Course Overview

Course Overview

Hey everybody, I'm Joe Eames and this is Jim Cooper, and welcome to our course, Angular Fundamentals. Jim and I have been working with Angular for a long time and we are super excited to present this course to you. Angular is an amazing and comprehensive framework, which lets you quickly and easily put together applications without having to worry about manually updating the user interface when your data changes. It provides everything you need to create compelling applications on the web. In this course, we are going to learn all the major features of Angular and how to get the most from it, even if you are just a beginner. The features we'll cover are controllers and templates, which display the interface, services which contain your business logic, directives for creating your custom HTML components, how to get data to and from your server, and how to test your application. By the time we're through, you'll know how to build production-ready applications in Angular. Before starting this course, you should be familiar with JavaScript and web development, but don't worry if you're not an expert, a basic knowledge will be enough. We hope you'll join us on this journey to learn Angular with the Angular Fundamentals course at Pluralsight.

Introduction to Angular

Course Introduction

Hi, this is Joe Eames, and this is Jim Cooper, and welcome to Pluralsight's course on AngularJS Fundamentals. Client-side JavaScript used to be simple enough that we could get away with very little thought as to the structure of our JavaScript, but as our web applications have become bigger and bigger, we need something to reign in all the resulting complexity. AngularJS allows us to toss out all that client-side spaghetti code and write simple and elegant MVC-style single-page applications. In module 1, I'll show you how simple it is to get your project bootstrapped with Angular seed and how to start working with Angular markup. Then, in module 2 I'll show you how to start organizing that code into controllers and how to use the built-in directives to control your view. Then I'll jump in, and in module 3 I'll show you how to abstract some of the complexity out of your controllers and into services to facilitate the single responsibility principle and make testing easier. After talking about services, I'll show you in module 4 how to use routing to turn your app into a true single-page app, and I'll demonstrate all the built-in power and functionality that comes with Angular routing. And finally, in module 5, I'll talk about one of the most exciting and powerful pieces of AngularJS, directives. I'll demonstrate how you can use AngularJS directives to create your own custom elements, observe changes, and handle events. And last, but certainly not least, I will jump back in and show you how the Angular team has designed AngularJS to allow you to do all this in a fully test-driven way. I'll demonstrate how to test your controllers, services, and directives, and even how to do full end to end UI testing.

Module Introduction

In this module, we will cover the following topics. First, what is Angular, what kind of library is it, what does it have in common with other libraries of its kind, and what separates it from the pack? We'll also discuss why you should consider Angular when building the frontend to your web applications. And we'll also take a glance at the architecture of Angular and talk about how it does some of the magic it does. Next, we will build the simplest possible Hello World application, just to dip our toe into the water. In that run-through we'll meet a few interesting pieces of Angular, including controller and scope, Angular markup, and the ng-app directive. Then we will discuss the application that we will be building throughout this entire course. That application is called EventReg, which is an Angular Event registration site. We will spend a few minutes talking about the features and requirements of this application, and how it will be used to teach Angular.

Course Updates

Before we get into the rest of this module, I want to take a minute and talk about the course update site. We are using a GitHub repository to both supply some files that are difficult to get elsewhere, and as a way to update viewers of the status of the course. Angular is one of those technologies that is rapidly changing, and we are committed to keeping this course up to date. It's pretty easy when an update to Angular is released for it to cause a clip or two of this course to get out of date and not work correctly with the latest version of Angular. Because of the complexity of publishing video courses, it can take several weeks for an update to go live, so in order to facilitate that, this GitHub repo will be used as a way to note if anything in the course is out of date, how to deal with that for now, and when the update to the course will be published. So take a minute right now and go out to this URL and see if the course is up to date.

Introduction to Angular

So what is Angular and why would you want to use it? The simplest answer to that question is that Angular is a JavaScript library, but it's so much more than that. It's probably more accurate to categorize it as an MV\* framework. If saying that doesn't tell us everything about it, at least we have some context since we can compare and contrast it to some other popular MV\* frameworks, such as Knockout or Backbone. Although we won't really make any direct comparisons to other MV\* frameworks, we will discuss things it has in common with other frameworks, and the things that set it apart from the crowd. Angular isn't just an MV\* framework, it's an opinionated MV\* framework. What is opinionated software? Well, opinionated software guides you into certain ways of doing things. Opinionated software has a vision. It may limit itself to solving fewer problems, but generally, software with opinions solves those problems with less fuss. That doesn't mean that there isn't more than one way to solve a problem with Angular. For example, take DOM manipulation. Angular wants you to only manipulate your DOM inside of directives, but with Angular inside of a directive you can use jQuery, Zepto, any other library, or even just raw JavaScript, to manipulate your DOM. That doesn't mean that you can't manipulate the DOM from within any other piece of an Angular application, but Angular definitely has an opinion about whether or not you should, and it gently guides you into doing things a certain way. So why should you use opinionated software? There are many reasons. At first glance it may seem like a lack of freedom is a bad thing, but that's really not true. The guys at 37signals, the company which invented Ruby on Rails, have even written a great article on the benefits of opinionated software. You can find it here. In essence, they say, the best software has a vision, the best software takes sides. This is quite true. You will find that using opinionated software will make it faster to do things that fall within the vision of the software. When it comes to Angular, that vision is extremely broad. It is well-suited to building anything from line-of-business applications to video players. In fact, if you are watching this video on your computer, then you are using an Angular application. But first, let's make sure that we understand exactly what an MV\* framework is. The M in MV\* stands for Model. The model is where you store the data and state of your application. The V in MV\* stands for View. The view is where you actually render to the user the information that you want them to see, and the view is where your receive input from the user. The \* in MV\* stands for, well, something else. In many common MV\* frameworks, the \* is either a controller, or a presenter, or a ViewModel, or even something different besides those three. In fact, some frameworks even allow you to choose one of those three, and don't necessarily prescribe which one. Now you may be asking yourself, which one of those does Angular use? Well, Angular uses a controller. So some people may call Angular an MVC framework that would definitely be true, but in the web world there are so many frameworks that fall within this categorization, but don't necessarily fit into just MVC, that this term MV\* is commonly used. Angular is an open source library built by the folks at Google. This gives us the best of both worlds. It is maintained by a team of highly-skilled professionals whose employment is based around building Angular. That means that you won't have to worry about it stagnating because the primary contributors get bored or busy, but because it's open source, it can benefit from the contributions of the entire JavaScript community. In fact, at the time of this recording, Angular has over 100 unique contributors to its GitHub repository. Angular is also comprehensive. What makes Angular comprehensive? Let's take a look at a diagram. Angular handles the Ajax communication with your server so that you can both send and receive data from your backend. This data is stored as plain JavaScript objects, so you won't have to make any special calls like get and set when you to update your data. Angular handles showing that data on the page, which you can do using partial templates or just modify the HTML DOM that already exists. Angular also handles updating the data or model based on user interaction, so when a user types into a text field, that new value can automatically be copied into your model. You don't have to wait for certain events, you can just tell Angular that a certain textbox owns a certain piece of data, and it will keep them in sync. This feature is called two-way binding. And lastly, Angular handles routing, or moving from one view to another. This is the key piece in building single-age applications or SPAs. This way you can completely change your view based on user interaction with your page. Angular will also update the URL in the browser so that the new view can be bookmarked for later. The next important aspect of Angular is its testability. Angular was built from the ground up with testing in mind. When the Angular team built the first versions of Angular, testability was a primary consideration. That means that Angular not only supports isolated unit tests, but it also supports integrated end to end tests. Also, a lot of the built-in objects that Angular provides have special versions that you can use to make your testing easier. In fact, testing is so critical to Angular, that while building the framework the team created a fantastic testing tool which used to be called Testacular, and is now called Karma. This tool is used by lots of development shops to test their code, and many of them aren't even using Angular. Perhaps the one attribute of Angular that sets it most apart from the rest of MV\* frameworks is the fact that it extends HTML by providing its own elements and properties called directives, that are used to interact with your HTML DOM. Basically, Angular lets you teach your HTML new tricks. Imagine if HTML had been designed for applications and not for documents. That is how Angular thinks of HTML and the ability it gives you to teach your HTML new tricks is a result of this thinking. Let's look at some examples of this. Here I have a simple input tag, but look at this last property. This is a custom attribute that I might build that tells Angular to make the element have focus when the page loads. What some of you may recognize is that HTML5 already has an attribute like this, it's called auto-focus. So HTML5 is already thinking along these lines. The problem is that that property only works in newer browsers. A custom directive like this one can work in a much wider range of browsers through the magic of Angular. Here's another example. This time I'm being a little more ambitious. This is maybe a toggle button or possibly even a button that cycles through several states, but here I can create some kind of a multiStateButton that works the way I want it to work in my application. In my HTML, all I see is this, but after Angular gets done processing this element, the browser will see the HTML that it needs in order to accomplish what I want. And finally, something even more ambitious. This is, perhaps, a display widget customized to a particular application that displays users in a specific manner. I just have to pass in a user object using the user property that I created on my custom tag, and Angular knows how to turn this custom HTML element into HTML that my browser can render. Think of how this keeps so much of your view logic where it belongs, in your view. The last important attribute of Angular that we will discuss is how it is forward thinking. Angular is basically supporting the future of what we will see in web technology in the coming years, and as that technology becomes more widespread, our Angular applications will already be built to take advantage of that technology. Let's look at a couple specific technologies that apply here. Remember when I talked about what HTML would be if it had been designed for applications and not for documents? Well that's what web components are. Web components allow you to make truly encapsulated components and widgets for your page, encapsulating HTML, JavaScript, and CSS. You can look at this article for more information about web components. Another up and coming feature that will soon be supported by some browsers is Object.observe. This technology lets you watch an object or a property on a JavaScript object for changes and react to those changes. Most MV\* frameworks make you stick your data into special structures and call methods whenever you want to read or write to that data. Because Angular doesn't do that, it can support Object.observe when it becomes widely available, and Angular will simply benefit from the performance improvements of having things handled by faster, lower-level code. You can read more about Object.observe at this URL here. Now one of the last things I want to do is take a quick look at Angular's official site. You can see it right here in my browser, and there's actually a lot of content on there. On the home page there's a lot of introductory text about it. There's this Learn section which has links to videos, and tutorials, and case studies, and a link to the Seed project template, which we will cover later on, but a key piece of the Angular site I want to show you that you're going to use and refer to a lot is under this Develop link, and it's these two links right here, the Developer Guide and the API Reference. Let's look at the Developer Guide first. This page has a list of all of the concepts that are part of Angular, so anytime you have any questions about something specific to Angular, you can come in here, look for the concept, say we have a question about modules, come in here, click on Modules, and look at Angular's official documentation on modules. Now let's look at the API Reference. This page has a list of a lot of the very specific pieces of Angular, for example, a list of all the directives that Angular provides is given right here. If we scroll down a little bit we can see a list of all the filters that Angular provides, and a list of all the services that it provides, and a lot of other stuff. So those are two pieces of Angular's official site that you ought to be comfortable with and visit frequently. Let's review with a quiz. Angular thinks of HTML as if it had been designed to do what? Angular thinks of HTML as if it had been designed to build applications instead of documents. What kinds of tests does Angular support? Angular supports both unit tests and integrated end to end tests. Name one of the ways that Angular is forward thinking. Angular is forward thinking because of its future support for web components and Object.observe.

Angular Architecture

Let's take a look at some of the architectural choices that Angular has made. First, Angular supports two-way binding. This means the user input into form fields is instantly updated in your Angular models. That means that in most cases you don't need to watch for specific events and respond to them, and then manually update your HTML. Instead, Angular will handle that for you. Angular also employs a technique called dirty checking. The net result of this is that you don't have to put your data into special structures and call getter and setter methods to read and write to your data. you can simply put your model data into plain old JavaScript objects and Angular will respond whenever your data changes and automatically update your view. Lastly, Angular is built on dependency injection. This lets you encapsulate pieces of your application better and also improves testability. You can read more about dependency injection here. In addition, Pluralsight has an entire course on this subject which is called Inversion of Control by John Sonmez. Go check that course out if you want to learn more. Now let's take a look at the primary components of Angular and their relationship to each other. With Angular, everything starts with the controller. The controller is the central player in an Angular application. Controllers contain both logic and state. Next we have the View. Views are made up of bindings and directives. This is how Angular talks to and listens to the user. Controllers can communicate with views through both one-way and two-way binding. Directives, which are a heavily talked about piece of Angular, are really just part of the view. And the last major piece is Services. Services give you a place to contain the real logic and state of your application. If you think about what is the essential tasks of your application, this would likely happen in your Services. Complex business logic, important application state, etc., Services are the place to house all that. Also, Services are the place where you will want to communicate with the server. Alright, let's review. What is the central component in an Angular application? The central component in an angular application is the controller. Directives are a part of which component? Directives are part of the View. In which component should you put your complex business logic? You should put your core business logic in the Services.

Demo: Hello World in Angular

In this section, I'm going to build the simplest possible Hello World application that I can using Angular. If you look at my screen, you can see that I've got a minimal project with one file in it, an html file, and inside of that html file I've got a very small amount of HTML. I'm going to start by adding a script tag to point at the Angular library, and I'm going to point at the Angular library on Google CDN. I'm going to use version 1.4.0, which has been recently released, but you can use the latest stable version of Angular. Now that I've got that script file here, the next thing I need to do is tell Angular that this html page is an Angular application. I'm going to do that by going up to the html tag and adding an ng-app attribute to the html tag, and I'm going to set it equal to "app". This is going to be the name of my application. Now that I've got that in place, the next step will be to add some HTML will display our Hello World message, so down in the body I'll add an h1 tag and I'm going to do two things to that h1 tag. First I'll give it a special attribute called ng-controller, and I'm going to set that equal to the value of HelloWorldCtrl. This will be the name of the controller that I'll create in just a second. And inside of that h1 tag, I'm going to make the content two curly braces, which tells Angular to replace this value with something else, and then I'm going to give it the value of helloMessage. That will cause Angular to look for a helloMessage variable and put its value inside this h1 tag. Now I've got to write a little bit of custom code and create this controller, which I named HelloWorldCtrl, and then create the helloMessage property. I'll do that by creating a new script tag, and inside of that I'm going to call angular.module. This creates a module which is a container for every piece of an Angular application. The first parameter of this function is the name of the module, which is going to be "app", which matches the name I gave it up in my html tag, and the second parameter is a list of other modules that this module that I am creating depends on. In this case, I'm not depending on any other modules, so I'm going to give it an empty array. Now that I've created the module, I can call the controller function, which will create a controller, which takes in two parameters. The first is the name of the controller, which is going to be HelloWorldCtrl, which matches the name I gave it in my h1 tag, and the second parameter is a function, I'm going to put this on another line to make it easier to read, and I'm going to have that function receive one parameter called $scope. This is a special parameter which we'll talk about more later on. And inside of here I will call $scope and create a new property called helloMessage, which matches the value that I put inside the curly braces on my h1 tag, and I'm going to set it equal to the string Hello World. And now if I open up that file in a browser, we will see the message Hello World.

The Angular Event Reg Application

In this course, we will be learning about Angular through building a real application. This application is called Angular EventReg. Now we're not talking about some simple Hello World application. This application will have a reasonable amount of features. When we are done, we should have written around 500 lines of JavaScript and 200 lines of HTML, plus tests. Now realize that this is a showcase application and not necessarily a reference application on best practices, especially for things that are external to what Angular itself actually handles, such as CSS, deployment, data access, performance, and other concerns like that. We will be showing quite a few best practices with Angular itself, but as Angular is relatively new, many best practices are yet to be discovered and Angular is evolving all the time. New versions are regularly release, and with each new version new best practices are waiting to be found. One of our goals here is not to just teach you how to use Angular, but how to use Angular in a real-world application. This should give you the confidence you need to go and start your own Angular application from scratch. Now let's look at the major features of EventReg. EventReg is an application for creating and viewing Angular events or conferences, and for viewing and voting for the sessions available at these events. So the first major feature of EventReg is the ability to show a list of Angular events. From that list, the user can click on an event to see that event's details including a list of sessions and the details on each session. Also users should be able to create new sessions and events. There will be some validation here so that the necessary information is available on each event. Users who create an event or a session should be able to edit that event or session. Next, any operations that require knowing who the user is should require the user to login. As such, users should be able to freely register a new user account, and should be able to edit their account. Since we want to have a realistic application, we are going to be sending data to and from a server, therefore we will need some kind of server technology in place. You can really use any technology that you want, but in this course we will show you how to build and use both a node server and an ASP.NET MVC server, if you're on Windows. Of course the node server will run on both Windows and Linux. We will be making Ajax calls to the server just like we would in a real application, so it will show us what it's like to run Angular in a more production-like environment. For this application we will be using Twitter Bootstrap to make the styling easier. In addition, we will be using an open source theme for Bootstrap so that we have a slightly more interesting look and feel. Here's a screen shot of the finished application with that Bootstrap theme. So that, in a nutshell, is the application that we are going to be building in this course.

Angular Version

This course was originally authored using Angular version 1.0.5. Since then, several major versions of Angular have been released. The current version of Angular is now 1.4. We have updated the course to be compatible with this version. In order to make the updates as seamless as possible, we have gone through all the demos and updated them to be compatible, and we've also gone through the entire course and updated it for 1.4. In some cases, those updates involved a complete rerecording of a clip, and in some cases the changes were minor enough that we simply added an explanation that was different, or left the clip alone. So it's possible that you might occasionally see a place where it looks like we're using version 1.2 or 1.0. You can safely ignore this, the course is up to date for 1.4. As newer versions of Angular come out, we will continue to update the course to be compatible with those versions. If you're following along, you should almost always be using the latest stable version of Angular. That might be 1.5 or even 1.6. You should always check the GitHub repo mentioned earlier, which will tell you if anything is out of date, but otherwise you can use the latest stable version of Angular.

Tools Used

In this course, you will see a lot of work done from the command line. Although we recorded this course using Windows boxes, we are going to be using the Bash Shell for our command line work. For those of you who have only developed on a Windows Box it may seem a little foreign, but even though it may look a bit different from the Windows command prompt, you can follow along using either the standard Windows command prompt or the Bash Shell. None of what we do will be unique to the Bash Shell, and there will be only 1 or 2 places where the commands you use won't be exactly the same. In those cases we will note the difference. Most of our command line work will be running prebuilt scripts. From the Bash Shell we will run scripts that end in the sh extension. For Windows users, you'll be using batch files. If you have the downloaded code, you can choose between the .sh and .bat files based on your chosen command shell. If you're wondering how we installed the Bash Shell on Windows, it is installed when you install Git for Windows. The editor you will see us use in this course is Webstorm. Webstorm is a product by JetBrains, the same folks who make IntelliJ and ReSharper. The reason we chose Webstorm is because it is an excellent product. It gives you a lot of the features that you can't get from most text editors, but it's lighter weight than a full IDE like IntelliJ or Visual Studio, and it's integration with the Karma testing tool is second to none. You'll get to see that in the last module. Naturally, any text editor is fine to follow along. Nothing we do will be dependent on Webstorm. The last tool we need to talk about is the web server. As mentioned previously, you can use any web server that you want. In this course, though, we're going to show you how to create and use two different web servers, a Node web server and an ASP.NET MVC web server. You are free to use whichever one suits your purposes best, or if you're very familiar with another technology you're free to create your own. The web server that we're going to use is very simple. The web server itself will be responsible for not only serving up the files that we create, but also we're going to be sending and receiving Ajax to that server. In order to save that data, we're just going to write it to the hard disk, so therefore requests will come from the browser to the web server, which will then persist the data to the hard disk and correspondingly pull the data from the disk and serve it down to the browser when requested. Files relating to the web servers will be available on the GitHub repo for this course.

Summary

In this module we started by taking a look at JavaScript MV\* frameworks, some of the various options there, and how Angular fits into that world. Then we looked at some of the benefits of AngularJS such as it being forward thinking and very testable. After that we took a brief look at the application that we will be building throughout this course, the EventReg application.

Angular Controllers & Markup

Introduction

Hello and welcome to the Angular Fundamentals course. My name is Joe Eames. In this module we will be learning about markup and controllers. Let's look at the agenda. We are going to start by looking at controllers and scope. We'll create our first controller in this section and we'll look at how the controller and the scope interact. Next, we'll look at how to tie that controller to a page and how markup lets us display data on our page with binding. We'll also look at how to receive events from the user and how to use expressions when binding. Then we will look at filters and how they let us modify what we tell Angular to do. Lastly, we'll look at how the built-in validation works in Angular. Before we get into learning about markup and controllers, we need to set up our starting project. We have created a skeleton project that we'll be using through this course, and slowly building up a full application. You can download that skeleton project by going to the GitHub repo for this course. The URL is shown here. This is also a great time to check and make sure that the course is up-to-date. The file we want is this DemoApp.zip file right here. You can either clone the repo or download the whole repo as a zip over here, or just download the demo app file itself. Once you have downloaded and extracted the file, go ahead and open it up in whatever editor you prefer. Again, in this course we'll be using WebStorm, but any editor will work just fine, whether that's a simple text editor or a full-blown IDE like Visual Studio or Eclipse. So let's take a moment and walk through the pieces of the sample project here. Here I've got it open in WebStorm. I'm going to go through and start out by looking at the app directory. This is the directory that's going to contain all the files that we will write. We've got a directory here for our CSS. Inside of here there's an app CSS that contain the custom rules for our site and the Bootstrap CSS file. This is Bootstrap Version 2. Bootstrap has been upgraded to Version 3, but because this course is not about Bootstrap, we are still going to be using Version 2. If you want to learn about the latest version of Bootstrap, Pluralsight has a great course on it that you could watch. Next we got an image directory for images. The JS directory is where we'll put the code that we write. Inside of here, you can see I've got two folders, one for controllers and one for services. This will be where we place our controllers and services, each inside of their own file. We've also got this app JS file right here, which we'll look at in just a second, then we've got the directives and filters files. These are convenient places to put directives and filters. We'll be dealing with directives and filters later on in this course. Let's open up that app JS file and look at the code very briefly. Here in our app JS file, this one line of code is doing several things. First off it's creating a module named eventsApp. Modules are the basic building blocks of Angular. Everything exists inside of our module. This empty array tells Angular that our module does not depend on any other modules. Later on, we will add set of dependencies to our module. But for now it has done. The other thing we are doing is capturing that module into a global variable called eventsApp. This will make it easy for us to create objects inside of this module, such as controllers and services. There is also this LIB directory, this is where we place all of our third-party code. You can see here we've got the Bootstrap files, the jQuery file and underscore file, and also where we've got our Angular files. The sample project on GitHub will have the Angular files in it. But if you want to use the absolute latest version of Angular, then you can go to code.Angularjs.org and download the version that you want. You can see that the current stable release for Angular is version 1.4.0 right there. And if you wanted to download the latest files, you just click into that directory, and then take the files that you want to download. You will need to be sure and download the same files that are in the starting project and replace them with the specific versions that you want. And you can either grab the minified or non-minified version. For our purposes, it really doesn't matter which one and throughout the project you might see us use the minified version or the non-minified version. That only matters when you're deploying your app to production. No other you use the included version of Angular or the very latest release that will affect what we do in this course. The next folders are scripts folder. This will contain our node server, if you choose to use it. Right now this file is just a blank file, but we'll build it in the next sections. As mentioned previously, you can choose between a node server and an ASP.NET MVC server. The next two clips will show you how to build each of those. So choose the appropriate clip or you can skip them and just build your own server. For now it just needs to serve up the files, later on down the road we'll add more capabilities to the server. We've also added these two command files: server.bat and server.sh to assist you in launching the node server. Use the SH file if you are on a Linux box and the bat file if you're on a Windows box, or ignore them if you're going to use the ASP.NET server. In addition to the directories that we've looked at, there is a test directory which we won't look at now, they'll use later on in the course. In the last file that you see here is this readme file that just has some simple directions about the project. You may also notice later on in the course that some of the clips will show a directory named Config and one named Logs. You can safely ignore those folders, they are not used in any part of this course. All right, let's review with the quiz. Can modules depend on other modules? Yes, models can have other modules as dependencies. Should a small application use multiple modules? No, in general, small application should use just a single module. There are reasons to use multiple modules. For example, if you have pieces of your application that you might be sharing with other applications or for testability reasons, but those of the main reasons why should split a small application up into multiple modules. Does Pluralsight have a course on Twitter Bootstrap that you should watch? The answer to that, of course, is yes.

Demo: Creating the Node Server

In this section we are going to build our node web server and show how to launch it. Node is a very useful tool for developers. It provides lots of developer tooling and an easy way to run web servers. Now if you have an already installed node, Pluralsight has a great course on node that you can use, or you can just go to the node official website by Googling for node, and there is great information on how to download and install node. We are going to be building an express web server, so we are going to go to our command line and we are going to install Express. Now there is two ways that we can do this. If you downloaded the starting project from GitHub, then you can just type in npm install. This will make npm look inside the package.json file and install the correct versions of Express and the body-parser module, which express needs. Or if you didn't download the starter project or you'd just want to know how to install these manually, you can run this command, npm install express, then we are going to use an @ sign so we can install a specific version of express, which is 4.13.0, a space, then body-parser, another @ sign 1.13.1. This will install the compatible versions of Express and body-parser to make sure that they work with our demo. No matter which way that you install it. At this point you just hit enter, and those models are now installed. If I go back to my project, I'll notice that I now have a node modules directory. And now I'm ready to create my express web server. The first line of code I'll write is to bring in the Express library itself. This require call grabs the express library and makes it available for me inside of this file. I'm also going to grab the path library. Next, I'll create the express application by calling the express function. I'm also going to create a root path variable by calling path.normalize and then pass into that underscore underscore dirname, D-I-R-N-A-M-E, which is a variable that points at the current directory, which in our case is going to be the scripts directory that our webserver.js is inside of. So I'm going to add to that a string which is slash dot dot slash, which will take me up a directory to the root directory of my application. Next, I'm going to tell express to serve up my pages using app.use, calling express.static The static function serves the pages in a given directory without processing them at all, and I'll give that a parameter root path, and my app directory. This will tell express to serve all the files inside of my app directory without processing them at all, just passing them out as they are. Then I'll tell my app to listen on a specific port and the port that we'll use is going to be 8000. Of course, if you've already got this port in use, you can change this to a different port. And finally I'm going to log out to the console. A message that lets me know that my server is now running. That way when I launch the server, I get a message that says everything is okay. And now that I've got this file created, I can launch it by going back up to the command line and using the server.sh file, or if I'm on the Windows command line, I'd use server.bat. We execute that file and we get our console message saying that it's now listening on port 8000 and a test that's running, I'm going to go out to a browser and I'm going to go to the URL: localhost:8000/image/Angularjs.logo.png. The reason I'm using this URL is because I don't really have any other files yet in my application that are going to be able to be processed, so it's simplest just to test this by viewing an image. And if I bring that up, we can see the image for the Angular logo, and we know that our web server is running.

Demo: Creating the ASP.NET Server

In this section we are going to be creating the ASP.NET version of a web server for our application. Initially we are just going to create a very simple website in IIS. Later on we'll add some ASP.NET MVC components to it. So to start off with, I'm inside my internet information services manager. I'll go to my sites folder, right-click and say add website. For the site name I'm going to call this DemoApp. For the physical path I'm going to use C:\demos\demoapp\app. Of course, you want put in the correct path to your app directory inside of your demo application. Next, I'll go down and change the port from 80 to 8000, and that's all I have to do. Now I can click okay and my website starts up. And if I go over to a browser and I go to the URL: localhost:8000/image/Angularjs logo.png, we can see our website is working because the Angular logo has appeared.

Angular Controllers & Scope

In this section we are going to talk about controllers. But we can't talk about controllers without talking about scope. So let's look at the relationship between controllers and scope. A controller's primary responsibility is to create a scope object. A scope object is how we communicate with the view, and the scope is able to communicate with a few through a two-way communication. The view can bind the properties and results of functions on the scope, and events on the view can call methods on a scope. Data passes in this way from the controller to the scope, and from the scope back and forth to the view. The scope is used to expose the model to the view, but the scope is not the model. The model is the data that is put into the scope. If we want to modify the model, we can either use methods that are on the scope to modify the model, perhaps in response to events fired by the view, or using two-way bindings we can modify the model. In this way users through the view can make modifications to the model, or in other words can make modifications to the data. Let's review with a quiz. What is the primary responsibility of the controller? Controllers primary responsibility is to create the scope. Is the scope the model? No, the scope is not the model. The scope merely contains the model. Can the view bind to functions on the scope? Yes, you can bind your view to both functions and properties on your scope object.

Demo: Controllers

In this section, we are actually going to see controllers in action. We are going to do that by creating our first page. So I'm going to go on my app directory and then right-click and say new file. And I'm going to call this file EventDetails. This page is going to show the detailed information about a single event, both the information about the event, such as the name and location and time, and also the information about each session in the event. So I'll start out this page by just pasting in a very simple HTML header. I'm using the container class from Twitter Bootstrap. Then I also need to close up my body and HTML. In addition to this, I'm going to want to put on my script tags in, so that Angular can run, and I can have references to all the Angular pieces that I need. So, of the scripts that I pasted in, the first three I just pasted in in case I need them. JQuery, underscore and bootstrap. The real scripts that matter here are Angular and app. Angular, of course, is the Angular library, and app JS is my starting script that loads at the module for this application. And now we are actually ready to make this file do some stuff. So we are going to start out by going up into the HTML tag. I'm going to add my NG app. Now remember in our app JS file, we named our app eventsApp, that's this right here. So I'm going to put that same name over here, and that tells Angular that this is the module that runs this page. Now that I got that in place we are going to add our first controller. Right here I'm going to add a div, and I'm going to tell Angular that this div is owned by one of my controllers. I named the controller, event controller, this controller is going to be responsible for rendering information about an event. So now I've told Angular that this is the controller that manages this div, but I haven't actually created this controller. So I'll do that by adding it to controllers. And now we can cut out this controller. Now I always like to be in strict mode. And in order to create a controller, I'm going to reference that same module, which I created in the app JS file, which will remember his name in eventsApp. And use the controller function then pass in the name of the controller. And now my second parameter is actually a function, which will be the controller itself. Now remember how we talked about the relationship between controllers and scope, controllers are responsible for creating scope. So they actually receive an empty scope in, and then they populate that scope with the information that is going to be needed by the view. Now that I got my controller created, I'm going to add the script file to the page that we created. And I'm going to do that after my app JS. And now that's all hooked up, let's go ahead and view this in the browser and see how it looks. So you can see that there is really nothing exciting going on. We haven't put any render void to be on the page, and the controller isn't do anything. So let's go back, and let's actually make the controller display something. I'm going to go on to the controller and I'm going to create an event item. Now this event item is going to represent an event. I'm just going to use an in-line JavaScript object here. In a real application, you actually pull this from a server somewhere. So I'm going to give my events some basic information. Now let's go and display some of this information in the page. When I go inside of the div that's controlled by my controller, and I'm going to add some Angular markup that will display the name of the event. Let's go look at that in the browser. We can see that it's displaying the name of the event. Let's add a few more items. Okay, everything is rendering out okay. And let's analyze what we've done. You can see here these double curly braces. These till Angular that whatever is inside those curly braces is an expression that Angular should evaluate and then replace and put into the page. So I prefixed everything with event. Let's go back to our controller, and look at what it's got. The scope object has an event object. So what's going on here inside the page, anything within this div that is controlled by my controller has any object that exists on the scope. So event which also exists on the scope is now available, and I can write it in and access any properties and print them out to the page. This markup format with these expressions is the simplest way to put information on a page with Angular. So now we got basic information displaying on the page. But let's make it look just a little bit prettier. So let's go in and add a navigation bar. Okay, let's see how that looks. All right, better. And let's also go into our event controller and style that up a little bit. All right, let's see how that looks. Okay, now that's trying to look a little bit better. Now some information we are missing about our event is a location where the event is located at. So let's go back into our controller and let's add that information. But this time I don't want to just put the address right on the root of the event object. It makes more sense to me that there is a location object that contains the address information. Else go put that into our page. Now let's check that out. All right, looking good, and we can see the benefit we are getting out of Twitter Bootstrap. This makes everything look a lot nicer. Now let's talk about what we've done here. You can see the inside of this markup, I'm actually drilling down inside of the event, inside location object to a property that I want to render. Again, this is Angular handling this for us, and actually allowing us to drill down in so we don't have to format our data in a simple single level. Instead we can actually let the data be shaped in a way that makes sense. Now one other thing I would like to add that I think will make our sessions look a little sharper, is some kind of an image to indicate the event. I'll throw one of the logos for Angular into my image directory, and I'm going to reference that. So now I can go back to my HTML page and right up here at the top, I'm going to add that image. So you can see that I'm using the same kind of markup that I'm using down below in order to get the image URL out of the event. But I'm also doing something a little unique. Instead of using the source attribute on the image, I'm using the NG dash source attribute. What this does is delays the requesting of the image until Angular has a chance to replace this text right here with the actual URL for the image. Otherwise as soon as the HTML renders, your browser will kick-off a request to find an image of this URL, which of course is going to return a 404. I'd also like to get this an alt attribute. I'm going to use the event name again. All right, let's see how that looks in our browser. Great, this is starting to come together. Now one more small adjustment I want to make is just add a little bit of extra padding on the sides.

Demo: Displaying Repeating Information

All right, now that we've got the basic information for the event showing up, let's talk about displaying information about the sessions for an event. If my event had some sessions, I'd imagine the data would look something like this. Now I want to render these sessions inside my page. So let's go back to the page, go way down here. Let's add a little horizontal line to separate things out, and then we'll display our sessions. All right, so now I'm going to display multiple sessions. I don't know how many sessions each event is going to have, so I'm not going to put in a certain number of sessions that matches that event. Instead what I want is to create a list item for every session inside of the event. This is easy to do. I'm going to instruct Angular to repeat this item using the ngRepeat directive. And now the format of this is pretty straightforward. You can see what I've done is inside here I've created an object called session that will repeat for every time there is a session object inside of the sessions variable within the event. Now that I've got that, I can display the name of the session. Let's go look at that in our browser. Okay, we can see that our session names are now putting out. I'm not really crazy about the layout, but at least we got a start. So let's fix our layout just a little bit. All right, let's see how that looks in the browser. Okay, that's trying to look pretty nice. Okay, let's go in and add some more information to our sessions. All right, now we've got that information inside the event object, let's put it on our page. Okay, we'd use the same syntax we used before to add in the information about the session. Okay, that's looking pretty good. All right, now we've got our page pretty well created. We've been able to see through this, the basics of getting the information from an Angular controller on to an Angular page, and associating that controller with sections of that page. While we did that, we saw the basic expression syntax for getting data out of a scope and putting it on to the page. We've also seen a couple of Angular directives, the NG source and the NG repeat, in addition to NG controller and NG app.

Demo Handling Events

Now that we are able to render data on our page, let's see about responding to events. One of the things we want our event page to be able to do is to allow people to vote for sessions, so the popular sessions can be chosen for an event. So let's start by going in, and on every session we'll add an upVoteCount. Now you can see I am setting this to zero. Eventually that number is going to change. Now that I've got the upVoteCount available in each one of these sessions, I am going to need to somehow increment that count and display it on the page. So we are going to add another little section on our page. This piece of our page will be a voting widget that allow people to vote up or down for sessions. And here is where we'll actually display the upVoteCount. Let's see how that looks. Our voting widget looks okay, but I wanted on the same line as the information about the session. So I'm just going to move it down inside the row. Okay, that is looking much better. Now if I go into my data and I change some of these upVoteCounts, they'll reflect inside of the page. That's all great, but what I can't do yet is actually vote, so let's implement the voting. What I want to do is I want to be able to respond to a click event on that little icon showing up, thankfully with Angular doing that is really easy. I'll just go into my div and I add an NG click directive. And I want to call upVoteSession. Now this is a method that doesn't exist yet, but I'll add it. And I want to pass in this session that I'm going to be up voting. All right, now let's do the same thing down here. Now since those methods don't exist, if I try to run this right now I'll get an error. So let's go and create those methods on the controller. The first one if you remember was named upVoteSession, and that takes in a parameter of a session. Okay, that's simple. Let's add our downVoteSession. All right, let's see if that works now. And look at that, and if we click that up chevron, the upVote is going up. Now one of the things we should do is clean up after ourselves a little bit and reset the initial upVoteCounts to zero. All right, now let's talk about how we implemented that functionality. You can see that these methods, upVoteSession and downVoteSession are very simple. And what's really nice about them is instead of having to take it like a session ID, they actually just take in the session we want to upVote. And that way we don't have to look the session up, we actually have access to the object itself and we want to manipulate it. Let's look at how that's possible. What Angular is doing right here when we call this method is, it's going to call that method for us whenever we click, and it's going to pass in the session. Well, what's the session? Well, that's defined right up here. The session object inside of the current session. So Angular is actually matching up with sections of HTML, the session that belongs to that section. That way we can refer to the session itself rather than having to do something silly like putting the session ID on the data attribute and then looking it up in order to have our code be able to manipulate the correct session. This is one of the real beautiful pieces about Angular, the ability to work with your data directly, and not have to worry about constantly passing around IDs, storing IDs and data attributes, and having to look them up everywhere when you want to manipulate one of your pieces of your model. Now, of course, besides click, we can respond to lots of other events, and we'll be looking at that in the next section. But now we've learn the basics for not only displaying data from our model on an Angular page, but also for responding to input from the user and manipulating that data. Now let's have a quick review quiz. Can you use basic JavaScript objects as your model in Angular? Yes, Angular doesn't require any kind of a specialized object to be its model. What is the markup indicator for binding data to a view? The double curly brace is the binding syntax for Angular. Which directive do you use to bind an image's source? You use the NG source directive. This directive also allows you to delay rendering the image until Angular has had a chance to process and replace the source of the image with the correct value.

Built-in Directives

As we've been building our application we've already encountered a few directives. Now we have an entire module later on in this course devoted to building custom directives, but here in this module we are going to cover the built-in directives that angular provides. So now let's talk about what directives are. According to the Angular documentation, directives are a way to teach HTML new tricks. Essentially directives get HTML a new functionality. As Angular parses through your HTML, it will look for directives and then take action based on what it finds. So in the case of NG click, whenever it encounters an NG click, it will register a click handler event on that DOM object. If you remember, that was an attribute of a tag. There are actually four ways to specify directives with Angular. The first one is actually is the tag itself. For example, the NG form directive is a tag. The other way is the way that we've seen before with NG click where the directive is an attribute of a tag. Here we can see NG form right now as an attribute. And the third way that we can write directives is as a class. Now not all directives can be written out as tags, attributes and as classes. Often times a particular directive could only be written out in one or two of these forms. I mentioned that there are four ways to read our directives, and we have only looked at three. The fourth way to write out a directive, I'm only mentioning for completeness. And we are not going to look at any code samples for this format. This fourth way to write a directive is inside of an HTML comment. How to write directives using HTML comments is beyond the scope of this course. So we are going to be satisfied by just mentioning the fact that some directives can be written as HTML comments. For questions about any particular directive, always refer to the Angular documentation itself. Let's have a quick review. In how many ways can an Angular directive be written? Angular directives can be written a total of four different ways: as a tag, an attribute, a class and as an HTML comment. Can all directives be written all four ways? No. You should check the Angular documentation for specifics. Can Angular dig down into your model? Yes. Even if you have a model that has a property that's five objects deep that you need to bind to, Angular supports the dot syntax, so you can specify the path all the way down to the property that you wish to bind to.

Event Directives

The first set of directives that we are going to look at is the directives that handle events in Angular. We've already seen NG click. There is also an NG double-click, we've also got mouse down, mouse enter, mouse leave, mouse move, mouse over and mouse up. All of these directives are very simple and simply respond to a mouse event, and lets you call a method on your scope. The last event directive we want to look at is ngChange. ngChange will detect a change event on a lot of different HTML input elements. Let's look an example of this. Here I've got the code for a checkbox, and I've added the ngChange directive. This directive is going to call handle change whenever the checkbox is checked or unchecked. It's important to note that ngChange requires that the ngModel directive also be present on the tag. We are going to cover the ngModel directive later on in this module when we cover two way binding. Let's have a quick review. True or false: The ngChange directive requires the ngModel directive to also be present. That's true. True or false: Angular includes an ngRightMouseClick directive. That is false. Angular includes a lot of mouse directives, but right-click directive is not one of the ones it's included.

Other Directives - Part 1

All right, let's look at the other directives that Angular provides for us. These don't necessarily fall into one simple category like the event directives do, but they do a lot of different things, they are very useful to know about. We've already seen ngApp so we know what that one does. Next, we've got the bind directives, which are Bind, Bind Template, Bind HTML and Bind HTML Unsafe. Let's start off with bind and see what that does. So let's go to our application and we are going to make some modifications to it. Let's go down to our H2. So far we've been using the expression syntax to film the value of the H2 tag with the name of our event. But there is another way we can do this, and that's with the bind directive. So I'm going to remove this and add in the NG bind, all right. Now I'll go out to our website and refresh the page, and you can see that the title is still Angular Boot Camp. So that's bind. We've also got bind template. So I'm going to change this to a bind template. Now what bind templates lets us do is go back to the expression syntax, but we can put in multiple items in here. So I'm going to add event name, and then a space, and then event date. Now I'll go and look at the title. You can see that is giving us the name and the date together. There is a couple of different ways that we could have done this. This is just one of them, but this way could be pretty convenient. Okay, so let's look at Bind HTML and Bind HTML Unsafe. Now one of the things it's kind of tricky is that the Bind HTML directive is not actually in the main Angular script file, instead it's in the Angular sanitized script file. So we'd have to do a couple of things in order to get that directive to work. First, we go down to the scripts, we are going to add that script file in. So now we've got that included, we have to make another change. Need to go to our app JS, and in here we are going to have to put a dependency on the ngSanitize module. If you remember back we talked about having multiple modules and how this is where we specify dependencies of our module, so I'm going to type in ngSanitize. All right, now that we've got that in there, it's going to load the sanitize module before it loads our module, so that anything in the Sanitize module is available to ours. Now that I've got that I can go back up. And instead of the H2, I'm going to come down here and I'm going to add-in a new div. I'm going to do NG Bind HTML. All right. So I've got a div that's actually got some text content in it, but using the NG Bind HTML, which again is in that Sanitize module, I'm going to the replace the content of that with what's in the snippet variable inside of my scope. So I'm going to go to my controller, and I'm going to create a variable called snippet and I'm going to put a little bit of HTML in here. So I created this string that's valid HTML. It's a span with the red ForeColor. Okay, if you look right here, you can see that it got the 'hi there' text, but you'll notice that the ForeColor is not set to red. If we view the HTML, we can see that there is a span with a 'hi there' but the style directive is actually been removed from it. So this is what Sanitize does, it goes through and removes anything that could be dangerous set, but still allows you to actually have HTML inside some kind of a variable and populate a page with portions.html. Now we used to have the NG Bind HTML on safe directive, but that's been deprecated. The Angular team determined that it was a complex enough scenario that, it needed to be a little bit more than just a directive. So now there is a service involved, and it's still possible to bind unsafe HTML, but doing so is somewhat complex and will be on scope of this course, so we won't be covering it. The next directives we are going to look at are the hide and show directives. These two directives are kind of mirror images of each other. I'm going to go into my controller, I'm going to add a bullion, call bool value and set it to true. Now I'll go back into my page and I create a couple of H2 tags. One that says show this, and the other that says hide this. And I'm going to use the show directive on the first H2 and the high directive on the second H2. Now this is a good spot to reiterate one of the primary principles of Angular, and that is to only manipulate the DOM using directives and never manipulate it directly in your controller. Instead of using NG show and NG hide, I could have just grabbed a handle to the DOM moment inside of the controller, say using jQuery and then call show or hide on it. But that would be mixing the concerns of the controller class, which would create more highly coupled and brittle code, so never manipulate the DOM from within a controller. Or as we'll see in our next module, a service, only do that through a directive. So we can see the show this, if we look at the HTML. You can see that the HTML for the hide this is actually in there but it's got a style display none. So long as that bullion value is true, then the hide directive will hide the value. And as long as this is true, show will show it. But if I flip, show that bullion value to false, now the hide this is showing and the show this is being hidden. The next directive we'll look at is the cloak directive. This directive allows you to hide portions or all of your page until Angular has had a chance run and go through and parse the HTML and replace the directives or the bindings with the actual values that it's supposed to display. This is really nice because otherwise you might get flashes, especially on the slower client computer where it shows the Angular syntax and not the value you want to show. The simplest way to do this is to go into your page and add it to the body tag. Once you do this, we need to go into our CSS file and we need to add a special CSS rule. And that is this rule right here. This rule is actually documented in the Angular documentation for the cloak directive. So if you ever need to get it, you can get it from there. And what Angular will do when it encounters the NG cloak directive is process that and hide that element until it has a chance to parse the rest of it and replace it with the correct values, and then it'll unhide it. The alternative to doing this is to move your script tag for Angular from the bottom of the page up into the head, and now Angular will process sooner during the page, rendering lifecycle and find the NG cloak directives and replace them with display none, until it has a chance to parse the HTML, write it's coding in your code and replace all of the directives and binding values with their actual values. Unfortunately this is difficult to demonstrate because on a decently powered developer machine, that flash of uninitialized HTML is often so quick that it's difficult to catch. All right, next up, we've got the style directive. The style directive is pretty straightforward. In our code, I'm going to go over to our HTML page and maybe go take this H2 right here. I'm going to add the style directive, and set it equal to my style. Now I'll go back to our controller and at my style variable. And I'll set that equal to an object that is a CSS command. Now let's see that in the browser. And you can see it's made the H2 red. Now we've got some companions to the style directive. Those are ngClass, ngClassEven and ngClassOdd. They are class even and class odd directives, work just like class, only you use it inside of an NG repeat directive, and they will apply only to the even or odd elements. So let's look at an example. I'm going to go back to my controller and I'm going to add a variable on my scope called My Class, and I'll say that it's the class blue, and I going to go into my app CSS, I'm going to add a blue class. And instead of the style directive into the class, go to our browser and refresh that. And now the text is blue. Let's review we've learned so far with a quiz. What is the purpose of the ngCloak directive? The cloak helps us avoid a flash of unbound HTML. True of false: ngBind supports multiple bindings? False, the ngBindTemplate directive is the directive that supports multiple bindings. The ngClass directive has two companion directives. What are they? They are ngClassEven and ngClassOdd.

Other Directives - Part 2

The next set of directives are directives that allow us to do some funny parts of the HTML spec. In the HTML spec, attributes like disabled don't actually take a value, but instead the presence of the disabled attribute indicates that the element itself is to be disabled, and the value of the attribute really has no bearing, but certain browsers like Chrome and Firefox where you set the value, where other browsers like IE don't. So Angular give some attributes to deal with that. Those attitudes are the ngDisabled, ngChecked, ngMultiple, ngReadonly and ngSelected. These directive will either add or remove their respective attributes when you set them to a true or false value. So let's look at an example of this, we'll just go over the disable directive, but of the other directives work exactly the same. Turn over here, and before these divs let's create a button. Now we can see that button here in the browser, and you can see it's enabled, I can click it. Back over here to our HTML and let's add the disabled directive and set it equal to a value of button disabled. Now I go to my scope and I create a button disabled value and set it to true. Back to the webpage, refresh and you can see the button is now disabled. And the nice thing is that just like everything else in Angular, this binding is dynamic. So if I were to change the value from true to false, it would immediately enable the button. The next directive we'll look at is the form directive. The reason we have a form directive is that the HTML spec does not allow forms to be nested. But if you want a section to form off into smaller sections, say for example while doing validation, then the ngForm directive could be nested inside of other forms and allow you to create smaller sections of forms. The submit directive is a lot like the event directives that we saw earlier. It only allows you to call a method on your scope if the form is submitted, regardless of how that happens, whether or not you click a submit button or hit the enter key. As long as the form is submitted, instead of submitting the form, it will call your function. The next two directives are somewhat similar to each other. Those are the Href and source directives. The Href directive is providing the Href attribute on an anchor tag, and the source directive is for binding the source attribute on an image. As we discussed before, if you set the source attribute of an image to a binding, the browser will immediately try to request that binding as the URL and fail. So the source directive allows you to give Angular a chance to run its bindings before the browser request the image. Href is a little similar, in that if you have a binding to an Href on an anchor tag, and you worry that that anchor tag might get clicked programmatically or manually before Angular has a chance to run, then the Href directive will assure that that doesn't happen because the anchor tag won't actually have an Href attribute until Angular runs and replaces that directive with an actual Href attribute. The next directive is the NonBindable directive. This one is unique. In that it specifies an area that may look like an Angular directive, but you're telling Angular specifically do not parse and do the binding that happens inside of this directive. This is easiest to illustrate with an example. Let's go to our HTML and let's add a div. Now as we'll discuss in a moment, this is the expression syntax with Angular. So if we run this in the browser, what we see is the result of that, one plus two, which is three. But let's assume that we actually wanted the text, two curly braces and one plus a two to show up in our browser. We simply add the ngNonBindable directive, rerun and we see the actual text instead. And Angular won't parse what's inside of that element. And so with that, we've looked at the majority of the basic directives that are available in Angular. Let's finish up a quiz. What does the ngForm directive allow you to do? The ngForm directive allows you to nest forms within each other. Which directive delays fetching an image until after the binding has occurred? That is the ngSource directive. And why do we need the ngDisabled directive? Because of ambiguities in the HTML spec. For example, setting disabled equals false doesn't actually disable the control in many browsers.

IE Restrictions with Angular

If your website supports older versions of IE, there's a few things you need to keep in mind if you're going to use Angular. The first one is that you are going to have to polyfill JSON.stringify. If you don't know how to do that, then Googling this will quickly bring you to plenty of sites that will tell you exactly what you need to do. The other thing you need to do is whenever you use an Angular directive, don't use the tag form of the directive. Let's look at exactly what I mean. IE has some restrictions on custom tag names. The problem is that Angular directives in their tag form aren't recognized by older versions of IE. So, for example, the pluralize directive can be used as a tag, but in order to work with IE we can't use this directive this way. As we discussed previously, directives can be written in multiple ways, which are the tag form, the attribute form and class name form. So and supporting older versions of IE, never use the tag form of the directive, instead try to use the attribute form or class form. Here we see the NG pluralize written as an attribute. Remember that not all directors can be written in all three forms. Some of them have restrictions on which form they can be written in. Be sure and check the official Angular documentation for specifics. So if you are going to support older versions of IE in your application using Angular, be sure and follow these guidelines. It's worth noting that there are ways to use custom tag names in older versions of IE, but there is actually a few hoops you'll need to jump through in order to get that done. Let's have a quick review quiz. What you need to do to support older versions of IE with Angular? There are two things you need to do. First, polyfill JSON.stringify; and second avoid custom tag names when writing directives. What are the three ways that directives can be written? Directives could be written as tags, attributes or class names. Remember that not all directives can be written in all three ways, so be sure and check the official Angular documentation for details.

Expressions

In this section we are going to talk about expressions in Angular. Expressions are JavaScript like code snippets that you can put inside of the HTML of your page. They are generally placed inside of bindings, and typically these are variable names that we have inside of our scope, but expressions can incorporate more than just a variable name, such as this math expression. Now you should know that expressions are not full JavaScript syntax. For example, you can't call functions on the math object, or call an alert inside of an expression, but you can do a lot of things inside of an expression. Let's take a look at some of the possibilities you can use with expressions. So let's just go inside of our event details page, and right after the address let's add an expression. Let's start with something simple, just a little bit of math. Now let's look at that in the browser. And we can see the result of 30 has been printed on the page. Let's try something else, let's do a little bit of string concatenation. Okay. Now let's do something just a little bit funky. Let's create an array, and let's access that array and get the last object from it. And that works too. So as you can see there is quite a few things you can do inside of expressions, but as I said before there are some restrictions. This isn't a full JavaScript engine, so you don't have access to things like the math and alert objects. So you can see here that that doesn't work, but expressions do have a lot of power and learning what you can and cannot do inside of an expression will really help you as you are writing your Angular applications. So let's review really quick. True or false: Expressions support all JavaScript syntax. False. Expressions only support a subset of JavaScript. True of false: You can create an array inside of an expression. That's true. You can create arrays, another basic data types inside of expressions.

Filters

In this part of our course, we are going to talk about Angular's filters. Filters are a way for you to tell Angular that you want to modify something for output. The best way to think of them is as a filter that you run your data through and they modify that data before it is rendered. Filters can do three main things. The first one is formatting. This might include things like making strings uppercase or formatting numbers or dates. The next thing that filters can do is sort records in a dataset, so that when the records are rendered, they are rendered in the order in which you want. Because Angular's bindings are always dynamic, whenever you change the sort order in a filter, Angular will automatically update your HTML. The last thing that filters can do is actually filter the records in a dataset. Again, this is dynamic, so whenever you change the filter, the records that are output are automatically updated and the HTML is automatically adjusted. So let's talk about how you use filters in Angular. This code example shows how a filter looks. Using the binding syntax, you simply give the expression on the left-hand side, then a pipe character, and then name the filter on the right-hand side. That data is automatically sent through the filter and the adjusted output is then used as the binding.

Built-in Filters

Let's look at the filters that are provided by Angular. Angular provides us two filters that work with strings. These are the uppercase and lowercase filters. These do exactly what you would think. They take a piece of string data and either uppercase it or lowercase it. So here is our app as it currently stands. Let's say that we want the title of our course to be in all uppercase. We can just go into our event details page, go to where the event name is at, and using the format of filters we add a pipe and then the name of the filter, which in this case is uppercase, and let's see that in the browser. And you can see that it's uppercase, the name of the event as it rendered it. Angular also provides a couple of different filters to format numbers. Those are the number filter and the currency filter. Now we don't really have any numbers we are binding to an application, so let's just show an example. I'll go down here just above our HR and I'll create a new div. And inside of that div, I'll do some binding, and I'll just paste in the first several digits of pi. Now let's say I want to format that and only printout the first two digits. So I'm going to say number, and then two. Refresh that and you can see them that I'm only getting the first two digits of pi. But if I had a shorter number, let's say just three and I wanted it to always show at two decimal places, could be the same thing and it forces it out to decimal places. Now for currency, let's imagine that we are going to have a price with our events, and I'll just change this to 34. So I'll add the currency filter. And now you can see the price is showing up with both the currency symbol and the right number of decimal places. Now it's displaying the default currency symbol for my locale because they didn't specify different currency symbol. But you can add a parameter that indicates this currency symbol that you want to show. Angular also provides a filter for it to formatting dates. Let's open up our controller and let's imagine that inside of our data we are actually using real dates instead of just strings. So let's change this over to a JavaScript date. Now, of course, we don't want that number to print out, instead we want to format as a date. So let's go into our date and we'll add the filter, date, and we'll go at the parameter of medium. And we can see that it's taken that date and format it out according to its medium date formatting specification. I don't really like that one, so let's change it to medium date. And that one just gives us the actual date and not the time. There are a lot of different built-in formats for dates that Angular provides, and if you look at the official documentation you can see all the variations. And, of course, there is a custom format as well, so you can specify everything down to the commas and how many digits to the year and month and day and all that. The next filter that Angular provides is a JSON filter. This is mostly going to be used for debugging, in case you need to printout exactly what an object is at a given moment, you can use the JSON filter and write this out. So as an example of this, I'm going to go down in here and add another div. And this time I'm going to put an object in here. If you look at this object, it's got two levels to it. It's got three properties and the third property is another object. And then I'll specify the JSON filter, go back to our browser and let's refresh. And you can see that it's taken that object and printed out according to the JSON specification, which can be useful when debugging our program, and certainly could be useful in other applications as well. The last three built-in filters that we are going to look at are used with the ngRepeat directive, and they modify the dataset that is used in that ngRepeat. Those filters are orderBy, limitTo and filter. The orderBy obviously let you order your data by a certain field; limitTo allows you to limit your records to a certain number of records; and filter allows you to filter the dataset by an expression. In order to demonstrate this we are going to add some new capabilities to our application. Let's go down into our ngRepeat, and after the expression then I give the ngRepeat, I'm going to add a filter here, and I'm going to add orderBy, and I'm going to give it a parameter of sort order. I'm going to go to my controller and I'm going to specify a default sort order. I will set that to name. So by default Angular is going to sort our data by the name field. Now if we scroll down a little and look, you can see that the names of our sessions are actually already in alphabetical order by name. So if we go to our website and refresh it, we are not going to see anything new. But if we go over here and change this to something else, and let's change this to upVoteCount, again, nothing has changed, but if I come over here and vote, all of a sudden it's begin to sort by the upVoteCount. Now it's not doing what I want it to do because it's putting the vote count down at the bottom. So in order to make this go by descending, I simply put a dash in front of the expression. Now when I refresh and vote on one of them, that will go to the top. Now this is nice, but let's say that we want to allow the user to be able to change the sort order. So let's go back into our HTML and we are going to add a dropdown. Right after our session's header, let's create an orderBy, and let's set it to a select. I'll use the ngModel directive, which we will explain later on in this module, and set it to sort order, and I'm going to reset the class to input small, just so it's formatted nicely. I'm going to give it two options. And then I'll go back to my event controller and I'll make the default name again. Now let's view this. Okay, I'm ordering by name but I'm going to vote this up a couple of times, and then change the sort order to votes. And now you can see the Angular is automatically updating the HTML whenever I change this dropdown value and setting this sort order correctly to what I wanted to be. Now let's look at an example of the limitTo directive. Let's say for some reason, I only wanted to show two sessions at a time. I can go in here and I can add the limitTo filter, and give it a parameter of two. Now when we go back to our browser, if I refresh the page, you can see that I'm only showing two sessions, and not all three. In this particular scenario, that's not very useful, but in other scenarios this can be very useful to make sure that you are only showing a certain number of values from this dataset. Now let's add one more piece of functionality. Let's give the users the ability to filter the sessions by difficulty. In order to do that we'll have to go back into our HTML and we are going to have to give them another dropdown in order to choose the difficulty they want to see. And we'll create a variable called query to hold that expression. We'll default it to all, but we'll also add in an option for each of the levels that we've got. Now since the default is to show all, and that corresponds to no value inside of query, we don't have to set a default inside of our controller. So the only other piece we have to do is go down to our ngRepeat and add-in that filter. Now if we go back to our browser and refresh it, we can see that we've now got this other dropdown that indicates what level we want to see. And if we select a value, it will filter our dataset or intimate records that match that value. Now one of the things that you may have noticed is that nowhere did I actually specify which field I want to limit my query to. What Angular is actually doing is searching the entire record for any piece of data that matches. So, for example, if I go into our data and I go into this directives masterclass, which is of level advanced, and add in the word 'introductory' to the name, go back to our browser and refresh and select introductory, you can see it's actually still selecting these two records. So let's go back into our HTML and let's specify which field we actually care about. And we do that by going on to the ngModel and we specify that we are looking for a query.level. This will mean that we are only looking in the level field when we want to match this query. And then let's go back to our browser and refresh our page. And if we select introductory, now we are only seeing the record that has an introductory level. And that is a brief introduction to the built-in filters in Angular. Now let's review with a quiz. How do you indicate a parameter to a filter? The way you indicate parameters to filters is by after the filter name, you give a colon and then the parameter. Can you limit a filter to only search into a specific field? Yes, you can modify the filter by giving the field name as well as the filter name, and that will limit that filter to only look in a certain field. Can you specify custom date formats with the date filter? Yes. The date filter not only has a bunch of predefined formats, it also allows you to specify a custom format.

Writing Custom Filters

Writing our own custom filters is pretty easy to do. In order to create your custom filter, you simply start by taking an existing module and call the filter function on it. The first parameter is the name of your new filter, and the second parameter is a function. That function needs to return another function. That inner function is your actual filter. That function needs to take as its first parameter, the input that represents the input to the filter, than any additional parameters that you might want your filter to take can be specified after that point. Once you have that function in place, the next step is to write whatever logic is going to modify the input, and then lastly return the modified input as your output. As you can see this is pretty straightforward, and simple filters will be relatively trivial to implement. Looking at our application one of the things that I notice is the duration as specified as a string. And what's probably reasonable to assume that there are only as few certain durations that can be given for a session, and having the option to fill in the blank on your session duration isn't going to really work. So let's change our duration data, and instead of a string we are using integer to represent one of several different possible values. We'll give ourselves four different options for duration of sessions, and I'm going to make three of my sessions have a duration corresponding to the first, second and fourth option. Let's assume that those options correspond to a half hour, one hour, a half-day and a full-day session, representing one, two, three and four respectively. Now that I've got the data setup correctly, I actually need to create the filter. So now I'm going to go on to my filter's file and I'm going to write that filter. So we need to grab our module, which is eventsApp and call the filter method on it and pass in the name of the filter. I'm going to name this filter Durations. So you can parameter as a function, and that function returns another function. And that functions first parameter is the input, which I'll call duration. And now I want to return the modified value, so I'm just going to do a simple switch on the duration. And I know that that duration is an integer so I'll just setup four case statements. All right, so now my filter should return the correct value based on what the input value is, but in order to wire up this filter, I've got to go onto the HTML. So I'm going to go into my event details, and the first thing I need to do is go down to where the scripts are included and add that filter script. Now that the script is included, the last piece I need to put into place is to add that filter to the binding, and let's see how that looks in the browser. Okay, we can see that our durations have changed from their previous text values. It can be the display values that we created that are mapped to the integer values that are actually in the data. And there is how you create a very simple custom filter in Angular.

Two Way Binding

So far we've seen how to render data into HTML and we've seen how to respond to user events like clicks to update that data. But Angular still has one more trick up its sleeve when it comes to interacting with the user. That is two-way binding. Two way binding allows you to use the typical form controls and keep your model up-to-date automatically. This is an extremely valuable convenience. You won't have to worry about all the hassle of writing code to read the contents of the form filled, and then getting that data into your model. This capability revolves around the model directive. The model directive works with three different HTML elements: Input, select and text area. You can add the model directive to each of these three HTML elements. This will hook up two-way binding between the HTML form field and some item on your scope. Using the model directive is simple, here is an example of how to use the model directive with a text input element. You can see that the model is using some object.property. That object could belong to the scope and the property would be a property on that object. Of course, you can bind directory to a property on your scope, and if your model is more complex, using the dot notation you can dig down into your model to get to the property that you need to bind to.

Demo: Two Way Binding

Let's walk through some examples of using two-way binding with the ngModel directive. We're going to add a new capability to our application. We're going to make a page that will allow users to create new events. I'll start by going into my app and creating a new HTML page, and I'll call this new event HTML. Now in order to get to that new event HTML page, I want to be able to navigate to it through my menu bar up at the top. So I'm going to go to my event details page, and I'm going to go out to my nav bar and I'm going to add an item to it. All right, that will give us a link to that page. Now to start off with that page, I'm just going to grab all the HTML from this page and copy it into the new HTML page. I'm going to go up here and I'm going to take the contents of this div and delete them. Now I'll create a new div, and I'm going to give that div a controller, and I'm going to name that Edit Event Controller. I'm naming it Edit Event because at some point in the future, I plan on making this page not only create new events but also edit them. I'll throw out some padding to match the other page, create a container and a title. And now I'll create a form to contain the input elements that the user is going to put information into. Now I'm going to create a label and input pair for the name, date, time and location of the event. Now couple input fields for city and province, and lastly a label and input for an image. And how to make it convenient for users as they put in their image URLs, I'll create an image tag will be bound to that image URL. Like before I'm using the source directive for the image. And now I'll put in a couple of buttons, both the save and cancel. Okay, your HTML is pretty much ready. But one of the things that's going to cause us a problem if we try to render this is that there is no edit event controller that exists. So before we create that file, we need to go down and add it to the list of scripts included in the page. And now we can create that controller. And now that our controller is in place, let's go check out what we've done. I refresh the page and we can see we've got that new navigation event up at the top. Click on that and I've got my new event page. Now those looks really nice, but unfortunately it doesn't really do anything. No matter what I type in here, if I click save or cancel, nothing is happening. Of course, we saw that there is no click directives on these buttons, so nothing could happen. But more importantly, if I did handle the click event on these buttons, I would have no data inside my scope. What I want is to have data inside my scope. So let's go back into our HTML and add a model directive to each of these fields. Starting with the name I'll add the model directive and I'm going to call the name of this model event.name, and I'll add corresponding items to each of the other fields. Now if you'll refer back to the controller with the sample data in it, you'll see that I've ratchet up the names with the structure of the sample data that will be created. Okay, that should be the last one. Now since we have no click events on either of our buttons, not much is going to happen, but one thing did get wired up correctly, and let's see what that is. If I to come to the image and put in an image URL, we can see that that image tag is automatically bound to what I put into this image URL. As soon as I change this URL at all, it updates and tries to render the new image, but because that's not a valid image URL, nothing comes back. And that is to a binding. Going back to our code, we can see that the input box has a model of event.image URL, and the image has a source of event.image URL. That's created an event object on my scope and given it an image URL property that is bound to the image of source. And that right there is two-way binding. As I change the contents of that text box, that automatically updates the contents of the model and the scope, which in turn automatically updates anything bound to that. So, for example, if I was to come up in here and add in a div, and bind that to one of the properties on my event, that would get updated as well. You can see as I type, it updates the binding. This is one of the really beautiful parts of Angular. This sort of automatic two-way binding can really save yourself a lot of coding. Now for completeness, let's add some click handlers to our buttons. Now I want you to take note of something. Right here in this call to save event, I've passed in an object called event. Remember when we created the click handlers for the up-and-down voting, we passed in a session object. Here we are passing in the event object. Again, Angular is finding the correct object that belongs to this HTML and allowing us to pass it in to a function. So now when we create this function, that event object is populated with the data from each of those fields. We can prove this by using alert and printing out one of those fields. In here we see that that data is available back in the scope. Let's go back and implement that cancel button as a handler, and we'll have it navigate back to the event details page. All right, and there you have it. We've implemented two-way binding by creating form fields, giving them the model directive and binding those form fields to pieces of data in our model that are held inside of our scope object. Now let's wrap up two-way binding with a quick quiz. True or false: Two way binding will update on every keystroke? This is true. Which HTML elements work with two way binding? That is the input, select and text area elements. What will happen if you reference a property that doesn't exist in a model directive? Whenever you reference a property that doesn't exist, Angular will create that property for you on the scope.

Validation

The last thing we'll be covering in this module will be validation. Now we are only going to cover the basics of validation in Angular. There is a lot more that can be done then we'll look at here, but this would give us the basic tools that we can use to implement more complex and more full-featured solutions. We'll start by looking at the required attribute in our form fields. We'll discuss how Angular looks at that to determine a form's validity. Then we'll learn about the ngPattern directive, which is the meat of validation using Angular. We'll also look at the properties Angular gives us to tell us about the validity of our form. And finally we'll look at the different CSS classes that Angular puts on our form fields to tell us about their validity, which we can use to manipulate the display of those fields. So let's start by going over to our new event form. Now what we want to do is make sure that the name is required when a user creates a new event. So I'm going to markup with the required attribute. Now my browser is going to honor this attribute, not allow me to submit the form. So if I go over to the browser and leave the event name blank and try to click save, I'm going to get an error. What we can't see is behind-the-scenes Angular is actually marking that field as invalid and that form is invalid. And that's something we could take advantage of, but we'll get to that in just a minute. Now let's talk about the pattern directive. Over in my event date, I also want this to be required, so I'm going to mark that as required as well. but I also want to make sure that users enter data in the pattern of MM-DD-YYYY, so I'm going to use the ngPattern directive. And I'm going to fill this in with a regular expression that will validate that pattern. Now before I can utilize this, I've got to do something else. I've got to go on my controller and I've got to wrap my save event and make sure that the form is valid before I do anything with it. Now I can see that I don't have a new event form and scope, so I'm going to go up and add that as a parameter. Then I'll go back to my HTML. And in order to get that as a parameter on the save event method, I've got to do two things. I need to go to my form and give it a name, and then I got to go down to the method and add that as a parameter. And now that I've got that in place, Angular is going to validate those first two fields for me. Refreshing the page, so I'm going to give name a value, and I'm going to go down to event date and give it a value, but I'm not going to give it the right pattern. Now you can see when I click save, I get no alert. This is because Angular realizes that the form is not valid yet. But if I go on and give it the right pattern, now the save event is passing through that valid check and firing my alert. But if you noticed, there is kind of an issue here. Event data is required but if I fill it at any value that doesn't match the pattern and try to click save, nothing is happening. I'm not getting any kind of feedback. So let's improve the user experience, and let's disable that save button unless the form is valid. So go back to our code, and I'm going to do one thing. Going to add the ngDisabled directive, and in here I'm going to say my new event form is invalid. So that will disable the button as long as the form is invalid. And I'll save that, let's go back to our browser and refresh, so no matter what I do, I can't click on it and I'm getting visual feedback that I can't do anything. So let's go up and give the name field a value, and let's try as putting in invalid pattern in here. And you can see the button has not been enabled. But if I give it a correct value, the instant becomes correct, the save button gets enabled. So that's very helpful. In addition to the invalid property that exists on forms, the form actually tells us a few more things. Let's go on our controller and let's logout that form. As we can see some of the other properties that Angular gives us about that form. All right, looking at the form, there is only few properties I want to pay close attention to. Dirty, which indicates that the form has actually been changed; Invalid, which indicates the form is invalid; Pristine, which is the exact opposite of dirty; Invalid, which is the exact opposite of valid. These four properties will always contain up-to-date values. And so you can use the one that makes most sense for any expressions that you need. Now let's look at one more thing about validation. Angular conveniently sets CSS classes on our input fields and our forms to indicate what state those fields and forms are in, and then we can use that to adjust the display of our fields and forms. So let's look at the new event field. And we can see down here that the classes it has are ng dirty, ng valid and ng valid required. This indicates that it's dirty because the value has been changed. It's still valid and it also has the required attribute. But if I come up here and take out the value, all of a sudden it becomes dirty, invalid and invalid required. Now let's take a little bit of advantage of this knowledge and go back to our code and let's create a style that will help us see which fields are invalid. So as you can see, any input that is both invalid and dirty is going to have a background color of pink. Going back to our form, let's refresh the page and see everything has got a background color of white, but the moment I put an incorrect value in here, the background color goes pink. And that will be true until I give it a correct value. And then the pink goes away. So using these tools, you can create any kind of form validation you need to do with Angular. Now let's review with a quiz. In form validation, what is the opposite property to dirty? The pristine property has the exact opposite value as dirty. Which directive lets you create a regex for validation? That is the ngPattern directive. And what property does a form require in order to check validity? A form must have a name in order for you to check any of the validation properties.

Summary

In this module, we were introduced to Angular's controllers and markup. We started by learning about controllers and how they create our scope object. We talked about how to define an area of our page to be handled by a controller and how to use the ng app directive to tell Angular to process a page for us. Then we looked at how to bind data in our model to render in the page using binding markup. We learned about the built-in directives that Angular provides us. After that we talked about using filters. Filters allow us to modify the output of our model, either by modifying a piece of data, such as upper casing it or by changing a collection either by filtering it or filtering the data somehow. We also learned how to create our own directives in Angular. And then we finished up our module by looking at how to do validation in Angular using the HTML5 required attribute and the Angular pattern directive. We saw the different form properties that Angular gives us to validate a form and the CSS classes it sets to let us style the input fields on our forms. The last thing we are going to do in this module is look at some optional things we can do in the project to solidify the concepts that you learned in this module. First, you could go in and add more comprehensive validation. Another thing you could do is implement the ngCloak directive. You might notice if you are in the project, the refreshing the page really quick will allow you to sometimes see a flash of unparsed binding directives. Use the ngCloak in order to prevent this. And the last thing you could do is use a filter to display an icon for the session level instead of the text that we see. And in addition to these three ideas, you may have come up with some ideas of your own. I highly recommend for you to go back, play around with the concepts that we've learned and solidify them in your mind.

Creating and Using Angular Services

Introduction to Angular Services

Hello, and welcome to this module on AngularJS Services. My name is Jim Cooper, and in this module we'll be talking about how to create custom services in Angular and how to use the built-in services that come with Angular. So first of all, what do we mean when we refer to a service? The term "service" is an overloaded term. Sometimes when people use the term "service", they're referring to an over-the-wire service, such as a web service or a WCF service. Or sometimes they just mean simply an object that is used to encapsulate some sort of business logic, or just does some sort of work for us. With respect to Angular, service refers to the latter description; a service is just a worker object that performs some sort of business logic. It is not accessed over the wire, although it may be used to perform operations that do go over-the-wire, such as making AJAX calls. Also services are often stateless, although it isn't unusual for a service to cache data that is accessed frequently. So, when we talk about services in this module, know that we aren't talking about anything special. Just an object that has methods and properties on it that we can reuse. Angular ships with a whole host of built-in services that are useful for working with Angular. These services make it possible to handle things like navigation, AJAX interactions, and a whole host of other common tasks, but Angular also makes it easy to create your own services, and almost any application you create with Angular will necessitate creating your own services. This module will dig into creating your own services as well as utilizing the built-in Angular services. So why would we want to use services anyhow? Services allow you to encapsulate reusable business logic into an object that can be shared and reused throughout your application, so reusability is one reason. And it also allows you to break your application down into pieces that are easier to maintain. You could put all of your logic inside a controller, but that becomes difficult to manage as your system grows and it violates the single responsibility principle. The single responsibility principle is one of the five solid principles of object-oriented design, and it states that an object should only have a single responsibility. If you'd like to learn more about solid, Wikipedia has a nice write up here. Another reason to use services is that you can then inject them into your controllers and other services that need them. This allows you to have access to the functionality that you need when you need it. Angular behaves like a fully-featured dependency injection container, which makes it easy for it to get access to the services that you need when you need them. And finally, using services makes it easier to test your code because you can isolate your test to just the pieces of code that you're trying to test. This is thanks to dependency injection, again, because when I'm testing a piece of code, I can inject any services that it needs, and instead of injecting the real services inject mocks in its place. AngularJS was actually built from the ground up to be testable, and services are one of the things that have made that possible. An AngularJS Service, whether it's built-in or custom, is really just like the services that we just talked about. It's just a simple worker object and you create it just like any other object, except that when you create an object intended to be used as a service, Angular is unaware of its existence until you register that service with Angular. Registering a service with Angular is simple, and, once registered, it now becomes part of the Angular world and it can be used like any other Angular service. It can now be easily injected into your controllers and directors and filters and even into other services. This really is all it takes to create a custom service in AngularJS. So let's have a quick review. True or false: in Angular, the term "service" refers to an over-the-network service, like a restful web service. This is false because an Angular service is not over-the-network. It's just a local client object, and it may do things like make over-the-network calls to a restful service via AJAX, but it doesn't necessarily do that. Why do we need services? Well, there's a whole host of reasons, but some of the ones that we talked about are: services are reusable, so that you can reuse them throughout your application; and it helps you to write code that adheres to the single responsibility principle; and it helps with testability due to dependency injection. Last question, how is an Angular service different than any other service? It really isn't different except for the fact that it's registered with Angular so that Angular can inject it whenever it's needed.

Creating Your First Custom Service

All right, so we're going to pick up this project where Joe left it, so you can see that we still have an EditEventController and an EventController, and the pages that he created for those controllers. And if I refresh our browser here you can see that we have those pages still functioning here. Let's take a look at the EventController. You see in here that we have all this event data in here, and actually, before we get into that, take a look at the scope parameter that's being passed into the EventController. That actually is an Angular service, and you can see that we're able to use it here in our controller just by injecting it in or by specifying it as a parameter on the EventController; that's how the dependency injection stuff works with Angular. You just specify the services that you want and the parameters of the controller, and it will inject those in. So, let's take this data out of the controller and move it into a service. Eventually, this data will be retrieved from an AJAX request but for now let's just move it out of here into a service so we can see how we create services. So let's start by creating a directory for our services. In that directory, let's create a new JavaScript file for the EventData service. And here we will create a service called eventData. Now, notice that I did not use a $ here, whereas the Angular services use a $. That is a naming convention for built-in Angular services and you should not use the $ on your own services because, if you do, you may end up creating a service that overwrites one of the built-in Angular services, so just create your services without the $. You can see here that I'm creating an event data service, and I'm doing that by calling the factory method on the eventsApp module. Now, remember, Joe created this module previously, and it's called the eventsApp module, and that's a variable that's available on the window scope, so I have access to that here in the services, or in the service that I'm trying to create where I call eventsApp.Factory. What you pass into the factory method is the name of the service, and then a function that returns the object that will become that service. So the name of my service is eventData. Let's come over here and grab this data out of here, and let's leave that with an error for now, and my EventData service is going to have an event property on it, and that event property is going to now have the event data. This is not a very interesting service. Most often, services will have methods on them. Right now, I'm just using this for storing the event data. Now, if I come over here into my controller, I can just inject eventData, and then down here I can set my $scope.event = eventData.event. If I run that now, you'll see I'll get an error. So I'm not getting any data, and if we look at our console you can see that it's saying that there's an unknown provider for the eventDateProvider. The reason why that is that although I created this service, I haven't included it yet in our page, so if we go to the EventDetails page and I register it here, services/EventData, now, if refresh this page, there we go. We have our data back. And so, it is now calling the controller to get the scope.event, and then the controller is getting that data off of the eventData service. And again, if we go over to our eventData service, you remember, we just created it by calling the factory method on the events app, and that's really all there is to creating a service. Remember, earlier I said that an Angular service isn't really any different than any other service or object. The only difference is that it's registered with Angular. And the way we registered it with Angular is this eventsApp.factory call, and that's really all there is to creating services.

Another Custom Service Example

All right, just to get you a little bit more familiar with creating custom services, we're going to go ahead and create one more service. And this is going to be a more traditional service that just has a method on it, and that method is going to do something useful for us. So, for this, we're going to create a new page. We're going to create an EditProfile page so that users can come to this site and create a profile, and it's going to look like this. Now, if you don't have the exercise files, so that you don't have to type in all this HTML, we've made this EditProfile.html file available inside the AngularFundamentalFiles GitHub repo available here. So, inside this DemoApp zip file, you'll see there's an app folder, once you download it, and inside that app folder there is a EditProfileFiles folder, and inside there you'll want to grab the files out of the Original folder. So here, you see the EditProfile.html file, and then there is also an EditProfileController and GravatarUrlBuilder that you want to grab for the upcoming demo, so go ahead and grab those files if you don't have the exercise files already. So in here you'll see HTML and CSS with a few Angular bindings. So, you'll notice we have bindings to a user model, and take particular note of this, user email address. That's what we're going to be working with in our service. And then, notice that we are bound to the EditProfile controller via this ng-controller directive. That controller doesn't exist yet, so let's go ahead and create that. Okay, so now we have an EditProfileController, and that's going to look like this to start out with. Okay, so we just have a basic EditProfileController. It has a user model on the scope. If you go back to the EditProfile page, what we're going to be working with here is this image tag that is currently pointing to the static profile jpg image. And if you don't have the exercise files, that image is available in the GitHub repo. If we go over now to the EditProfile page in our browser, this is what it looks like, so just a page that's got our blank image, and that's what we want to work on is replacing that blank image. So let's go over to our controller, oops, wrong controller. Our EditProfileController, and let's create a function. This is a Gravatar URL method and, again, if you don't have the exercise files you'll want to grab this file from the DemoApp zip file since it has that ugly regex and MD5 hash function in it, so go ahead and grab that now if you don't have the exercise files. So this function takes in an email address, and this is how you build a Gravatar URL, this is the URL to just kind of a generic Gravatar profile image. And what you do is you build up a URL like this, here's the base URL, and then you need an MD5 hash of the email, of the email address, and then .jpg? and then the size of the image, so 200 pixels, and the rating. You can just set ratings on images to epg, pg13, et cetera. So this is the format of the URL, and it does a 75 hash. In addition, before doing all the MD5 hash and actually making a call out to the Gravatar or returning the Gravatar URL, it doesn't do any of that if the emails, regex does not match a valid email address or regex. By the way, this MD5 hash function is much bigger than it looks, you can see that it goes on forever here. But that's a nice, clean way of representing that we're just going to MD5 hash this email. So, that's really all that's going on here, nothing really important particularly with Angular. Just know that we're building a URL from an MD5 hashed email. And if we come over to our EditProfile page now and instead of using the static image we are going to use ng-src instead of just src, and we're going to tell Angular to set the source of the image to the result of this expression, which is going to call that function on the controller that we just created, getGravatarUrl, and that's going to pass in the user email address. And remember, that user email address is bound down here, to the model and to this email input box. So, as we typed the email address in, Angular is going to reevaluate this expression. So, let's go try that out. If you're not familiar with Gravatar, it's a site where you can go and you can create a Gravatar account and you bind it to an email address, and then you setup profile images that you can use based on your email address, and a lot of sites support and use Gravatar. So now, if we refresh our profile page, notice that we're getting just our generic Gravatar image here. If I inspect this, notice my ng-src and my src on my image element now point to that generic Gravatar image. That's what's being returned by that controller function, and if I come in here now and I put in a valid Gravatar email address, angularjsdemo@gmail.com, now I've got my profile image for that account. That's the image that I set up for the profile image for this email address. So now we see that that's working fine, but it's all on the controller right now, right? So this is making our controller a little bit more messy than we'd like to have our controllers, and this isn't really a responsibility of a controller to be formatting a Gravatar URL. So, let's take this and move it out into a service. So we're going to cut this out of here. Let's go create a new service. We'll call this a GravatarUrlBuilder. In here, let's create a new service by calling the factory method on our module. This is going to be the gravatarUrlBuilder. This is now going to return an object, and that object is going to have a buildGravatarUrl method on it. Let's paste in our logic here for building that. This is going to take in an email address. So now, we have this buildGravatarUrl method on our gravatarUrlBuilder. Fix in an email address, or return whatever that returns. And we need to inject this into our controller. So now, if I've done this all right, when we come over here and refresh, and this isn't rendering because I created that service but I forgot to include that service on my page. So now, if I refresh this, there we go. Now we're getting our default Gravatar image, and if I put in my email address, there we go, it's working, and now our controller looks much cleaner and it's just making this call to this other service. So here's another example of creating a custom service and that illustrates how useful they can be.

Introduction to Built-In Angular Services

I mentioned previously that in addition to being able to create your own custom services, Angular comes with a whole host of built-in services, so let's take a look at those built-in services. There are a number of those services that we'll be touching on in detail in this module. Those include $http, $resource, $anchorScroll, $cacheFactory, $compile, $parse, $locale, $timeout, $exceptionHandler, $filter, and $cookieStore. And then there are some other services that are not very commonly-used and we will just talk about those briefly, and those include $interpolate, $log, $rootScope, $window, $document, and $rootElement. And then there are also the $route, $routeParams and $location services which are used in routing and will be touched on in the next module. And then there are the $httpBackend and $controller services which are used in testing and those will be touched on in the testing module.

Using Angular's $http Service

Okay, so let's take a look at how the http service works. Remember, we have the EventController that was calling the EventData service, and the EventData service was just creating an object in line. We talked about how we'd like to actually retrieve that object via AJAX, so let's go ahead and do that using the http service. So we're going to inject the http service into our EventData service and let's go ahead and we're going to rename this to getEvent. So that's going to be a function, and let's get rid of this data. So, we now have an EventData service that has a getEvent function and we're going to make an http call as part of this function, so we're going to call the http service and we're going to pass into it the GET method and the URL of the resource we're seeking. That URL data/event/1, by the way, is not currently pointing to a valid location. Our web server doesn't currently have an end point to serve data at URL; we'll add that shortly. For now, we have to supply a success method, so what happens when the http Ajax call succeeds. That returns on success or returns all that data and then we need to provide an error callback also. So, what do we want to do upon success? Well, because the http call is asynchronous, we can't just return it, so what we need to do is when we call the getEvent function, we need to pass in a success callback that we can call when this succeeds. And let's pass back to that callback the data that we received back from the AJAX call. And then here, in the error, let's just log some data, but instead of just using console.log let me just quickly show you how the log service works from Angular. Log essentially is just used for diagnostic purposes and for writing to the console log. So, in this case, let's write out a warning and we'll pass in the data, status, headers and config. Now, let's come back over to our EventController and let's call the getEvent method, and remember, that takes in a callback to call when it succeeds. We'll just use an in line function for that. And what we'll do when we call that is we will set scope.event equal to the event that's passed back. So now, if we switch over to our browser and refresh, you can see on the console that we're getting a 404 error because our server isn't handling that URL. You can also see that the data that we logged about the error is being output to the console. So let's go update the server to return data to that URL. In the next two clips, we'll show you how to update our express server and our IS server to get and save event data, then we'll come back and show it working in our Angular app. You can choose which of these two clips you want to watch based on whether you are using the node server or IIS.

Updating a Node Server for JSON Requests

Okay, so let's update our node web server to handle the JSON request that our app is now sending to it. Usually, an application will load the data from a database, but for the purposes of this demo we're just going to be loading it straight from disk. We're going to want to handle getRequest to a URL like this, and then we'll want to get that data from our EventsController which we haven't yet created. And we'll want to handle post request too so that we can save EventData from our application. So let's pull in that EventsController that we haven't yet created; we'll create that in a minute. We'll create it at that location. Then we need to use another NPM module called body-parser, so let's go to NPM and install that. Okay, now it's pulled that into our web server. Now we need to use body-parser so that our web server knows how to pull JSON out of the request body and parse it correctly. Okay, now let's go create this EventsController and it's going to have a get and a save method on it. We're going to need nodes file system object to read and write to disk. All right, now let's create our get method. Now we're going to need to read the file from disk and we're going to pull the ID from the request parameters. Then we're just going to append a .json extension to the ID. So you can see now, when a request comes in, we're going to pull the ID off of the request and then we're going to expect the file to exist at app/data/event and then the ID .json. And you can see that I've already created some of those files here. Here's an example. Now, that's a lot of data to type in and you can type all of that in if you wish, but these files are included in the exercise files and if you don't have access to the exercise files, they are in the DemoApp.zip file available on the GitHub repository. I highly recommend pulling them from one of those locations rather than trying to type it in because that's a lot of data to try and type in without any errors. Okay, so now we just need to set the content type and return to JSON. Okay, there. Now we're finding the file on disk, reading it in and then returning it as application/json content type. Now, we're just going to do a similar thing for the save method, only we'll write the file to disk instead of reading it. Okay, there we go. Now, when we try to save an event, we'll be able to save it out to disk. Now, let's go start our server and check it out. Let's refresh this, and there you can see that we're getting our data back. And if we take a look in the Network tab, you can see that we made a request for this /data/event 1 URL and the response that we got back from that was JSON, and that's what we're now displaying here. So, just as a reminder, the way we did that was in our EventData we used the http service to perform a GET against this URL and then return that data and that data we pass into our success callback. That success callback is this function right here, so we called in getEvent, pass in this function and then the EventData is passed back and we put that on our scope and now we have our node server all set up so that it can save and retrieve data from disk.

Updating an IIS Server for JSON Requests

Okay, so now let's update our IIS server to serve the JSON files we want. This is a little more difficult than with node, but it's not too difficult. In order to do this, we're going to open up Visual Studio and create a new Web API project. And we're going to put that right here. This is the root directory where we've been working in with our application and we'll call this application AppData. And we're just going to create a new MVC application and we'll use the Web API template. Let's just rename this default controller that they created and we'll call it EventController. Okay, so now let's create our get method and this is a method that we'll use to retrieve our events. So our method is going to return a JToken, which will help us return the appropriate JSON response and it's going to take in an optional ID parameter. Then we're just going to use that ID to build our filename and read the file from disk and return that using JObject. And again, that JObject.Parse just helps us make sure we're returning a valid JSON response. That isn't usually necessary with Web API because typically we're just returning the actual classes that we're working with and API just transforms those classes into JSON for us, but this is necessary because we're just returning a string that is already in JSON format that we loaded off of disk. Just to see what we're actually returning, let me show you that we already have these JSON files here on disk. So 1.json, 2.json, et cetera, are the files that we're loading based on the ID and this is what the file looks like. You should have this file available if you have the exercise files, and if not, it's in the DemoApp.zip file in the Git Hub repo. I just recommend pulling it in from one of those places rather than trying to type all of this in. Just make sure that you put them in the same directory structure. So, again, you can type all of this in if you wish, but I'd recommend that you pull it from one of those two locations. Then just make sure you put it in this directory. All right, now that we have our get method, let's also write our post method that we'll use for saving our events later. Okay, one last edit, we're just going to want to edit the default route in the Web API config to get rid of this API/ so we don't have to use it in our URLs. Okay, so let's build this and then we need to go over to IS and create the application in our website that points to this. You can see here that we already have our data directory here, which is where all our JSON files are, and we're basically going to mount our new Web API application over that same location, but we're going to leave that directory there for those that are following along in the node server. This is just fine, in IS we'll just leave that directory in place even if we create a new application with the same name. So let's just right click on our website and choose Add Application and we'll name it Data. And then, if you don't have an ASP.NET4.0 application pool, you're going to want to create one over here in application pools, just make sure that when you do that, that you choose the 4.0.net framework. So we're just going to use this one, and then we're going to point that to our new AppData web API application and we created that right here. So we now have that application mounted here and we've built our applications, so let's go over to our browser and check that out. Okay, there now I've refreshed, you can see that we have all of our application data here, and that's pulling from our server. And, again, just as a reminder, that's being sent here from our event controller and we're fetching that from our EventData service using the http service, and now we have that working in a Web API application on the back end.

Using $http Promises

Okay, so we've been using a callback method here on our event data service, but there's actually a better way to do this. You may not have noticed, but we're actually using promises here where we're using the success and error methods. Since the http call returns a promise, we don't really need to pass in a callback method, so I'm going to copy this error function so we can reuse that in a second. Then, let's stop passing in this callback function and we're just going to return the result of this http call and we won't need the log service in here anymore. And then, over here in the controller, we can just call the success and error methods on the result of the getEvent call which is now returning the promise, and I'll just paste in the error function I copied from the EventData service and we'll need the log service here. So, if we go take a look now, you can see this is working the same, but we aren't passing around a callback function, which is a little better. And if the http call fails, we'll see the error log. Let's just cause that by renaming our 1.json file. And then, if we refresh, you can see we're getting our error and we're logging the information here, which is what we were doing in the error function here on the promise. Let's go back and fix that file before we forget. Now, if we refresh, it's working again. So that's how you use the http service promises.

Using Angular's $resource Service

All right, let's take a look at how we use the resource service. This functionality is used for the same sort of things as the http service for making AJAX calls, but it is based on a restful architecture, so it assumes that your web server is using rest-based architecture. So let's replace all of this, for starters, with this. So now we have a resource call, and I'll come back and explain each part of that resource call in detail here in just a minute. And instead of these, we're going to inject resource, and then, let's go over to our Event Controller, and unlike the http service, you actually can bind directly to resources, so let's replace this with this, so we're just going to bind directly to what's being returned by GetEvent. Back over here in our Event Data Service, remember, our GetEvent method is going to return this resource object. Let's go over to the browser and I'll just show you that we're getting an error. Notice that we're getting this resourceProvider error. That's because resource actually is in a different module, so we need to go to our app.js, our module, and include ngResource, and then in our EventDetails HTML we need to include the resource module, which is this one right here, angular-resource.js. Now, if we refresh this, we're not getting any errors. We're now getting that data, but we're getting it from the resource, so let's go back and take a look at what we actually did in our EventData service. So the resource service that we're using takes in a few parameters. The first one is the URL or the format of the URL. And then the second parameter is an object that specifies default values that are used in that route. So we're saying in the route the ID parameter is going to be replaced with the ID of the object being saved or retrieved or whatever. And then we're calling get on that resource. So we set up the resource using those parameters, and then we call get and we're passing in an object and we're saying we want to retrieve the object where the ID is equal to one, and usually it would be passing in that ID to the getEvent and then we would use that parameter, but this is just a simple case to show how this works. So that get works much like a promise, although it's not a promise. So over here in the EventController, it will take a while for that request to be fulfilled, and, once it does, then event will be set to the result of that. And so it kind of works like a promise where we set event to what is immediately returned synchronously from this get, and then when the call finishes, that event object gets populated and displayed on the page. But it's not really a promise, so I can't do a .then like I can with a promise. But the object that's returned from the resource call does have a promise available on it. So you call $promise.then. This isn't necessary when all you want to do is bind to the data, but it does let you handle the data when it is returned so that you could, for example, inspect the data and take some action based on the data returned before binding to it or something like that. So instead of assigning this to the scope directly, let's use this promise, just like we did with our Qpromise in the last clip. So, on success, we will assign the return event to the scope, and then let's just also log it out so that we can see that we can do other things in this success function. And then on failure, we'll log out the failure response. Let's take a look at this now on the page. If we open up the console and refresh, you can see that we did get our log, and notice that it was a resource, so the event that was passed back is actually a resource. That resource, you actually could use that to do gets and saves with, but I don't recommend doing that in the controller anyways, but it does work just like an event object: you can bind to it, you can say scope.event.name and it works just as though it were an event because you can see that it has a name property, but it does also have resource methods on it. And you can see that if an error occurs while retrieving the event that we receive some response data and we're going to log that out. It should just be console. So, to cause an error, let's go out to disk where the JSON files are that we're retrieving. I'm going to rename this 1.json file to 1x. That way, the event that we're retrieving using our resource actually doesn't exist or we'll get a 404. So now, if we go back over to our browser, and open the console and refresh, you can see that we got a 404 error and the data that was logged out from our console.log you can see has error data so it has our 404 error that we received back from the server. It has a status of 404 and has headers and config data, which is the configuration of the resource object that was used to make this request. So you can see that that response data can be useful in determining and responding to different types of errors from the server. And then there's just one last thing that I want to clean up here in our code. Notice that I'm passing two functions into the then function of the promise. There's actually a more semantic way to do that using the catch function, so I'm going to move this down here and our then function will be called here and then we're going to do a .catch and pass this function in. That's a little more semantic way to write this, so that it's more obvious that you have a then and a catch there and the second one is for handling errors. Okay, and that's all I really wanted to show you there. Next, let's take a look at how we can use the resource object to save data.

Saving Data with $resource

So let's come over here to our EventData and let's add a save method on our EventData service. Let's start by moving this resource object up to the top here and let's create a save method. All right, and we haven't been saving events yet, so let's go over to our EditEventController and here, previously when we were saving, we were just calling window.alert. Let's change this now so that it saves the event. Then we'll need to eject the EventData service. Okay, now we just need to add a couple of references to our new Event page. We need to add Angular resource and EventData. Okay, so now we're prepared to save an event. However, there is one thing about saving an event that we need to do still. If we go to the Create Event page, notice there is no event ID that is specified here, but events do have IDs, and actually if we look at them on disk you can see that we're saving them using the ID, so 1.json, 2.json, etc.. So we need to populate an ID and you would do that most likely by-- you know-- having some function that goes out and gets the events and finds the highest ID and then come in here. We won't take time to do that for this demo, so let's just come into our EventData and when we save the event, before we save it, let's just give it an ID, 999. So, coming back to our page now, let's refresh and put this in again, and we got a success response. And if we were to go and change the URL or something to be bad you would see failure here and the failure response. And if we go out and look on disk now, you can see we have a 999.json, so we were able to save our event, if we open this up and look at it, there's our EventData, so that all worked just fine. Now, I did want to clarify that with resources you can use the more semantic then and catch. So we can format this like this, which is a little better. There, that's a little more legible. And there are just a couple of more things to touch on with regard to resources. I've shown you the get and save actions. There are also three other actions that come built-in with a resource surface. There is the query action, which is just like get except that it requires or it expects an array to be returned. And then there are the remove and delete actions. Remove and delete are identical. They just perform a delete action. And then you can also create your own actions by providing another parameter here when you create your resource and that is just an object and that expects the name of your action that you're creating. In this case, I'm calling this one getAll, and then you specify three properties, the method, whether it expects an array or not, and params. So the method in this case is a get and I am telling it to expect an array and then params just allows you to specify some predefined parameters that will be included on the http request, so in this case when the http request is made on this resource it will include the parameter something equals foo. So, that's really all there is to creating your own custom actions. This is not entirely-- You know, it's not incredibly useful, but it does allow you to create actions that conform more to maybe terms that you're used to using if you want to use the term retrieve instead of get and allows you to pass in custom parameters on your http calls. And that's about all there is to using resources. Okay, time for another quick quiz. Which service would be the best fit for accessing a restful web service? That would be the resource service. Resources were created specifically for working with restful data. Which service would be the best fit for accessing a non-restful web service? That would be the http service. The http service is perfect for making raw http calls regardless of the type of end point. In which module does resource reside? It's an ngResource, so you have to include that in your list of modules and you have to reference that JavaScript file.

Using Angular's $anchorScroll Service

Let's look at what the anchor scroll service does for us in Angular. For that, we're going to need an element with an ID on it, and so we'll use this for the ID on this session. This session's in a repeater, so we needed to include the session ID as part of the HTML ID so that it's unique. And then we're going to need a button up here at the top, and it's going to look like this. So it's going to have an ng-click that just calls scrollToSession on it. So let's go over to the Controller and add that method. So notice it's just going to call anchorScroll. I'll inject that in. So when we call this scrollToSession method, it's just going to call anchorScroll. So now, if we refresh this page, we now have the scroll button. Notice, if I click on it now, nothing happens. But if I put this #session3 on my URL and then click it, it's scrolled down to that element with the ID session3. So that's really all anchorScroll does. It takes what's in the hash and it scrolls to an element with that ID. The reason why there's two hashes in the URL here is because we don't have HTML5 routing turned on. We'll talk about that in the module on routing, but that's really all there is to anchorScroll. It just scrolls to an element whose ID matches what's in the hash sign.

Using Angular's $cacheFactory Service

Let's take a look at how the cache factory service works now. In order to demonstrate this, I've created a CacheSampleController, it does not have any members in it yet, and a CacheSample page. And we're going to need to create a service. So instead of creating a service, a file for this, I'm just going to do this in line here in the app.js. So, for this, we're going to create a service, just like we have the other services that we've created, and the name of the service is myCache. But notice that we're taking in a cacheFactory in the function that we used to create this service. And to actually create the service, we call cacheFactory. We give our cache a name and we're going to call the cache myCache and it's going to have a capacity of three. And giving it a capacity of three actually causes it to act like an LRUcache, so it limits the size of the cache to three items. And we don't have to provide that, we can delete this right here and we could create a service just like that but we do want to limit it to three for this sample so we'll leave that there. So let's come over to our new CacheSampleController and we're going to add three methods to this: an addToCache method, a readFromCache method and a getCacheStats method. Okay, so you can see that the addToCache method takes a key and a value and we add that to the cache using the put method and this is how the cache work, it's a set of key value pairs. And then readFromCache does a get given key, and then getCacheStats just returns myCache.info. We'll show you what that returns here in a minute. So I need to inject myCache, which is myCache service. Now, let's go over to the webpage and we're just going to put this HTML in here. So notice that we have added two input boxes, one for entering a key and one for entering a value, and then a button that when we click it calls that addToCache method that we just created our app control and it passes in the key and value that we've provided in the input boxes. And then there's a third input box where we can type in a keyToRead and then in the line right below that on the page we will show the value from the cache for that key, so when we type in keyToRead, that method, readFromCache will be called and it will pass in the keyToRead. This is all just happening with the built-in Angular data binding. And then, Cache Stats is the next line that we're going to show the cache stats which was that method that is just calling the get info method on the cache. So let's take a look at what this look like. So, if we refresh this page-- Okay, there are our fields. Notice, Cache Stats down here is already displaying the info about the cache, so it shows that we have a capacity of three, it has an ID of myCache and a size of zero. So let's add some items to this. So we will add favorite colors for people, so let's say Jack's favorite color is blue. notice, we now have a size of one for our cache. Jill's favorite color is pink, now our size is two. John's favorite color is green, and Jane's favorite color is yellow. Okay notice, when we added that fourth one, that the size didn't change, because it has a max size capacity of three, so which item is missing here? It is not Jane. Notice, when I type in Jane here, we're data binding a field here to show the value of the cache, you'll notice that Jane's favorite color is still there and John's favorite color is still there, and so is Jill. The one that's missing is Jack's because he was the first one to be put in to the cache and so he was the first one to be ejected from the cache when we added more than three items. So that's really all there is to the Cache Factory. We use the Cache Factory to generate a cache and you give it a capacity, and then you add items to the cache by doing a put and a get.

Using Angular's $compile Service

So now let's take a look at the compile service. The compile service is used heavily internally by Angular and it is used whenever a page is loaded, Angular uses the compile service to look through the page for directives and process them. But you can also use the compile service yourself, and typically you'd do that inside a directive, but since we haven't shown you directives yet we'll just quickly show you how you would use the compile service, and we'll do it inside a controller. So for this example I've created a CompileSampleController and this CompileSample page, and you can just copy the cache sample HTML file in the CacheSampleController to make these. That's what I did; just make sure that you update the ng-controller reference and the JavaScript reference to the controller. So we're going to use this HTML here, so we have this appendHere div, that's just an empty div to start with, and then we have these two input boxes, one that's bound to name and one's that bound to markup. And then we have a button that when we click on it, it will call appendDivToElement passing in the markup. So let's go over to our controller now and let's insert this method here. So this is that appendDivToElement that's going to be called by that button on our page and it takes in markup. And then you can see here that we're calling the compile service and we're passing in the markup that was passed into our method. And so we're compiling that markup, and we're doing it within the context of the scope, which is what we're passing in. So you'll notice that the compile method actually returns a linking function, so it actually returns a function and then that function is called and the scope is passed into that function. So we compiled a markup in the context of the scope and then that markup is appended to the appendHere div that we created. So Angular.element that we're using here is similar to JQuery so you can find elements on the page using similar selectors to what you would use in JQuery, and so we're just going to append the results of the compile to that element. So now, if we go over and take a look at this page, it's currently blank, let's refresh that. So there's our two input boxes that we've created. So in here we'll just put somebody's name, John, and then the markup is going to be just an h3 element with an Angular binding to name. And remember this text box, this first text box is bound to name and so we're going to, that's what we're referencing here. Now, this markup is what's going to be passed into the function on our controller that's going to call the compile service, so now, if I click Append, watch what happens, look at that. We got our name written out in an h3 here. Of course, you probably also noticed that we got a console error, that's because what we're doing here is actually quite dangerous. Angular is trying to protect us from opening ourselves up to JavaScript injection attacks. Notice that if we do this again, I can just inject some script into the HTML and it will skew the script. So let's refresh this, do this again. Only this time let's add some JavaScript to it. Now, notice that when I click the button our JavaScript renders; that's really dangerous. So you really shouldn't do this, but this does demonstrate how the compile service works and you might use the compile service and directives. But any time you use the compile service to create an element and then try to render that on your page, you'll get this error, and you should avoid doing that. But, just to continue this example, let's get rid of the script, and let's take a look at one more example. What if we were now to, instead of an h3, pass in the markup button, and it's a button now that has the text of name and now we append that. Okay, so we're actually creating elements here and we're using Angular to compile them, including-- and that will compile even the Angular markup that's in there, so name is getting replaced with John. But notice, that's not just getting replaced with a string. This is actually a binding, so I can type whatever I want in here and that is now bound to those elements that we created. So, going back to our code and our controller, the markup that we were typing into the text box gets passed in and then it compiles it and it appends the compiled element to the appendHere method. So that's kind of the idea of the compile element is that we'll take a bit of HTML and it will compile that and process any directives or bindings inside the HTML and create a function that then, when it is called and given a scope, it will generate HTML.

Using Angular's $parse Service

The parse service is quite similar to the compile service and that's also used internally by Angular when processing a page. Parse service is used to evaluate an expression and turn that expression into a function that can be evaluated against a given context. So, a very simple example here, this does not demonstrate using any context but here we have a very simple expression, 1 + 2, and notice that we're using that parse service to turn that into a function and then we're executing the function and we're just going to log that out. So let's go and take a look at this in a browser. So if I refresh this page now, you can see that we wrote out to the Console the number 3, so this expression was valuated 1 + 2 when we called the function that was returned by parse. Now, let's take a look at something similar but passing in a context, so here we're using parse to parse the expression event.name and I'm assigning that to a variable called getter. Remember, that's a function. And then I'm creating these two contexts. One has a context where there's an event object and the name of the event object is AngularJS Boot Camp and the second one is a context where the name of the event is Code Camp. And then I'm going to call the getter a couple of times, passing in those two different contexts, and we're going to log those out. So, notice, we're going to call the getter, which is the function representing the expression event.name and we're going to pass in the two different events. Let's see what this looks like when we log this out. Okay, so the first one returned AngularJS Boot Camp and the second returned Code Camp. So you can see that passing a context into the function return by a parse takes that context and evaluates the expression with that context. And the function returned by parse can actually take in a couple of parameters. You can take in a context plus a local context and in this case we're passing in context2 and then context1, and context1 represents a local context and anything that's in context1 will take precedence over context2 if there's any overlap. So if I come out and log this, remember context2 says Code Camp for the name and context1 says Angular Boot Camp. And if we refresh this now, we get AngularJS Boot Camp, it's this last one here. And that's because context1 is overriding context2. One last example, I'd like to show that you can also assign values to the contexts, so you do that by, remember this getter variable is the function that was returned by parse, and that has an assign property on it and we're going to set the setter value in our variable to getter.assign. And then we're going to execute that, and that is also a function, so I'm going to execute setter with the context of context2 and I'm going to pass in the value Code Retreat. Context2, remember, is the event with the name Code Camp and I'm going to set that instead to Code Retreat and then I'm going to log out context2.event.name, so I'm going to log out this value, or, excuse me, I'm going to log out this property. So let's run that, and you can see you have set to Code Retreat. So you can use the assign property of a function return by parse to set values on the context that's passed in. And that is about all you can do with the parse function, and again this is something that you typically wouldn't use. If you do use it, you're going to use it inside a directive, most likely, just like the compile service.

Using Angular's $locale Service

The locale service is a fairly straightforward service that's used for localization of date-time and numeric formats. To illustrate that, we're just going to use this page that has an h3 element on it that is binding to myDate and myFormat, and it's using the built-in Angular date filter. So let's go over to the controller that we've created here for the sample and let's create those bindings for myDate and myFormat. Notice that myDate is being set to date.now and myFormat is being set to the full date property of the locale date time format. So I'm injecting the locale service here and I'm using it to get the fullDate format. So, if we go over to our browser and refresh the page, we now have the current date showing in the long date format. If we come over to the controller and change this to a shortDate format, then you can see the format changes. But let's leave that set to full date so that we can better demonstrate the localization. So to change the localization, you need to use one of these localization files from the Angular site available at this URL. So we're going to use the Spanish localization file, so let's highlight everything from here and copy that. And then we're going to come over and add that in here, so let's create our file here. Okay, so we're just going to put it right here, here we go. So now we have that in our project, let's go reference that. Okay, here we go, and if I come over to my browser now and refresh, there, now you can see that we are getting the Spanish localization of that date time format, and that's really all there is to using the locale service. Just to clear up what exactly this date time format's fullDate is, let's log out the locale service and take a look at that. So you can see it's an object, and it has a date time formats property, which is also an object, and it has these different formats on it. And the same thing with number formats, including your currency symbol. So that's kind of the whole purpose of the locale is to provide these formats in a way that will work with the localization files.

Using Angular's $timeout Service

So let's take a look at Angular's timeout service. The timeout service is very much like JavaScript's set timeout with some important differences, which we'll illustrate here. So let's create this h3 element that binds to name, and then let's come over to our controller, and when the controller is first created we're going to use the timeout service to execute this line of code, setting the scope.name = John Doe after three seconds. If we go over to our browser and refresh, you'll see nothing here for three seconds, and then there, the name shows up. We refresh that again and you can see after three seconds the name appears. But you may have noticed that this timeout is returning a promise that we can then use to cancel the timeout. So if we come over to our page again and let's create a button that's going to call cancel on the controller. So let's create that function, cancel, so cancel, when we call it, is just going to call timeout.cancel and it's going to pass in that promise that was created when the timeout was created. So let's go back to our page again and refresh. Now I have that cancel button. If I do nothing, then the name still appears, but if after refreshing I hit cancel before the timeout finishes, then you can see that the timeout never fired, okay? So this works very much like JavaScript's setTimeout method. So let's take a look at why we need a timeout service. To illustrate that, we will need another element on the page, something that will cause the bindings to be reevaluated on our page. So let's just come over here and create an input box and it will have a binding. It really doesn't matter what we're binding to. So if I come back over to our controller and let's change this to instead of doing a using the timeout service, let's just let it use JavaScript's setTimeout function, which had the same syntax. And then we've added this other input box on here just so that we can cause a change in the model that causes Angular to reevaluate the bindings. So if we come over to our page now and refresh, notice that we're sitting here and we're waiting and we don't see the name. Well, that setTimeout has run, but it occurred on a separate thread outside of the Angular world and Angular has no idea that the value of that name binding has changed and that can be demonstrated by causing Angular to reevaluate bindings, which it does whenever a value on the model changes, so if I type in something here, again, remember this is not bound to name, this is just bound to a field called foo, but that did cause Angular to reevaluate the bindings which immediately showed John Doe. And what that shows is that in the controller, setTimeout did run immediately or did run after three seconds and it did set scope.name to John Doe, but it just never showed up on the page because Angular wasn't aware that it happened. And so using Angular's timeout service allows timeout to occur inside the Angular world so that it can watch for changes that occur when that timeout expires, and that's all there is to the timeout service. Okay, here's another quiz. Which service would you use to display localization-based date-time or numeric formats? The answer, of course, is locale. Why would you use timeout versus setTimeout? Because Angular is aware of model changes made by the timeout service. Which service does Angular use internally to process a page and handle directives on the page? A compile service.

Using Angular's $exceptionHandler Service

It's useful to know about the exception handler service for when you want to override the default behavior and handle Angular exceptions yourself, so let's take a look at how to do that. First, we're going to create a service for the ExceptionHandler, so here's that custom ExceptionHandler service we're going to use, and notice that I am using the $ sign in the name of my service, and that is typically not a good idea because you don't want to overwrite Angular services with your own, but in this case we actually do want to overwrite the ExceptionHandler with our own. So what the ExceptionHandler needs is a service that is just a function, essentially, and that function takes in the exception. You can see here we're creating an ExceptionHandler service and when an exception is passed in we are just logging the message of that exception out to the console log. So let's go use the locale sample page for this. So we need to include that service, so we'll just put that there. So we're including our ExceptionHandler service here. Then let's go into the controller for the locale sample and we will just throw an exception here. Okay, so now we're throwing an exception, and the message is error message. So let's go over to our webpage and refresh and you can see that an exception was thrown and our message got logged here and that shows that we are handling the exception using our own custom ExceptionHandler service.

Using Angular's $filter Service

So let's take a look at the filter service. The filter service allows you to get access to any filters that you've created or even the built-in Angular filters, like order by, for example. So remember, Joe created this durations filter that allows you to pass in a value 1 through 4 and it will translate that into a text based duration. Let's utilize that now in our FilterSampleController. So, to get access to that, I'm going to inject the filter service and then I'm going to use that filter service here to get access to my durations service, and then I call that durations service, passing in the values 1 through 4, and then notice that I'm sticking that on the data model on our scope in the values duration 1 through 4. So let's come over to our webpage now and this is the FilterSample page that I created and, again, you can create this just by copying one of the other sample pages. Just remember, in addition to changing the name of the controller and the reference to the controller, that you also, in this case, need to reference this filters.js file. Okay, so here we will output those data bindings to those durations. So let's now go look at our browser, and you can see that we have successfully translated those using our durations filter. Now, one thing that I think important to note is you really don't need to use the filter service and that's because you can instead inject in directly the filters you want. So this filter is called durations, so we're going to inject the durations filter directly like this. You use the name of the filter, followed by the keyword "filter". So now I have that durations filter and I can use that directly without having to use the filter service. I think this is much more often how you will get access to your filters, but it's good to know that that filter service exist and that you can inject directly your own filters.

Using Angular's $cookieStore Service

All right, let's take a look at the cookie store. The cookie store is actually in a different module, Angular module, just like Angular resource is, so we need to reference a new module. So it is the angular-cookies module. And now we need to go download that from Angular, so I'm just going to come out here and get this angular-cookies file, and then let's just add it here. So now we have that reference and we need to go over to our app and also add it there to our module here. Okay, now we're ready to get going with our sample, so I've created this CookieStoreSample page and I've already created the CookieStoreSampleController and referenced it here. So let's go ahead and check this out. We're going to create three buttons for putting things into a cookie, retrieving them and removing them. Okay, so we have a button that's going to save the cookie and it will pass in the event, the event is going to be on the scope and then we will get the cookie and remove the cookie, so we have buttons for all three of those, so let's go create those in the controller. First of all, let's create an event on the scope and then let's wire up those methods. So there is the saveEventToCookie. Notice that it takes an event and then we'd call cookieStore.put and then we give it a key and a value. The key is the cookie name and the value is the cookie value we want to store. So here we are just going to store the event object and we are going to store it in a cookie named event. Okay, then here we're going to retrieve the cookie, so we're just going to call cookieStore.get and pass in the name of the cookie we want to retrieve. And for this demo I'm just going to log that cookie out to the console. And then the last method, the removeEventCookie method, is just going to call cookieStore.remove and remove the event cookie by name. Oops, I named this one wrong. There we go, uppercase that C. Okay, now let's go take a look at it. So, we refresh here and then store the cookie and then remember when we store that cookie, we are storing this event object here, so it's got an ID and a name, and that's what we're storing in the cookie. Now, if I click Get Cookie, we're going to retrieve that cookie from the cookies and display it in the console log, here we go. So there's the object that was retrieved. If I click that a couple of times, you can see I can keep getting the cookie. But if I click remove and then try to get the cookie, now this is undefined. So that's all there is to storing cookies, except that you should be aware there is no way to specify an expiration date currently with cookies in Angular. So if you need to do that, you'll need to use a third party cookie library and you can wrap that in your own service so that you can inject it just like you do the cookie store and they can be injectable and testable. But if you don't need to specify an expiration date for your cookies, you can just use the built-in cookie store. Now, before we move on to the next module, let's just go into our app.js and remove this reference to the ngCookies module. We're not going to be using that in the future modules and we don't want to have that cause problems by having that left over there.

Overview of Less Common Angular Services

Okay, so we've looked at these services in the past several clips, now let's take a brief look at these other services that are not used very often, and we'll start with the interpolate service. The interpolate service is used internally by the compile service in Angular and it is not something you will typically need to use directly but it is useful to know that you can modify the interpolate service. For example, some JavaScript frameworks use the same double squiggly delimiters that Angular does and sometimes this can cause a conflict between the two JavaScript frameworks. And so you can modify the delimiters that are used in Angular expressions by modifying the interpolateProvider. And you can do that when configuring your module and you simply inject the interpolateProvider and then call startSymbol and endSymbol on that provider and give it the characters that you want to use for delimiting Angular expressions, and in this case we're using the double square brackets. So that's really all there is to the interpolate service and it's good to know that you can modify those if you need to. The log service is just used for diagnostics logging and debugging, and it has four methods on it that you can call: log, info, warn and error, and those just provide different types of output to the console log. So, again, this is just used for debugging, typically. Now, the rootScope is used throughout the application, mostly behind the scenes, and there is one rootScope per Angular application and it is used whenever a new scope is generated, so whenever you create a controller, a new scope is created from the rootScope and it prototypaly inherits from the rootScope, which means that you can put properties on the rootScope or methods and those become available to you in all your other scopes, but be careful that your rootScope doesn't become just a dumping ground for global data. The rootScope is injectable into controllers and services, although you won't typically need to use it, and it's kind of a best practice to avoid using the rootScope if you don't need to. There may be cases where you need to use it, but just be aware that if you're using rootScope frequently there might be better ways to do what you're trying to do. All right, the last services that we're going to look at are the three services that give you access to the DOM, and that is the window, document and rootElement services. The window service and document service give you access to what you think they would, the window and document JavaScript objects, and they're provided as a service so that if you do have to use them you can inject them and when you're writing unit test you can inject mocks of them and it doesn't become a problem to use them in your code, doesn't cause problem in your tests. And the rootElement is similar, but it gives you access to the Angular rootElement which is whichever element you put your ng app directive on, that becomes the rootElement. So these services just allow you to have access to those DOM elements but it is really not a good practice to use these. Most often you can get away with not doing any DOM manipulation directly in Angular and instead you manipulate the DOM through data bindings. So if you find yourself needing to use any of these elements, consider whether you could use them through other means that are more consistent with the Angular way. And that pretty much covers all of the built-in services that come with Angular, except for those few that will be covered in the routing and testing modules. All right, one last quiz for this module. Which service would you modify if you want to change the double squiggly expression delimiters? The interpolateProvider, and you do this during the configuration phase of the application. How many root scopes are there in an application? There is just one root scope, although there are many child scopes. Can child scopes access items on the root scope? Yes, this is made possible through prototypal inheritance.

Suggested Exercises

All right, now that you've learned how to use services, here's some recommended exercises for you to try on your own. Try updating the save method of the eventData service to lookup next eventId by finding the highest event ID of all existing events and incrementing by one instead of just setting it to 999. So you may remember that in the eventData service, when we save an event, we just hardcoded for it to set the eventID to 999, which is not going to work for saving more than one event. Then, second, try wiring up the save button on the EditProfile page to save the user using a new service called userData so that save button right now isn't doing anything. So try creating a userData service and use a resource in that service to save them to the URL data/user/:userName. That data user path is critical. The web server that we are using, the embedded web server, is going to require it to be in that format. Store the user's session-upvotes in a cookie and only allow the user to upvote each session once. Or, for a more advanced challenge, allow users to log in after storing their user profile and when they upvote, track their upvotes in their user profile object. Those are a few exercises that will hopefully give you a flavor for creating and using services in Angular.

Angular Routing

Introduction to Angular Routing

Hi, this is Jim Cooper and welcome to this module on Angular JS Routing. In this module, we'll be taking a look at routing in AngularJS, and why routing is important in creating single-page applications. We've already seen some pain points because we haven't introduced routing into our demo application. For example, every page we've created has had to include all of the JavaScript and CSS references for that page. This is because our demo application is not yet a single-page app, and routing is a big part of making that leap. In this module, we'll explore how the routing in AngularJS can help us to transform this app into a single-page app.

Websites of Yore

Okay, let's take a quick trip in the way-back machine and look at a site from the mid-1990s. I came across this 1996 gem the other day. Does anybody remember the 1996 movie Space Jam, starring Bugs Bunny and Michael Jordan? This was the website marketing that movie. So let's take a quick look around. Let's go look at Planet B-Ball. First of all, kudos for the 1990s-style background for the site, not to mention the hard-to-read red text on a busy background and the awesome spinning games gif. Now let's go in and take a look at the Player Bios page. This has an even more awesome background. I mean, who doesn't want a bright, neon green background to their website? But what's even more awesome is the fact that this site uses frames. Here's the main frame. Here's another frame here and another frame here, just for the logo. In case you forgot what those look like, we have a frameset and inside the frameset, we have frames, so this is a fairly indicative site of the mid-1990s, but what's really interesting is that there is little or no JavaScript on this site. Remember that JavaScript had only come out in 1995 and it wasn't very widely used by the time this site came out, and so this is the type of site that we started to become used to as we all started playing around on the internet, and it really was just a site with links that take you to different pages and occasionally a little bit of JavaScript, but typically, that JavaScript was just little snippets, right? When I click on this, I want to open a new window, but as JavaScript became more readily-used in browsers, and as browser technology increased, we began to be able to do some pretty cool stuff with JavaScript, like making Ajax calls, for example, so now we could finally have a page loaded, and we could go out and fetch a piece of data that we needed and then display it on the page without reloading the entire page, but until then, we were kind of stuck with this style of website, and, it's just what we became used to on the web.

Single Page Applications

Now, fast forward about two decades since the release of JavaScript, and the expectations for web apps has changed. We have now begun to expect rich and responsive applications that have functionality in them that we used to only be able to get out of desktop applications. A good example of this is Gmail. The Gmail site is a single-page application. You'll notice that, as I do various things in Gmail, such as changing my themes, notice that it's changing it on the fly, but if I was to close my browser and come back, this setting would be changed, so it's doing saves via Ajax, if you watch the URL, it's changing but the page is not reloading, and as we navigate around the site, you see that it is all one big, single-page application, and it's just making requests behind the scenes. Even though the URL is changing, it's not actually navigating to new pages. Gmail was one of the first really big single-page applications and it has transformed the way we think about the web, and one of the main reasons why is because of that ability to look like you're routing to multiple pages when really you're just navigating around locally within the client and fetching resources behind the scenes as you need them and so, routing is a fairly key part of creating single-page applications.

Adding Your First Angular Route

Okay, let's take a look at adding routing to our application now. This is the same application that we used in the services module, and the only difference is that we have removed some of the sample pages that we were using in that module to demonstrate some of the services, and those pages weren't critical to the application, so we've removed those, and then we've also cleaned up our app JS a little bit. We got rid of the modules that we aren't using anymore like NG Cookies and NG Sanitize. Other than that, this application is the same, so the first thing that we need to do here for adding routing is we need to add a single page that is going to be the page for our application, and that is going to be called index html, and let's copy the html out of the new event page into here, and by the way, index dot html is the document that is returned by default in both the express and IAS web servers that we're using. Okay, so we pasted in that html from the new event page, and we're going to get rid of a lot of this html, so essentially, all we have here now is this navigation bar that will show across the top of the page, and we're going to add a NG view element to our page. This NG view element is where all of the templates will be displayed inside our application as we load different templates in our site. Notice that we also have all our JavaScript references and CSS references in our index html page. Okay, so let's create a new place for our templates. We'll create a templates directory. These templates will essentially be the html that will become our views in our application. Notice I called this folder Templates, not Views, and that's because a view is really the final product that a user sees when a template is combined with a model, and the templates are just the html, so we're going to move our new event html page into the templates folder, and let's come in here, and there's a bunch of html in here that we don't need anymore, so we're going to delete all of this html and this html here at the bottom. Okay, so, notice now that our view does not have all that surrounding html. We no longer have the pain of having to put all of our JavaScript includes and everything on the individual pages so the last thing that we need to do to this template is we do not specify inside our templates the controller that is associated with that template. We do that when we set up the route, so I'm going to remove that here. Now, let's go ahead and set up that route. So, we'd come over to our app JS, and when you create your modules, you can add a config block to the modules, and that config block is run when the application is first bootstrapped by Angular, and this is the function that's called when the application is being bootstrapped and this module is being configured, so we're going to need a route provider, so we'll inject that in, and then, notice that the config block returns a module so that you can keep chaining methods on the module, such as calling a factory. Factory also returns a module, so you could call factory again after calling this factory. We're actually not going to need this MyCache service anymore, and that was left over from the services module, so we're going to go ahead and configure our route here using the route provider. You do that by calling when on the route provider. Essentially, you're saying when the route is new event, then use this information to create the view, and there are a few properties that you can provide here. The first one we're going to provide is the template URL, and we're going to say that the template for the new event route is templates new event html and then we're going to also specify the controller for this route, okay? So here we've successfully configured a route for the new event route, and have indicated that the template that should be used is templates new event html and the controller is edit event controller. Now, there's just one more thing that we need to do here, and as of Angular version 1.2, the routing capabilities are in a different module, so we need to reference that module here, and we'll need to reference that JavaScript file from the html page also. Okay, so now let's go back over to our index page, and notice that our menu bar had a create event link on it, and it was previously pointing to the new event html. Now we want to point it to our route that we just created so our new event route, and we need to provide this hash sign, so the reason why we have to create that hash sign is that Angular is going to take advantage of the fact that browsers allow you to put hashes at the end of the URL and specify additional information after the hash that essentially is for linking within that page. Don't worry too much about the fact that that means that you're going to have hashes on your URLs because we're actually going to show you how you can change that for html five compliant browsers, so that that hash does not show on your URL, so, anyhow, for now, we're going to link using that hash, so we now need to reference that Angular route module. Okay, now let's go take a look at the web page. Here's our application that we've been familiar with, and we're going to go ahead and refresh this by hitting the root of the site, and just to show what actually happened there when I hit that page, let's take a look at Fiddler. Let's go ahead and clear this out and refresh our page and you can see that we requested the root of the site and if we look at the response that was returned, it is our index html page. Notice here's our NG view tag, so that's what got served with the index html page, and we don't have any views specified and so all we got was our blank index html page and if I click on create an event now, then it loads the new event page, but if we switch back to Fiddler, notice that this URL, the root of the website slash new event was never requested. What was requested was template slash new event dot html, so what happened was when we clicked on that link, Angular intercepted that link and noticed that it had a route for it, so it went and requested the templates new event html, and pulled that template in, and it then used that template in order to create this view with the corresponding edit event controller, but I'd like to show you one more thing with Fiddler and that is, if we look at Fiddler, here we see that, where we loaded that template, and if we go back and then click on that link again and then go back to Fiddler, notice that no request was made to the server this time, and that's because when Angular loads a template, it actually stores it in its template cache. There is an internal template cache service that it's using to store those templates, and so, as you navigate back and forth to pages, if you navigate to a page that you've been to already, it doesn't take the hit to go and request that template again from the server. One last thing that I'd like to show, let's go ahead and clear Fiddler again, and let's just go to, I'll copy this URL and then we'll just go to Google, and then, let's navigate back to our site, and we're going to navigate not to the root page, but to this fully-qualified URL pointing to our new event route. And you can see that it did load the new event page, so, what exactly happened here? If we look at Fiddler, you can see that we requested the root page here, and so it loaded the index html, which you can see here, so here's our NG view again, so it loaded our index html, and then you can see that it immediately requested the template for the new event page, and it had to request it again this time because we reloaded the entire app by navigating back to the site again, and so it loaded that template immediately after loading index html and so, when Angular was bootstrapped, it noticed that there was something on the URL that it had a route for and it went ahead and routed to that page, so, that's the basics of adding a route and our application is kind of broken now because the only route that we've set up is the new event page, but in the next few clips, we'll get to fixing those routes.

More Routing and Browser History

Okay, so you've seen how to create a route for the new event page. To make this app a little more interesting and usable, and to further demonstrate routing, let's add another page that lists all of the events. To do that, we're going to have to create a event list template, and the html for that is going to look like this. Nothing really particularly interesting here. We just have a repeater that's repeating for every event and then we just have bindings throughout here to the event, and we'll need a controller to go along with this page so let's create a event list controller and that's going to be very simple. Notice that all it's doing is it's calling event data get all events, and assigning that to the events on the scope, so we're going to have to create this get all events method on event data. We haven't created that yet so let's come over to our event data and create that method. That just looks like this, so notice that we're just going to call query on the resource. Query is basically like get but it expects an array back and notice that we're not sending an ID in like we are with the get method above. This is going to make a request to slash data slash event without passing in an ID, so let's go update our web servers to return all the events if no ID is passed in. Okay, here's our node web server. If you're using IAS, we'll get to that in a moment. So we just want to add a new route like this. And that will call a get all method on the controller, so let's go create that now. That method's going to look like this. You can see here that we're selecting all the files in the directory, and then down here, we're going to loop over all the files in the directory, read their contents, and append it all together into a JSON array, and then, we'll return those results, and that's it for our node server. Now let's go update our web API application for those using an IAS. Here's our web API controller. Now let's just add this method to get all the events. So that's just going to loop over all the JSON files in the directory and append them all together into a JSON array. Now let's abstract our get method into a new method, and finally, let's have our get method call the appropriate method based on whether or not an ID is passed in. Okay, so, if an ID is passed in, we'll return all the events. Otherwise, we'll return a single event. That should do it, now let's build. Now, both our web servers have an end point for returning all results. Since I'm using the node server, I'm going to go stop and restart my server to pick up the new changes. If you're running IAS, just remember to build your solution. Okay, so now both of our servers should be able to handle this resource query, so we already created a template and a controller for our new event list page, and of course, we can't get to that template without a route, so let's go ahead and add that route and that's going to be fairly simple. It looks like this. Okay, so that looks just like the other route that we created, so events is going to use the event list template and the event list controller, so now let's go look at our page. So, if we go to our new route, if we go to events, then notice that we're getting nothing and that's because we're getting an error because we have not yet added the event list controller resource to our index html, so let's go add that. So, we now have an event list controller, and I know that page also uses the filters, so let's go ahead and add that resource. Okay, now if we go and look at the page-- that should render, and there we go, now we have a page that is listening each of the events. Right now, if I click on them, nothing happens because I haven't wired up any links, but, you now have a nice-looking page for listing all of our events, and let's go wire those up so that they link to the event details, so inside the event list page, inside this repeater, we're going to add a link that is going to link to the event route, and we are going to pass into that route andevent ID, so we're just going to use an Angular expression in here, and this whole event thumbnail will be a link. If we were to just demonstrate that right now, you can see that it at least is a link. It's not going to work but if I click on this, you can see that goes to event slash one, so let's go and it's going to look very similar. When the route is event slash and then notice I am putting colon event ID. That's how you specify parameters in a route. We'll get into that in the next clip, and we're going to have a event details template, and event controller, so we already have those, but, the event details template is still a page. Let's move that over so that it's a template, which means we need to get rid of this extra stuff in here, and we don't need the controller specified, okay? So now we've got that page set up and we already have the event controller, we had that before, so now let's go and take a look at our site. Let's refresh this page. Now when I click on one of these, notice I get the event page, but, of course, every time I click on this, I'm not getting the event that I'm clicking on. You might remember in the prior module, we set up the event resource so that it always returns just event one, so in the next clip, we'll show how to wire up this parameter that's being passed in, and use that inside the controller. One last thing that I want to point out that's happening here is as I navigate around each one of these, notice that I'm hitting back, and that's interesting because I'm not actually ever navigating to any new pages, so the browser wouldn't know to be adding things to the history if I hold down my mouse here, you can see that I have several items in the history here, and, as I navigate around to different places, I get more items in the history. So, Angular JS is actually also taking care of managing our histories. It's an important thing in single-page apps for that to work because people are used to when they're in a browser, if they click on something like this and it goes to, what to them looks like a new page, they're going to expect to be able to hit back and go back to the prior page, and where we're not actually loading pages, we have to handle that and Angular handles that for us so that as we navigate from view to view, it adds those views to the history in the browser, so that's a little bit more about routing and history. Next, let's take a look at how to create a default route.

Creating a Default Route

Another thing that's important to know when setting up routing is how to set up a default route, so that if somebody goes to an invalid page on your site or to the root of your site, you can send them to some default page. That's really easy to do when setting up routes. You just use the route provider again, and instead of saying dot when, you're going to say dot otherwise, and then for this route, you can just pass in redirect to and the name of an existing route, so let's just redirect to events. That way, whenever anybody navigates to a page that does not match one of our routes, and it will just redirect to the events route, so let's take a look at that. So now my normal route works as it always has, but if I go to some garbage page on my site, notice it takes me back to events, and if I go to the root of my site, it takes me to events, so, that way you can have a default route set up that handles any exception cases in your routes.

Accessing Parameters from the Route

So in the last clip, we left this events page broken, so we are passing in event ID two here, but it's still loading event one, and that's because the event ID is hardcoded so we're always loading the same event no matter which of these we click on. If we go back over to our code and look at the event controller, we're calling get event on event data that in the event data service that when we're calling get event here, you can see that right here, we are just hard coding event ID one, so let's change that so that we pass in the event ID to the get event function. Okay, so that's fairly straight-forward but what do we pass in here for the event ID? Where are we going to get that from? Remember, that's being passed on the URL if we go back over to the browser and click on one of these, here it is, event ID's being passed in here, and on our route, we're expecting that by specifying this colon event ID here, we are expecting a parameter to come in, and we are giving that parameter the name event ID, but how do we access that? Well that's where the route param service comes in. So if I inject the route param service here, I can easily access those parameters that are being passed on the URL by just simply doing this. Okay, so, because we specified that the name of the parameter in the route is event ID, that's going to get put on the route param service, and can be accessed here using event ID, so, if we go back over to our page, and now if I click on this one, I get the Angular, but if I click on this one I get the code camp event, so that is all now working, and that's really all there is to accessing parameters off the URL. I could come over to this route and specify colon foo, colon bar, and then those could be accessed in our controller using route params dot foo and route params dot bar, and whatever was passed in on that position in the URL would be set to those properties. That's really all there is to handling parameters on routes. Okay, let's take a quick quiz of the things that we've talked about in the last few clips. First question is, That's the route provider service. Second question, That is the otherwise function. And last question, That is the route params service. The only exception to this is in the code where you're actually configuring your routes, the route param service is not available and so you have to use the route service there, but typically, you'll use the route param service.

Using Angular's $route Service

Okay, so we demonstrated how to capture this event ID parameter off the route, and how to pass that in and access it in the controller. I wanted to demonstrate something else that can be useful and that is that you can create your own custom properties on this route such as foo, and I'll set that equal to bar, and that property is also now accessible in the controller, although you access it in a little bit different way, and that's where the route service comes in. The route service allows you to access various things about the route, so let's output that foo property and we do that using route dot current dot foo. So now if we come over to our web page, and open the console and refresh, you can see bar got output, and that was the value that we set to foo on that custom property on our route, so, route dot current can be used to access those custom properties on our route. There's also another way to pass data along and to capute data off the query string. If we delete that here, and let's go over to our web page and add question mark foo equals bar, and refresh, and you can see that that is currently undefined because you access that in a little bit different way, so to access that in our controller, we're going to access route dot current dot params dot foo, and you can see that that is now being output, so whatever I put in the query string up here gets output, and also, on the params you can get access to the route parameters such as event ID. You can see that that is now outputting the value one because that is the value that's being passed in for event ID, and there's another way to access that also. There is the path params property on route dot current, and if I refresh that, notice that I can still get that. Now the difference between route dot current dot path params and just params is notice that, if I try to output foo here, that it's undefined even though it's here on my query string, so, path params only includes parameters that are part of the route, whereas, just the plain params includes everything including the query string. So, one more thing while we're using the route service here that we can demonstrate is the ability to reload a page. Sometimes in a single-page app, you will want to reload a page but you don't want to reload the entire app, so, let's come over to the event details template and let's add a reload button on here, so let's create a button and NG click on this button is going to call reload, okay? So we're going to have to create a reload method on our controller. Let's go ahead and do that. Okay, so now when we call reload on our controller, we're going to call route dot reload. Now let's go over to our page. Let's clear all this out so we're just looking at the event here. Now, let's come in and change that to votes and filter for just a trajectory courses. Now if I wanted to reload the state of this page so that these get reset and everything, and just kind of reset this page back to its default, then I could do that by just doing a refresh on the page, but that would cause the entire app to reload, and I really just want to reload this page, oh, and I'm missing my reload button, so let me refresh and set these back again. Okay, so now I've got this page in a non-default state. I'm going to click reload and notice that it reloaded the page. It set this back to normal, but I was able to do that without reloading the entire application, so in this clip we've demonstrated some advanced things that you can do with the route service, including accessing query string values and custom route properties that you create and the ability to reload a page.

Enabling HTML5 Routing

All right, I've been promising you that I would show you how to get rid of the hash sign in the URL. So let's come over to our module and here, where we're setting up routes in the config, we can tell it to use html five routing, and you do that using the location provider, and you just call html five mode, and pass in true, so, let's go ahead and inject this as a parameter. Okay, so now we are injecting the location provider, and we're using it to specify that html mode should be set to true, and we'll have to go over to our application and edit the routes, so in the index html, we were linking to the new event page using the hash, and then also in our event list, we were linking to our events using the hash. We don't need to do those anymore, so now let's come back over to our page and let's get rid of this hash sign, and we'll just go to our events page. Now you can see here that our app is broken. It seems like our routing is not working at all, but this isn't actually an error from Angular. This is our server telling us that it doesn't know how to find that page. This is basically a 404 error from our server, and the same thing happens in IAS. If we go stop our node server and start our IAS server, then you can see that we get a 404 error there too. So what's actually happening here? Because we're telling our server please send me the events page, but we don't have an events page on our server. All our server knows about is our index dot html document. That events page is a completely manufactured URL inside our Angular app, so whenever you turn on html five routing, you have to take that into consideration on your server. You have to consider that users will be hitting your server with all sorts of URLs, like events or event slash one, and these are all URLs that are just routed within Angular and your server doesn't really know about them, so basically, what we want our server to do is just always return the index dot html document, no matter what URL is requested, so let's go do that. We'll start with our node server. So let's stop our IIS server and then let's go make the change. So inside our node web server, we're just going to add a route like this. So basically, that's going to take any route that isn't routed up here above, and always return index html, and by the way, express will also still be able to serve our JavaScript, CSS, and template files, et cetera, that are stored on the server because those are static files and if express finds a static file that the URL requested, it will always serve that file. Otherwise, it will fall through to these routes. Okay, so this one line is all we need for our node server. Now let's go update our IAS server, which is a little trickier, but it's not too bad. So in our IAS server, we need to create a URL rewrite rule, so let's create a new rule. We'll just start with a blank rule and this is going to be a rule for everything that does not match the pattern that we put in and that pattern looks like this, so basically, we're saying, any files that are not html, CSS, image, JavaScript, et cetera, any files requested that are not these file types, basically we just want to rewrite that to index html. Okay, so we just need to give this rule a name. All right, and let's save that rule. All right, now let's go check that out. Let's restart our node server, and let's go over to our app and refresh. Actually, we want to go to slash events, but either way, you see our app is loading, only, not quite, so notice that we requested slash events, and it loaded our index html document, so our rewrite is working, but Angular is giving us an error here. It's giving us this no base error. Basically, it wants us to add this base tag, so that Angular understands where in the URL the single-page app is mounted, so we just need to go over to our index html page, and here in the head, we just want to add a base tag, and set the href to just slash, and let's go take a look at our app. There, now our app is loading, so that wasn't too much effort to get html five routing working, so just to recap what's happening here, we are sending the URL slash events to our server and our server is returning our index html, no matter what we pass in on the URL. Notice if I hit slash foo, our server still returns index dot html and then our Angular routing kicks in, and remember we created a default route that takes anything that we type in and redirects it to slash events if it doesn't match an existing route, and notice that our links work too. So now our app is working quite nicely and we don't have the hash signs in our URLs. Now, just to clarify what that base tag is doing that we added here, basically, it's telling Angular, in our URL, which is slash events, or local host 8,000 slash events, in our URL, our single-page app is mounted right here. It's very possible that our server could be set up such that our index html is actually served right here and by default, then, that would be here, or with our redirects, here would work. And if that were the case, then we would need something like this in our base href, so basically, this is just telling Angular, when it parses the URLs, where to start parsing from. But we don't need that since our app is just at the root. Now, there is one other way that you could handle that no base error so that you don't have to use this base tag here. I don't think it's particularly useful, but, over here, where we enable html five mode, we can just pass in an object like this. So we enable html five mode, and then we tell Angular not to require the base tag, but a couple of things you should be aware of here is that first of all, this will not work with older browsers like IE 9, and then also, if we do this, Angular is going to get confused at these template URLs and we'll have to prefix them with a slash like this. But we don't want to do this, we're just going to leave that enabled like that, and we'll leave our base tag in here like this. Okay, so that's all there is to enabling html five mode. There is a little bit of work that you have to do on your server to handle that, but, then your URLs look much nicer. So, Angular will allow this for any browsers that support html five routing, and for any older browsers, it will revert back to the hash sign approach. All right, let's take another quiz. You do that by calling reload on the route service. You do that by setting html five mode to true on the location provider. Really, the answer to this question is yes. Even though routing occurs on the client side whenever you navigate from page to page, that's all happening within your single-page app, the first time somebody navigates to your site, if they try to deep link into a route within your single-page app, your server needs to know to serve up the index html page as opposed to trying to locate the page that is represented in the whole URL.

Template and Resolve Properties

All right, there are a couple other properties on a route that I'd like to show you. The first one is really simple. It's just called template as opposed to template URL. So, first of all, going back to our event page, remember, this is what our page currently looks like, and it's getting that html from the template URL. But, what if we just instead supply a template and the template in this case is just going to be hello world, and let's switch back to our browser and refresh. Notice, now, that template is what got output, and so instead of specifying a URL for a template, you can actually just give the template a string, and this could allow you to do some kind of funky things like generate your templates on the fly, for example, or you could pass in a service to this config block and you could use a service to generate your template for you, so that's not something that you will commonly use but it's something to be aware exists. The other thing that I wanted to demonstrate here that might be more common is the resolve property of a route. And to demonstrate that, first of all, let's reload this page, and we now have our event page back, working normally, and now let's go over and take a look at Fiddler. This is going to help us understand why you might want to use the resolve property on a route. So, on the filter tab in Fiddler, I'm going to check this box, break response on content type. So, when we receive a response with the content type of application JSON, Fiddler is going to pause that response, and this will give us an opportunity to see what our page looks like, in the event that our Ajax request takes awhile to respond, so with that checked, let's come over and refresh our page. You can see the page loaded but none of the data loaded because the Ajax request hasn't responded yet, and if our page took a few seconds to respond, this is what the user would see while we were waiting for that response, and that's not very nice, so, if we go over to Fiddler and let that continue, then you can see our data returned and it populated our page correctly. So let's take a look at how we'd use resolve to help with that. So we would need to add a resolve property to our route, and that resolve property is an object and that has members on it, so I'm going to create a member called event, so this is going to eventually contain our event data, and it's going to be a function, and we're going to take in the route service, and the event data service, so we can retrieve our data, and then here, we're just going to return a promise and that promise is going to be the promise that returns our event data. Remember, resources come with promises on them already, so we can just do this. And here, I need to pass in the event ID and that's available on my route already. And then here I just want to return the promise. Okay, so we're going to call get event on our event data, pass in the event ID, that'll return a resource, and then we will return the promise off of that resource, so any member on the resolve object that returns a promise, Angular will wait until that promise is resolved until it actually resolves this route, and tries to display something on the page. Now to use that, we're going to have to go over to our event controller, and we're going to have to replace this with this. Okay? So we're going to use the route service to get that event that we created in our resolve member of our route. So now, if we go take a look at this, when I refresh this page, notice that Angular is sitting here waiting for that data to come back, is actually paused over here in Fiddler, and because we are using resolve instead of showing that page, it actually hasn't re-finished resolving the route yet, and so does not display anything on the page. As soon as I let Fiddler continue, by clicking Go here, it's going to return that Ajax event and that promise will resolve, and Angular will display the page. Here we go. So, you can use the resolve member of a route in order to do that for you so that you don't get partially rendered pages. Now, if you're just getting a little bit of flicker on your page when the page first loads, you don't have to use resolve to fix that. You can use NG cloak if it's just an issue while the page is first rendering, but if you have a page that doesn't look good for a period while data is loading after the page is loaded, then that's what resolve is useful for.

Using Angular's $location Service

Okay, let's take a look at the location service. For that, instead of navigating using an anchor tag, We're going to take a look at how we can use the location service inside our code to navigate, so let's just let this href go to just pound, and then, we're going to use an NG click to call create event on the controller, but this is the index page and it doesn't have a controller. We don't need a controller for the whole index page. Let's just create a controller for our main menu. So--I'll create a main menu controller. I'll need to reference that here, and then we'll need to create a controller, okay? So let's get this controller set up. That's going to take a scope-in as usual, and the location service. Okay, so there's our basic main menu controller, and we're expecting there to be a method called create event, so let's create that. Okay, so now, when we click the create event link, it's going to call create event on the controller, and that is going to call location dot URL and is going to pass in the path slash new event, which matches the route that we created for the new event page, so let's go ahead and try that out. Let's refresh our page here, and now if I click on create event, it still works, and so it navigated me to the new event page, but instead of using a link, we used the location service. There are a few more things to demonstrate about the location, so there's a lot of information on the location object about the current location, or the current URL that you can grab, so let's just output all of those. Okay, so, there's all that information that you can look at on the location object. A whole bunch of information about different parts of the URL. Instead of doing this inside the create event method, let's just do it out here. So, when the main menu is first loaded it will show that. So, let's just go over to our web page now and just refresh this. It doesn't matter which page we're on. Okay, notice that the absolute URL is the entire, fully-qualified URL. The protocol is obviously the http. If I was to change this to https, that would show here. Port is the port we're accessing. Host, obviously, is the host name. Path is what comes after the host name and port, but let's add a little bit more here so that we can see exactly what this includes, so we'll say, let something to the query string that says foo equals bar, and-- event ID equals two and then we'll add a hash. Okay, it looks like I typed something incorrectly when I was coding. It took the output of the hash, so, let's go fix that. That should be hash, not search, and let's refresh. Okay, notice that the path, now, is just slash new event, even though there is more in the URL than that, so, this is everything after the port, not including the search screen or hash. Search is just the query string portion. Our search string portion of the URL, and notice that it is an actual object, so it's nice that it breaks that down for you into properties and values, and hash is whatever comes after the hash sign, and then, just plain URL is the relative URL, so everything after the port, so there's some useful information that you can inspect on the URL itself using the location object. Now, one last thing to remember about these last four, path, search, hash, and URL, those are all read right methods, so, they are both getter and setters, so you can pass in values for those. The rest are getters only, so you can replace just certain parts of the URL, as opposed to replacing the entire URL. Then we have one more thing to show you about the location service, and that is the replace method, so if we come over to our page here, and let's open this in a new tab so we don't have any history. Now you see new tab is the only thing in our history, and if I hit create event, notice that there is something in our history now, and it is the page we just came from, but if we come over here and in this method we say location dot replace, just like that, no parameters or anything, and then come back over to our web page, and again, let's go to events, but we will do it in a new tab again, so notice we have no history, and if I click create event now, we still have no history, and that's because replace basically says replace the current page that I'm on with this new page, not navigate to a new page, so it's a way to switch to another view without adding anything to the history, and that pretty much covers the location service. Okay, one final quiz for this module. That's the resolve property, which expects you to return a promise indicating when it can proceed with loading the view. You use location dot URL and pass in the new route or URL you want to navigate to. You can use location dot search, which returns the query string parameters as an object, also known as search parameters.

Summary

In this module on routing, we talked about single-page applications and the importance of routing in single-page apps. We showed you how to create and navigate routes, and how to access route parameters. We even showed you how to handle slow-loading pages so that views don't load before their data is ready. We showed how to turn on html five routing in Angular and how to inspect the various parts of a URL using the location and route services.

Suggested Exercises

Okay, here are a few routing-related exercises for you to try. First, convert the EditProfile page from a stand-alone page to a template, and then add a route and link to that new route. Second, create a route that requires no template or a controller such as a "site version" page or a simple "about" page. In other words use template instead of templateUrl Third, add a 'resolve' property to the events route so that the page doesn't load until the events have been loaded from disk. And last, match all the monsters in the super high-tech Space Jam matching game.

Creating Custom Angular Directives

Introduction to Angular Directives

Hello, and welcome to this module on AngularJS Directives. My name is Jim Cooper and in this module, we'll be talking about how to use directives to create custom elements and events in Angular. First let's talk about why we need directives. When html was first created, it was never intended to do the things that we do with it today. It was intended to create static documents. But before long, we started trying to use it as a platform for collecting and showing dynamic data. We wanted dynamic documents. And as our needs grew, we started trying to stretch html to do more and more. As our needs evolve, we find ourselves needing html to be more of a programming language than just markup for creating static documents. Evidence of this can be found in web components. If you aren't familiar with web components, there's a draft proposal in the current W3C specification and it allows for creating your own custom html elements with embedded functionality. Considering the following markup from the W3C draft specification, this markup is creating a new custom element called fancybutton. Notice that it has built-in functionality, razzle and dazzle. If you want to learn more about web components, you can read the specification here. The bad news is web components is currently a draft specification and so it is not currently supported in most browsers, a big problem for most sites. But the good news is AngularJS directives give you cross-browser, web component-like functionality today. And especially when you consider its integration with the rest of the AngularJS framework, I actually believe that Angular's directives are more powerful, and once you understand them, easier to write than web components will be. So we have already talked about how you can use directive to create your own custom elements, but you can also use them to create custom events. And actually, in the previous modules, we have already been using some of the built-in event-type directives when we have used things like ng-click, which allows you to wire up the click event to a method on your controller. You can also use directives to observe and react to changes on the model. In this module, we'll show you how to create all three types of directives.

Creating Your First Angular Directive

All right, let's take a look at creating our first sample directive. For that, I've created a SampleDirective template and Controller and an associated route, SampleDirective. So you can see that I can go to SampleDirective and currently, it is blank because our template is blank. Now to create a directive, it's similar to how you create a service. You use the model where you're at, in our case, it's called eventsApp. And instead of calling factory, like you do with a service, we're going to call directive and then give it the name of our directive. I will call my directive mySample. And what you return from the function that you use to create the directive is a directive definition object and there are a number of properties on that. We'll get into all of those properties later. For now, we are just going to show the link function. So our directive definition object is going to have a link method on it. And that link method takes in four parameters: scope, element, attributes and controller, okay? So for our SampleDirective, we're going to simply create an input field and a bit of text to output whatever is typed into that field. So I'm going to create a string here called markup, and it is going to look like this. Okay, so there is our markup. It's just an input box and it is bound to sample data on the model. And then we just output sample data right after the input box. Now to output that on to our page, wherever the directive is, we're going to call angular.element, which is similar to using jQuery to identify an element on the page. In fact it uses jQuery light behind the scenes unless you include jQuery. And we are going to find the element and then set the element's html to our markup. Okay, this isn't going to quite function the way we need it to yet. Let's go take a look at that. So first of all, we need to go over to our html page, to our template, and we need to use our directive, which we'll do by specifying an attribute here that matches the name of our directive. Now notice that our directive name is CamelCased. CamelCase is not an html standard, and so Angular will translate CamelCase to dashes for you. So to use that directive, I am going to create an attribute called my-sample, which matches my sample here, except with a dash instead of CamelCase, okay? So now we need to update our index.html file and add the directives file and our new controller. All right, now let's go take a look at our page. And refresh. Now notice that we've got a problem. So we have our textbox here, and it is binding the sample data but notice the sample data just got output to the screen here. That's because we have not compiled this. You'll remember from the services module that we talked about the compile service and how to compile service is something that typically you'll use in a directive. So what we need to do is use the compile service here and we need to compile this markup like this and passing it in the scope that we received here. We're referencing two of the parameters here that are passed into our link function. The scope in this case is the parent scope, or the scope for the SampleDirective template. But that is not always the case. It's not always the parent scope. Sometimes it is an isolate scope, and we will talk about that here soon. But here, in this case, it is the scope of the SampleDirective page. And then we're using the element here that is passed in. The element is the element on which the directive is being used in the template. So our element in this case is this div element. Okay, so we're going to then compile the markup using this parent scope and then that compiled html is what will be output. Now if we go take a look at that, now if I type something here, notice that it outputs just fine. But what if instead of using a div with an attribute on my sample, what if we just wanted to create an element called my-sample? Well, that is as easy as coming over to the directive and using the restrict property of the directive definition object and setting that to E. By default, this is A, which means it expects the directive to be used as an attribute. Here I am saying I am expecting it to be used as an element. And so my element can be named my-sample, which matches my directive name. And if I refresh the page now, we get the same behavior. Only now I have an element called my-sample. Now one more thing to point out here about what we have done with this directive is that if all you are going to do is replace your directive with some html, then you really do not need all of this complexity. Instead of doing this, you can just set the template property of the directive definition object to the html that you want to use. So notice that we do not have to compile this. If I come over to the page now, we're still going to be functioning the same way. Okay, so by using the template, Angular will compile this for you, and it will just work as you expect it to work. And then one last thing to show here. This restrict can also be C, which is class. If I come over here now and say div class="my-sample" now I could have CSS that is tied to this also and both style this and specify its functionality through a directive at the same time. I refresh this, notice we got the same behavior. Another thing to be aware of is you can set this to M, for comment, and then you can put your directive name inside of the html comment and it will render it wherever that comment is. So that's a brief introduction into creating a sample directive. In the next clip, we'll get into details of creating a much more useful directive for our events application.

Domain Specific Language via Custom Elements

One of the great things about using directives to create custom components is that it allows you to write your html using domain-specific language. In other words, language that matches the domain or business problem that you are working on. For example, instead of having a div that contains five other divs that contains spans, that contains labels and text that you want to display, you can just have a single element that encapsulates all of that. In our event rich application, that might be an event or a session or a user profile. So let's take a look at how we might do that. But before we do that, let's clean up a little bit. We created this SampleDirective, and we did it in this directive's js file. Let's move this out of here and into its own file, just like we did with services. We'll create a directory for our directives, and we will create a mySample directive. And we'll paste that in here. Then we can delete this directive's js file. And over in our index html, instead of referencing directives, we will reference the directive specifically. Okay, so the first directive we're going to create here in our application is the eventThumbnail directive. So if we go back to our site and go to the root of our site, we have these eventThumbnails that display for all of the events in our system. Let's encapsulate one of these in its own directive. So we'll come over to the EventList and here is the html that represents one of these thumbnails. So we're going to take that out of here and we're going to come over here and we're going to create a directive. We'll call this eventThumbnail. Let's go ahead and create the shelf for our event. Okay, now if we look at our sample event and remember how we did, we restrict this to class, in our case, we're going to use an element, and then we had this template. Well, remember that html that we cut out of the EventList page was quite a lot of html. Putting that all in-line here in our template seems really messy. Luckily, there is a solution for that. Just like in creating routes, you can use a templateUrl instead of a template and point to a location where the html resides. So let's go ahead and create that html template that will be called... Let's first create a folder in here for our directives and then we'll create a file, eventThumbnail.html. And we'll paste our html in here. Okay, so this is the html that we pulled out of EventList. And now, it is in eventThumbnail under directives in our templates. So now back to our directive, so we are going to restrict this to elements, and we're going to provide a templateUrl. And that is, /template/directive/eventThumbnail.html. Now we need to go reference that in our EventList page. And we do that just like this. Okay, so now we have our event-thumbnail. Notice how this is domain-specific language now, instead of this just being a div with other divs in it. We now have a thing called event-thumbnail, and when you look at this html, it makes a lot more sense when you start reading html like this that has a whole bunch of these domain-specific elements in them. And we'll have to add this to our index.html. Now let's go take a look at this. I refresh, there we got our thumbnails, but our html now is, if we look at one of these, you can see that even though we just had event-thumbnail here, that it has replaced that with all of the html inside our template, and it is wired it up and everything to the scope. So everything should be working here. If I click on this, then we get our various events. One thing that you may have noticed though when we were looking at the html, is that that event-thumbnail element is still there, which is not valid html. We can tell this directive to replace the element with the html, instead of appending to it. So let's go ahead and do that. It's really easily, we just come over to our directive and we just need to add the replace property and set that to true. Now, if I refresh and we look at the html, now you can see that the element got replaced with the directive html instead of appending to it. Now one thing to be aware of is that in our directive that we created, notice that we're accessing event.name, event.date, event.id, et cetera. This requires that the page that includes this directive have an object called event on its scope. And we will talk about that coming up soon but just remember here that now we kind of have this binding between this directive and our page, which may or may not be a good thing. Sometimes this is okay, but typically this is not what you want to do.

Isolating Directive Scope

Now we've got a problem with the directives that we created, and it didn't seem like a problem when we first created them because they seemed to work fine. But to illustrate this, let's take a look at our SampleDirective. So if we go over to this page and navigate to SampleDirective, remember this directive that we created, whatever you type in here gets echoed out next to it. This seems to work fine, but let's come over to our page and let's actually put three of these on our page. And then come over and refresh. Now you'll expect that if I type in something here, that will echo here. If I type in something else here, that will be displayed here. But watch when I type in Hello World, notice that all of these are linked together. So if I change one, it changes all of them. This is obviously not our intended behavior for this directive. We want our directives to be isolated from each other, and that is where the isolate scope comes in. So to fix this for this directive, it's actually quite easy. All we need to do is create an empty isolate scope, which we can do like this. So when you provide the isolate scope here, you can provide properties as part of the isolate scope that map to attributes on the directive in the html. In this case, we do not need any. This is a very simple directive. We go back to this page and refresh now, now these are nicely isolated from each other. Okay, so that is kind of the basics of isolate scope. I can come over also to my eventThumbnail directive and create an isolate scope on this also. And now if I come back to the root of the site, this is not working and that is because whereas in the SampleDirective, the only bindings was to some things... We're creating, we're binding to it here and we're using it here. In the case of our eventThumbnail, we're not generating our binding within here. We're just binding to an already existing event, and when we created this isolate scope, this scope does not, unlike other scopes in Angular, this one does not inherit from its parent scope. And so we have the scope here, and the scope does not have an event in it and so we cannot bind to it, so that broke this directive. So if we come over to our EventList page, where we use our eventThumbnail, we need to pass in our event. So we'll say event equals, and it's just going to be the event. So now we have an event attribute on our eventThumbnail directive, and we're going to pass in the event here to that attribute. So now let's come back over to our directive and then we expose that on our scope here by saying event is set to, and then we're going to use the equal sign here, event. There are three other symbols that you can adhere too. You can use an @ sign or an ampersand, and we'll get into those next. But here we're just going to use =event and we'll come back over here and refresh. And now that is working. So we can see that when we did not have this, there was no event on the scope because it does not inherit from the parent scope. But as soon as we put this in, basically, we're saying here that I want the event on our scope for our directive to be set to the object that is represented by the event attribute on the directive. Now this does not have to be event, it just has to match whatever is over here, so this could be my-event, and over here, that would translate to CamelCase myEvent, and that should work just fine. Okay, so this just has to match whatever attribute exists over here on the page. And by the way, if this matches, so if we change this back to just event so that this matches, so it's just going to be like this, we actually do not need this event here. We just need the equals only. And that works just fine. So that's kind of a brief introduction to isolate scope. It is a place where you can expose things from the html page through attributes on your directive, and it creates an isolated scope for each directive on the page.

Exploring Isolate Scope Bindings

Okay, we're going to introduce you now to the different ways to bind to attributes when creating an isolate scope. If you remember when we created our eventThumbnails directive that we bound to the event attribute using this equals, and I mentioned that you can also use an @ sign and a & sign. So let's create a little bit more complex directive that we can use to demonstrate that. For that, we're going use this upvote html right here. If we look at the event page, we have this upvote, a little widget right here that we can click on to upvote and downvote sessions. And we're going to turn that into a directive. So we're going to take this html out of here, and let's create an upvote element. We'll create a directive for that. First, let's go create the template. We'll paste that html in here. You can see already that we're not doing things really the way we should because we're going to be binding directly to things on the scope. We'll come back to that. Right, now let's go create the directive itself. Let's just copy the eventThumbnail directive, call it upvote. And we will use the upvote template and let's not use the isolate scope yet. So now, we just need to come over to our index.html and add that directive. Now if we come over to our page and refresh, we still have a functioning upvote element here, but it is being provided by our directive. Of course, we're not really doing things the way we really want to here because we are not passing anything in our attributes here, and everything here is just being bound to the parent scope on the page. So what we want to do is create an isolate scope here, and let's take a look at our html for our directive. There are three pieces here that we are going to need to isolate. There is the upVoteSession and the downVoteSession methods, and there is the upVoteCount. And I want this to be an upvote widget that I can use all over, not just for upvoting sessions, but say I could use them for upvoting question and answers like Stack Overflow or user profiles. So let's go over to our EventDetails.html where we embedded our directive, and we're going to have to pass in three things here: an upVote method, a downVote method and the upVoteCount. So we're passing in the session.upVoteCount off of the session using the count attribute. We're passing in a function call to the downVoteSession method on the controller, passing in the session and passing that in on the downVote attribute. And we are calling the upVoteSession, passing in session and passing that in on the upVote attribute. So now I'm back over to our upvote directive, we can use those here by saying upvote, and we'll use an ampersand for that. Remember, I don't have to specify the name of the parameter because it matches the name of my isolate scope member, upvote. And same thing here, this is going to be a method also, so we'll pass in an ampersand. And then the count is going to be passed in using an equals. Okay, so basically, what we're saying here is we expect the upvote attribute in the html to pass in a function to us that we are going to execute within the scope of the parent scope as opposed to the isolate scope. That's what the ampersand provides you as a way to execute a function in the parent scope as opposed to the isolate scope. Same thing with downvote. And then with count, we're going to be expecting to receive an object in that is the upVoteCount. So now we need to use these in our directive template instead of the values that are there. So up here, we're just going to call upvote, and we do not need to even pass in the session here because we're just going to say call the upvote method on the scope, which is going to call that other method on the parent scope, which actually is passing in the session. And then we can say downvote here. And this is just going to be count. So now let's go look at it. Okay, so now we have our upvote widget here. I can upvote and downvote, and the count is showing here. And the nice thing about this directive now is that notice nowhere in here is a session mentioned. Same thing in the directive here. Nothing is referencing the session or directly referencing the parent scope at all. We are allowing the methods for the upvote and downvote to be passed in and the count to be passed in. So we looked at two things here. Now the ampersand and the equals, what happens if we use an @ sign here for the count? Let's go look and see how that renders. If I refresh that, notice that session.upVoteCount is what actually showed up here. And if we go look at our html, where we're using our directive, that matches this string right here that we're passing in. That's because the @ sign indicates that we are going to expect a string to be passed in, and we're just going to use it like that. Now I could make this work again by passing this in here and if we refresh, now we see we have our count. And this works fine. So that's just another way to pass it in. I like passing it in since it's a number using the equals here and then we're actually getting an integer passed in instead of a string. But you can see that it worked and so sometimes, that @ and the equals are interchangeable. Just know that if you use the @, you have to evaluate it first in your html like we did here using the Angular expression because it's just going to pass in whatever is here. So what actually happened when we were using the @ sign, was it evaluated session.upVote first and then it passed it in as a string. So that is how you use the ampersand and equals and @ signs to pass in values in different ways to your directive. And this all provides an isolated scope that you can use within your directive. Okay, let's take a quick quiz of some of the things we have learned so far about directives. First question, what are the different values you can use for the restrict property of a directive? The possible values are E, A, C and M for element, attribute, class or comment. Next question, what are two ways to specify the html that will be used by your directive? You can use either the template or the templateUrl property. And last question, in an isolate scope, how would you bind a local scope property to a function that you want to execute in the parent scope? In order to do this, you would use the ampersand prefix such as myAttr:'&' which would look for a attribute on your directive called my-attribute, and it would execute the function specified by that attribute in your html.

Handling Events with Directives

So we've shown how to create custom elements using a directive. Let's take a look at how we can use directives to handle events. For this, we're going to create a directive that prevents users from typing invalid keys in this date field. Let's come over and take a look at the markup for this page. So we have the eventDate field here, and we want to create a directive called date-keys. Okay, so this will just be a directive that we can use anywhere we need to ensure that users can only enter keystrokes that are valid for a date. So to do that, let's come over to our directives and create a new directive. This will be called dateKeys. And we won't need a templateUrl, we're actually going to restrict this to attributes because it is going to be an attribute that we'd place on other elements. And we won't need a scope. But what we will need is a link function. It's in this link function where we will bind to the key down event and cancel any keystrokes that are not valid. To do that, I am going to need some helper functions here. Do not worry too much about these, these are not specific to creating the directive. These are just functions that I'll use for verifying that the key codes are valid date key codes. And then here in the directive, we're going to access the element that's passed in, and this is the element on which the directive exists. So I'm going to say element.on('keydown'). So when the keydown event occurs on this element, then we want to call this function. And what we'll do in here is just simply return true if the KeyCode is valid and false if it is not, which will cancel the input. Okay, so if it's a numeric KeyCode or a /KeyCode or a navigation KeyCode, then I'm going to return true. Otherwise, we'll return false. Okay, I need to go over to our index page and reference this directive. All right. So now I can put in a valid date. But I cannot type characters. I'm trying to type letters right now, and it is not allowing it, it's cancelling those. And I can enter, I can use arrow keys, et cetera, because I allowed those keystrokes. So there is an example of using a directive to handle an event. But the nice thing is that it is very expressive, right? So here in our input, I can see here that I am only going to allow date keys, and this is a directive now that I can use anywhere on any input fields. And you may find yourself creating a number of events like these. There are some that are already built into Angular, like ng-click and ng-mouse move. But others like keyup and keydown are missing, and so you can just create your own.

Observing and Responding to Changes

Okay, let's take a look at a directive that observes and reacts to changes on the model. For this, we're going to revisit our editProfile page. And since we have added resources and routing to this app since we first created this page, we need to add those references. Okay. If you remember, the email address on the editProfile page is tied to the Gravatar image. So if I put in my email address here, then the Gravatar image is updated for that email address. Well, let's change this from what it's currently doing into using a directive. And if you remember here, we had just an image tag and we were using ng-src in order to retrieve the Gravatar url from the controller by calling the getGravatarUrl method. And that, in turn, was calling a service that would generate that url for us. So let's change that all to be done with a directive. Basically, what I want to be able to do here is to just say Gravatar. Instead of an image tag, I'll use a Gravatar tag. And I will bind an email attribute to the email address. Okay, so that will be our new element instead of this. Let's go ahead and delete that. And let's go create our directive. And this directive is going to need the gravatarUrlBuilder service. So we're going to inject that in here. Okay, so here is the frame for our directive, and this is an element, so we're going to restrict to element. Our template is going to just simply be an image tag and we're going to replace the Gravatar tag with the image tag, so we're going to set replace to true. Now remember the email address is being passed in as an attribute on the Gravatar directive. So in order to observe that, we're going to use the link function. Remember that the third element on the link function is the attributes. Okay, so that attributes parameter that is passed in has a couple of methods on it. The first one that we're going to use is observe. And we're going to tell it to observe the email attribute. And observe takes two parameters: the name of the attribute to observe and a function to call when that value changes. And that function takes in the new value and the old value. And then in here we're just going to, if that value changes, then we're going to set the source of the image element. Now remember, the image element is going to replace the Gravatar element. And so when we set the source here, we're setting the source on the replaced image element, not the Gravatar element. And we're going to use the gravatarUrlBuilder in order to generate that value for us. And we'll pass in the newValue. Okay, so whenever the value of the email attribute changes, we will set the source attribute of the image element to the URL returned by the gravatarUrlBuilder for the new value of that email attribute. So let's go add this directive to the editProfile page. Remember, this page has not been converted yet to a routed page. So we need to update it here on the page. Okay, now let's go check it out in action. Let's refresh. Type in our email address. There we go, that is working. If we inspect this element, you can see that it is simply an image tag. It has replaced our Gravatar tag, and our source has been set here to our url from our gravatarUrlBuilder. And so that shows you how you can use the attributes passed into the link function to observe changes to the attributes and react to them and then set values on the attributes. And you can do other things here too, of course. You could manipulate the DOM in other ways or take other actions in here if you wanted to, but this is a very common thing to do with attributes to observe changes that occur and react to them in some way.

Using Controllers within Directives

All right, now that you've seen how to create various types of directives, let's look at some other properties that you can set on directives. And the first one that we'll look at is the controller property. Before we do that, let's take another look at the upvote directive that we created and just refresh our memory on how we were calling methods on the parent controller with that. So you'll remember that we had this upvote attribute on our upvote directive, and that was calling upVoteSession on the controller. Now this controller is the controller for this view. So it was the parent controller of the directive. It was not a method inside the directive itself. If we go over to the html for that, you remember that ng-click on the upvote icon would call upvote, and that was actually calling this upvote method on the isolated scope, which was mapped to that upvote attribute on the directive. And that, in turn, was calling upVoteSession on the controller. So that is a great way for your directive to communicate with the controller on the page in which the directive is hosted. But what if you want a self-contained directive that has a controller of its own? That is where the controller property comes in. And for that, I have created this DirectiveControllerSample page, which currently is blank. So if we come over here and look at it, there is nothing currently on that page. You'll see that it references this new directive, greeting.js that I have created. And right now, that does not have anything in it. So that is why nothing was showing on the page. So this is just a shell for a new directive called greeting. So we're going to use a template for this directive. It's going to be very simple. Okay, so it is basically just a button that when you click it, it will call Say Hello. So if we run this now, let's go over to the page here and create the directive in the page, remember that was restricted to elements, so I can use a greeting element. And let's go see what this looks like. Okay, so we have a Say Hello button, which, when we click, of course it does nothing because we haven't (mumbles) this up. Now I could at this point put Say Hello on the controller for this page, but I do not want to do that. I want to demonstrate how to have a controller that is built into the directive so that you can encapsulate functionality of the directive that is tied to the directive's template, that all is self-contained in the directive itself. So let's go back over here to our directive, and let's create a controller. Okay, so now we have a controller inside our directive, and it has a sayHello method on it. So now let's go take a look at what happens over here. If I click on this, now we get that alert that says Hello. So you can see how you can encapsulate logic and a directive-specific model inside of a directive using this controller, and then you can use it in your template. Of course this would work the same if I was using a templateUrl here. But just for simplicity, I am just using a template. Now before we wrap up this discussion on using controllers inside directives, I wanted to point out that there are a few different ways that you can specify the controller inside your directive. One way we've shown here is that you can just define it in line. But sometimes, you want to define your controller outside your directive and specify that you want to use that controller for your directive. So let's take a look at one way to do that. For that, we're going to take this function out of here, and we're going to actually define that controller. Okay, so here I have defined this GreetingController, and it has the sayHello function on it. And I don't have to define this inside this directive file. I could define it just like we do all of our other controllers in its own separate file. But just for simplicity, I'm going to do it here. And then I can specify this controller here by using the controller name. So now I have defined a controller name, GreetingController, and then my directive, I am saying I want to use this controller. And this controller can be defined anywhere. So it could be defined in its own file as I had mentioned. Now if we come over here and refresh this, you can see that this is still working just fine, only now we're using that controller that was defined outside of the directive. But what if I do not want to define the name of the controller here inside my directive? It is feasible that you might want to have the controller that is used with the directive be somewhat dynamic and defined outside of the directive itself. For that, we can specify for the controller, just an @ sign. And then for the controller name, we can use this name property and I'm going to say ctrl. This could be anything. I could say food here. But what this is, this is pointing to the name of an attribute in your html for this directive. So if I come over to our html here, and I say ctrl="GreetingController", so now I'm able to specify in my html what controller I want to be used for this greeting directive. So let's go take a look at this working. There we go, that works just fine too. So just be aware that there are those few different ways that you can specify the controller and define a controller for a directive. Next, we'll show you how directives can communicate with each other using a common controller.

Sharing Directive Controllers via Require

Okay, so we just created this controller that has a Say Hello button, and when you click on it, it calls the sayHello method inside the controller embedded inside the directive. Now let's take a look at how you can have multiple directives communicate with each other using a common controller. So for that, let's create another couple of directives here. Okay, so I have two directives here that I am going to use to say hello in different languages. One in Finnish, and one in Hindi. So to do that, let's come up to our first controller that we created and change the way it works a little bit. We will create a greetings array, and when we alert the greeting, we will alert whatever is in that array. Okay, and then let's add on this controller a way to add different greetings. So now we have this addGreeting method on the scope that when we pass in a greeting, it will just get pushed on to that array. And then when we alert, it is going to join the contents of that array separated by commas and alert that out. Now we need a way for the Finnish directive and the Hindi directive to communicate with that controller. And you do that using the require property. And what you require here is a name of a directive that has a controller in it. So here I am saying, in order to use this Finnish directive, I am going to require that there be another directive called greeting, and that greeting directive needs to have a controller on it. And that will become the shared controller for these directives. So let's do the same thing down here. Okay, now we need these directives to do something. And we're going to go back to using the link function that we have used before. Again, the link function gets called for every instance of this directive. And so when this Finnish directive is hit in the html, it will call up this function. And we've used this link function before, but we've never used this controller attribute of the link function. And so here I can say controller.addGreeting and pass in a new greeting. Hei is Finnish for hello, so the controller that is getting passed into this link function, because we required the greeting directive, it is going to be the controller from that greeting directive. So it will be this controller here. Let's do the same thing down here. And again, here we'll call the addGreeting and pass in ??????, which is Hindi for hello. Okay, so now we have a greeting directive, a Finnish directive and a Hindi directive. Notice that the Finnish and Hindi are attributes, and they are going to call addGreeting on the greeting directive's controller. And that controller still, when we click the button, it will alert out all of the greetings. So let's come over to our page here and notice we have just greeting here. We're not using our other controllers. So if I refresh this, it's still just going to say hello. But now if I add Finnish, it will say hello and hei. And if I add Hindi, it will say, hello, ?????? and hei. And so that is how you can use a shared controller across directives so that directives can work together in the same context of the same controller. Okay, another quick quiz. How would you use a directive to observe changes to an attribute on an element? In the link function, you would use the attrs.$observe function and pass in the name of the attribute that you want to observe. Next question, how would you use a directive to handle an event on an element? In the link function, you would use element.on and specify the name of the event that you want to handle. Last question, what directive property allows you to use the controller from another directive? That is the require property. And you specify the name of the directive that contains the controller you want to share.

Directive Priority and using Terminal

Okay, the way we left our directive, we have our greeting directive and we have Hindi and Finnish. And if I execute this directive when I click on this button, then we get the languages in the expected order. What if I come over however and changed Hindi so that it's after Finnish? We'd normally expect that Hindi would have changed and come after Finnish, but the order that you put the directives on your element really doesn't affect their order of execution. If you want to specify the order of execution for a directive, then you need to set the priority. So let's take a look at how to do that. If I come over to my greeting directives and put our priority on these, let's put a priority one on the Finnish directive, and two on the Hindi directive. And we'll put one on the greeting directive. Now if we come over and take a look at this, then when I click on this, now Finnish is coming first, and that is because the directive priority was a higher priority. Priority one, being higher than priority two with respect to the link function that we're using here. So this link function gets executed before this link function. So you'll notice if I change this to two and this one to one, then we should get Hindi first. And we do. The other thing that I wanted to show you was the terminal property of a directive. What if let's change this back to one and this to two, and what if we want to specify that if I put the Finnish directive on an element, that I do not want any other directives with a lower priority to execute for that element? So I can say terminal: true, and this will make it so no directives with a lower priority will execute. So let's come over and take a look at that over here. So now, watch what happens when I click this button. Notice I am clicking it here, but nothing is happening. And we're not getting any errors or anything, it is just the click directive is not working, and that is because the click directive now has a lower priority than our Finnish directive. And so we've actually disabled this ng-click directive by specifying terminal to true here. And so if you run into this, you can get around it by making your priorities negative. So if I make this -1 and this -1 and this -2, now when I run this, it executes as I would expect. So specifying a terminal property on the Finnish directive caused the Hindi directive not to execute. But now that I put their priorities negative, it didn't interfere with Angular's directives. So that's how you can use the priority and terminal properties of a directive to specify that directives execute in a specific order and that directives of a lower priority do not execute.

Using Require with Nested Directives

Okay, one last thing that I'd like to demonstrate with respect to sharing controllers across directives is what you need to do when those directives are nested within each other. So for that, let's just get rid of our priority in our terminal so those aren't messing with things here. Okay, so our directive should be working normally. There we go, so we're still getting our three different greetings. But what if we wanted to come here and make the greeting directive contain the other directives? So something like this? Okay, so we have greeting and inside that, we have a div with Finnish and Hindi on there. You might expect this to work normally because we still have a greeting and a Finnish and a Hindi directive, and the Finnish and Hindi directives require that there be a greeting directive with a controller, which there is. But what this really is saying when you require a directive, is it is requiring a directive on the same element as this directive. And because greeting is not on the same directive, it's not on the same element here, now when we click on Say Hello, we only get hello. And that is because we need to tell our directive to actually look up and traverse the DOM element upwards until it comes across a parent element that has a greeting directive. And that's what putting the caret in the required property does. It says look upwards until you find a greeting directive. So now if we come back over here and refresh, you might be surprised to learn that this still is not working and that is because we need to use the ngTransclude directive, and that will be covered next. But essentially, I just need to say here transclude: true, and then here on the template, I'm going to have to add a couple of divs. And these divs really aren't related to what I am trying to show you with require. But in order to have a directive nested inside a directive, you have to use transclusion. And I can't do transclusion on a button so I need a couple of divs. And on this div here, I want to add the ng-transclude directive. Okay now, if we refresh the page, now it is working just fine. So we'll cover that transclusion topic next. But just remember that if you have directives nested within directives and you want them to share controllers, that you have to use this caret in front of the name of the directive in your require property.

Understanding Transclusion

In the last clip, I briefly introduced you to transclusion because I needed it in order to demonstrate nesting directives. So let's take a little bit more time now to demonstrate what transclusion is and how you would use it. The word transclusion typically refers to taking a portion of a document and embedding it inside another document. The AngularJS team has used that term here to refer to something similar, and that is taking html and embedding it inside of a directive. And don't let the terminology of transclusion confuse you too much. It really is quite simple and straightforward. So let's take a look at a use case where we might want to do this. Over on our EventDetails page, we have this ng-repeat that is repeating for every session and showing a thumbnail of the session. That looks like this here. So for every session, we get one of these tiles showing the information about the session. Let's make it so that when we click on the title of the session, that it hides the details of the session or toggles the visibility of the session. So in order to do that, let's come over to our page. And in place of this h4, we're going to create a directive called collapsible. And this directive will take in a title. That will be the title of the collapsible element that remains visible, that you can click on to show and hide. So for that, we're going to pass in the session.name. And that is going to encapsulate all the rest of this information about a session. Now let's create our directive. Okay, so here we have the basic shell of our directive here, and we're going to create a template that's going to look like this. Okay, so it is going to have a template that contains a div with an h4 inside of it. And inside that h4, we're going to display the title. And it is going to have an isolate scope. And the only property that we'll need on that scope is the title. Remember, we passed in session.name as the title. All right, let's go add this to our index.html. And let's take a look at what this looks like now. Okay, notice that we're not really getting what we expected at this point because all I have done is I have wrapped the content that was here inside a directive and suddenly, all that information is missing. If we come back over to our html, you can see, we had this collapsible directive, and all of this html suddenly, inside here, is not being displayed. And that is because what we're doing is we're saying in our directive, we're saying replace the collapsible element with this div. And so that element and all the contents inside that element get replaced with this. And that is where transclusion comes in. So we need to tell this directive to use transclusion. And we do that by setting transclude to true. And then in addition to telling it to use transclusion, we also have to specify where in the template the transcluded content should be displayed. So we'll create another div here, and it will have the ng-transclude directive on it. Okay, so now let's go take a look. Okay, now our content is back. So we just need to finish this up by making it so that when we click on this, it shows and hides that content. So let's add an ng-click to this h4 that contains the title. And when they click on that, we will toggle visibility. Okay, so let's create a controller for this. And in here, we need a visibility property on our scope, and that toggleVisibility method. Okay so calling toggleVisibility will just toggle that value and then on the div where the transcluded content is going to be, we will just say ng-show="visible" Okay, so this refers to the visible property on our scope. And then we toggle it by clicking on this here. Let's try that out. Now I can click on this title and show and hide the contents. That is basically how you use transclusion in order to embed content inside of a directive.

Using Compile to Transform the DOM

Okay, the last piece of functionality that we have yet to talk about with respect to directives is the compile function. We've seen the link function used in most of our directives such as the Gravatar directive, where we used the link function to observe changes to the email attribute and respond to those changes. And the compile function is similar, but it is a function that is used to manipulate the DOM prior to the link function executing. So if we go over to this new page that we created, the DirectiveCompileSample, what we're going to create to demonstrate this is a directive that allows you to repeat an element X number of times. So we'll create a div, and we'll call our directive repeat-x. And it will take in the number of times that we want to repeat the element. So to start with, we will just repeat the word, Hello, five times. Okay, I have created already a shell for our repeat-x directive. And before I demonstrate the compile function, I want to demonstrate how you might do this using the link function. So let's create a link function. And then inside our link function, we are going to loop the number of times that was passed in on the repeat-x attribute. Okay, and then every time we loop, we are going to clone our element and append it to the existing element. Okay, and that is all that we would need to do that. However, there is a gotcha here, and that is because we are going to clone an element that has the repeat-x directive on it, the new element will have the repeat-x directive on it, and that directive will get processed and it will clone five new elements and will get an infinite loop. So when we clone this element, we need to modify that attribute. Okay, so the cloned elements will repeat zero times, but they will still have the repeat attribute on them so that they will still get processed as a directive, just repeat zero times. So let's take a look at this in the browser. Okay, there we go, we are repeating the word Hello five times. So if we can do this like this already, why do we need a compile function? Well, let's make this example a little bit more complex. First of all, let's create an input box where we can input the text to be repeated. Okay, and then the text that we repeat will be bound to that text property. And let's see what happens when we do that. So now I can refresh here. And notice that the initial element is working fine, but the cloned elements have our expression in them. And that is because that expression is never getting compiled. If we come over here, we could fix this in our link statement by passing in the compile service and compiling this cloned element before we append it. And for more information on the compile service, you can look at the compile clip in the services module. Let's go take a look at this now. Okay, so that works. So still, you may be wondering why you use the compile service. Well, for one thing, this call to the compile service is a somewhat expensive call. Compiling requires the DOM to be traversed for Angular to look for directives that need to be processed. And that can be expensive. So what we really need is we need to stamp out five clones of the template and then let that all be compiled all at once, and that is the main reason why the Angular team chose to break out the compile and link functions, is for that performance reason, that compiling is a somewhat expensive procedure. So let's change this so that it uses a compile function. So we'll change this link function to compile function. The compile function takes in only the elements and attributes. So it's important to remember that when you are using a compile function, the scope is not available for you because it really is only about making DOM manipulations, not for finding to scope or adding watches. Since all we're going to be doing is duplicating the template, we don't need to compile this here. So we can get rid of that. And now this will just clone the element. So let's take a look at how that worked. Okay, so this is working fine now. What happened was the compile function was executed for this element. It cloned the template and then that template was processed, and any directives or bindings inside there were compiled. Now there is more to the compile function in that you can return from the compile function a link function, and then that link function will be executed for each element. So it is important to remember that the compile function runs once and affects all instances of the directive the same way. And then the link function runs individually for each directive. So let's have our compile function return a linking function. And we need something interesting for this linking function to do. So let's come over to our html, and let's add a text attribute to our element, and that will also contain the text binding. Now let's come over here and create our linking function that will be returned by our compile function. Inside this linking function, we will observe the text attribute. And when its value changes, if it is equal to Hello World, then we'll change the color of the element to red. We do not need this compile service anymore. Now let's go try this out. Okay, you can see that our compile function still works, and as soon as this is set to Hello World, you can see that the linking function on each one of the cloned elements was fired, and because it equaled Hello World, it got set to red. So this is how you use the compile function. And just keep in mind that most often, when you create a directive, you will not create a compile function. Typically, you can take care of kinds of things that you need to do using the linking function and the template to manipulate the html or replace the element with html that you want to use. There's only just a few places, and typically, it is things like this where you're repeating elements. The Angular ng-repeat directive actually uses the compile function in order for it to do its work. But that's not typically something that you'll need, unless you're doing something advanced. Okay, the last quiz for this module, first question, how can you prevent lower priority directives from executing? You set terminal to true on a directive where you have already set a priority. Next question, how do you avoid losing content nested inside a directive? You do that using transclusion. So you set transclude to true on the directive. And inside your directive html, use the ng-transclude attribute to specify where in your directive you want the nested content to appear. Last question, which directive function should be used to manipulate the DOM? That would be the compile function. Compile function is not very often used, but when you do need to directly manipulate the DOM as part of your directive, use the compile function.

Making jQuery More Explicit with Directives

Okay, before we wrap up with directives, there are a couple of things that I would like to point out with regards to AngularJS directives and jQuery. First of all, I would like to demonstrate that you can use jQuery plug-ins alongside AngularJS directives. And in fact, directives can help make your jQuery plug-ins more explicit. And then secondly, I'd like to show how you could create jQuery plugin-like functionality purely with directives if you wanted to. Of course there's no reason to do that, there is no harm in using jQuery plug-ins with your AngularJS implementations, but just to show that it is possible, I'd like to show that. So first, let's take a look at this eventDate field and see what we would do using a pure jQuery approach in order to add a datepicker to this. Okay, so first we need to go grab jQuery UI so we can have the datepicker. And all we want is the Datepicker, so let's uncheck everything here and only select the Datepicker. And we're going to use the UI darkness theme. Okay, so let's download that. I am just going to save that in our application's root directory. And let's go open that up, we'll copy that directory out. Let's put it inside our app directory in the lib directory. So put jQuery UI right here. And then inside here, there are two files that we need to reference in our application: jquery-ui.js file, and the jquery-ui.css file. So let's go reference those. Okay, so over in our index.html file, we want to reference that jQuery JavaScript file, and that's right here, it's in our lib directory where we put our jQuery UI directory. There's this jquery-ui.js file. And then up here, we want to reference the CSS like this. Okay, so now we have jQuery UI downloaded. Let's come over to our NewEvent page and add a datepicker to that eventDate field. So that's just going to be a call like this, where we call datepicker on the eventDate field, and let's go see what that looks like on our page. Okay, so here we have our jQuery datepicker. That is working just fine. But one thing that we can do to make this a little bit more explicit is we can add a directive for this. So let's add a date-picker directive, and we'll use the jQuery plug-in within that directive. So let's remove this here. And let's add a reference to a new directive. We'll call it datepicker. Let's go ahead and add that here. And there is a starting point for our datePicker directive. So over in our page here, we have our date-picker attribute on our eventDate. So this is an attribute style directive. And we were just going to need a really simple link function for this. Okay. So we're just going to call the jQuery datepicker method on the element that the directive is placed on. So let's come over and take a look at how that works. Refresh. And notice that our jQuery datepicker works fine. But now, it is a little bit more clear when we're looking at our html that this element is going to have a datepicker. And so directives can be used to just make jQuery plug-ins more obvious inside the html that they are being applied to an element. And now I'd like to just quickly demonstrate that we could do this without jQuery at all, if we wanted to. Again, there's no need to do that, but just to demonstrate that you can do the same sort of functionality purely with directives. Let's remove our reference to jQuery. Let's still reference our datePicker directive, but this datePicker directive is going to be a little different. It's also going to use a service called calendarHelper, and you can see I've already added that service here. So don't worry too much about what is going on in here, this is just calendar-related functionality that I have created and extracted into this calendarHelper service. And this file is available inside the exercise files and also in the demo (mumbles) in the Github repository, if you don't have access to the exercise files. I'd recommend that you go grab that rather than trying to type all this in. And now let's replace this datePicker directive with a new one. And again, I'd recommend grabbing this file from the exercise files or the Github repository. You really do not need to do this if you do not want to follow along with this part. It is not an important part of the application. I just wanted to show that you could do something like this with a directive, and that is fairly involved, there's a number of methods that we need on our directive that will be called from our webpage. Okay, so this is now going to be an event-style directive. Let's come over to our html, and we'll demonstrate what that is going to look like. So here we're going to have a date-picker element, and it is going to have an attribute for, where you specify the element for which this datepicker is being created. Or you can use this datepicker with our event date. And notice that this is using a templateUrl for the html for this directive. So let's go ahead and create that here. And this file is also available if you want to download it. Okay, a couple of key things to note here. Notice that it is only going to be displayed if showDatePicker on the scope is true. And notice that we have this ng-repeat over that days property on the scope. Then we have these calendar-nav classes for navigation. So just a couple of things to be aware of as we start working with this directive. Now if we look more closely at the directive itself, you can see that it has a controller. There's a lot of things going on in here but just be aware, a lot of it is just calendar-related stuff. So when navigating next month or previous month or next year or previous year, this is just being used along with the calendarHelper to refresh the display of the calendars, show the correct days. And then here is our showDatePicker property on our scope, defaulting to false. So it'll be hidden by default. And a selectDate function on our scope. Again, this is all inside our controller. And when we select the date, then we are going to set the value of the element to the value selected. Now this element got set down here in our link function. So in our link function, we are finding the element that was specified in the for attribute. Remember that in our html, we specified in our for attribute the element for which we are creating this datepicker. So this finds that element in the DOM and it sets that element on the scope so that is what this element is here. Then we're just adding some event handlers for showing and hiding the datepicker at appropriate times. So one more change that we need to make over here, this is going to be displayed relative to the other element. Okay. Now I just need to add a few styles to our app CSS, like this, this is just to format our new calendar. And these are available in the exercise files also, if you want to download those instead of typing them in here. And let's go take a look at this over here. So now notice this is my own custom datepicker control. This is not the jQuery UI control. In fact, we removed the reference to jQuery UI. But notice that this works just fine. So this just demonstrates that you can use jQuery with AngularJS directives, and you can also create your own jQuery-like functionality using purely directives alone.

Summary

So in this module, I have demonstrated how to use directives to create your own custom elements, handle events, observe and react to changes and even how to use directives to make using jQuery plug-ins more explicit in your html. So here are a few exercises that you can try that will help you practice some of the things that you have learned about directives. First, create a "focus" directive, which when placed on an input element causes that element to receive the focus when a page first loads. Next, create a directive that allows the save button to be clicked whenever the Enter key is pressed on the newEvent page. Third, replace the event and session html on the eventDetails page with custom "eventDetails" and "sessionThumbnails" directives. And finally, on the NewEvent.html page, combine the eventDate input element and the date-picker directive that we just created into a single date-input directive.

Testing Angular

Introduction

In this module, we're going to learn how to test our code with Angular. In Angular, we do our testing with Karma. We can test our code without Karma, but Karma is such a great utility, that we're definitely going to want to leverage it when we're testing our code. If you're interested in learning more about testing with tools other than Karma, my course on testing JavaScript covers the main testing libraries that are used when testing JavaScript, which are Jasmine, QUnit, and Mocha. Angular by default uses Jasmine, although, you can use any testing library. But, Jasmine happens to be better supported, so we're going to use Jasmine in this module. It's important to node that in this module, I'm going to assume that you already know how to use Jasmine. If you don't, again I'll refer you to my course on testing clientside JavaScript. I have an entire module on Jasmine. I highly recommend you go watch it. In addition to that, we'll be covering mocking. In that same course, I have an entire module on the sign on mocking library. That would be another good module to watch if you have no idea what sign on is and how to use it. After we talk about Karma, we're going to talk about how to unit test. We're going to start by looking at how to unit test controllers. Then, we'll look at how to unit test services, then filters. Then finally, we'll wrap up our unit testing by looking at how to unit test directives. The last thing we're going to look at is how to do end to end testing with Angular. End to end tests is a very unique and very powerful feature of Angular that allows you to write tests that exercise your application in a production environment. You can essentially automate the browser and go through scenarios that utilize your actual application, just like a user would, and make sure the application works the right way. These kinds of tests can be very valuable, but they can also be very brittle because they might break easily when you change your HTML. So, be careful when writing your end to end tests, that you don't make them too brittle by tying them very closely to your HTML. Because we'll be using some tools that are external to Angular, now is a good time to check the GitHub Repo for this course, and make sure that the course is up to date.

Installing Karma

In this module, we're going to be using Karma as our test runner. Karma is a testing automation tool built by the Angular team at Google. It was formerly known as Testacular, but has been renamed as Karma. So when you're looking for information for or about Karma, you should keep in mind that it used to be called Testacular. Karma is completely independent of Angular. And you can test Angular without Karma. And you can use Karma to test other types of JavaScript, and not just Angular. Karma is also extremely fast, especially when running against Chrome. We're going to briefly see how to install Karma. If you have any trouble following along with this part, my course on testing clientside JavaScript takes a deeper look at Karma. You can go through an in depth tutorial on how to install Karma in this course, in the module on testing utilities in the third clip, which is all about Karma. Let's just quickly look at installing Karma, and a couple of other necessary pieces. I've opened up the command line, and we're going to install Karma using NPM. There are three modules I need to install. Karma itself, Karma Chrome Launcher, which is a module that lets Karma launch and run its tests inside of Chrome, and finally, Karma Jasmine. Now depending on your version of NPM, you may also need to install Jasmine Core. It's installed automatically with Karma Jasmine, but newer versions of NPM potentially may not install automatically. This module allows Karma to run tests using the Jasmine unit testing framework. I've installed those modules locally, but there's another module that I need to have installed globally in order to run Karma from the command line, and that is the Karma command line interface. So again, I'll install using NPM, but I'm going to install the module Karma-cli, which is the Karma command line interface, and I'm going to install it globally using the dash G flag. Since I've already installed the Karma-cli on my machine, I'm not going to execute this command at this time. But if you don't have it installed, you'll need to install this module in order to be able to execute Karma from the command line. Installing Karma is only the first step. The next thing you need to do is give Karma a configuration file. The project that you downloaded at the beginning of this course includes a configuration file that's set up for our project. But we can also generate a Karma configuration file by hand using the Karma init command. This is going to walk me through a series of questions that will create a Karma configuration file for me. The first question is which framework I want to use. I'm going to accept the default answer of Jasmine. Next, it asks me if I want to use required JS. Now if you're following along, the questions that you encounter may or may not be the questions that you'll see here. As new versions of Karma are released, they sometimes change the questions that are asked. Don't worry, usually the default answers are what you want. And reading the questions carefully will usually clue you in as to how you should answer them. So I set the default for this question. This is an important question about which browsers I want to capture. I definitely want Chrome, but I can certainly add more by pressing Tab, and selecting other browsers. For example, if I wanted Fire Fox, I would then hit Enter at this point. For now, let's just stick with Chrome. The next question is which files do I want to include. I'm just going to give it a simple answer of all of my JS files, which would be \*\*/\*.JS and that will be the only pattern that I'll put in. Then, I'm asked if I want to exclude any files. I'm going to skip this. And it asks me if I want to watch the files and rerun whenever they change. I'll select yes. And now, the process is complete and Karma has generated a configuration file for me. Now instead of examining the configuration file that we just created, instead, let's examine the configuration file that's in the demo project that you downloaded in the beginning of this course. Here's the configuration file that's included with the sample project. Let's go through each of these pieces and go through them one by one. We start off with a base path. This is the path that relative to the configuration file, is the root for all the other paths inside of this file. Next, we have a collection of files. This is an array of strings. It's very important here to put you files into the proper order. For example, you'll see the Angulars listed first. That is because it is a dependency for just about every other file listed in my files list. After Angular, I include all the Angular modules, which in our case is going to be Angular Route and Angular Resource. Then we've got the Angular Mocks file. If you are updated to the very, very latest version of Angular, you'll also need to update this file as well. Then we've got the sign on testing library. You can see that we show 1.10 here in the video, but this has actually been updated to the latest version on the demo that you'll download from GitHub. After that, I include all of the JavaScript source files that we created earlier in the course. And then finally, is all the files that include our unit tests. The next setting, auto watch, tells Karma that we want to watch the files for changes and rerun the tests whenever they change. After that is our testing framework. In this case, we're using a Jasmine testing framework. Next, is a list of browsers that we want Karma to run our tests in. Here, I've just listed Chrome, but I can easily add more browsers, such as Fire Fox, or Chrome Canary, or IE. After that is a list of plug ins for Karma to load, and you can see I've listed the Karma Chrome Launcher and Karma Jasmine, which we installed earlier. And then finally, some output settings. And there's the Karma configuration file that we'll be using for this course. The final piece we're going to show is how to run Karma from the command line. Running Karma from the command line is very simple. You simply run Karma, start, and then the path to the config file. In this case, the path is at test Karma.conf.JS. And then I'll hit enter, and Karma will launch the Chrome browser and run my tests. Now, I've actually loaded up some sample tests. If you're following along and execute this command at this point, you'll actually execute zero tests, because we haven't created any unit tests yet. And now, I want to terminate Karma, so I'll hit Control+C, and Karma will exit. So, there's how to install and run Karma in order to run your unit tests.

Karma with Webstorm

In this section, we're going to show how to use Karma inside of WebStorm. Now if you're not using WebStorm, you can go ahead and skip this part, but you may want to follow along just to see what you're missing out on. The integration between Karma and WebStorm is really slick, and can be really convenient when building an application. The first thing I need to do is tell WebStorm how to run my Karma. In order to do that, I'm going to go up here to this button, click it, and select Edit Configurations. These are run configurations that allow WebStorm to run external programs. After I select that, I'm going to click the green plus button. And now, I get to look at a list of possible types of run configurations I want to create. Of course, I'm going to go down here and select Karma. At this point, I see a few things. First, is the name of my configuration. Unnamed doesn't work very well, so I'm going to rename that. This is also asking me where my node interpreter is at, and it's filled it in with what it thinks is the correct answer. In this case, this is correct. This is where my node interpreter lives. On your box, it might be different. The next piece of information that needs to be filled in, is where Karma is at. If you followed along and installed Karma the way that I did, then this is what should show up, and that's going to be the correct path. If you installed Karma globally or in some other location, you may need to help WebStorm by pointing it at the right path. The last piece of information that we have to fill in, is the configuration file. This is going to be our karma.conf.js file. This is very simple and just click Browse, select the file itself, and then hit OK. And now, my configuration is set up correctly, so I can just hit OK, and I'm ready to run. Now that we've got it configured, let's go ahead and run it and see what happens. So you can see after a brief pause, Karma has launched and ran all of my tests. We're getting a few pieces of information that's very interesting. First off, is this little message here that says if we want to rerun our tests, we can just hit Alt+Shift+R. I'm going to click Got It on that. Up here at the top of the window that pops up, you can see that it says, done nine of nine. That's indicated that nine of nine tests have run. The green bar across the right hand side is a progress bar that will slowly fill in as Karma runs my tests. In the main section here, on the right hand side, we've got the log output, and on the left is a tree that shows me which tests have run, and whether they've passed or failed. Let's go ahead and look at this. First, you can see that the top node is the test results. Then, it shows my Karma configuration file. Underneath that, it's going to show a node for each browser. This way, if you're running your tests in multiple browsers, and some browsers are failing and some are not, you can see that very easily. Let's pop open Chrome, and we can see that there are several different test suites. These are the test suites that we're going to be building inside of this section of the course. I've already installed them, so that you can see what it looks like when you're running multiple tests. Inside of a test suite, if you open it up, it will show you each specific test an whether or not that test passed or failed. In addition to that, we have a few tool buttons across the top that will give us some nice features. For example, we can hide our past tests, and look at only the failing tests. We can sort our tests alphabetically. We can also expand all of our tests, or collapse all of our tests. And then using these two buttons, we can quickly navigate between each failing test. There's another tab here I want to show you and that is the Karma server itself. This is the output of the Karma server, and this allows you to stop the entire Karma server using this button, so that you can restart it in case your configuration has changed. In general, WebStorm will detect when your configuration file has changed and relaunch the Karma server for you, so you'll rarely have to do this yourself. Now, let's go back to our Test Run tab, and I want to show you a couple more things. One interesting feature that WebStorm has is the ability to look at statistical information about your test run. You can right-click on your test and select Show Statistics. At this point, another window pops up that shows you statistics about the tests. You can see here that it's taken 56 milliseconds, and nine tests have passed. I can select a subsection in my tests and do the same thing, and it will show me the information for just that subsection of tests. Should I want to turn that Statistics Panel off, there's a tool button for that. But, I need to expand this to show all of my tool buttons. Right here is a Settings tool button. If I click that, I get a little drop down. I can turn off Show Statistics. And there's also a couple of other interesting features there. If I want, I can select the first failed test when the run is finished. I can tell it to scroll to my Stack Trace. And I can also set the Auto Test Delay. By default, WebStorm turns off auto running of tests, so whenever you change your code, Karma will instantly rerun. Instead, it waits for to hit the keystroke that will rerun the tests. If you actually want Karma to automatically rerun when your code changes, you can come over here and select this tool button right here, Toggle Auto Test, and that will turn auto testing on. Then inside of here, you can configure the delay that you want WebStorm to wait between running your tests. Whether you prefer to use the keystroke or have WebStorm auto rerun your tests for you, is up to you. And that will conclude our look at using Karma inside of WebStorm.

Testing Controllers in Angular

In this first section, we're going to learn how to test controllers. Since controllers are the heart of any Angular application, it's only natural that we learn to test them first. We're going to start by learning to test our event list controller. This is a great starting point because it's an extremely simple controller. All it does is one thing on line five, which is call eventdata.getAllEvents, and then assign it to the events properties on the scope. So, let's create a test that verifies that whatever comes out of the getAllEvents call, gets put onto the events property on the scope. I've got a blank test file here. I've named it Event List Controller Spec, and I put it inside of the unit folder inside of my test folder. This will contain the tests for our event list controller. I'm going to start out by calling the describe function. This is the way, in Jasmine, we group our tests. This function takes in two parameters. The first is a title or name. And I'm going to name it event list controller. And the second is a call back function. Inside of this describe function, is where I'm going to actually write my tests. Any tests we write will go inside of an it function, and that takes two parameters, the name of the test, which is a string so that we can put in spaces. And I'm going to name this test should set the scope events to the result of event data .getAllEvents. Now, I know that's a really long name, but it's nice to have descriptive names for your tests. The second parameter to the it function is just like the describe function. It's a call back. Inside of here, is where we'll do our actual testing. Now ultimately, what we need is a handle to the scope that the controller creates. Going back to look at the event list controller, we can see that it receives in this scope as a parameter, and it sets the events property on that scope. So in our test, ultimately we'll need to have the scope and check its events property. When we check something in a Jasmine test to make sure that it's the way that we want, we use a function called expect. And we're going to expect that our scope.events is equal to something. We'll use the to be function and we'll check to make sure it's equal to a specific mock events object that we'll create. So, I'll call this mock events. Now, I've got to create this mock events object, so I'll start by creating a variable for it, and set that equal to an empty object. And the next step will be to get the events list controller to actually initialize that scope for us. Now, this is going to take a few steps. First off, the event list controller actually lives inside of the Events App module. So, we're going to have to load the module up. In order to do that, we're going to go up inside of our describe, and we're going to call the before each function. The before each function is a function that will run before each test. This function is where we want to initialize our module, and we do that by calling the module function. The module function is not specific to Jasmine, it's specific to Angular testing. Because in our tests, we loaded up the Angular mocks library using Karma, the module function is available for us to call. This takes in one parameter, which is the name of our module. And since our module is Events App, we just simply feed in that name, and now the module has been loaded up, and now I have access to the event list controller. So now, we need to get a handle the event list controller. But in order to do that, we'll need a special service from Angular called the Controller Constructor. In order to get a handle to that service, we're going to use another before each function, and this one we're going to call the inject function. The inject function is another special function inside of Angular. This function takes in a call back. And this call back is an injectible call back. So any parameters that we pass, Angular's going to try to look them up in its registry and find matching services for that. So, we want the Controller Constructor, and that's available as $controller, and we'll assign that a variable called controller constructor. And since we need access outside of this before each function, we'll create it as a variable up here. Once we have that service, we can now construct our controller, and doing so will initialize the scope. So, we go down into our test, and we're going to create a variable called controller, and assign it equal to the value of that special service. And that takes in two parameters, the name of the controller itself, which is event list controller. And the second parameter is a list of all the parameters that go into the event list controller. So if you go back to the event list controller, we can see that there's three parameters, $scope, $location, and event data. Of course, right off the bat we should notice that $location is completely superfluous, and isn't used by the controller. So, let's remove that. So now, we need to just pass in the scope and the event data service. So first, let's tackle the scope. We can't just pass in an empty object here because scopes are special things. Instead, what we need is a blank scope, that our controller can fill out. So again, we're going to need a special service to create a blank scope. We'll go up here into our before each, and we'll ask for the root scope service. Once we have that service, we can create a new scope by calling rootScope.$new. And let's assign that to a scope variable, and we'll declare that up here. And we indicate the parameters to this controller constructor by putting them inside of an object with keys indicating the name of the parameter. So again, back to our controller. We can see the first parameter is $scope. So, our first key in our object will be $scope and the value of that will be the scope object that we created on line 10. The second thing we need to pass in is event data. Now again, we can't just pass in a blank object because inside of the controller, we call eventData.getAllEvents there on line five. So what we need is an object that has a getAllEvents method. What we don't want to do is create a real event data service, because now we'd be testing two different things. And if there is a bug in the event data service, it would show up in our tests for this controller. We want to isolate our tests, because they're unit tests, and only test one thing at a time. So instead of trying to use the real event data service, instead, let's create a mock event data service. So, let's go up to the top and we'll create a new variable called mockEventData, and inside of our before each, we'll set that variable equal to a call to sign on. This is where that sign on library comes in handy. If we call signon.stub and pass in a template object, and again we want to match the signature of the event data service, at least as far as what the event list controller calls. And all it knows, is it has a getAllEvents method. So, let's create a blank events object with a getAllEvents method, and we can just create a blank function there. And, I need to spell that right, I getAllEvents. And now what sign on, we'll do, is take this template, and create an object that looks just like this, but it gives us control over how that object functions. And, we're going to use that functionality in our test. So now, we can go back down onto line 18, and for event data, we're going to pass in our mock event data, and remember, our expectation is that scope.events is the mock events object that we created on line 15. So, in order for that to work, what we need is for our mock event data service to return the mock events object when getAllEvents is called. So now what we need to do is tell our mock event data service to return the mock events object when getAllEvents is called. And we do that using the functionality of sign on, where we can tell it mockEventData.getAllEvents, which is a function, but because this is sign on, this function has a returns function itself. And this is us telling sign on, hey, when somebody calls getAllEvents, I want you to return this value. In our case, thats mock events. So now, whenever anybody calls getAllEvents on the mock events data object, they're going to receive back the mock events object. And that matches up exactly with what we want, because inside of our control, we call getAllEvents and assign it to the events property, and we're going to check and make sure that what gets assigned to the events property is the mock events object. This will fix that functionality so that if anybody comes in and changes the event list controller so that the return value from getAllEvents is no longer what goes on to the events property of the scope, this test will break and warn them that they've broken functionality. And one thing that you may notice is we don't actually use this controller variable that we're creating here on line 18. So even thought the controller gets constructed, really the only thing we care about is what it does to the scope. We don't actually need to do anything with the controller inside of our test after it's done its job of initializing the scope. So, let's remove this and then, let's run our test. So, go down and click the Run button. And you can see that we've now got a message done one of one, we have all of our tests passing, and our test for our event list controller is now working.

Testing Controllers in Angular Part 2

Now, let's write another controller test. But this time instead of testing event list controller, let's test our edit profile controller. So, let's open up the edit profile controller, and we can see that it's doing a couple of things. It's creating a blank object and assigning it to the user property, and it's also setting up a get gravitar URL function. So, let's test that get gravitar URL function. I'm going to open up a test file for this controller that I've created. And I've already created the empty wrapper for it. And we'll just do a little bit of copy and paste magic and go back into our other test. And let's grab the before each. Well, let's grab from this var all the way down to here. And we'll copy it into our test. But, we don't need a mock event data for this test. Instead, we're going to need a mock gravitar URL builder, because that service comes in alongside the scope here on line four, and so we'll need that mock service, and we will change this mock event data here to that. And the signature of that isn't get all events. Instead, we call build gravitar URL, so we will add a build gravitar url function, and then we can create our actual test. In this case, our test will be that it should build our gravitar with a given email address. If we look back at our call to our function, the function itself takes in an email address, and then passes that same email address into the build gravitar URL. So, that's the functionality that we will test. We'll make sure that whatever comes in on the parameter on line seven, is what goes into the build gravitar URL call on line eight. So again, we'll have to construct our controller, and we do that by passing in the name of the controller, which is edit profile controller. And the second parameter, of course, is the parameters that go into the controller, which we have two, the scope, and the gravitar URL builder, which is going to be our mock gravatar URL builder. Now, we need an email address to pass in, so create an email variable. And now that that's created, we're going to call the get gravitar URL function that's on the scope, which is the function that gets built by our controller there on line seven. So again, remember, that function does not exist on the controller, it exists on the scope. So, we call scope.getGravatarUrl, and we're going to pass in that email address, and then, we can set up an expectation that the mock gravatar URL builder, we want to check that the function build gravitar URL was called with, and again, this is a little bit of sign on magic. Sign on tracks any function that was called on a sign on object, every invocation of what parameters it was called with. So, we're going to check that it was called with the email address that we created, and we're going to expect that this is truthy. And again, this will verify that whatever email address gets passed into the call to get gravitar URL, is the same address that gets passed into build gravitar URL. And if we run our test, we now have two of two passing. And there's the basics of testing controllers in Angular. Again, a few key pieces that are important to know and that is, how to create a controller how to pass a blank scope object, so that it can fill out that scope object, and how to create and pass in any mock services that that controller needs to do its job so that you can test the functionality for the controller. Since the controller's job is usually to set up the scope, when testing a controller, you usually end up testing the scope that it creates. So, just be aware of that, that what you'll usually end up doing is just testing the scope that the controller creates.

Testing Services in Angular

In this section, we're going to learn how to test services. Specifically, we're going to learn how to test very simple services. So, we're going to start by looking at testing the calendar helper. The reason that we're going to start with this one is, as you can see, it has no dependencies. So, we don't need to worry about mocking anything. Instead, it just returns back an object with a couple of functions, and each of those functions takes in some parameters, and returns a value. That's going to make this very easy to test. And really, we're going to just use this as a way to get warmed up and see the very basics of testing services, so that we can be comfortable when it comes time to test more complex services. Because of that, we're not going to test both of these functions extensively. Instead, I'd just like to test the get month name function, and I'm just going to write one simple test for that function. So you can see, I've already got the spec created for it. So, I'm just going to write one simple test. I'll write a test to verify when we pass in the zero, January is returned. Now if you'll remember when we were testing our controllers, in the before each function, we called the inject function. That gave us the controller constructor and the root scope. Inject is a way to ask Angular to give you services. So when we're testing services, we just the inject function to get the service so that we can test it. So, I'm going to call inject here, and then inside that, pass in a function. And the parameter I'm going to give that function is the service that I want to test, so in this case, it's calendar helper. Now, this looks a little different than the test that we wrote for the controller, and probably any other Jasmine tests you may have written, because instead of just passing a function into it, we passed the inject. But, the net result is the same. The tests still work just fine, but this allows us to get a calendar helper that's constructed by Angular. So now that I've got that calendar helper, I can write my expectation. Now, one more thing I need to do before I can run this that we learned about when we were testing controllers is I need to invoke the module that has this service inside of it. And the way that I do that is in a for each function, where I call the module function, and give it the name of my module. And now you can see that as soon as I saved it, it ran the tests, and executed my new test, and that one passed as well.

Testing Services with Dependencies

Now that we've seen how to test simple services, let's learn how to test a slightly more complex service. This service, the user data service, has one dependency, the user resource service. It has three functions, get user, save, and users, and each of those utilizes user resource. So in order to learn how to test a service like this, let's write some tests for get user and for save. Looking at get user, it does essentially one thing. It calls userResource.get and returns that value. So, let's write two tests, one that tests each of those aspects. Now, since I'm going to be writing tests for two different functions, I'm going to make this a little bit easier on myself, and I'm going to separate them out in their own describes. Now that I got a describe for get user, I'm going to write the tests for get user inside of that. Now remember, I have to call inject in order to get an instance of my user data function, and then, I'll take the important action on the user data service, which is to call get user. And pass in Bob as the user name. Now since user data internally calls user resource, and I want to test what parameters it calls user resource with, I know am going to need a mock user resource. So, I'm going to set up an expectation that utilizes the mock user resource. And I'm going to check that the first argument of the first call is an object that has a user name of Bob. I used to equal here instead of to be, so that it doesn't check identity, but instead, just checks that the objects are equivalent and have the same properties with the same values. Now, the next problem I need to solve is that I don't actuallly have a mock user resource object. So, I'm going to create that in a before each. And since the variable has to be accessible both in the before each and in the it, I'm going to define that variable outside of them both. Now, the user resource object has one function that I care about and that's get. Okay, and now I created the mock user resource object. But, the user data object, when it gets constructed by Angular, is still going to use the real user resource service that I created in my application, and not this mock one. So, I need to tell Angular that when the user data service asks for a user resource service, to use my mock instead of the actual service. And the way that I do that is by getting a hold of Angular's provide object. And in order to get a hold of the provide object, I call the module function and pass in a function. The provide object has a value function. And that value function takes in two parameters, the name of the service and the value that should be provided whenever anybody asks for that service. By calling this line of code right here, I've told Angular that whenever anybody asks for a user resource object, to use my mock user resource instead. And now that I got that set up, I've got one last piece to do, and that is to invoke the module. Now that I got those in place, I can save and run the file. And now that that's ran, you can see that the new test is passing also. Okay, let's write another test for get user. Looking at the get user function, we can see that it returns whatever resource.get returns. So, let's just verify that with a test. So, I'm calling it user and capturing the output. I want to expect that output as some known value. So now, I just need to tell my mock user resource to return a string Angular Rocks. All right, let's save that and see what we get. Okay, and that test is passing too. Now that we've got those two tests going, let's get one more test. And we'll write this one for the save function. So, I'm going to go after my describe, and write a new describe. And I'm going to go look at the save function, and I can see that it calls userResource.save with whatever value you pass in. So, let's just write a test for that. Okay, so I've called userData.save, and I've created an expectation that the mock user resources save function was called with that same value. Let's save and that will run our tests and see what we get. Okay, we're getting a failure, so let's look at it and see what the problem is. All right, our error is that the object has no method save. Well, the object it's trying to call save on is my user resource object, which in my test, is the mock user resource object. So if I go up in my test, we can see that I've only defined a get function for my mock user resource object, and not a save function. All right, now that I've got a save function defined, let's try running this test again. Okay, now our test is passing. So, there's a great example of how to test a service that has a dependency, and be able to provide a mock dependency. Whenever we're testing services, we always want to use mock dependencies for all the dependencies on our service, so that we're testing our service in isolation. That way, whenever any tests for any of our services break, we know that what broke was the service, and not one of its dependencies.

Testing AJAX Services in Angular

The other kind of service we'd like to learn how to test, the services that talk to the server using http or resource. So, we have a great service as an example of that, which is our event data service. We'd like to test that. Now, of course using sign on, we could just pass in a mock $resource service into this event data service, and then, just test the code that we write. It gets called the way that it's written. But, that's a little bit ineffective, because we want to know that we configured our resource correctly. And the only way to really know if we've configured it correctly is to know that is makes the calls the way that we expect them to be made. So, that it calls to the correct URLs using the correct verbs like a get or a post. So, other than using a mock resource, it'd be nice to use a real resource, but instead of letting it actual make real http calls, we'd like to intercept those issue http calls and just find out what they were. And there's a special service inside of Angular's mocks library that we can use to do exactly that. So, let's go to our event data spec, which I've created, of course, with our empty describe function. And, of course, we'll start off with a before each to invoke the module. And our first test will be to check that when we call get event on our event data service, that we call to the right URL. So, we'll say it should issue get request to /data/event. Must use and ID of 11 when get event is called and the ID is 11. And the second parameter for our it function is not going to be just a simple call back function, but instead, we're going to pass in the inject function. Because, of course, we need the live event data service, but we also need another service from Angular, and that is the http backend service. This is the service that intercepts any actual XML http request calls, and will tell us what they are. And so inside of this call back function, we're going to do a couple of things. First, we know that we're going to be making a call, hopefully, to /data/event/11. So, we're going to take that http backend service, and we're going to say that you are going to expect a get of /data/event/11. And then, we can call eventData.getEvent and we'll pass that on the ID of 11. And again, going back to our event data service, when get event is called, it takes in an ID, and then internally, then calls resource.get, with an object whose ID is set to the ID that was passed in. So, when that happens again, it should call /data/event/11 based on how we configured the resource inside of our event data service. The other thing we have to do now that we've actually caused the resource to do something and make some kind of an http call, is we need to flush the http backend. And we call .flush, and this basically just makes that service go through any calls and report out what happened and respond to them if a response is necessary. Then finally, we can check and make sure that we only made the calls that we expected. There on line nine, we set up an expectation for our get, so let's check that we only have the calls that we expected, if we can call httpBackend.verify no outstanding expectation. And we can also call httpBackend.verify no outstanding request. And that's most of the way. But, the other thing that we need to do is tell our http backend what to do when this actual call to /data/event/11 happens, what it should do, and what it should return. In this case, we don't really care what it returns, we just need it to return something. So, we call httpBackend.when. And this allows us to say, hey, when a specific call comes through that matches this verb, so we pass in the verb, which is going to be get, and this URL. Then, I want you to respond by returning a specific object or a return value. In this case, we're going to return an empty object. And because we're not doing anything with the return value, going back to our event service, all it does is on line five is return whatever comes back. And so, we don't really care what gets returned in this test. We just want to make sure that only that specific call gets made. So now, if we go and run our test, we can see that all of our tests are passing. And so, if we were to go down here and change this ID from 11 to 12, then the URL would no longer be /data/event/11. It'd be 12 in our test would fail. And we can verify that by running and we get, indeed, one failure. So, we'll change that back to 11 and run again. And our tests are passing.

Testing AJAX Services in Angular Part 2

So now that we've got our first tests running, let's write a couple more. Let's check that we returned the correct data when get event is called. And, of course, we pass in our inject, bringing in the same event data, and the http backend. Now, this test is going to do almost the same thing as the previous test, but in this case, we won't set up expectations. We'll just check that whatever we respond with is what actually gets returned out. So again, we'll use http backend and we'll say, when you receive a get request to /data/event/11, that you respond with an object. In this case, we can form the object any way we want. So, we'll give it a name property and call it my event. And then, we'll call that get event, but capture the output. And then, of course, we have to flush the http backend. And then, we can set up an expectation that the event.name is the string my event, which is what we set in the object to be returned. So if we run our test, you can see we've got five passing tests. Now, these two tests essentially do the same thing, but they do them in different ways. The first one, we set up these expectations, this expect get right on the http backend, and the test fails if that get doesn't get called. Whereas the second test, we just check that the value that comes out of the call matches. But, we've also set it up so that that value only comes out when a specific URL gets called. So, we've said that when we get a get request to data event 11, then respond with this object. Then, we call the method and check that that's the actual object that we got. Now, one interesting thing here is that even if we were to change and make this the wrong URL, if we run the test, we'll get an error that says, hey, unexpected request to get data event 12. So, it already clues us in on what the problem is. So, it's pretty much just like the previous test, only it's set up a little bit differently and it requires a little bit less code. So, I kind of prefer the second method of doing tests with the http backend. But, either method is completely valid. So, now we've effectively tested the get event method on our event data service. Let's also test our save method. And, we know that when we call eventData.save, if we look at the code, it actually hard codes the ID of the new event to 999, which of course, is just a little bit of hard coded value here because we didn't want to create an ID algorithm. So, we'll just test that that functionality works. In a real program, you'd probably generated IDs and then you'd rewrite this test just a little bit to make sure that the new ID was set correctly to a unique value. But in our case, we'll just check that it's set to the value that we expected. And again, we pass in inject. And we'll say, http backend when, this time, a saved call causes a post to happen. So, we'll check for a post with data/event/999. And we'll respond with a blank object, then we'll create a new event to pass in. This will be an object that has a name of my event. And then, we'll call the save method. Then, we'll flush the backend. And then, we'll expect that event.id is now 999. And if we run our test, we can see that that test is passing too. If I were to change this to 998, of course, I get a failing test because it's going to set the value to 999 instead. So now, we've got tests for get event and for save. We got one more method on our event data, that's to get all events. Let's write a test for that. And we'll just check that it makes the right get request. Passing in our inject function to grab the event data service and http backend service. And in this case, we'll set up our http backend to say that when we get a get request for data/event, and 'cause that's the URL that we get a request to when we call a get all events. Again, get all events, it just calls resource.query, so that uses the URL that we initialized the resource without any ID, so that would be just data/event, and then we respond. In this case, we're going to respond with an array that has one object in it. It has a name of my event. And then, we'll capture the result of that and, of course, flush the back end. And then, we'll set up an expectation that our first event, which would be the zeroth element in the events array, that it's name property is equal to my event. And we'll run our tests. And we have seven passing tests now. So, there's the basics of using http backend in order to intercept http calls and check that only the calls that we expect are made. And one of the features of http backend, of course, is that it will respond with whatever you set it up to respond to, based on the get request or the post request and the URL, whatever the verb in the URL is. But, another nice feature of it, is if it ever receives any unexpected http requests, then it will let you know, hey, I received an unexpected request. And that way, you're not getting any unexpected http requests inside of your services that use http or resource.

Testing Filters in Angular

In this section, we're going to learn how to test filters. Filters are fairly easy and straightforward to test. We only have one filter in our application, the durations filter, so we're going to write a test for that. The durations filter takes in one of four values. It returns out a string based on that value. Rather than writing all four tests, we're only going to write one test today, and I'll leave it as an exercise for you to write the other tests. Now, just like always, we know we need to invoke our module, so let's just start out with that. Now, I'm going to write a test that verifies whenever I pass in a one to the durations filter, it returns the string half hour. And as with services, I need to call the inject function in order to get my filter. But unlike services with filters, the paramater that I pass in needs to have the suffix filter. Going back to the filters object, you can see that the name of the filter is durations, but when I want to get a hold of it, I give the parameter durations filter. Now, my expectation will be very straightforward. The filter is just a function and I pass in the value and I verify that the output is what I expect. Now, let's run that test and see what we get. All right, you can see that the test has passed. But, if I come in here and change the expected string, the test is going to fail. So as you can see, testing filters is really straightforward. I still recommend that you play around with this and write other tests around the durations filter, or your own filter if you have one to test.

Testing Directives - Overview

In this section, we're going to look at testing directives. Testing directives can be rather complex. So in order to try as much as possible to clarify how it works, we're going to look at a diagram of the pieces involved and how they interact in a directive test. The first piece is the directive we have created. We won't explicitly tell our test to include this piece. Angular will get it for us and include it. Next, is the template for our directive. If this is an external template, then we will have to make sure that it is loaded for the test. If it's embedded in the directive, then we won't have to worry about it. Next, we create and set up a test scope for the test. We can give it whatever values make sense for the test. Then, we have a string that is some HTML that references the directive that we're trying to test. For example, if our directive is named my directive, then this is a node that's named my directive. Now, if we are using an external template in our directive, then we'll also need to tell Karma to run that directive through the HTML to JS preprocessor, which will take the HTML and create a snippet of JavaScript that will create the corresponding nodes in the DOM. Now that we have all these four pieces, we're going to run them through Angular's directive compiler. The essence of this piece is that it will look at the test HTML node represented by the green box, see that it has a directive in it, and go grab that directive. If that directive reference is an external template, then Angular will gather that piece from the HTML to JS preprocessor. Then, we will explicitly tell it to compile with our test scope, and Angular will take all of those pieces, and produce a set of valid HTML nodes that are attached to the DOM, but do have events wired up if our events specify them. Say for example, an NG click directive will produce an event list there under click events. Now that we have these DOM nodes, we can test them to see if they were created in the manner that we were expecting. As a result of this method, utliimately we're just testing a part of the DOM, and as such, these tests can evolve using a lot of jQuery to find a particular node. And then, either looking at that node, or triggering an event to see if that event was correctly wired up to some method on our scope, or that the directive takes some specific action when that event occurs. Like the other sections in this module, the most effective way to learn about testing a directive is to dig right into the code. So, let's turn to our project and write a directive test.

Setting up Karma for testing Directives

In this section, we're going to set up Karma to test our directives. Testing directives is a bit different than testing the other pieces of Angular, because of external templates. Normally, when Angular encounters a directive that specifies an external template with the template URL property, Angular will then make a get request for the template that that directive uses. When running in Karma, we don't want that get request to fire. Instead, we want to have our templates already loaded to allow Karma to supply those templates for us. So in order to do this, we have to do a little Karma wizardry. So, the first thing we'll do is install another NPM module. So using NPM, we need to install the HTML to JS preprocessor for Karma. This is the name of the node module, Karma NG HTML to JS preprocessor. I'm going to go ahead and install that. And once that's done, I'll go back into my project and make some changes to my Karma configuration file. Here in my Karma configuration file, I've got to change three things. First, I need to add my partial HTML files to the list of files that Karma will load. So, I'll go down to my files list, and add a new entry. Here, I've told Karma to load all of the HTML files inside of the directives subdirectory inside of my partials directory. Next, I've got to add the module that I just installed to the list of plug ins that Karma will load. And finally, I've got to add a preprocessor section to my configuration. It doesn't really matter where you add this. So, I'm just going to put it at the top. This preprocessor section is an object whose key is the pattern of files that you want to be preprocessed. So in this case, I want all of my HTML files to be preprocessed. And the value will be the name of the preprocessor. In this case, the name isn't exactly the same as the name of the module. It's just NG-HTML to JS. These three changes will get Karma to load all of my partial files for all of my directives. This will allow Angular to run directives under Karma, where it's not possible to request external templates for directives using the normal manner. Now, the most common way to get this wrong is through the paths. The path to your template URLs needs to match exactly the path to the templates inside the files section in your Karma config module. Let's look at an example of this. I'm going to open up the event thumbnail directive. And you can see right here, my template URL is partials directives event thumbnail. So, that path needs to match path in my Karma config file. If we go here, you can see here that all of our partials start with partials/directives/start.html. So therefore, the path is the same. It begins with the word partials. If I were to change the base path, so that instead of starting from app, it starts from the directory above app, then in order to get the partials, I'd have to put app in here. But now this path, from app partials directives doesn't match the path inside of the template URL, which is just partials no leading app. So, we've got to make sure that our base path matches up with how the templates are referenced. Now, there is one way to get around this. You can add a section to your Karma config file that gives directives just to the HTML to JS preprocessor. Let's add that section and show how it would work. Let's assume that our base path is a directory app, and even though that breaks the rest of our file paths, and we'll just ignore the fact that all the other paths are wrong and just worry about the partials path. So, this is our partials path in our Karma config file, yet the path inside of our directives is different. What we need to do is tell Karma to take off the leading app. So, we'll add a section to our configuration file called NT HTML to JS preprocessor. Inside of here, we're going to tell the preprocessor to strip the app prefix. We do that by adding a strip prefix key, and then setting app/ as the key. Now whenever the HTML to JS preprocessor processes an HTML file, it will strip off the app as part of the path and treat the path as if that wasn't there. And that will make our templates load correctly once again. This section can also preprint a prefix, if you need to add one. So in either case, whether you need to strip a prefix or add one, you can use this section to do it. Now, I'll clean up these changes, and now we're ready to start testing our directives.

Testing Directives in Angular

Now that we've got ourselves set up in order to test directives, let's go ahead and write our test for our directive. So, we'll use the event thumbnail directive as the directive that we're going to test. And if you look at this directive, you can see that the directive itself really has no functionality. All that it does, if you look at the HTML is you can see that it binds a bunch of data onto the DOM. So, let's just test that it binds the event name correctly. We can see that on line three here, that it binds the event name value in there. So, let's write a test to check that it does exactly that. Of course, the complex piece here is going to be all the set up. We'll start off by invoking our module. And the other thing that we need to do is make sure that our template is available. Again, the event thumbnail JS file has a template URL of /template/directives/eventthumbnail.html. I'm going to copy this exact string and go over to my event thumbnail spec. And remember that in our preprocessor, we're loading up all of the templates. And in order to make those available to our test, we actually have to invoke them, and using the same function that we invoke our module. So, that's a little bit confusing. We call the module function, but instead of passing in the name of the module, we pass in the name of the template. And, of course the string has to match exactly. Now that we have that, our template is available, so that when our directive makes a get request, instead of actually doing a get request for the template URL, it will internally use the preprocessor and just grab that HTML template out of the preprocessor's cache. So, or test is going to be that it should bind the data, so we'll do a very, very, very simple expectation here. We're going to expect that the HTML that gets created has the name of the event in it. So, let's assume that we've got a handle to some element which is the root of the HTML that's created by the directive, so we'll call that el. And remember in Angular, elements are wrapped by jQuery, so there's a text property, which spits out all of the HMTL as text. And we're going to check that this contains the name of our event. So, let's make the name of our event something a little unique. We'll call it Angular Expo. Now, of course, we're not done. We haven't given our directive any data to bind to, and we really haven't even gotten the directive itself to run. So, here's where all the complex magic happens. We're going to need another before each function, and we'll pass in inject, because we need Angular to give us a couple of services. The first one is the compile service and the second one is the root scope service. We need the root scope service because we're going to need a scope for our directive. We need the compile service because we actually need to compile the directive. Angular itself essentially just compiles directives, that's its main function. That's how Angular runs as it goes through your HTML, finds directives and then compiles them. And this compile service is the main service that does this. So, this is kind of the heart of Angular right here. So first, let's create our new scope. And then, let's created on that scope an event object. And the reason we need that is if we go back into our thumbnail JS, we can see that the event is some data that is passed into our directive. So, we will say scope.event and we'll set that equal to an object that has all of the data that needs to be available to bind onto our HTML. The key piece is our name, because that's going to be what we're searching for in our expectation in our test. So, the name will need to be that Angular Expo, and in order for this to work, we're going to need the other pieces of data, which are the date. In this case, it's just a string. So, pass in one, one, 2050. Add the time. Another string, 1030. We need a location, which is itself an object, and that has an address, which we will set to 1234 Fake Street, and a city, which we'll set to New York, and a province, which we'll set to New York. And let's double check our HTML and make sure that we've got all this address, city, and province. We can look up at the top. Ooh, it also needs an ID, so let's give our event an ID. And we'll set it to one. So, now we got our event created. Next, we need to create and compile the directive. So, this is where things get a little bit tricky. Again, we have this el property down in our test, that's going to be the DOM node that is the root of the directive. We're going to need to access that both in the test end here and in before each. So, let's create a variable up here and inside of our before each, we'll say, el equals angular.element, which lets us create an element. So, we're going to create the DOM element that is the directive itself and causes Angular to compile it. So, that's our event-thumbnail. And this is what we'd find somewhere else in our HTML that causes the event thumbnail to get run when Angular is running live. But, here we actually have to trigger it, so we're creating a brand new one. And of course, it needs that event property. So, we'll just say event is the name of the property we're passing in and we'll close it off. And so now, we got this element. And again, when Angular parses through our HTML, it finds directives and then compiles them. And in the compilation process, it turns them from whatever they are into whatever they should be after the directive itself runs. So, that's what the compiler service does. So, we'll call the compile service and we pass in that element, or DOM node. And what that compile function returns out is another function itself, which we have to invoke with a specific scope. This is kind of like a template function for the specific directive, but it can't do anything until it actually has a scope to operate against. So, since we've created a scope up on line nine and filled that scope out with the data it needs on line 10, we can just immediately invoke that function, and pass in the scope. Now, there's one final thing that we need to do in order for Angular to update the bindings. And that is, we actually have to run a digest cycle. Again, Angular does not update its bindings to HTML until a digest cycle runs. So at this point, we've turned that element into the expanded template that our eventthumbnail.html has, but none of the data has been bound. But, once we call scope.$digest, that kicks off a digest cycle and it'll go through and bind all of the data into the HTML, and now the HTML will have the values that we want. And now if we go and run our test, we can see that we've now got nine of nine tests, and all of them are passing. And this new test is actually checking that somewhere in our element we have the text Angular Expo, which is a pretty simple test. We could certainly do more complex stuff, but good enough to show the basics of testing directives. Now, let's talk for just a second about troubleshooting. One of the things that easy to do is to somehow get this wrong. For example, what if we misspelled that thumbnail? We said event thumbnail two, but of course, we only have the event thumbnail directive. If we run our test, and it's going to be failing. And all that we get is expected a blank string to contain Angular Expo. Again, the text property grabs all of the HTML that is there and looks at the inner text of each piece and appends it all together. But, what we're getting for that is a blank string. And that really doesn't tell me what's going wrong. So, one of the things that's really easy to do in order to diagnose what might be happening is to log out to the console the element, and this is a jQuery wrapped element, so we only need to grab the first one of those, because it's the jQuery wrap set. And we just want the first one, which is going to be R node, and then we can call the outer HTML property. And what this will do is actually write out to the console the entire HTML that gets compiled. So if we run out test again, we can look, and we see right here that the directive itself really didn't even get compiled. It's still just event-thumbnail two, with an event object. So, if we go in and change this to event thumbnail, and run our test again, then we can see that the HTML that gets printed out is actually the compiled HTML. So, it's taken that event thumbnail and replaced it with the contents of the event thumbnail HTML. So, now we know that the directive is getting compiled correctly. If we see just the original node, like when we have the two in there and nothing got changed, then we know that there's something wrong with just recognizing that this is a directive that we want. So, that's a good way to troubleshoot if you have a test for a directive that's failing and figure out why it's failing. And so, there's the basics of testing directives in Angular. As you can tell, there's quite a few steps that are involved. Understanding each of these steps is important so that whenever you have any problems you can diagnose and troubleshoot what the problem is.

End to End testing - Overview

In this section, we're going to cover end to end testing. End to end testing in Angular is a really, really cool piece of functionality that Angular provides. It's essentially integration testing, except it takes advantages of the internals of Angular, to make it a lot easier to integration test your application. If you've ever tested an application using something like Selenium, you'll know it can be very difficult and painful to test a web application that way because of the asynchronous nature of the web and constantly changing HTML. Angular's end to end testing mitigates a lot of those problems. Let's start by looking at a conceptual overview of Angular's end to end testing. First, you'll need to launch two things, Webdriver server, and your application server. Webdriver server is a special NPM module that you'll download and install. Your application server will either be your node server, your asp.net server or really whatever server that's serving up your application. Once those two pieces are running, you're ready to execute your end to end tests. Whether you're going to execute some pre-existing tests, or you're in the middle of writing some, the way that you execute tests is with Protractor. Once you launch Protractor, Protractor will read your tests and determine the steps that you want to execute in order to verify functionality in your application. Protractor then communicates with Webdriver server, Webdriver server then launches a browser and executes you application. At this point, the test is executed and the results are communicated back to Protractor, which reports to you whether the tests pass or failed. This process does have quite a few moving parts, but overall, it's fairly simple to get these parts installed and up and running.

Setting up Protractor

Setting up Protractor isn't particularly difficult, but it does involve a few steps. So, the first thing we need to do is install Protractor and Webdriver Manager itself. The first prerequisite that you must have installed is the Java Development Kit. And you can check and see if that's already installed in your box by typing in java-version. As long as you get something back, then you should be okay. If you don't, you're going to need to install the Java Development Kit. And installing that is a little bit beyond the scope of this course, because that could depend on your platform. But, the easiest thing to do is just Google JDK and the first result will be the download page for the Java Development Kit on oracle.com, and just follow their instructions to download it and get it installed correctly. Once you have that installed, the next thing to do is to install Protractor. Protractor is a node module so we'll use NPM and we will install Protractor, and we will install that globally with a -g. I'm not going to run this myself 'cause I've already got it installed and it can take a minute to download and install everything. But, it's a very simple process. And you may even notice, if you're on a Windows box, that you might get an error. You can possibly just ignore it 'cause some of the errors that it reports out aren't critical errors. Once you've installed Protractor, the next thing you need to do is update Webdriver Manager. Webdriver Manager is installled with Protractor. And to update that we use webdriver-manager and update. And once you run this command, that will update your Webdriver Manager. Updating it and once it's updated, we can start it up. And we'll need to have the Webdriver Manager started and running at all times. This is also known as the Selenium server. So, we just issue the start command to the Webdriver Manager, and we'll notice that it's now running, and this we will just leave running the entire time. The last piece we'll need to do in order to set up Protractor is to go back into our project, and we're going to need to create a config file for Protractor. So, I'm going to go to my test directory and I'm going to create a new file, and I'll name that protractor.conf.js. And this file is relatively simple. We will say exports.config equals an object. And if you're familiar with module systems, you'll recognize this syntax. If you're not, it's no big deal. Just follow along exactly. And this object needs a few properties. The first one is the address for the Selenium server and that's specified with a Selenium address. And we give it the string of the URL of the server. By default, the server's going to be running at localhost: 444/wd/hub. So unless you made any changes to your Webdriver Manager when you started it up, this is the URL that it's going to be available at. Next, we need to tell it where our tests are at, and that's with specs. And this is an array. We're only going to feed in one string property into this array, but of course, we could feed in multiple if we needed. And this is going to be our e2e, which is our folder, and inside of that, all the JavaScript files. And finally, we have to give it a capabilities property. And this is an object. And the only property that we need in here is the browser that we want to use. So, we're going to use Chrome. You give it a browser name, property, and inside of here specify Chrome. Be sure it's all lower case. We could use Fire Fox, or something else. For more information, you could always check out the Selenium documentation to see how to use other browsers, but we'll just use Chrome. Now, we could run Protractor at this point, but we get an error because there are no files that match this pattern here on the specs property, because there's no JS files inside of this e2e directory. So, let's go into our e2e directory, and let's create a new file. And we'll just name this scenarios.js. And we can just leave this blank. It doesn't actually have to have any code in it. Then, we'll go back out to the command line, and I've opened up a new console window because I've got to leave the Selenium server running in that other tab over here. So in this new terminal window, I'm going to run Protractor and then give it the path to the configuration file, which would be test/protractor.conf.js. And once I execute that, Protractor launches, runs. It found zero tests, so it didn't execute anything, but it did run, but now we can write tests and come back and run Protractor to run our tests.

Setting up Protractor with Webstorm

Setting up WebStorm to run Protractor is really nice because that way, you don't have to leave WebStorm in order to run you Protractor tests and go all the way back out to the console. So, we set it up kind of similar to how we set up Karma. We go to our configurations and edit configurations. Of course, you can find this on the Run menu if you don't have that toolbar visible. And I'm going to create a new configuration. And I'll select node. And now, I'm going to name this Protractor. And, there's a few things that are important. This node interpreter has to be set correctly. It by default should find where your node interpreter is, but you may need to double check that. Node parameters, we can leave blank. The working directory is going to usually be defaulted to the correct value, which is just the root of our application. The JavaScript file is going to be a little bit different. I'm going to need to actually find the cli.js inside of Protractor, which I installed globally. So on my computer, this is the path to my cli.js file. If your node is installed somewhere else, you may need to browse around to find it. But, inside of your node directory there's usually a bin directory that has your node modules, which will contain all of your globally installed node modules. And inside of that directory is a lib directory. And inside of that is the cli.js. Then, the final thing we need is application parameters, which is actually the path to the configuration file for Protractor. In which case, it's just test/protractor.conf.js. And once we have all the settings, we can click OK. And then, we've got Protractor selected and if we just click Run, WebStorm will run our Protractor tests for us. So, there we got Protractor set up in WebStorm.

Angular End to End testing - Part 1

In this section, we're going to write our first few end to end tests with Protractor. So, we'll start off in our scenarios file and put in a use strict, and then our describe. This will be our event registration app. And within this, we'll start by testing our events page, so let's put another describe. And events page will be that. And we'll start off by just navigating to the right page. So, before every test, we'll tell Protractor to go to the page we want to go to. And we do that in a before each function, and there's going to be a global brower variable, because we're inside Protractor, and we can call get, which will tell it to issue a get request, and we give it the URL that we want it to go to, which is local host and we are going to be at port 8000, which is where our app is running, and events is where we want it to point at. So before each test, Protractor will now browse to that URL. But, we do need to make sure that our server is actually running. So, this is one more moving part of running Protractor tests. Not only do we need Selenium Webdriver server running, we've also got to have our website running, and then we execute the tests inside of Protractor. So, there's kind of three pieces in that. So, I'll just go out to the command line and I still got Selenium running in my other tab. And inside of this tab, I'll run my server. And that's running on port 8000, and now if I execute this test, we've still got zero tests, zero assertions, but we're not getting any errors. So, next we'll actually write the test itself, which is going to be inside of an it function, and we'll say that it should have the correct title and first event. So, let's go ahead and look at our events page and see what I'm talking about. On our events page, we've got four events that are showing up. The first one has a title of Angular Boot Camp. So, this test will verify that that's the first event that's showing up. We've already told Protractor to go to the correct page, but let's look at the HTML that we're specifying inside of our Angular application, and that is at eventlist.html. You can see that what we got here is a repeater that is repeating over events in events. So, let's go back and we can actually find that repeater as a list. We'll say list equals element, which is a global that Protractor provides, and we'll say .all, which means that we're getting a list of elements. And we're going to find these elements by repeater. So, by.repeater is another thing that's provided to us by Protractor, and we can give it the event in events, which tells Protractor go and find a repeater that's on this page that's repeating over event in events. And now, we have that as a list. Now, let's first verify that we got the right number of events. So, our list should auto count it for us. So, we can expect that the list.count, which is a function, we'll expect that that equals four. So now if we run our test, it has come back and said that we've got one assertion and it's passing, and we've got no failures. So in addition to checking and making sure that we got four in our list, let's also make sure that the first one has the right title. So again, we want to find the correct element. In this case, we want to find the title of the first element, but let's check out the HTML and see what we're looking for. If we go into event list, we can see that we're just iterating over a list of event thumbnails. So, let's go into the event thumbnail HTML, and we can see that the event name is inside of this H2, and this is kind of binding to event.name. Well, we can take advantage of that. Going back into our file, we can find the element that has that binding. So, we can say for our title element equals element, which is again a function provided by Protractor, which helps us find elements. And we want to find this element by binding. Again, a little special function that Protractor provides that we can give it the name of a binding. So, in event thumbnail we're binding to event.name. So inside of a string, I'll say event.name and that will get the element that has a binding to event.name, which would ultimately be that H6 element. Once I've got the element, then I can set up an expectation, and expect that the title element, if we call get text on that, which will get the inner text, that that should equal the string Angular Boot Camp, which again, we can see that if we go back to the browser. You can see Angular Boot Camp with the first letter capitalized in all three words is what that text is. So, if we go and run our tests, we get two assertions now with zero failures. So, that's a good enough testing of our event list page. Let's also test our event details page. So, we'll put in a new describe. And we'll put in a before each instead of at. And in this case, we're not going to navigate to events. Instead, we will navigate to a specific event. And we'll go to event/1, and we can see what that looks like if we go to our browser, and browse to events/1. And here is our page and there's a lot of things on this that we can test. So, the first thing that we should do is make sure that it's sorting correctly and sorting by name. And that's our default sort order. And in that default sort order, our very first event is the Directives Master Class. So, let's say that it should sort by name. And we need to know what HTML we want to grab a hold of. And we can do that by looking at the event details HTML page. And if we scroll down, we can find the sessions inside this page. And we can see, here's our session's section, where we repeat over the sessionevent.sessions, and we can see that each session is detailed out by a collapsible directive. So, let's look at our collapsible directive and see what it's HTML is like. And there's the template for our collapsible directive and we can see that it's got an H4. And inside that H4, it binds to the title. So, let's go back to our scenarios, and we can do this a few different ways. Let's get a list again of all of the elements that are in the repeater. So our list equals element.all by repeater. I'm going back to our event details. We can see it's session in event.sessions. And that gives us our list of elements that are repeated over. And we can grab the very first one by calling first as a function, and then we can within this, find a element by calling the element function. And we can use that by binding. And this time, it's going to be a title. And let's capture that into a title element again. And then, let's expect that the title element get text is equal to Directives Master Class. And now if we run our tests, we can see that we got two tests with three assertions total and none of them are failing. So, one of the things that we see with this new test is that calling the element function on an element that we've already gotten right here where we call .first and then call .element after that, it actually searches just within that first element. And that's really nice. But, if we go up to our previous test and compare, we can see on line 13, right up here, that we don't do any filtering. We just call this element function which searches all elements on the entire page. And therefore on that page, there's actually four different bindings to event.name across the entire page. So because there's four, this function will actually just pull the first one by default. So, it's kind of the same thing as grabbing the list, grabbing the first element, and basically doing this. This is essentially the same thing, but it's a little bit more relying on the defaults inside of Protractor. Now, let's add one more test to our event details page. Let's test that we have the right number of sessions. So, if we go and we look at the browser, we can count and see that we've got three sessions inside of this event. And that will be what our test will verify. And so again, let's get our list of sessions. And let's expect that list.count is equal to three. And we'll run that test as well. And we can see that we've got three tests, four total assertions, all of them passing. So, there's some basic Protractor tests that show us how to find elements and check that the text inside of those elements is what we expect it to be. In the next section, we'll look at actually making some changes to the page interacting with it, and testing that the changes that we expect to happen, actually happened.

Angular End to End testing - Part 2

In this section, we're going to actually manipulate the page just like a user would, and test that the changes that we made produced the results that we were expecting. So, let's look at some of the things you might do. Let's go back to our page and if we were to select Introductory, then we can see that we're only seeing one session. And that session is Scopes for Fun and Profit. So, let's make a test that verifies that when we select the introductory filter, that we only see one session. So, there's the name of our test. And we will first grab a handle to the select element by creating the select el variable, and calling element. And this time if we look at the event details page, we can see that that filter is based on the NG model of query.level. So, one of the things that we can grab an element by is its model binding. So, we can say by.model and pass in query.level. And now we've got the select element that is bound to the query.level using the NG model directive. Now that we have it, we can say select el and we call the element function again, so that we can find the right option that we want and click it. And in this case, we're going to use the by css containing text, which is kind of a different selector. It takes in two parameters. The first parameter is sort of a css selector, so we're going to say it's an option tag. And the second one is the text that we're looking for, which in this case, is the word Introductory with a capital I. And once we have that element, we're going to click it. So, we just call .click. And now that we've changed that, our list should be different, so let's grab the list of elements, and we can again use element.all by.repeater, and this is session in event.sessions. And let's set up an expectation that list.count is equal to one. And let's run our test. And we have four total tests, five assertions, all passing. And of course, I should go in here and fix my spelling errors. Much better. Now that we've tested the filter, let's also test the sorter. So, let's just go back to the original unfiltered state, but instead of sorting by name, we're going to sort by votes. And here we can see why we used a different set of initial data, so that we actually have some votes specified. And now, this one sorts up to the top, whereas when we have name, Directives Master Classes had sort at the top. So, let's change the sorter to votes, and then check that this is the first one. So, our new test will be it should sort correctly when sort order is changed. So again, we'll grab the select element, but this time, it's not bound to the model of query level. Instead, it's bound to the model sort order, all lower case. So, we specify that. And now, we can find the right option using the same by css containing text, and that's an option. And the text that we want it to have is the word Votes, with a capital V, and we'll click that. And let's grab the first session, which would be element.all by repeater session in event.sessions and we want the first one of those. And that is going to be our first session. Now, we can grab the first session name, by saying first session and calling element on that, which looks for an element in this first session. And in order to find the first session, we'll go to our event details and look and see what we would have for a first session. Again, it's that collapsible and if we remember, the collapsible uses a binding to the title varible. So, let's go back and we can say by binding title. And since you want just the name, let's call get text on that. And now, we can set up an expectation, that first session name is equal to Scopes For Fun and Profit, making sure to get the casing right. And let's run those tests now. Make sure that our new test is passing. And of course that is passing. If we were to change what we're looking for and make it incorrect and run our test again, we get a failing test. We can see five tests, six assertions, one failure. So, let's fix that and rerun our tests, so we're back to a green state. And now, we'll write one more test. Let's test that if we were to up vote a particular session, that the vote count actually changes. So, go back to the default sort order and the default filter, and then we can see that our very first session has zero votes. And if we were to up vote it, it would go to one. So, our new test will be that it should increment the vote count when a session is up voted. Now this time, we need to figure out how to grab the voting buttons so that we can click it. So, let's go back into our event details HTML, and we can see that right here on line 43 is the up vote directive. So, let's look inside of the up vote directive and let's find out what element we want to grab to click and it's going to be on line two, this div with a class of voting button. And if we go back and look at the page, we can see that the very first div with a class of voting button is that one that we want to click. Of course, there's going to be multiple divs with class of voting button. There's one for the up chevron, and the, or the up vote, and one for the down vote, and there's one on each session. So between three sessions, we have six total buttons that are going to be div.voting button. But, that's okay, we just want the first one. So, let's grab element.all, and this time, we'll use the by deep css filter, and we can give it just a css selector, which will be div.voting button, which grabs a list of all six of those divs, and let's grab the first one, and then click it. And now, let's grab the first vote count element, and this is going to be element.all. And in this case, you want to grab the element that shows the vote. So if we go back to our up vote, we can see that this div here on line six has a binding to the account variable. So, we can just do there by binding selector, and we're going to find the one that's bound to the account variable. And again since we're dealing with the first one, we can just select that one, specifically. And of course, on our page there's three of those vote count bindings, one for each session, but we only care about the first one. And now, we can set up our expectation. We'll expect that the first vote count, and that if we call get text on that, that will give us just the text inside of that div. And we can make sure that it is equal to the string one. Now, let's run our tests. And we have six tests, seven total assertions, all of them are passing. So, there's some examples of how to interact with a page. Of course, there's a lot more than you can do. This course isn't an exhaustive coverage of Protractor and how it works. And remember, Protractor is just a wrapper over Webdriver. So, anything that works in Webdriver, will work with Protractor. But, Protractor adds some additional features and functionality. If you are interested in that, there are other courses that you can check out, or you can, of course, check out the official documentation. But, at least here, we've seen the basics of how to interact with a using Protractor and write end to end tests for our application.

Summary

In this module, we talked about testing with Angular. We started out by looking at Karma, our test runner. We looked at how to install and configure it. We also saw how to run it within WebStorm. We looked at some of the configuration options, and we saw what it looks like when we actually run Karma. Then we talked about unit testing. We learned about testing controllers, services, filters, and directives, and how to get Karma to test each of those pieces. And finally, we looked at end to end testing. We saw several examples of how to write end to end tests and talked about how to troubleshoot end to end tests. In Angular, testing is an integral piece of Angular. Writing very large JavaScript applications is very detailed and very complex. Because JavaScript is missing a lot of the features that other languages have, it can be very difficult to manage large applications. Therefore, testing is even more critical when it comes to JavaScript applications. I highly recommend that whenever you use Angular for you clientside code, that you test that code extensively with both unit tests and end to end tests.