JASMINE FUNDAMENTALS

# Jasmine general information

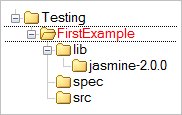
Jasmine is behavior-driven development framework for testing Javascript code. It does not depend on any other Javascript frameworks. It does not require a DOM.

Version: 1.3.1 and 2.0.

At the current, we use Jasmine 2.0 for testing.

# First example with Jasmine

We organize folders/files as:



Folder *lib* contains Jasmine library. Folder *src* contains real source code. Folder *spec* contains test-case files.

Jasmine library has multiple files:

boot.js  
console.js  
jasmine.css  
jasmine.js  
jasmine\_favicon.png  
jasmine-html.js

Sample, we have source code *NumberUtils.js*. It has only one method *toCelsius* as:

function NumberUtils() {  
}

NumberUtils.prototype.toDouble = function(number) {  
 return number \* number;  
};

Now, we setup to test this function.

## Setup test-case files

We store test files in folder *spec*. To test NumberUtils.js file we create new file with name NumberUtilsSpec.js.

Example:

describe("NumberUtils", function() {  
 var util;

beforeEach(function() {

util = new NumberUtils();  
 });

it("should return 0 when getting double value for 0", function() {  
 var number = 0;  
 var result = util.toDouble(number);  
 expect(result).toEqual(0);  
 });

it("should return positive double value when getting double value for negative number", function() {  
 var number = -5;  
 var result = util.toDouble(number);  
 expect(result).toEqual(25);  
 });  
});

In this file, I create two test cases.

## Call test cases

At the root folder, you create file SpecRunner.html to call all test files as:

<!DOCTYPE HTML>  
<html>  
<head>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>Jasmine Spec Runner v2.0.0</title>

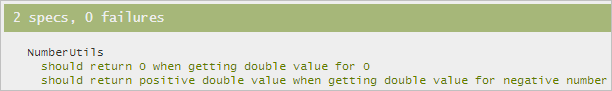
<link rel="shortcut icon" type="image/png" href="lib/jasmine-2.0.0/jasmine\_favicon.png">  
 <link rel="stylesheet" type="text/css" href="lib/jasmine-2.0.0/jasmine.css">  
 <script type="text/javascript" src="lib/jasmine-2.0.0/jasmine.js"></script>  
 <script type="text/javascript" src="lib/jasmine-2.0.0/jasmine-html.js"></script>  
 <script type="text/javascript" src="lib/jasmine-2.0.0/boot.js"></script>

<!-- include source files here... -->  
 <script type="text/javascript" src="src/**NumberUtils.js**"></script>

<!-- include spec files here... -->  
 <script type="text/javascript" src="spec/**NumberUtilsSpec.js**"></script>  
</head>

<body>  
</body>  
</html>

You include all source and test code into this html file. You can open it with any browser and see the result. Example passed cases:



Example failed cases:



With this example, you know how to organize folders/files and the way to call test cases.

# Complex example

We use a complex Javascript files with many function to test. We also use a lot of matchers in Jasmine.

## Creating source file

Sample, we have a Javascript files (*Player*) with one object and many functions.

function Player() {  
}

Player.prototype.play = function(song) {  
 this.currentlyPlayingSong = song;  
 this.isPlaying = true;  
};

Player.prototype.pause = function() {  
 this.isPlaying = false;  
};

Player.prototype.resume = function() {  
 if (this.isPlaying) {  
 throw new Error("song is already playing");  
 }  
 this.isPlaying = true;  
};

Player.prototype.makeFavorite = function() {  
 this.currentlyPlayingSong.persistFavoriteStatus(true);  
};

function Song() {  
}

Song.prototype.persistFavoriteStatus = function(value) {  
 // something complicated  
 throw new Error("not yet implemented");  
};

## Create custom match

Jasmine has a lot of matchers and it supports us to create new custom matcher. We create custom matcher in Javascript file and will include it to use later.

Sample of custom matcher:

beforeEach(function () {  
 jasmine.addMatchers({  
 toBePlaying: function () {  
 return {  
 compare: function (actual, expected) {  
 var player = actual;

return {  
 pass: player.currentlyPlayingSong === expected && player.isPlaying  
 }  
 }  
 };  
 }  
 });  
});

With this sample, we create matcher toBePlaying. We provide condition to pass matcher.

*Note: don’t use Jasmine matchers name to name for your custom matchers. We will have the list of Jasmine matchers name later.*

## Create spec file

To test Player.js, we create new file PlayerSpec.js as:

describe("Player", function() {  
 var player;  
 var song;

beforeEach(function() {  
 player = new Player();  
 song = new Song();  
 });

// Write all test cases here  
}

We use *describe* to group test cases. The content of *beforeEach* is called before each test cases. We need test Player object so we declare player variable. Player uses Song object so we declare song variable.

Writing a **simple case to test play function** as:

it("should be able to play a Song", function() {  
 player.play(song);  
 expect(player.currentlyPlayingSong).toEqual(song);

// demonstrates use of custom matcher  
 expect(player).toBePlaying(song);  
});

Structure *it* is used to write a case. “*should be able to play a Song*” is case name and it should be started with “*should*”. We call all needed functions in the content of case. Function *expect* uses to compare real result with expected result. We can call built-in matchers (*toEqual*) and custom matchers (*toBePlaying*).

Writing a **group to test cases when song has been paused** as:

describe("when song has been paused", function() {  
 beforeEach(function() {  
 player.play(song);  
 player.pause();  
 });

it("should indicate that the song is currently paused", function() {  
 expect(player.isPlaying).toBeFalsy();  
 // demonstrates use of 'not' with a custom matcher  
 expect(player).not.toBePlaying(song);  
 });

it("should be possible to resume", function() {  
 player.resume();  
 expect(player.isPlaying).toBeTruthy();  
 expect(player.currentlyPlayingSong).toEqual(song);  
 });  
});

“*when song has been paused*” is the name of group with two cases. For each case, it runs the *beforeEach* function of parent describe, after that it runs the *beforeEach* function of group.

With this code, before each test case, it will run:

var player;  
var song;  
player = new Player();  
song = new Song();  
player.play(song);  
player.pause();

You can use multiple expect in one test case. This test case uses multiple built-in matchers as: toBeFalsy, toBeTruthy, toEqual.

Writing case **to test expected exceptions** as:

describe("#resume", function() {  
 it("should throw an exception if song is already playing", function() {  
 player.play(song);

expect(function() {  
 player.resume();  
 }).toThrowError("song is already playing");  
 });  
});

Remark: *describe* can contains multiple *describe* items and multiple *it* items.

# Jasmine built-in matchers

Jasmine has many built-in matchers as:

|  |  |  |
| --- | --- | --- |
| **#** | **Matchers sample** | **Description** |
| 1. | expect(result).toEqual(25); | Compare result === 25 |
| 2. | expect(result).toBe(25); | Compare result == 25 |
| 3. | expect(n).toBeDefined(); | Whether a variable is defined |
| 4. | var n;  expect(n).toBeUndefined(); | Whether a variable is not defined |
| 5. | expect(null).toBeNull(); | Whether a variable is null |
| 6. | expect(t).toBeTruthy(); |  |
| 7. | expect(f).toBeFalsy(); |  |
| 8. | expect(a).toContain('leo'); |  |
| 9. | expect(f).toBeLessThan(t); |  |
| 10. | expect(t).toBeGreaterThan(f); |  |
| 11. | expect(function(){fn();}).toThrowError(); |  |
| 12. | toHaveBeenCalled |  |
| 13. | toHaveBeenCalledWith |  |
| 14. | expect.not.to… | Not of … |
| 15. |  |  |
| 16. |  |  |