D. TESTING

1 PREPARANDO EL ENTORNO DE TESTING

Para poder implementar testing en nuestra aplicación y poder, en mayor o menor medida, trabajar con metodologías TDD, vamos a instalar la dependencia Jest, la suite de testings desarrollada por los ingenieros de Facebook.

En este punto tenemos dos opciones:

- Abrir y usar una nueva consola para instalar los nuevos paquetes.
- Terminar la ejecución, instalar y volver a arrancar el script de desarrollo.

Elegid una de las dos opciones e introducid este comando en el terminal:

npm i -D jest

```
c:\node1>npm i -D jest

added 346 packages, and audited 1154 packages in 18s

104 packages are looking for funding
  run `npm fund` for details

56 vulnerabilities (44 moderate, 12 high)

To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.
```

Veremos que nuestro fichero `package.json` contiene una nueva dependencia de desarrollo:

```
"devDependencies": {
  "jest": "^29.7.0",
```

```
"scripts": {
  "dev": "parcel src/index.html --open",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Si ejecutamos nuestro script de `test` que venía por defecto al instalar la aplicación mediante:

npm run test

Obtendremos un mensaje de error puesto que es lo que queda especificado en el fichero con `echo \"Error: no test specified\" &&exit 1`.

```
c:\node1>npm run test

> node1@1.0.0 test
> echo "Error: no test specified" && exit 1

"Error: no test specified"
```

Modificaremos su valor para llamar al binario de la librería de testing:

```
"scripts": {
  "dev": "parcel src/index.html --open",
  "test": "jest --passWithNoTests --silent"
},
```

Nota: El flag `--passWithNoTests` evita que se muestre un mensaje de error por consola en caso de que no haya ningún test implementado. El flag `--silent` evita que se muestren errores adicionales no relacionados con los test fallidos.

Ejecutamos este comando para ver la librería de testing en funcionamiento:

```
c:\node1>npm run test

> node1@1.0.0 test
> jest --passWithNoTests --silent

No tests found, exiting with code 0
```

1.1 JEST Y TIPOS DE TEST

Jest es una librería de **testing unitario**, que son los que nos permiten someter una función o una parte de nuestro código a unas entradas establecidas por nosotros y valorar su resultado de modo que procesen los datos como nosotros esperamos. Esto nos da consistencia en el código desde que creamos la función (unidad) y durante todo el proceso de desarrollo mientras aumentamos la base de código sin tener que manualmente comprobar que el comportamiento se mantiene.

El siguiente tipo son los **test de integración**, que nos permiten valorar cómo se comporta un servicio contra otro. Esto podría ser también a nivel de microservicios en el servidor o bien partes diferentes de la aplicación. Se puede hacer con Jest pero valorando que trabajaremos no de manera exclusiva con nuestra función desarrollada, sino contra otras partes del código o servicios de terceros.

Por último, tenemos los **E2E (end to end testing**), que son los que tradicionalmente han realizado los ingenieros de QA (quality assurance) de forma manual. En cada nueva iteración del código debían realizar de modo manual una serie de pasos y procesos desde el navegador como si fueran usuarios pasando por los test cases definidos para someter a prueba la aplicación. Este proceso manual puede automatizarse con herramientas como Selenium y Cypress. Estas librerías y frameworks han dado paso a una nueva categoría profesional que son ingenieros de automation QA.

Este último tipo de test son los que deberíamos implementar para poder valorar de forma automatizada el comportamiento de nuestra aplicación. Sin embargo, puesto que su implementación requiere conocer el uso de esos frameworks, la carga adicional para esta asignatura sería excesiva.

1.2 PRIMEROS PASOS CON TDD

Test Driven Development es una práctica de desarrollo en la que se definen en forma de test qué funcionalidades debe pasar un programa (o una parte de este) para ser válido, y posteriormente se implementa el código que cumple ese requisito.

Vamos a ver un ejemplo sencillo. Debemos crear un fichero `sum.js` y otro `sum.spec.js`.

Nota: Es la parte `spec` del nombre la que define que se trata de un archivo de test. **Jest** recogerá todos y cada uno de los ficheros `spec`, pasará los test uno por uno y mostrará el resumen por consola.

En el fichero `sum` vamos a añadir la siguiente función:

eje.js >

```
function sum(a, b) {
    return a + b;
    }

module.exports = sum;
```

En el fichero `spec` vamos a añadir el siguiente test:

eje.spec.js >

```
import sum from './sum';

test('suma 1 + 2 igual a 3', () => {
expect(sum(1, 2)).toBe(4);
});
```

Si ahora ejecutamos el comando de test, veremos que se salta un error. Según la filosofía TDD, para que un test sea válido primero hemos de asegurarnos de que falla en un caso incorrecto para luego corregirlo, ver qué pasa con éxito y seguir iterando.

```
c:\node1>npm run test

> node1@1.0.0 test
> jest --passWithNoTests --silent

 FAIL  src/eje.spec.js
  ✕ suma 1 + 2 igual a 3

    expect(received).toBe(expected) // Object.is equality

    Expected: 4
    Received: 3

      2 |
      3 | test('suma 1 + 2 igual a 3', () => {
    > 4 | expect(sum(1, 2)).toBe(4);
        |                   ^
      5 | });

      at Object.toBe (src/eje.spec.js:4:19)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.529 s
```

Modificamos el contenido del test por su valor correcto:

eje.spec.js >

```
const sum = require('./eje');

test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3);
    });
```

Ahora, al ejecutar el comando de `test`, veremos que los resultados son correctos.

```
c:\node1>npm run test

> node1@1.0.0 test
> jest --passWithNoTests --silent

 PASS  src/eje.spec.js

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.478 s, estimated 1 s
```

1.3 MÓDULOS ES6

Lo que pretendemos en este punto es habilitar la posibilidad de usar módulos y que en los ficheros de test podamos usar la nomenclatura de módulos `ES6` (`import/export`) en lugar de los `ES5/CommonJS` (`module.exports/require`).

`Parcel` nos permite usarlos a nivel de nuestra aplicación, pero no para los test, ya que no son parte directa de esta (los test no van en un bundle de producción); Jest es quien trabaja los ficheros de test y por ello debemos instalar un nuevo paquete para poder habilitar esta funcionalidad para nuestros test.

npm i -D babel-jest @babel/core @babel/preset-env

```
c:\node1>npm i -D babel-jest @babel/core @babel/preset-env

up to date, audited 1154 packages in 5s

104 packages are looking for funding
  run `npm fund` for details

56 vulnerabilities (44 moderate, 12 high)

To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.
```

Nuestro fichero `package.json` tendrá nuevas entradas en su apartado de dependencias de desarrollo:

```
"devDependencies": {
  "@babel/core": "^7.23.5",
  "@babel/preset-env": "^7.23.5",
  "babel-jest": "^29.7.0",
  "jest": "^29.7.0",
  "parcel-bundler": "^1.12.5"
}
```

Babel es un transpilador entre versiones de ECMAScript (la especificación de JavaScript). Nos permite escribir ES6 y automáticamente hará una conversión a ES5 para que node.js pueda ejecutarlo como si fuera el mismo código (node.js no soporta aún de forma nativa los módulos ES6).

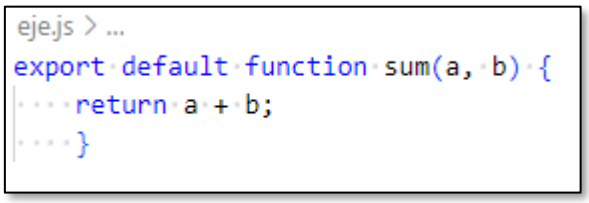
Con estos paquetes ya tenemos instaladas las funcionalidades para poder usar la sintaxis de módulos en nuestros test de aplicación. Sin embargo, necesitamos indicar mediante un archivo adicional de configuración cómo vamos a hacer esa conversión.

Debemos crear un nuevo fichero `babel.config.js` en la raíz del proyecto (no en /src) con este contenido:

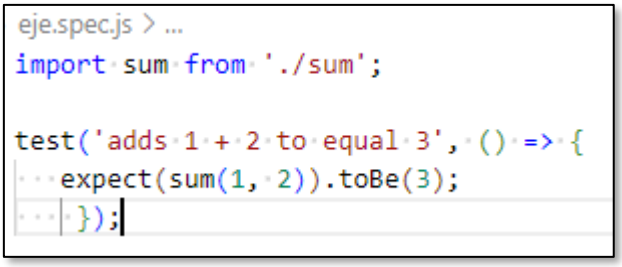
babel.config.js >

```
module.exports = {
    presets: [
        [
            '@babel/preset-env',
            {
                targets: {
                    node: 'current'
                }
            }
        ]
    ]
};
```

Ahora cambiaremos el formato de nuestro fichero `eje.js` por:



Y el de `eje.spec.js` por:



Procederemos a ejecutar de nuevo los test y veremos cómo, a pesar del cambio de sintaxis, no se nos presenta ningún de error de ejecución, ya que hemos habilitado el uso de las nuevas funcionalidades en nuestros test como lo hace `parcel` en nuestros ficheros de aplicación.