**7. Unit Testing in angular Js.**

For testing Angular applications there are certain tools that you should use that will make testing much easier to set up and run.

**Karma**

[Karma](http://karma-runner.github.io/) is a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests. You can configure Karma to run against a number of browsers, which is useful for being confident that your application works on all browsers you need to support. Karma is executed on the command line and will display the results of your tests on the command line once they have run in the browser.

Karma is a NodeJS application, and should be installed through npm. Full installation instructions are available on [the Karma website](http://karma-runner.github.io/0.12/intro/installation.html).

**Jasmine**

[Jasmine](http://jasmine.github.io/1.3/introduction.html) is a behavior driven development framework for JavaScript that has become the most popular choice for testing Angular applications. Jasmine provides functions to help with structuring your tests and also making assertions. As your tests grow, keeping them well-structured and documented is vital, and Jasmine helps achieve this.

In Jasmine we use the describe function to group our tests together:

describe("sorting the list of users", function() {

// individual tests go here

});

And then each individual test is defined within a call to the it function:

describe('sorting the list of users', function() {

it('sorts in descending order by default', function() {

// your test assertion goes here

});

});

Grouping related tests within describe blocks and describing each individual test within an it call keeps your tests self documenting.

Finally, Jasmine provides matchers which let you make assertions:

describe('sorting the list of users', function() {

it('sorts in descending order by default', function() {

var users = ['jack', 'igor', 'jeff'];

var sorted = sortUsers(users);

expect(sorted).toEqual(['jeff', 'jack', 'igor']);

});

});

Jasmine comes with a number of matchers that help you make a variety of assertions. You should [read the Jasmine documentation](http://jasmine.github.io/1.3/introduction.html#section-Matchers) to see what they are. To use Jasmine with Karma, we use the [karma-jasmine](https://github.com/karma-runner/karma-jasmine) test runner.

angular-mocks

Angular also provides the [ngMock](https://docs.angularjs.org/api/ngMock) module, which provides mocking for your tests. This is used to inject and mock Angular services within unit tests. In addition, it is able to extend other modules so they are synchronous. Having tests synchronous keeps them much cleaner and easier to work with. One of the most useful parts of ngMock is [$httpBackend](https://docs.angularjs.org/api/ngMock/service/$httpBackend), which lets us mock XHR requests in tests, and return sample data instead.

Testing a Controller

Because Angular separates logic from the view layer, it keeps controllers easy to test. Let's take a look at how we might test the controller below, which provides $scope.grade, which sets a property on the scope based on the length of the password.

angular.module('app', [])

.controller('PasswordController', function PasswordController($scope) {

$scope.password = '';

$scope.grade = function() {

var size = $scope.password.length;

if (size > 8) {

$scope.strength = 'strong';

} else if (size > 3) {

$scope.strength = 'medium';

} else {

$scope.strength = 'weak';

}

};

});

Because controllers are not available on the global scope, we need to use [angular.mock.inject](https://docs.angularjs.org/api/ngMock/function/angular.mock.inject) to inject our controller first. The first step is to use the module function, which is provided by angular-mocks. This loads in the module it's given, so it is available in your tests. We pass this into beforeEach, which is a function Jasmine provides that lets us run code before each test. Then we can use inject to access $controller, the service that is responsible for instantiating controllers.

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

it('sets the strength to "strong" if the password length is >8 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

});

});

Notice how by nesting the describe calls and being descriptive when calling them with strings, the test is very clear. It documents exactly what it is testing, and at a glance you can quickly see what is happening. Now let's add the test for when the password is less than three characters, which should see $scope.strength set to "weak":

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

it('sets the strength to "strong" if the password length is >8 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

it('sets the strength to "weak" if the password length <3 chars', function() {

var $scope = {};

var controller = $controller('PasswordController', { $scope: $scope });

$scope.password = 'a';

$scope.grade();

expect($scope.strength).toEqual('weak');

});

});

});

Now we have two tests, but notice the duplication between the tests. Both have to create the $scope variable and create the controller. As we add new tests, this duplication is only going to get worse. Thankfully, Jasmine provides beforeEach, which lets us run a function before each individual test. Let's see how that would tidy up our tests:

describe('PasswordController', function() {

beforeEach(module('app'));

var $controller;

beforeEach(inject(function(\_$controller\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$controller = \_$controller\_;

}));

describe('$scope.grade', function() {

var $scope, controller;

beforeEach(function() {

$scope = {};

controller = $controller('PasswordController', { $scope: $scope });

});

it('sets the strength to "strong" if the password length is >8 chars', function() {

$scope.password = 'longerthaneightchars';

$scope.grade();

expect($scope.strength).toEqual('strong');

});

it('sets the strength to "weak" if the password length <3 chars', function() {

$scope.password = 'a';

$scope.grade();

expect($scope.strength).toEqual('weak');

});

});

});

We've moved the duplication out and into the beforeEach block. Each individual test now only contains the code specific to that test, and not code that is general across all tests. As you expand your tests, keep an eye out for locations where you can use beforeEach to tidy up tests. beforeEach isn't the only function of this sort that Jasmine provides, and the [documentation lists the others](http://jasmine.github.io/1.3/introduction.html#section-Setup_and_Teardown).

Testing Filters

[Filters](https://docs.angularjs.org/api/ng/provider/$filterProvider) are functions which transform the data into a user readable format. They are important because they remove the formatting responsibility from the application logic, further simplifying the application logic.

myModule.filter('length', function() {

return function(text) {

return ('' + (text || '')).length;

}

});

describe('length filter', function() {

var $filter;

beforeEach(inject(function(\_$filter\_){

$filter = \_$filter\_;

}));

it('returns 0 when given null', function() {

var length = $filter('length');

expect(length(null)).toEqual(0);

});

it('returns the correct value when given a string of chars', function() {

var length = $filter('length');

expect(length('abc')).toEqual(3);

});

});

Testing Directives

Directives in angular are responsible for encapsulating complex functionality within custom HTML tags, attributes, classes or comments. Unit tests are very important for directives because the components you create with directives may be used throughout your application and in many different contexts.

Simple HTML Element Directive

Let's start with an angular app with no dependencies.

var app = angular.module('myApp', []);

Now we can add a directive to our app.

app.directive('aGreatEye', function () {

return {

restrict: 'E',

replace: true,

template: '<h1>lidless, wreathed in flame, {{1 + 1}} times</h1>'

};

});

This directive is used as a tag <a-great-eye></a-great-eye>. It replaces the entire tag with the template<h1>lidless, wreathed in flame, {{1 + 1}} times</h1>. Now we are going to write a jasmine unit test to verify this functionality. Note that the expression {{1 + 1}} times will also be evaluated in the rendered content.

describe('Unit testing great quotes', function() {

var $compile,

$rootScope;

// Load the myApp module, which contains the directive

beforeEach(module('myApp'));

// Store references to $rootScope and $compile

// so they are available to all tests in this describe block

beforeEach(inject(function(\_$compile\_, \_$rootScope\_){

// The injector unwraps the underscores (\_) from around the parameter names when matching

$compile = \_$compile\_;

$rootScope = \_$rootScope\_;

}));

it('Replaces the element with the appropriate content', function() {

// Compile a piece of HTML containing the directive

var element = $compile("<a-great-eye></a-great-eye>")($rootScope);

// fire all the watches, so the scope expression {{1 + 1}} will be evaluated

$rootScope.$digest();

// Check that the compiled element contains the templated content

expect(element.html()).toContain("lidless, wreathed in flame, 2 times");

});

});

We inject the $compile service and $rootScope before each jasmine test. The $compile service is used to render the aGreatEye directive. After rendering the directive we ensure that the directive has replaced the content and "lidless, wreathed in flame, 2 times" is present.

**Underscore notation**: The use of the underscore notation (e.g.: \_$rootScope\_) is a convention wide spread in AngularJS community to keep the variable names clean in your tests. That's why the [$injector](https://docs.angularjs.org/api/auto/service/$injector) strips out the leading and the trailing underscores when matching the parameters. The underscore rule applies ***only*** if the name starts **and** ends with exactly one underscore, otherwise no replacing happens.

Testing Transclusion Directives

Directives that use transclusion are treated specially by the compiler. Before their compile function is called, the contents of the directive's element are removed from the element and provided via a transclusion function. The directive's template is then appended to the directive's element, to which it can then insert the transcluded content into its template.

Before compilation:

<div transclude-directive>

Some transcluded content

</div>

After transclusion extraction:

<div transclude-directive></div>

After compilation:

<div transclude-directive>

Some Template

<span ng-transclude>Some transcluded content</span>

</div>

If the directive is using 'element' transclusion, the compiler will actually remove the directive's entire element from the DOM and replace it with a comment node. The compiler then inserts the directive's template "after" this comment node, as a sibling.

Before compilation

<div element-transclude>

Some Content

</div>

After transclusion extraction

<!-- elementTransclude -->

After compilation:

<!-- elementTransclude -->

<div element-transclude>

Some Template

<span ng-transclude>Some transcluded content</span>

</div>

It is important to be aware of this when writing tests for directives that use 'element' transclusion. If you place the directive on the root element of the DOM fragment that you pass to [$compile](https://docs.angularjs.org/api/ng/service/$compile), then the DOM node returned from the linking function will be the comment node and you will lose the ability to access the template and transcluded content.

var node = $compile('<div element-transclude></div>')($rootScope);

expect(node[0].nodeType).toEqual(node.COMMENT\_NODE);

expect(node[1]).toBeUndefined();

To cope with this you simply ensure that your 'element' transclude directive is wrapped in an element, such as a <div>.

var node = $compile('<div><div element-transclude></div></div>')($rootScope);

var contents = node.contents();

expect(contents[0].nodeType).toEqual(node.COMMENT\_NODE);

expect(contents[1].nodeType).toEqual(node.ELEMENT\_NODE);

Testing Directives With External Templates

If your directive uses templateUrl, consider using [karma-ng-html2js-preprocessor](https://github.com/karma-runner/karma-ng-html2js-preprocessor) to pre-compile HTML templates and thus avoid having to load them over HTTP during test execution. Otherwise you may run into issues if the test directory hierarchy differs from the application's.

Testing Promises

When testing promises, it's important to know that the resolution of promises is tied to the [digest cycle](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest). That means a promise's then,catch and finally callback functions are only called after a digest has run. In tests, you can trigger a digest by calling a scope's[$apply function](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply). If you don't have a scope in your test, you can inject the [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope) and call $apply on it. There is also an example of testing promises in the [$q service documentation](https://docs.angularjs.org/api/ng/service/$q#testing).

Using beforeAll()

Jasmine's beforeAll() and mocha's before() hooks are often useful for sharing test setup - either to reduce test run-time or simply to make for more focused test cases.

By default, ngMock will create an injector per test case to ensure your tests do not affect each other. However, if we want to usebeforeAll(), ngMock will have to create the injector before any test cases are run, and share that injector through all the cases for thatdescribe. That is where [module.sharedInjector()](https://docs.angularjs.org/api/ngMock/function/angular.mock.module.sharedInjector) comes in. When it's called within a describe block, a single injector is shared between all hooks and test cases run in that block.

In the example below we are testing a service that takes a long time to generate its answer. To avoid having all of the assertions we want to write in a single test case, [module.sharedInjector()](https://docs.angularjs.org/api/ngMock/function/angular.mock.module.sharedInjector) and Jasmine's beforeAll() are used to run the service only once. The test cases then all make assertions about the properties added to the service instance.

describe("Deep Thought", function() {

module.sharedInjector();

beforeAll(module("UltimateQuestion"));

beforeAll(inject(function(DeepThought) {

expect(DeepThought.answer).toBeUndefined();

DeepThought.generateAnswer();

}));

it("has calculated the answer correctly", inject(function(DeepThought) {

// Because of sharedInjector, we have access to the instance of the DeepThought service

// that was provided to the beforeAll() hook. Therefore we can test the generated answer

expect(DeepThought.answer).toBe(42);

}));

it("has calculated the answer within the expected time", inject(function(DeepThought) {

expect(DeepThought.runTimeMillennia).toBeLessThan(8000);

}));

it("has double checked the answer", inject(function(DeepThought) {

expect(DeepThought.absolutelySureItIsTheRightAnswer).toBe(true);

}));

});