🔬 Pruebas unitarias con Jest

Configuración, desarrollo y ejecución de unit tests con Jest.

> *"Es difícil encontrar un error cuando lo estás buscando; es aún más difícil cuando supones que el código está libre de errores."*
>
> -- ✍️ Steve McConnell

Jest

Jest es un framework para pruebas unitarias y de integración.

fáciles de preparar y cómodas de ejecutar

Instalar Jest

```
yarn add jest
npm i --save jest
```

```
{
  "devDependencies": {
    "@babel/core": "^7.9.6",
    "@babel/preset-env": "^7.9.6",
    "@types/jest": "^25.2.2",
    "babel-jest": "^26.0.1",
    "eslint": "^7.0.0",
    "eslint-config-prettier": "^6.11.0",
    "eslint-plugin-jest": "^23.13.1",
    "eslint-plugin-prettier": "^3.1.3",
    "jest": "^26.0.1",
    "prettier": "^2.0.5"
  }
}
```

Configuración

`jest.config.js`.

```
module.exports = {
  verbose: true
};
```

Jest Config

Ejecución

```
{
  "scripts": {
    "test": "jest --watch -o"
  }
}
```

CLI de Jest

Desarrollo de pruebas con Jest

`tu-prueba.spec.js` .

```
export const basic = {
  balance: 0,
  deposit(amount) {
    this.balance += amount;
  },
  withdraw(amount) {
    this.balance -= amount;
  }
};
```

el repositorio del laboratorio

Hola Mundo

```
import { basic } from './basic';

test('basic exists', () => {
  expect(basic).toBeDefined();
});

test('basic balance is 0', () => {
  expect(basic.balance).toEqual(0);
});
```

> Las pruebas se definen como funciones dentro de otras funciones que las ejecutan.

`test()`

`expect(recibido).toEqual(esperado)`;

Comportamiento unitario

Usamos ***Given When Then*** de las pruebas de comportamiento, o el simple ***it should***.

Lo importante, **que quede muy claro el objetivo de la prueba**.

Recuerda el acrónimo **DAMP (Descriptive And Meaningful Phrases)**

```
describe('GIVEN: a basic object', () => {
  test('WHEN: read the balance THEN returns 0', () => {
    expect(basic.balance).toEqual(0);
  });
  test('WHEN: make a deposit of 6 THEN should have a balance of 6', () => {
    const input = 6;
    basic.deposit(input);
    const actual = basic.balance;
    const expected = 6;
    expect(actual).toEqual(expected);
  });
});
```

Fíjate también en la nomenclatura propuesta para las variables.

- `input` : valores de entrada
- `actual` : valores reales obtenidos
- `expected` : resultado esperado

¿Te recuerda a algo?. Sí a la triple ***AAA: Arrange, Act Assert***.

Antes de nada

```
describe('GIVEN: a basic object with a previous balance of 6', () => {
  beforeEach(() => {
    basic.balance = 0;
    const input = 6;
    basic.deposit(input);
  });
  test('WHEN: make a withdraw of 4 THEN should have a balance of 2', () => {
    const input = 4;
    basic.withdraw(input);
    const actual = basic.balance;
    const expected = 2;
    expect(actual).toEqual(expected);
  });
  test('WHEN: make a withdraw of 8 THEN should have a balance of -2', () => {
    const input = 8;
    basic.withdraw(input);
    const actual = basic.balance;
    const expected = -2;
    expect(actual).toEqual(expected);
  });
});
```

Vuelve la triple AAA

```
describe('GIVEN: a basic object with a previous balance of 10 WHEN: i ask for a borrow of 4', () => {
  beforeAll(() => {
    arrangeBalance();
    const input = 4;
    actBorrow(input);
  });
  test('THEN should have a balance of 14', assertBalance);
  test('AND THEN should have disposed of 4', assertDisposed);
});
function arrangeBalance() {
  const input = 10;
  basic.balance = input;
}
function actBorrow(input) {
  basic.borrow(input);
}
function assertBalance() {
  const actual = basic.balance;
  const expected = 14;
  expect(actual).toEqual(expected);
}
function assertDisposed() {
  const actual = basic.disposed;
  const expected = 4;
  expect(actual).toEqual(expected);
}
```

Estas micro funciones cumplen varios cometidos:

- Al tener nombre dejan un rastro para depuración.
- Aclarar el propósito a otro programador: preparar, actuar y comprobar.
- Además, algunas veces son reutilizables; pero eso no es lo fundamental.

Sin fallos

Todo código puede fallar. Pero si la la excepción ya es esperada, entonces la prueba debe también comprobar que lo excepcional funciona como se espera.

```
  borrow(amount) {
    if (amount + this.disposed > this.balance) {
      throw "you can't request so much credit";
    }
    this.disposed += amount;
    this.balance += amount;
  },
```

Esta función comprueba una condición lógica y lanza un error si no se cumple. Debemos capturar ese error y asegurar que se lanza cuando es preciso.

```
describe('GIVEN: a basic object with a previous balance of 10 WHEN: i ask for a borrow of 40', () => {
  beforeAll(() => {
    arrangeBalance();
  });
  test('THEN should receive an error', assertDisposedThrowsError);
});
function arrangeBalance() {
  const input = 10;
  basic.balance = input;
}
function assertDisposedThrowsError() {
  const input = 14;
  expect(() => actBorrow(input)).toThrow();
}
```

Aquí la sintaxis es un poquitín rara, pero necesaria para que *Jest* haga internamente el trabajo sucio de envolver en `try{}catch(e){}` tu sujeto de pruebas.

Código cubierto

¿Cuántas líneas se ejercitan? Al porcentaje sobre el total se le llama cobertura.

```
  {
    "coverage": "jest src/unit/basic/basic.spec.js --collect-coverage",
  }
```

Configuración que especifica umbrales aptos en `jest.config.js`:

```
module.exports = {
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80
    },
  },
}
```

En principio **una mayor cobertura es un signo de mayor confianza en la prueba**.

En el se incluyen el famoso 80% para líneas, ramas condicionales y funciones.

> No te obsesiones con esta métrica.

Cada test que hagas estarás más cerca del objetivo: **tener confianza en tu código y dormir tranquilamente**.

Tienes el ejemplo completo en el laboratorio del curso de **Testing con Jest**.