Jasmine and Karma

JavaScript Testing

Last updated: 25 August 2017

Contents

Introduction

Jasmine is an open source (unit) test framework for JavaScript. It does therefore have the same purpose as what JUnit does for Java.

In contrast to Mocha, which is another popular open source (unit) test framework for JavaScript, Jasmine does not include its own test runner. A popular and very powerful choice as a test runner for Jasmine is Karma.

Karma is an open source test runner that helps running and reporting on Jasmine tests. Karma can run Jasmine tests from command line and in Continuous Integration environments, for example with build servers like Jenkins or JetBrains TeamCity.

Download Jasmine and testing the installation

1. Download Jasmine from https://github.com/jasmine/jasmine/releases and unpack it. That's it – no further installation is required.
2. To test the installation, run "`SpecRunner.html`" in a web browser like Google Chrome or Mozilla Firefox. It shows the results of a sample test suite:

A full Jasmine example

The following example uses a sample stock market investment tracker application with two domain base classes ("`Investment.js`" and "`Stock.js`"). It contains one suite of tests ("`InvestmentSpec.js`") with a test helper class ("`SpecHelper.js`").

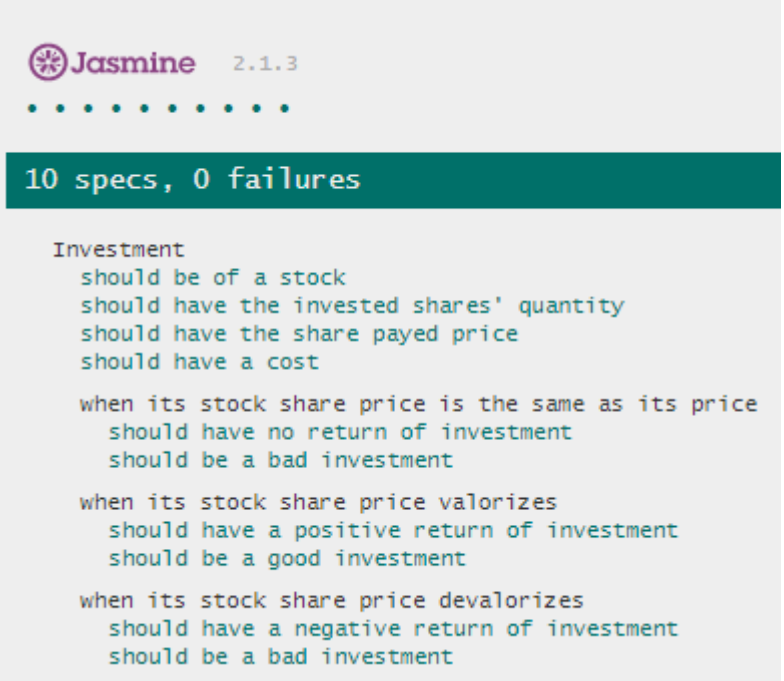
The tests are executed through the HTML page "`SpecRunner.html`".

The two spec files (containing the tests) should be saved in the "`/spec`" folder.

The two source files should be saved in the "`/src`" folder.

The HTML file should be in placed in the root folder.

The result from running the HTML page "`SpecRunner.html`" in web browser will produce this result:



The "Investment.js" class in the "/src" folder

```
function Investment (params) {
  this.stock = params.stock;
  this.shares = params.shares;
  this.sharePrice = params.sharePrice;
  this.cost = this.shares * this.sharePrice;
}

Investment.prototype.roi = function() {
  return (this.stock.sharePrice - this.sharePrice) / this.sharePrice;
};

Investment.prototype.isGood = function() {
  return this.roi() > 0;
```

```
};
```

The “Stock.js” class in the “/src” folder

```
function Stock () {};
```

The “InvestmentSpec.js” class containing the actual tests in the “/spec” folder

```
describe("Investment", function() {
  var stock;
  var investment;

  beforeEach(function(){
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });
  });

  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });

  it("should have the invested shares' quantity", function() {
    expect(investment.shares).toEqual(100);
  });

  it("should have the share payed price", function() {
    expect(investment.sharePrice).toEqual(20);
  });

  it("should have a cost", function() {
    expect(investment.cost).toEqual(2000);
  });

  describe("when its stock share price is the same as its price", function() {
    beforeEach(function() {
      stock.sharePrice = 20;
    });

    it("should have no return of investment", function() {
      expect(investment.roi()).toEqual(0);
    });

    it("should be a bad investment", function() {
      expect(investment).not.toBeAGoodInvestment();
    });
```

```
  });

  describe("when its stock share price valorizes", function() {
    beforeEach(function() {
    stock.sharePrice = 40;
    });

    it("should have a positive return of investment", function() {
      expect(investment.roi()).toEqual(1);
    });

    it("should be a good investment", function() {
      expect(investment).toBeAGoodInvestment();
    });
  });

  describe("when its stock share price devalorizes", function() {
    beforeEach(function() {
      stock.sharePrice = 0;
    });

    it("should have a negative return of investment", function() {
      expect(investment.roi()).toEqual(-1);
    });

    it("should be a bad investment", function() {
      expect(investment).not.toBeAGoodInvestment();
    });
  });

});
```

The "SpecHelper.js" class in the "/spec" folder

The "`toBe`" and "`toEqual`" matchers are the two base built-in matchers in Jasmine. It is best to use the "`toEqual`" operator in most cases and resort to the "`toBe`" matcher only for checking whether two variables reference the same object.

However, it is possible to add custom matchers, and the best place for this in the "`SpecHelper.js`" class. The following example adds a custom matcher called "`toBeAGoodInvestment`". It also generates customised matcher results messages via the "`result.message`" property of the object returned as the result of the matcher.

```
beforeEach(function() {
  jasmine.addMatchers({
    toBeAGoodInvestment: function() {
      return {
        compare: function (actual, expected) {
        // matcher definition
```

```
          var result = {};
          result.pass = actual.isGood();
          if (actual.isGood()) {
            result.message = 'Expected investment to be a bad investment';
          } else {
            result.message = 'Expected investment to be a good investment';
          }
          return result;
        }
      };
    }
  });
});
```

The “SpecRunner.html” page to run the tests in the web browser

```
!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.1.3</title>

  <link rel="shortcut icon" type="image/png" href="lib/jasmine-
2.1.3/jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.1.3/jasmine.css">

  <script src="lib/jasmine-2.1.3/jasmine.js"></script>
  <script src="lib/jasmine-2.1.3/jasmine-html.js"></script>
  <script src="lib/jasmine-2.1.3/boot.js"></script>

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

   <script src="spec/SpecHelper.js"></script>


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

</head>

<body>
</body>
</html>
```

Jasmine built-in matchers

```
- toEqual
  - should pass equal numbers
  - should pass equal strings
  - should pass equal booleans
  - should pass equal objects
  - should pass equal arrays
- toBe
  - should pass equal numbers
  - should pass equal strings
  - should pass equal booleans
  - should pass same objects
  - should pass same arrays
  - should not pass equal objects
  - should not pass equal arrays
- toBeFalsy
  - should pass undefined
  - should pass null
  - should pass NaN
  - should pass the false boolean value
  - should pass the number 0
  - should pass an empty string
- toBeTruthy
  - should pass the true boolean value
  - should pass any number different than 0
  - should pass any non empty string
  - should pass any object (including an array)
- toBeNull
  - should pass null
- toBeUndefined
  - should pass undefined
- toBeNaN
  - should pass NaN
- toBeDefined
  - should pass any value other than undefined
- toContain
  - should pass if a string contains another string
  - should pass if an array contains an element
- toMatch
  - should pass a matching string
- toBeLessThan
  - should pass when the actual is less than expected
- toBeGreaterThan
  - should pass when the actual is greater than expected
- toBeCloseTo
  - should pass when the actual is closer with a given precision
- toThrow
  - should pass when the exception is thrown
```

Spies (mocking)

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. There are two ways to create a spy in Jasmine: "`spyOn()`" can only be used when the method already exists on the object, whereas "`createSpy()`" will return a brand new function. "`CreateSpy()`" will track calls and arguments like a "`spyOn()`", but there is no implementation.

Karma Test Runner

Instead of running tests directly in a web browser, a test runner can and should be used. Karma is a popular open source choice.

Karma is based on Node.js and therefore Node.js needs to be installed first from https://nodejs.org .

After installing Node.js, the node.js package manager NPM can be used to install Karma: https://karma-runner.github.io/1.0/intro/installation.html . Please note that in Microsoft Windows, you will also need to install the command-line interface (karma-cli) as well, if you want to run Karma from the command line.

To run the full Jasmine example with Karma, please create a Karma configuration file with the name "`karma.config.js`":

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      'src/Investment.js',
      'src/Stock.js',
      'spec/SpecHelper.js',
      'spec/InvestmentSpec.js'
    ],
    exclude: [
    ],
    preprocessors: {},
    reporters: ['dots'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: true
  });
};
```

To run it using the defined web browser (Google "Chrome" in this example), use this command from the command line:

```
karma.start
```

The execution will look like this:

```
karma start

22 08 2017 17:30:32.751:INFO [karma]: Karma v1.7.0 server started at http://0.0.0.0:9876/
22 08 2017 17:30:32.755:INFO [launcher]: Launching browser Chrome with unlimited concurrency
22 08 2017 17:30:32.763:INFO [launcher]: Starting browser Chrome
22 08 2017 17:30:34.324:INFO [Chrome 60.0.3112 (Windows 10 0.0.0)]: Connected on socket b-
gTKN7zwNk6vyzlAAAA with id 77138432
..........
Chrome 60.0.3112 (Windows 10 0.0.0): Executed 10 of 10 SUCCESS (0.019 secs / 0.004 secs)
```

Note: The “dots” reporter prints a dot for each (of the 10) test executed.

Karma provides a wide range of plugins (https://www.npmjs.com/browse/keyword/karma-plugin) and integrations with (Continuous Integration) build servers such as Jenkins CI (https://karma-runner.github.io/1.0/plus/jenkins.html) or JetBrains TeamCity (https://karma-runner.github.io/1.0/plus/teamcity.html).