Curso HTML, CSS y JavaScript

Junio de 2017

#### Autor: Fran Linde Blázquez

# Tema 3: Test unitarios en JavaScript

ÍNDICE

[Tema 3: Test unitarios en JavaScript 1](#_Toc485978821)

[1. Introducción 3](#_Toc485978822)

[2. Test unitarios 4](#_Toc485978823)

[2.1 NPM 4](#_Toc485978824)

[2.2 Mocha.js 4](#_Toc485978825)

[2.3 Chai.js 4](#_Toc485978826)

[3. Creando un proyecto con test unitarios 5](#_Toc485978827)

[3. Chai JS 7](#_Toc485978828)

[3.1 Assert 7](#_Toc485978829)

[3.2 Expect 8](#_Toc485978830)

[3.3 Should 9](#_Toc485978831)

[3.4 Entendiendo los tests 9](#_Toc485978832)

[3.5 Ejecutando los tests 10](#_Toc485978833)

## 1. Introducción

Uno de los pilares de la programación de calidad es la realización de tests. Si estos pueden ser programados, nos asegurarán rápidamente si nuestro código cumple con nuestras necesidades.

Para poder realizar test automatizados es necesario programarlos. Veamos cómo.

## 2. Test unitarios

Dentro de los test podemos encontrar de varios tipos. Principalmente se dividen en dos:

* Test unitarios: Nos sirven para asegurar de forma individualizada que cierta parte de nuestro código o cierta función, cumple con los requisitos funcionales indicados.
* Test End To End: Estos tests comprueban sobre un navegador que nuestra página o aplicación cumple con los requisitos exigidos a nivel de interacción del usuario.

Vamos a centrarnos en el primer tipo, para lo cual necesitaremos de varias tecnologías, cada una con un fin. Vamos a verlas:

### 2.1 NPM

Es el gestor de paquetes Javascript de Node.js más utilizado. Nos proporciona casi cualquier librería disponible simplemente ejecutando un comando. Esto hace posible que podamos trabajar con una nueva librería en cuestión de segundos.

### 2.2 Mocha.js

Mocha es un framework de pruebas de JavaScript que se ejecuta en Node.js. Nos da la posibilidad de crear tanto tests síncronos como asíncronos de una forma muy sencilla. Nos proporciona muchas utilidades para la ejecución y el reporte de los tests.

### 2.3 Chai.js

Es un librería de aserciones, la cual se puede emparejar con cualquier marco de pruebas de Javascript. Chai tiene varias interfaces: assert, expect y should, que permiten al desarrollador elegir el estilo que le resulte más legible y cómodo a la hora de desarrollar sus tests:

## 3. Creando un proyecto con test unitarios

Una vez presentados lo actores con los que vamos a trabajar, podemos entrar en la parte más técnica y comenzar con nuestro proyecto. (Las instrucciones siguientes servirán para LINUX y MAC OS).

En primer lugar, instalamos npm para poder gestionar de una forma más sencilla las librerías que necesitemos:

sudo apt-get install npm

Creamos el directorio de nuestro proyecto:

mkdir tests\_mocha\_chai

Creamos el fichero package.json. Este fichero es utilizado por npm para, entre otras cosas, almacenar la información referente a nuestro proyecto, como pueden ser las dependencias.

cd tests\_mocha\_chai

npm init

A continuación, instalaremos la librería de mocha.js a través de npm. Le añadimos el parámetro –save al comando para que guarde dicha librería en el fichero package.json.

npm install mocha --save

Por el momento tenemos todo lo que necesitamos, así que vamos a empezar con nuestro código.

En primer lugar, vamos a crear una función, para la cual definiremos test unitarios en pasos siguientes. Para ello, creamos el fichero calculator.js dentro de una carpeta que llamaremos “app” que crearemos en la raíz de nuestro proyecto:

exports.addTested = function(value) {

var result = value + " tested";

return result;

};

Con este código tendremos una función bastante sencilla que simplemente recibe un parámetro y lo devuelve añadiendo la cadena “tested” al final del mismo.

Ya tenemos el código a testear, así que vamos con los tests.

Lo primero que haremos será instalar la dependencia de chai.js de la misma forma que instalamos en pasos anteriores mocha.js:

npm install chai --save

Anteriormente hablamos de tres opciones a la hora de desarrollar tests con Chai: assert, expect, should. Para no centrarnos en ninguna de ellas, vamos a ver como testar la función “addTested” con cada uno de ellas.

## 3. Chai JS

Para ver los tres tipos de aserciones que nos ofrece Chai.js crearemos una carpeta test, y dentro de ella crearemos 3 ficheros: “calculator\_chai\_assert.js”, “calculator\_chai\_expect.js” y “calculator\_chai\_should.js”

Vamos a dar un ejemplo para cada uno de los ficheros:

### 3.1 Assert

Nuestro fichero “calculator\_chai\_assert.js” nos demostrará el uso de should.

Veamos cómo queda nuestro fichero:

var assert = require("chai").assert;

var calculator = require("../app/calculator");

describe("Calcultator tests using ASSERT interface from CHAI module: ", function() {

describe("Check addTested Function: ", function() {

it("Check the returned value using: assert.equal(value,'value'): ", function() {

result = calculator.addTested("text");

assert.equal(result, "text tested");

});

it("Check the returned value using: assert.typeOf(value,'value'): ", function() {

result = calculator.addTested("text");

assert.typeOf(result, "string");

});

it("Check the returned value using: assert.lengthOf(value,'value'): ", function() {

result = calculator.addTested("text");

assert.lengthOf(result, 11);

});

});

});

### 3.2 Expect

Nuestro fichero “calculator\_chai\_expect.js” nos demostrará el uso de expect.

Veamos cómo queda nuestro fichero:

var expect = require("chai").expect;

var calculator = require("../app/calculator");

describe("Calcultator tests using EXPECT interface from CHAI module: ", function() {

describe("Check addTested Function: ", function() {

it("Check the returned value using: expect(value).to.equal('value'): ", function() {

result = calculator.addTested("text");

expect(result).to.equal("text tested");

});

it("Check the returned value using: expect(value).to.be.a('value')): ", function() {

result = calculator.addTested("text");

expect(result).to.be.a('string');

});

it("Check the returned value using: expect(value).to.have.lengthOf(value): ", function() {

result = calculator.addTested("text");

expect(result).to.have.lengthOf(11);

});

});

});

### 3.3 Should

Nuestro fichero “calculator\_chai\_should.js” nos demostrará el uso de expect.

Veamos cómo queda nuestro fichero:

var should = require("chai").should();

var calculator = require("../app/calculator");

describe("Calcultator tests using SHOULD interface from CHAI module: ", function() {

describe("Check addTested Function: ", function() {

it("Check the returned value using: value.should.equal(value): ", function() {

result = calculator.addTested("text");

result.should.equal("text tested");

});

it("Check the returned value using: value.should.be.a('value'): ", function() {

result = calculator.addTested("text");

result.should.be.a('string');

});

it("Check the returned value using: expect(value).to.have.lengthOf(value): ", function() {

result = calculator.addTested("text");

result.should.have.lengthOf(11);

});

});

});

### 3.4 Entendiendo los tests

Por fin tenemos nuestros ficheros de tests creados. Ahora vamos a hablar de su estructura:

**Cabecera:**

En la parte superior del fichero podemos ver algo similar a:

var assert = require("chai").assert;

var calculator = require("../app/calculator");

Esto no son más que las importaciones necesarias para trabajar desde nuestro fichero de pruebas. La primera de ellas es para importar la interfaz de Chai con la que trabajaremos y la segunda para importar el fichero calculator, el cual incluye la función que vamos a probar.

**Describe:**

Utilizamos este elemento para definir bloques de pruebas relacionadas entre sí. Podemos concatenar varios “describe” si lo vemos necesario para nuestra estructura de tests.

**It:**

Cada elemento it sería una prueba. Podemos definir todas las pruebas (it) que sean necesarias dentro de un elemento describe. Las pruebas pueden estar divididas entre varios elementos de tipo “describe”. Esto es útil si queremos diferenciar varios bloques de pruebas en un mismo fichero.

Esta sería la estructura más sencilla de un caso de prueba utilizando Mocha.js. No pensemos que esto es todo. Esta librería nos das muchas más posibilidades. Por ello, hablaremos de las que me resultan más interesantes al final de este post.

### 3.5 Ejecutando los tests

Seguimos avanzando en nuestro proyecto. Ya tenemos tanto una función, como varias pruebas que comprobarán que funciona a la perfección.

Ya que estamos en un proyecto con npm, utilizaremos las posibilidades que nos ofrece para ejecutar nuestras pruebas. Para ello, editaremos el fichero package.json y le daremos valor al parámetro “test” que por defecto lo encontraremos vacío.

Ejemplo de package.json:

{

"name": "tests\_mocha\_chai",

"version": "1.0.0",

"description": "Package de Nuestro proyecto de Tests",

"main": "index.js",

"scripts": {

"test": "./node\_modules/.bin/mocha --reporter spec"

},

"author": "Fran Linde",

"license": "ISC",

"dependencies": {

"chai": "^4.0.2",

"mocha": "^3.4.2"

}

}

Después de actualizar el fichero package.json podemos lanzar los tests con el comando:

npm test

Tras ejecutar el comando debemos obtener una salida en la cual se muestra el resultado de cada uno de los tests ejecutados y la duración completa de los mismos:

