Jasmine

Victor Herrero Cazurro

Contenidos

1. Introduccion

Framework que permite describir pruebas sobre cosigo JS.

La definicion se basa en **BDD**, siguiendo la estructura de **given-when-then**, en este caso, las palabras a emplear son **describe-it-expected**

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(false).toBe(true);
  });
});
```

La idea del framework, es acompañar el test de documentacion que hagan mas legible la prueba.

2. Instalacion

Tres formas de trabajar con **Jasmine**

- Node.js
- Ruby
- Python

2.1. Node.js

Se recomienda inicializar un proyecto **Node.js** lanzando el comando

```
> npm init
```

Obteniendo así el fichero **package.json**

Se ha de lazar el siguiente comando para crear un proyecto **Jasmine**

```
> npm install jasmine
```

Y se ha de lanzar el siguiente comando para inicializar un nuevo proyecto de tipo jasmine

```
> node node_modules/jasmine/bin/jasmine init
```

Si se desea se pueden obtener unos ejemplo con el comando

```
> node node_modules/jasmine/bin/jasmine examples
```

Para ejecutar las pruebas habra que lanzar el comando

```
> node node_modules/jasmine/bin/jasmine spec/appSpec.js

> node node_modules/jasmine/bin/jasmine spec/**/*Spec.js
```

Al estar en **Node.js** se pueden emplear los modulos de **Node.js** para referenciar a las librerias a probar **SUT**

Crando un modulo con

```
function Song() {
}

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

module.exports = Song;
```

Y referenciandolo con

```
var Song = require('../../lib/jasmine_examples/Song');
var song = new Song();
```

Se puede incorporar a los scripts de **npm**, añadiendo al fichero **package.json** lo siguiente

```
"scripts": {
    "test": "jasmine"
  }
```

Y luego ejecutarlos con el comando

```
> npm test
```

3. Suite

Se defienen con la funcion **describe(enunciado,callack)**.

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Sirven para agrupar varios **spec**, normalmente es parte de una frase comun a todos los **spec**.

Pueden contener otros bloques **describe**.

Permiten inicializar variables globales del **Suite** como la carga de modulos de **node.js**

```
describe("Player", function() {
  var Player = require('../../lib/jasmine_examples/Player');
});
```

Se pueden deshabilitar **Suite** con **xdescribe**

4. Spec

Se defienen con la funcion **it(enunciado,callack)**.

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Sirven para definir una prueba, definen un texto que completa el texto del describe

donde se incluyen y son los encargados de ejecutar la pueba sobre el **SUT**.

Dentro contienen las validaciones **expect** necesarias.

```
describe("Player", function() {
  it("should be able to play a Song", function() {
    player.play(song);
  });
});
```

Se pueden marcar **Pruebas** como pendientes con **xit**

5. Assert

Se defienen con la funcion **expect()**.

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Permiten realizar las validaciones que garantizan que el codigo probado funciona como debe.

Su construccion se basa en **Matchers** que permiten aplicar una expresion **booleana** de comparacion entre en elemento actual y el esperado.

```
expect(true).toBe(true);
expect(false).not.toBe(true);
```

Se dispoe de una funcion **fail()** que si se ejecuta hace fallar la prueba

```
describe("A spec using the fail function", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };

  it("should not call the callBack", function() {
    foo(false, function() {
      fail("Callback has been called");
    });
  });
});
```

6. Preparacion del entorno

Se proporcionan funciones que permiten preparar el entorno de ejecucion de la prueba, a nivel del **suite** con las funciones:

- **beforeAll**: Se invoca una sola vez antes de procesar el **describe**

```
beforeAll(function() {
    foo = 1;
});
```

- **afterAll**: Se invoca una sola vez despues de procesar el **describe**

```
afterAll(function() {
    foo = 0;
});
```

o del **spec** con

- **beforeEach**: Se invoca antes de cada **it**

```
beforeEach(function() {
    foo += 1;
});
```

- **afterEach**: Se invoca despues de cada **it**

```
afterEach(function() {
    foo = 0;
});
```

Lo mas abitual es emplear los **before** para la inicializacion de variables empleadas en las pruebas

7. Reportes

Se pueden configurar el volcado de los resultados de los test a distintos formatos

- AppVeyor - POSTs results to AppVeyor when running inside an AppVeyor environment.
- JUnitXmlReporter - Report test results to a file in JUnit XML Report format.
- NUnitXmlReporter - Report test results to a file in NUnit XML Report format.
- TapReporter - Test Anything Protocol, report tests results to console.
- TeamCityReporter - Basic reporter that outputs spec results to for the Teamcity build system.
- TerminalReporter - Logs to a terminal (including colors) with variable verbosity.

Para ello es necesario añadir el modulo jasmine reporters

```
> npm install jasmine-reporters
```

Añadir la declaracion del uso del modulo de reportes a los test.

```
var reporters = require('jasmine-reporters');
var junitReporter = new reporters.JUnitXmlReporter({
    savePath: "D:\\GitLab\\Javascript\\testing\\Jasmine\\workspace
\\GettingStarted",
    consolidateAll: false
});
jasmine.getEnv().addReporter(junitReporter)
```

Y lanzar el comando con el modificador **--report** eligiendo alguna de las opciones.

```
> npm test --report=JUnitXmlReporter
```