**Configuring Karma**

Karma comes equipped with an automated way to create a configuration file. To use the automated way, type the following command:

**$ karma init**

Here is a sample of the options chosen:

**Customizing Karma’s configuration**

The following instructions describe the specific configuration required to get Karma running for the project. Customization includes the test framework (Jasmine), browser

(Chrome) to test with, and files to test. To customize the configuration, open up karma.conf and perform the following steps:

1. Ensure that the enabled framework says jasmine using the following code:

frameworks: ['jasmine'],

2. Configure the test directory. Note that the following definition needs to include the tests required to run along with any potential dependencies. The directory that will hold our tests is /test/unit/:

files: [

'test/unit/\*\*/\*.js'

],

3. Set the test browser to Chrome. It will then be initialized and will run a pop up after every test:

browsers: ['Chrome'],

**Confirming Karma’s installation and configuration**

To confirm Karma’s installation and configuration, perform the following steps:

1. Run the following command to confirm that Karma starts with no errors:

**$ karma start**

2. The output should be something like this:

**$ INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/**

3. In addition, the output should state that no test files were found:

**$ WARN [watcher]: Pattern "test/unit/\*\*/\*.js" does not match any file.**

4. The output should do this along with a failed test message:

**$ Chrome 35.0.1916 (Windows 7): Executed 0 of 0 ERROR (0.016 secs / 0 secs)**

This is expected as no tests have been created yet. Continue to the next step if Karma is started and you will see your Chrome browser with the following output:

**Common installation/configuration issues**

If Jasmine or Chrome Launcher are missing, perform the following steps:

When running the test, an error might occur saying missing Jasmine or Chrome Launcher. If you get this error, type the following command to install the missing dependencies:

**$ npm install karma-jasmine -g**

**$ npm install karma-chrome-launcher -g**

Retry the test and confirm that the errors have been resolved.

The following is what you need to do to provide permissions (sudo/administrator):

In some cases, you might not be able to install npm\_modules globally using the –g command. This is generally due to permission issues on your computer. The resolution is to install Karma directly in your project folder. Use the same command without –g to do this:

**$ npm install karma**

Run Karma using the relative path:

**$ ./node\_modules/karma/bin/karma --version**

Now that Karma is installed and running, it’s time to put it to use.

**Testing with Karma**

In this section, you will create a test to confirm Karma is working as expected. To do this, perform the following steps:

1. Create the test directory. In the Karma configuration, tests were defined in the following directory:

files: [

'test/unit/\*\*/\*.js'

],

Go ahead and create the test/unit directory.

2. Create a new file named firstTest.js in the test/unit directory.

3. Write the first test as follows:

describe('when testing karma', function (){

it('should report a successful test', function (){

expect(true).toBeTruthy();

});

});

4. The preceding test uses Jasmine functions and has the following properties:

describe: This provides a brief string description of the things that will be

tested

it: This provides a brief string of the specific assertion

expect: This provides a way to assert values

toBeTruthy: This is one of several properties on an expectation that can be used to make assertions

This test has no real value other than to confirm the output of a passing test.

5. Check your console window and see that Karma has executed your test. Your command line should say something like this:

**$ INFO [watcher]: Added file "./test/unit/firstTest.js"**

This output means that Karma automatically recognized that a new file was added.

The next output should say something like this:

**$ Chrome 35.0.1916 (Windows 7): Executed 1 of 1 SUCCESS (0.02 secs / 0.015 secs)**

This means your test has passed!

**Confirming the Karma installation**

Now the initial set up and configuration of Karma is complete. Here is a review of the steps:

1. Installed Karma through the npm command
2. Initialized a default configuration through the karma init command
3. Configured Karma with Jasmine and a test/unit test directory
4. Started Karma and confirmed it could be opened with Chrome
5. Added a Jasmine test, firstTest.js, to our test/unit test directory
6. Karma recognized that firstTest.js had been added to the test directory

Karma executed our firstTest.js and reported our output

With a couple of steps, you were able to see Karma running and executing tests automatically. From a TDD perspective, you can focus on moving tests from failing to passing without much effort. No need to refresh the browser; just check the command output window. Keep Karma running and all your tests and files will automatically be added and run.

**Using Karma with AngularJS**

**Getting AngularJS**

An easy method for installing AngularJS into projects is to use Bower. Feel free to install AngularJS into your project in any way you prefer. Following is a brief description on how to install and use Bower.

**Bower**

Bower is a package manager for JavaScript components. Bower allows client-side JavaScript components to be versioned and automatically downloaded into your projects.

This allows you to upgrade third-party tools and components and provide an easy, standard way to use tools such as AngularJS, Bootstrap, and many more.

**Bower installation**

Bower is an npm module, just like Karma. Ensure you have Node.js installed before you try to install Bower using the following steps:

1. Ensure you have Bower installed using this code:

**$ npm install bower -g**

2. Initialize the bower.json configuration in the root of the project:

**$ bower init**

**//This will create a bower.json file which contains the dependent packages**

**//Answer default to all the questions.**

The output should be something like what is shown in the following screenshot:

That is it. Now Bower is installed and ready to download JavaScript packages into your project.

**Installing AngularJS**

Use the following command to install AngularJS using Bower:

**$ bower install angular**

Type the previous command in your command prompt for the directory you will be working in. After the installation is complete, look at your directory and confirm that a bower\_componets directory was created. Inside this, there should be a folder for AngularJS:

**Installing Angular mocks**

Angular mocks allows you to test AngularJS components. The official definition, which is found at https://docs.angularjs.org/api/ngMock, is as follows:

*“The ngMock module provides support to inject and mock Angular services into unit tests. In addition, ngMock also extends various core ng services such that they can be inspected and controlled in a synchronous manner within test code.”*

To install Angular mocks, simply use Bower:

**$ bower install angular-mocks**

**Initializing Karma**

A karma.conf file is required to tell Karma how it should run for the application in question. The best way to initialize it is to run the following command in the command prompt:

**$ karma init**

Use the default answers. After karma.conf has been created in the current directory, open up the configuration. The one configuration that needs to change is the definition of the files for Karma to use.

Use the following definition in the files section, which defines the

files required to run the test:

files: [

'bower\_components/angular/angular.js',

'bower\_components/angular-mocks/angular-mocks.js',

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

'spec/\*\*/\*.js'

],

The preceding configuration loads angular.js, JavaScript files in the app directory, and your tests in the spec folder.

Ensure that Karma can run your configuration:

**$ karma start**

The command output should state something like this:

**$ Chrome 35.0.1916 (Windows 7): Executed 0 of 0 ERROR (0.01 secs / 0 secs)**

That is it. Karma is now running for the first AngularJS application.

**Testing a list of items**

The first development item is to provide you with the ability to have a list of items on a

controller. The next couple of steps will walk you through the TDD process of adding the

first feature using the TDD life cycle that is test first, make it run, make it better.

**Test first**

Determining where to start is often the hardest part. The best way is to remember the 3 A’s

(Assemble, Act, and Assert) and start with the base Jasmine template format. The code to do this is as follows:

describe('',function(){

beforeEach(function(){

});

it('',function(){

});

});

describe: This defines the main feature we are testing. The string will explain the feature in readable terms and then the function will follow with the test.

beforeEach: This is the assemble step. The function defined in beforeEach will get executed before every assert. It is best to put the test setup required before each test

in this function.

it: This is the act and assert step. In the it section, you will perform the action being tested, followed by some assertion. The act step doesn’t have to go into the it function. Depending on the test, it might be more suited in the beforeEach function.

**Assemble, Act, and Assert (3 A’s)**

Now that the template is there, we can start filling in the pieces. We will again follow the 3 A’s mantra.

The following are the two parts of the assemble section.

In the first part, we initialize the module using the following code:

...

beforeEach(function(){

module('todo');

});

...

This code will use the Angular mocks JavaScript library to initialize the AngularJS module being tested. We haven’t defined the todo module, but we will do this after we get a failing test.

The second part talks about the scope of TodoController. The TodoController scope will contain the list of items on its scope variable. It is required that the test has access to the scope of TodoController. Angular mocks will be used to get this. Add the following code to beforeEach to get the controller’s scope:

// scope –hold items on the controller

var scope = {};

beforeEach(function(){

//...

//inject – access angular controllerinject(function($controller){

//$controller – initialize controller with test scope

$controller('TodoController',{$scope:scope});

});

//...

});

The following is a brief explanation of each of the code elements:

scope: This variable is used to hold and test the list items on the controller.

inject: The Angular mocks function is used to access AngularJS’s $controller.

This essentially allows you to get access and inject dependencies into AngularJS objects.

$controller: This initializes the scope of TodoController. The test’s scope variable will now contain the controller’s scope.

In the case of “act”, there is no method to act on. The scope object has already been retrieved as part of the assemble step.

In assert, there are two parts again:

The first assertion is to ensure the TodoController scope has a list variable defined with three items. The list variable will be used to hold the list of all the items:

it('should define a list object',function(){

expect(scope.list).toBeDefined();

});

The second, third, and fourth assertions will be used to confirm whether the data in the list is in the correct order, that is, first is test, second is execute, and third is refactor:

//Second test

it('should define a list object', function(){

expect(scope.list[0]).toEqual('test');

});

//Third test

it('should define a list object', function(){

expect(scope.list[1]).toEqual('execute');

});

//Fourth test

it('should define a list object', function(){

expect(scope.list[2]).toEqual('refactor');

});

**Make it run**

The next step in the TDD life cycle is to make the application run and fix the code so that the tests pass. Remember, think about the smallest components that can be added to make the test pass by proceeding with the following steps:

1. Run Karma by typing the following command:

**$ karma start**

2. If you encounter [$injector:moduler] Failed to instantiate module todo due to error, then it can be due to the following:

The preceding error message is saying that the todo module hasn’t been defined.

Since the error message is telling you what is required, this is the perfect place to start. Create a new file in the app directory named todo.

The working directory should now look something like this:

Add the todo module to the beginning of your new file as follows:

angular.module('todo',[]);

Review the console window where Karma is running. You should now see a new error.

3. Error: The [ng:areq] argument TodoController is not a function, got

undefined:

This error message is describing exactly what needs to be done. There is no need to decipher error messages or stack traces. Simply update the todo.js file so it contains an AngularJS controller as follows:

angular.module('todo',[])

.controller('TodoController',[])

In the previous code, we didn’t try and define the logic required; we only added the smallest component to meet the error message. Review the console window for the next error.

4. Error: The expected undefined to be defined as follows:

The new error message is again clear. We can also see that the code has now passed up to the point of our assertion at the following point:

expect(scope.list).toBeDefined();

As there is no list on the scope, you need to add one. Update the app/todo.js

file as follows:

.controller('TodoController',['$scope',function($scope){

$scope.list = [];

}])

Review the console window.

5. You should now see one of the four tests pass! This means you have successfully used TDD and Karma to get your first test to pass. Now you need to fix the other three. The next error is Expected undefined to equal 'test':

The error output again describes exactly what needs to happen. You just need to initialize the array with the elements test, execute, and run. Go to app/todo.js and add the data to the array initialization:

angular.module('todo',[])

.controller('TodoController',['$scope',function($scope){

$scope.list = ['test','execute','refactor'];

}]);

Review the output in the Karma window.

6. The output is in green and states that all the tests have passed.

The result module and controller code from this step is as follows:

//A module for the application

angular.module('todo',[])

//A controller to manage the to-do items.controller('TodoController',

['$scope', function($scope){

//the initialization of items on the controller scope

$scope.list = ['test','execute','refactor'];

}]);

Now that the “make it run” step is complete, you can move on to the next step and make it better.

**Make it better**

Until this point, there was nothing required to directly refactor or that had been identified in the development to-do list. A review of the development to-do list shows that an item can be crossed out:

View a list of to-do list items:

The example list consists of test, execute, and refactor

Add an item to a to-do-list:

The example list after you add the item will consist of test, execute, refactor, and repeat

Remove an item from a to-do-list:

The example list after you add and then remove the item will consist of test, execute, and refactor

Next up is the requirement to add a new item to the list. The TDD rhythm will be followed again: test first, make it run, and make it better.

**Adding a function to the controller**

The next task is to give the controller the ability to add items to the scope list. This will require the addition of a method to the scope. This walk-through will follow the same

TDD steps as done previously.

**Test first**

Instead of creating a new file and duplicating some of the assemble steps, the following test will be inserted under the last it method. The reason is because the same module and controller will be used:

describe('when using a to-do list', function(){

var scope = null;

beforeEach(function(){

//...

});

//...

describe('',function(){

beforeEach(function(){

});

it('',function(){

});

});

});

**Assemble, Act, and Assert (3 A’s)**

Now that the template is there, we can start filling in the gaps using the 3 A’s mantra:

1. **Assemble**: There is no initialization or setup required, as the module and controller scope will be inherited.

2. **Act**: Here, you need to act on the add method with a new item. We place the act function into the before each function. This allows us to repeat the same step if/when more tests are added:

beforeEach(function(){

scope.add('repeat');

});

3. **Assert**: Here, an item should be added to the list, and then you need to confirm that the last item in the array is as expected:

it('should add item to last item in list',function(){

var lastIndexOfList = scope.list.length - 1;

expect(scope.list[lastIndexOfList]).toEqual('repeat');

});

**Make it run**

The next step in the TDD life cycle is to make it run. Remember, think about the smallest components that can be added to make the test pass, as follows:

1. Ensure Karma is running in your console by typing in the following command:

$ karma start

2. The first error will state TypeError: undefined is not a function:

The error refers to the following line of code:

scope.add('repeat');

The error is telling you that the add method hasn’t been defined. The add function will need to be added to the app/todo.js code. The controller has already been defined, so the add function needs to be placed on the controller’s scope:

angular.module('to-do',[])

.controller('TodoController',['$scope',function($scope){

//...

$scope.add = function(){};

}]);

Notice how the add function doesn’t contain any logic. The smallest component has been added to get the test to satisfy the error message.

Review the console window for the next error.

3. Error: Expected 'refactor' to equal 'repeat':

Have a look at the following expectation:

it('should add item to last item in list',function(){

var lastIndexOfList = scope.list.length - 1;

expect(scope.list[lastIndexOfList]).toEqual('repeat');

});

The failed assertion in step 2 is telling us that based on the preceding expectation, the expected result of repeat is not what the last item in the list has. The smallest possible thing that can be added to make this assertion pass is to push repeat to the end of the list in the add function. Here is how to do this:

//...

$scope.add = function(){

$scope.list.push('repeat');

};

//...

Review the console to see what the next output says.

4. All five tests have now passed.

The resulting code added to get the tests to pass is as follows:

//A module for the application

angular.module('todo',[])

//A controller to manage the to-do items

.controller('TodoController',['$scope', function($scope){

//the initialization of items on the controller scope

$scope.list = ['test','execute','refactor'];

$scope.add = function(){

$scope.list.push('repeat');

};

}]);

**Make it better**

The main thing that we need to refactor is that the add function still hasn’t been fully implemented. It contains a hardcoded value, and the minute we send in a different item into the add function, the test will fail.

Keep Karma running so we can keep passing the tests as changes are made. The main issue with the current add method is as follows:

It doesn’t accept any parameter It doesn’t push a parameter onto the list but uses a hardcoded value

The resultant add function should now look as follows:

$scope.add = function(item){

$scope.list.push(item);

};

Confirm that the Karma output still displays success:

**$ Chrome 35.0.1916 (Windows 7): Executed 5 of 5 SUCCESS (0.165 secs / 0.153 secs)**