**JasmineJS**

Jasmine is one of the popular JavaScript unit testing frameworks which is capable of testing synchronous and asynchronous JavaScript code. It is used in BDD (behavior-driven development) programming which focuses more on the business value than on the technical details. It supports both client and server side testing.

**1. Jasmine Setup Configuration**

First [download jasmine framework](https://github.com/jasmine/jasmine/releases) and extract it inside your project folder. Create a separate folder /jasmine under /js or /javascript folder which may be already present in your application.

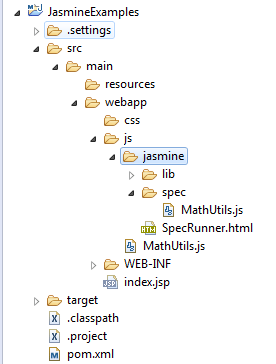
You will get below four folders/files in distribution bundle:

1. /src : contains the JavaScript source files that you want to test
2. /lib : contains the framework files
3. /spec : contains the JavaScript testing files
4. SpecRunner.html : is the test case runner HTML file

You may delete /src folder; and reference the source files from their current location inside SpecRunner.html file. The default file looks like below, and you will need to change the files included from /src and /spec folders.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8">      <title>Jasmine Spec Runner v2.4.1</title>        <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.4.1/jasmine\_favicon.png">      <link rel="stylesheet" href="lib/jasmine-2.4.1/jasmine.css">        <script src="lib/jasmine-2.4.1/jasmine.js"></script>      <script src="lib/jasmine-2.4.1/jasmine-html.js"></script>      <script src="lib/jasmine-2.4.1/boot.js"></script>        <!-- include source files here... -->      <script src="src/Player.js"></script>      <script src="src/Song.js"></script>        <!-- include spec files here... -->      <script src="spec/SpecHelper.js"></script>      <script src="spec/PlayerSpec.js"></script>  </head>    <body></body>  </html> |

The current folder structure is below:

Jasmine Folder Structure

To concentrate on what Jasmine is capable of, creating a simple JS file MathUtils.js with some basic operations and we will unit-test these functions.

|  |
| --- |
| MathUtils = function() {};    MathUtils.prototype.sum = function(number1, number2) {          return number1 + number2;  }    MathUtils.prototype.substract = function(number1, number2) {      return number1 - number2;  }    MathUtils.prototype.multiply = function(number1, number2) {      return number1 \* number2;  }    MathUtils.prototype.divide = function(number1, number2) {      return number1 / number2;  }    MathUtils.prototype.average = function(number1, number2) {      return (number1 + number2) / 2;  }    MathUtils.prototype.factorial = function(number) {      if (number < 0) {          throw new Error("There is no factorial for negative numbers");      } else if (number == 1 || number == 0) {          return 1;      } else {          return number \* this.factorial(number - 1);      }  } |

And after adding file reference in SpecRunner.html, file content will be :

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8">      <title>Jasmine Spec Runner v2.4.1</title>        <link rel="shortcut icon" type="image/png"          href="lib/jasmine-2.4.1/jasmine\_favicon.png">      <link rel="stylesheet" href="lib/jasmine-2.4.1/jasmine.css">        <script src="lib/jasmine-2.4.1/jasmine.js"></script>      <script src="lib/jasmine-2.4.1/jasmine-html.js"></script>      <script src="lib/jasmine-2.4.1/boot.js"></script>        <!-- include source files here... -->      <script src="../MathUtils.js"></script>        <!-- include spec files here... -->      <script src="spec/MathUtils.js"></script>  </head>    <body></body>  </html> |

**2. Jasmine Suite and Specs**

In Jasmine, there are two important terms – **suite** and **spec**.

**2.1. Suite**

A Jasmine suite is a group of test cases that can be used to test a specific behavior of the JavaScript code (a JavaScript object or function). This begins with a call to the Jasmine global function **describe**with two parameters – first parameter represents the title of the test suite and second parameter represents a function that implements the test suite.

|  |
| --- |
| //This is test suite  describe("Test Suite", function() {      //.....  }); |

**2.2. Spec**

A Jasmine spec represents a test case inside the test suite. This begins with a call to the Jasmine global function **it**with two parameters – first parameter represents the title of the spec and second parameter represents a function that implements the test case.

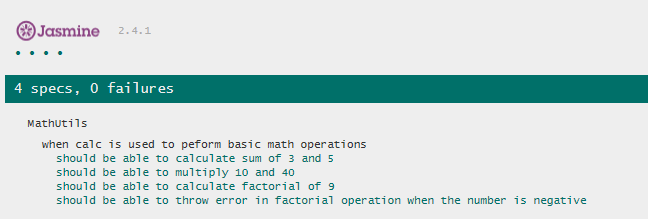
In practice, spec contains one or more expectations. Each expectation represents an assertion that can be either **true or false**. In order to pass the spec, all of the expectations inside the spec have to be true. If one or more expectations inside a spec is false, the spec fails.

|  |
| --- |
| //This is test suite  describe("Test Suite", function() {      it("test spec", function() {          expect( expression ).toEqual(true);      });  }); |

Let’s start writing unit tests for MathUtils.js to better understand suite and specs. We will write these specs in spec/MathUtils.js.

|  |
| --- |
| describe("MathUtils", function() {      var calc;        //This will be called before running each spec      beforeEach(function() {          calc = new MathUtils();      });        describe("when calc is used to peform basic math operations", function(){            //Spec for sum operation          it("should be able to calculate sum of 3 and 5", function() {              expect(calc.sum(3,5)).toEqual(8);          });            //Spec for multiply operation          it("should be able to multiply 10 and 40", function() {              expect(calc.multiply(10, 40)).toEqual(400);          });            //Spec for factorial operation for positive number          it("should be able to calculate factorial of 9", function() {              expect(calc.factorial(9)).toEqual(362880);          });            //Spec for factorial operation for negative number          it("should be able to throw error in factorial operation when the number is negative", function() {              expect(function() {                  calc.factorial(-7)              }).toThrowError(Error);          });        });  }); |

On opening the SpecRunner.html file in browser, specs are run and result is rendered in browser as shown below:

Jasmine Output

**3. Setup and Teardown**

For setup and tear down purpose, Jasmine provides two global functions at suite level i.e. beforeEach() and afterEach().

**3.1. beforeEach()**

The beforeEach function is called once before each spec in the describe() in which it is called.

**3.2. afterEach()**

The afterEach function is called once after each spec.

In practice, spec variables (is any) are defined at the top-level scope — the describe block — and initialization code is moved into a beforeEach function. The afterEach function resets the variable before continuing. This helps the developers in not to repeat setup and finalization code for each spec.

**4. Jasmine Describe Blocks**

In Jasmine, describe function is for grouping related specs. The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec’s full name. This helps in finding specs in a large suite.

Good thing is, you can have nested describe blocks as well. In case of nested describe, before executing a spec, Jasmine walks down executing each beforeEach function in order, then executes the spec, and lastly walks up executing each afterEach function.

Let’s understand it by an example. Replace the content in MathUtilSpecs.js will following code:

|  |
| --- |
| describe("Nested Describe Demo", function() {      beforeEach(function() {          console.log("beforeEach level 1");      });      describe("MyTest level2", function() {          beforeEach(function() {              console.log("beforeEach level 2");          });          describe("MyTest level3", function() {              beforeEach(function() {                  console.log("beforeEach level 3");              });              it("is a simple spec in level3", function() {                  console.log("A simple spec in level 3");                  expect(true).toBe(true);              });              afterEach(function() {                  console.log("afterEach level 3");              });          });          afterEach(function() {              console.log("afterEach level 2");          });      });      afterEach(function() {          console.log("afterEach level 1");      });  }); |

Now execute this file by opening SpecRunner.html in browser. Observe the console output, it is written as:

beforeEach level 1

beforeEach level 2

beforeEach level 3

A simple spec in level 3

afterEach level 3

afterEach level 2

afterEach level 1

**5. Jasmine Matchers**

In first example, we saw the usage of toEqual and toThrow function. They are matchers and use to compare the actual and expected outputs of any jasmine test.

Below are few sample matchers:

|  |  |
| --- | --- |
| **MATCHER** | **PURPOSE** |
| toBe() | passed if the actual value is of the same type and value as that of the expected value. It compares with === operator |
| toEqual() | works for simple literals and variables; should work for objects too |
| toMatch() | to check whether a value matches a string or a regular expression |
| toBeDefined() | to ensure that a property or a value is defined |
| toBeNull() | to ensure that a property or a value is null. |
| toBeTruthy() | to ensure that a property or a value is true |
| toContain() | to check whether a string or array contains a substring or an item. |
| toBeCloseTo() | for precision math comparison |
| toThrow() | for testing if a function throws an exception |
| toThrowError() | for testing a *specific* thrown exception |

The Jasmine not keyword can be used with every matcher’s criteria for inverting the result. e.g.

|  |
| --- |
| expect(actual).not.toBe(expected);  expect(actual).not.toBeDefined(expected); |

**6. Disable Suites and Specs**

Many times, for various reasons, you may want to disable suites – for some time. In this case, you need not to remove the code – rather just add char x in start of describe to make if xdescribe.

These suites and any specs inside them are skipped when run and thus their results will not appear in the results.

|  |
| --- |
| xdescribe("MathUtils", function() {      //code  }); |

In case, you do not want to disable whole suite and rather want to disable only a certain spec test, then put the x before that spec itself and this time only this spec will be skipped.

|  |
| --- |
| describe("MathUtils", function() {      //Spec for sum operation      xit("should be able to calculate the sum of two numbers", function() {          expect(10).toBeSumOf(7, 3);      });  }); |

**7. Working with Jasmine Spies**

Jasmine has test double functions called spies. A spy can stub any function and tracks calls to it and all arguments. A spy only exists in the describe or it block in which it is defined, and will be removed after each spec. To create a spy on any method, use spyOn(object, 'methodName') call.

There are two matchers toHaveBeenCalled and toHaveBeenCalledWith which should be used with spies. toHaveBeenCalled matcher will return true if the spy was called; and toHaveBeenCalledWith matcher will return true if the argument list matches any of the recorded calls to the spy.

|  |
| --- |
| describe("MathUtils", function() {      var calc;        beforeEach(function() {          calc = new MathUtils();          spyOn(calc, 'sum');      });        describe("when calc is used to peform basic math operations", function(){            //Test for sum operation          it("should be able to calculate sum of 3 and 5", function() {              //call any method              calc.sum(3,5);                //verify it got executed              expect(calc.sum).toHaveBeenCalled();              expect(calc.sum).toHaveBeenCalledWith(3,5);          });        });  }); |

Above example is very much most basic in nature, you can use spies to verify the calls for internal methods as well. E.g. If you call method calculateInterest() on any object then you may want to check if getPrincipal(), getROI() and getTime() must have been called inside that object. Spy will help you verify these kind of assumptions.

When there is not a function to spy on, jasmine.createSpy can create a bare spy. This spy acts as any other spy – tracking calls, arguments, etc. But there is no implementation behind it. Spies are JavaScript objects and can be used as such. Mostly, these spies are used as callback functions to other functions where it is needed.

|  |
| --- |
| var callback = jasmine.createSpy('callback');    //Use it for testing  expect(object.callback).toHaveBeenCalled(); |

If you need to define multiple such methods then you can use shortcut method jasmine.createSpyObj. e.g.

|  |
| --- |
| tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);  tape.play();    //Use it for testing   expect(tape.play.calls.any()).toEqual(true); |

Every call to a spy is tracked and exposed on the calls property. Let’s see how we can use these properties to track the spy.

|  |  |
| --- | --- |
| **TRACKING PROPERTY** | **PURPOSE** |
| .calls.any() | returns false if the spy has not been called at all, and then true once at least one call happens. |
| .calls.count() | returns the number of times the spy was called |
| .calls.argsFor(index) | returns the arguments passed to call number index |
| .calls.allArgs() | returns the arguments to all calls |
| .calls.mostRecent() | returns the context (the this) and arguments for the most recent call |
| .calls.first() | returns the context (the this) and arguments for the first call |
| .calls.reset() | clears all tracking for a spy |