongs inside of the HTML that is transcluded. So that means that this button cannot call a method that's on the directive that has the transclusion, because the transclusion is basically through prototypal inheritance transported up the chain so that it lives on the directive that wraps around the transcluded directive. We can see that inside of our template. This button here is our Next State button. Its ng-click method, nextState, has got to live on the personInfoCard directive itself, it can't live on the userPanel directive because the userPanel directive is technically inside of the personInfoCard directive, even though our transcluded elements are inside of that. Again, because of transclusion, this HTML is transported up the chain so that its bindings only work on the personInfoCard directive and not on the userPanel directive. So what we need to do is actually move our nextState button to be inside some HTML that's actually controlled by the userPanel directive so that the userPanel directives nextState function, right here, will actually get called by our ng-click when we call the nextState function. If we didn't want to move our button, we'd simply just have to duplicate this nextState function inside of our two directives, our droidInfoCard and our personInfoCard directive. But we're going to show how we can use that by moving our button. So I'm going to go into our userPanel, and right up here inside of the head, we're going to add the button right here. I need to set the ng-click, and I also want this button to sit far to the right, so I'm going to just add an inline style and float it over to the right. At this point, when I run the code, we've now got this Next State button up in the header, and if I click it, we've got a problem that's actually passing the click event onto the entire div, so the event is bubbling up. In addition to that, it's not actually advancing the state. So we have two problems we need to fix. Let's go back over to our script file, and inside of this directive in order to stop the event from bubbling up to the containing div, I need to bring in the event parameter and call stopPropagation on it. While I'm at it, it's also customary to call preventDefault. Now I have one more problem and you can see that right here in the code I'm calling scope.droid.level++. Since we brought in the level as a two-way binding on a variable just named level, I can actually take out the word droid here, here, and here. And then we'll go back into our userPanel partial, and in the nextState, I'm going to pass in the special variable $event. That is the event object for this click event, which I've mapped to the variable named evt. And of course since we moved that button, we can also go into our two info card directives and take the button out of the HTML. And at this point if we run our code, the functionality is correct. If I click Next State it advances the state on the user. If I click in the header, it collapses the header. So let's review the changes that we've made. We've moved a lot of code outside of our directives, the droidInfoCard and the personInfoCard, up into our userPanel directive. This collapsed function, the nextState function, and the collapsed variable have all been moved into a common place. In addition to that, we removed a lot of duplicated HTML. Inside of our personInfoCard and droidInfoCard directive, we're no longer duplicating a bunch of HTML, instead all of that common HTML is now inside of our userPanel directive. So this is a slightly advanced usage of transclusion. We've gone inside of two different directives and added a transcluded directive that is removing duplicated HTML, but it lets you see how powerful transclusion could be and how useful it can be to remove duplicated HTML and duplicated functionality.

Summary

In this module, we saw how to create directives that use transclusion. We looked at how scopes and transcluded directives interact, and how we should always create an inherited or isolated scope for transcluded directives. We also saw how we can use transclusion to allow us to customize sections of HTML. And we saw how we can reuse common HTML with transclusion in a way that would be difficult otherwise. Transclusion is a powerful feature of directives in Angular, and it can be confusing and a bit difficult to talk about. This module has covered most of transclusion, but not all of it. In the next module, we will look at how to build structural directives, but we will also look at the final piece of transclusion, the transclude function, which gives us the maximum amount of flexibility and control when using transclusion.

Structural Directives

Introduction

In this module, we will be learning about Structural Directives. Structural directives are our third and final type of directive. They are also the most complex and most rarely created type of directive. These directives have also been referred to as template directives. For the purposes of this course, I will mostly refer to them as structure directives, but either name is appropriate. A structural directive modifies the overall HTML structure. Let's look at some examples of built-in structure directives. The first example of a structure directive is the ng-if directive. You can see this hr has an ng-if directive on it. ng-if is very similar to ng-show and ng-hide in that it displays or doesn't display an element. But where ng-show and hide use the display none css rule, ng-if actually removes the entire element from the DOM. So for example, this h4's ng-if uses the same variable as the ng-model on this input box. So if I check the input box, the h4 shows up. If we actually look at the DOM, here's our h4, but if I come up and uncheck that checkbox, you'll see the h4 has completely disappeared, it's no longer in the DOM at all. Another example of a structural directive is the canonical ng-repeat. ng-repeat takes a DOM element and repeats it once for each item in a collection. So if we look at our HTML, you can see that this li exists three times. Another good example of a structural directive is the ng-switch directive. ng-switch is a little bit like the ng-if directive, because it actually removes the node from the DOM. You can see here there are two divs, but based on which value I select in my radio button, only one of those divs is going to show. And again if we look in the HTML, here's the div that showing right now, but when I switch the radio button, an entire different div shows up and that first div is removed from the DOM. Now one thing you may have noticed is that each of these directives uses transclusion, but because of the way they control where and when the transcluded node appears in the page, the transclusion that we have learned about in the last module isn't sufficient to create this functionality. That doesn't mean that the Angular team has cheated and created these directives using something that isn't available to us, instead there's a vast feature of transclusion that we didn't cover in the last module that we will be covering in this module, and that is the transclude function. This function allows us to decide when and where a transcluded piece of HTML will be included in the DOM. That is how structural directives do their magic. Indeed in this module we will actually create a basic version of the ng-repeat from scratch and a special purpose version of the ng-if directive. This module will really do some advanced work with Angular, and we'll see some of the most complex algorithms we have seen yet. So let's dig right into it.

The Transclude Function

In this section, we're going to cover how the transclude function works. This is an integral part of how structural directives work so it's important that we understand it. Let's go up to our index file, and in here I'm going to create a very simple transclusion directive. I'm going to create a directive that has essentially no display of its own, simply transcludes its contents. Inside of it, for demonstration purposes, I'll put something a little complex, so let's put an ng-repeat, and let's go down to our script and let's create that data. We'll just do a simple array of integers and that will be fine. Now again, I want this directive to simply put this inner content into the page and not have any display of its own. Using what we learned in the last module, that would be extremely simple. We would create a basic directive, which would be an element, we'd set transclude to true, and then we'd give it a very basic template with the transclude directive on the outer element, and that's all we need to do. If we go ahead and run this, we will see that it's working. Now just inserting a custom element into the HTML won't block the rendering of the interior content, so we have to actually inspect the HTML, and we'll see right here is our transclude directive, but there's our template that's actually getting rendered. So we know that this is working. If we go ahead and rename this to something that doesn't match, like put a 2 at the end, we still get the same looking HTML, but when we inspect it, we'll see that we just have our my-transclude element, there is no template that's being rendered into the content. So now what I want to do is recreate this, but I'm going to use a different method. I'm not going to set transclude to true and I'm not going to use a template. The reason being, I want more control over how this is getting transcluded. We're not going to take advantage of that in this demo, but in the future sections we'll see why that's important. So I'm going to set transclude to its other magic value, which is the string element, and I'm going to take template off, and I'm just going to leave this as is. Now we if run our code, you'll see something interesting has happened. It's no longer rendering that ng-repeat. Let's inspect the HTML, and you can see that our custom element, that my-transclude element, has been replaced with a comment. This is very important, this is what happens when you set transclude to element. It actually pulls your transcluded element out of the DOM and replaces it with a comment, and that comment is very important. Let's see why. I'm going to add a link function. Again, remember the parameters on the link function are scope, element, attribute. Let's close up our preview. These are the three parameters that we're familiar with. Now there's actually two more parameters on the link function that we can bring in. A controller function, which we won't use until a future module in this course, and finally, a transclude function. We're not going to do anything with that transclude function right now, but what I am going to do is logout the element that the link function is giving us. Since this is a jQuery wrapped element, I'm going to access the 0 item within that element to get the actual DOM object. Now when we run our code, if we go to the console, you can see that the element that the link function gives us is that comment inside the DOM. So now let's go back to our code and let's remove this console log, and instead we're going to take advantage of this transclude function and see how it works. This transclude function is often called the linker function, which I'm not a big fan of. I really think maybe a binding function would be a better name for it. What the transclude function actually does is give you an opportunity to take a scope and a template of HTML that has bindings and other directives in it, and bind the two together so that the data in the scope is now being put into the template HTML. At that point, you can then insert that HTML into the DOM wherever you want. Let's see how that works. We call transclude and we pass into it the scope that we want to bind to. So in this case, it's the scope that comes in on the link function, so immediately we see that some of the power that we've got is we can pick any scope that we want and bind to it. The second parameter the transclude function receives is a function. This function takes in one parameter, a clone of the DOM node that our directive is on. There's actually a second parameter, which is the same scope that we passed in in the first part of our transclude function right here. This is mostly a convenience in case you want this function to not have to close over that scope object. But most of the time people just write this function in line, so you won't need that second parameter. Inside of this function, this is where we can manipulate the DOM and actually tell it where we want to insert the transcluded HTML. In our case, that would be the ng-repeat div. To do that, we'll actually use jQuery functionality. So we'll take that element, which is the comment element, and I'm going to tell it that I want it to insert this cloned element after that comment. Once I do that, if I run my code, we can see that our directive is back to working exactly as it was when we were just setting transclude to true and had no link function and had a basic template. If we go and inspect the HTML, we'll see some interesting things. First, of course, there's our comment that was the element inside of our link function. After that is our my-transclude node. Again, this is a clone of what we created. This isn't the actual node that we wrote into the index.html. This is a clone of it that's been bound to that particular scope. I'm going to save the work that we've done so far, and-- no, let's make one more change. It's very common for structural directives to actually be implemented as an attribute and not as an element, so let's make the change here and come up and change this from a my-transclude element into a div. And that really doesn't change anything, but it's a lot more common for these sorts of structural directives to actually be attributes and not elements. Now that we've done that, if you go inspect the HTML, you'll see that it's still working in the way that it was as an element. So knowing how to do this seems at this point probably to be a little academic. Obviously we see some value as to where we can put this information into the DOM, even though inserting it into odd places is probably not a very useful thing, but we'll see in future sections how to take advantage of not only where to put in this transcluded content, but also when to put in the transcluded content. And we'll also see the power of controlling how the content is rendered.

Demo: Lazy Load Directive

Let's imagine the scenario where we have an application, which has several forms that appear on pages that are hidden until the user clicks a button to show them. And often times users don't click the button, so we don't want to pay the performance penalty of rendering out those forms and doing all the binding and linking, unless the user clicks a button to actually view that piece of our page. So what we'd like to do is have some kind of a lazy render control that won't render that portion of the page unless some action is taken, and also keeps Angular from doing all the bindings. Now this may be similar to the ng-if directive if you're familiar with that, but it's subtly different in that once we render it, we don't want to pull it out of the page anymore. This component is actually an idea of Dave Smith, who talked about this in his talk on directives at ng-conf 2014. Let's start by considering how that directive might look in HTML. We'll just use a div, and we'll add the directive my-lazy-render, and we'll set a variable that allows us to control whether or not it's been shown initially. This, of course, would be false to start with, and then set to true later on. Within here let's do another repeater, and we'll just do something like we did in the last section. And then afterwards we'll create a button and we'll give it a click event that sets the showit variable equal to true. Now let's go down to our script and let's put that data in there, and now let's create our directive. We'll restrict this to an attribute. As before we'll tell it to transclude the element. I'm also going to put a new attribute on our directive, and that is the priority attribute. I'm going to set the priority here to 1200, and that's a carefully chosen value, and we'll cover what it is and why we've set it specifically to 1200 in just a minute. Next, I'll create the link function with the same five parameters. Now I'll start off by tracking whether or not we've actually rendered this element. Then I'll set a watcher on the value that I've put into our directive. In this case, the value is showit, so I'll set scope.$watch, and that will give me that value, and then I'll check the value and make sure that it's true. And I'll make sure that we've never rendered this before. Now we can set the hasBeenShown to true, and we can call our transclude function. We'll of course bind to the scope that we're given, and we'll take our element and insert the clone after it, just like we did before. Now there's one other piece that I want to do. Now that we've rendered this once, we no longer need to watch this value, which in our case is the variable showit. So the watch function itself returns another function, which is a function that will turn off this watcher. Now that I have a handle to it, I can actually call that, which tells Angular to turn off this watch right here. So we're no longer paying a performance penalty by having an unnecessary watch running. Now let's go down and fix this right here. Okay, our code looks good, so let's go ahead and run it. And we'll see that there's no data that's visible. So let's inspect the DOM elements first. We can see that all that we've got is this one comment for our directive. Now let's click the Render Content button, and it's inserted this div. If we open this up we'll see inside of it our repeat, and of course, the data has shown up. So that's given us the ability to lazily bind areas of DOM only when we need them. Now let's go back and discuss the priority attribute that we added. Let's assume for a moment that we have another directive, and I'm just going to call this directive echo, because all it's going to do is echo out to the console. This will be a very plain directive. I'm going to give it a lower priority, and in its link function I'm just going to logout to the console. Now let's go into our HTML, and I'm actually going to add this directive onto the same element that has my lazy render directive. Now the point of our lazy render directive is to avoid paying the penalties of binding until we actually click a button that shows this element. So therefore whatever this echo directive does, I don't want it to run until the user clicks the button. And we'll see that if we run our code and open up the console, you can see that no message echo has been printed out. Once I click Render Content, echo gets printed out. So again, this directive hasn't been run, its link function hasn't been called until our directive allows this element to render. But if we were to go back into our code and change the priority to something higher, like 1300, let's open up the console and clear it out, and then let's run our code, and you can see that the echo directive is running even before we click Render Content, and that's not what we want. So let's go back into here and let's discuss why we want this at 1200. Most priorities tend to be pretty low on most directives, but I happen to know that in the current version of Angular-- and again, this is subject to change-- the priority of the ng-repeat directive is 1000. So let's change our scenario just a little bit. Let's set this to 900, so that it's lower than ng-repeat, let's go into our index, and let's actually set the ng-repeat and the my-lazy-render on the same element by doing this, and getting rid of this div. Now if we run our code, we can still see that nothing is showing up, but if we actually look at the HTML, you can see that we've got our lazy render showing up several times, one time for each data element. That's because ng-repeat is run first and duplicated that element before our lazy render ran and transcluded it away, and that's not what we want. We want our lazy render to run first and then ng-repeat to run second. Let's go back in and change the priority to 1200 again. And now if we look at the HTML, you can see that our lazy render is there and nothing else. If we click Render Content, all of a sudden the ng-repeat shows up. You can see a little bit of how ng-repeat works, because you're seeing the same comments just like we have for our directive. So therefore, on a directive like this, it's very important that we pick an appropriate priority to make sure that we don't conflict with other directives that we're using. In fact, if we go over here and set this to 1000 to be exactly the same as the ng-repeat, we'll actually break our directive. Things no longer work. It works fine when it's lower, and it works fine when it's higher, but it doesn't work fine if it's the same. Of course, we don't want it to be lower because that way ng-repeat will run first and do its bindings before the lazy render gets a chance to run. So this has all been very interesting. We've learned about how priority works, and we created ourselves a little lazy render directive. This shows us the power of deciding when our elements are rendered, which is provided for us by the transclude property when it's set to the value element.

Demo: Recreating ngRepeat

In this section, we're going to recreate ng-repeat. We're not going to do this because this is something you would frequently do in a production environment, but instead to give us some insight into how ng-repeat works, and the kinds of things we can do with structural directives. This is going to be one of the more complex directives that we create in this course, so now is as good a time as any for a shot of caffeine to make sure that you're awake and paying attention. Now the directive that we're going to create is not going to fully replicate all of the functionality of ng-repeat. ng-repeat has a ton of features and it's really not feasible to try to replicate it all. We're just going to replicate the basic functionality of ng-repeat. So let's start off by writing an HTML that we're going to do. We'll use an ng-repeat to start off with, and then we'll switch it out with our own directive. So let's display a list of Bounty Hunters. I'm going to make this a container and I'll make this h1 a well, just so it looks a little bit better, and then we'll use a div and the ng-repeat, and with the h1 we'll just have a span that shows the hunter's name. And let's put in a little horizontal rule, and then we'll have a couple of buttons. One of these will let us add a new bounty hunter to the list, and the other one will let us remove the last bounty hunter from the list, just very basic functionality. All right, let's see how that looks. Okay, it looks okay. Let's put in our data. All right, now let's implement our add function. And in here we'll just add on one bounty hunter. And then our remove function simply shortens it up by one element. So now if we run it again, we see our list. If we Add, we see 4LOM, we can Remove him and others. Okay, so now it's working with ng-repeat, let's now create our own directive. And I'll call this myRepeat, and this will be restricted to an attribute, and we want to set the transclude, and of course, we'll set that to element, and finally, the link function. The first thing we'll need to do is take the expression given to our directive, which is going to be this string right here, "hunter in bountyHunters," and parse that out into the two parts, the name of the variable, and then the name of the collection. So to keep this simple, I won't use a regular expression. Instead I'll just split it out into its pieces by calling the split function and splitting on the space. So now I've got an array and I know that the string name of the variable, which I'll call itemString, is in the 0 width element, and the name of the collection is in the element at index 2. So now I've got the pieces of that expression split out into two different variables. Of course one of the things I want to do is watch the collection, so I'm going to call scope.$watchCollection. This is very similar to the watch function, except it watches an entire collection and notifies you when anything in the collection changes, whether an element gets removed or added or some value on it changes, it'll notify you. So I give it the name of the collection to watch, and then I get a function that receives the collection. Now we're going to go with a very simple algorithm here. In the ng-repeat itself, it actually watches and only makes adjustments based on the actual changes made. For example, if an element is removed from the collection, then the corresponding DOM node is removed and nothing else. If a new element was added, then it just adds one DOM element. We're just going to clean the whole thing out, and recreate the entire set of DOM elements, so it's going to be a lot less efficient, but it'll be much simpler code to write. So, let's start off by just inserting a DOM element for every element in our collection. So we'll create a for loop. Now if we remember back, we create a DOM element by calling the transclude function, we pass in a scope, so let's just use our parent scope, and we pass in a function that receives a clone of the DOM node that's getting transcluded. In our case, that's going to be this div right here. Let's go up and change the ng-repeat to my-repeat. And so that we can see that we're actually putting something in even before we have bindings, since span doesn't have anything, let's put in a little bit of static HTML. Here in our transclude function, we'll call el.before. Remember el is the comment that gets created when we transclude the entire element. And so before that, we'll insert the cloned node. And now here in our code, we'll see that we're creating the right number of nodes, but we're not getting any bindings working. So let's make our bindings work. What we need is to have a variable named hunter. Back up here in the HTML, you can see that we've got a variable named hunter that has a name property. That's going to be the current item in the array. So we need a hunter object on our scope. Well, I don't just want to take the parent scope and add scope.hunter equals the current item in the collection, that would be silly because then we're binding ourselves to the variable named hunter, so we need to be the variable name of whatever the users put in, which is our itemString. But in addition, we don't want to put it onto the parent scope, because then for the next item we'll just be overwriting it. So what we need to do is create a brand new scope. So let's create a new scope by calling the $new function on our parent scope, and that will create a new scope that inherits from the parent scope, and then we'll also set the variable on that child scope. And remove that line from right here. And then we change the scope, we're passing in a transclude to be our new child scope, and let's run our code. We can see that we're now getting the bindings correct. Unfortunately, what's not correct is if we ever make any changes to the collection, like the Add an element, you can see it's just inserting new DOM nodes and not cleaning up the old ones. So let's fix that. For each DOM node that we create, let's keep track of it and remove it at the very beginning of our watchCollection function. So I'll create a variable called elements, initialize it to an empty array, and down here after we insert the DOM node, let's create an object called item, we'll set that items element property equal to the cloned DOM node that we just stuck into the DOM, and then we'll push that item object onto the elements array. And then at the beginning of our watchCollection function, we'll just check to see if that elements array has any elements, and if so, we'll remove them from the DOM. And we could remove those easily by calling elements which is the jQuery call to remove the DOM node from the DOM. And after we've done that, we can just set the elements to an empty array. So again, this is a very simple algorithm. We're just removing all the elements and then recreating them all with the new values. So if our collection has five elements in there and we have five DOM nodes in the document and we add a sixth element, then we remove all five DOM nodes and insert six new DOM nodes. So let's run our code and see what we get. Bindings are still correct if we click Add, it's adding a new item. If we click Remove it removes that item, and that works too. So we still have one problem, and that is down here we are creating a new scope for every element in our collection every time the collection changes. That scope object holds data inside of it, which can't be garbage collected, because it's still got references to it. So what we need to do is actually destroy that scope object every time we remove the corresponding DOM node. So first we'll have to track the scope object. We'll use our item variable and create a scope property on it, and set it equal to this new child scope. And then up here when we remove the element, we can also call elements of it to be garbage collected. Now let's just double check that we didn't break anything. Add still works and so does Remove. Now at this point, we've recreated the basic functionality of the ng-repeat directive. And you can see that it didn't take very much code to do it, and you can see how powerful it is that we can call transclude when we want and how often we want.

Demo: Business-Specific ngRepeat

In this section, we're going to show how to create a business specific repeating element. We're going to start with our my-repeat directive that we created in the last section, and customize it a little bit for some business specific purposes. So let's assume that what we wanted to do was create some kind of a user list directive that we could put in anywhere we want that would still be able to transclude and show the data we wanted, but also would have some built-in display. So let's go rename our directive from myRepeat to userList, and we'll go up to the index and change this right here to user-list. We'll also have to change right here from myRepeat to userList. Now what I want to do is add some business specific display. Let's assume that our userList is supposed to put every user inside of a div that has a class of well. In order to make that happen, we can go down inside of our transclude function, and right here we can create a wrapper object, and we'll create a brand new element using angular.element, and create a div object with a class of well, and let's append our cloned object inside the wrapper object. And now instead of inserting our cloned object, we're going to insert our wrapper object, and therefore we'll need to track that DOM node instead of the cloned DOM node, because the wrapper is what we want to remove from the DOM whenever we clean up. And if we run this code, we'll see that we're getting everything wrapped inside of a div that has a class of well. Looking at the HTML, we can see that we've got a div with a class of well, and inside of that is our actual user-list div with our transcluded content inside that. Now let's take this one step further, and let's change this from a well into a panel with a header that has the name of the user and a body that has the transcluded content as to whatever we want. So, that would let us take the name out of here, and maybe display something else like age. And our display would end up being a panel with a header and a body, and the body would contain this div and its contents, but we don't have to specify all that HTML, we just get that HTML for free. So in order to do that, we'll have to go back down to our script, and instead of a div where the class is well, we're actually going to put in quite a bit more complex HTML. So this is going to have a class of panel, and panel-primary, and inside of that we'll have another div that's got a class of panel-heading, and inside of this heading, we're going to have a binding to the name of the user. Now we don't know what the variable is that the username is coming in on, so we can't just use hunter.name, because in our HTML, we might not be hunter and bounty hunters, it might be user in users, or droid in droids, or something like that. So we've got to actually use our variable itemString. So we just do a little bit of string appending here, .name, and the binding and the div for the heading, and then we can have the div for the panel-body, and then we have the ending div for the entire panel. Now unfortunately, just creating an element like this won't actually do our bindings anymore. We're going to have to use a service that we saw earlier on, and that is the compile service. So let's bring that in to our directive, go back down here and we will call $compile, and instead of calling this wrapper, we'll call it template, because that's just a compiled template with the bindings in it that they're not bound to any particular scope yet. Next we'll create our wrapper object by calling the template, which is a function, and passing in the scope that we want to bind the template to. And that will run this template that we created, binding it to that scope. Then we need to find the panel-body within this wrapper element, and then to that div we will append in the clone DOM node. And we can get rid of this line of code here, and the rest stays the same. Now I'd like to run our site, but we're also displaying an age, so let's go up and add an age to our data. And if we run our code, we'll see that we're getting the age binding, we're getting the names showing up inside of the panel header, and we were able to do this all without having to specify the panel, the panel header, or the panel body, it's just automatically taking our HTML that we created here and sticking that inside the panel-body, and everything else is done for us. So something like this is a very powerful technique. You want to be careful and judicious in doing things like this, because even though it's nice to have some of that HTML already specified, there are other ways to skin this cat, and we're also losing out a lot of the functionality that exists in the ng-repeat itself. But you do see some of the power that is available by creating business specific structural directives.

The Compile Function

In this section, we're going to talk about the compile function. The compile function is a strange function inside of a directive, and you'll find that for the most part, you won't need it. But we're going to cover what it is anyway and look at how to use it. Let's start with a very simple directive that's just for demo purposes. We'll call this directive simple-directive. Let's go down into our script and let's create that directive, return an object, and now we would do things like setup our controller and our restrict setting and our other settings. And then down here we would have our link function, but instead of the link function, instead we're going to create a compile function. And that uses the compile key and that's a function, and the compile function is somewhat similar to the link function, but instead of taking in a scope, it just takes in the element, the attributes that are on the node, and it does have a third parameter, which is the transclude function, but that's been deprecated since Angular 1.2 because it's now included on the link function. This is the reason why the compile function is almost never used anymore. In almost all cases, directives that did use the compile function did so only to get the transclude function. Now that it's been added to the link function, it's no longer necessary. But we're still going to look at how the compile function works. Here in the compile function we can do some work, and ultimately what we need to do is return our link function. So the compile function gives you the opportunity to do a little bit of work before the link function itself is run, but the business case for this is almost non-existent. Again, almost no directives currently use the compile function. One of the few exceptions to that is the ng-repeat directive. So on the off chance that you do need to do something before your link function runs, this is how you can do it using the compile function. But even though the compile function runs before the link function, it really runs at a time when there's not very much valuable work to do. For example, the bindings have already been processed, the scope hasn't actually been bound if we were doing transclusion, but again, since we used the transclude function to do that and we have access to the transclude function in our link function, there really isn't any value to working inside the compile function. But on the off chance that you do find a business case for needing to do things before the link function runs, the compile function is how you can do it. We will actually see later on in the section on nested controllers, a scenario where we do need to do some work before our link function runs, but we'll actually see a better solution for that than using the compile function.

Summary

In this module, we learned about structural directives. We started by learning about the transclude function, how it works, and how it lets us control when and where transcluded content appears in the DOM. We saw a practical use of this with the Lazy Load directive. We also learned how powerful the transclude function is by recreating the basic functionality of the ng-repeat directive. Finally, we created a business specific ng-repeat directive. Although these last two demos may have more limited usefulness in production scenarios, they showed us the power of structural directives and gave us a taste for what you can do with them. Structural directives are by far the most complex type of directives, but they can be very useful in certain scenarios, and knowing how to work with them can be very important.

Controllers and Nested Directives

Introduction

This module is all about controllers and directives and how nested directives work. Controllers are an important piece of directives. They turn directives into a sort of mini Angular environment, where we can essentially use everything that we can use outside of a directive inside of a directive. That lets us break our pages down into pieces with directives and they all have the features and functionality we're used to, but also are more compact and reusable. After we look at controllers and directives, we'll look at how to nest directives inside of each other and how they communicate. We'll start with the link function in its pre and post variations, then get into requiring controllers, which is a much better API to communicate between directives.

Specifying Controllers

In this section, we're going to talk about all of the different ways that you can specify a controller for a directive. So we're going to start by going back to our old friend, the userInfoCard directive. Here we're looking at that directive, and this controller is specified right here. Now this controller is specified in line, but because it's just a function, we can actually take it out of line. We can grab the entire controller here, remove it, replace it with a name of a function, and move that controller up to the top. And a little bit of formatting, and if we run our code, we'll see that things are still working. Now there's good and bad to this method. The good is it makes the directive itself smaller, but on the bad side, the controller for the directive, which is an important part, is now removed from the directive itself and put somewhere else. This would make it difficult to look at and understand the entire directive. In addition to that, this method has a vulnerability if you're minifying your code. Before you minify your code, you need to annotate it so that it will work after the minifier. But if you're using an automated minifier, it may not catch that this function is a controller, and therefore, it may not annotate it. So let's move our controller back in line. Another thing we may want to do is use this directive with the controllerAs syntax. If you're not familiar with the controllerAs syntax feature in Angular, it's the ability to put properties and methods on the controller itself and have that available inside of your template using a specific name. We can do that by specifying controllerAs and giving it a name. I'm going to use vm for view model. At this point, we can replace most of the uses of scope with this. Unfortunately, we can't remove all the uses of scope because the two values that come in on our isolated scope, user and initialCollapsed. Now at this point we need to go down into our template and add vm where necessary. So that's how we can use the controllerAs syntax with a directive. Now as of version 1.3, we can actually make this a little bit better. We can completely remove all of the uses of the scope variable inside of this controller. The way that we do that is with the bindToController setting. With this setting set to true, all values that come in on the scope are actually added to the controller and not to the scope itself. That way we can remove all uses of the scope inside of our controller. And we'll go down into our userInfoCard directive, and we'll add the vm prefix now to user. Let's go back up to our script, and since the scope was no longer being referenced at all, we can actually take it out of our controller as a parameter. Now if you do use the controllerAs syntax, it is important that you understand something, and that is that there's not some kind of magic going on inside of Angular to make this work. All that's happening is Angular is taking this scope, creating a variable on the scope, named whatever you specify the controllerAs setting, and allowing you to use that as the base of your binding. The bindToController setting simply makes the scope variables also be added to that new variable on the scope, rather than to the base of the scope itself. If we weren't using bindToController, we could essentially do the same thing ourselves by bringing in the scope, and then creating a variable on the scope called vm, and assigning this to it. That is essentially what Angular is doing for us. Now if we run our code, we're going to notice that our address is no longer showing up. That's because of the way that the address directive was created. If we look at the address directive, we'll see that the scope is set to true. That means that the address is relying on certain values to be on the root of the scope itself, but they're no longer there, we've put them on the vm variable. In order to fix this, we'll need to go into our address template and add a vm prefix to everywhere that we see a user. And now if we run our code, we can see that everything's working. So there are a few different ways to specify controllers on directives.

Pre-Link and Post-Link Functions

In this section, we're going to take a deeper look at the link function by looking at the Pre and PostLink functions. The demo I've got set up here has two custom directives, the emperor directive and the vader directive. Down here in our script, you can see that these directives are extremely simple. They simply have a scope property set to true, so they're inheriting scope, and they have an empty link function. Now in our scenario, we're assuming that we're integrating with some kind of a third party component that requires us to put our name in the HTML5 data container on the node for the directive. In addition to that, if the directive has a master, then that is also supposed to be stored in a data for the node. So let's start by implementing the name storage, open to this link function, and we just call data, which is a jQuery function, and we'll do the same thing down here. Now on this directive we also need to store the master, but of course, we don't want to duplicate that data by just hard coding in the word The Emperor. Now we could use some jQuery to look up the DOM and get the name from the parent node, but that would kind of painful as well, so let's just grab it off the scope. We'll go up here into The Emperor directive and add master to the scope. And down here in our vader directive, we'll grab the master and the scope and store it here. Now just to make sure everything's working right, let's just logout to the console, the name of the master. Now let's run our code and open up the console. And we've got a problem. We get the message that My master is undefined. That's actually a surprise. If we go back to our code, we're setting the master up here, and then grabbing the master down here, but yet it is undefined when we actually try to do this. The reason for that is that Angular actually has two link functions, a PreLink and a PreLink function, and whenever we use just a Link function, we're actually using the PostLink function. Now we can make a small change here and specify that this is our PostLink function explicitly by changing the link function to an object, which has a post key, and I'll make the same change up here, and we haven't made any functional change. Now let's quickly explain what's going on here. Here we have a diagram of a typical scenario, a Child directive and a Parent directive, each with a Link function. Again we now know that these are PostLink functions. And the direction of execution actually starts in the Child and goes up to the Parent. So when Angular actually executes these Link functions, first it executes the Child Link function, then it executes the Parent Link function. But if we were to go into our emperor directive and change this post to a pre, and then run our code, now we can see that things are working as expected. The reason for that is that the PreLink function runs before the PostLink function. Now let's complicate our scenario a little bit. Instead of just two levels, let's add a third. We'll go down into our script and add this new directive, and we'll make it just like the other directives. Now in order for Starkiller to get his correct master, we've actually got to set the master up here on the Vader directive, and now we have the same problem as before, the Vader directive's PostLink function is going to run after the Starkiller directive's PostLink function. So we can change this one to a pre, but you may be thinking to yourself, uh-oh, we're going to mess things up now, because the Vader PreLink function is going to run before The Emperor PreLink function. So let's make this a little bit more explicit. When we logout, rather than saying my master, we'll say Vader's master is. Now let's execute the code and look at the output. And we can see that we're actually getting the correct result. Vader's master is The Emperor and Starkiller's master is Vader. So let's look at a diagram to understand why this is. When a Parent and Child have Link functions, both PreLink and PostLink functions, the order of execution is different for PreLink than for PostLink. For PreLink it goes from Parent first down into Child, for PostLink it goes from Child first up into Parent. So this is the diagram of the direction of flow. We start with the Parent PreLink, then the Child PreLink, then the Child PostLink, and finally, the Parent PostLink, And with a third level, the order is exactly the same. Parent PreLink, Child, GrandChild, then we go into the PostLinks, the GrandChild, finally, the Child, and then the Parent. So there we've seen how to use the Pre and Post Link functions with Angular directives. Now our current solution has a few problems. The biggest one is we're really polluting the scope in order to transmit data around between controllers. And when we have two levels, there's also a little bit simpler solution, that we can just add a controller to each directive and add the master information for the parent to the controller, because controllers run before link functions run. But that only helps us when we have two levels. When we have three levels, things are a lot more difficult. So in the next section, we're going to look at a much better solution for communicating between directives like this.

Requiring a Controller From Another Directive

In this section, we're going to look at how to require in another directive's controller. Requiring in another directive's controller lets you get a handle to that controller from another directive. This can be very useful in scenarios where you need to communicate between directives. It is important to note that you can only grab the controller of a directive that lives on your same node or on a parent node. So let's redo our scenario, but this time we'll leverage being able to require controllers. The first thing I want to do is go onto the emperor directive and create a controller. On this controller, I'm going to set the name property on the controller itself, and next I'll go to my Link function and remove all this to just go back to the default PostLink function, and now I can remove this line of code. Now of course I see some duplications, so let's remove that by creating a variable here, and now we have much nicer code for our emperor directive. Now let's see how to leverage this in our vader directive. So the first thing I want to do is go in and add a new specification, which is require, and I'm going to give it a carrot and then emperor. This tells the vader directive that it should grab a hold of the controller on the emperor directive, which is on a parent node of the vader directive. And if we go back to our HTML, we will see that emperor is a parent of vader. Our next step is to simplify our Link function back to just regular function and not the PreLink function. And now that we've told Angular that we want to require in the emperor directive's controller, we need to get a handle to that inside of our Link function. And that's what this fourth parameter is for. This parameter is a variable pointing at the controller that you require in. Now if we go down to this line of code and get the name property off of the emperor controller, let's specify that here, and now I can remove this line of code, and since Starkiller also needs to know the name of its master, we've got to add a controller to vader, and we'll do the same thing to remove duplication. Now one thing that you may notice is that on the controller, we're not setting the name variable on the scope, we're actually adding the name variable on the controller itself. That's because when we get in and handle the controller, we're not getting the scope of the emperor controller, we're getting the actual emperor controller itself. So anything that we want to utilize has to be on the controller and not on the controller scope. And this is true whether you are using the controllerAs syntax or not. And finally we want to go down into starkiller and make the same fixes. And now if we run our code and look at the console output, we can see that we've got the correct information again. So doing this we can see that we significantly cleaned up the code inside of our directives, compared to what we were doing before when we were trying to communicate between them. And of course we're not just limited to simple property names, we can actually call methods on the parent directive's controllers if we need that as well. In our starkiller directive, we required in the vader controller and we prefixed it with a carrot. The require specification is a lot like the scope specification, where there are special characters which will determine where Angular will look to find that particular directive. For example, if we don't add the carrot, then we will only look for and find a directive that's on the same node. Let's run our code and see what happens in this scenario. So we can see that we're getting an error that the controller for the 'vader' directive required by the 'starkiller' directive can't be found. In this case, we can fix this by making the starkiller directive and the vader directive on the same node. And let's look at the output from that. We can see that it's working correctly again, because they're on the same node. Another thing we can do is specify that the directive itself may not be present. We can do that with a question mark. In this case, when Angular can't find the directive, it will simply give us a null. Here we get the message that starkiller doesn't have a master. Now we can mix the two and we can put both a question mark and a carrot, which means that it's going to look on both the current node and on all parent nodes, and if the directive isn't there, we're just going to get a null. And there's one other variation, which is two carrots at the same time, which means that we're not going to look on the current node, only on parent nodes. And we can do this with or without the question mark. So there we've seen how to require in controllers on directives on either the same node or on parent nodes, which facilitates between communicating between directives.

Requiring Multiple Controllers

In this section, we're going to look at one other variation on requiring controllers. Let's assume that in our demo the starkiller directive not only needs to communicate with its master, but also with its master's master. So let's go back to a simple specification. And now to require in not only the vader directive's controller, but also the emperor directive's controller, we can do that by adding an array to the require specification, and giving a comma separated list of require specifications. At this point, we've asked Angular for two different controllers. So it's no longer correct to say that this is going to be the vader controller, instead this is going to be an array of controllers, and they will be in the same order specified in my require statement. That means that the vader controller is going to be in the 0 index, and the emperor's controller is going to be in the first index. Now let's run our code and look at the output. And you can see that Starkiller can now communicate not only with its own master, but also with The Emperor. Now this is one of those scenarios where the need for this in production is extremely rare. And it's very common to never run into a situation where this is necessary. But if it ever becomes necessary, this is how you can make your directive require in multiple controllers.

Demo: Panel Directive

In this section, we're going to look at a more realistic demo of requiring in controllers from a directive. We're going to do a tab control directive, which is kind of the canonical example for requiring in controllers. Let's take a look at how we want our HTML. We're going to start off with a tabstrip control. I'm going to name it sw-tabstrip. And inside of this tabstrip, we'll have a set of panes, and the pane will have a title, and some content. And we'll make a couple more panes. And there is what we want our HTML to look like. So let's go down to our script and start creating these directives. We'll start with a tabstrip directive. We'll make it an element only. Of course it'll be transcluded, and we'll add in a scope so we'll have a unique scope on this directive, and finally give it a controller, and one last thing, a template. Let's create the template next. We'll start with a div, and inside of that we'll have a ul, which is class nav and Nav-tabs. This is Bootstrap's tab navigation UI. Inside of that a list item, and inside of the list item an anchor tag, and inside of that we'll bind the title. Now the anchor tag itself isn't going to have an h-ref, but instead we'll give it an ng-click, and we're going to select the panes. We'll just call select and pass in the pane. Now our list item will need to repeat for every pane that we've got, so let's add an ng-repeat, and then finally we'll have the tab content itself, and we'll have to add our ng-transclude directive to that. So let's go back to our script. And we know that we have a select method that we need to implement, which receives a pane, and this will loop through a list of panes and select the correct one. So since we know we're going to need a list of panes, let's go ahead and create that. And then we can loop through each pane, and if the current pane is not the pane we're trying to select, then we'll deselect it, and we can just select the current pane. So there's our select method. We've also got to have a way to get panes into the list, so we're also going to need an addPane method. I'm going to delay creating the addPane method until we first create the pane directive. This one will be like the other directive in that it will be an element, it will be transcluded, and it'll have an isolated scope, but we know we've got a title on that scope, and that's a simple text value, and a template, and let's go and create that template file, and this will be just a div with a class of tab-pane. We'll have to stick the ng-transclude directive on this, because our transcluded content for the pane will go inside of this. Now let's go back up to our script. And whenever a pane is created, we want to add it to that tab strip. So, a link function is a great way to do something every time something is created, and since we just learned about bringing in controllers from other directives, we're going to bring in our tabstrip controller. Our tabstrip controller is going to need an addPane method, which we'll call, and pass in our scope. So let's go ahead and require that in now, fix our comma there, and now we know we need to implement the addPane method on our tabstrip controller. And again, we're not going to put this on the scope, we're going to put this right on the controller itself. So we'll add the pane to our panes collection, and let's check and see if this is the only pane in the collection, and if so, let's select it. Now let's run our code and see how it's looking so far. All right, we're getting our tab strip up at the top, but we've got a couple of problems. One, we're seeing the content for everything all at the same time. We need to hide everything but the current tab. So let's do that first, and go down into pane, and we'll set an ng-show on the pane, and let's run the code, and that looks right. We also want to visually show what the current tab is, so we're going to do that by going into our tab strip, and on this li we're going to add an ng-class, and we want the active class to be on this li if the pane is selected. Running our code, and let's select a different tab, and they correctly select now. So there's our tab control working. It's somewhat of a simple example, but it does show a really good scenario of when we can use requiring in a controller from another directive, so that one directive can communicate with another directive and even call methods on it.

Summary

In this module, we learned about controllers and nested directives. We started out by looking at the different ways we can specify controllers, both pulling them out of the directive's definition object into a name function, and also using them with the controllerAs syntax. Then, we learned about the PreLink and PostLink function, their relative execution order, and how we can use them as a way to get information from one controller to a child controller. Finally, we learned about how to require controllers so that we can get access to another directive's controller so that we have an API to communicate from one controller to another. This gave us a much more manageable way to communicate between controllers and a much cleaner API.

Directives and Services

Introduction

The interplay between services and directives is very important to understand in Angular. So, in this module, we'll be talking about the relationship between directives and services. You can use services to hold business logic for directives, to allow directives to communicate, and sometimes there's functionality we think should be in a directive, but should actually belong in a service. We'll look at all these items and more in this module.

Extracting Services

In this section, we're going to talk about how to extract services out of directives. So far the directives that we've been dealing with have had fairly simple logic inside of them, but that's certainly not going to be the case in a real world scenario. A realistic directive will likely have some fairly complex logic in it. Some of this logic will be related to the UI and some of it will be related to business rules. In order to keep our code maintainable, we want to us the correct pieces of Angular for their correct purposes. For example, controllers should not be doing business logic. Their job should be to mediate between the view and your model. So let's look at an example of some more complex business logic in a directive, and then we'll look at how to make that more maintainable. So I'm going to start by closing off this left panel, and closing down my preview. And we're going to change this so that when we knight a user, the rank isn't set to just knight, but actually Jedi Knight. Now, of course, it doesn't make sense to be able to make any user a Jedi Knight. For example, making Han Solo a Jedi Knight really just doesn't make any sense. Those of you who are Star Wars fans will know that Han Solo can't become a Jedi Knight, because he's not a Jedi. So let's add some logic to make sure that only appropriate users can become Jedi Knights. We'll do this with a simple if statement, and we'll just check that the appropriate conditions are true. First off, our user has to have the force, also that user has to have a minimum number of years of training in order to become a Jedi Knight, or if the user is the chosen one, that's good enough, or if the person was trained by Yoda, then the minimum number of years is going to much smaller. In addition to that, their master will have to approve, and of course they will have to have passed the trials. Great. And there's our conditional statement, and we can move this code inside. And now our users need to have appropriate data, so we'll scroll up, and you'll see that I've already added the appropriate data for the Luke Skywalker user. Okay, and at this point, we can see that line 51 has just got way too much logic to be put inside of our controller. Obviously it's really long, and we could start by just formatting it so that's all visible on the screen, but really all of this logic about whether or not a person could become a Jedi Knight doesn't belong in the userInfoCard controller. We might want to do this same operation from someplace else, and then we certainly wouldn't want to duplicate the logic. So let's extract this out into an appropriate place. So I'm going to go up here, and I'm going to create a new service. And this service will be in charge of deciding whether or not a user can be promoted to the rank of Jedi Knight. Therefore, I'll call this service the jediPolicy service. It'll be an object that's got a method that we'll call advanceToKnight. And that receives a candidate, which will be the user. And here we can ask our question. So I'll come down here and extract my if statement, enter it in here, and swap over user to candidate, and while we're at it we'll do a little bit of formatting to try to make this a little more readable. (Typing) And at this point, we can set our candidate's rank to Jedi Knight. Now I'd like to give some kind of notification to the user when they can't advance a user to the rank of Jedi Knight, rather than just silently failing. In this case, I would normally do something like this. After my if statement, I'd put in an else, and maybe do an alert, but in this case I'm no longer asking that question down here in the knightMe function. Instead this code down here is going to be changed to this code here. In order for this to work, of course, we need to bring in our jediPolicy service, and now this code works if we run it. We'll see that we can't knight Han Solo, but we can knight Luke Skywalker, but again, we're not getting a failure notification. And of course I don't want to go up here into my service and add the else clause here, because this is display, and that's not what this service is concerned with. The service is concerned with the business logic. So thinking about how we should handle this in Angular, what we should do is return a promise, and that promise will let the controller know whether or not the operation succeeded or failed. So let's bring in the q service and let's create a promise, and I'll use the new promise interface that's available in Angular 1.3, and return the promise. And up here, if the candidate was able to be advanced to the rank of Jedi Knight, then I'll resolve the promise and pass in the candidate. Otherwise, I'll reject the promise. And now I can go down into my controller and call then on the return promise. And the then function takes in two parameters. The first one is the callback function if the operation succeeds, in which case I'm going to do nothing, so I'll just pass in a null. And then second is the callback function if the operation fails or in promise to \_\_\_\_\_ if it rejects. In this case, I'll do an alert, and now our code is complete, and we can run it, and see that Luke can be advanced to a Jedi Knight, but if we try to advance Han, we get our alert error message. So there's a good example of looking for business logic that creeps its way into our controllers, extracting that out into services, but leaving the display related logic inside of the directive's controller.

Using Services to Communicate Between Directives

In this section, we're going to be discussing how to communicate between directives. It's often the case where we have two directives on the same page that are independent of each other, but need to communicate in some way or another. We've seen in the past that if they are parents, or on the same node, that there's a way that we can get a hold of the other directive's controller. But if these directives are not in a parent child relationship or not put on the same node, we still need to be able to communicate. Often the communication method is what is shown here. We rely on the Containing Scope to communicate or mediate between the two directives. Many times this is done by just setting the directives to inherit scope from the parent, that way any data that's already on the parent scope that they change is visible to the other directive. This can also be done with isolated scopes by explicitly specifying what pieces of data on the containing scope need to be visible to both directives. Let's look at an example of directives that do exactly this. We have two directives here, a master directive and a detail directive. These two directives are composed together to create a master detail form on a page. Now, of course, often with master detail forms, we can use a single directive, but the benefit of using two directives is that they're more composable. We can swap out the master directive with a different one, or the detail directive with a different one, and we can also place them anywhere on the page and let the page itself decide where they go. The implementation of these two directives is fairly straightforward. They both take in a list of the users and the currently selected user. The detail directive doesn't do much else, The master directive also has a controller which sets the initially selectedUser and it implements a selectUser function so we can set what the selectedUser is when somebody selects one in the master section. Using these directives is pretty straightforward. You can see I've got them arranged here in two columns. The left column has the master section and the right column has the detail section, and of course, I'm passing in both required pieces of data. The template for the master is fairly straightforward. We have an unordered list that's unstyled, and then for each user we create a list item that has a button in it. The click event calls selectUser. And then based on if the current user is selected or not, we either set the button success class, or we set the button primary class. Our details template is also straightforward. It's simply a panel that repeats for each user. We only show the selectedUser by using the ng-show directive, and then of course, we bind the data in appropriate places. Let's take a quick look at how this directive looks. Here we've got our two directives. If I select a directive, it's selected over on the right and on the left hand side as well. Pretty straightforward. Now let's go back to our code and discuss some of the benefits and drawbacks of this implementation. First off, this implementation is fairly simple and easy to reason about. We have the selectedUser that kind of determines everything that happens. We do have one very interesting piece that's going on here. In our masterUsers directive, we have to implement a selectUser function. Now obviously a master detail form is going to have a way to select a user, so putting the logic about selecting users inside of the masterUsers directive isn't too big of a deal, but imagine that that algorithm became fairly complex and needed a whole bunch of checks, and we wanted to have several different masterUsers directives that we could put in just the one that we needed in a particular page. In that case, you might be duplicating the logic across those directives. Also, again, even though that directive has to do with selecting a user, but the actual object about selecting a user could potentially belong to somebody else, and this directive could simply be in charge of coordinating user events and handing them off to somebody else. The second drawback is that we have to put so many pieces of data onto our isolated scope. Right now it's only two, but you can imagine in a more complex real world scenario, this could be four or five or six pieces of data. In most cases like this, what we're usually trying to do is communicate the state of the page between the different players on the page-- in this case, these two directives. So what if we changed our implementation so that instead of communicating with each other through the parent scope, they actually communicated through a service. Let's see what that might look like. I'm going to start by creating a new service. Now since this service is going to hold the state of our application, I'm going to call this the userListState service. And of course, how much of the state of our application this service is responsible for is going to determine its name. One of the benefits of this implementation is that we can use this state service on more than one page, and that way multiple pages can share the same state. Of course that's always the power of componentization. We get the composability and flexibility that allows us to create complex applications out of simple pieces. This service is going to be extremely simple, it'll just be an object which has one property, selectedUser which we'll default to null, and of course, in a more complex scenario this could have a lot more pieces of data, but for now this is all we'll put on it. Now that that service has been created, we'll go down and add it to our first directive, which allows us to take this piece of data here off of our isolated scope, then we'll go down and set the service on the scope so that we can bind to it. And we're still going to set the initial selectedUser, which we'll do by using this service, and now we can get rid of this selectUser function. Now let's go put our master template and make the changes necessary there. So we've got a bit simpler implementation, which means we no longer need to call this selectUser function, instead we can just set the state.selectedUser equal to the current user, and in our class specification, we'll just check that the current user equals state.selectedUser, and the same thing for primary, and now our master template is ready. So let's go and update our detail directive. Again we can get rid of this piece of data, but we're going to have to add a controller, and bring in the userListState, and set that onto the scope, and then we go into the detail template, and we can change our ng-show to be state.selectedUser. And finally we can go into our index page, and we no longer need to set the selectedUser property on each directive, since we now get that through a service. And now we can run our code and see that everything is still working just fine. So let's go back to our code and discuss the net effect of our changes. In this particular example, we've made some fairly minor changes in order to move this state from something that is communicated through the controlling scope from one directive to another into a service that allows the two directives to communicate with each other. The net effect of our changes has been fairly small. We've removed one of the pieces of data on the isolated scope and simply moved it into a service. At first glance, this seems like an innocuous change that doesn't really have much benefit, but the truth is, the controlling scope no longer has to know about that piece of data. That makes the API for these directives simpler, which we can see by looking at our index; instead of having to set two properties, we now have to set one, the second of which seemed very ceremonial. When we put it back, we can see that both of these have to have the same exact thing set, and the controlling scope needs to know about it and set it, but other than that, the controlling scope has nothing to do with the selectedUser property. Also, we've now given ourselves a place where any logic about the selectedUser can go. That means if selecting a user becomes a complex operation that requires checks, we can simply add it to this userListState service, and any directives that use that service will now benefit from that additional logic. And there we've seen how to get directives to communicate with each other and share state through a service.

Directives That Should Be Services

It is a well known rule in Angular that anytime you need to manipulate the DOM, you should do that in a directive. Therefore, we often make the assumption that anytime we're dealing with a view, we should be creating a directive. But there are scenarios where certain pieces of functionality, which we would normally think of implementing with a directive, should actually be implemented with a service. These are the kinds of view functionality that are more Imperative and not so Declarative. We don't want them to react, we want them to do something. In order to understand that, we're going to have to see the difference between user a directive and using a service that do the same thing. So in the next few sections, we're going to build a modal directive. This will serve two purposes for us. First, it will let us see an example of a more imperative need that's implemented in a declarative fashion, and it will also give us the opportunity to build yet another complex directive. In fact, this will probably be the most complex directive we'll implement. After we build this modal, we're going to implement it in a more realistic scenario, and then we'll do the same thing using a modal service, and then we'll compare the two implementations and see the differences and why certain things like modals, dialogs, et cetera, should be implemented as services and not as directives. Building this modal directive may seem like an exercise in futility, since I'm recommending not to build modal directives, but it's actually going to be really interesting to build one. We're going to put a lot of different pieces together and there's going to be a lot to learn. So, if you're not interested in learning to do that, you can go ahead and skip forward to the section on using a modal directive. But if you are interested in seeing yet another example of a complex directive, then follow along while we build this modal directive.

Creating a Modal Directive

In this section, we're going to start building our modal directive. As always, when building a directive, I like to start with the HTML. So let's start by changing our title, and let's create a simple anchor tag that we will click when we want the modal to open, and next we'll come to our directive itself, I'll simply call it modal, and since we need to react to a change in order to open our directive, we're going to need some kind of a variable that we can change, which will trigger the modal to open. Directives don't have an API, so from a controller I can't get a hold of a directive and say do this action in an imperative manner, it's got to react in a declarative manner. So therefore I've got to bind it to some kind of a variable. Obviously our modal needs to know about this variable, so we'll create an attribute called open. And I'll name that variable modalOpen. A good modal will also let you set some options such as size and position, so let's create an options attribute, and since we're talking about several different options, we'll just use an object. And let's say in our case I want to set the top to 35 pixels and the height to 300 pixels, and override the defaults. Another thing a good modal should do is let you know when it's closed. So let's add an on-close attribute, and from here we can call a method on our parent controller to notify it that the modal was closed. So just call that method modalClosed, and we want to be able to pass back some data, so we'll specify some data. And then inside of the modal will be a transcluded HTML, which is the template that we want to render. So let's do a very simple panel, and in the heading of the panel we'll put some simple text, and in the body let's put a couple of buttons. And yes I do realize that these are div tags and not button tags, but because of the beauty of Bootstrap and Angular, they're going to look and function completely identical to buttons. So I'll just leave those as divs. When we click yes, we want to call some kind of a method that closes the dialog with a positive response, and when we click no we want to do the same thing with a different response. And we'd probably like to be able to have our own controller inside of the dialog, in case this dialog ever gets fairly complex. So let's add a controller up here. And that makes a fairly versatile modal directive. Now we'll go down to our script and start building this directive. Now we're going to want an isolated scope on our directive, but it's got to take in a few values. First off the variable that will determine whether or not the modal is open or closed, and we call that attribute open. Also our options, we'll just leave them with the same name as the attribute, and then our onClose event. And we know we want this directive to support transclusion. We're also going to need to specify a template to have the HTML that a modal needs. And finally, our controller and a link function. Next let's create the template. This will be an outer div that will give a class of modal-blackout, and that will be the div that grays out the rest of the forms so that you know that you can't use it. And this is where we'll put our ng-show directive, which will bind to that modalOpen variable. Inside of that, we'll have a modal-container, which will be the actual dialog. Now we would like a close button to show up by default, so let's put in a little anchor tag, and an ng-click on it that will close the dialog, and then finally our transcluded content. In this case, we'll use the ng-transclude element. Now to be typical, we often set it so that if users click anywhere on the blackout area, it also closes the dialog. We're not going to implement that for now, but it's a fairly simple manner to implement if we wanted to. Next we'll implement our css. I'm not going to explain most of the css since it's really not the point of this, but we'll just set our blackout to cover the entire area. You may notice that we haven't specified a width or a height. We'll actually do that in code in just a minute. Next, we'll set our modal-container to sit inside of our modal-blackout div. Again, we haven't set the position or size. We'll do that using code so that the options given can override the defaults. And last we'll position our little close button. And now we'll go back into our code and we'll set the size of our blackout div by getting the height and width of the document. So first we'll bring in the document. We're not going to use the document global, we're going to use the document service that helps keep our directives testable. And since our containing element is the actual modal element, we'll take that element and find our blackout element, and specify the height and width properties. Now of course we need to set some defaults for our dialog's position and width, so let's start by just creating a simple object that will be the defaults. And then we'll take the options given by the user and combine them with our default options to produce the actual options that will be used when creating the dialog. And now that we have those options, we can go ahead and set the size and position of our dialog box. Again, we'll use the modal element, and find the modal-container. And then we'll set the position and size. Now we've got most of our directive created, let's go back up into our containing controller and let's create a function that can be called when the user wants to open up the dialog. And that of course is the function that's called by the anchor tag that we created. Now let's go ahead and run our code as we've got it so far and see what we're dealing with. We have the basic elements in position, but we can't close the dialog, and this background probably shouldn't be semi-transparent. So let's go back to our css, and let's give it a background, and a border. We'll run it again and look, and that looks much better. Let's go back to our code. Let's slide this a little bit open. And let's respond to the click event on the X in the upper right hand corner. So we'll go to our controller on our directive, and we'll implement the close function. And inside of here we'll set the modalOpen variable to false, and that will hide the directive. And let's also call the onClose callback that we fed into our directive. Now it's worth noting that the modalOpen variable, because we're using the equal sign, is a two-way binding. If we were to use the at sign, we would simply receive the value of what modalOpen is. We're using an equal sign, so we've got this variable that we can actually change inside of our directive, and that variable's value will be updated outside of our directive. So our modalOpen variable inside of our directive will always be kept in sync with the modalOpen variable outside of our directive, even though these are two separate variables. Because of Angular's bindings, Angular will keep those in sync. Now let's test that functionality. And we are able to close our modal down. Now so far we've got a good start, but it isn't very useful because I can't click the yes or no buttons inside of that modal and have it do anything. So let's implement those. Going back to our index, we can see that those two buttons call a close method. So let's create this modal controller, and we'll implement that close method. That function receives a response, and of course inside of here we could simply just set the scope's modalOpen variable to closed, but we're going to have two problems with that. One is this controller inherits its scope from the main controller, which means that if we set modalOpen to false, we'll actually just be setting a brand new variable called modalOpen inside of the modal controller, and the modalOpen variable that the main controller created, which is used to control the visibility of our modal, will not get changed. We could add an extra dot in there, so instead of scope.modal, it could be maybe scope.modalOptions.modalOpen. But a better option is to create a new method on the parent scope called closeModal, which receives a response, and this will set the modalOpen variable to false, and then take whatever action is appropriate. In our case, we'll simply logout to the console the response. And now within our close function inside of our modal controller, since we inherit from the main controller, we can see the closeModal function on the main controller and simply call that. And now let's run our code and open up the console so that we can see the result. And we'll click yes and we'll see that modal is closed with Yes, if we click No, the modal is closed with No. What happens if we click our close button in the upper hand corner, nothing happens. So let's fix that last issue. Remember again, inside of our index we set the on-close function equal to modalClosed passing in a response. So let's implement that modalClosed function, which receives a response, and we can again call it our scope.closeModal, but this time we'll pass in the correct response, which would be no. And now if run our code and look at the console, if we open up our modal and click the close button up at the top, we can see that it's closed with the response that we want. So there we have the implementation for a very simple modal directive. In the next section, we'll look at implementing this modal directive in a slightly more realistic scenario.

Using a Modal Directive

In this section, we're going to take the modal directive that we created and implement it in a realistic scenario. So we're going to add it to our userInfoCard directive. I've got the userInfoCard directive in the state where the Knight Me function simply sets the rank to knight. This is where we're going to use our modal dialog. We're going to confirm that the user really should be set to a rank of knight. So let's start by bringing in our directive, and we'll just copy it out of here, and paste it into here, and we'll also need to bring over the styles. And of course we need that modal.html. And now we can go ahead and add the directive to our HTML. So you might think initially that we would maybe go inside the body and add in our modal here, but the problem with that is that when we run our code, we want to be able to click the Knight Me button and have a modal pop up for that user specifically. That means we want to get some data about that user into the modal dialog. I'd like the modal dialog to include the name of the user when confirming that they're being advanced to the rank of Jedi Knight. But if the modal is down here, then I can't put any bindings that will bind specifically to that one user, since user.name doesn't have any context out here. It only has context inside of the userInfoCard directive. So instead of putting our modal here, we're going to go into the userInfoCard directive and add our modal down here. We'll give it the open variable, which will be vm.showKnightModal, and for options, we'll set a width and the left. And for the on-close callback, we'll call this function knightModalDone, passing in the response. And inside of this, let's put in a panel with a simple header, and inside the body, we'll ask them if they're sure, and bind to the user's name, which would be vm.user.name. And finally, our yes and no buttons. And here, instead of calling a method on a controller inside of the modal, like for example, putting in a controller down here and calling a method on that, we'll simply call a method on the userInfoCard scope. So we'll name this the knightDialogDone method, and we'll pass in a true if they click yes and a false if they click no. And now we go up to our code and implement that method. And inside of here if the response was true, then we'll set the user's rank to knight. So that means we can remove that from here. Now the Knight Me function just turns on the modal dialog, and if you'll remember, the showKnightModal variable controls that. So we'll set this.showKnightModal to true, and of course, in the done function we want to set it to false. Now let's run our code and see how it looks. If we click No, then nothing happens. If we click Yes, the rank gets set, but I'm not really crazy about that visual. I don't like this extra border, and this extra close button up here does close it, but it kind of seems unnecessary. So there are a few visual changes that our modal dialog needs, but overall it works correctly. So there we have an example of an implementation of a modal dialog done as a directive. In the next section, we'll look at doing the same thing, but with a service.

Using a Modal Service

In this section, we're going to implement a modal using a service. Now we're not going to build our own modal service for two reasons. The first of which is building your own modal service is actually a very long and complex process, although if you do enjoy doing some very advanced Angular, I highly recommend you give it a try as an exercise. The second reason we're not going to implement our own modal service is that there are already several great third party modal services available. The most notable of which are the Angular UI modal service and the new Angular material modal service. In this section, we're going to be using the Angular UI modal service, but if you are interested in using a modal, I highly recommend you check out the new Angular Material modal service. Now the point of this course is not to learn about Angular UI, so we're not going to be discussing much about the Angular UI modal service, we're just going to implement it to meet our needs and not look at any other features of that modal service. Now the very first thing we'll need to do is go up into our index and add Angular UI. It needs to come after angular.js, so I've placed it on line 8. Now that we've got the Angular UI script added to our page, we go back into our script file, and down inside of our Knight Me function we will open up a modal dialog. We're going to be using the $modalservice that Angular UI gives us, so we'll have to bring that into our controller by adding it as a dependency. And now we can call the modal. We do this by calling modal.open and this returns a modal instance object, which we'll capture, and inside of this object, we give it our options. The first option we'll give it is our template, which is a templateUrl, and we'll make a template called knightConfirmation, and we can specify a controller if we want one, we do in this case. And we'll name this controller knightConfirmationCtrl. And then the last thing we need to do is pass into our modal any data that it needs to bind to. In this case, we're only binding to the user, and we do this through a resolve property, and we tell it we're going to be giving it a user piece of data, and then we give it a function that returns the actual data we want to be the user object, which in this case will be our current user object. Normally we would type in this.user, since we're using the controllerAs syntax, unfortunately, inside of this function, the this object is not the same thing as the controller. So instead of using this.user, we'll use that.user, and then up here in our controller, we'll set that equal to this to capture the this variable so that we can close over it inside of that resolve function. Now you may have noticed that we're also bringing in the user as a parameter to this Knight Me function, so we could also use that as well. Now that we've opened up the modal, we also need to capture the result that the modal gives us. So down here we're going to call modalInstance.result.then, which takes in a function, and we'll get our answer back here, and we'll check if the answer is true, and if so, we'll set the user's rank to knight. Again, we can't use this, because we're inside of a function that's not going to be executed with the same context, so we'll use that and we'll pull out this line of code here. Now one very important piece that's very easy to forget, is to add Angular UI as a sub-module to our module. In order to do that we go up where we declare the module, and we add in ui.bootstrap, since the modal service is inside the Bootstrap module of Angular UI. Now our next step is to implement our knightConfirmation template. So create a new file, we'll name it correctly, and this again will be a panel. We'll make a simple header, and ask our question inside the body, and since in our resolve we specified that the data coming in was user, then we don't have to prefix this with vm or anything else, it's just going to be user.name. And of course, our two buttons. The Yes button will call a method named yes, and the No button will call a method named no. And lastly we need to create our knightConfirmation controller. So right here we'll create that new controller. And this controller is going to receive three dependencies. Obviously we need to bring in our scope, but we can also bring in the modal instance. Down here where we create the modal instance, that same variable can be injected into our controller by simply bringing in $modalInstance. And lastly, in the resolve, we said that we wanted to bring in a variable called user, so in our controller, we'll simply have an injected variable called user. Now first we'll take that user variable and stick it on the scope, then we'll create our yes function, and in here we want to close the modal, so we'll call modalInstance.close, passing in any data that we want to go back to the caller. In this case, we'll pass in a true, and we'll do the same thing with scope.no, passing in false instead. Calling modalInstance.close will actually resolve this promise that we see down here, so the true or false we pass into the close method on the modalInstance will be the same data that's our answer parameter in the callback function on the result. And so now we should be pretty much done, so let's double check our code and make sure we didn't make any mistakes. And we can see that on line 37, I had a typo where I didn't put in the t in knight, so let's fix that, and now if we run our code, we click the Knight Me button, we do get our dialog, but we see we've got a little bit of a visual problem. That's because the panel has a margin bottom set to the like 20 pixels. So let's go into our css file and we'll add in a new rule. We'll create a panel-modal class and set the margin bottom to 0, and we'll go into our knightConfirmation and add that class to our panel. And now we'll run our code again, and the dialog looks much better. If we click No, nothing happens, and if we click Yes, then the user receives the rank of Jedi Knight. So let's go back to our script and now let's examine the implementation of these two modal dialogs. Here we can see the code for using a modal service. You can see that this code is very imperative in nature. We create a modalInstance, we give it some parameters that it needs, and then using a promise we receive the result of that modalInstance and can act on that result. What's great about this is that the pieces of code can exist together. If we go and look at our declarative modal directive, we can see that we open up the modal by setting a variable to true, we close the modal by setting it to false, and we have to handle the response in an entirely different location. So even though both of these methods work to accomplish our goals, the imperative method is definitely more in line with what's actually going on in the code. We're asking the modal to show itself, and we're asking it for its result. In the declarative method, we simply set a variable, and as a byproduct, the modal becomes visible, and then the modal calls back into a function to let us know that it's done, at which point we have to remember to turn the modal off. Now neither of these two methods is necessarily right or wrong, but there is a difference between the two, and the Angular team themselves highly recommends that any functionality you have that is imperative in nature should be implemented as a service and not as a directive. That certainly includes things like dialogs, both modal and non-modal. It's also true that in the case of dialogs and modals, there are many great third party solutions out there, so writing your own from scratch is usually not the best use of you time, and it's also interesting to note that the vast majority of those third party solutions are implemented as services and not as directives. If nothing else, that should tell you something right there.

Summary

In this module, we learned about the relationship between directives and services. We first saw how we can extract services from directives in order to keep business logic where it belongs, and to make our applications more composable. We also saw how to use services to allow directives to communicate with each other. And finally, we looked at the difference between services and directives with imperative view functionality like dialogs. Services give us the ability to create imperative APIs to functionality where directives do not. Overall, services and directives have a close relationship and both must be tools in your tool belt when authoring Angular applications.

Course Summary

I'd like to thank you for watching this course. I truly had a good time building it. Directives are one of the more complex pieces of Angular, and understanding their role and getting comfortable with building them so that they are a tool that you can easily reach for is an important part of progression on the path to competence in Angular. Angular lets us be expressive in our HTML, using custom elements to express the intent and the functionality of our application. This is the core of Angular. Angular itself is essentially just a set of directives, starting with the ng-app directive and the ng-controller. So learning to write your own directives and write them properly is a skill that will always benefit you. Far too many applications are built without using enough directives. Don't let your next application be one of those. Again, please say hi over Twitter or connect with me on LinkedIn and say hi, and most of all, thank you for giving me the opportunity to go on this journey with you.