CURSO DESAROLLO WEB CON MEAN

WEB FULL STACK DEVELOPER

Germán Caballero Rodríguez
germanux@gmail.com

Cómo construir una API Rest

The streaming build system

INDICE

TDD (Test Driven Development)

- Refactorización:
  - La refactorización es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.
- La refactorización se realiza a menudo como parte del proceso de desarrollo del software
- Los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad.
- La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad.
- El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro.

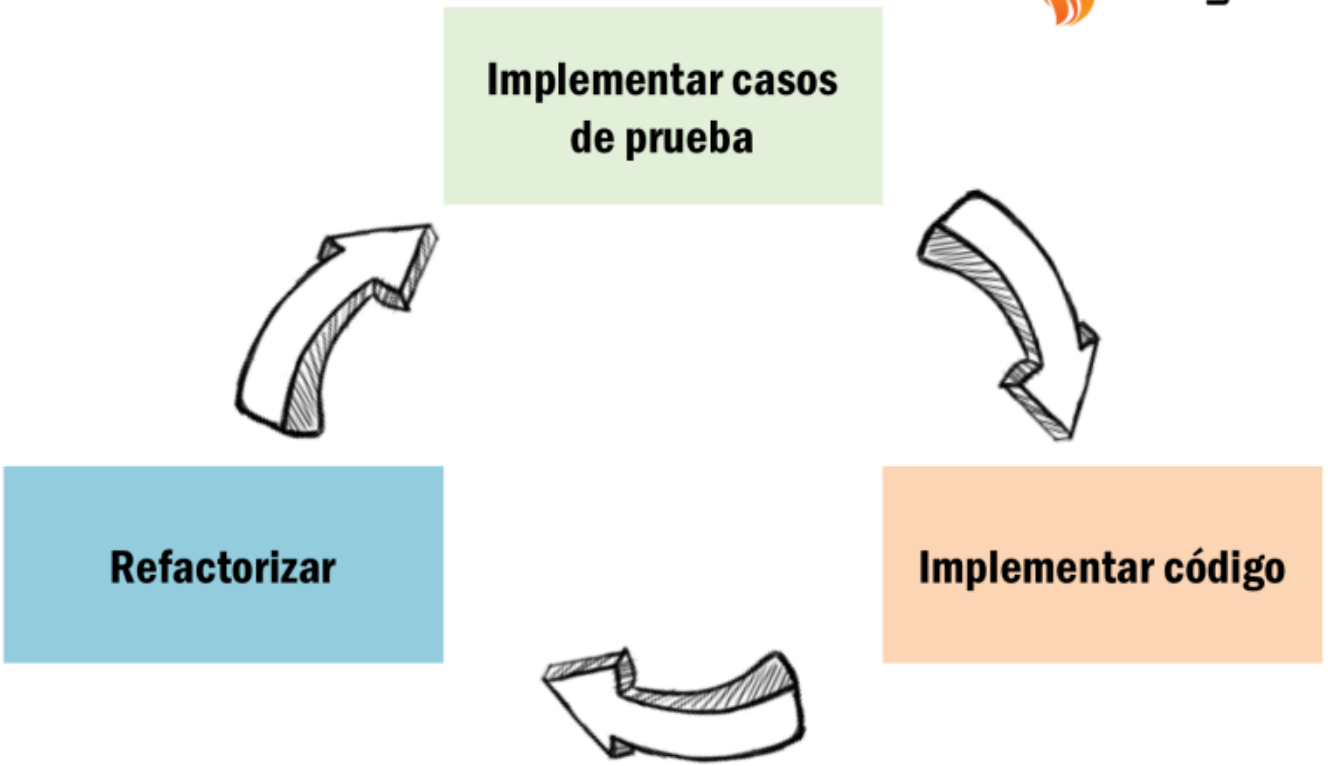
TDD (Test Driven Development)

- También llamado Desarrollo guiado por pruebas (TDD):
- Es una práctica de ingeniería de software que involucra otras dos prácticas:
  - Escribir las pruebas unitarias primero con anterioridad al código (Test First Development)
  - Luego implementar el código que pase esas pruebas y una vez escrito el código, la refactorización posterior al código desarrollado.

TDD (Test Driven Development)

- En primer lugar, se escribe una prueba y se verifica que las pruebas fallan.
- A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.
- El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione.
- La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

TDD (Test Driven Development)



TDD (Test Driven Development)

- La idea de las pruebas unitarias es probar y verificar de manera aislada el funcionamiento de partes específicas de una aplicación, de tal manera que al ejecutar todas las pruebas unitarias se puede asegurar que todas las partes del código están funcionando de manera esperada.

Mocha

- Mocha es un framework de pruebas para JavaScript que se ejecutan en NodeJS y nos ayuda a tener un marco de trabajo para realizar nuestras pruebas de manera ordenada. Mocha es el sucesor oficial de Expresso (otro framework de pruebas de JavaScript).
- Algunas características de Mocha son:
  - Soporte para diferentes navegadores
  - Informes de cobertura de código
  - API Javascript para ejecutar pruebas.
  - Soporte de Debugger para NodeJS.

Mocha

- A efectos prácticos, durante nuestro desarrollo, Mocha nos ofrecerá un entorno en el que agrupar nuestros tests y mostrar los resultados de los mismos.
- Cada bloque de tests se delimita mediante un objeto de tipo “describe” mientras que cada caso a testear esta dentro de un objeto de tipo “it” como se puede ver en la siguiente imagen

Mocha

- Veamos esto con más detalle en un caso práctico:

```
var assert = require("assert");
describe("Cadenas",function(){
    describe("subcadenas",function(){
         it('deberia retornar una subcadena',function(){
            assert.equal("prueba","Esto es una demo de prueba".substring(20,25));
         })
    })
});
```

- La primera línea de este código llama a la librería “assert”para comprobar una funcionalidad específica y poder realizar las “asserts” (se utiliza las clases Assert del espacio de nombres UnitTestingFramework).
- La cuarta línea describe el nombre de la prueba unitaria.
- La quinta línea verifica si la subcadena se encuentra dentro de la cadena.

Mocha

- Comenzando:
  - npm install mocha
- Crear un fichero test.js en la carpeta de desarrollo y donde queramos probar los test.
- Contenido de ejemplo:

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

Mocha

- Para probar el test, volver al terminal:
  - ./node_modules/mocha/bin/mocha
- O simplemente
  - mocha

```
Array
  #indexOf()
    ✓ should return -1 when the value is not present


1 passing (9ms)
```

Mocha

- En nuestro package.json:

```
"scripts": {
    "test": "mocha"
  }
```

- Para ejecutar los test:

```
$ npm test
```

Mocha

- Asserciones:
  - Mocha te permite usar cualquier biblioteca de aserciones que desees.
  - En el ejemplo anterior, estamos usando el módulo de aserción incorporado de Node.js, pero generalmente, si lanza un Error, ¡funcionará!
  - Esto significa que puede utilizar bibliotecas como

should.js - BDD style shown throughout these docs

expect.js - `expect()` style assertions

chai - `expect()`, `assert()` and `should`-style assertions

better-assert - C-style self-documenting `assert()`

unexpected - "the extensible BDD assertion toolkit"

Mocha

- Expect:

```
expect(window.r).to.be(undefined);
expect({ a: 'b' }).to.eql({ a: 'b' })
expect(5).to.be.a('number');
expect([]).to.be.an('array');
expect(window).not.to.be.an(Image);
```

```
$ npm install expect.js
```

Then:

```
var expect = require('expect.js');
```

Mocha

- Expect API:

ok: asserts that the value is *truthy* or not

```
expect(1).to.be.ok();
expect(true).to.be.ok();
expect({}).to.be.ok();
expect(0).to.not.be.ok();
```

be / **equal**: asserts `===` equality

```
expect(1).to.be(1)
expect(NaN).not.to.equal(NaN);
expect(1).not.to.be(true)
expect('1').to.not.be(1);
```

eql: asserts loose equality that works with objects

```
expect({ a: 'b' }).to.eql({ a: 'b' });
expect(1).to.eql('1');
```

Mocha

- Expect API:

match: asserts `String` regular expression match

```
expect(program.version).to.match(/[0-9]+\.[0-9]+\.[0-9]+/);
```

contain: asserts indexOf for an array or string

```
expect([1, 2]).to.contain(1);
expect('hello world').to.contain('world');
```

length: asserts array `.length`

```
expect([]).to.have.length(0);
expect([1,2,3]).to.have.length(3);
```

Mocha

- Modo asincrono en Mocha:

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(function(err) {
        if (err) done(err);
        else done();
      });
    });
  });
});
```

Mocha

- Capturadores:

```
describe('hooks', function() {

  before(function() {
    // runs before all tests in this block
  });

  after(function() {
    // runs after all tests in this block
  });

  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

  // test cases
});
```

Mocha

- Una vez escrita la prueba unitaria ejecutamos el siguiente comando $mocha y por el mismo terminal te mostrará si has completado o no el test (lo veremos más adelante).

Mocha

- Algo más de mocha. Empezamos creando la descripción del caso usando las sentencias describe e it.
- “describe” es una función que acepta dos parámetros:
  - El Primero es un string que describe la principal acción del test. En este caso es: maybeFirst (ahora lo vemos con más detalle).
  - El segundo es una función en la que tenemos que escribir qué es lo que ha de hacer el test.

Mocha

- La sentencia it se trata de una función que, al igual que describe, también acepta dos parámetros:
  - El primero es un string con la descripción exacta de lo que haremos (podemos decir que el nombre de nuestro test).
  - La segunda es otra función que contiene el test definido en expect (expect es una librería de aserciones que nos permite redactar test.

Mocha

```
var assert = require('assert');
var maybeFirst = require('../src/first');

describe('maybeFirst', function() {
    it('returns the first element of an array', function() {
        var result = maybeFirst([1, 2, 3]);
        assert.equal(result, 1, 'maybeFirst([1, 2, 3]) is 1');
    });
});
```

http://www.javiergarzas.com/2016/03/pruebas-unitarias-tdd-nodejs-mocha-parte-2-caso-practico.html