Getting Started

Course Introduction

Hi I'm Bradley Braithwaite and welcome to this course. AngularJS Unit Testing in-depth, Using ngMock. This course aimed at to get you up to speed quickly in writing unit tests for your AngularJS applications. We will learn by doing, building this movie application in AngularJS using a TDD approach. It looks at movie data via an HTTP API and has functionality that will require us to write unit tests for common AngularJS patterns. Along the way we will explore ngMock in detail which is essential when writing unit tests for AngularJS code. By the end of the course you will have greater fluency when writing unit tests for your AngularJS applications. This an intermediate level course and knowledge of AngularJS and its core principals such as modules, controllers, services and directives is assumed. Also knowledge of basic TDD principals will help in following the course. The minimum software required to complete this course is a text editor and a web browser. All you need the ability to edit JavaScript and HTML. And a modern web browser with JavaScript support.

Introduction to ngMock

So what is ngMock and why should I care? It was designed by the Angular team to make unit testing Angular applications easier. In fact it's used to unit test the Angular framework itself. It decorates core Angular services adding features that make core modules and components easier to test. And why are unit tests important for AngularJS? Well, firstly Angular is written using JavaScript which is a dynamic language without the compiler. Any typing errors we make in our code will not be seen until run time which can be time consuming to spot. Unit testing is part of the Angular philosophy and the framework was designed with this in mind. Therefore if you're writing an Angular application it's likely that unit tests will be required. Finally, as the complexity of front end applications increases writing unit tests is a great way to keep these sort of code bases under control.

Introduction to Jasmine

The first library we'll take a look at is Jasmine. Jasmine is a behavior-driven development framework that gives us a mechanism to write and then execute unit tests in JavaScript. You can find more detail about Jasmine via the documentation. We will be using version 2.3 for our application. You can find the link to the release page at the bottom of the documentation. Following the link will lead us to the page where we can download the necessary files to use the library. We will use the Jasmine standalone zip folder which you can download from here, manually. Or via the command line which I will demonstrate next. Note that here I'm using the Mac terminal, but if you're using Windows you can enter the same instructions via the command line. First I will create and then navigate to a new folder to hold our application files. Next I will download and then zip the Jasmine standalone zip file. Reviewing the contents of our new folder in the text editor we can see a lib folder that contains the Jasmine source

files. There is also a spec and source folder that contain example logic and unit tests to help us get started. We won't use these files since we will create our own example test. So we will delete them but we will keep the folders since we will use the same folder structure to separate our logic and our tests. The next important file to look at it is the SpecRunner.html page. This is where we link up the Jasmine library, our source code and finally our unit tests, so that we can run everything together in a web browser. To review the Jasmine basics we will create a simple unit test for some calculator logic. To start off we will create a new source file called calculator.js and the unit test in the spec folder called calculator spec.js. If we open the Spec Runner HTML file in the browser we will see nice looking Jasmine interface. But no unit tests. Let's have a test. We will switch back to our calculator spec file and add a describe block. The describe function is a Jasmine construct and allows us to create logically grouped unit tests by name. As you can see here I've given this block the name calculator. Now we're ready to add the unit test within our calculator block, and we can do this using the it construct which is also part of Jasmine. As we did with the describe block we need to give our it function a name. In this example we are going to write basic addition logic so the name of our unit test will be should add two numbers. Next we set up the assertion for our test, by creating an expectation. The expect function is part of Jasmine and we can pass expression arguments such as primitive types, functions of objects that will be evaluated to produce a value. We can then make use of Jasmine's matches to state what the value produced by the expectation should be. If we switch back to the Spec Runner HTML page and refresh the browser we now have a reference error. This is because we haven't yet created the add function for the calculator logic. Let's do that now. Switch to the calculator.js file and create a basic add function that returns zero. Switching back to the SpecRunner we now have a different type of error. This time the interface is telling us that the expectation failed. We can see in the output that it expected zero to be three. Let's update the function to be correct so that the test passes. Switching' back to the test runner and refreshing. And we now have a passing test. This work flow of switching' back and forth between the text editor and the browser it's okay to get started, but there's a more efficient way to execute the tests whilst we're changing code which we will look at next.

Introduction to Karma

Karma is a tool that was also designed by the Angular team in order to make working with unit tests easier. The main benefit it gives us is the ability to execute our unit tests via the command line. This is especially useful in creating a tight feedback loop, when making code changes and reviewing test impact which we will see shortly. Karma is installed by the NPM Package Manager which comes with Node.js. Therefore Node.js is a required install. If you would rather not install Node.js then you can use Jasmine in the browser as we saw in the previous clip and skip the installation.

Installing Karma

Ensure that the location is the root of our new movie app folder and run the npm init command. Fill out the basic information and select yes. And we will have a new package.json file in the root of our movie

app folder. Now we can install Karma. To do so we can follow the same commands as listed via its documentation. (pause for five seconds) The second step is to install a command line add on so that we can type the Karma command rather than the explisive location of the executable from the command line. Windows users will especially want to do this since there can be headaches when trying to navigate to the exact path. The third step is to install the Jasmine add on and choose which browsers we wish to target when executing the tests. Chrome is the default value given in the documentation, but it is possible to target one or more browsers at the same time. Each browser adapter comes as a separate add on that we need or install via npm. For the time being we will install the default Chrome adapter. We can run Karma's init command to help set up the default configuration. We will use Jasmine it's also worth noting if you are a Node.js user that it's also possible to use marker with ngMock. We've chosen Jasmine in our application as it's more commonly used for Angular projects including the Angular project itself. Here we can select which browsers we wish to target. We will choose Chrome for the time being since that's the only adapter we have installed. Next we need to tell Karma where our source files are located. Which is the source and spec folders we saw previously. And finally, we can state whether we wish to execute the tests each time we update our source files. This creates a Karma config .js file in the root of our application that we can modify by hand should we wish to. Now we can run our tests via the command line using the Karma start command. Passing a newly created configuration file as an argument. And now we have the same passing test for our calculator logic as before. A tip I often use is to add the Karma start command configuration to the package.json file in the test section. This means that we can follow the convention and just type npm test from the command line to execute our tests.

Using Karma with Multiple Browsers

You can find the list of additional browser launchers via the npm package website. Keep in mind that for standard browsers you need to have the browser itself already installed. I will install the PhantomJS launcher which will allow me to use the headless browser for even faster test feedback. I now need to update the Karma config file to use PhantomJS. Rewriting the tests I can see that the tests are being executed using PhantomJS. If I update the logic we can see the results of the change immediately. Note that when using the Chrome runner it also started up another instance of Chrome, which made things a little slower. To show how I can target multiple browsers at once this time I will update the Karma config file to use Chrome and PhantomJS and restart the tests in the same way. Now I can see the results from both in the command line. For the remainder of the demo I'll be using PhantomJS only. But we can see the power of this functionality. Imagine targeting Firefox, IE and Chrome all at the same time from the command line.

Configuring Angular and ngMock

The final library we need is of course AngularJS itself and ngMock. The ngMock documentation can be found via the develop APR Reference and ngMock section of Angular's documentation. To use ngMock we need to install the angular-mocks.js file after the main Angular include. We will update our movie

app to include these files. First if you're not going to use the Karma tool, we need to update the Spec Runner file. Also keep in mind that any new files we create will need to be included in this HTML page should you follow this option. And for Karma we need to update the config file to include these new files. (pause for five seconds) The remaining step is just to download the files which we can find via the download section of the AngularJS.org website. Here I'll quickly download them using the command line.

Libraries Overview

Let's recap the libraries we just installed and configured and review how they fit together. We started with Jasmine which is the test framework we will use to write and run the unit tests. This can be used for any JavaScript code not just Angular. Next we look at Karma which allows us to run Jasmine tests from the command line which also supports Jasmine. And finally we installed ngMock which makes Angular testable and is also designed to work with Jasmine. Now that we have our tools configured we can start writing code and exploring ngMock in depth.

First Steps with ngMock’s Core Functions

Introduction

In this module, we will look at the fundamental functions of ngMock that are required for any unit test that targets AngularJS code. We will begin by adding logic to our movie application, starting with the service that will provide the movie data that we need to search and find movies. All the movie data, including the images we see, is provided by The Open Movie Database, which you can find over at www.omdbapi.com. By exploring the documentation, we can see that it has search and lookup by ID capabilities. As we work through the code, we aim to answer the following question when starting our unit testing in AngularJS application, which is, how do we get instances of Angular components into our unit tests? NgMock's module and inject functions provide this functionality and therefore will be our starting points as we dive into ngMock.

Defining the Movie Data Logic

Let's begin by brainstorming in code the logic for the movie search and lookup by ID functionality. Reviewing the OMDB websites, it gives us examples of searching by title and by ID. The search query looks as follows, and we will use this JSon data in our first unit test. Now that we have a sense of the data we will be using, let's start writing a unit test. Switching back to our movie app folder from the previous module, we will start by deleting our example test. Create a new folder in the spec folder called omdb and create a new file called service.spec.js. We will add a describe block to contain the specific logic. I like to work with realistic data so let's copy paste in the JSon data from the search example we

looked up previously. Then we can add a simple unit test for the search logic. We want to test that, if we call our search function, that it returns the data that we expect for the given query. The expectation will be something along the lines of this. Running the test will, of course, fail. So let's now write just enough code to make it pass. Now we have a passing test and a sense of how the data will flow. The next step is to make our code Angular code. So let's think about how we could go about this. We would likely do this by creating a new Angular module for the omdb logic. We could then add a factory to this module and return the functions for the search and find functionality. In the main Angular app that runs in the browser we would load our omdb module via the ng-app directive which would bootstrap our application and load the modules. But, in the unit tests, we don't have the main application to bootstrap our code. So, how can we access our module if we can't use ng-app? Our test would look the same as what we have currently, but for one important aspect. How do we obtain an instance of our module and containing factory? That's our next step.

Using the Module Function

The angular.mock.module function allows us to configure modules individually to be used in unit tests. There are three ways to configure modules, by a string alias, by an anonymous object, or by an anonymous function. The first is by referencing the string alias of a module. We could reference the Angular module for the omdb logic that we presented a few slides back. The other two methods allow us to create components in line as opposed to referencing modules by their names. These techniques can be useful when brainstorming ideas in tests. The first of the two techniques is to create a module via an anonymous function. We can see in this code snippet that this uses Angular's provide service to then create an Angular factory which then contains our movie search logic. The third and final technique is to pass an argument that is an object literal that also creates an Angular service. This is very similar to the anonymous function example we saw previously. But the key difference between the two is the object form creates a service via Angular's provide value, restricting it to the capabilities of the value service. This means that other services cannot be injected when using the object form. The module function accepts multiple arguments in any of the three types we just saw. Let's put this together in our code. Going back to our unit test, we can remove our prototype code and turn it into an Angular module. We can access the ngMock module function via the angular.mock object. We will pass an anonymous object literal argument as we saw in the previous slides. We will add a service to this object called omdbApi which will return an object that contains the search function. We will continue to use our hard coded data as we build upon this, as we work through the course. If we run our tests, we see that we have an error. This is to be expected since we've not yet initialized the service variable. The module function allowed us to setup the configuration, but the next thing to consider is how to get instances of things from this configuration. That's what we'll look at next.

Using the Inject Function

As we saw, the module function allowed us to configure what we wanted to use in the test. But how do we get instances of what we set up? This is where the inject function comes in. NgMock's inject function acts as a wrapper for the angular.injector service. As a reminder, the main angular.injector acts as a service locator for our application. Conceptually, each time we configure a module in our tests, via the module function, they are added to a list to be processed. We can't get at these modules until inject is called. When this happens, ngMock, and then the main Angular module itself, are added to the front of the modules to be loaded. And then they're loaded in sequence. This effectively bootstraps the application allowing us to get instances of anything from the modules that have been loaded by the injector by their names. Using our movie app example, we can get an instance of our new omdbApi service by asking the injector for it by its name. So how do we ask for things by their names from the injector? Let's see how this would work in code. We can call the inject function after the module function, again via the angular.mock object. Calling this function invokes the bootstrapping process we just looked at. We can pass a callback function as an argument. This means that we can pass the argument, omdbApi, and the injector will pass back an instance of this, if it exists. To make this clearer, we should rename the generic service variable to be omdbApi. But now we have a failing test. This is due to a name clash of our variable and the callback argument. The inject function has a solution to this by using a convention of underscore wrapping. We can wrap the argument in underscores and they will be removed when looking at the service by it's name. To contrast the second technique we could use for brainstorming an Angular code, let's use the function argument technique with a module function. Next, we will refactor our prototype code into an Angular service in a separate file.

Creating the OMDB Module and Refactoring

Now that we are comfortable using the module and inject functions we can create the omdb Angular module to contain our logic. We will start by creating a new folder in the source folder called omdb and add a new file for our service logic called services.js. In this file, we can create the omdb module and then add a factory called omdbApi in the standard Angular way. We can then migrate the functionality we prototyped from the tests. Now we can pass the string alias to the mock.module function to reference our new Angular module. And we have a passing test. Now, we can add the second unit test for the lookup by ID functionality. This is very similar to what we already have but we just need to find some example data from the OMDB website. If we run the tests, of course it fails, because we don't have the functionality yet. Let's add this to our new service. And we see a passing test. Reviewing our test code, we can see that we have some duplicate code within the tests. Let's clean that up. We can start by moving the mock.module calls into Jasmine's before each function. This will be performed before each unit test within the same describe block. We can do the same thing with calls to the inject function. And again we have passing tests. The module and inject functions are so commonly called that ngMock makes them both available on the window object. This means that they are global and is therefore optional to use the angular.mock prefix just for these two functions. Finally, if you are using the specrunner.html approach to write the code, your html should resemble this.

Debugging with Dump

NgMock has three public functions available via its Api. Two we've already seen in module and inject. And the final is a function called dump. This method serializes common Angular objects such as scope, elements, functions, et cetera into strings which can be useful for debugging. In the example here, if we print out the standard object to the console we can see that it's not so easy to read. Whereas, if we pass it via the angular.mock.dump function first, it prints it into a much easier to read format. The dump function will neatly print out primitive types, DOM elements, JavaScript objects and Angular scope objects. One thing that can cause confusion for developers when using Karma is Karma also provides a function called dump which is shorthand for console.log that we commonly use when debugging JavaScript code. We can also use this feature in the browser and we can see here in Chrome's console output that this functionality can be very useful in debugging data.

Summary

The aim of this module was to understand how to get instances of Angular components into unit tests. We saw how to do this with ngMock's module and inject functions and understood the bootstrapping process and that we need to use these two functions together in all of our unit tests. We looked at how we could use the function and object argument types of the module function to prototype Angular code in unit tests. And we saw how we could use the dump function for debugging. In learning these concepts, we also wrote the first Angular module for our application. But, at the moment, we still have hard coded data. We will address that in the next module by adding http interactivity.

Unit Testing HTTP Interactions with ngMock

Introduction

In this module, we will add HTTP lookup logic to our OMDb API service. HTTP interactivity in an Angular application requires us to use its $http service. Of course we will do this using a test-driven approach, and therefore see how we can use Angular's $http service in our unit tests.

Adding HTTP Functionality the OMDB Service

In the last module, we used hard-coded data in the search and find functions to make our tests pass. Now we're ready to add the HTTP interactivity to OMDb API logic using Angular's $http service. We will start by injecting the $http service into our factory, and we'll update the search function first, removing our hard-coded data and using the $http.get function. We create a dynamic URL based on our search query, which we will also need to encode to make the search invalid for an HTTP request. The base URL for our requests will be the same, so we will abstract that to a variable. Refer back to the OMDb website

for the base URL, and we will also add the version number for consistency. By adding the HTTP functionality, we've now made that search function asynchronous. And we need to update the return type of the function accordingly using Angular's $q promise service. If we run the tests, as expected, we see a failing test. The expectation fails because the return type is an unresolved promise, and we can see this in the failure message. We need to update our test code to work with the promise return type from our search function. So let's go ahead and do that now. Reviewing our test output shows us that the response variable is undefined, which means that the promise in our search function is not being resolved. This is because the HTTP request in the OMDb API search function is not completing. So how do we make it complete? That's our next task.

Introduction to the $httpBackend Service

ngMock's $httpBackend service is a fake HTTP backend implementation suitable for unit testing applications that use the $http service. To better understand what it does, let's review what happens when we use the $http service in a regular Angular application. In an application that runs in the browser, our code will interact with the $http serivce, which in turn would use Angular's $httpBackendProvider. This does as the name suggests, and acts as a layer of abstraction between the $http service and the HTTP capabilities of the browser that's running the code. The $httpBackendProvider interacts with the browser directly, making our request via XHR or JSONP, which returns a response to the backend provider, which in turn returns the data to our application code, which could be an Angular service or controller. When using the $http service in unit tests, we use the same core service, except for one important difference. ngMock replaces the backend provider with its own version, which does not make real HTTP calls using the browser's capabilities. We have to pre-configure the backend provider in our test code with hard-coded responses to the interactivity that we expect in our test. This explains why the test result previously was not returning any data, because we had not pre-configured the mock backend. This also means that we cannot make direct HTTP calls in our unit tests without using creative hacks or workarounds for this. when and expect are the two main functions for pre-configuring $http responses in ngMock's backend implementation. We can configure $http responses as per the syntax we see here. For a matching URL, we return the 200 status code and some response data. Here is the same example, only this time using the expect function. As you may notice, they are practically identical, which is a source of confusion for many developers. The method signatures are the same, and both support the standard HTTP verbs supported by the $http service. However, it's the behavior that distinguishes the two. The when function is best suited for black-box testing. What determines the output of these types of tests is how we process the response. This makes when more suited for testing with data. The expect function is better for testing if we are using an API correctly, as opposed to how we are processing the data that we set to be returned from the call. The order in which the HTTP fakes a setup, and then calls in the test in the code, matters when using expect. And we can only use each fake HTTP configuration that we set up once. Let's visualize the difference. Here we have the same $http responses set up; the first using when, and the second using expect. If our code on the test calls them in the order that they were set up, as we move through each request, note that the when definitions can be reused, but the expect requests are removed as they are processed.

We see that the behavior of the two differed, but we still have passing tests. Let's repeat this process, only this time, the code on the test calls the mock $http requests in sequence that is different from how they were set up. The test using the when definition still passes, but the expect request will cause the test to fail, as they're expecting the caller to move through each request in the A-B-C sequence in which they were set up. If there are any uncalled expect requests, the test will also fail.

Using the $httpBackend Service

To pick up where we left off, we have the failing test because our HTTP call in the code on the test wasn't completing. Now that we have a grasp of the when and expect functions from the ngMock $httpBackend, we can put them into practice. We can get started by injecting the ngMock $httpBackend instance into our test. We will start with a backend definition with the when function. The first argument is the HTTP verb we wish to target. For example, GET, POST, PUT. In this example, we are setting up a GET request. The next parameter is the URL that we expect. We can pass a string value that must match what URL our code in the test will request. The expect and when functions return an object that has a respond function. We can use this to configure the HTTP status code and the data that our fake request will return. We return a 200 success code in the same example data we've been using. As this operation is asynchronous, we need a way of controlling the flow synchronously for use in our test. This is achieved by the flush function, which resolves all the configured HTTP requests that have been called in our code on the test. We can check the test, and everything looks good. In the fake setup, we passed a string argument to match the URL we expect to be called. In addition to a string, we can also pass a function or regex to match the URL. This can give us greater flexibility when dealing with trickier URLs. For example, here we will use the function to match on the URL. The URL called by the code on the test is passed as an argument to this function, and we can use it to return a boolean value from the function to indicate whether the URL that we're given is what we expected. Here we will check that the start of the URL matches what we expect. If I introduce a typo, we can see that the test fails because now there is no matching fake $http response for the search logic we're coding. We can also use regular expressions for wild card matches, or we can construct more complex expressions such as this. Now we need to add HTTP functionality to the second unit test to look up the movie by ID. The logic is similar to what we have already coded, but with a different URL and return data. To contrast the two functions, we will use the expect function in this example. Of course our test fails because we've yet to add this logic, so let's add enough to make our test pass. The logic again is similar, so we can copy what we did for the search function and tweak the URL. Great, our test passes, and now is also a good time to refactor the code we just wrote to keep it nice and dry, and remove the duplicate code. The final thing we need to do is handle potential HTTP errors from the API. In this case, we return an $http response code that corresponds to an error such as a 404 or a 500, and handle that accordingly in the test. We see that the test fails, and now we need to add the error-handling code to our service in order to make the test pass. And now we have a passing test.

Unit Testing $Resource with the $httpBackend

The homepage of our application has a carousel that cycles through popular movies. This data is powered by server-side logic of the application. Administrators of the application need to manage this data, which is a list of movie IDs that correspond to the OMDb data. We need to add the logic to be able to use this API to our application. Create a new folder called movie-core, and add a file called service.js. We're going to use an Angular resource, as this will provide the simple CRUD mechanism that we need to manage the popular movies data. Here's our starting point for the resource. As you can see, the URL is configured for the endpoint /popular. The resource also has been extended so that the update function is made via a PUT request. To use the resource module, we need to include it manually, since it is separate from the main Angular include. First, we will update the karma config file, and remember that you'll need to update the spec-runner.html file if you're not using karma. Then we need to download the file. We need to add a test for this, so let's do so in a folder called movie-core in the spec folder, and add a new file called service.spec.js. The starting point for our test is similar to what we've already seen, but note that we are injecting an instance of our PopularMovies resource. The first test will be for creating a new popular movie record. Here, we define the schemer for the data, the movieId, and the description. We then call the save function, which will trigger an HTTP post to save the record. If we run the test for what we have so far, they pass, but what we want to test, for the time being, we just want to verify that we've set up the Angular resource correctly. By correctly, we mean that the HTTP calls where the URL properties that we expect are being called. Given that we haven't set up any fake HTTP calls yet, we should expect this test to fail. A feature of the mock $httpBackend is the veryNoOutstandingExpectation function. If we include this via a Jasmine after each block, it will be called after each unit test. If we check the tests, they are now failing. The call to verifyNoOustandingExpectation has thrown an unexpected request error. This is because an HTTP request was made via the call to the save function of the resource, but nothing had been configured for this. Let's go ahead and set up this expectation. This time, we will use the POST action. In the first example, we passed the HTTP action as a string value. This time, we will use the shorthand methods that when and expect both have. Behind the scenes, they call the same function, but as they are so common, the shorthand form saves us extra typing as well as being more readable. Since this is a POST, we also need to consider the POST data. As we saw with the URLs, we can pass multiple types to match the expected POST data. We will start with the function form. Here, I will use ngMock's dump function to show the actual data in the console. The expectation of our test is that no exceptions will be thrown when we call the flush function, which we can do via Jasmine's toThrow matcher. A handy tip is to make use of Angular's fromJson function to confer the JSON to a JavaScript object, which we can then use to return a boolean value that determines whether the properties of the object match what we expect. Here, we are checking that the movieId property of the object matches the value we expect. It does, and our test passes. We can also pass JavaScript objects, but note that these must match exactly. We can also pass a string value, which again must match exactly on all the properties. And lastly, we can use a regular expression. In this example, the regular expression states that the movieId must match a specific value, but the description can be anything. I will update the description to demonstrate this. And as we see, the test still passes. Next, we will add a test for the findById functionality so that we can check the URL is configured correctly. We're expecting the format of popular/movieId. The first step is to add the

call to the resource's get function, passing the movieId value. The test fails, of course, so we need to set up with a fake GET request using the expect function. To match on the URL, we will use the function type so that we can use the argument passed to the function to double-check the output in the console. This gives us another chance to show how useful these function types can be in debugging when working through the tests. As we can see in the console, the URL represents what we expect, so let's update our test with this value. The final configuration we need to check is that the PUT method for the update function is correct. We can reuse what we've already done, only this time for a PUT request. So let's copy and paste the code, and change it to use a PUT request by calling the update function. The test fails, as expected, and then we can set up a fake $http response of the PUT request. And we have a passing test.

Test the $httpBackend Service with HTTP Headers

The PopularMovie API we just used will require an authentication token so that the endpoint can be secured. The resource code has been updated to reflect this. Each of the methods have been configured to append an authentication token to the header. For now, we will hard-code the key value so that we can refactor it later to something different. As we did in the existing unit test for this resource, we want to write some unit tests that verify we've set up the resource correctly to send an authentication token with every request. We will start with a test for the resource call to get a popular movie by its ID. Only this time, we need to verify that the header information is what we expect. As with the URL and POST data, there is more than one way of doing this. In fact, there are two types for this: a function and an object type. We will use the function in the first example, following the same approach for the URL and the POST data. We will again make use of the dump function to get visibility of what's being generated for the HTTP request being made in the code on the test. Then we need to set up the fake HTTP GET request using expectGET, using the expectedHeaders function we just created. By adding the call to flush and then running the tests, we can see that our dump outputs. And note how we can also see the except type in the debug output. We're really only interested in checking the auth token, so we can again use the angular.fromJson function to convert our JSON to an object to allow us to check the specific property value. The second type is to use a JavaScript object, which we can add next. Note that the object properties must all match exactly, such as the except headers. This approach can therefore be more restrictive than using the function, where we can select which properties we want to match against. In the interest of time, here are the completed tests for checking that each request supported by the resource sent the auth token with the header. Note the usage of when and expect. We have two GET requests being made in our code on the test, so when has been used for reuse. The rest are one-off requests, so expect has been used for precision. One thing that you may also notice is that the flush function is being called with a number argument. This is an optional parameter we can use to give us control of how many pending HTTP requests we wish to process. Passing no arguments, which we have done to date, processes everything. This test passes, but there is a deliberate mistake in the test code. Can you spot it? I'll give you a clue: how many fake HTTP requests have been set up, and how many have been made in the code on the test? I'll give another clue by adding a call to the verifyNoOutstandingRequest function. Now we can check the test output for clues. Aha, we see an

error, that there is one unflushed request. The call to verifyNoOutstandingRequest checks that there are no requests that we set up, and we called in our code on the test, but we forgot to flush. It's good practice to include both of these functions as we see here, since it acts as a safety net for mistakes that we may have made in our code on the test. There's also a function we've not used in this demo called resetExpectations, which does as the name suggests, and restores expect functions that have been set up for reuse.

Summary

In this module, we added HTTP functionality to our application, and in doing so, we saw how ngMock's $httpBackend allows us to set up fake HTTP requests and responses in our unit tests. We used pretty much all of its features in our unit tests, for each of the HTTP verbs. We used both the expect and when functions and contrasted the difference between the two, and we took a deep dive into the techniques that are available for matching on URLs, POST data, and headers. So far with our app, we've built a solid backend to power our data needs. Our next steps will be to start using our logic in controllers.

Unit Testing Controllers with ngMock

Introduction

Hi this is Bradley, in this module we're going to focus on unit testing our Angular controllers. In the last few modules we started building out the API to power the data needs of the application. But up until this point we don't have anything visual. In this module we're going to add the controller module to our app, starting with the logic for the search control. Once we have this in place we can look to add HTML to the application and it will really start to take shape. The main question we will be answering in this module is how do you use Angular's controller service in our unit tests?

Introduction to the $controller Service

Before we dive into our demo let's first recap AngularJS controllers and how they would look in an application. The snippet we see here creates a simple controller called GreetingController that is attached to the myApp module. When we call the controller function on the module object, we are calling the controller provider function, passing the details to register our controller. Such as the name and of course the function body for the controller itself. Then in the view of our Angular app, we would use the ngController directive to link our controller to the relevant block of HTML. This would also take care of creating an instance of a controller and wiring up the necessary elements using Angular's controller service to get an instance of the controller. Based on what we've seen so far with ngMock, it would be logical to assume that we could do something similar to the code we see here in order to get an instance of our controller for our unit tests. We would access the instance of a controller by its name

in the same way we did with our OMDb API service in earlier modules. But unfortunately this won't work, as controllers work in a different way. We need to get an instance of the controller via ngMock's controller service. The controller service from Angular is wrought with ngMock's own version. Which acts as a decorator for the controller service with an additional bindings parameter that's useful when testing controllers of directives that use binds to controller. ngMock's controller function signature accepts three arguments. First is the constructor argument, which can be either the name of a controller which is a string type or a function that creates a new controller. In this snippet here the first example using the string value will find an instance of the GreetingController we saw a few slides ago. The second example allows us to create a controller inline but passing a function. The locals argument is an object type that maps by key names to the arguments of the controller constructor function. This would commonly be the scope object, an http service or some other service. Referring again back to our controller example, this allows us to set the parameters that are passed to the function. The binding argument is optional and accepts an object. Where the values of an object, such as properties or functions, will be bound to the controller's this binding. We can see how this would work as demonstrated in a controller code here. Notice how the property past has a binding is accessible by the controller's this object. So that's the theory and now we're ready to put this into action and progress the movie application we've been building.

Unit Testing a Controller

We're going to start with our logic for the search controller. It will handle the incoming query from the user and redirect to the search results controller if there is a search value. As we've done this far we will start with a test. So first add a new folder called Movie App to the spec folder. And create a new file in the folder called Search.controller.spec.js. We will call the spec search controller for the test name. And we will state what the functionality should do which is to redirect to the query results page for our non-MD query. Let's start by brainstorming the functionality. Since we're going to be using our controller, we know that we will need a scope object. We will probably want to set up a binding for the query parameter, which we can set here as a simple string value. Our UI will have a search button, so that when it is clicked or the search form is submitted we will be looking to call a search function. If all is well so far our controller should then redirect to the results page which will have its own controller. We can check this in our test by verifying the status of the URL. In Angular we would use a location service for this. So we will add a location variable to represent this. We will start up the tests which will of course fail and now we can write a little more code to brainstorm our idea by adding logic for the search function. To make our test pass we need to mimic the search controller redirecting to the results page without a query string parameter. Which we can do by setting the URL property of the location object. Keep in mind that this is not exact Angular code which is fleshing out the functionality. Our next step will be to add the second unit test which will be to not do anything if the query value is empty. This would be the case where the Angular user is trying to search for an empty value. In this scenario the URL would not change. So we just need to verify that. We have some shared logic for our two tests which we can move into a before each block. We then need to set up the default states of the location service so that we can verify in the second test that it does not get changed. Now we can add the logic to satisfy our

second test case which is a simple if statement to check the query value. A quick check of our tests tells us that everything went well. Now that we have a sense of what we wish to test let's see how we can evolve our code into Angular code. When we talked about ngMock's controller service, we saw that the first argument it accepts, called the constructor parameter can be a string or a function. Using the function allows us to create controllers inline in our tests which can be especially useful for prototyping ideas. If we want to try this out in our test, we first need to inject the controller service via the inject function. Our first step is to extract our current logic into a function that we can pass as a constructor value to ngMock's controller service function. We can then pass our function as the first argument to the controller service. Our controller function uses the scope object so we need to pass that as the second argument, which is called the locals argument. To represent local variables required by the controller. The sharp eyed viewer would also noticed that we need an instance of Angular's location service. We could mock this if we wanted to. But as a rule of thumb I always use the native Angular services where possible to cut down on codes to mock objects which can clutter tests. We can get an instance of the location service via the injector and add it to our locals object. We will also store the location service that's a local variable instance, since we will need to use it in the insertion for both of our tests. In our prototype code we weren't using the location service API precisely. So let's go ahead and update that so it's using it correctly. Now we can check our tests and everything looks good. In our tests we passed the instance of the location service obtained by the injector to the locals argument. It's worth noting that this is optional, since both the controller and the location services come from the same injector. Therefore any instances of Angular services required by the controller will be resolved when the controller is created. This means that if we didn't pass the location service as we're doing here the injector would pass the same instance to the controller. Which is effectively the same thing. As a personal preference, I pass the service in the test code as I find it more effectively communicates what the controller needs in the test. Now that we're happy with our prototype code, we can move it into its own Angular module. In the source folder, create a new folder called Movie App and add a file within called search.controller.js. We can migrate our test code into the new Angular module called Movie App. And we will name the controller search controller. And our final step will be to upgrade our test to include our new module and we can now use the string type constructor arguments to reference our new controller by its name. Now we have a grasp of how to use the constructor and the locals argument, we can see how we could use the bindings argument. This is especially useful if using the controller as approach in controllers. If you're unfamiliar with this concept, in short it means that the controllers this binding can be used instead of the scope object to act as the glue between the controller and the views that interact with it. If we update our controller code to use this approach, we would simply remove our reference to the scope object and use the this binding instead. But of course our test will now fail. Here's the updated code, it's very similar to what we had before only we are passing the values we wish to set on the this binding via the third argument to the controller service. And did not on the scope object via the locals argument. Whether you should use the scope approach or the this binding is a discussion for another course, but given that I've most commonly seen the scope object approach used in the wild, that's the approach we will continue to use in this course. We covered quite a lot there so let's recap what we just did. In using ngMock's controller service we saw how to use the first argument, called the constructor argument, that could be a function to represent an inline controller for brainstorming, and we saw how to use the string type to reference an existing controller by its name.

The second argument, an object type called locals, is used to path the scope object and other services required by the controller we wish to test. And the third argument which is also an object allows us to set values on the controllers this binding should we wish to use the controller as approach for our controllers.

Unit Testing with Promises

We just wrote the logic to redirect to the search results page with our search query so the next step is to write the controller logic that will power the results page, loading the movie data based on the query. As we can see on screen, once we are redirected to the results page we have a collapsed list of results. Eventually we will be able to click on each row and it will expand to show more movie data. But as a starting point we will just list out the data on the screen. So let's get coding. Add a new file to the spec movie app folder called results.controller.spec.js. Here's the basic setup for the results controller test. I've included some example data to represent the small sample of what data we will get back from the real API call when we perform a search. We will use this data for the assertions in our tests. The test will verify that the expected movie data will be loaded and set to a results array on the controller scope object. We can use the test data we set up for this, checking the title values to ensure that the data is listed in the prder that we are expecting. In the expect calls we are checking that the results array on the scope object of the controller matches the order of our test data that we assigned to the variable called results. We will start the tests which we will expect to fail. So our next task is to add enough logic to make the test pass. In the source movie app folder, create a new file called results.controller.js. We can copy most of what we did for the search controller and update the controller name to be called results controller. Since we're now adding both of the controllers to the same movie app module we need to move the code that creates the module into its own file called app js. Switching back to our new results controller, we can set up the results array on the scope object. With the same sample data from the test just to make the test pass. Our test passes so next we look to use our OMBd service in the controller. We will use the search function we coded a couple of modules ago. To remind ourselves of the API we will revisit the OMDb service spec we wrote a few modules ago to see how the search function was used in our test. We can use the same function call in the controller. We just need to update how the data returned by the function is set to the results array on the scope object to meet our requirements for this test. Our test will now fail since we still need to include the OMDb module in our tests so injector can find an instance of the OMDb API service. But our test still fails. Our result property on the scope object is not being set. Why? This is because the promise returned by the call to the OMDb service's search function isn't being resolved. We could use the approach we use for the OMDb API tests which was to use ngMock's http back-in service to resolve the http call that we know the OMDb API search function will make. I've seen this approach used a lot. But as we're writing unit tests, we should mock the OMDb API service directly as if we used the http back end approach while testing the internals of the OMDb API search function again in addition to our results controller. But this leaves us with another question. How do we mock Angular promises in our tests? The answer is that we need to use Angular's Q promise service in conjunction with Jasmine's mocking features. So let's do that next. We need to include Angular's Q and root scope services in addition to the OMDb API service we want to mock. We can use

the inject function to get instances of these in the usual way. We can use the Q service in combination with Jasmine and our OMDb API instance in order to mock the call to the search function and resolve the promise when we want to. Jasmine's spy on function enables us to specify what we wish to mock by passing the object itself as a first argument and the name of the function we wish to target as the second argument. By adding the syntax and.callfake, we're doing what the name suggests, that when our controller and the tests calls the search function on the OMDB API object, we will call this fake function instead, allowing us to control the execution flow and not concern ourselves with inner workings of the real search function in the OMDb service. Now that we have a way of mocking the search function, we need to make it resolve the promise. This is where the Q service comes into play. Within our fake function we need to return the promise by using the defer function from the Q service to create an instance of a deferred object. We can call the resolve function on this object which will resolve our search function with the data that we passed. We will use the same hard coded results data from the test. The return type will be the promise which provides the dot then syntax we used in our controller code. So everything is in place with our mock code. And our test should now pass. But it doesn't. You may have noticed that we haven't used the root scope service yet. There is a quirk in that we need to call the apply function on the root scope service in order to resolve the promise. This pulls us into some inner workings of Angular and in this example, when we call resolve on our deferred object we added our promise to a list of resolve promises ready to be processed by Angular's own event cycle. In our applications that run in the browser, Angular takes care of this for us behind the scenes. But for our tests we have to control when we want this to happen sometimes. By calling root scope dot apply, we are triggering this event cycle. Which will pick up our resolve promise and finish off the final part of the process which then means that our then function block will get called. Finally we need to update our assertion logic since we were initially setting the dot data property in our first iteration so we will have refracted that away. And our tests are passing. We need to check that our search function is being called with the correct argument. Our code currently will resolve the promise regardless of the query. We can use Jasmine's to have been called matcher to check in our tests that the search function is being called with the value we expect. If I go in and change this in the controller code so that it doesn't match we see that the test now fails. As we saw when we demonstrated the application, the search query is passed by the query string parameter so we need to add the logic to read this value to our test. We need to use the same location service we used in our last test, so let's add that to our controller code. And as before we need to bring the location service into our test so that we can set up the correct query string value. We can use the location service's search function to set a parameter called Q with a search string Star Wars. Verifying our change we can see that our tests are looking good. We can also add a second unit test to catch cases where there has been an error in the search function. The code we see here is similar to what we did for the first test. Only this time we are calling reject on the deferred object. We need to update our controller code to handle this case by setting the necessary error message in the catch block. This catch block will be called in the event of the promise being rejected. Now we're in the position to add HTML for the application which we will look at in the next clip.

Adding HTML to the Application

We have the controller logic in place for the search and result controller, so the next step is to add the HTML elements. In the source folder create a new file called index.html that will be the default page. We're going to make use of the Angular version of Twitter Bootstrap. As we can see from the HTML shell here, we've included the Bootstrap CSS in the document header. And the page is made up of three components. The header section. The main container section. And the footer. We also have the Angular and the Bootstrap JavaScript includes at the bottom. Expanding the header section, we can see it has some basic page header details. Such as the title and the strap line. And the footer has some basic accreditation. We need to serve our files from a local server. And thus I've been using no JS to run the tests from the command line, I will install the express framework so that we can serve static files from local host. Windows users could set up an application via IS if you're not using no JS. Once express is installed I will add the basic template code to an index JS file that will start a server and make the static files for our application publicly available. We can then start the server and examine what we have in the browser. We need to add the ngApp directive to initialize our movie app. And update our movie app module setup code to use Twitter Bootstrap. Viewing the page in the browser shows that we have the basic shell up and running. We can add the HTML for the search controller by first creating the containing div and using the ngController directive to reference our search controller. The HTML for the form has the relevant Bootstrap CSS properties to start it nicely. The key points for this markup bar, the reference to the controller search function, via the ngSubmit directive. This ensures that the function gets called when the user hits enter to submit the form. We've also included this on a click event of the search button. We've used the ngModel directive to bind the query property to the input field we used on the scope object in our tests. Refreshing the browser shows the updated page. But if we type a query and submit we don't get redirected anywhere yet because we haven't included our Angular code. We're going to use Angular's rooting so we need to add the ngView directive within our container div to place the controller's content. The JavaScript includes need to be updated. First we need to add Angular's rooter. Then the include files for our movie app, the module's controllers and the OMDb service. Since we are using rooting, the results page needs its own template. So let's go ahead and add an HTML file called results.html to the movie app folder. The roll, expand and collapse functionality we saw will come from Twitter Bootstrap, which we see here, called an accordion. The accordion uses the ngRepeat directive to iterate over the movies and the results array on the scope object. We will display the movie title initially. The other properties are for the row expand collapse data that we don't need at this stage but are in place for later enhancements. The final piece of the puzzle is to link up our new template and results controller. To bring everything together we need to update our my app module creation code to know about ngRoots and OMDb service. Then we can add the setup for the rooter so that it links the results controller and template so the results URL our search controller is going to redirect to. If we refresh the browser and search, we see movie data. It's worth noting that since we unit tested everything to get this stage, once we do include the logic into our app, everything just works. Before we wrap up and move onto the next task, we should check our tests. Oh dear. We see some very unreadable error messages. So what's gone wrong? We updated the my app module to use Twitter Bootstrap and Angular rooter which the test configuration doesn't know about. To remind ourselves of what these values are, let's switch back to our new index.html page. We need to download these files to our lib folder as we did at the start of our course. First let's download the Angular rooter and then Twitter Bootstrap. We need to update the karma convict file to reference these two new library

includes. And upon rechecking the tests, things are looking a little better. As a quick reference for those still using the spec runner dot HTML approach. Your file includes should resemble this.

Summary

In this module we looked at how to use Angular's controller service in our unit tests to get instances of controllers that we wish to target in tests. We looked at the controller service and its three function arguments and used each of them in working through our application. We looked at some of the trickier cases of unit testing promises in controllers and how to resolve them and reject them for our unit tests. And we also added the first HTML elements to our movie app which is beginning to take shape nicely. Now we're in a good spot to keep on adding more features to our movie application in the next modules.

Unit Testing with Timeouts and Intervals

Introduction

Hi, this is Bradley. In this module, we're going to look at how we can unit test timeouts and intervals in Angular, using the timeout, and interval service, respectively. We will build upon the visual features we added in the last module. Our first enhancement will be to add functionality similar to auto complete for our search input field, where after a one second pause in typing, movie data will be returned. The second will be to add the rotating movie data that we see on the homepage. Every five seconds, the movie title will be rotated. In order to achieve this, we need to use the timeout and interval services. In order to unit test these, we need to understand how we can unit test the timeout and invariable services in Angular JS, using ngMock.

Using ngMocks $timeout Service

We will need to use Angular's timeout service for the feature that is almost like an auto complete for the movie query. Once the user starts typing, if there is a one second pause of no typing, the application will try to show movie data to the user. To refresh our knowledge of the Angular timeout service, it's Angular's wrapper for the window.setTimeout function. The function signature of the service is as follows. The first parameter is the function that will be executed after the set period of time. The second parameter is the delay, in milliseconds, where the default is zero. When the value is zero of course, the function will be executed immediately. The invoke argument can be a Boolean, and is set to true by default. If this is set to false, it skips model dirty checking, which means that the apply function doesn't get called within our delayed function block. The side effect of this would be that, if we were setting a value on a scope object in the controller, that we see in the snippet here, it would not necessarily be updated in the UI. The pass argument is also optional, and allows us to pass additional parameters to

the executed function, as we can see in this snippet. We should also remind ourselves of how we can cancel a timeout, should we wish to revoke an instance once it has been started. Trying to unit test code that uses timeouts can be difficult. The function is asynchronous, and of course, unit testing asynchronous code can be tricky, when components of the code in the test can complete at varying intervals. The second problem is that the delay value can often be several seconds or more, such as 10 seconds. Not only is the fact that this function is asynchronous makes it difficult to test, if we had several unit tests that had 10 second delays, our test could be slowed down significantly. Fortunately, there's a solution to this problem. NgMock has its own timeout service that acts as a decorator for Angular JS's timeout service, for the context of unit tests. The timer and decorator service in ngMock retains a standard method available in Angular JS's implementation, with the addition of two new methods that enable us to test code that uses the service. The first is the flush function. It flushes, and executes any timeouts that are set in the code under test. We can optionally provide a delay in milliseconds, to take us to a certain point in time. This gives a synchronous control over the asynchronous operation, which means we do not have to wait for the actual time delay set up for the timer we are testing. The second function is called verifyNoPendingTasks. This will throw an exception if there are any outstanding timeouts that were not flushed in our code under test. So, that's enough theory to refresh our memory about Angular's timeout service, and understand how ngMock's timeout division decorates it, adding features to make it easier to unit test. Now, we are ready to write some code.

Demo Using ngMocks $timeout Service

We will navigate to the spec folder for the search controller, and add a new unit test. The name of the test will state that the user should be redirected to the results page after one second of inactivity. We will set up a query property on the scope object, as we did before, and we need to trigger the check after each keyUp event. We will add a keyUp function to our scope option for this. The expectation for the test is that the controller will redirect with the expected query value after one second of an activity. Let's switch to the controller, and think about how we could code this functionally. The first thing to do is add the keyUp function. It's within this function that we need to set the timeout. We need to have the timeout injected into our controller function, so let's add the time out service to our constructor arguments. Now we can use Angular's timeout service. The first argument is the function to be called, that will invoke our search function, and we will set up a delay of one second, in milliseconds. For the purposes of our demo, we're going to break the TDD rules slightly, and check this in the browser first. We need to wire up the event, so that it calls our keyUp function, which we can add to the search input field, in our index HTML page, via the NG keyUp directive. Now we can start up the node server, and check what happens in the browser. We can see that, if I stop typing after a partial query is entered, we see the results automatically. So, switching back to our tests, how do we go about unit testing this? Well, our first step is to start the tests. The test fails, because the redirect didn't happen, which indicates that the timeout didn't trigger. We need to add ngMock's timeout service to our test, so that we can use it for the testing question. As we've been doing in our controller tests so far, we will add the service to the locals object, passed to the controller service. As we discussed in the last clip, the flush function is added to ngMock's timeout service, so let's call that in our test, so that our timeout will be triggered.

Checking the test, we can see that it passes. Calling flush caused our timeout to happen, so the location service was updated accordingly. In calling flush, we didn't pass a parameter. The default behavior is to complete all the timeouts that are set up in the code in the test. To see how we could control multiple timeouts in the same function, let's add a delay value in milliseconds of half a second. We can see that the test fails, and this makes sense, since we set the timeout delay in the controller to be one second, but we only flushed the timeouts for up to half a second. The second function on ngMock's timeout service was the verifyNoPendingTasks function. By passing the 500 millisecond value to the flush argument, I made a mistake in my test set up. The verifyNoPendingTasks function is useful in checking for these types of mistakes. By adding this, and checking the test, we can see that we have a more useful error message in the test output, telling me that I have an uncompleted timeout in my code and the test. We can remove the argument, and re-factor our code a little, by wrapping the call to verifyNoPendingTasks in Jasmine's not toThrow matcher, and removing the redundant function to the search function in our timeout code. There is a floor in what we just added, in that once the keypress happens, the search function still gets called, even if we continue typing. We want to prevent this, so we need to cancel the timeout if another key is pressed within one second limit. We want to be able to cancel the timeout service once the user presses the next key, which we can trigger by adding a keydown event. In the keydown event, we want to cancel the timeout. We can use the verify no pending tasks function again to verify that there are no running timeout functions once the keydown function is called, after the keyUp function. We need to add the keydown function to the controller. The test output confirms that we are not canceling the timeout, as indicated in the test failure message. Let's update our controller code to call the cancel function on the timeout service correctly. We will also need to cancel the timeout once the search form is submitted, which is very similar to what we just started. We can copy and paste the test code, call the search function, and then update the controller code to cancel the timeout. Finally, we need to update the search control HTML in the index dot HTML page, to call our keydown function via Angular's ngKeydown directive. If we review the application in the browser, we can see that the functionality is working as expected.

Using ngMocks $interval Service

To add the rotating image on the homepage, we need to use Angular's interval service. At each five second interval, we need to call the IMDb API service to get the movie data for the given movie ID, and update the UI. This will happen indefinitely. We need to use Angular's interval service for this, which is Angular's wrapper for the window dot set function. The function signature of the individual service is as follows. The first argument is the function we wish to execute at the set interval. It will be called at each interval, which we can set via an argument, which is the delay in milliseconds between each execution of our function. The third argument, the count value, allows us to optionally supply a number indicating how many times we wish to repeat the interval until it is stopped. If we do not supply a value, or pass an argument of zero, it will run indefinitely, or until we cancel the interval via the cancel method. The invoke apply argument is the same as with the timeout service. The default value is true. If we set this to false, any changes made to the scope object within the function will not be immediately visible. The pass argument is also the same in principle as we saw with the timeout service, which allowed us to pass

arguments for the function to be executed. We can also cancel an interval instance via the cancel function, which works in the same way as with the timeouts, and again, this function is asynchronous, like the timeout service, which can be complicated to test. We have the same problem of handling the delay in milliseconds between each function execution, and since an interval can happen N number of times, we have the problem of advancing through time in our tests to verify our application's behavior. Fortunately, ngMock provides us with a solution to this problem, by providing its own interval service. In contrast to the timeout service, which was a simple decorator for Angular's service, ngMock's interval service is a complete mock implementation of the interval service. In addition to the functionality we already looked at for the service, ngMock as the flush function, which executes any interval tasks that are set up in the code in the test. We can optionally provide a delay to move forward to a point in time by milliseconds, and trigger any functions scheduled to run in that time. That's the theory. Now, let's put what we've discussed into practice.

Demo Using ngMocks $interval Service

The logic for the rotating movie detail will feature on the homepage, so it makes sense to place this logic in the controller for the homepage. We will start by adding a new file in the movie APP specs folder, home dot controller dot spec dot JS. The name of the test will be for the home controller, which is a new controller we are going to flash out as we work through some tests. Here's the starting point for the test. At the top, we have a variable, called results, which is an array that represents a list of popular movies that will control which movies will be displayed on the homepage. The intention is that this data will come from the popular movie service we developed a few modules back. That service will return a list of movie Ids that match data from the IMDb service. Now that we have an idea of the data, we can add the first test. This test will verify that we are rotating through the movie data in the correct sequence. We want to check that we start with the first movie, and since we have three movies set up, after two five second intervals, we should return back to the movie data we began with. That's what we want to happen, now we need to look at making it happen. Add a new file in the source movie app folder, called home dot controller dot JS. We need to add the standard controller set up code, to create the controller, and attach it to our movie app module. Let's start up the test, which will fail, and we can now turn our focus to making the test pass. We will copy across the results variable from the test, so that we can set the data we expect to satisfy the test. We will add a default value to satisfy the first expectation. To satisfy the next two expectations, we need to add the interval service to our controller. Once we've done this, we can set up the five second intervals that will change the movie data as we move through each cycle. To step out this functionality, we will track the index position by using a counter, and the modulus operator to look up a value from our results array for the correct index. We then, of course, need to add the five second delay, and now, every five seconds, the counter will be incremented, and we will set the result valuable to one of the three items in our results array. Switching back to our test, we need to set up moving through the interval service we just added to the controller. We need to add ngMock's version of the interval service, so that we can use its flush function. Once you've added this to the test, we can call flush, with five second intervals, to move through the execution in a synchronous manner. Let's fire up the test, and check everything passes. It does. Now we

need to remove the hardcoded data we used in home controller, and make the test pass with the a real API calls. We need to inject our IMDb API, and popular movie services we created in earlier modules. We need the IMDb API service for its find function, so that we can look up movie data by its ID, and we need the popular movie resource we created to return the list of movie Ids that we should use to look up the data from the IMDb API service. First, let's add a function that encapsulates the IMDb API find function. Next, we need to call the get function on the popular movies resource. It will return a list of Ids in the format we see in the comment added here. Once this resolves, we can use this list of Ids. Firstly, to set the default movie data, and then to call our final movie function at the given intervals. The flow is the same as before, only we've swapped out the hard coded data. Adding these services to our test means that we now need to update our test code to accommodate these two new services our controller code requires. As we did in the test for the results controller, we need to mock these services, since we don't wish to make real HTTP calls in our tests. Let's start out by mocking the get function on the popular movies resource. We will use the same Jasmine functionality we already covered, which is to use a spyOn function to set up a fake response for the function call, and resolve the promise to return the same list of movie IDs we've been using so far. The second thing we need to mock is the find function on the IMDb API service. This code is a little more complex, since we have to link up the data result in the promise to the Ids that we are returning from the popular movies mock. We are using more functionality from Jasmine here to find out what argument is passed to the find function in the controller code, which we are using to base what data we will return. We are, again, using the results array to return matching data based on the array index, which is very similar to what we did in our step code. Using the Jasmine calls dot most recent function means we can find out which of our predefined movie Ids we used to call the find function, which we can use to return the matching data, thus completing our testing cycle of checking that each interval is calling with the ID we expect. If we don't see an ID that we recognize, we will reject the promise, which will fail our test, indicating that we've broken something in the test set up. Finally, we need to set up the constructor with the correct locals argument. We need to add the interval service, along with the two new services. Since we are resolving promises in our test, we need to include the root scope service as we did before, so that we can call it's supply function. We covered a lot there, so let's recap what we did. We mocked the popular movies get function to return a list of movie Ids that should rotate on the homepage. Then, we mocked the IMDb API find function that will return the full movie data for a given ID, and we made use of Jasmine's functionality to check what parameter the function was called with, so that we could return the matching movie data. We then extended the locals argument we're passing to the controllers constructor with the new services we've mocked. Before we check the tests, we need to add the movie core module to our movie app module, so they can find the popular movies resource, which is located in the movie core module. Let's, again, recheck our tests, to make sure the mock code is correct, and everything is passing. That's it for the tests. Now, let's make the necessary tweaks to the front end. We will start by adding a new home dot HTML file in the movie app folder. At this stage, we'll just output the movie title. We will display full movie data in the next module, in fact. Then, we need to update the movie at module, linking up the template in the home controller. Finally, we need to add our new controller, and popular movies service to the index page's JavaScript includes, and also, Angular's resource include, as we're going to be using the popular movies service, which relies on this. We are ready to fire up the web application, but we have one snag, we don't have any server-side logic yet,

which we need for the popular movies get function. To circumvent this problem, we will temporarily add hardcoded data in the home controller. We will look at a way to remove this without having any server-side logic in a later model. If we fire up the application, we can see that the movie title is being set from the IMDb API service call at five second intervals, which is exactly what we wanted.

Summary

In this module, we saw how to use ngMock's timeout and interval services to unit test features that use Angular's timeout and interval in our application. In doing so, we looked at all of the features of the Timeout service, and did the same for the interval service. In particular, we made use of the flush function to control the services in our tests. We saw how the features added by ngMock made these asynchronous functions synchronous, and therefore much easier to test.

Unit Testing Directives with $rootScope

Introduction

Hi, this is Bradley. In this module we're going to discuss unit testing directives in ngMock, which comes hand in hand with understanding how to use Angular's $rootScope service in our unit tests. As we've done in all our modules to date, we will add the new feature to our movie app, which in this module we'll be taking our movie data from the simple text we currently have on the homepage to a more visually appealing layout as we see here. More formally, this means we're going to explore how to unit test in Angular Directive, and we will discuss when we should use the $rootScope service in our unit tests. Let's get to it.

Unit Testing a Directive

In the last module, we added rotating movie data to the homepage, but the display was limited to simple text. I commented that we would add more movie data to this. Let's pick up where we left off and enhance this. We're going to create a directive to display movie data that we can reuse on the homepage and on the search results page. To give us a sense of how this will work I will update the home.HTML template. The new directive we are going to create will be an element called movie results, so let's add this. We will set the movie data on the directive via a property called result. Of course, this directive doesn't exist, so let's create it. Our starting point is, of course, with a test, so I'm going to add a new test in the spec's movie app folder called movie-results.directive.spec.js. We will add the spec called movie result directive and name the test, which will be to check that our movie data produces the correct HTML output. As a starting point, I will paste in some sample data to lay the foundation of our initial tests. We will think about what the expectation for this test should be. We want to check that the generated HTML matches the market we will define in our directive and is populated with the data we

expect. Here I'm just wrapping the movie title in a simple div to get started. The followup question is, what do we need to add so that we can go from our new directive to generated HTML? The first thing we need to include is Angular's compile service. This service isn't specific to ngMock, so we won't go into depth with this, but to recap on the basics this service compiles an HTML string or DOM into a template and produces a template function, which can then be used to link scope and the template together. We will include the compile service so that we can use it in our test. This, of course, requires us to add a beforeEach block and use ngMock's inject function. Once we have the compile instance, we can call it as a function passing our movie results element, so we expect it to be compiled into the target HTML. We set the movie data on the element via a scope property called result, which in this example will also be called result. We then need to pass a scope object to the function returned by the compile service constructor, which we've initialized in a test and set the property on the object called results to our result data at the top of the test code. The next logical step is to begin creating a directive. In our movie app source folder, I'm going to create a new file called movie-result.directive.js. Our movie result directive will be attached to our movie app module and that will set the default properties of the directive starting with restricting as directed to be an element, setting the replace property to true so that the element is replaced with HTML, setting up a scope object with a result property configured, and we will add a very simple starting point for the template of just a simple div that wraps the movie title. Before we start the test, we need to include the movie app module for using the test. We can see the test fails, but we see an unusual error message. What does the reference to $new mean? It implies that the directive hasn't found a function called new on our scope object. To correct this, we need to use a proper Angular scope object and not a native JavaScript object. We can do this by bringing in the $rootScope service. We can replace our standard JavaScript object that represents our scope. We will introduce a new variable called element to make our test code more readable, since the return tag of this line gives us an element and not HTML just yet. Next we need to call the digest function on the $rootScope service, otherwise the template won't be compiled against our scope data, since this happens as part of Angular's digest cycle. Then we can call the HTML function on the element to get the HTML contents of the element. If we review the test output it shows that we are processing the data correctly, but our HTML is not quite what we expected. If we write the outer HTML of the element to the console we can see that the div that wraps the content is there, albeit with some dynamically added attributes. But when we call the HTML function on the element, we are getting the contents of the div. Let's switch to the directive code and add an additional pair and div tag so that the call to the HTML function matches our test output. At the same time, I'm updating the template to be a string array, which I find easier to read as templates become busier with more HTML. Our test still fails as we overlook the fact that Angular appends the div tag with an ng binding class property. We can tweak our test expectations slightly to match this, and the test passes. Here is the remaining code for the directive. It largely repeats what we did for the first property, and we've also included an image. We're using the ng source directive to set the image's source. I've also completed the test which builds upon what we started, only this time I've extracted the expected HTML into a variable using the same string array style format for readability. The principle is the same, our directive should give us the HTML we expected based on the scope data we've provided. Our tests are all passing, which is a good sign. I realize I've missed other property to display the plot text of the movie, so let's add that in in the TTD style. First I will update the expected HTML value, pasting in the plot text as I have from the clipboard. Our test fails,

so now let's make it pass. First we need to add the plot text to our scope object. Then switch to the directive and then add the property to the template to display the plot text. Refreshing the test, and we see green. We can now update the homepage to use our new directive. We need to update the index to our HTML page to include our new directive. Then we can start up the server, head over to the browser and see lovely looking movie data in place of the boring text.

Using ngMock's $rootScope Service

We used $rootScope in the last couple of modules so that we could call the apply function to resolve promises, and we used it in the last hour in calling digest. But what is $rootScope? It's an Angular service which represents the root scope of an application, which every Angular application has. All of the scopes are descendant scopes of the root scope. When we use the $rootScope service in our tests we're getting ngMock's version of $rootScope which has all the matters from the default $rootScope service with the addition of two utility methods for unit testing. The two additional methods are count child scopes, which counts all the direct and indirect child scopes of the current scope and count watchers, which counts all the watchers of direct and indirect child scopes on the current scope. The most common use case of the $rootScope service in unit tests is creating new child scopes for controllers and tests. This is done by its new method as shown in this snippet. But in the tests we've written to date we've made use of a simple JavaScript object, so what would be the use cases for creating a new scope via the $rootScope service for a test? Let's work through a quick demo and explore the differences. We're going to change track slightly and look at these features separately from our movie application. The code you see here is for a controller that represents a menu. The menu has some default items and if the selected property on the scope object is set, the message should be displayed in the view as set on the scope. The test we see has an expectation that the message property on the controller scope object should be, you selected beverages, which would mean the scope selected property has a value of zero. I'm going to write just enough code to make this pass, which means I need to inject the controller service, initialize the scope variable to be an ng object, and set the selected property to have a value of zero. So we can track the state of the scope object, I will output the scope object to the console. Starting up the test shows us a passing test and the state of our scope object from the console output. In this example, we made use of a simple JavaScript object to represent the scope, and that sufficed. Upon reviewing the $rootScope service, we noted that each Angular application has a root scope and multiple child scopes. Imagine the scenario where we wanted to have our selected property set on the root scope and we wanted to test with this. This would mean that we wouldn't set the property directly on the controller scope object. We would need a way of setting the value on the parent scope of the controller. The first thing we would need to do is inject ngMock's $rootScope service. This time we will set the selected property on the root scope, and we also need to create a child scope for our controller via the new method. If we start our test we can see that this worked. In our controller code, the selected property wasn't found on the controller scope, so the root scope was checked and the property was found, and so our test passed. Note how the console output has changed considerably. We can now see the native methods of the $rootScope service. If we use ngMock's dump function we now see a much neater string representation of our root scope and scope object. We can also see the root scope and child scope,

which can be very useful when debugging tests with complex scope hierarchies. As we saw in the introduction, ngMock's version of $rootScope is appended with two helper methods. The first is count child scopes, which we can call on any scope object provided it was created from the $rootScope service. If we (mumbles) this to the console and run the tests, we can see our root scope has one child and the controller scope has zero. This is what we expect based on our test setup. To see how this value changes I'll add individual calls to the new function to create new child scopes, which we can see reflected in the console outputs. The second helper function we saw is called count watchers. A watcher is what the name suggests, it watches properties on an Angular scope so it can react to changes, such as a user changing a value in an input field. So that we can demonstrate this, I've created another example test. In this test I've created a simple directive that represents an element called search control that is replaced with basic HTML for an input control when the template is compiled. While using the $rootScope service in conjunction with the compile service and the (mumbles) to write to the count watcher value from the $rootScope to the console. If I run the test we can see it passed, and the console output we set up tells us that we have no child scope and no watches. I'm going to update the directive code so we can set the placeholder value of the input field in the template via a property on the scope object. I'll set the value for this on the root scope object and rerun the test. The test still passes, only now we see that the watch counter has incremented by one. This is because our new property on the scope object has been watched. If I set the second property on the $rootScope object we can see that the property is actually set on the object but the watch count stays at one because we haven't linked this property to the template. If we add the query property to the template and update the expected HTML to make the test pass we can also see that the count watchers value is now two. Before we wrap up this demo, we should remind ourselves that the $rootScope service gives us access to its standard methods, therefore, in most cases we can use its functionality as is. As a quick example, I've updated our menu example controller to set the scope's message property when an event called, selected is triggered, so how would we trigger this event in our unit test? Well to do so we would simply call the admit function as we normally would in our Angular code using the $rootScope service in our tests. This principle applies for any of the standard $rootScope services we were to use in our tests.

Adding the Directive to the Movie Results

Before we wrap up and move on we need to update the search results page to use our new directive. We want to be able to click on a row and have it expand to show the full movie data. We need to change the results of our HTML file in the movie app source folder. We can add the element for the directive and update the property from the scope object that will have the movie data. For the results page we will add a property called data to the result object that will hold this additional movie data. We need to update the movie results controller, adding an expand function that we can call from the UA. The function accepts the index of the movie result in question, along with the ID. We call the find movie function, getting the additional data for the movie and attaching it to the movie result based on the array index. We also set the open property for the accordion in our view. The final part is to add an ng click directive that calls the expand function once the result row is clicked. If we refresh the results page and click on a row we can see a nice expanding row with more movie data.

Summary

In this module we created a new directive for our movie app and saw how to unit test it. We saw how we needed to use the $rootScope service to compile our directive into HTML. We took a deeper dive into the $rootScope service contrasting when you could use its features in unit tests and controllers. We saw how the helper methods added to the $rootScope service by ngMock could help us track scope behavior across complex directive code, and of course, we put everything we learned together by adding a new directive to the movie app.

Unit Testing with Dates

Introduction

Hi, this is Bradley. In this module we're going to work with dates and unit tests. Dates can be tricky in unit tests. But, ngMock can help us out a little with this. As it has its own date type for unit tests called TzDate. The Tz meaning time zone. Therefore, our aim in this module is to explore how we can unit test dates using ngMock's TzDate type. The running theme in this course has been to cover the functionality of ngMock by adding features to our movie app. So in this module we're going to extend our movie result directive. Adding a feature to use the release date of the movie, giving us a user-friendly display, telling us in years or months how long ago a movie was released. We're going to create an Angular filter for this, so we will also see how we can unit test a filter in Angular.

Unit Testing Dates in JavaScript

Before we dive into the features of TzDate, let's quickly refresh our understanding of JavaScript Dates. So that we can appreciate why they can be difficult to work with in unit tests, and the problem that TzDate aims to solve. In JavaScript time is measured in milliseconds since the 1st of January, 1970. This means that creating a new date object with a new date constructor passing a value of zero equates to Thursday, the 1st of January of 1970 at midnight GMT. The millisecond value can be any negative or positive number. Negative numbers will construct dates before 1970. And positive numbers will be dates starting from 1970. We can also construct new dates via string representations. Using a simplification of the ISO 8601 Extended Format. We won't examine every possible string combination, but be aware that we can create dates with or without time information. The zed suffix we see here is used to note that the time set is zero hour. You may hear this called zero hour or Zulu time. We can specify time zone offsets by using plus or minus, followed by time expression in hours, minutes, seconds, in place of the Z. In the example we see here, after the T you have the hour, minute, second details on the end. Then a plus sign with the hour and minute offset. Whenever we construct a new date its base value will always be in UTC. UTC is an abbreviation for Coordinated Universal Time. Which is a time standard by which the world regulates clocks and time. When we specify a date without specifying

an offset, UTC is assumed. A date object instance has two states. The UTC time and a local time based on the system settings of the machine executing the JavaScript code. It's these two states that can cause us problems in unit tests. If I write a test that passes on my machine in a GMT time zone, and another developer was to run the same test in a different time zone, there are cases where a test could break for them but pass for me. Let's see some code examples to better visualize this. In this slide we are creating a new date and calling native functions on the date object, to show us the date and the hours for the time zone of the machine running the code and the same for the UTC time. If we were to run this code on the machine configured to be in Paris or Central European Time, we see that midnight UTC time is actually 1:00 a.m. in Paris. Since this time zone is one hour ahead. If I change my system settings to be London GMT and run the same code again, we see that the local time is the same as the UTC time. Because for this date time, the time zone in London is the same. In this slide, we are creating a date with a negative offset. The negative offset suffix of minus one hour, indicates that the date time information provided is for a place that is one hour behind UTC, and that the time in that place is midnight. The result is that our date has a base UTC value of 1:00 a.m. And the local time is 2:00 a.m., since the local time settings of the machine running this code is in a place that's one hour ahead of UTC, just like our Paris example. In this slide we are creating a date with a positive offset. This is the inverse of the previous example. Here we give a date time for a place that is one hour ahead of UTC. So midnight results in 23:00 hours in UTC. The local settings of the machine executed in the JavaScript is, again, one hour ahead of UTC like Paris. Giving us a local time of midnight. The key point of these examples is that the output from calling on the functions like toLocaleString and getHours, and so on, changes based on the local settings. This makes sense, but can cause problems when it comes to verifying behavior with dates when writing unit tests. Here's a trivial application that will highlight some of the problems we face when unit testing with dates in JavaScript. The controller code has a basic condition to check if the $scope.nowTime property value, rolls into the next year. And says an appropriate message, if so. There are two simple unit tests for this controller, for each date. The first is when we're into the New Year. And the second is when we're not. The code is pretty simple. We create a date object with a date time for each date that will control the behavior of our controller code. If you are based in London, these tests would both pass as expected. But what if you change your date and time settings to be somewhere else, like Paris? When running the tests again, the second test fails with the exception: Expected 'Happy new Year!' to be 'Keep on counting down...!' So how has the date of 2014 turned into 2015? When we call the getFullYear method in our controller, we're getting the year value of the local time zone based on the UTC date we set up. For this test, and in Paris, the UTC date time at 23:00 hours is actually midnight. Nudging us into the new year. In terms of logic, this is perfect and exactly what we want. But, it's not so good for unit tests since the system settings could potentially cause the tests to fail. For example, when two developers are working in different locations or if a running test is part of a continuous integration process that has a server situated in a different country. If those examples left you feeling a little confused, then you're not alone. JavaScript dates are difficult to work with, but the takeaway point from this is that it's difficult to work with the UTC dates and the local date time settings when trying to work with dates in unit tests.

Using ngMock's TzDate Type

In the last clip we looked at examples where JavaScript dates are difficult to use in unit tests. ngMock's TzDate type was designed to make this task a little easier. TzDate is a wrapper for the native JavaScript date type with an added API to make it easier to construct dates with time zone information. The official line from the documentation states its main purpose is to create Date-like instances with time zone fixed to the specified time zone offset, so that we can test code that depends on local time zone settings without dependency on the time zone settings of the machine where the code is running. But, I think what they meant to say is we can set up a time zone that will not change according to the local machine. Here are the same tests using the angular.mock TzDate type to create the date objects. Given that we've provided the time zone offset in each test, the test will always pass regardless of the system settings of the computer executing the tests. In the snippet we see here, we're passing a zero for the time zone information. Which means that the local time will be the same as the UTC time in all cases. The TzDate type has logic that takes into consideration the offset we pass, with the local time settings of the machine executing the JavaScript to return consistent responses when we call the methods, such as getHours on the date object that will use the local time settings. The TzDate type is not a complete implementation of date object, which may result in some run time exceptions for certain methods. For example, if we were to call the getUTCDay in our controller code in the test, we would see the exception Uncaught Error: Method ‘getUTCDay' is not implemented in the TzDate mock. This is clearly stated in the documentation. And here is the full list of unimplemented methods. We can sometimes run into this problem when writing out angular logic and tests. Take the following example, in this test we want to assert that the user given value in hours is added to our base time. We set up a new TzDate type and then pass it to our methods that will increment the time by the number of hours we supply. The test will fail when we call setTime on our mock date object, since it's not implemented. When encountering such issues it's useful to take a few steps back and consider what we're actually testing. Do we want to test the date object functionality works? Or do we just want to test that we're passing at the correct arguments? For unit tests, we should be testing that we are using the date type correctly as opposed to its functionality. Therefore, in many cases we can make use of Jasmine's spy functionality and check that setTime was called with the correct argument that we were expecting based on our state. I've updated the code here to verify the same function is called with the argument we expect in milliseconds. Which is the same as our base time plus the additional hour we want to add.

Unit Testing an Angular $filter

Let's begin with the end goal in mind. We want to use this filter in our movie result directive. We can pass the release property, which is an invalid date format, as we see in the code snippet here. And it will return a string value that displays the time from now in either years or months. So we need to turn our attention to creating this from now filter. I'll create the new spec file called from-now.filter.spec.js and add a basic starting test which is the throw error if we pass an undefined argument to the filter. Next, I'll create the from now filter itself in the movie app source folder, and write the logic to throw the exception if the value argument is not truthy. I'll start up the tests to verify that everything is where we

expect it to be. And we see a passing test, which is what we want. Now, we can start to add the logic. The next test we will add is to check that if a string value is passed through our filter, that it's in the correct format to make a valid date. After ending the test, we see that this fails because we're not returning a value from our filter. In the filter we will add an if statement to check the date type of the value. If it is a string, we will create a new date. And then call the getTime function of the date object. In the case of an invalid date or time, this call will return the Not a Number type, in which case we can return the value as is. We can see that this test is now passing. Now, we can start to add the call logic. Which will be to take two dates and calculate the difference between the two. The value will be the release date of the movie and the base date will represent the current date and time. Although, for our tests, we will want to specify what this value is. Our expectation is the filter will correctly calculate that the value date was two years since the base date. Note how we are using the TzDate type in the tests. This means we don't have to worry ourselves in our test setup to try and pick dates that might not cause problems when executed on different machines in different time zones. Our logic calculates the difference, in milliseconds, between the two dates and divides this value by a year represented in milliseconds to tell us the difference in years. Note also that the now value will default to the current date time of the machine if no base value has been passed. Which is what will happen when the filter is used with any application. We also need to add the base value arguments to the filter function. The test passed, so we can move on to the next test and logic. Which is to display the correct message based on the year value being singular or plural. This means that our filter will use the word year or years correctly. Here is the remainder of the test. It's similar to what we did for years, only this time for months. And here is the updated code for the filter. With the additional code for checking how many months ago a film was released. If the difference is less than one year, we return the months instead. We could extend this to work with days also, but that's enough logic for now to show how we can use dates in unit tests. Now we're ready to update our directive code so that it uses the filter. Which is where we started at the beginning of this clip. I will update the test so that we expect the from date to be extended with 38 years ago, which is when this film was released. The test will fail because our directive isn't using the filter yet, so let's go ahead and add this. And the test is still failing, as I forgot to update the watch account we looked at in the last module. The watch account is incremented because we've added a new property to our directive. So let's update the counter and re-verify the tests are passing. And we're looking good! The last thing to do is include our new filter in the index.html page and then fire up the application to check how it looks in the browser. And as we can see, everything's looking good.

Summary

In this module we reviewed JavaScript dates and saw why they are difficult to unit test when it comes to local versus UTC time settings. We looked at ngMock's TzDate type and understood the problem that this solves and we saw how to use it in unit tests. Finally, we put all this knowledge into action by creating the from now filter. So that we could use it in our movie result directive to see how long ago a movie was released. That's it for this module! In the next module, we look at how to handle exceptions in our application and unit tests.

Unit Testing with Exceptions

Introduction

Hi, this is Bradley. We're getting near the end of this course, and are near to finishing our movie app, but one thing we don't have yet is a way of handling exceptions in our application. In this module, we're going to address that, and in doing so, we will explore how to use Angulars $exceptionHandler service in our unit tests. We won't be adding a specific feature to our application in this module. Rather, we will explore how to make our application more robust by handling any potential errors that may crop up. So let's get to it.

The $exceptionHandler Service

Before we dive into the demo, let's first brush up on our knowledge of the $exceptionHandler service. Any uncaught exception in angular expressions is delegated to this service. The default implementation simply delegates to the log service, which logs it to the broader counsel as an error. When it comes to using this service in unit tests, ngMock has its own implementation of the $exceptionHandler service that rethrows or logs errors passed to it. The mock service is backed by an $exceptionHandler provider which we can use to configure which of the two modes we want to use in a unit test. The first mode is log which does as the name suggests as it catches the error while logging it. Errors will not be rethrown with this mode. The second mode is to rethrow which does as the name suggests and rethrows the error as a standard exception. This also maintains a log of errors. We can configure the provider in our tests by the snippet we see here where we would use one or the other. So that's enough to grasp the main concepts. Now we can use this in our movie application.

The $exceptionHandler Service - Demo

Now we're ready to use the $exceptionHandler in our movie app. In our search results controller, we have already added some code to catch an exception. But to keep things simple at that point, we simply set a message on the scope object to indicate that something had gone wrong. Let's change this to throw an actual exception with the same message by using JavaScript's throw key word. If we fire up the tests, we see a failure because we're not doing anything in our tests to handle this newly thrown exception. Let's switch our focus to the results controller test to address this. Our test to check that the error message property was being set on the scope object is no longer valid. So let's update this test accordingly. We need to wrap our code in a function and use Jasmine's to throw match it to verify that the code inside our function throws an exception. This works, but is not ideal for a couple of reasons. Firstly, it's a better practice in Angular not to throw native exceptions when we can make use of its $exceptionHandler service. Using this service allows us to take advantage of the exception handling features offered by Angular out of the box or if we want to change how we handle exceptions in the

future, we can inject our version of the service in one place rather than having to troll through our entire code base changing the $exceptionHandler code. The second reason to use the service, is that we can simplify our test code which we will see shortly. If we check the test, it's failing because the error message isn't being passed via the rejected promise to the cache block in our controller. If we pass its value it is then passed as the E arguments to the cache block which in turn is passed to the $exceptionHandler and our test still passes. This works well, but I did say that using the $exceptionHandler service in our tests would help to simplify our test code. By default, the $exceptionHandler rethrows the exception which is why our test still passed. But if we want to set this up to use log mode, things will work a little differently. We start by using Angie mocks module function to get access to the $exceptionHandler provider. This allows us to set the mode function to set what mode we want. Passing rethrow, as we see here, is the default condition. So let's change this to log and see what happens in our test. The test fails as we're no longer rethrowing the exception. Now the exception is being caught and stored as a list internally. So the next obvious question is how can we get at the list of errors to check the contents? The first thing we can do is remove the function wrapper we added to catch the exception. This is much better for test readability. We can use the $exceptionHandler service within the expectation by accessing it's errors property. This is an array of objects so we will call Jasmine's to equal matcher to check what the array list of errors should contain. Which, in this case, is one item with the value "Something went wrong!" Don't forget that we need to inject the service as we've done with all the others. Now we're ready to check the tests and everything looks good. To demonstrate to Luther Moore how exceptions are added to the errors array of the mock and exception handler, I will write the contents to the consult so we can see the raw contents in the test output. If we call more exceptions, we can see that they are added to the array. They are stored in the sequence in which they happen as we can see here. Now we can take what we've learned and use the $exceptionHandler service in other areas of our application. Let's add some error handling logic to the home controller. We will remove our hard coded call ID's from the popular movie service so that we can add some error handling code to the find movie function that we call at five second intervals. I'll add the new test which will be the check that if an error happens in the middle of updating the movies on the homepage that the error is handled and the process continues onto the next movie. The test setup is the same as before, only I will add a new entry to our mock code to reject the promise for the find function on the OMDPAI service. Since we need the popular movies API to retain different ID's for this test, we have to setup a mock for each test for the popular movies get function. The mock in our new test will return the ID for the movie we know will throw an error. If we skip to the completed test, we see the mock setup for the find function. We updated it to reject the promise for the specific movie ID. Then, for each test, we set up the spy for the OMDPAI's find function and initialized the controller. We can go ahead and update the find function in the home controller to use the $exceptionHandler service to handle an error when calling up OMDPAI's find function. We see that the error is being thrown in the test output. So we need to update the test code to handle the error. We will do the same as before. We will set the mode to be logged, inject the $exceptionHandler into the test, and set up the expectation to check for the error as being logged. We need to check the tests to verify that this worked and we see passing tests. Note that the exception didn't interrupt the code since we set up the service to use log mode for this test.

Summary

In this module, we answered the question how do we unit test with the $exceptionHandler service in ngMock? We saw how to use Angular's $exceptionHandler in our Angular controllers, and we saw how we could test this using ngMock's mock implementation of the $exceptionHandler service. We saw how we could configure its two modes to log or rethrow, to control how we wanted the exceptions to behave in our tests. And, of course, we added a better method of handing errors to our movie application. In the next module, we will add more features to manage what's happening when an application is running which will be logging.

Unit Testing with Logging

Introduction

In the last module, we added exception handling logic to our application to catch errors as they happen. Another useful feature we will want to add to the application is a mechanism for logging to give us the ability to trace what's going on once a application is running in the browser. In this module we will add logging functionality to our application using Angular's $log Service and to answer the question, How do we unit test with the $log Service in ngMock?

Introduction to the $log Service

To show a type of logging we could add to our movie app, we can see here using Chrome's developer console that the debug information is being upward from the search result controller. Angular's $log Service is being used here to log when the search request is made, what the query was, it tells us when the data has returned and we're also displaying the data returned allowing us to expand the object and see its properties. If I click on a row, we can also see the full movie data that's downloaded when the row expands. This level of logging can be very useful for understanding what's going on in an application when trying to understand the logic of finding error. To recap, Angular's $log Service is a simple service for logging. The default implementation safely writes the message into the browser's console. ngMock has its own implementation of the $log service that we can use in Unitest to verify that we are writing content to the $log in service in the way that we expect. It gathers the $log messages into an array with one array per logging level. Here's a code snippet that shows at a high level how we would use the service in our Angular code. The top part is a module configuration. By using the $log provider we can configure whether or not the debug level logging is enabled. Typically we would want this disabled in production and enabled in our dev environments. The controller underneath has a $log service injected and we can interact with it by calling the appropriate function for the $logging level we want to target. The $logging levels are standard, info, error, warn and debug. That's enough theory for now, let's put this into action in the next clip.

Using the $log Service

The code we see here is the home controller of our movie app. I've injected the $log service and have added a core for each logging level that we saw in the last clip. If we look at this in the browser with a developer toolbar visible, we can see the output from $log service in the console. But you may also notice that we don't see the debug level detail we logged. We can enable this via the log provider we saw in the last clip. We need to add the configuration to the movie app module and set the debug enable value to be true. Now if we refresh the browser we can see our debug message at the bottom. Now let's imagine we want to unitest that we're calling each of the expected logging levels in the controller. We would need to extend our home controller tests firstly, by injecting the $log service into the test. So that we can see what's going on, I will start by writing the detail for the logging level to the console so that we can see it in the test output. As we see here, it's an array in an array. If I go ahead and output the remaining levels it gives us an idea of how the data is stored as a single array per level. Note again that the debug level is empty, as we've yet to enable it for the test. We need to use ngMock's module function to configure the log provider in the same way we did in the movie app module. As the default is false, we need to set this to true. Now we can see the debug level outputs in the console. To make expectations on the log data, we can use Jasmine's to equal matcher to check that the contents of each array per level is what we expect it to be. A feature that is available on ngMock's log service is the reset function. This does as the name suggests and clears all of the arrays. If I call it and end comment to the console output code we can see from the test output that our logging detail has been cleared. The second useful feature it has is the assert empty function, which will throw an exception if any of the log levels have a value. In the event of such an error, the details are displayed in the test output. Here the test fails and it's telling me the contents of the log service.

Unit Testing $log in the Movie Application

Now we understand the basics of using the log service in Unitest, we can use it for our movie app. This is the results controller from the movie app and for the purposes of the demo, I've already injected the log service and have commented out the calls to the log service level I want to use. We're going to write the test for this first, see them fail and then uncomment these calls. They both call to the debug level logging. The first is to state what query was received and the next is to confirm when the data is returned from the server and display the data. If you recall, this is what we saw at the beginning of this module. To get going with the test we need to add a variable for the $log service and set that with the injected service. We can then extend our existing tests with expectations to check what we expect is being set on the debug level of the $log service. The first expectation we will add is for the output that tells us what the query was. If we fire up the tests we see the expected failure because we haven't called the $log service yet in the controller. If I switch back to the controller and uncomment the first call to log the debug information our test is passing. We then need to repeat this for when the data is returned. This time we're going to check the data returned in addition to the string value. Again we see the test

fail because we haven't implemented this in the controller code. If we uncomment the tests look good. You may have noticed that we didn't have to configure the log provider to enable debug logging in this demo. This is because we are using the movie app module in the test, and if you remember in an earlier clip we configured this to be true for the movie app module. If I update this to be false it affects our test output. If we want to override this in our test we can use ngMock's module function along with a log provider as we did before and set the debug level to be true. This will override what was in the movie app module configuration. Checking the tests and we can see that this worked.

Summary

In this module we looked at Angular's $log Service that allows us to write to the browser console. We also saw how ngMock has its own version that allows us to check the $log service behavior in our tests. We worked through some demos to see how to use the $log service in our controllers and how to check we were using the service as we expected in our unitests. We've concluded the penultimate module of this course. In the next and final module, we will finish up the application seeing how we can make use of ngMock's end to end feature to mock service side calls to a server.

Testing with ngMockE2E BackEnd

Introduction

Hi, this is Bradley. Welcome to the final module of this course. In this module, we're going to wrap up our movie application by using the ngMockE2E BackEnd module. If you cast your mind back to a few modules ago, we added the rotating slides to the home page of the MovieApp but the list of our dates were hard coded because we didn't have a service side for the API. When we looked at the code to throw exceptions, we took out the hard-coded list and restored the correct code to our PopularMovies resource that would return the list from the server, but we don't have a server. If we fire up the application again, we see that it's broken and we're greeted with a complicated-looking error message along with a 404 in the console. Our unit tests are still passing since we're doing TDD and therefore testing in isolation without needing a real server. Now we want to carry out the testing into the full-stack using the browser. We will see how we can set up a fake backend using the ngMockE2E $httpBackEnd Service.

Testing with the ngMockE2E Module

Before we dive into the code, let's take a few steps back and review what the ngMockE2E module is. It's an angular module which contains mocks suitable for end-to-end testing. There is only one mock present in this module, the e2e $httpBackend mock. The $httpBackend service included in the ngMockE2E module is similar to the $httpBackend service we looked at from ngMock. This is a fake

HTTP backend implementation suitable for end-to-end testing or backend-less development of applications using the $http service. ngMock's version of the the HTTP backend allowed us to test in isolation, whereas, this implementation is aimed at in-browser testing. To visualize the differences, ngMock's version of the HTTP backend allows us return canned answers to HTTP calls in our code of ender tests, meaning that it was never possible to make real HTTP requests. The end-to-end version provides its functionality too, but with the ability to make real HTTP calls, meaning that when we run the code in the browser, we can use a mixture of fake and real HTTP responses. The $httpBackend has a similar looking when function we saw back in the module that looked at HTTP in more depth. We can set up a matching URL for a given HTTP request and return a fake response. Additionally, we can use a function called passThrough which does as the name implies and passes through the request until it makes a real HTTP request. In this snippet, if a GET request is made to the API endpoint, a real HTTP request with the same properties is made. The standard verbs are supported just like the ngMock $httpBackEnd version. The key differences between ngMock's Backend are only the when function is the same. We can make real HTTP calls via the passThrough function. We don't have to call the flush function to complete requests like we did back in module three. We cannot use this in unit tests which means it's not possible to use this to make real HTTP calls in unit tests which we sometimes may want to do for debugging or writing integration tests. The key point is that they are not in the same module but are conceptually grouped in the same JavaScript filing group.

Using ngMockE2E's $httpBackEnd Service

To pick up from Clip One, we had the error on the Home page. Now we have an idea of the ngMockE2E module. Let's use it to fix this problem. We need to add code to the MovieApp's module to configure the fake HTTP backend. Here, I will add code for a fake HTTP request that returns the list of Popular Movie IDs that we need. I will set up the basic response headers and add a rule using the when-GET function that checks if the URL endpoint has the value, popular, which means it's a GET request to our "popular" resource endpoint. We will return a 200 status, the data, and the headers. Since we have templates and other requests to the OMDB service that need to make real HTTP requests, I need to add a catch-all rule at the end that calls the passThrough function for any other URL. Doing this means that everything but the Popular Movie API endpoint will work as it did before. I need to include the ngMockE2E module to the MovieApp. We also need to reference the angular mock's JavaScript include in the index-to-HTML page. Refreshing the app, we see that we still have an error. This is because in the test we wrote, we weren't using Angular's resource API correctly. Since we were mocking this in unit tests, we didn't notice this until now. This highlights a pitfall in doing everything TDD as we have done to date. Sometimes, it's not until we bring everything together that we realize something doesn't fit. It's a simple change to fix this. We call the query function instead and pass our existing function as a callback. Now, everything is working together. We see the movie data from the omdbAPI which means our fake call to the "popular" API endpoint returned the movie IDs we set up. We also see an exception in the console that's generated by our fromNow data filter. Let's go ahead and clean this up a little. Since we changed the code in the Home Controller, we also need to update the tests. Starting the tests confirms that they fail. I can make them pass by updating the PopularMovie spy to call the query function instead and resolve

the callback with the movie IDs we expect. I need to do the same in the second test where we were checking for errors. All our tests are now back to passing. To correct the console errors, we should update the filter to not through an exception for a null value. Since once the template initially loads, there isn't yet a value. This is not necessarily an error, it just creates unwanted noise in a developer console. Changing it to simply return will prevent any further exceptions caused by the code below trying to execute without a valid date. Refreshing the application now shows us a noise-free developer console. We also need to tweak the tests to check that no exception is thrown from the fromNow date. We are back to all passing tests. The reason for showing these imperfections is that it can be a good idea to run things together soon in the process rather than later. Since mocking and testing in isolation can mask these type of integration problems. We should also note that save for those couple of tweaks, the application just worked when we fired up the browser. Until this module, we didn't really do any in-browser checking or testing or debugging, which is where a significant amount of developer time can be spent when writing new code. As we saw, it's not perfect. It would be great if we could use the ngMockE2E features in our unit tests. As it stands, this is not supported. There is an open issue for this in the Angular projects and GitHub repo which you can find at this URL. Who knows, maybe one day they might add it. Some users have left work-arounds to this problem in their comments. You may even find one from me in there.

Summary

This concludes the module and this course. In this course, we covered all of ngMock and its features. We stuck to a mostly TDD workflow writing a unit test for each bit of functionality we added. We put all this into action in building out the movie application. In this module, we saw how to use the ngMockE2E module and use its only feature, the fake HTTP backend implementation so that we could stub out the service-side code of our application. I hope that you enjoyed this course, I certainly enjoyed producing it. I hope that we can meet again in the future courses. Until next time.