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, 2.0.0, 2.0.1 and 2.0.2 …

At the current, we use Jasmine 2.0.2 for testing.

References:

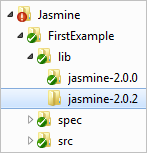
<https://github.com/pivotal/jasmine>

<http://jasmine.github.io/2.0/introduction.html>

<https://github.com/twilson63/grind/tree/master/public/jasmine>

# 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 NumberUtils**Spec**.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 v2.0.2</title>

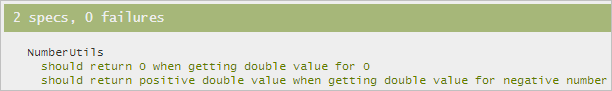
<link rel="shortcut icon" type="image/png" href="lib/jasmine-2.0.2/jasmine\_favicon.png">  
 <link rel="stylesheet" type="text/css" href="lib/jasmine-2.0.2/jasmine.css">  
 <script type="text/javascript" src="lib/jasmine-2.0.2/jasmine.js"></script>  
 <script type="text/javascript" src="lib/jasmine-2.0.2/jasmine-html.js"></script>  
 <script type="text/javascript" src="lib/jasmine-2.0.2/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 functions to test. We also use a lot of matchers in Jasmine.

## Creating source file

Sample, we have two Javascript files (Player.js and Song.js) with one object and many functions.

The content of Player.js file:

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);  
};

The content of Song.js file:

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 (SpecHelper.js):

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 the matcher. This beforeEach function just runs one time when SpecRunner starts.

*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 with name 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 case in group runs. 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 functions 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 (strict equal). Can use for object. With object, it compares number of elements, key and value. |
| 2. | expect(result).toBe(25); | Compare result === 25 (strict equal). Cannot use for object. |
| 3. | var pi = 3.1415926;  expect(pi).toBeCloseTo(3.141, 2); | When rounding numbers with 2 precision then pi 🡺 3.14 and 3.141 🡺 3.14. So, it returns true. |
| 4. | var first, second = 2;  expect(first).toBeDefined();  expect(second).toBeDefined();  expect(third).toBeDefined(); | Whether a variable is defined   * Checking first: false * Checking second: true * Checking third: ReferenceError |
| 5. | var first, second = 2;  expect(first).toBeUndefined();  expect(second).toBeUndefined();  expect(third).toBeUndfined(); | Whether a variable is not defined   * Checking first: true * Checking second: false * Checking third: ReferenceError |
| 6. | var temp = null;  expect(null).toBeNull();  expect(temp).toBeNull(); | Whether an expression is null. null and temp are null. |
| 7. | expect(NaN).toBeNaN();  expect(parseInt("abc")).toBeNaN(); | Whether an expression is NaN. |
| 8. | var temps = ["aa", "bb", "cc"];  expect(temps).toContain("bb");  var temp = "abc";  expect(temp).toContain("b"); | Whether an array/string contains an element/sub string. |
| 9. | var message = "foo bar baz";  expect(message).toMatch(/bar/);  expect(message).toMatch("foo"); | Whether a string matchs to an expression. |
| 10. | var one = 2, foo = "foo";  expect(one).toBeTruthy();  expect(foo).toBeTruthy();  expect(5 > 3).toBeTruthy(); | Whether a variable/expression is true. |
| 11. | var unsign, zero = 0;  expect(unsign).toBeFalsy();  expect(zero).toBeFalsy();  expect(5 < 3).toBeFalsy(); | Whether a variable/expression is false. |
| 12. | var pi = 3.1415926, e = 2.78;  expect(e).toBeLessThan(pi); | Whether a number (e) is less than another number (pi). |
| 13. | var pi = 3.1415926, e = 2.78;  expect(pi).toBeGreaterThan(e); | Whether a number (pi) is greater than another number (e). |
| 14. | function checkSqrt() {throw 'Invalid number for sqrt function';}  expect(checkSqrt).toThrow('Invalid number for sqrt function');  expect(function(){checkSqrt();}).toThrow('Invalid number for sqrt function'); | Whether function throws something. |
| 15. | function checkSqrt() {throw new ReferenceError("ABC");}  expect(checkSqrt).toThrowError("ABC");  expect(function(){checkSqrt();}).toThrowError(ReferenceError); | Whether function throws error.  <http://www.ecofic.com/about/blog/testing-for-exceptions-with-jasmine> |
| 16. | toHaveBeenCalled | Whether function has been called. Using it with spy. |
| 17. | toHaveBeenCalledWith | Whether function has been called with parameters. Using it with spy. |
| 18. | expect.not.to… | Not of … |

# Jasmine other keywords and features

Other keywords and features in Jasmine:

|  |  |
| --- | --- |
| **Keyword OR feature** | **Description and example** |
| describe | describe keyword begins a test suite in Jasmine.  Structure:  describe(name, function);  Sample:  describe("The 'toEqual' matcher", function() { … }); |
| it | it keyword begins a spec in Jasmine.  Structure:  it(name, function);  Sample:  it("should work for objects", function() { … }); |
| beforeEach  afterEach | Helping Don’t Repeated YourSelf. beforeEach uses for setup code and afterEach uses for teardown code.  The beforeEach function is called once before each spec in the describe is run.  The afterEach function is called once after each spec.  Sample:  beforeEach(function() { … });  afterEach(function() { … }); |
| xit | Pending specs do not run, but their names will show up in the results as pending.  Any spec declared with xit is marked as pending.  Any spec declared without a function body will also be marked pending in results.  And if you call the function pending anywhere in the spec body, no matter the expectations, the spec will be marked pending. |

# Spies

We have two matchers (toHaveBeenCalled, toHaveBeenCalledWith) to check whether a function has been called. But with two these matchers, we don’t need to run the content of function. We can use spy for this case.

Sample, we have simple object as:

var Person = function(firstName, lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;

this.sayHello = function() {  
 alert("Hello " + firstName + " " + lastName);  
 }  
};

We setup a test suite to test this object as:

describe("Person", function() {  
 var person;  
 beforeEach(function() {  
 person = new Person("Harry", "Potter");  
 });

afterEach(function() {  
 person = null;  
 });

// Write specs here

});

We only want to test whether sayHello function is called OR not. If we call person.sayHello() then browser will appear an alert message. Mean that, it performs the content of function. Spy helps us to prevent executing function content.

Sample spec with spy as:

it('should toHaveBeenCalled', function() {  
 spyOn(person, 'sayHello');  
 person.sayHello();  
 expect(person.sayHello).toHaveBeenCalled();  
 expect(person.sayHello).toHaveBeenCalledWith();  
});

We call spyOn for the function of object. After that, we can call the function in many times and don’t see any alert message.

## Spies and.callThrough

If we want spy to execute the content of function then we will use and.callThrough as:

it('should call through function content', function() {  
 spyOn(person, 'sayHello').and.callThrough();  
 person.sayHello();  
 person.sayHello();  
 expect(person.sayHello).toHaveBeenCalled();  
 expect(person.sayHello).toHaveBeenCalledWith();  
});

With these commands we will see two alert messages because we call sayHello function in two times and permit spy to execute function content.

## Spies and.returnValue

By using this structure, all calls to function will return a value. If you use this structure the content of function is not executed. You cannot use and.callThrough and and.returnValue both at the same time.

Sample

it('should return a value', function() {  
 spyOn(person, 'sayHello').and.returnValue(101);  
 person.sayHello();  
 var returnValue = person.sayHello();  
 expect(person.sayHello).toHaveBeenCalled();  
 expect(returnValue).toBe(101);  
});

We call sayHello function in two times but don't see any alert message because the content of function is not executed. returnValue variable calls sayHello function and it receives value 101. After that, we use this value for comparing.

## Spies and.callFake

By chaining the spy with and.callFake, all calls to the spy will execute a declared function.

Sample

it('should call a function', function() {  
 spyOn(person, 'sayHello').and.callFake(function() {  
 alert(123456);  
 return 1001;  
 });

person.sayHello();  
 var returnValue = person.sayHello();  
 expect(person.sayHello).toHaveBeenCalled();  
 expect(returnValue).toBe(1001);  
});

When person.sayHello is called then the function content - that is defined at callFake - will be executed. So, we can see two alert messages. callFake function returns 1001 so the value of returnValue will be 1001.

## Spies and.throwError

By chaining the spy with and.throwError, all calls to the spy will throw the specified value as an error.

Sample

it('should throw an error', function() {  
 spyOn(person, 'sayHello').and.throwError("Error name");  
 expect(function() {person.sayHello()}).toThrowError("Error name");  
});

throwError does not execute function content so you will not see any alert message. It just throws an error so you can use toThrowError matcher to check.

## Spies and.stub

When calling and.stub function with a spy, the original behavior will be returned. Mean that, it resets spy to original behavior.

Sample

it('should reset to original behavior', function() {  
 spyOn(person, 'sayHello').and.callThrough();  
 person.sayHello();  
 person.sayHello.and.stub();  
 person.sayHello();  
 person.sayHello();  
 expect(1).toBeTruthy();  
});

We call person.sayHello in three times. At the first time, it shows alert message because we tell spy to call through function content. After that, we call and.stub to set spy to original behavior. So, at the two last times, there is no any alert message shown.

## Spies other tracking properties

Every call to a spy is tracked and exposed on the calls property.

.calls.any function returns false if the spy has not been called at all, and then true once at least one call happens.

Sample

it('tracks if it was called at all', function() {  
 spyOn(person, 'sayHello');  
 expect(person.sayHello.calls.any()).toEqual(false);  
 person.sayHello();  
 expect(person.sayHello.calls.any()).toEqual(true);  
});

.calls.count function returns the number of times the spy was called.

Sample

it('tracks the number of times it was called', function() {  
 spyOn(person, 'sayHello');  
 person.sayHello();  
 person.sayHello();  
 person.sayHello();  
 expect(person.sayHello.calls.count()).toEqual(3);  
});

.calls.argsFor(index) function returns the arguments passed to call number index.

Sample

it("tracks the arguments of each call", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");  
 person.sayHello();

expect(person.sayHello.calls.argsFor(0)).toEqual([123]);  
 expect(person.sayHello.calls.argsFor(1)).toEqual([456, "baz"]); expect(person.sayHello.calls.argsFor(2)).toEqual([]);  
});

.calls.allArgs function returns the arguments to all calls.

Sample

it("tracks the arguments of all calls", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");

expect(person.sayHello.calls.allArgs()).toEqual([[123], [456, "baz"]]);  
});

.calls.all function returns the context (the this) and arguments passed all calls.

Sample

it("can provide the context and arguments to all calls", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");

expect(person.sayHello.calls.all()).toEqual(  
 [{object: person, args: [123]},   
 {object: person, args: [456, "baz"]}]);  
});

I called person.sayHello function two times and expected result contained two items.

.calls.mostRecent function returns the context (the this) and arguments for the most recent call.

Sample

it("has a shortcut to the most recent call", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");

expect(person.sayHello.calls.mostRecent()).toEqual(  
 {object: person, args: [456, "baz"]});  
});

.calls.first function returns the context (the this) and arguments for the first call.

Sample

it("has a shortcut to the first call", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");

expect(person.sayHello.calls.first()).toEqual(  
 {object: person, args: [123]});  
});

When inspecting the return from all(), mostRecent() and first(), the object property is set to the value of this when the spy was called.

Sample

it("tracks the context", function() {  
 var spy = jasmine.createSpy('spy');  
 var foo = {  
 fn: spy  
 };

var baz = {  
 fn: spy  
 };

foo.fn(123);  
 foo.fn('abc');  
 baz.fn(456);

expect(spy.calls.first().object).toBe(foo);  
 expect(spy.calls.mostRecent().object).toBe(baz);  
});

.calls.reset function clears all tracking for a spy.

Sample

it("can be reset", function() {  
 spyOn(person, 'sayHello');  
 person.sayHello(123);  
 person.sayHello(456, "baz");

expect(person.sayHello.calls.any()).toBe(true);  
 person.sayHello.calls.reset();  
 expect(person.sayHello.calls.any()).toBe(false);  
});

## Spies - createSpy

jasmine.createSpy function 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.

Sample

var sampleSpy;  
beforeEach(function() {  
 sampleSpy = jasmine.createSpy("sample spy");  
});

it("checking the spy was called", function() {  
 sampleSpy("sample Spy");  
 expect(sampleSpy).toHaveBeenCalled();  
 expect(sampleSpy).toHaveBeenCalledWith("sample Spy");  
 sampleSpy.calls.reset();  
});

## Spies - createSpyObj

In order to create a mock with multiple spies, use jasmine.createSpyObj and pass an array of strings. It returns an object that has a property for each string that is a spy.

Sample

var tapeObj;  
beforeEach(function() {  
 tapeObj = jasmine.createSpyObj("tape", ["play", "pause", "stop"]);  
});

it ("checking each method was called", function() {  
 tapeObj.play();  
 tapeObj.pause();  
 tapeObj.stop();  
 expect(tapeObj.play).toHaveBeenCalledWith();  
 expect(tapeObj.play.calls.mostRecent()).toEqual({object: tapeObj, args: []});  
 tapeObj.stop.calls.reset();  
});

# jasmine.any

jasmine.any takes a constructor or “class” name as an expected value. It returns true if the constructor matches the constructor of the actual value.

Sample, match with any value

it("matching any value", function() {  
 expect({}).toEqual(jasmine.any(Object));  
 expect(123).toEqual(jasmine.any(Number));  
 expect('123').toEqual(jasmine.any(String));  
 expect(true).toEqual(jasmine.any(Boolean));  
 expect(new Date()).toEqual(jasmine.any(Date));  
});

Sample, compare with function arguments

describe("when used with a spy", function() {  
 it("is useful for comparing arguments", function() {  
 var foo = jasmine.createSpy('foo baz');  
 foo(12, function() {  
 return true;  
 });

expect(foo).toHaveBeenCalledWith(  
 jasmine.any(Number), jasmine.any(Function));  
 });  
});

# jasmine.objectContaining

Partial matching.

jasmine.objectContaining only cares about certain key/value pairs in the actual object.

Sample with common object

describe("jasmine.objectContaining", function() {  
 var foo;  
 beforeEach(function() {  
 foo = {a: 1, bar: "bar"}  
 });

it("should contain partial", function() {  
 expect(foo).toEqual(jasmine.objectContaining({bar: "bar"}));  
 });  
});

Sample with Spy

describe("jasmine.objectContaining", function() {  
 it("should contain partial with spy", function() {  
 var callback = jasmine.createSpy("callback");  
 callback(123, {a: 1, foo: "foo"});

expect(callback).toHaveBeenCalledWith(  
 jasmine.any(Number),   
 jasmine.objectContaining({foo: "foo"}));  
 });  
});

Con nua

describe("Manually ticking the Jasmine Clock", function() {

var timerCallback;

beforeEach(function() {

timerCallback = jasmine.createSpy("timerCallback");

jasmine.clock().install();

});

afterEach(function() {

jasmine.clock().uninstall();

});

it("causes a timeout to be called synchronously", function() {

setTimeout(function() {

timerCallback();

}, 100);

expect(timerCallback).not.toHaveBeenCalled();

jasmine.clock().tick(101);

expect(timerCallback).toHaveBeenCalled();

});

it("causes an interval to be called synchronously", function() {

setInterval(function() {

timerCallback();

}, 100);

expect(timerCallback).not.toHaveBeenCalled();

jasmine.clock().tick(101);

expect(timerCallback.calls.count()).toEqual(1);

jasmine.clock().tick(50);

expect(timerCallback.calls.count()).toEqual(1);

jasmine.clock().tick(50);

expect(timerCallback.calls.count()).toEqual(2);

});

}); describe("Asynchronous specs", function() {

var value;

beforeEach(function(done) {

setTimeout(function() {

value = 0;

done();

}, 1);

});

it("should support async execution of test preparation and expectations", function(done) {

value++;

expect(value).toBeGreaterThan(0);

done();

});

describe("long asynchronous specs", function() {

var originalTimeout;

beforeEach(function() {

originalTimeout = jasmine.DEFAULT\_TIMEOUT\_INTERVAL;

jasmine.DEFAULT\_TIMEOUT\_INTERVAL = 10000;

});

it("takes a long time", function(done) {

setTimeout(function() {

done();

}, 9000);

});

afterEach(function() {

jasmine.DEFAULT\_TIMEOUT\_INTERVAL = originalTimeout;

});

});

});