Karma is a test-runner, so it runs your test. [Jasmine](http://jasmine.github.io/2.1/introduction.html) is the framework that let you write test

In my opinion in Angularjs you :

* **must** unit-test services, because your business code is there.
* **should** unit-test controller, because users actions are there.
* **may** unit-test custom directives (if you plan to share that directive with others, it's a **must**)

Protractor is made for E2E testing (tests navigation like a real user). It combines WebDriverJS with Jasmine and lets you write End-to-End tests (you simulate a real browser and taking real actions) with Jasmine syntax.

That kind of test is also really important in a web app.

You should not test everything, especially at the start of the project, those kinds of tests usually come with a high level of maintenance (i.e., when you change a screen you may have to change the test).

What I do is test the critical path and features. I made a reading app, so in my case, it was login, sign up, payment, access book, and access reader.

Why both Karma and Protractor? When do I use which?

Karma is a great tool for unit testing, and Protractor is intended for end to end or integration testing. This means that small tests for the logic of your individual controllers, directives, and services should be run using Karma. Big tests in which you have a running instance of your entire application should be run using Protractor. Protractor is intended to run tests from a user's point of view - if your test could be written down as instructions for a human interacting with your application, it should be an end to end test written with Protractor.

### Configuration

Navigate to your working directory in your terminal or command prompt and follow the below instructions.

* **Install AngularJS**
  + npm install angular --save
* **Install Karma**
  + npm install -g karma --save-dev
* **Install Jasmine**
  + npm install karma-jasmine jasmine-core --save-dev
* **Install ngMock**
  + [ngMock](https://docs.angularjs.org/api/ngMock) allows you to inject and mock angular services to help you test your application.
  + npm install angular-mocks --save-dev
* **Browsers**
  + Install browser launcher on which you want karma to run your tests. We need to install atleast one browser. I’ll use PhantomJs.
  + npm install karma-phantomjs-launcher --save-dev
* **Create two folders in  your working directory.**
  + mkdir app //your script files, controllers,filters etc.
  + mkdir tests //here we will keep our tests.
* **karma.conf.js**
  + karma init
  + Select Jasmine as your testing framework.
  + Select browser, I’ve selected PhantomJS.
  + Specify the paths to your js and spec files. Eg. 'app/\*.js', 'test/\*.js‘.
  + After answering a few more questions you should be done.
  + Open up your karma.conf.js and add the location of angular.js in to the files array.
    - node\_modules/angular/angular.js
  + Add the location for **ngMock** just below that.
    - node\_modules/angular-mocks/angular-mocks.js

### Introduction

*Why are you doing this?*

Angular was written from the ground up to be testable and yet there are scores of Angular developers who are not writing enough (if any) tests for their application. Why is this? I believe that it is because that while testing Angular is easy, actually getting to that first step is hard. I remember the feeling of despair I felt the first time I was tasked with writing tests for a project I was on. Where do I start? How do I get this Karma thing to actually run? Find my files? How do I instantiate my controller? What if it has a service dependency? Aaaaaaaaargh!

I have since covered a lot of ground over the last couple years and I found that once you understand a few basic rules of engagement that testing in Angular is actually pretty straight forward and formulaic.

I wanted to try to illustrate some of these basic patterns in an easy to read, approachable way so that developers can get to that first test without losing their mind

Please use this material as a bridge to making tests a natural part of your development process; and if you find something amiss, do not hesitate to create a pull request on the [repo](https://github.com/simpulton/angular-testing-quick-start-guide). Nothing would make me happier! Enjoy!

### The Sample Project

*Where can I see the actual tests?*

The companion repository for this quick start guide can be found here <https://github.com/simpulton/angular-testing-quick-start>

### Unit Testing vs E2E

*How does Karma differ from Protractor?*

I like to split my tests up into three different categories:

1. **End-to-End Tests** - These are the tests where you want to mimic an actual user that visits your website. Each test contains a series of simulated user events (ex. go to http://mysite.com/home and then click on the button with ID 'my-button') and expected results (ex. after 200ms a new window should appear that says "Thank You").
2. **Integration Tests** - These tests will call directly into your code. For example, you can use an integration test to call an Angular service. Typically each test will focus on one function. The test calls the target function with a set of parameters and then checks to make sure the results match expected values.
3. **Unit Tests** - These are the same as integration tests except you take extra steps toensure that nothing is executed besides the one function you are testing. For example, when you test a service that uses $http to call a back end API, an integration test would include the API call. A unit test, however, would use a utility we will discuss later called $httpBackend to replace $http so that the code executed by the test is restricted to just the target function.

Protractor is the tool you will use for end-to-end tests while Karma handles integration and unit testing.

### Installing Karma

*How do I install Karma?*

npm install -g karma

### Configuring Karma

*How do does Karma know which files to include?*

karma init karma.conf.js

After installing Karma, we need to set up the karma.conf file. The files array in karma.conf contains all the files needed to run a test. This includes:

* Any angular libraries used by your code
* All your custom code
* All your test specs

Below is an example of a full karma.conf file.

module.exports = function (config) {

config.set({

// base path that will be used to resolve all patterns (eg. files, exclude)

basePath: '',

// frameworks to use

// available frameworks: https://npmjs.org/browse/keyword/karma-adapter

frameworks: ['jasmine'],

// list of files / patterns to load in the browser

files: [

'src/vendor/angular.min.js',

'src/vendor/angular-mocks.js',

'src/vendor/angular-ui-router.min.js',

'src/assets/js/unit-test-helpers.js',

'src/app/\*\*/\*.html',

'src/app/\*\*/\*.spec.js',

'src/app/\*\*/\*.js'

],

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Note: this was added AFTER karma init was completed.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ngHtml2JsPreprocessor: {

stripPrefix: 'src/',

//stripSufix: '.ext',

// setting this option will create only a single module that contains templates

// from all the files, so you can load them all with module('foo')

moduleName: 'myAppTemplates'

},

// list of files to exclude

exclude: [],

// preprocess matching files before serving them to the browser

// available preprocessors: https://npmjs.org/browse/keyword/karma-preprocessor

preprocessors: {

'src/app/\*\*/\*.html': ['ng-html2js']

},

// test results reporter to use

// possible values: 'dots', 'progress'

// available reporters: https://npmjs.org/browse/keyword/karma-reporter

reporters: ['progress'],

// web server port

port: 9876,

// enable / disable colors in the output (reporters and logs)

colors: true,

// level of logging

// possible values: config.LOG\_DISABLE || config.LOG\_ERROR || config.LOG\_WARN || config.LOG\_INFO || config.LOG\_DEBUG

logLevel: config.LOG\_INFO,

// enable / disable watching file and executing tests whenever any file changes

autoWatch: true,

// start these browsers

// available browser launchers: https://npmjs.org/browse/keyword/karma-launcher

browsers: ['Chrome'],

// Continuous Integration mode

// if true, Karma captures browsers, runs the tests and exits

singleRun: false

});

};

Note that most values in the karma.conf file can be overridden at the command line. For example, if you wanted to run Firefox instead of Chrome you could either change the value in karma.conf or keep the value the same and use this command:

karma start --browsers Firefox

I strongly suggest you take an existing karma.conf file like this one and adapt it to meet your needs. As long as you follow the conventions we outline in this guide, the only things you will likely want to change are:

* files - to include all your custom code, dependencies and test code
* reporters - if you want test coverage you will need to include the 'coverage' reporter
* autoWatch and singleRun - most of the time you will want autoWatch=true and singleRun=false so that Karma will automatically re-run your tests as you make changes. However, if you are running Karma as part of a script like a git hook or continuous integration, then you will want to flip these two boolean values so that Karma only runs the tests once.

### Basic Jasmine Spec Structure

*How is a basic spec organized?*

File Structure

In general, you want to have one test file for each and every non-test code file in your app. You should have a common naming scheme so that your build tools and test runners can pick out test files from non-test files. We are using one of the most common test file naming schemes: "{filename}.spec.js". So if you have a code file called "app.js", the file that contains all the tests for app.js would be called "app.spec.js". You will often see all test files in a separate directory from the rest of the code (usually called 'test'), but in the sample code we have put all specs right along side the code they are testing for your convenience.

Spec Code Structure

In general, your spec files should follow this structure:

// src/app/app.spec.js

// Containing describe block (or "suite"), usually named for an app feature.

// In this case the feature is the App itself.

describe('Unit: App', function () {

// Include Modules

beforeEach(module('myApp'));

beforeEach(module('myApp.about'));

beforeEach(module('myApp.experiments'));

beforeEach(module('myApp.home'));

beforeEach(module('myApp.models.messages'));

beforeEach(module('ui.router'));

// Suite for testing an individual piece of our feature.

describe('App Abstract Route', function () {

// Instantiate global variables (global to all tests in this describe block).

var $state,

$rootScope,

state = 'app';

// Inject dependencies

beforeEach(inject(function (\_$state\_, $templateCache, \_$rootScope\_) {

$state = \_$state\_;

$rootScope = \_$rootScope\_;

$templateCache.put('app/home/home.tmpl.html', '');

}));

// It block (or "spec") to test expectations for the

// Expectations return true or false.

it('verifies state configuration', function () {

var config = $state.get(state);

expect(config.abstract).toBeTruthy();

expect(config.url).toBeUndefined();

});

});

});

Two things to note from this example.

First, you should make liberal use of before, beforeEach, after and afterEach to set up and tear down the appropriate context for tests. Ideally you only have a couple lines of code within each **it()** function.

The second thing to note is that the first parameter for the **describe()** and **it()** functions may be used by the test runner when tests are executed. For example, when this spec is run, some test runners may output:

Unit: App App Abstract Route verifies state configuration

So, make sure the string values are descriptive.

### Including a Module

*How do I inject a module in a spec?*

The first thing your spec should do is define all the Angular modules that are needed for the tests in that spec. This is done using the**module()** function that comes from the **angular-mocks** library. For example:

beforeEach(module('myApp'));

This code will enable the spec to test the code from the **myApp** module. It is best practice to use **beforeEach()** instead of just**before()** so that each test is essentially running from a blank slate. If you don't do this, the state from a previous test may bleed into another test and affect the results.

### Testing a Controller

*How do I instantiate a controller in a spec?*

Here is our controller that we want to test:

// src/app/about/about.js

.controller('AboutCtrl', function (Messages) {

var about = this;

about.title = 'About Page';

about.body = 'This is the about page body';

about.message = Messages.getMessage();

about.updateMessage = function (m) {

Messages.setMessage(m);

};

})

The **angular-mocks** library provides a service called **$controller** that we can use to help us test our controllers. In the **beforeEach()**below, we are injecting **$controller** along with any other dependencies we need to instantiate our controller.

// src/app/about/about.spec.js

describe('Unit: About', function () {

// Define global references for injections.

var ctrl, messages;

beforeEach(module('myApp.about'));

beforeEach(module('myApp.models.messages'));

beforeEach(module('ui.router'));

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

messages = \_Messages\_;

spyOn(messages, 'setMessage');

spyOn(messages, 'getMessage').and.returnValue('Hello!');

// Instantiate the controller with an object of the dependencies

ctrl = $controller('AboutCtrl', {

Messages: messages

});

}));

describe('AboutCtrl', function () {

// Test some basic expectations about the controller

it('should have title defined', function () {

expect(ctrl.title).toBeDefined();

});

it('should have body defined', function () {

expect(ctrl.body).toBeDefined();

});

it('should call Messages.getMessage', function () {

expect(messages.getMessage).toHaveBeenCalled();

expect(ctrl.message).toEqual('Hello!');

});

it('should call updateMessage on message', function () {

var message = 'Hello Message';

ctrl.updateMessage(message);

expect(messages.setMessage).toHaveBeenCalledWith(message);

});

});

});

Note that we are using the underscore syntax with **\_Messages\_** to get a global reference to the **Messages** service. The underscores are ignored by the injector when the reference name is resolved.

<https://docs.angularjs.org/api/ngMock/function/angular.mock.inject>

### Including a Service

*How do I inject a service in a spec?*

Testing an Angular service is a piece of cake. You can use the **inject()** function from angular-mocks to get a reference to either internal Angular core objects or any of your custom objects in the modules that are defined at the top of the spec. For example:

beforeEach(inject(function(someService, someFactory) {

localSomeService = someService;

localSomeFactory = someFactory;

}));

As a best practice, we suggest injecting objects in **beforeEach()** and saving the object to a local variable. Then in the test we just reference that local variable.

### Testing a Template

*How do I test if an element is in the DOM?*

The most tricky thing to test with Angular is code within templates. That is why you should try to reduce the amount of code in your templates as much as possible. Even if you are really good about this, though, you will always have some template code that you want to test.

You can split template testing into two categories. The first category includes templates that don't have any controllers, includes or custom directives. Essentially you are just testing logic that uses basic Angular expressions and core Angular directives. This use case is relatively easy to test. Take the following example:

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

<h1>{{home.title}}</h1>

<p>{{home.body}}</p>

<hr>

<div>

<h2 class="text-error">Home: {{home.message}}</h2>

<form class="form-inline"><input class="form-control" placeholder="Message" type="text">

<button class="btn btn-default" type="submit">Update Message</button>

</form>

</div>

</body>

</html>

We want to have access to this HTML file in our tests, but it is generally a bad idea to make an http call within your tests if you can avoid it. Fortunately, there is a nice feature in Karma that allows us to automatically package all of our HTML template files into an Angular module that we can access easily in our tests. Just add the following to your **karma.conf.js** file:

// karma.conf.js

ngHtml2JsPreprocessor: {

stripPrefix: 'src/',

// stripSufix: '.ext',

// setting this option will create only a single module that contains templates

// from all the files, so you can load them all with module('foo')

moduleName: 'myAppTemplates'

}

This will automatically package any file ending in .html within your src/ folder into an Angular module called myAppTemplates. Each template is accessible by using the **$templateCache** service. You can can test this template simply by injecting the **$compile** service with some test data and then checking the resulting HTML:

describe('UNIT test myTemplates', function() {

beforeEach(module('myAppTemplates'));

describe('home template', function() {

var homeTemplate,

compile;

beforeEach(inject(function($templateCache, $compile) {

homeTemplate = $templateCache.get('app/home/home.tmpl.html');

compile = $compile;

}));

it('should have an h1 with my title', function() {

var testScope = {

home: {

title: 'blah'

}

};

var element = compile(homeTemplate)(testScope);

expect(element.find('h1').text()).toBe('blah');

});

});

});

The second category of template testing is unfortunately more complex. We need to deal with dependencies within the template or surrounding the template such as a controller, the UI router, directives, etc. Let's look at one example of a more complex use case that includes a UI Router state and a controller:

// app/home/home.js

.config(function ($stateProvider) {

$stateProvider

.state('about', {

url: '/about',

templateUrl: 'app/about/about.tmpl.html',

controller: 'AboutCtrl as about'

})

;

})

.controller('AboutCtrl', function (Messages) {

var about = this;

about.title = 'About Page';

about.body = 'This is the about page body';

about.message = Messages.getMessage();

about.updateMessage = function (m) {

Messages.setMessage(m);

};

})

In order to test the template and associated controller code, we need to instantiate the controller and surrounding context. To that end, we have created a helper function called **compileRouteTemplateWithController** that does everything we need.

function compileRouteTemplateWithController($injector, state) {

var $rootScope = $injector.get('$rootScope');

var $templateCache = $injector.get('$templateCache');

var $compile = $injector.get('$compile');

var $state = $injector.get('$state');

var $controller = $injector.get('$controller');

var scope = $rootScope.$new();

var stateDetails = $state.get(state);

var html = $templateCache.get(stateDetails.templateUrl);

var ctrl = scope.home = $controller('HomeCtrl');

$rootScope.$digest();

var compileFn = $compile(angular.element('<div></div>').html(html));

return {

controller: ctrl,

scope: scope,

render: function () {

var element = compileFn(scope);

$rootScope.$digest();

return element;

}

};

}

We are using this helper function to get all the dependencies we need to run our tests. This includes creating scope, a controller and a render function. Feel free to use this and adapt it to your needs. Also, as an aside, whenever you see a lot of repetitive code within your tests, make sure you create your own helper functions.

OK, now we have everything in place to test our template.

describe('home page', function () {

// Define global references for injections.

var element, render, ctrl, scope;

beforeEach(inject(function ($injector) {

// Call the helper function that "creates" a page.

// This just creates references to the attributes

// on the returned object for use in this suite.

var routeDetails = compileRouteTemplateWithController($injector, 'home');

ctrl = routeDetails.controller;

scope = routeDetails.scope;

render = function () {

element = routeDetails.render();

};

}));

// Test your expectations. You can use ordinary jQuery methods

it('should render the page title', function () {

scope.home.title = 'Hello';

render();

expect(element.find('h1').text()).toBe('Hello');

});

it('should have body defined', function () {

scope.home.body = 'body...';

render();

expect(element.find('p').text()).toBe('body...');

});

it('should call Messages.getMessage', function () {

render();

expect(Messages.getMessage).toHaveBeenCalled();

ctrl.updateMessage('yo!');

expect(Messages.setMessage).toHaveBeenCalled();

});

it('should call Messages.setMessage when submit is clicked', function () {

render();

element.find('input').val('Lukas');

element.find('input').triggerHandler('input');

scope.$digest();

element.find('button').triggerHandler('click');

scope.$digest();

expect(Messages.getMessage).toHaveBeenCalled();

expect(scope.home.message).toEqual('Lukas');

});

it('should call updateMessage on message', function () {

var message = 'Hello Message';

ctrl.updateMessage(message);

expect(Messages.setMessage).toHaveBeenCalledWith(message);

});

});

The key part of this test was the use of the helper function to instantiate the template and all required dependencies:

var routeDetails = compileRouteTemplateWithController($injector, 'home');

ctrl = routeDetails.controller;

scope = routeDetails.scope;

render = function () {

element = routeDetails.render();

};

### Testing a Route

*How do I test route changes?*

Testing a route essentially means testing that we configured the UI router $stateProvider correctly during the config phase. For example, given the following state configuration:

// app/about/about.js

.config(function ($stateProvider) {

$stateProvider

.state('about', {

url: '/about',

templateUrl: 'app/about/about.tmpl.html',

controller: 'AboutCtrl as about'

})

;

})

Our basic strategy for testing is to use the **$state.go()** and **$state.href()** methods to modify the current state and then check to make sure the route is changed appropriately.

// app/about/about.spec.js

describe('About Route', function () {

// Define global references for injections

var $state,

$rootScope,

state = 'about';

// Inject and assign the $state and $rootScope services.

// Put the template in template cache.

beforeEach(inject(function (\_$state\_, $templateCache, \_$rootScope\_) {

$state = \_$state\_;

$rootScope = \_$rootScope\_;

$templateCache.put('app/about/about.tmpl.html', '');

}));

// Test whether the url is correct

it('should respond to URL', function() {

expect($state.href(state)).toEqual('/about');

});

// Test whether our state activates correctly

it('should activate the state', function() {

$state.go(state);

$rootScope.$digest();

expect($state.current.name).toBe(state);

});

});

### Testing a Directive

*How do I set up a spec for a directive?*

Similar to how we test a template, we use the $compile service to help us test a directive. The key is to pass in a HTML snippet to**$compile()** that refers to the target directive. For example, if you had an element directive called 'experiment', you would simply call**$compile("<experiment></experiment>")**. You can see the full example here:

// app/experiments/experiments.js

.directive('experiment', function(){

var linker = function (scope, element, attrs) {

element.on('click', function(){

scope.doExperiment();

})

};

var controller = function($scope){

$scope.doExperiment = function() {

$scope.$apply(function(){

$scope.experiment.completed++;

});

};

};

return {

scope: true,

restrict: 'E',

template: '<div class="experiment">' +

'<h3>{{experiment.name}}</h3>' +

'<p>{{experiment.description}}</p>' +

'<p><strong>{{experiment.completed}}</strong></p>' +

'</div>',

link: linker,

controller: controller

}

})

And the spec.

// app/experiments/experiments.spec.js

describe('Experiments Directive', function () {

// Define global references for injections

var element, experiment;

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

// Mock out an experiment

$rootScope.experiment = {

"name": "Experiment 1",

"description": "This is an experiment",

"completed": 0

};

// Create an html element with the directive

element = angular.element('<experiment></experiement>');

// Compile the element and $rootScope,

// which essentially instantiates the directive as it would be on a web page.

$compile(element)($rootScope);

}));

it('should increment experiment completed count', function () {

// Get the directive's scope

var localScope = element.scope();

// Test some expectations about the directive.

expect(localScope.experiment.completed).toBe(0);

localScope.doExperiment();

expect(localScope.experiment.completed).toBe(1);

});

})

Note: depending on what your directive does, you may need to modify the HTML passed into **$compile**. For example, if the directive expects other attributes on the element or if you are testing a directive with transclusion (in which case you will want to put different snippets of HTML as children to the element that has the directive).

### Mocking

*How/Why do I mock a service call in a controller spec?*

We mentioned at the beginning that you can create both unit tests and integration tests with Karma. When you are writing a unit test, your goal is to test just one thing and either eliminate or mock out all other dependencies. In general, you can mock any object through the use of the **$provide** service. You use this service when you define the module you are using in your spec. For example:

beforeEach(module("app", function($provide) {

$provide.value("SimpleService", {

someFn: function() {}

});

}));

In this case, we are overriding the SimpleService object. Once we do this, any code that injects SimpleService will get our mock object instead of the actual SimpleService.

There is one special case with mocking where Angular helps you out. Whenever you use **$http** to make a remote call, Angular has another service behind the scenes called **$httpBackend** that actually does all the hard work. The angular-mocks library has its own version of **$httpBackend** with a number of goodies to help us mock out calls to the back end. For example look at this code which makes an **$http** call:

// app/experiments/experiments.js

.service('Experiments', function($http) {

var service = this;

service.getExperiments = function() {

return $http.get('data/experiments.json');

};

})

If we call **getExperiments()** in our test, it will make an actual http request to data/experiments.json. We can intercept that call with**$httpBackend**, however, and define what should be returned instead of making a remote call.

// app/experiments/experiments.spec.js

describe('Experiments Model', function () {

// After every spec, do the following:

afterEach(inject(function ($httpBackend) {

// Make sure we have flushed all of our requests.

$httpBackend.verifyNoOutstandingExpectation();

$httpBackend.verifyNoOutstandingRequest();

}));

it('Should get experiments', inject(function (Experiments, $httpBackend, $rootScope) {

var mockResponse = [];

// Every time we hit the specified url,

// respond with mockResponse( in this case an empty array).

$httpBackend.when('GET', 'data/experiments.json').respond(mockResponse);

// The promise reference now holds the $http call returned

var promise = Experiments.getExperiments();

// Flush the backend

$httpBackend.flush();

// Use .then() like you would normally.

promise.then(function (result) {

expect(result.data).toEqual(mockResponse);

});

// Manually trigger a $digest cycle

$rootScope.$digest();

}));

});

Note that **$httpBackend.flush()** is needed because normally $http is asynchronous, but we want to execute our test in a synchronous fashion. The call to **flush()** will ensure that the **.then()** on the promise returned from $http will be executed immediately.

### Spying on Methods

*How do I determine if one method successfully calls another method?   
 How do I determine what arguments were included when I call a method?*

Jasmine uses a spy to determine whether a method has been called and/or what arguments are set into a method call. So, for example:

var foo, bar = null;

beforeEach(function() {

// Here we create a simple object foo with a method setBar, which takes one argument.

foo = {

setBar: function(value) {

bar = value;

}

}

};

// Now whenever foo.setBar is called in this spec or in the source code,

// we can perform assertions on it.

spyOn(foo, 'setBar');

foo.setBar(123);

it("tracks that the spy was called", function() {

expect(foo.setBar).toHaveBeenCalled();

});

it("tracks the arguments of its calls", function() {

expect(foo.setBar).toHaveBeenCalledWith(123);

});

A spy only exists within the **describe()** or **it()** function where it has been defined.

### Advanced Spying

In addition to simply seeing if a spy has been called, we can also define what value the spy should return (using returnValue()) or what fake function the spy should run instead of the target function (using callFake()). For example:

// app/experiments/experiments.js

.controller('ExperimentsCtrl', function (Messages, Experiments) {

var experiments = this;

experiments.title = 'Experiments Page';

experiments.body = 'This is the about experiments body';

experiments.message = Messages.getMessage();

Experiments.getExperiments()

.then(function(result){

experiments.experiments = result.data;

});

experiments.updateMessage = function (m) {

Messages.setMessage(m);

};

})

And the spec.

// app/experiments/experiments.spec.js

describe('ExperimentsCtrl', function () {

// Inject and assign services and create the controller.

beforeEach(inject(function ($controller, \_Messages\_, \_Experiments\_, $q) {

messages = \_Messages\_;

experiments = \_Experiments\_;

// Here we create a spy just like before,

// but now it is spying on a method attached to a service.

spyOn(messages, 'setMessage');

// Every time Messages.getMessage is called,

// we want the value 'Hello!' to be returned.

spyOn(messages, 'getMessage').and.returnValue('Hello!');

// Every time Messages.getExperiments is called,

// we execute the function contained within the callFake method.

spyOn(experiments, 'getExperiments').and.callFake(

function () {

// Since we don't want to deal with an $http call

// in the controller, we mock it out using $q.

var deferred = $q.defer();

deferred.resolve({data: []});

return deferred.promise;

}

);

ctrl = $controller('ExperimentsCtrl', {

Messages: messages,

Experiments: experiments

});

}));

// Now we can see if the methods we are spying on actually execute;

// also, we can test some expectations on the portions of the controller

// that interact with our spies.

it('should call Messages.getMessage', function () {

expect(messages.getMessage).toHaveBeenCalled();

expect(ctrl.message).toEqual('Hello!');

});

it('should call updateMessage on message', function () {

var message = 'Hello Message';

ctrl.updateMessage(message);

expect(messages.setMessage).toHaveBeenCalledWith(message);

});

});

### Testing a Promise

*How do I handle async operations in a spec?*

By default, each test runs synchronously. So, if you have any asynchronous operation, the test will complete before the operation completes. There are ways of handling specific use cases (for example **$httpBackend.flush()** as mentioned earlier), but you can also use the Jasmine **done()** function. For example:

it('should do something', function(done) {

someAsyncOperation

.then(function() {

done();

})

.catch(function(err) {

done(err);

});

});

In this example, the test will not complete until **done()** is called. If **done()** contains a parameter, Jasmine treats that as an error and fails the test.

One last note on async operations. You may have noticed in our examples a call to **$rootScope.$digest()**. This will force the digest cycle to run which is needed whenever we are testing anything athat involves watchers (so, anything with templates).

### Debugging

*How do I troubleshoot problems with my specs?*

Spec code is run in the browser just like any other client side code. So, how do you debug your Angular app? That's right, the Chrome/FireFox dev tools. For example, after running Karma with Chrome there should be a Chrome window open on your machine that contains the output of the test. To debug, simply open up the Chrome dev tools and refresh the page.

### Best Practices

*How do I know I am formatting my specs in the most efficient manner?*

Here is a quick list of best practices. Some of these we touched on earlier.

* Use **beforeEach()** to set up the context for your tests.
* Make sure the string descriptions you put in **describe()** and **it()** make sense as output
* Use **after()** and **afterEach()** to cleanup your tests if there is any state that may bleed over.
* If any one test is over 10 lines of code, you may need to refactor the test
* If you find yourself repeating the same code for many tests, refactor the common code into a helper function

Theory url’s:

<http://angulartestingquickstart.com/>

<https://blog.logentries.com/2015/01/unit-testing-with-karma-and-jasmine-for-angularjs/>

<https://www.airpair.com/angularjs/posts/testing-angular-with-karma>

<http://www.bradoncode.com/blog/2015/05/19/karma-angularjs-testing/>

<https://docs.angularjs.org/guide/unit-testing>

Video URL’s

<https://www.youtube.com/watch?v=ejHUJvukhOI&list=PLqL-uCNdScf9wnt4YUKK5cTVpjb-qicVW&index=4>

<https://www.youtube.com/watch?v=NRzuSX1lpls>

<https://www.youtube.com/watch?v=UDB-jm8MWro>

<https://www.youtube.com/watch?v=2MaG5snoOK4&index=3&list=PLtpFIFgmB4mdlhRrVI1hWOL6B_T27VORR>