Introduction to Angular test-driven development

## **Introduction**

If you have ever taken the step to building something more complex than a simple To-Do List application with AngularJS or any other app-development technology, you know that implementing new features often involves having to fix older parts of your code. This is a common problem with many technologies; no matter the language your software is built with, you will most definitely break things that previously worked when releasing new builds.

This is where [Test-Driven Development](http://hackguides.org/), or unit testing, comes into play.

Unit testing helps you answer all the 'Is this going to work if I insert X?' or Does X still return the same results now that I have implemented Y?' scenarios. It does it by simulating actions, or **cases**, with mock data and checking if everything is outputted in the format you expect it to be.

This guide is a great starting point for your journey in testing Angular applications and unit testing in general. In order to get the most out of the guide, you need to have some knowledge of JavaScipt and building Angular applications.

Angular is built with [testing in mind](https://docs.angularjs.org/guide/unit-testing). The framework allows simulation of server-side requests and abstraction of the [Document Object Model](https://en.wikipedia.org/wiki/Document_Object_Model) (DOM), thus providing an environment for testing out numerous scenarios. Additionally, Angular's dependency injection allows every component to be mocked and tested in different scopes.

We'll start off by setting up the environment and looking at how [Karma](https://karma-runner.github.io/1.0/index.html), [Jasmine](http://jasmine.github.io/2.4/introduction.html), and [Angular Mocks](https://docs.angularjs.org/api/ngMock) work together to provide an easy and seamless testing experience in Angular. Then, we will have a look at the different building blocks of tests. Lastly, we will use the newly acquired knowledge to build sample tests for controllers, services, directives, filters, promises and events.

Every great developer knows his or her tools' uses. Understanding your tools for testing is essential before diving into writing tests.

## **Karma, Jasmine, and Angular Mocks**

**Karma** is an environment which runs the tests of your application. Simply put, it's your testing server. Originally started as a [university thesis](https://github.com/karma-runner/karma/raw/master/thesis.pdf), Karma aims to make a framework-agnostic environment that automates the running of your unit tests. In its core, it is a [Node](https://nodejs.org/en/) server that watches for changes in your testing and application files, and when such changes occur, it runs them in a browser and checks for mistakes.

**Jasmine**is an unit testing framework for JavaScript. It is the most popular framework for testing JavaScript applications, mostly because it is quite simple to start with and flexible enough to cover a wide range of scenarios.

**Angular Mocks**is an Angular module that is used to mock components that already exist in the application. Its role is to inject various components of your Angular application (controllers, services, factories, directives, filters) and make them available for unit tests. It can be said that Angular Mocks is the middleman between the Angular components in your application and the unit testing environment.

## **Setting up the testing environment**

To ensure a speedy and efficient setup, I recommend using the [Node Package Manager (npm)](https://www.npmjs.com/) to maintain the dependencies of your Angular project.

First, install the Karma CLI globally. We'll need this to be able to run the karma command directly from the command line.

1

npm install -g karma-cli

bash

Then, create a directory where you'll store your project files. Open your terminal and run the following commands:

1

2

mkdir myitemsapp

cd myitemsapp

bash

### **Installing packages**

Once you are in the directory, start setting up your project dependencies. First, initialize your package.json file:

1

npm init

bash

You will get asked several questions regarding your project's details. You can skip them and simply copy the contents below in your package.json file.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

{

"name": "myitemsapp",

"version": "1.0.0",

"description": "",

"main": "karma.conf.js",

"directories": {

"test": "tests"

},

"dependencies": {

"angular": "^1.5.7",

"save": "^2.3.0"

},

"devDependencies": {},

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

json

Then, start installing your project's dependencies, starting with Angular:

1

2

3

4

npm install angular --save

npm install karma --save-dev

npm install karma-jasmine jasmine-core --save-dev

npm install angular-mocks --save-dev

bash

Next, you have to choose a browser launcher for Karma to use: You can use one for [Chrome](https://app.pluralsight.com/guides/introduction-to-angular-test-driven-development)(npm install karma-chrome-launcher --save-dev) , [Firefox](https://github.com/karma-runner/karma-firefox-launcher) , [Internet Explorer](https://github.com/karma-runner/karma-ie-launcher), [Opera](https://github.com/karma-runner/karma-opera-launcher), [PhantomJS](https://github.com/karma-runner/karma-phantomjs-launcher) and others.

I will go with Google Chrome:

1

npm install karma-chrome-launcher --save-dev

bash

### **Configuring Karma**

Configure your testing environment using a configuration file (karma.conf.js). This is similar to configuring package.json, which we used to configure the project environment.

There are [many options](http://karma-runner.github.io/1.0/config/configuration-file.html) for configuring Karma. The most essential ones are the following:

* **Frameworks** - the testing frameworks that are going to be used. In this guide, we are using Jasmine.
* **Files** - files that will be used for the tests. You would normally include both the framework files (in this case, Angular) as well as the project and the testing files themselves.
* **Browsers** - You can specify which browsers must Karma use in order to run its tests.

Before starting, create two folders in your working directory.

1

2

mkdir app

mkdir tests

We'll use app to store our applicaton code and tests to keep our tests. This way, we'll keep the two separated. In more advanced cases, you would opt for a more generic approach by using wildcards and storing the test and application files together.

With your terminal, run the following command in the directory of the project:

1

karma init

bash

Answer the questions as following:

* Testing framework: Select jasmine.
* Do you want to use Require.js?: no
* Browser: Chrome.
* Specify the paths to your js and spec files: Enter each of the following, pressing Enter after each one. After the last one is entered, press enter again:
  + app/\*\*.js
  + tests/\*\*.js
    - **Note**: You will get a warning in your console after each of these saying that there are no files that match this pattern. That's ok, it's normal.
  + node\_modules/angular/angular.js
  + node\_modules/angular-mocks/angular-mocks.js
* Any excluded files? Just press Enter
* Do you want Karma to Watch all files and run tests on change? yes

In the end, you should end up with a karma.conf.js file looking like this:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

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: [

"node\_modules/angular/angular.js",

"node\_modules/angular-mocks/angular-mocks.js",

"app/\*\*.js",

"tests/\*\*.js"

],

// 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: {},

// 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,

// Concurrency Level

// how many browser should be started simultaneous

concurrency: Infinity

});

};

javascript

Once you are done with this step, you are ready to dive into testing features.

### **Running your tests**

To run your tests, you can run either of these commands in the terminal:

1

2

karma start

npm test

## **Building blocks**

Writing tests revolves around events and their expected outcomes. When you start writing the tests for your application, you must always think in terms of the desired outcomes of your app components' processes.

Although the terms that are going to be used are specific to Jasmine, they share many similarities with terms used with other behavior-driven testing frameworks:

* **Suites** - the top-level element of the testing framework. They accept a title (explanation) and a function containing one or more specifications.

describe(string, function)

* **Specs** - constructs that take a title and a function containing one or more expectations. Specs are nested into suites.

it(string, function)

* **Expectations** — assertions that evaluate to true or false.

expect(actual).toBe(expected)

* **Matchers** - redefined helpers for common assertions. They are the constructs that do the evaluations and decide whether a test has failed or not.

toEqual(expected)

To give you a better idea of what matchers can do, **here is a list of the most essential matchers**:

1

2

3

4

5

6

7

8

9

10

11

12

13

expect(fn).toThrow(e);

expect(instance).toBe(instance);

expect(mixed).toBeDefined();

expect(mixed).toBeFalsy();

expect(number).toBeGreaterThan(number);

expect(number).toBeLessThan(number);

expect(mixed).toBeNull();

expect(mixed).toBeTruthy();

expect(mixed).toBeUndefined();

expect(array).toContain(member);

expect(string).toContain(substring);

expect(mixed).toEqual(mixed);

expect(mixed).toMatch(pattern);

javascript

[The full list of matchers is also available on GitHub.](https://github.com/JamieMason/Jasmine-Matchers)

## **Teardown**

Teardowns are а building block fоr tests and are used to "prepare" the code for its specs in a particular suite. Suppose that before or after each spec (i.e a describe function), certain setups have to be done so that you can test a function. Instead of doing it for every spec, you can write a beforeEach or an afterEach function in your suite to do that.

1

2

3

4

5

6

7

8

// single line

beforeEach(module("itemsApp"));

// multiple lines

beforeEach(function() {

module("itemsApp");

//...

});

javascript

In the block above, you can see two ways in which you can initialize the Angular module in a testing suite using teardown.

## **Injecting**

Injecting is essential feature of unit testing Angular applications. It is characterised by the inject function (provided by the Angular mocks module). Injecting can be regarded as a way of accessing Angular's built-in constructs or your application's constructs by giving the correct arguments. This gives you the ability to access the code of your application and put mock data through it in order to test it.

Injecting gives access to Angular's building blocks -- $service, $controller, $filter , $directive, $factory. Additionally, it allows to mock built-in variables such as $rootScope and $q. It also provides $httpBackend, which simulates server-side requests in the testing envirionemnt.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

// Using \_serviceProvider\_ notation

var $q;

beforeEach(

inject(function(\_$q\_) {

$q = \_$q\_;

})

);

// Using $injector

var $q;

beforeEach(

inject(function($injector) {

$q = $injector.get("$q");

})

);

// Using an alias Eg: $$q, q, \_q

var $$q;

beforeEach(

inject(function($q) {

$$q = $q;

})

);

javascript

In the code snippet above, you can see three ways of injecting and instantiating the Angular [$q](https://docs.angularjs.org/api/ng/service/$q) variable, which is used for asynchronous requests, in the testing environment.

## **Testing a controller**

Time to put all this knowledge to use. Let's do go through some scenarios that you might encounter when writing your tests:

First, let's instantiate an Angular applicaiton in app.js.

**Code**

1

2

3

// app/app.js

angular.module("ItemsApp", []);

javascript

Then, write a simple controller:

1

2

3

4

//app/app.js

angular.module('ItemsApp', [])

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

$scope.title = 'Hello Pluralsight';

javascript

We have a controller with one simple $scope variable attached to it. Let's write a test to see if the scope variable contains the value we expect it to have:

**Test**

1

2

3

//tests/tests.js

// Suite

describe("Testing a Hello Pluralsight controller", function() {});

javascript

Let's go through building our first test step-by-step. First, we use the describe function to make a new testing suite for the controller. It contains a message as its first argument and a function that is going to contain the tests as the second argument.

Next, let's inject the controller in the suite:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

//tests/tests.js

describe("Testing a Hello Pluralsight controller", function() {

var $controller;

// Setup for all tests

beforeEach(function() {

// loads the app module

module("ItemsApp");

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

// inject removes the underscores and finds the $controller Provider

$controller = \_$controller\_;

});

});

});

javascript

Here, using beforeEach, we define our teardown flow. Before each of the tests, we are going to inject the controller provider from Angular and make it available for testing.

Next, we'll move to the the tests themselves:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

//tests/tests.js

// Suite

describe("Testing a Hello Pluralsight controller", function() {

var $controller;

// Setup for all tests

beforeEach(function() {

// loads the app module

module("ItemsApp");

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

// inject removes the underscores and finds the $controller Provider

$controller = \_$controller\_;

});

});

// Test (spec)

it("should say 'Hello Pluralsight'", function() {

var $scope = {};

// $controller takes an object containing a reference to the $scope

var controller = $controller("MainCtrl", { $scope: $scope });

// the assertion checks the expected result

expect($scope.title).toEqual("Hello Pluralsight");

});

// ... Other tests here ...

});

javascript

We start our first test specification (spec) with the it function. The first argument is a message, and the second argument is the function with the test. First, we instantiate MainCtrl that we created in the application. Then, we use a matcher to check if the $scope.title variable is equal to the value we assigned in the application.

## **Testing a service**

Next, we are going to add a service to our application in order to test it. We'll write a service with one method, get() that returns an array. Just below your controller code, add the following snippet:

**Code**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

//app/app.js

angular.module('ItemsApp', [])

//..

// MainController

//..

.factory('ItemsService', function(){

var is = {},

\_items = ['hat', 'book', 'pen'];

is.get = function() {

return \_items;

}

return is;

})

Here is how we are going to test it:

**Test**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

describe('Testing Languages Service', function(){

var LanguagesService;

beforeEach(function(){

module('ItemsApp');

inject(function($injector){

ItemsService = $injector.get('ItemsService');

});

});

it('should return all items', function() {

var items = ItemsService.get();

expect(items).toContain('hat');

expect(items).toContain('book');

expect(items).toContain('pen');

expect(items.length).toEqual(3);

});

});

Even though we have only one spec, we stick to the good practice of using beforeEach in our suite in order to instantiate what we need.

In the spec, we use the toContain matcher to check the contents of the array that the get() method returns. In the end, we use languages.length with the toEqual matcher to check the length of the array.

## **Testing directives**

Directives differ in terms of purpose and structure than services and controllers. Thus, they are tested using different approach.

Directives have their own encapsulated scope which gets its data from an outer scope, a controller. Directives first get "compiled" (in the Angular.js sense), and then their scope gets filled with data. To properly test them, we must simulate the same process and see if we get the desirable outcomes.

We are going to add a simple directive that will display the user's profile. It will get its profile data from outside and apply it into its scope.

**Code**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

//app/app.js

angular

.module("ItemsApp", [])

//rest of the app

.directive("userProfile", function() {

return {

restrict: "E",

template: "<div>{{user.name}}</div>",

scope: {

user: "=data"

},

replace: true

};

});

javascript

**Test**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

//tests/tests.js

describe("Testing user-profile directive", function() {

var $rootScope, $compile, element, scope;

beforeEach(function() {

module("ItemsApp");

inject(function($injector) {

$rootScope = $injector.get("$rootScope");

$compile = $injector.get("$compile");

element = angular.element('<user-profile data="user"></user-profile>');

scope = $rootScope.$new();

// wrap scope changes using $apply

scope.$apply(function() {

scope.user = { name: "John" };

$compile(element)(scope);

});

});

});

it("Name should be rendered", function() {

expect(element[0].innerText).toEqual("John");

});

});

javascript

The code might be confusing at first. Upon second glance, however, it's pretty consistent with Angular basics:

1. You create a scope to which you apply the directive. First, you get the $rootScope and create a new one using $rootScope.$new().
2. Once you have a scope, you create a new DOM element using angular.element().
3. You attach the element to the DOM using [$compile](https://docs.angularjs.org/api/ng/service/$compile). $compile will get the element and apply a scope to it. It's essentially the function that parses the HTML element, finds the directive it corresponds to, and attaches its template and scope into the outer scope.
4. We have to simulate an action of putting data in a scope, but there is no browser and no DOM. In such cases, we use [$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope). It is used to add a variable to the scope ({name: 'John'}) to a scope.

When you're done, the element variable will be filled with HTML generated by the directive in your code. All you need to do is use a spec to test if it returned the result you wanted.

## **Testing filters**

Filters are used to transform data in Angular applications. Compared to other constructs, they are relatively easy to test. For this guide, we are going to create a filter that reverses strings:

**Code**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

//app/app.js

angular

.module("ItemsApp", [])

//rest of the app

.filter("reverse", [

function() {

return function(string) {

return string

.split("")

.reverse()

.join("");

};

}

]);

javascript

**Test**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

//tests/tests.js

describe("Testing reverse filter", function() {

var reverse;

beforeEach(function() {

module("ItemsApp");

inject(function($filter) {

//initialize your filter

reverse = $filter("reverse", {});

});

});

it("Should reverse a string", function() {

expect(reverse("rahil")).toBe("lihar");

expect(reverse("don")).toBe("nod");

//expect(reverse('jam')).toBe('oops'); // this test should fail

});

});

javascript

Here, we inject the filter and assign it to a variable reverse. Then, we use the variable to call the filter and test whether the result is reversed.

## **Testing promises**

Promises are the standard tool for handling client-server communication. Angular services such as [ngResource](https://docs.angularjs.org/api/ngResource/service/$resource) and [$http](https://docs.angularjs.org/api/ng/service/$http) use promises to interact with the back-end service.

For this example, we are going to take the ItemsService and mock it so that it simulates server-side interaction.

**Code**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

//app/app.js

angular

.module("ItemsApp", [])

//rest of the app

.factory("ItemsServiceServer", [

"$http",

"$q",

function($http, $q) {

var is = {};

is.get = function() {

var deferred = $q.defer();

$http

.get("items.json") //'items.json will be mocked in the test'

.then(function(response) {

deferred.resolve(response);

})

.catch(function(error) {

deferred.reject(error);

});

return deferred.promise;

};

return is;

}

]);

javascript

The service structure is simple -- you have a method get() that uses [$http](https://docs.angularjs.org/api/ng/service/$http) with a promise ([$q](https://docs.angularjs.org/api/ng/service/$q)) to resolve the request. If the request succeeds, the promise is resolved; if the request fails, the promise is rejected.

**Test**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

//tests/tests.js

describe("Testing Items Service - server-side", function() {

var ItemsServiceServer,

$httpBackend,

jsonResponse = ["hat", "book", "pen"]; //this is what the mock service is going to return

beforeEach(function() {

module("ItemsApp");

inject(function($injector) {

ItemsServiceServer = $injector.get("ItemsServiceServer");

// set up the mock http service

$httpBackend = $injector.get("$httpBackend");

// backend definition response common for all tests

$httpBackend

.whenGET("items.json") //must match the 'url' called by $http in the code

.respond(jsonResponse);

});

});

it("should return all items", function(done) {

// service returns a promise

var promise = ItemsServiceServer.get();

// use promise as usual

promise.then(function(items) {

// same tests as before

expect(items.data).toContain("hat");

expect(items.data).toContain("book");

expect(items.data).toContain("pen");

expect(items.data.length).toEqual(3);

// Spec waits till done is called or Timeout kicks in

done();

});

// flushes pending requests

$httpBackend.flush();

});

});

javascript

As you can see, there are several differences in testing a service with and without promises. Here, Angular mocks' [$httpBackend](https://docs.angularjs.org/api/ngMock/service/$httpBackend) is used to simulate a server-side request. First, we instantiate a variable jsonResponse that has to be structured in the same way we expect to retreive the server-side data.

Then, in the beforeEach block, we inject $httpBackend and use whenGET() to assign jsonResponse as the response of items.json.

The testing part is similar to the one we did with the service. Note that items.data is used instead of items because the data of the response (jsonResponse) is contained into the data property of the response object.

We call done() to finish the test after the promise returns. **If**done()**is not called, then the promise has timed out, and the test fails**.

We end the spec with a $httpBackend.flush(), which lets $httpBackend to respond to other request directed to it in the rest of the tests.

## **Testing events**

The last thing we'll test will be events. Events can be spawned with $broadcast and caught with $on . We'll make a service that will broadcast when a new item is added. We'll test if the event is broadcast, if it's caught by a controller, and if its contents match the contents of the broadcast.

**Code**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

//app/app.js

angular

.module("ItemsApp", [])

//rest of the app

.factory("appBroadcaster", [

"$rootScope",

function($rootScope) {

var abc = {};

abc.itemAdded = function(item) {

$rootScope.$broadcast("item:added", item);

};

return abc;

}

]);

javascript

In your MainController, add a listener:

1

2

3

4

5

6

7

8

9

10

11

12

13

//app/app.js

angular

.module("ItemsApp", [])

//rest of the app

//update MainCtrl by injecting $rootScope and adding a listener

.controller("MainCtrl", function($scope, $rootScope) {

$scope.title = "Hello Pluralsight";

$rootScope.$on("item:added", function(event, item) {

$scope.item = item;

});

});

javascript

**Test**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

//tests/tests.js

describe("appBroadcaster", function() {

var appBroadcaster,

$rootScope,

$scope,

$controller,

item = { name: "Pillow", id: 1 }; //what is going to be broadcast

beforeEach(function() {

module("ItemsApp");

inject(function($injector) {

appBroadcaster = $injector.get("appBroadcaster"); //get the service

$rootScope = $injector.get("$rootScope"); //get the $rootScope

$controller = $injector.get("$controller");

$scope = $rootScope.$new();

});

spyOn($rootScope, "$broadcast").and.callThrough(); //spy on $rootScope $broadcast event

spyOn($rootScope, "$on").and.callThrough(); //spy on $rootScope $on event

});

it("should broadcast 'item:added' message", function() {

// avoid calling $broadcast implementation

$rootScope.$broadcast.and.stub();

appBroadcaster.itemAdded(item);

expect($rootScope.$broadcast).toHaveBeenCalled(); //check if there was a broadcast

expect($rootScope.$broadcast).toHaveBeenCalledWith("item:added", item); //check if the broadcasted message is right

});

it("should trigger 'item:added' listener", function() {

// instantiate controller

$controller("MainCtrl", { $scope: $scope });

// trigger event

appBroadcaster.itemAdded(item); //pass the item variable for broadcasting

expect($rootScope.$on).toHaveBeenCalled();

expect($rootScope.$on).toHaveBeenCalledWith(

"item:added",

jasmine.any(Function)

);

expect($scope.item).toEqual(item); //match the broadcasted message with the received message

});

});

javascript

To check if a function gets called, Jasmine provides spies. Here, we can see an implementation of spies by using spyOn() with .and.callThrough(). callThrough() and stub(); enables you not only to detects if the funciton is called, but also enables you to get the original implementation of the function and check its arguments.

## **Patterns and directory structure**

**From the tests we just wrote, we can see a clear pattern emerging:**

1. Describe the spec with a type and name
2. Load the object's module.
3. Load mock modules if needed.
4. Inject dependencies and spy on methods.
5. Initialize the object:

* 5.1.Services just need to get injected.
* 5.2.Controllers are instantiated using the $controller service.
* 5.3 We need to $compile directives.

1. Write expectations grouped in describe blocks.

This guide featured only two files being tested -- app.js for the code and tests.js for the tests. However, larger and more complex projects may force you to opt for a different file structure:

You'd want to group your code files and test file together, using the .spec.js suffix to differentiate test files.

1

2

3

4

5

6

app/some-controller.js

app/some-controller.spec.js

app/some-directive.js

app/some-directive.spec.js

app/some-service.js

app/some-service.spec.js

## **Conclusion**

This is everything you need to know in order to start testing your application. I have made a [Github repository](https://github.com/Kaizeras/angular-tests) with the code from the guide in case you missed something.