Jasmine: a BDD framework to develop and test Javascript Applications

Student: Lorenzo Valgimigli

April 29, 2019

Contents

Introduction to Jasmine

Official definition

Jasmine is a behavior-driven development framework for testing JavaScript code.

Jasmine is an open-source JavaScript framework, capable of testing any kind of JavaScript application.

Main Features

Principal features:

- BDD : Behavior-Driven Development framework

Main Features

Principal features:

- BDD : Behavior-Driven Development framework
- TDD : Suitable for Test-Driven Development

Main Features

Principal features:

- BDD : Behavior-Driven Development framework
- TDD : Suitable for Test-Driven Development
- Easy Syntax : Easy to learn and master

Main Features

Principal features:

- BDD : Behavior-Driven Development framework
- TDD : Suitable for Test-Driven Development
- Easy Syntax : Easy to learn and master
- No DOM Required : It makes tests light and fast

Main Features

Principal features:

- BDD : Behavior-Driven Development framework
- TDD : Suitable for Test-Driven Development
- Easy Syntax : Easy to learn and master
- No DOM Required : It makes tests light and fast
- Open Source : It comes with different versions

Usage

First of all you need to set up Jasmine.

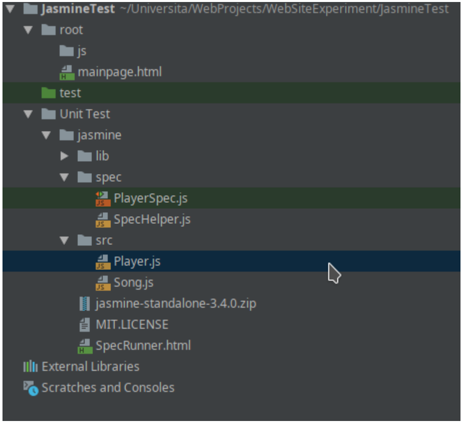
Download

You have to download the latest version from:
`https://github.com/jasmine/jasmine/releases`

Import in your project

Create a directory for Jasmine and copy the content of the .zip file you downloaded in the previous step in this new directory.

Usage - Example



```
JasmineTest ~/Universita/WebProjects/WebSiteExperiment/JasmineTest
    root
        js
        mainpage.html
    test
    Unit Test
        jasmine
            lib
            spec
                PlayerSpec.js
                SpecHelper.js
            src
                Player.js
                Song.js
            jasmine-standalone-3.4.0.zip
            MIT.LICENSE
            SpecRunner.html
External Libraries
Scratches and Consoles
```

Figure: WebStorm Project with Jasmine

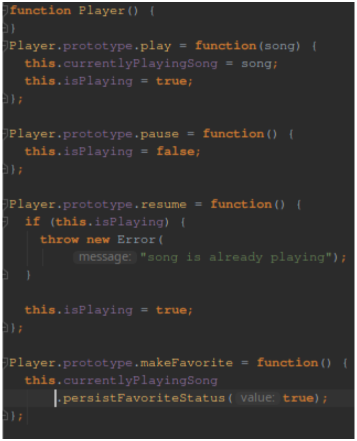
Example

Looking inside the directory we can see three sub-directories:

- `lib` : Where all Jasmine functions are stored
- `spec` : It contains some tests created as example
- `src` : It contains the source code for tests

Example: Player.js

Inside `src` we can find `Player.js`

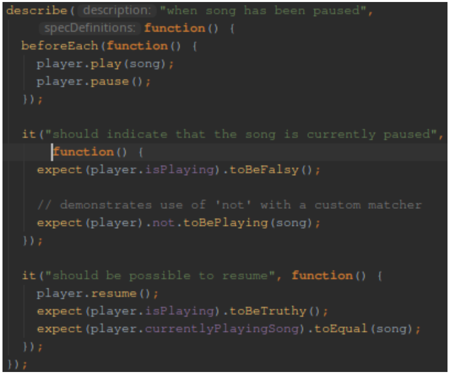


```
function Player() {
}
Player.prototype.play = function(song) {
  this.currentlyPlayingSong = song;
  this.isPlaying = true;
};

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

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

  this.isPlaying = true;
};

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

Figure: Player.js

Example: Tests

Inside `spec` we can find `PlayeSpec.js` where a list of tests are stored like the one in the image

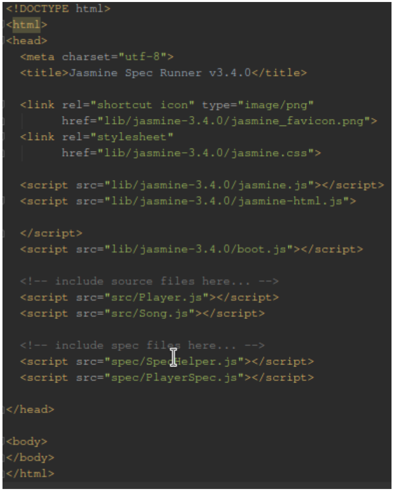


```
describe( description: "when song has been paused",
    specDefinitions: function() {
  beforeEach(function() {
    player.play(song);
    player.pause();
  });

  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);
  });
});
```

Figure: Jasmine test

Example: HTML

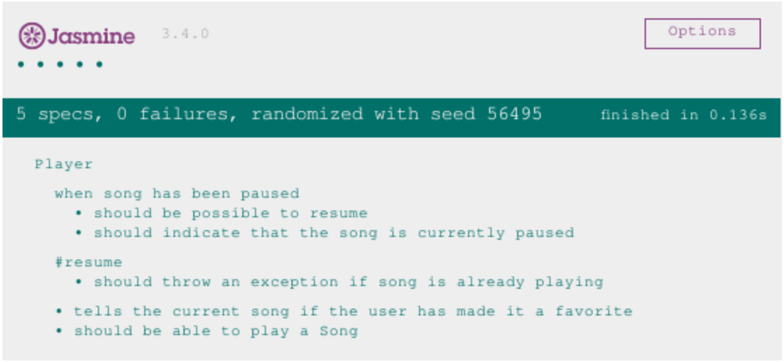
Inside `jasmine` we can find `SpecRunner.html`. It is an HTML file that takes care to combine source and test.



```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v3.4.0</title>

  <link rel="shortcut icon" type="image/png"
        href="lib/jasmine-3.4.0/jasmine_favicon.png">
  <link rel="stylesheet"
        href="lib/jasmine-3.4.0/jasmine.css">

  <script src="lib/jasmine-3.4.0/jasmine.js"></script>
  <script src="lib/jasmine-3.4.0/jasmine-html.js">

  </script>
  <script src="lib/jasmine-3.4.0/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>
```

Figure: Jasmine HTML

Example: Running tests



```
• • • • •

5 specs, 0 failures, randomized with seed 56495        finished in 0.136s

  Player

    when song has been paused
      • should be possible to resume
      • should indicate that the song is currently paused

    #resume
      • should throw an exception if song is already playing

    • tells the current song if the user has made it a favorite
    • should be able to play a Song
```

Figure: Output from Jasmine tests

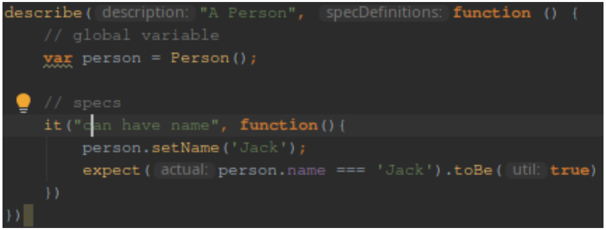
Simple Jasmine Test

describe

The block `describe` collects all tests about a single concept.

Spec

The block `spec` is piece of code that tests a single aspect of the behaviour of the described concept.



```
describe( description: "A Person", specDefinitions: function () {
    // global variable
    var person = Person();

    // specs
    it("can have name", function(){
        person.setName('Jack');
        expect( actual: person.name === 'Jack').toBe( util: true)
    })
})
```

Figure: Template of a test

Basic Elements (1)

Expectation

Inside a spec you can express an expectation using `expect` function:
`expect(true).toBe(true);`

Matcher

To check a result or a value you can use `Matcher`: `toBe()` or `not.ToBe()`

Basic Elements (2): DRY Approach

Before and After

Jasmine give you functions to collect code that you wish to execute before or after each test or all tests:

- beforeEach()
- beforeAll()
- afterEach()
- afterAll()

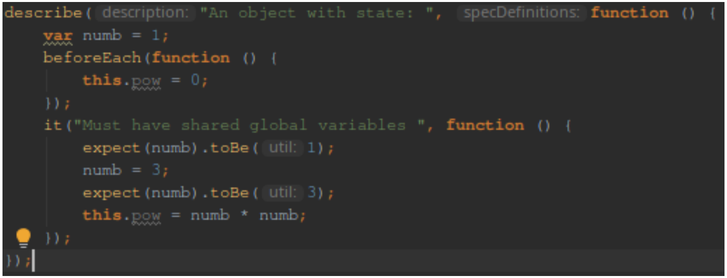
Basic Elements (3): Sharing The State

Global Variables

If you declare variables in the `describe` function you can access to them from each `specs` inside the `describe`.

this

Using `this` you can share variables between `specs` and `after` and `before` sentences

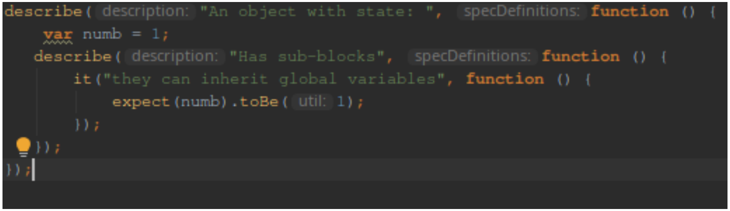


```
describe( description: "An object with state: ",  specDefinitions: function () {
    var numb = 1;
    beforeEach(function () {
        this.pow = 0;
    });
    it("Must have shared global variables ", function () {
        expect(numb).toBe( util: 1);
        numb = 3;
        expect(numb).toBe( util: 3);
        this.pow = numb * numb;
    });
});
```

Basic Elements (4): Nesting

Nesting describe sentences

You can nest some `describe` blocks. This can make tests more clear. The nested `describe` inherits global variables from outer block.



```
describe( description: "An object with state: ", specDefinitions: function () {
    var numb = 1;
   describe( description: "Has sub-blocks", specDefinitions: function () {
       it("they can inherit global variables", function () {
           expect(numb).toBe( util: 1);
       });
   });
});
```

Figure: Nexting example

Basic Elements (4): Spies

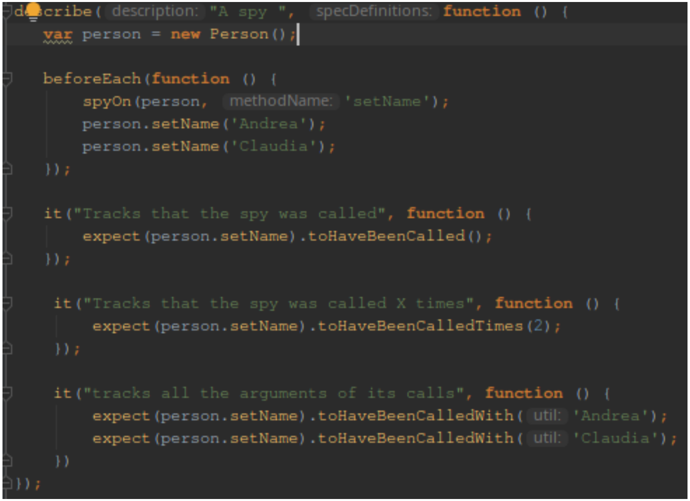
Spy

A Spy is an elements that stubs a function and tracks all calls to it and all arguments passed.

Syntax

Spy must be declared in `spec` blocks or in a before section. To create it you must write: `spyOn(object, 'method-to-track')`

Basic Elements (4): Spies



```
describe( description: "A spy ",  specDefinitions: function () {
    var person = new Person();

    beforeEach(function () {
        spyOn(person,  methodName: 'setName');
        person.setName('Andrea');
        person.setName('Claudia');
    });

    it("Tracks that the spy was called", function () {
        expect(person.setName).toHaveBeenCalled();
    });

     it("Tracks that the spy was called X times", function () {
         expect(person.setName).toHaveBeenCalledTimes(2);
     });

     it("tracks all the arguments of its calls", function () {
         expect(person.setName).toHaveBeenCalledWith( util: 'Andrea');
         expect(person.setName).toHaveBeenCalledWith( util: 'Claudia');
     })
});
```

Figure: Spy in Jasmine

Advanced Features

- **Finesse Testing**: Advanced Matchers concatenation for finesse testing.
- **Custom Spy**: You can create a spy on your own.
- **Jasmine Clock**: It's a functionality available for testing time dependent code.
- **Asynchronous Support**: Jasmine has support to test asynchronous operations:
    - **CallBacks**
    - **Promises**
    - **Async/wait**

Thanks

Thanks you all for the attention