**What is Jasmine framework?**

Jasmine is an open-source JavaScript framework, capable of testing any kind of JavaScript application. Jasmine follows Behavior Driven Development (BDD) procedure to ensure that each line of JavaScript statement is properly unit tested. By following BDD procedure, Jasmine provides a small syntax to test the smallest unit of the entire application instead of testing it as a whole.

**What is karma test runner?**

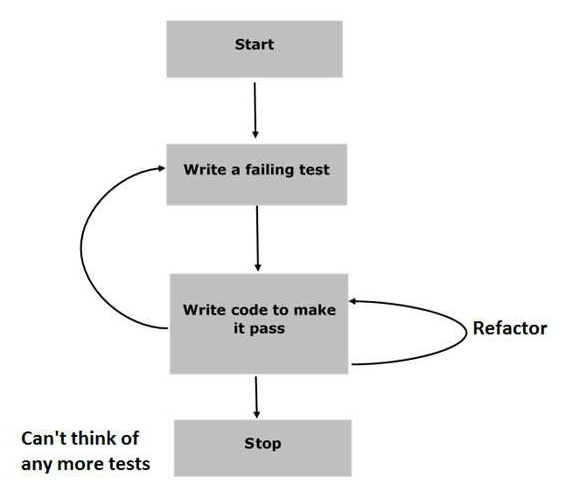
A tool called Karma is a JavaScript test runner created by the AngularJS team. Jasmine is the testing framework that can be used for unit testing for Angular JS post, and Karma provides helpful tools that make it easier to us to call our Jasmine tests whilst we are writing code.

The main goal for Karma is to bring a productive testing environment to developers. The environment being one where they don't have to set up loads of configurations, but rather a place where developers can just write the code and get instant feedback from their tests. Because getting quick feedback is what makes you productive and creative.

**Why Use Jasmine?**

Following are the advantages of using Jasmine over other available JavaScript testing frameworks −

* Jasmine does not depend on any other JavaScript framework.
* Jasmine does not require any DOM.
* All the syntax used in Jasmine framework is clean and obvious.
* Jasmine is heavily influenced by Rspec, JS Spec, and Jspec.
* Jasmine is an open-source framework and easily available in different versions like stand-alone, ruby gem, Node.js, etc.



**Step 1 − Start**

In this phase, we will make our environment ready for Jasmine application.

**Step 2 − Write a failing test**

In this step, we will write our first ever test case. It is obvious that this test is going to fail because there is no such file or function to be tested.

**Step 3 − Write a code to make it pass**

In this phase, we will prepare our JavaScript file or function that needs to be tested. This phase is crucial as we need to make sure that all the test cases we had prepared in the early stage will be successful.

**Step 4 − Refactor**

Refactor is a very important phase of BDD model where we need to prepare as many test cases as we can for that particular application or function.

**Step 5 − Stop**

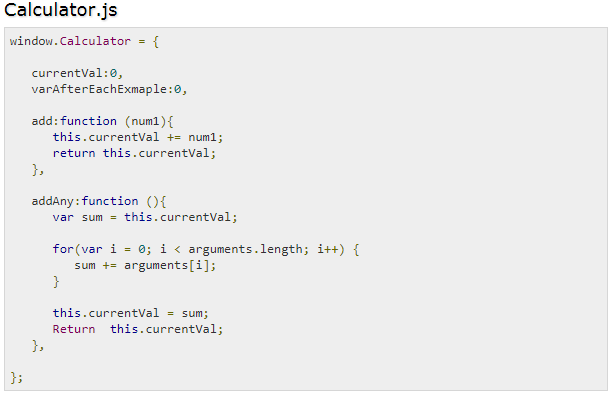
If everything is going well then your application must be ready and up. So we can consider this step as an end of our BDD application.

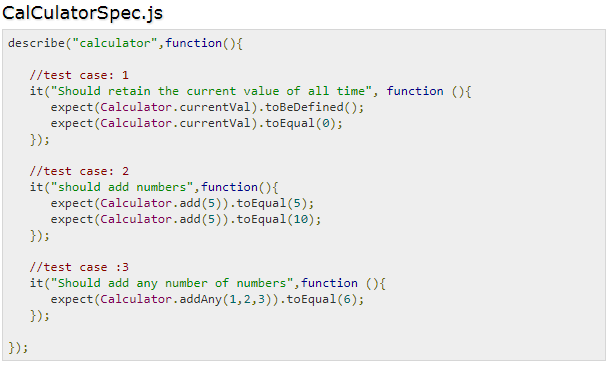
**Suite Block**

Jasmine is a testing framework for JavaScript. Suite is the basic building block of Jasmine framework. The collection of similar type test cases written for a specific file or function is known as one suite. It contains two other blocks, one is “Describe()” and another one is “It()”.

One Suite block can have only two parameters, one “name of that suite” and another “Function declaration” that actually makes a call to our unit functionality that is to be tested.

Example :





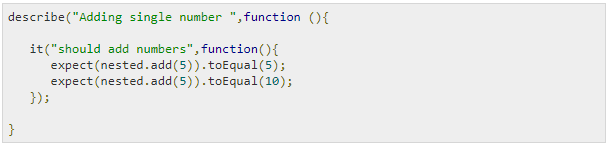
**Nested Suites Block**

Suite block can have many suite blocks inside another suite block.



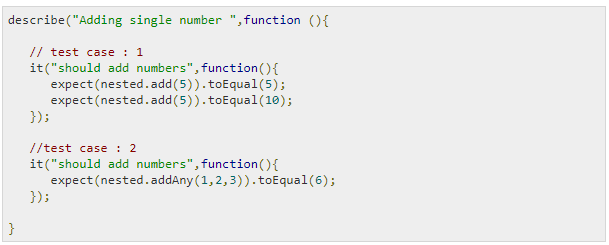
**Describe Block**

Describe block is a part of Suite block. Like Suite block, it contains two parameters, one “the name of the describe block” and another “function declaration”.



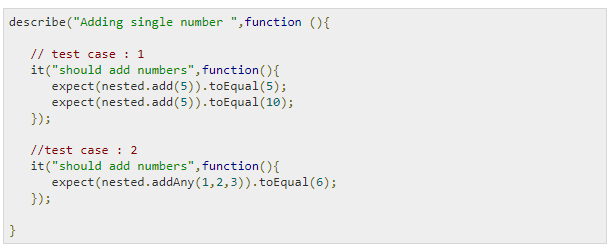
**IT Block**

Like describe block we have been introduced to IT block too. It goes within a describe block. This is the block which actually contains each unit test case.



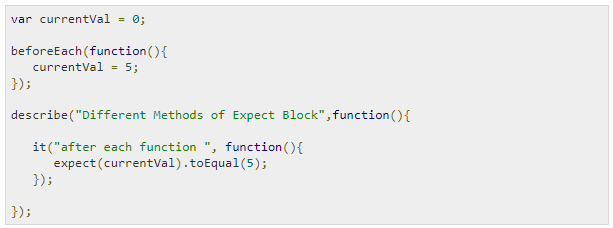
**Expect Block**

Jasmine Expect allows you to write your expectation from the required function or JavaScript file. It comes under IT block. One IT block can have more than one Expect block.



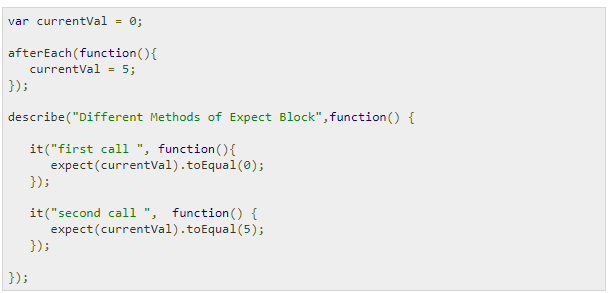
**beforeEach()**

Another notable feature of Jasmine is before and after each function. Using these two functionalities, we can execute some pieces of code before and after execution of each spec. This functionality is very useful for running the common code in the application.



**afterEach()**

Like beforeEach(), afterEach() works exactly the same way. It executes after the execution of the spec block.



**E2E Testing for AngularJs with jasmine and protractor:**

(Reference: <http://www.protractortest.org/#/tutorial>, <http://www.protractortest.org/#/toc> )

**Prerequisites:**

* Need to have Node.js installed.
* Download Protractor package using npm.
* Check version of node (node --version) and also check Protractor version compatibility (<https://github.com/angular/protractor#compatibility>).
* By default, Protractor uses the Jasmine test framework for its testing interface.[Here version 2.4 is being used]

**Note**: This tutorial will set up a test using a local standalone Selenium Server to control browsers.

You will need to have the Java Development Kit (JDK) installed to run the standalone Selenium Server.

Check this by running java -version from the command line.

**Setup**:

Run following commands:







Check: $protractor --version

Open: http://localhost:4444/wd/hub

**Step 0 - write a test:**

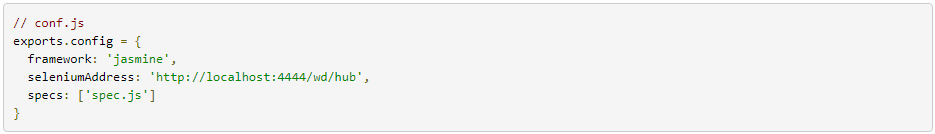
* Open a new command line or terminal window and create a clean folder for testing.
* Protractor needs two files to run, a spec file and a configuration file.
* We’ll use the Super Calculator application at http://juliemr.github.io/protractor-demo/.

*spec.js:*



**browser** is a global created by Protractor which is used for browser-level commands such as navigation with browser.get

*conf.js:*



Now run the test with: $protractor conf.js

You should see a Chrome browser window open up and navigate to the Calculator, then close itself (this should be very fast!).

The test output should be 1 tests, 1 assertion, 0 failures. Congratulations, you've run your first Protractor test!

**Step 1 - interacting with elements**:

Now let's modify the test to interact with elements on the page. Change spec.js to the following:

*spec.js*



* Here *element*, by are global and is used for finding HTML elements on your webpage. It returns an ElementFinder object, which can be used to interact with the element or get information from it.
* *element* takes one parameter, a Locator, which describes how to find the element. The by object creates Locators.

Run the tests with: protractor conf.js

You should see the page enter two numbers and wait for the result to be displayed. Because the result is 3, not 5,

Our test fails. Fix the test and try running it again.

**Step 2 - writing multiple scenarios:**

Let's put these two tests together and clean them up a bit. Change spec.js to the following

*spec.js*

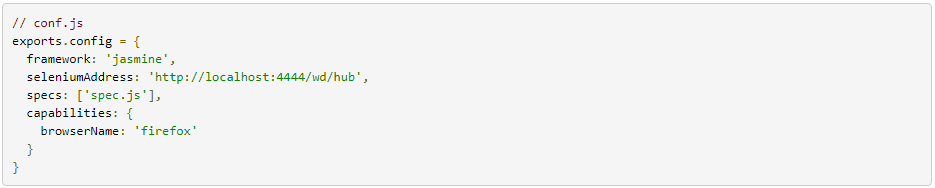
****

**Step 3 - changing the configuration:**

The configuration file lets you change things like which browsers are used and how to connect to the Selenium Server.

Let's change the browser. Change conf.js to the following:

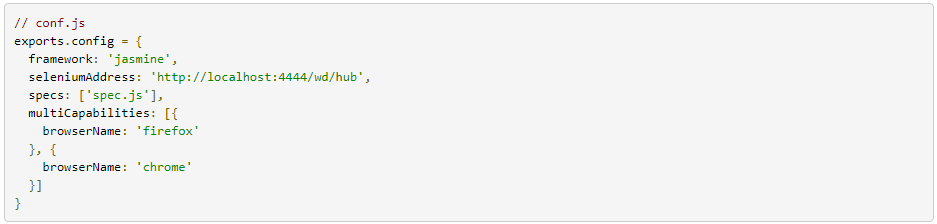
*conf.js*



Try running the tests again. You should see the tests running on Firefox instead of Chrome.

You can also run tests on more than one browser at once. Change conf.js to:

*conf.js*



Try running once again. You should see the tests running on Chrome and Firefox simultaneously, and the results

Reported separately on the command line.

**Step 4 - lists of elements:**

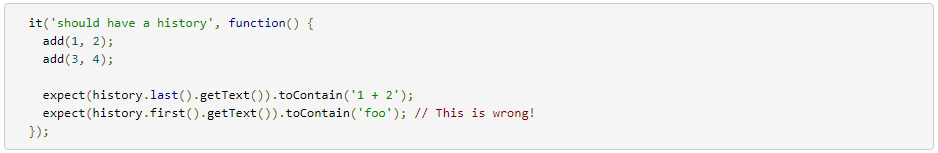
* To deal with list of multiple elements we can use element.all, which returns an ElementArrayFinder.
* In our calculator application, every operation is logged in the history, which is implemented on the site as a table with ng-repeat.

Let's do a couple of operations, then test that they're in the history. Change spec.js to:

*spec.js*



* We've done a couple things here - first, we created a helper function, add. We've added the variable history.
* We use element.all with the by.repeater Locator to get an ElementArrayFinder. In our spec, we assert that the history has the expected length using the count method. Fix the test so that the second expectation passes.
* ElementArrayFinder has many methods in addition to count. Let's use last to get an ElementFinder that matches the last element found by the Locator. Change the test to:



Since the Calculator reports the oldest result at the bottom, the oldest addition (1 + 2) be the last history entry.

We're using the toContain Jasmine matcher to assert that the element text contains "1 + 2". The full element text will also contain the timestamp and the result.

ElementArrayFinder also has methods each, map, filter, and reduce which are analogous to JavaScript Array methods.