Mocha

What is mocha

- Mocha is a testing framework for Javascript, created to be a simple, extensible, and fast testing suite.

- It's used for unit and integration testing.

- It handles test suites and test cases, and it offers nice reporting features.
- It uses a declarative syntax to nest expectations into cases and suites.

AngularJS testing with karma and mocha

We will go with the following set of tools for writing our unit tests, but most of these have various alternatives and can be used in multiple combinations:

- **Mocha** – testing framework for AngularJS unit tests.
- **Karma** – Karma as test runner.
- **Chai** – assertion library for AngularJS unit tests.
- **PhantomJS** – for window -less test environment.

Setting up karma with Mocha, PhantomJS and chai

- Install NodeJS .
- Install karma using command

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

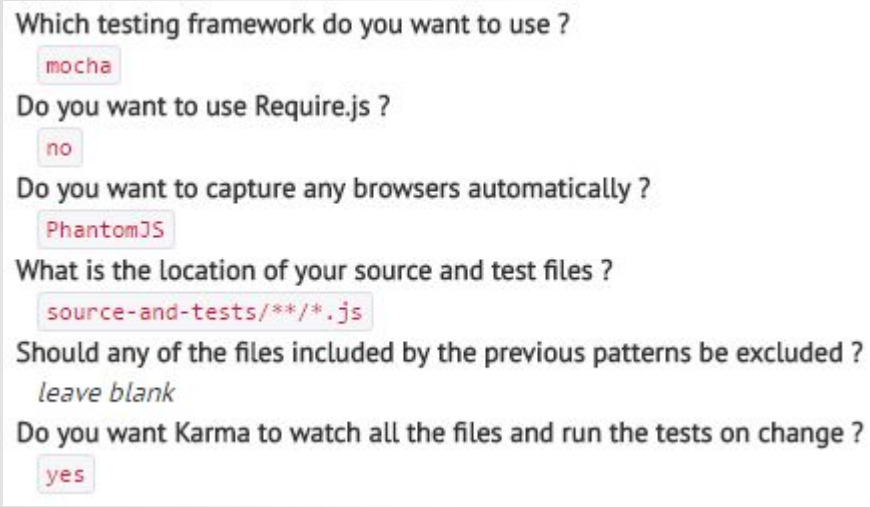
- Now install karma plugins for mocha, Phantomjs and chai.

```
npm install --save-dev karma-mocha
npm install --save-dev karma-phantomjs-launcher
npm install --save-dev karma-chai
```

- Tomake easier to run karma from the command line you can install karma-cli

```
npm install -g karma-cli
```

- Karma needs a configuration file. Generate it using karma init and answer the questions.



- This generates a file called karma-conf.js, which configures Karma for a test run. You can have multiple configuration files pointing to different test suites or browser configurations, which can be run by specifying the name of the configuration file (karma startmy.conf.ks).

- To get Chai included in the test pipeline, we need to edit karma-conf.js and add it to the frameworks setting:

```
frameworks: ['mocha', 'chai'],
```

- Running karma start will execute the default karma-conf.js (or karma-conf.coffee). First we need a test to run. In ./source-and-tests/ I created array-tests.js which just contains the first example from Mocha's documentation.

```
describe('Array', function(){
  describe('#indexOf()', function(){
    it('should return -1 when the value is not present', function(){
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    })
  })
})
```

- Running karma start should find and run this test, then watch for changes to the watched files and repeating.

```
λ karma start
INFO [karma]: Karma v0.12.31 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.8 (Windows 8)]: Connected on socket dOGVHO58uGRab0gX2eLx with id 45140814
PhantomJS 1.9.8 (Windows 8): Executed 1 of 1 SUCCESS (0.004 secs / 0 secs)
```

Adding AngularJS to the mix

- Using Bower to install AngularJS.

```
npm install --save-dev bower
npm install -g bower
```

- This installs Bower to ./node_modules/bower, then installs it globally. Now we need to create a configuration file for Bower:

```
bower init
```

- You can just enter through the configuration, accepting all the defaults. This creates a bower.json file, which will save the dependencies added by Bower. Now use Bower to install AngularJS and angular-mocks:

```
bower install --save angular
bower install --save angular-mocks
```

- This installs AngularJS to `./bower_components/angular` and angular-mocks to `./bower_components/angular-mocks`. The angular-mocks package gives us methods to resolve our application's components and create mocks of AngularJS services.

how to include AngularJS in the test suite

- create a simple controller, and write a test against a property exposed by the controller.
- To include AngularJS and angular-mocks in Karma's test run, edit the files config setting in karma.conf.js. Any future dependencies for the codebase and tests will need to be added here too, unless they are imported in some other way.

```
files: [
        'bower_components/angular/angular.js',
        'bower_components/angular-mocks/angular-mocks.js',
        'source-and-tests/**/*.js'
],
```

The controller to test is very simple at this stage (MyController.js):

```
(function(){
        angular.module('my-module', []);

        angular
                .module('my-module')
                .controller('MyController', [
                        function(){
                                var self = this;

                                self.firstName = '';
                                self.lastName = '';

                                self.getFullName = function(){
                                        return self.firstName + ' ' + self.lastName;
                                };

                                return self;
                        }
        ]);
})();
```

This creates a module called my-module and creates a controller called MyController that exposes firstName, lastName and getFullName(). I want to test the result of getFullName() (MyControllerTests.js):

```
describe('MyController', function(){
        beforeEach(module('my-module'));

        describe('getFullName()', function(){
                it('should handle names correctly', inject(function($controller){
                        var myController = $controller('MyController');

                        myController.firstName = 'George';
                        myController.lastName = 'Harrison';

                        myController.getFullName().should.equal('George Harrison');
                }));
        });
});
```

This does some interesting things.

```
beforeEach(module('my-module'));
```

This loads the `my-module` module before each test in the `MyController` suite.

```
it('should handle names correctly', inject(function($controller){
```

This injects $controller into the test. $controller allows resolving registered controllers.

```
var myController = $controller('MyController');
```

This resolves an instance of the `MyController` controller. The instance is then used as the test subject

$scope injection

The $scope that gets injected into an Angular controller is just a JS object. We'll assign a value and a method to $scope for another test. The controller declaration changes to this:

```
angular
        .module('my-module')
        .controller('MyController', [
                '$scope',
                function($scope){
                        var self = this;

                        // ...

                        $scope.songs = [
                                'Here Comes The Sun'
                        ];

                        $scope.addSong = function(song) {
                                $scope.songs.push(song);
                        };

                        return self;
                }
]);
```

The existing test can just pass in an empty object to the controller resolution:

```
var myController = $controller('MyController', {
        $scope: {}
});
```

Now the new test can inject, use and inspect a mock scope:

```
describe('addSong()', function(){
        it('should add songs', inject(function($controller) {
                var scope = {};
                var myController = $controller('MyController', {
                        $scope: scope
                });

                scope.addSong('While My Guitar Gently Weeps');

                scope.songs.should.contain('While My Guitar Gently Weeps');
        }));
});
```

Injecting and mocking $http

So now I've got a web service that I call to populate something on $scope:

```
angular
        .module('my-module')
        .controller('MyController', [
                '$scope', '$http',
                function($scope, $http){
                        var self = this;

                        // ...

                        $scope.instruments = ['foo'];

                        $http.get('api/get-instruments')
                                .success(function(data) {
                                        $scope.instruments = data;
                                });

                        return self;
                }
]);
```

The `$httpBackend` is an `angular-mocks` service that fakes the `$http` service:

```
describe('get-instruments result', function(){
        it('should be added to scope', inject(function($controller, $httpBackend){
                var scope = {};
                $httpBackend
                        .when('GET', 'api/get-instruments')
                        .respond([
                                'vocals', 'guitar', 'sitar'
                        ]);
                var myController = $controller('MyController', {
                        $scope: scope
                });

                $httpBackend.flush();

                scope.instruments.should.contain('guitar');
        }));
});
```

The `$httpBackend.flush()` simulates the async calls completing, so they can be tested synchronously

Simulating $http errors

If the call to api/get-instruments fails, We want to set a status to ‘ERROR’:

```
$scope.instruments = ['foo'];
$scope.status = '';

$http.get('api/get-instruments')
        .success(function(data) {
                $scope.instruments = data;
        })
        .error(function(e) {
                $scope.status = 'ERROR';
        });
```

To simulate the error, you can just tell the $httpBackend to respond with an error code (500):

```
describe('get-instruments with error', function(){
        it('should have a status with error', inject(function($controller, $httpBackend){
                var scope = {};
                $httpBackend
                        .when('GET', 'api/get-instruments')
                        .respond(500, '');
                var myController = $controller('MyController', {
                        $scope: scope
                });

                $httpBackend.flush();

                scope.status.should.equal('ERROR');
        }));
});
```

Thank you.