DESARROLLO WEB FULL STACK

Testing de aplicación

Pruebas de software

DWFS COR

Pruebas de software

Las **pruebas de software** (en inglés software **testing**) son las **investigaciones empíricas** y **técnicas** cuyo objetivo es **proporcionar información objetiva e independiente sobre la calidad del producto** a la parte interesada. Es una actividad más en el proceso de **control de calidad**.

Pruebas estáticas

Son el tipo de pruebas que se realizan sin ejecutar el código de la aplicación.

Pruebas dinámicas

Todas aquellas pruebas que para su ejecución requieren la ejecución de la aplicación.

Clasificación

Según su ejecución	Manuales	
	Automáticas	
Según su enfoque	Caja blanca	
	Caja negra	
Según lo que verifican	Funcionales	Unitarios
		De integración
		De humo
	No funcionales	De usabilidad
		De rendimiento
		De stress

Automáticas → Caja negra → Unitarias

Ejemplo

Supongamos que tenemos que realizar el promedio de de un grupo de números.

Para eso creamos dos funciones de utilidad.

```
function sumatoria(elementos) {
  var suma = 0;
  for(var i = 0; i < elementos.length; i++) {
    if (typeof elementos[i] === 'number') {
      suma += elementos[i];
    }
  }

  return suma;
}

function division(a, b) {
  return a/b;
}
```

Ejemplo

Usamos las funciones anteriores para calcular el promedio.

```
function promedio(elementos) {
  var suma = sumatoria(elementos);
  var promedio = division(suma, elementos.length);

  return promedio;
}
```

Casos de sumatoria()

Entrada	Salida
[1, 2, 3]	6
[-1, -2, -3]	-6
[1, -2, 3]	2
[1, 2, 'a']	3

Caso border

Implementación

Vamos a realizar un test para cada una de estas funciones.

Función **sumatoria**.

```
// Caso 1
function testSumatoriaNumerosPositivos() {
  var entrada = [1, 2, 3];
  var resultadoEsperado = 6;
  var salida = sumatoria(entrada);

  return salida === resultadoEsperado;
}

// Caso 2: testSumatoriaNumerosNegativos()
entrada = [-1, -2, -3]; resultadoEsperado = -6;
// Caso 3: testSumatoriaNumerosMixtos()
entrada = [1, -2, 3]; resultadoEsperado = 2;
// Caso 4: testSumatoriaNoNumeros()
entrada = [1, 2, 'a']; resultadoEsperado = 3;
```

Ejecutar los tests

Vamos a realizar un test para cada una de estas funciones.

Función **sumatoria**.

```
function run() {
  console.log(
    'testSumatoriaNumerosPositivos: ',
    testSumatoriaNumerosPositivos() ? 'OK' : 'FAILED'
  );

  console.log(
    'testSumatoriaNumerosNegativos: ',
    testSumatoriaNumerosNegativos() ? 'OK' : 'FAILED'
  );

  console.log(
    'testSumatoriaNumerosMixtos: ',
    testSumatoriaNumerosMixtos() ? 'OK' : 'FAILED'
  );

  ...
}
```

De esta misma forma implementamos los tests de las funciones que faltan.

haganlo :)

Extra mile:
Testear nuestra clase Electrodomestico

Tests runners

Test runners

Los test runners son **herramientas** que **localizan** nuestros **tests**, los **ejecutan** y **muestran el resultado** de forma ordenada y fácil de visualizar, ya sea en logs o archivos.

Mocha:
Test runner para javascript, con interfaz sencilla y fácil de utilizar.

mochajs.org

Assertion Libraries

Assertion library

Una aserción es un predicado, una sentencia que puede ser **verdadera o falsa**. Una librería de aserciones nos provee una forma sencilla de hacer esto, de usar un lenguaje natural para escribir nuestros test.

Chai:
Librería de aserciones BDD/TDD para javascript que puede usarse con cualquier test runner.

chaijs.com

Preparar nuestro entorno para empezar a testear

Entorno

Vamos a realizar un test para cada una de estas funciones.

Función **sumatoria**.

Descargamos:
- JQuery
- Mocha
- Chai

Y lo situamos en una carpeta tests/lib

También creamos un archivo con nuestros tests de la calculadora dentro de tests/

```
<!DOCTYPE html>

<html>
<head>
  <meta charset="utf-8">
  <title>Test Mocha</title>
  <link rel="stylesheet" href="js/test/mocha.css" />
  <script src="js/test/lib/jquery-1.11.2.js"></script>
  <script src="js/test/lib/chai.js"></script>
  <script src="js/test/lib/mocha.js"></script>
  <script> mocha.setup('bdd');</script>
  <script src="js/test/tests.js"></script>
  <script src="js/suma.js"></script>
  <script>
    window.onload = function() {
         mocha.run()
    };
  </script>
 </head>
<body>
  <div id="mocha"></div>
</body>
```

Como hacer nuestros test con mocha y chai

Para que **mocha** pueda **correr los test** que especifiquemos **necesita que sigamos una sintaxis** y hagamos uso de métodos predefinidos que mocha y chai inyecta en nuestros tests.

Hay varias sintaxis soportadas pero en nuestro caso solo utilizaremos

- describe
- it
- expect

Escribiendo tests con mocha y chai

Los tests son mucho más descriptivos y se leen como si fuera en lenguaje natural.

```
var expect = chai.expect;

describe('Test función sumatoria', function () {
  it('Con todos los elementos son positivos', function () {
     var entrada = [1, 2, 3];
     var salida = sumatoria(entrada);
     expect(salida).to.equal(6);
  });

  it('Con todos los elementos negativos', function () {
     var entrada = [-1, -2, -3];
     var salida = sumatoria(entrada);
     expect(salida).to.equal(-6);
  });

  ...
});
```

Interfaces de assertions

Should y expect para BDD nos provee un lenguaje encadenado, expresivo y de fácil lectura.

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Visit Should Guide

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Visit Expect Guide

Interfaces de assertions

Y assert para TDD nos provee una sensación de escritura de tests más clásica.

Refactorización De Código

¿Qué es la refactorización de código?

La **refactorización** (del inglés refactoring) es una **técnica** de la ingeniería de software **para reestructurar un código fuente**, **alterando su estructura interna sin cambiar su comportamiento externo**.

Ejemplo

Refactoricemos nuestra función sumatoria

```
function sumatoria(elementos) {
  var suma = 0;
  for(var i = 0; i < elementos.length; i++) {
    if (typeof elementos[i] === 'number') {
      suma += elementos[i];
    }
  }

  return suma;
}
```

Algunos recursos para nuestra refactorización

Métodos de arrays:

- forEach
- map
- filter
- reduce

Las ya conocidas funciones flecha. :)

() => {}

Ejemplo

refactorizando nuestra función **sumatoria**.

```
function sumatoria(elementos) {
  var suma = 0;
  return elementos
         .filter(el => typeof el === 'number')
         .reduce((suma, el) => suma + el);
}
```


Test Driven
Development (TDD)

¿Qué es TDD?

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una **práctica de ingeniería de softwar**e que involucra otras **dos prácticas**: **Escribir las pruebas primero** (Test First Development) y **Refactorización** (Refactoring).

Para escribir las pruebas generalmente se utilizan las pruebas unitarias. 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.

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.

Ejercicio:
Cuenta bancaria

Ejercicio: Cuenta bancaria.

1) Crea una clase llamada Cuenta que tendrá los siguientes atributos: **titular** y **cantidad** (puede tener decimales).

- El titular será obligatorio y la cantidad es opcional (tiene valor por defecto).
- Crea sus métodos get, set.
- Tendrá dos métodos especiales:
  ingresar(cantidad): se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
  retirar(cantidad): se retira una cantidad a la cuenta, si restando la cantidad actual a la que nos pasan es negativa, la cantidad de la cuenta pasa a ser 0.