Getting started unit-testing Angular

One of the fundamental reasons for choosing Angular is cited as that it is built with testing in mind. We can build complex web applications using various popular frameworks and libraries with features as tall as the sky and as equally complex. As with anything of increasing complexity and density, this can quickly grow out of hand.

Testing is a good approach to keep code maintainable, understandable, debug-able, and bug-free. A good test suite can help us find problems before they rise up in production and at scale. It can make our software more reliable, more fun, and help us sleep better at night.

There are several schools of thought about how to test and when to test. As this snippet is more about getting us to the tests, we'll only briefly look at the different options. In testing, we can either:

- Write tests first (Test-Driven Development | TDD) where we write a test to match the functionality and API we expect out of our element
- Write tests last where we confirm the functionality works as expected (WBT | Write-behind testing)
- Write tests to black-box test the functionality of the overall system

For the application that we're working on, it's really up to us to determine what style makes sense for our application. In some cases, we operate using TDD style, while in others we operate with WBT.

> We generally follow the following pattern when choosing a testing style: while we're at prototyping phase (or just after), we generally work with WBT testing as we don't always have a solidified API we're working with. Additionally, our team is generally pretty small. Otherwise, when our application starts to grow, we switch over to drive our functionality through testing.

Getting started

Enough chit-chat, let's test!

There are several libraries that we can use right out of the box when testing angular apps. The currently, most popularly supported framework released and sponsored by the Google team is called `karma`.

If we're not using the `yeoman.io` generator, we'll need to install karma for development puposes. Installing `karma` can by done using the `npm` tool, which is a package manager for node and comes built-in:

```
$ npm install --save-dev karma
```

> If we're using the `yeoman` generator with our apps, this is already set up for us.

`Karma` works by launching a browser, loading our app or a derivative of our source files and running tests that we write against our source code. In order to *use* karma, we'll need to tell the framework about our files and all the various requirements.

To kick it off, we'll use the `karma init` command which will generate the initial template that we'll use to build our tests:

```
$ karma init karma.conf.js
```

It will ask us a series of questions and when it's done, it will create a configuration file. Personally, we usually go through and answer yes to as many which are asked (except without RequireJS). We like to fill in the `files:` section manually (see below).

When it's done, it will create a file in the same directory where we ran the generator that looks similar to the following:

```
// Karma configuration
module.exports = function(config) {
  config.set({
    // base path, that will be used to resolve files and exclude
    basePath: '',

    // testing framework to use (jasmine/mocha/qunit/...)
    frameworks: ['jasmine'],

    // list of files / patterns to load in the browser
    files: [
      'app/components/angular/angular.js',
      'app/components/angular-mocks/angular-mocks.js',
      'app/scripts/**/*.js',
      'test/spec/**/*.js'
    ],

    // list of files / patterns to exclude
    exclude: [],

    // web server port
    port: 8080,

    // level of logging
    // possible values: LOG_DISABLE || LOG_ERROR || LOG_WARN || LOG_INFO || LOG_DEBUG
    logLevel: config.LOG_INFO,

    // enable / disable watching file and executing tests whenever any file changes
    autoWatch: false,

    // Start these browsers, currently available:
    // - Chrome
    // - ChromeCanary
    // - Firefox
    // - Opera
    // - Safari (only Mac)
    // - PhantomJS
    // - IE (only Windows)
    browsers: ['Chrome'],


    // Continuous Integration mode
    // if true, it capture browsers, run tests and exit
    singleRun: false
  });
};
```

This `karma.conf.js` file describes a simple `unit` test that karma will load when we start writing tests. We can also tell it to build an e2e, or end-to-end test that is specifically intended for building black-box style testing, but that's for another snippet/article.

> Note that we **need** to have `angular-mocks.js` and our angular code available to reference inside the `karma.conf.js` file.

Now that we have our `karma.conf.js` file generated, we can kick off karma by issuing the following command:

```
$ karma start karma.conf.js
```

If we haven't run it before, it's likely going to **fail** or at least report errors of files not being found. Let's start writing our first test.

In Write-Behind development, we're going to test the following controller:

```
angular.module('myApp', [])
.controller('MainController', function($scope) {
  $scope.name = "Ari";
  $scope.sayHello = function() {
    $scope.greeting = "Hello " + $scope.name;
  }
})
```

First, let's create the actual test file in `test/spec/controllers/main.js`. Since `karma` works well with Jasmine, we'll be using the Jasmine framework as the basis for our tests.

> For information on Jasmine, check out their fantastic documentation at http://jasmine.github.io/2.0/introduction.html (http://jasmine.github.io/2.0/introduction.html).

Inside our freshly created file, let's add the test block:

```
describe('Unit: MainController', function() {
  // Our tests will go here
})
```

Great! Now we need to do a few things to tell our tests what we are testing. We need to tell it what module we are testing. We can do this by using the `beforeEach()` function to load the angular module that contains our `MainController` (in this case, it's just `myApp`):

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));
})
```

Next, we're going to *pretend* that we're angular loading the controller when it needs to be instantiated on the page. We can do this by *manually* instantiating the controller and handing it a `$scope` object. Creating it manually will also allow us to interact with the scope throughout the tests.

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));

  var ctrl, scope;
  // inject the $controller and $rootScope services
  // in the beforeEach block
  beforeEach(inject(function($controller, $rootScope) {
    // Create a new scope that's a child of the $rootScope
    scope = $rootScope.$new();
    // Create the controller
    ctrl = $controller('MainController', {
      $scope: scope
    });
  }));
})
```

Now, we have access to both the controller as well as the scope of the controller.

Writing a test

Now that everything is all set up and ready for testing, let's write one. It's *always* a good idea to test functionality of code that we write. Anytime that variables can be manipulated by the user or we're running any custom actions, it's usually a good idea to write a test for it.

In this case, we won't need to test setting the name to "Ari" as we *know* that will work (it's JavaScript). What we would like to know, however is that the `sayHello()` function works as-expected.

The `sayHello()` method simply preprends the `$scope.name` to a variable called `$scope.greeting`. We can write a test that verifies that `$scope.greeting` is `undefined` before running and then filled with our expected message after we run the function:

```
describe('Unit: MainController', function() {
  // Load the module with MainController
  beforeEach(module('myApp'));

  var ctrl, scope;
  // inject the $controller and $rootScope services
  // in the beforeEach block
  beforeEach(inject(function($controller, $rootScope) {
    // Create a new scope that's a child of the $rootScope
    scope = $rootScope.$new();
    // Create the controller
    ctrl = $controller('MainController', {
      $scope: scope
    });
  }));

  it('should create $scope.greeting when calling sayHello',
    function() {
      expect(scope.greeting).toBeUndefined();
      scope.sayHello();
      expect(scope.greeting).toEqual("Hello Ari");
  });
})
```

We have access to all different parts of the controller through the `scope` of it now. Feel the power of testing? This is only the beginning.

Finally, we are writing a book all about testing with Angular. Interested? Let us know by showing your interest at it's page on leanpub (https://leanpub.com/testingangularjs). Enjoy!

(http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=ra-52b0ce0106c08d39&source=tbx32-300&lng=en-US&s=gmail&url=http%3A%2F%2Fwww.ng-newsletter.com%2Fadvent2013%2F%23!%2Fday%2F19&title=25%20days%20of%20Angular%20%7C%20%20Getting%20started%20unit-testing%20Angular&ate=AT-ra-52b0ce0106c08d3

g+1 4

Enjoy this snippet?

Check out our book that's heading to print this week at ng-book.com (http://ng-book.com)

The 600+ page book is packed full of Angular content written and designed to get you up to speed with Angular from beginner to expert.

Independently published with content just like what you've just read.

Brought to you by the team behind ng-newsletter (http://ng-newsletter.com)

Stay updated, Subscribe to ng-newsletter.

Get your weekly angular fill with handpicked angular content delivered to your inbox.

E-mail address Sub

PRODUCTS

ng-book (https://ng-book.com)
Newsletter Sponsorship (/sponsor)

RESOURCES

CURATED BY

Ari Lerner (http://twitter.com/auser)
Q (http://twitter.com/qookins)
Nate Murray (http://twitter.com/eigenjoy)

FOLLOW US ON