



Pruebas de código con Jest

Configuración, desarrollo y ejecución de 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**

Las pruebas de código deben ser **fáciles de preparar y cómodas de ejecutar**. Estas son las cualidades que aporta *Jest* al mundo del testing. Son pruebas de caja blanca, dónde conocemos y tenemos delante el código que vamos aprobar. Con Jest hacer las pruebas del código o el código que supere las pruebas es sencillo y cómodo. Vamos a verlo por pasos.

Jest

Jest es un framework para pruebas unitarias. Centrado en determinar que todo tu código es correcto. Para ello vamos a empezar por el principio.

Instalar Jest

Instalarlo es cosa de niños. Partimos de una aplicación *Node* y agregamos la dependencia. Puedes hacerlo con *npm* o con *yarn* y guardarlo como dependencia principal o para desarrollo.

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

Y ya está. Aunque yo recomiendo agregar unas dependencias extra para facilitar aún mas el uso. Incluyo el analizador de código *eslint*, el embellecedor *prettier* y el empaquetador *babel* (para evitar usar los *require* de Node). En VSCode también activo la extensión [orta.vscode-jest](#) para que me ayude en línea.

```
{
  "devDependencies": {
    "@babel/core": "^7.11.5",
    "@babel/preset-env": "^7.11.5",
    "@types/jest": "^26.0.13",
    "babel-jest": "^26.3.0",
    "eslint": "^7.8.1",
    "eslint-config-prettier": "^6.11.0",
    "eslint-plugin-jest": "^23.20.0",
    "eslint-plugin-prettier": "^3.1.4",
    "jest": "^26.4.2",
    "jest-fetch-mock": "^3.0.3",
  }
}
```

Configuración

Nada. Como lo oyes, no hay nada que hacer. Simplemente ejecutarlo y listo. Por supuesto que lo puedes adaptar o manipular, pero sólo lo haremos cuando lo necesitemos.

Ya, pero quieres saber cómo... Pues hay varias opciones; a mi me gusta configurarlo en su propio fichero `jest.config.js`.

```
// For a detailed explanation regarding each configuration property, visit:
// https://jestjs.io/docs/en/configuration.html
module.exports = {
  verbose: true
};
```

En este caso le he dicho que quiero explicaciones detalladas de lo que va ocurriendo.

Ejecución

Las pruebas se pueden lanzar desde línea de comandos o mejor *escribiéndolo* en el `package.json`. Algo así es suficiente para empezar.

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Aquí le decimos que se lancen todas las pruebas, que vigile los cambios para relanzarlas, pero sólo aquellas afectadas desde el último *commit*. Tienes más información en la página dedicada al [CLI de Jest](#)

Desarrollo de pruebas con Jest

¿Ejecutar? ¿El qué? Pues una **especificación**, o en el argot de *Jest* un fichero `tu-prueba.spec.js`.

Vamos a realizar un conjunto de pruebas muy sencillo. Es un *Hola Mundo* en JavaScript, pero un poco mejorado con algún control de argumentos como este:

```
export function sayHi(userName) {
  if (isNotAString(userName)) {
    throw `What kind of name is ${userName}?`;
  }
  if (isEmpty(userName)) {
    return `I don't know your name`;
  }
  return `Hello ${userName}!`;
}

function isNotAString(userName) {
  return typeof userName !== 'string' && !(userName instanceof String);
}

function isEmpty(userName) {
  return userName.length === 0;
}
```

Ya lo sé, es muy básico; pero esto es solo un *Hola Mundo*.

Hola Mundo

```
import { sayHi } from './0-hello-world';

test('Say Hi', () => {
  expect(sayHi('Jest')).toBe('Hello Jest!');
});
```

Evidente, ¿no?. Si somos novatos en las pruebas sólo tenemos que familiarizarnos con un concepto común a otros frameworks.

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

En este caso la función `test()`, también usable bajo el alias `it()` es la función clave de todas tus pruebas *Jest*.

Es una función que recibe dos argumentos. El primero es una cadena que describe la prueba y el segundo es otra función con la prueba en sí.

Dentro de esa función interna encontraremos llamadas a más funciones del framework; que casi siempre seguirán una sintáxis similar a esta `expect(actual).toEqual(expected);`

Código cubierto

Al lanzar las pruebas se ejecuta también el sujeto de pruebas, el SUT. Pero **¿Cuántas líneas se ejercitan?** Pues depende de lo cuidadoso que seas al especificar las condiciones y las expectativas. Al porcentaje de líneas que se ejecutan sobre el total se le llama cobertura.

En principio **una mayor cobertura es un signo de mayor confianza en la prueba**. Pero no es determinante. De todas formas se considera que 80% es un buen indicador y *Jest* te ofrece ese dato y otros muchos.

Como siempre, te sugiero que *escriptes* todos tus comando de consola.

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

Solicitar el informe de cobertura de código es así de simple; pero necesita que en la configuración le especifique qué umbrales consideras aptos. Yo suelo utilizar algo así en el `jest.config.js`:

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

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

Al sólo haber probado el *Happy Path* vemos que nuestra cobertura deja mucho que desear. Así que vamos añadiendo pruebas hasta que alcancemos los límites propuestos.

```
import { sayHi } from './0-hello-world';

test('Say Hi', () => {
  expect(sayHi('Jest')).toBe('Hello Jest!');
  expect(sayHi('')).toBe('I don't know your name');
  expect(() => sayHi(42)).toThrow('What kind of name is 42?');
});
```

Si estás empezando con las pruebas 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.**



Pruebas de integración

Pruebas de sistemas legacy complejos.

"Los probadores de software siempre van al cielo; Ya han tenido su parte de infierno."

--  **Tester anónimo**

La mayoría del software empresarial se ha escrito sin pruebas. Y esto dificulta mucho su mantenimiento.

Hacer pruebas sobre código heredado es costoso y poco atractivo. Pero hay que hacerlo.

Lo contrario implica hacer las mínimas modificaciones y siempre con la inquietud de haber roto algo. Desde luego ya no hablamos de refactoring. *Si funciona... no lo toques.* Y así nos va, con software mal diseñado en el cual es difícil arreglar defectos o añadir funcionalidad.

Y la solución pasa por hacer pruebas. **Las pruebas automáticas son una inversión rentable** que te permite asegurar el funcionamiento de un programa mientras lo modificas para corregir, mantener o mejorar.

Código heredado

Con *Jest* es relativamente sencillo probar este código heredado, que quizá sea reciente, quizás incluso sea tuyo. No importa, vamos a ver qué situaciones y problemas nos encontramos.

El ejemplo propuesto es un sistema bancario ridículamente simple en tres ficheros.

Se trata de una clase **Account** con métodos de negocio para ingresos y gastos y otra **Clerk** para ejecutar operaciones y otra **Tranasactions** almacenar dichas operaciones.

Happy Path

Con lo que sabemos de *Jest* es fácil entender esta prueba; y entendiendo esta prueba es fácil adivinar la funcionalidad del programa. Además seguimos usando el patrón **given-when-then** y los nombrados de variables también por convenio **sut**, **input**, **actual**, **expected**.

```
import { Account } from './bank/account';

describe('GIVEN a new account with a deposit', () => {
  const sut = new Account();
  const input = 20;
  sut.deposit(input);
  test('SHOULD have the correct balance', () => {
    const actual = sut.getBalance();
    const expected = 20;
    expect(actual).toBe(expected);
  });
});

describe('GIVEN a new account with two deposits', () => {
  const sut = new Account();
```

```
const inputA = 20;
sut.deposit(inputA);
const inputB = 10;
sut.deposit(inputB);
test('SHOULD accumulate the amounts in the balance', () => {
  const actual = sut.getBalance();
  const expected = 30;
  expect(actual).toBe(expected);
});
});
```

Excepciones

¿Todo bien? Más o menos. El caso es que este código puede demostrar que el programa funciona, eso está bien; y se puede extender para realizar pruebas más complejas (manejo de descubiertos, aportaciones más o menos generosas...). Por ejemplo en otro fichero podríamos probar las retiradas de dinero y sus límites aceptables.

```
describe('GIVEN a new account with slightly more withdraw than deposit', () => {
  const sut = new Account();
  const inputDeposit = 15;
  sut.deposit(inputDeposit);
  const inputWithdraw = 20;
  sut.withdraw(inputWithdraw);
  test('SHOULD have a negative balance', () => {
    const actual = sut.getBalance();
    expect(actual).toBeLessThan(0);
  });
});

describe('GIVEN a new account with a lot more withdraw than deposit', () => {
  const sut = new Account();
  const inputDeposit = 15;
  sut.deposit(inputDeposit);
  const inputWithdraw = 200;
  test('SHOULD throw an exception', () => {
    expect(() => sut.withdraw(inputWithdraw)).toThrow();
  });
});
```

Fíjate por ejemplo cómo se prueban las excepciones. Es muy importante comprobar que el código se comporta de la manera esperada justo en los peores momentos.

Asincronismo

En JavaScript más temprano que tarde te vas a encontrar con código asíncrono. En programación *front-end* es el día día; y en el *back...* también.

Pero actualmente tenemos técnicas de desarrollo asíncrono muy sencillas como los comandos `async` y `await` y que se implementan perfectamente en *Jest*.

Suponiendo que ahora nuestro sistema almacenase y recuperase las transacciones en un almacén remoto, todo el proceso pasaría a ser asíncrono. Y eso en *Jest* es casi transparente. Solamente tendremos que anotar las función de pruebas con los comandos `async` y `await`

```
import { Account } from './bank_async/account';

describe('a new async account with a deposit', () => {
  const sut = new Account();
  const input = 20;
  test('should have the correct balance', async () => {
    await sut.deposit(input);
    const actual = sut.getBalance();
    const expected = 20;
    expect(actual).toBe(expected);
  });
});
```

A pesar de que este código es correcto, es posible que te encuentres con que no puedes ejecutar estas pruebas. ¿Cómo? Pues porque al ser *Jest* un framework originalmente basado en Node, pues se lleva mal con algunas librerías típicas del front. Yo lo he forzado en el laboratorio usando `fetch` para las llamadas remotas. Forcé el problema para mostrarte una solución.

Mocks

La teoría del testing nos dice que podemos usar dobles de las dependencias reales. Sobre todo si estas nos impiden ejecutar las pruebas, como en este caso. Recorro a un *mock* de las funciones `fetch`.

Afortunadamente no hay que programar nada porque los problemas comunes tienen soluciones comunes y públicas. En este caso con el paquete `jest-fetch-mock` que se instala y luego se invoca en una sola instrucción.

```
require('jest-fetch-mock').enableMocks();
import { Account } from './bank _async/account';
```

¿Entonces? Ya está todo bien, ¿no?. Mas o menos, pues la trampa está en que es todo esto es una **prueba de integración**.

En esta prueba estamos ejercitando a `Account` y *sin querer* a a todas sus dependencias. Es decir damos por bueno que `Clerk` y `Transactions` también funcionan. ¿Y eso es malo? No necesariamente; si la prueba pasa es una buena dosis de confianza. Pero si no la pasa... entonces no sabremos gran cosa sobre el motivo del fallo.

Pero tras refrescar el concepto de *mock* la mejora de esta situación está cerca.



TDD, ciclo virtuoso RGR

Desarrollo guiado por las pruebas. Todo empieza definiendo un test.

"Haz lo más simple que pueda funcionar."

--  **Kent Beck**

La idea de **TDD, Test Driven Development**, es hacer antes las pruebas que el código. Al principio esto te resultará antinatural. Requiere esfuerzo. ¿Por qué hacerlo así?

- Si haces las pruebas antes... bien, porque al menos tienes **las pruebas hechas**.
- Solo tienes que hacer **el mínimo código** que pase la prueba. Nada más.
- Para poder probar fácilmente, harás un código fácil de manejar; **mejor diseñado**.

Todo empieza con los requerimientos

Antes de hacer nada, conviene **saber qué vamos a hacer**. Esto es, conocer los requerimientos funcionales del software. Te los pueden dar de manera más o menos formal. Pero en cualquier caso tú puedes adaptarlos al estándar que mejor te convenga.

A lo largo de este curso he empleado una versión sencilla de las historias de usuario. Creo que cualquiera las puede entender, y hay pocos estándares que exijan menos. Para el caso de las pruebas unitarias lo adaptaré tanto que casi debería llamarle **historias de programador**. Son mucho más granulares y definen el detalle de un proceso, conociendo las tripas del sistema. Son pruebas de **caja blanca** absoluta, y su documentación es una por tanto mucho más técnica y precisa.

```
FEATURE: a BankClient account
  As a: high level service
  I want to: have a class where deposit money
  In order to: accumulate several amounts of money for later
```

Esto aún es muy genérico, pero podemos mejorarlo con la especificación de lo casos de comportamiento esperado. Usando como plantilla el **GWT, Given When Then**, podemos ir poco a poco haciéndolos cada vez más detallados.

```
Given: GIVEN a Bank Client class
When: I make a deposit of 10
Then: it should returns the running balance of 10
```

De aquí sacaremos directamente las cadenas de texto que acompañan a las pruebas, tanto en ejecución como en desarrollo.

Las pruebas

Las pruebas **TDD son pruebas para programadores**. Las hacemos por nuestro propio bien. Sin que nos las pidan, sin esperar que las valoren.

Hacemos las pruebas para estar seguros de hacer lo que se pide, nada más, pero bien hecho.

La estructura, los textos y el cómo se hacen debe ser a nuestro gusto. Yo te propongo seguir con la estructura AAA y el nombrado GWT. Pero repito, estas pruebas son para ti, es posible que no las vea nadie que no vea el código. Están al mismo nivel.

Empezamos.


```
describe(`GIVEN a Bank Client class`, () => {
  test(`THEN I can create an instance`, () => {
    const sut = new BankClient();
    expect(sut).toBeInstanceOf(BankClient);
  });
  test(`THEN I can make a deposit`, () => {
    const sut = new BankClient();
    sut.deposit();
    expect(sut).toBeInstanceOf(BankClient);
  });
  test(`WHEN I make a deposit of 10 THEN it should returns the running balance of 10`, () => {
    const sut = new BankClient();
    const input = 10;
    const actual = sut.deposit(input);
    const expected = 10;
    expect(actual).toEqual(expected);
  });
});
```

Y la ejecutamos... y falla. ●

La implementación

Ahora que hemos visto fallar a nuestra prueba, vamos a hacer que la pase. ¿Cómo? Escribiendo **el mínimo código que satisfaga** la especificación funcional descrita.

```
export class BankClient {
  constructor() {}
  deposit(amount) {
    return 10;
  }
}
```


Listo , vámonos a casa que se está haciendo de noche.

La mejora


¿Sigues ahí? Ya, te crees que te estoy tomando el pelo. Pero no. Normalmente el mínimo código que pasa una prueba se resuelve con una constante y, la verdad, es poco práctico. Poco variable mejor dicho.

Es momento de hacer dos cosas. Lo primero enriquecer las pruebas agregando un nuevo caso que impida resolver el trabajo con una constante.,

```
test('WHEN I make a deposit of 15 THEN it should returns the running balance of 15', () => {
  const sut = new BankClient();
  const input = 15;
  const actual = sut.deposit(input);
  const expected = 15;
  expect(actual).toEqual(expected);
});
```

Ok, ya veo dónde falla . Realmente quieres que se te devuelva lo mismo que ingresas. No puede ser más fácil.

```
export class BankClient {
  constructor() {}
  deposit(amount) {
    return amount;
  }
}
```


Ahora sí que está bien .

ni mucho venos, ¿verdad? Vamos a seguir enriqueciendo la prueba explicando realmente lo que queremos que ocurra al agregar varios importes. Agrega este caso

```
test('WHEN I make a deposit of 10 and then a new deposit of 15 THEN the last one should return the accumulated 25', () => {
  const sut = new BankClient();
  const firstInput = 10;
  sut.deposit(firstInput);
  const secondInput = 15;
  const actual = sut.deposit(secondInput);
  const expected = 25;
  expect(actual).toEqual(expected);
});
```

Vaya , parece que necesitare algún tipo de acumulador... Hagámoslo


```
export class BankClient {  
  constructor() {  
    this.acumlador = 0;  
  }  
  deposit(amount) {  
    this.acumlador += amount;  
    return this.acumlador;  
  }  
}
```

Correcto de nuevo . Imagino que vas pillando el sistema. Pasito a pasito. Escribiendo el código que pase la prueba. Refinándola para cubrir más casos. Escribiendo código para pasar la nueva prueba.

El ciclo virtuoso


Este ciclo descrito se completa con un proceso de **refactoring, o mejora en el diseño**. Este trabajo se realiza sobre el código correcto; lo recalco, **es una mejora**. Pero al hacerlo sobre código respaldado por las pruebas nos permite realizar los cambios con plena tranquilidad.

```
export class BankClient {  
  constructor() {  
    this.balance = 0;  
  }  
  deposit(amount) {  
    this.balance += amount;  
    return this.balance;  
  }  
}
```

Pequeñas mejoras constantes . Caminando despacio sobre suelo seguro. Es el ciclo virtuoso completo:

 RED : definir la prueba y comprobar que falla.

 GREEN : Escribir el mínimo código posible que satisfaga la prueba.

 REFACTOR : Mejorar dicho código manteniendo el respaldo de la prueba.

Repetir este ciclo refinando y creando nuevas pruebas hasta completar el requerimiento funcional completo.

Diseño integrado

Mejores resultados y mejor diseño. Hacer las pruebas antes mejora el código de después.

"TDD te hace escribir código más desacoplado, lo cual mejora el diseño del sistema."

--  **Robert C. Martin (Uncle Bob)**

Aprender algo es costoso, incluirlo en tu rutina lo es aún más. Tenemos que visualizar el objetivo para motivarnos. E ir paso a paso para no desmotivarnos.

Venga, vamos a continuar con nuestro micro sistema bancario según vimos en la [introducción a TDD](#).

Test first

Supongamos que nos piden que el sistema sea capaz de obtener un balance a partir de transacciones anteriores.

```
FEATURE: a BankClient account
As_a: high level service
I_want_to: have a class where deposit money
In_order_to: accumulate several amounts of money for MUCH later
```

Pues empezamos por especificar nuestros deseos: un método llamado `getBalance` estaría bien.

```
describe('GIVEN: a calculate balance function', () => {
  test('WHEN I make a deposit of 10 THEN any new instance should returns the
running balance of 10', () => {
    const inputSut = new BankClient();
    const input = 10;
    inputSut.deposit(input);
    const sut = new BankClient();
    const actual = sut.getBalance();
    const expected = 10;
    expect(actual).toEqual(expected);
  });
});
```

Con esto podemos empezar, obviamente habría que incluir más casos. Pero tenemos la idea.

Better implementation

La implementación en la clase `BankClient` no es para el premio Turing de informática; pero tiene algo implícitamente bueno: Se ha creado un método, se ha nombrado según el uso esperado y usando el código ya hecho, como la propiedad `this.balance`.

```
getBalance() {  
  return this.balance;  
}
```

Dependencias

Recuerda que nos piden que las transacciones se persistan. De modo que habrá que disponer de funciones que almacenen y que lean transacciones. Pero no nos han dicho nada de su implementación.

Desde luego no queremos incorporar a la prueba el conocimiento de cómo se haga la implementación. Y al mismo tiempo nos interesará mucho mantener las pruebas sencillas, con la menor necesidad de dobles y mocks.

Mejoras paso a paso

Y con esta tranquilidad vamos incorporando funcionalidad. Siempre definiendo antes la prueba. Viendo el fallo por falta de implementación y luego hacerla pasar de manera minimalista.

Refactored

Es el siguiente nivel. Mejorar el código, y si es necesario la prueba, pero manteniendo segura la funcionalidad. Esto produce buenos resultados probados y mejor diseño.

Si el código está desacoplado es muy sencillo mantenerlo. Este es el objetivo del software bien diseñado. Para conseguirlo merece la pena el esfuerzo invertido.



Refactoring y rediseño

La mejora continua

"La excelencia no es un destino, es un viaje que no termina nunca"

--  **Brian Tracy**

Las pruebas de TDD vistas hasta el momento eran de integración. están muy bien porque por un lado prueban el sistema y con el TDD al escribir el código tras la prueba, nos obligamos a ocultar las dependencias.

Unit TDD

El problema de las pruebas de integración es que cuando fallan no sabemos con precisión dónde esta el fallo. Justamente para eso están las pruebas unitarias, par ir al detalle y saber dónde hay un problema.

Ejemplo

En el Laboratorio tienes la carpeta **3-design** con los ficheros de código y pruebas. Hay una versión previa al refactoring **3_0-bank-client** y otra posterior **3_1-bank-client**.

Dependencias

Lo que hacemos es eliminar las dependencias. Al empezar no queremos saber nada de cómo almacenar el balance para su futuro.

Pero sabemos que lo necesitaremos, así que en lugar de crearlo pediremos que nos lo den hecho. Es decir declararemos una dependencia.

```
export class BankClient {
  constructor(store) {
    this.balanceStore = store;
    this.balance = this.balanceStore.load();
  }
  deposit(amount) {
    this.balance += amount;
    this.balanceStore.save(this.balance);
    return this.balance;
  }
  getBalance() {
    return this.balance;
  }
}
```

Esta es la gran ventaja, al hacer TDD con mentalidad unitaria hemos incorporado el patrón de inyección de dependencias sin despeinarnos. El código es ahora mucho más flexible.

Dobles y espías

Ahora desde la prueba tengo que inyectar esa dependencia. ¿Y qué le paso en ese constructor? Elimina de tu cabeza la posibilidad de pasarle el almacenador real. Para empezar aún no existe, pero si así fuese estaríamos dependiendo de que funcionase bien. Es decir volveríamos a la integración. Y no quiero; sólo quiero probar BankClient

La solución esa por utilizar algún tipo de doble. Es decir algo que durante la prueba sustituya a lo que en realidad se usará en ejecución. *Jest* nos facilita mucho la creación de un tipo de doble muy útil: el espía.

```
test('WHEN I make a deposit THEN save to the store will be called one time', () =>
{
  const inputStore = {
    load: () => {},
    save: () => {},
  };
  const saveSpy = jest.spyOn(inputStore, 'save');
  const inputSut = new BankClient(inputStore);
  const input = 10;
  inputSut.deposit(input);
  expect(saveSpy).toBeCalledTimes(1);
});
```

Como ves, se trata de pasarle algo que el sistema bajo pruebas reconozca como una dependencia usable. En este caso que tenga los métodos `load` y `save`. Y lo más interesante, después comprobaremos sus invocaciones. Eso es lo que hace nuestro agente, espiar las llamadas a esas dependencias.

De esa forma tú sólo pruebas una clase, sin depender de sus dependencias. Esto es clave para un buen diseño flexible, que es la máxima aspiración del código: que sea correcto y mejorable.