UD08 - Testing

1. Introducción al testing

Es fundamental que nuestro código tenga un cierto nivel de calidad para minimizar los fallos del programa, más cuanto más compleja es la aplicación que estamos desarrollando. Para ello debemos testearlo y dicho testeo seguramente incluirá test automáticos. Dichos test nos permiten:

- comprobar que nuestro código responde como se espera de él
- evitar los *errores de regresión* (fallos tras incluir una nueva funcionalidad en cosas que antes funcionaban en nuestro programa)
- incluso mejoran la documentación del proyecto ya que el test indica cómo debe funcionar mi código

Como ya sabéis existen varios tipos de tests:

- unitarios: prueban un trozo de código que sólo hace una cosa (habitualmente una función)

- de integración: prueban que varias partes del código funcionan bien juntas
- de aceptación: prueba que el código permite hacer algo que el cliente quiere qu pueda hacerse

De momento desarrollaremos tests unitarios. Estos tienen 3 partes:

- Preparación (*Arrange*): perparamos el código para poder probarlo, por ejemplo, creamos las variables u objetos a probar
- Actuación (*Act*): realizamos la acción, por ejemplo, llamamos a la función
- Aserción (*Assert*): comprobamos que el resultado es el esperado

Ejemplo:

```
test('wordCount() returns 2 when the input is "Hello world"', () => {
  // Arrange
  const string = 'Hello world';

  // Act
  const result = wordCount(string);

  // Assert
  expect(result).toBe(2);
});
```

2. Testing en Javascript

Tenemos muchas herramientas para hacer tests unitarios. Usaremos una llamada ***Jest***. Para instalarla usaremos el gestor de paquetes ***npm*** que es el más utlizado para usar bibliotecas y sus dependencias en el FrontEnd.

Instalar npm

npm es el gestor de dependencias de ***nodejs*** y aprenderemos más de él en el bloque de **Vue**. Debemos instalar *NodeJS* para tener npm. Esto podemos hacerlo desde el repositorio de nuestra distribución (con `apt install nodejs`) pero no se instalará la última versión. Es mejor seguir las indicaciones de la página oficial de NodeJS. Aquí tenéis cómo hacerlo para distribuciones basadas en Debian/Ubuntu.

Instalar jest

Una vez instalado npm crearemos una carpeta para cada proyecto que vayamos a hacer y lo inicializamos ejecutando dentro de ella:

```
npm init
```

Este comando crea un nuevo proyecto y nos pedirá información sobre el mismo. Cuando nos pregunten por la herramienta para hacer tests escribiremos **jest**. Tras ello tendremos ya creado el fichero **package.json** de nuestra aplicación (es el fichero donde se configura el proyecto y sus dependencias). En el apartado de *scripts* encontramos uno llamado *test* que lo que hace es ejecutar *jest*:

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

Ahora falta instalar jest, lo que haremos con:

```
npm install --save-dev jest
```

Estamos instalando jest sólo como dependencia de desarrollo ya que no lo necesitaremos en producción (lo mismo abreviado sería `npm i -D jest`).

Aunque, como vamos a utilizar *jest* en muchos proyectos distintos, es más conveniente instalarlo globalmente con

```
npm i -g jest
```

De esta forma no tendremos que instalar *jest* en cada nuevo proyecto, sólo hacer el `npm init`.

Las dependencias que instalemos están en el directorio *node_modules*. Si estamos usando *git* debemos asegurarnos de incluir este directorio en nuestro fichero *.gitignore* (si no tenemos ese fichero podemos crearlo simplemente con `echo "node_modules" > .gitignore`).

Transpilar nuestro código

Vamos a crear las funciones de nuestro código en un fichero JS y para que se puedan usar el otro fichero Javascript (el de los tests) debemos exportarlas con `module.exports`. El fichero de test deberá importarlas con `require` (se explica más adelante, en el apartado de Usar Jest). Por ejemplo, tenemos un fichero llamado **suma.js** que contiene la función *add* que suma 2 números pasados por parámetro:

```
function add(a, b) {
  return a + b;
}
module.exports = add;
```

El fichero de test, **suma.test.js** (normalmente le llamaremos igual pero anteponiendo *.test* a la extensión .js) contiene los test a ejecutar:

```
const add = require('./suma')

describe('Addition', () => {
    test('given 3 and 7 as inputs, should return 10', () => {
        const expected = 10;
        const actual = add(3,7);
        expect(actual).toEqual(expected)
    });

    test('given -4 and 2 as inputs, should return -2', () => {
        const expected = -2;
        const actual = add(-4,2);
        expect(actual).toEqual(expected)
    });
});
```

Lo que hace es:

- importa la función que exporta *suma.js* y la almacena en la constante **add**. Ya pude llamar a esa función
- el bloque *describe* permite agrupar varios tests relacionados bajo un mismo nombre
- cada sentencia *test* es un test que se realizará

Si ejecutamos los tests en la terminal (`npm run test`) muestra un error ya que Jest no sabe cómo gestionar las sentencias ECMAScript *import* y *export*. Para solucionarlo debemos transpilar nuestro código de manera que Jest pueda entenderlo. Podemos hacerlo de 2 maneras:

- instalando el transpilador **Babel** y configurando *Jest* para que transpile el código
- utilizando un *bundler* como **Webpack**. En este caso no sólo transpilamos el código sino que juntamos todos nuestros ficheros JS en uno sólo que será el que enlazaremos en el fichero HTML de nuestra aplicación. Es la solución si queremos que nuestro código funcione en el navegador además de poder pasar los tests.

Usando Babel con Jest

Si queremos sólo poder pasar los tests pero no vamos a usar ese código en el navegador sólo tenemos que instalar el transpilador Babel:

```
npm add jest babel-jest @babel/core @babel/preset-env
```

Y crear 2 ficheros para configurarlo y que sepa trabajar junto a Jest:

- **jest.config.json**

```
{
    "transform": {
        "^.+\\.jsx?$": "babel-jest"
    }
}
```

- **.babelrc**

```
{
    "presets": ["@babel/preset-env"]
}
```

Ahora ya podemos ejecutar los test y comprobar que nuestro código los pasa.

En la siguiente página explica cómo configurar npm y jest con babel (sin usar webpack) e integrarlo con Travis-CI, la herramienta de integracion continua de GitHub:

- Automate NPM releases with Jest, codecov.io, Semantic Release, and TravisCI

Usando Webpack

Con la configuración anterior nuestro código es transpilado para ejecutar los tests, pero dará error si intentamos ejecutarlo en el navegador porque allí no está transpilado. Podemos solucionarlo usando *webpack* para empaquetar y transpilar el código (por tanto no sería necesario realizar lo indicado en al apartado anterior).

Webpack el un *bundler* o empaquetador de código que además puede usar transpiladores para convertir nuestro código que usa versiones modernas de ECMAscript en otro soportado por la mayoría de navegadores.

Por tanto nos va a permitir, entre otras cosas:

- Tener en nuestro *index.html* una sóla entrada de script (`<script src="./dist/main.js'>`) en lugar de una para cada archivo que estemos utilizando (index.js, functions.js, ...)
- Además podremos usar instrucciones como `module.exports` para exportar funciones o `require` para importarlas en otro fichero Javascript, que sin traspilar provocarían errores en el navegador

Existen infinidad de páginas que nos enseñan las mil posibilidades que tiene *webpack*, pero nosotros por ahora sólo necesitamos:

Instalar webpack

Tenemos que instalar webpack y webpack-cli. Como son dependencias de desarrollo (en producción no las necesitaremos) ejecutamos:

```
npm i -D webpack webpack-cli
```

Ejecutar webpack

Se ejecuta con el comando `npx webpack` y hay que indicarle:

- cuál es nuestro archivo Javascript principal de nuestro código (si no lo ponemos supondrá que es **./src/index.js**)
- cuál será el archivo que creará de salida (por defecto **./dist/main.js**)
- si estamos en desarrollo o en producción, para permitir o no depurar el código generado

Siguiendo con el ejemplo anterior de la suma crearemos un fichero **index.js** dentro de *src/* que importará el fichero *suma.js* (con el comando `require`como se hace en el fichero de tests) y que contendrá el resto de código de la aplicación (como pedir al usuario los números a sumar, mostrar el resultado, ...). Para que webpack empaquete y transpile esos 2 ficheros (index.js y suma.js) ejecutaremos en la terminal:

```
npx webpack --mode=development
```

Enlazar el fichero generado en el HTML

Por último, en nuestro *index.html* debemos incluir sólo el *main.js* generado por webpack `<script src="dist/main.js"></script>`

Testear la UI

Si queremos hacer tests unitarios de los cambios que produce nuestro código en la página web hay varios frameworks que podemos usar, pero también podemos hacerlo sin usar ninguno, usando sólo los módulos de Node que ya tenemos instalados y *Jest*, en concreto su herramienta *jsdom* que usa para emular un navegador.

Para usarlo debemos instalar la librería Testing Library. Para ello, tras configurar nuestro proyecto con *Babel* como hemos visto antes, instalaremos para desarrollo los paquetes **@testing-library/dom** y **@testing-library/jest-dom**:

```
npm i -D @testing-library/dom @testing-library/jest-dom
```

En el fichero de test debemos poner al principio:

```
const { fireEvent, getByText } = require('@testing-library/dom')
import '@testing-library/jest-dom/extend-expect'
import { JSDOM } from 'jsdom'
import fs from 'fs'
import path from 'path'
```

y antes de ejecutar cada test cargamos nuestra página HTML con:

```
const html = fs.readFileSync(path.resolve(__dirname, '../index.html'), 'utf8');
```

OJO: sólo debemos cargar así la página si confiamos totalmente en el código que vamos a probar (en este caso es nuestro código) y no deberíamos hacerlo para código de terceros.

Luego ya podemos acceder al HTML y mirar si existen ciertas etiquetas o su contenido, como en este ejemplo:

```
const { fireEvent, getByText } = require('@testing-library/dom')
import '@testing-library/jest-dom/extend-expect'
import { JSDOM } from 'jsdom'
import fs from 'fs'
import path from 'path'

const html = fs.readFileSync(path.resolve(__dirname, '../index.html'), 'utf8');

let dom
let container

describe('index.html', () => {
  beforeEach(() => {
    // Constructing a new JSDOM with this option is the key
    // to getting the code in the script tag to execute.
    // This is indeed dangerous and should only be done with trusted content.
    // https://github.com/jsdom/jsdom#executing-scripts
    dom = new JSDOM(html, { runScripts: 'dangerously' })
    container = dom.window.document.body
  })

  it('renders a heading element', () => {
    expect(container.querySelector('h1')).not.toBeNull()
```

```
24      expect(getByText(container, 'Almacén central - ACME
    SL')).toBeInTheDocument()
25    })
26  })
```

Además de *getByText* para comprobar los elementos de la página tenemos otras queries que podemos utilizar.

Fuente: How to Unit Test HTML and Vanilla JavaScript Without a UI Framework

Usar jest

La documentación oficial proporciona muy buena información de cómo usarlo. En resumen, en los ficheros con las funciones que vayamos a testear debemos '*exportar*' esas funciones para que las pueda importar el fichero de test. Lo haremos con `module.exports`:

```
function suma(a, b) {
  return a + b;
}
module.exports = suma;
```

Si tenemos varias funciones podemos exportar un objeto con todas ellas:
`module.exports`:

```
function suma(a, b) {
  return a + b;
}
module.exports = { suma, resta, multiplica, divide };
```

En el fichero de test (que normalmente se llamará como el original más *test* antes de la extensión, por ejemplo *funciones.test.js*) importamos esas funciones con un `require`:

```
const suma = require('./funciones');
```

y ya podemos acceder llamar a la función 'suma' desde el fichero de test. Si queremos importar varias funciones haremos:

```
const funciones = require('./funciones');
```

y accederemos a cada una como 'funciones.suma', ...

Ya podemos crear nuestro primer test para probar la función suma:

```
test('Suma 1 + 1 devuelve 2', () => {
  expect(funciones.suma(1, 1)).toBe(2);
});
```

Para crear un test usamos la instrucción `test` (o `it`) a la que le pasamos como primer parámetro un nombre descriptivo de lo que hace y como segundo parámetro la función que realiza el test. En general usaremos `expect` y le pasamos como parámetro la llamada a la función a testear y comparamos el resultado devuelto usando un *matcher*.

Matchers

Los más comunes son:

- **toBe()**: compara el resultado del *expect* con lo que le pasamos como parámetro. Sólo sirve para valores primitivos (number, string, boolean, ...) no para arrays ni objetos
- **toBeCLoseTo()**: se usa para números de punto flotante. `expect(0.1 + 0.2).toBe(0.3)` fallaría por el error de redondeo
- **toEqual()**: como el anterior pero para objetos y arrays. Comprueba cada uno de los elementos el objeto o array
- **toBeLessThan**, **toBeLessThanOrEqual**, **toBeGreaterThan**, **toBeGreaterThanOrEqual**: para comparaciones <, <=, >, >=
- **toBeTruthy**: el valor devuelvo es verdadero o asimilable a verdadero (si fuera la condición de un *if* se ejecutaría el *then*)
- **toBeFalsy**: el valor devuelvo es falso o asimilable a falso (si fuera la condición de un *if* se ejecutaría el *else*)
- **toBeUndefined**: el valor es *undefined*
- **toBeDefined**: el valor NO es *undefined*
- **toBeNull**: el valor devuelto es *null*
- **toMatch**: el valor devuelto debe cumplir con la expresión regular pasada
- **toContain**: el array devuelto debe contener el elemento pasado como parámetro
- **toHaveLength**: el array o el string devueltos debe tener la longitud indicada

Para comprobar si una función ha lanzado una excepción se usa `toThrow`. Podemos comprobar sólo que haya lanzado un error, que sea de un tipo determinado, el mensaje exacto que tiene o si el mensaje cumple con una expresión regular:

```
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK');
}

test('compiling android goes as expected', () => {
  expect(compileAndroidCode).toThrow();
  expect(compileAndroidCode).toThrow(Error);
  expect(compileAndroidCode).toThrow('you are using the wrong JDK');
  expect(compileAndroidCode).toThrow(/JDK/);
});
```

Podemos obtener la lsita completa de *matchers* en al documentación oficial de Jest.

Test suites

En muchas ocasiones no vamos a pasar un único test sino un conjunto de ellos. En ese caso podemos agruparlos en un *test suite* que definimos con la instruacción `describe` a la que pasamos un nombre que la describa y una función que contiene todos los tests a pasar:

```
describe('Funciones aritméticas', () => {
  test('Suma 1 + 1 devuelve 2', () => {
    expect(funciones.suma(1, 1)).toBe(2);
  });

  test('Resta 2 - 1 devuelve 1', () => {
    expect(funciones.resta(2, 1)).toBe(1);
  });
});
```

Mocks

Muchas veces debemos testear partes del código que llaman a otras funciones pero no nos interesa que se ejecuten esas funciones sino simplemente saber si se han llamado o no y con qué parámetros. Para eso se definen las fuciones *mock*. Consiste en declarar en nuestro fichero de test una función a la que llama el código como función de *jest*.

Por ejemplo, tenemos un método de un controlador llamado *addProduct* que llama a otro de la vista llamado *renderProduct* para renderizar algo. Nosotros sólo queremos testear que se llama a la vista y que el parámetro que se le pasa es el adecuado. En nuestro test haremos:

```
renderProduct = jest.fn();

test('renderProduct called once with product {id: 1, name: "Prod 1", price: 49.99}', () => {
  const product = {id: 1, name: "Prod 1", price: 49.99};

  renderProduct(product);
  renderProduct({});

  expect(renderProduct.mock.calls.length).toBe(2);
  expect(renderProduct.mock.calls[0][0]).toEqual(newProd);
  expect(renderProduct.mock.calls[1][0]).toEqual({});
})
```

En realidad no se llama a la función real sino a la definida por el mock y podemos ver las veces que ha sido llamada (`fn.mock.calls.length`) o el primer parámetro pasado en la primera llamada (`fn.mock.calls[0][0]`) o en la segunda (`fn.mock.calls[1][0]`).

Podéis obtener toda la información en la documentación de jest.

También podemos encontrar muchos ejemplos en otras webs, como en adalab

Testear promesas

Para testear una función que devuelve una promesa debemos hacerlo de diferente manera. Por ejemplo tenemos una función 'getData' que devuelve una promesa. Para testearla:

```
test('getData devuelve un arrya de 3 elementos', () => {
  return getData().then(data => expect(data).toHaveLength(3) );
});
```

No olvidéis poner el 'return', si no el test acabará sin esperar a que se resuelva la promesa. Si lo que queremos es comprobar que la promesa es rechazada haremos:

```
test('getData devuelve un arrya de 3 elementos', () => {
  expect.assertions(1);
  return getData().catch(err => expect(err).toMatch('404');
  });
});
```

En este caso esperamos que devuelva un error que contenga '404'. Hay que poner la línea de `expect.assertions` para evitar que una promesa cumplida no haga que falle el test.

En la documentación oficial de Jest podemos encontrar información de cómo probar todo tipo de llamadas asíncronas (*callback*, *async/await*, ...).

Hooks de Jest

Permiten ejecutar código antes o depués de pasar cada test o el conjunto de ellos. Son:

- **afterEach()**: Después de cada prueba.
- **afterAll()**: Después de todas las pruebas.
- **beforeEach()**: Antes de cada prueba.
- **beforeAll()**: Antes de todas las pruebas.

Por ejemplo podemos querer inicializar la base de datos antes de pasar cada test:

```
beforeAll(() => {
  initializeCityDatabase();
});
```

Si se trata de una función asíncrona habrá que añadirle un 'return' igual que hacíamos con las promesas:

```
beforeAll(() => {
  return initializeCityDatabase();
});

afterAll(() => {
  return clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});`
```

Desarrollo guiado por pruebas (TDD)

Es una forma de programar que consiste en escribir primero las pruebas que deba pasar el código (Test Dirve Development) y luego el código que las pase. Por último deberíamos refactorizarlo (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés).

El ciclo de programación usando TDD tiene tres fases:

1. Fase *roja*: escribimos el test que cumpla los requerimientos y lo pasamos. Fallará ya que nuestro código no pasa el test (de hecho la primera vez no tenemos ni código)
2. Fase *verde*: conseguimos que nuestro código pase el test. Ya funciona aunque seguramente no estará muy bien escrito
3. *Refactorización*: mejoramos nuestro código

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.

Para ello debemos en primer lugar se debe definir una lista de requisitos y después se ejecuta el siguiente ciclo:

1. Elegir un requisito: Se elige de una lista el requisito que se cree que nos dará mayor conocimiento del problema y que a la vez sea fácilmente implementable.
2. Escribir una prueba: Se comienza escribiendo una prueba para el requisito. Para ello el programador debe entender claramente las especificaciones y los requisitos de la funcionalidad que está por implementar. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces.
3. Verificar que la prueba falla: Si la prueba no falla es porque el requisito ya estaba implementado o porque la prueba es errónea.
4. Escribir la implementación: Escribir el código más sencillo que haga que la prueba funcione. Se usa la expresión "Déjelo simple" ("Keep It Simple, Stupid!"), conocida como principio KISS.
5. Ejecutar las pruebas automatizadas: Verificar si todo el conjunto de pruebas funciona correctamente.
6. Eliminación de duplicación: El paso final es la refactorización, que se utilizará principalmente para eliminar código duplicado. Se hace un pequeño cambio cada vez y luego se corren las pruebas hasta que funcionen.
7. Actualización de la lista de requisitos: Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requisitos de diseño (P. ej que una funcionalidad esté desacoplada de otra).

Tener un único repositorio universal de pruebas facilita complementar TDD con otra práctica recomendada por los procesos ágiles de desarrollo, la "Integración Continua". Integrar continuamente nuestro trabajo con el del resto del equipo de desarrollo permite ejecutar toda batería de pruebas y así descubrir si nuestra última versión es compatible con el resto del sistema. Es recomendable y menos

costoso corregir pequeños problemas cada pocas horas que enfrentarse a problemas enormes cerca de la fecha de entrega fijada.

(Fuente Wikipedia).