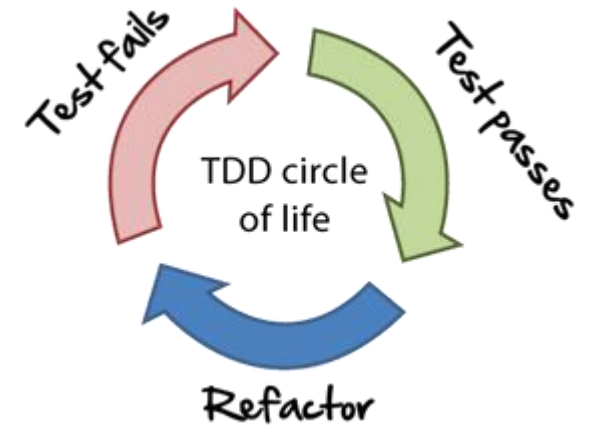




1º DAM/DAW EDE

U6. Diseño de pruebas de software

9 - Anexo - Jest



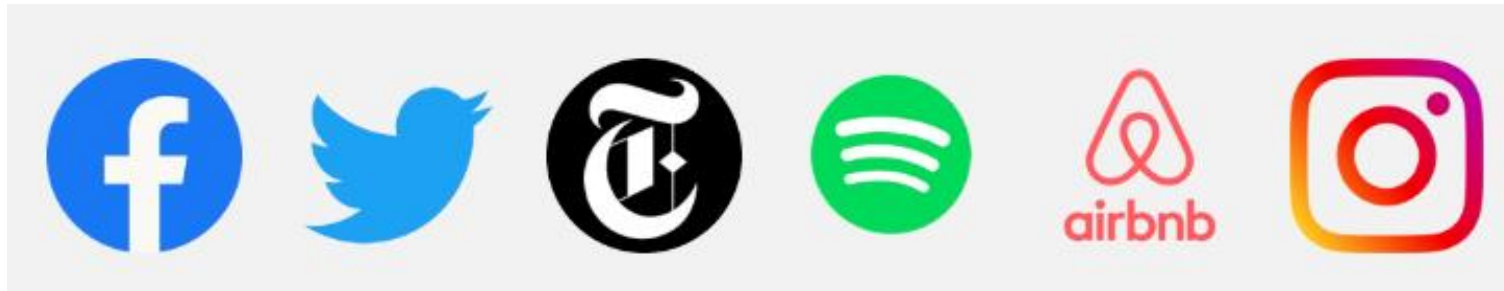
¿Qué es Jest?

- En el contexto de la automatización de pruebas, **Jest es un marco de trabajo (framework) de código abierto centrado en la simplicidad.**
- Es una **biblioteca de JavaScript para crear, ejecutar, automatizar y estructurar pruebas.**



¿Qué es Jest?

- Página oficial: <https://jestjs.io/>
- GitHub: <https://github.com/jestjs/jest>
- Utilizado por las siguientes compañías:

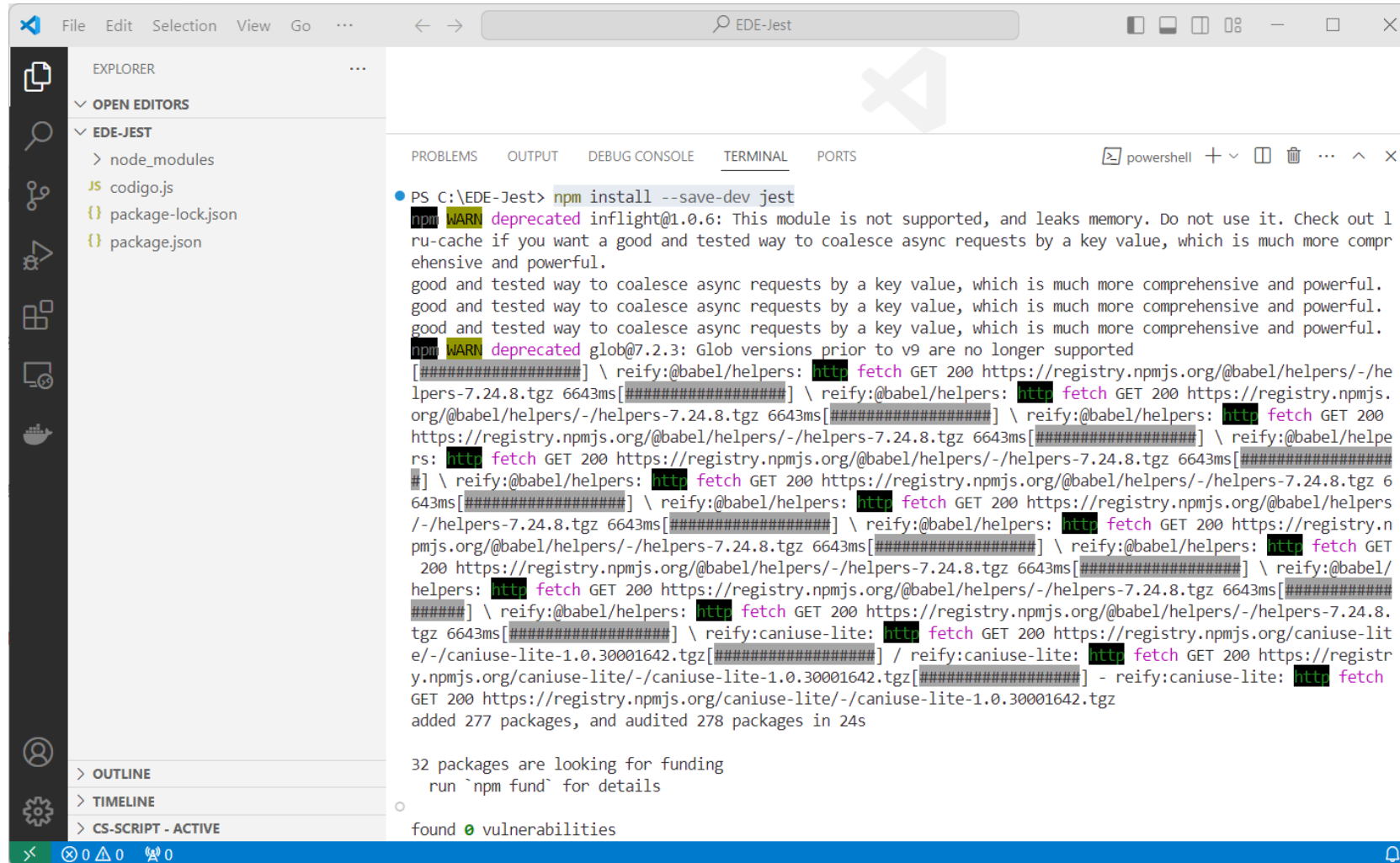


Instalación de Jest

- Jest se distribuye como un **paquete NPM** que puede instalarse en cualquier proyecto de JavaScript bajo entorno de ejecución Node.js
- Podemos abrir la carpeta de nuestro proyecto en Visual Studio Code y lanzar el comando de instalación desde la ventana del terminal:

npm install --save-dev jest

Instalación de Jest



The screenshot shows the Visual Studio Code interface with a terminal window open. The Explorer sidebar on the left shows a project named 'EDE-JEST' with files 'node_modules', 'codigo.js', 'package-lock.json', and 'package.json'. The terminal window displays the command 'npm install --save-dev jest' and its output, which includes several deprecation warnings and progress logs for installing dependencies like 'inflight', 'glob', and 'babel-helpers'.

```
PS C:\EDE-Jest> npm install --save-dev jest
npm WARN deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out l
ru-cache if you want a good and tested way to coalesce async requests by a key value, which is much more compr
ehensive and powerful.
good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
[#####] \ reify:@babel/helpers: http fetch GET 200 https://registry.npmjs.org/@babel/helpers/-/he
lpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/helpers: http fetch GET 200 https://registry.npmjs.
org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/helpers: http fetch GET 200
https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/helpe
rs: http fetch GET 200 https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####]
# \ reify:@babel/helpers: http fetch GET 200 https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6
643ms[#####] \ reify:@babel/helpers: http fetch GET 200 https://registry.npmjs.org/@babel/helpers
/-/helpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/helpers: http fetch GET 200 https://registry.n
pmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/helpers: http fetch GET
200 https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####] \ reify:@babel/
helpers: http fetch GET 200 https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.tgz 6643ms[#####]
##### \ reify:@babel/helpers: http fetch GET 200 https://registry.npmjs.org/@babel/helpers/-/helpers-7.24.8.
tgz 6643ms[#####] \ reify:caniuse-lite: http fetch GET 200 https://registry.npmjs.org/caniuse-lit
e/-/caniuse-lite-1.0.30001642.tgz[#####] / reify:caniuse-lite: http fetch GET 200 https://registr
y.npmjs.org/caniuse-lite/-/caniuse-lite-1.0.30001642.tgz[#####] - reify:caniuse-lite: http fetch
GET 200 https://registry.npmjs.org/caniuse-lite/-/caniuse-lite-1.0.30001642.tgz
added 277 packages, and audited 278 packages in 24s

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

found 0 vulnerabilities
```

Configuración inicial de Jest

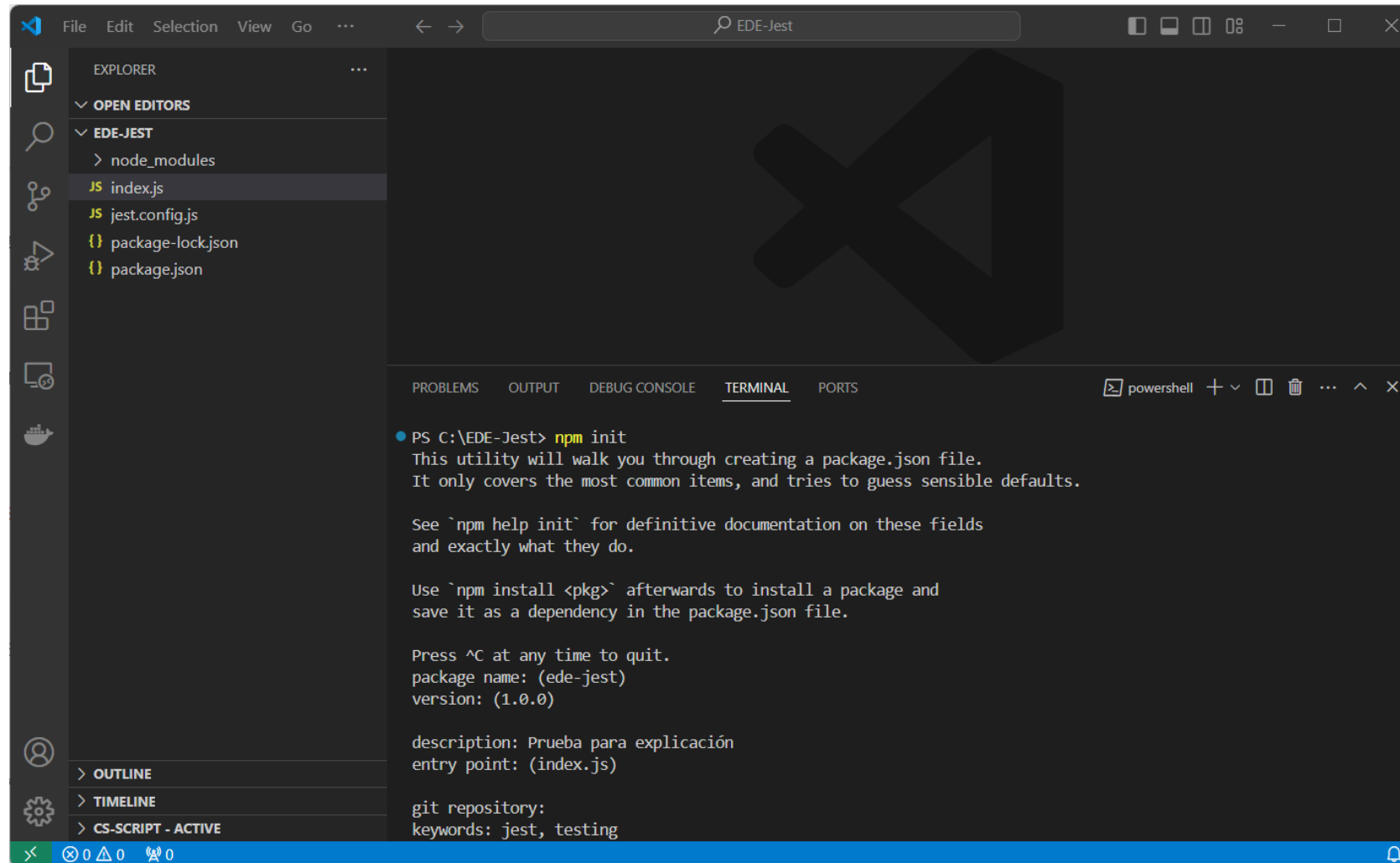
- Ejecutando los siguientes comandos desde la ventana del terminal podemos establecer la configuración inicial del proyecto y de Jest:

npm init

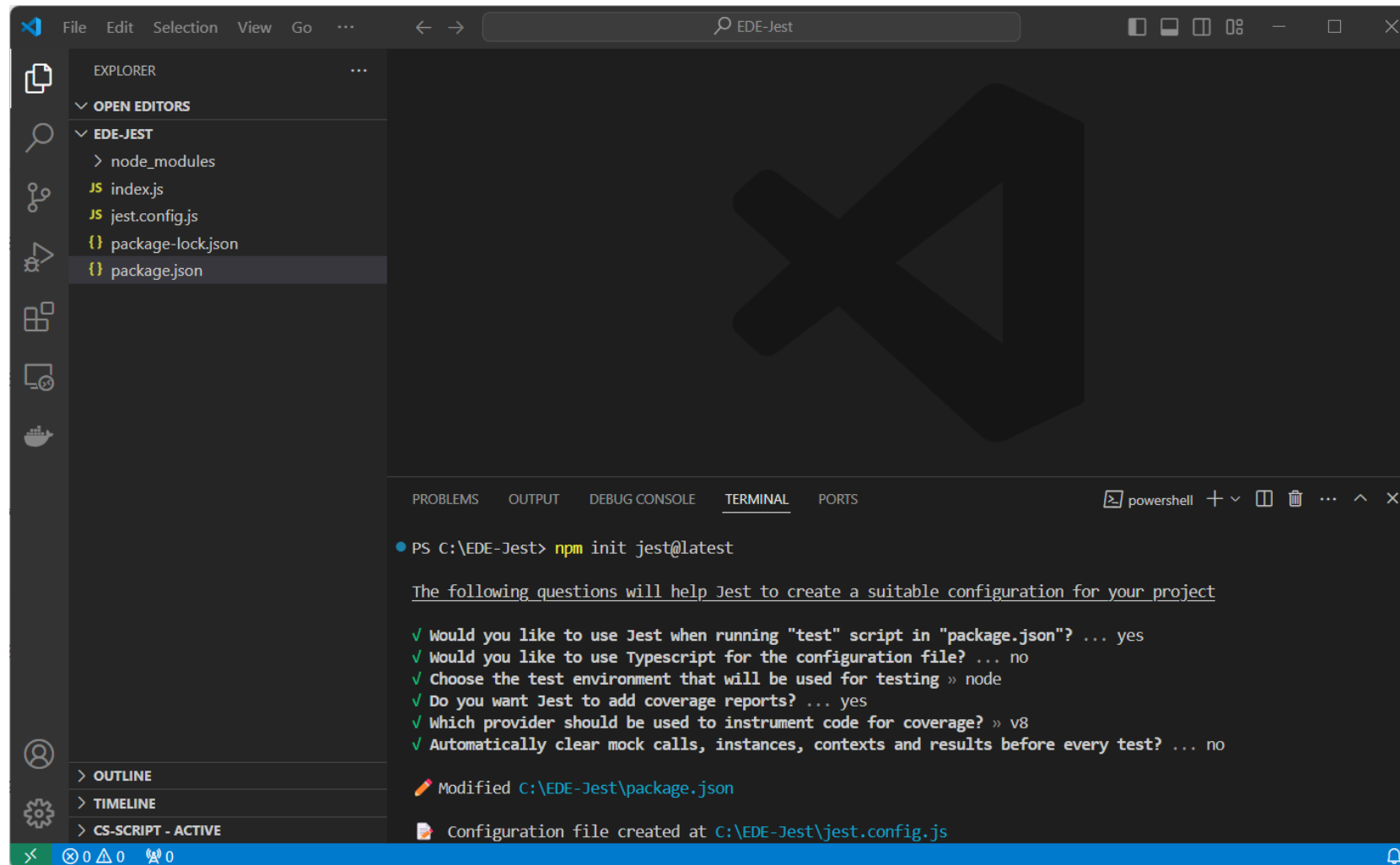
npm init jest@latest

- Jest nos hará una serie de preguntas y cumplimentará los ficheros **package.json** y **jest.config.js** con nuestras respuestas.

Configuración inicial



Configuración inicial

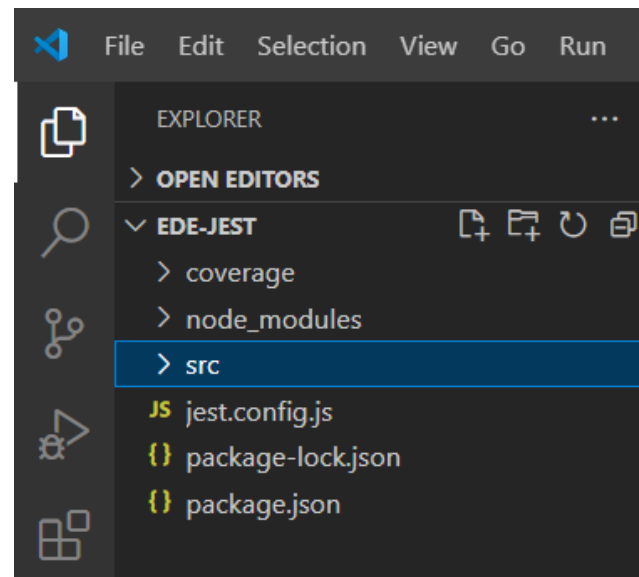


Pasos de desarrollo y pruebas unitarias

- Una vez instalado Jest, como nuestro marco de trabajo para pruebas (testing framework), los pasos a seguir a nivel de desarrollo serían los siguientes:
 - **Desarrollar un fragmento de código, función o método** de nuestro proyecto, como respuesta a la especificación de un requerimiento funcional.
 - Para cada fragmento de código, podemos **diseñar uno o varios casos de prueba**, que nos servirán para confirmar el adecuado funcionamiento del código.
 - Desarrollar un script de **prueba unitaria**.
 - **Ejecutarlo y confrontar los resultados** con lo esperado (este paso lo hace Jest...).

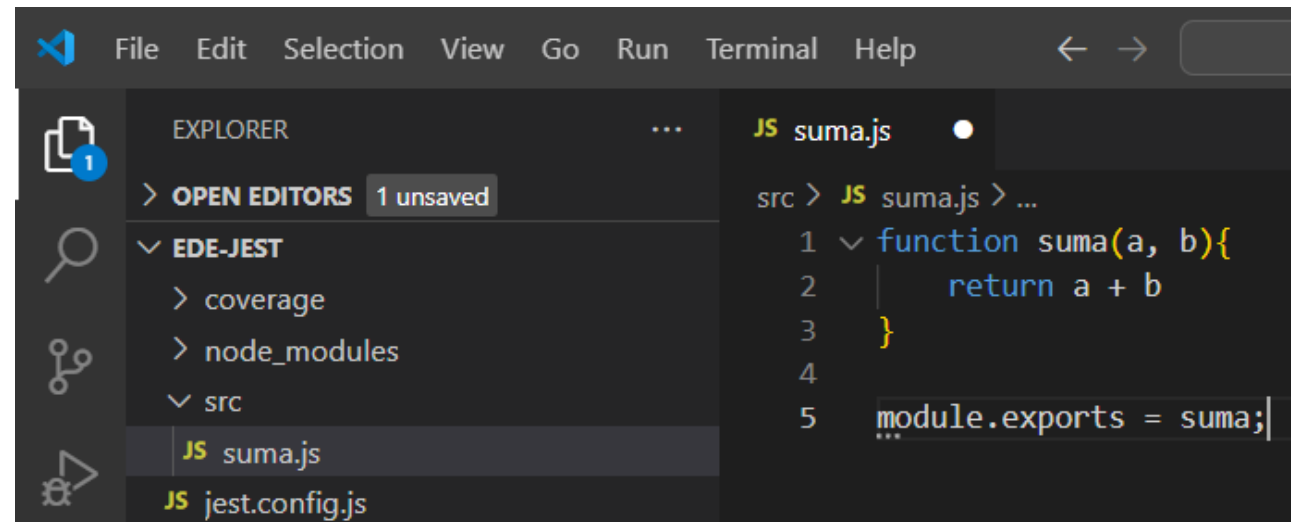
Desarrollo de la funcionalidad del proyecto

- Podemos crear una carpeta donde almacenar los ficheros con la funcionalidad del proyecto, únicamente por cuestiones organizativas. Se puede denominar, por ejemplo: **src**



Desarrollo de la funcionalidad del proyecto

- Dentro de ella podemos crear nuestro primer fichero de funcionalidad, que contendrá el primer **fragmento de código**, por ejemplo: **suma.js**

A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows a project structure with folders 'coverage' and 'node_modules', and a 'src' folder containing 'suma.js' and 'jest.config.js'. The 'suma.js' file is selected and open in the main editor. The code in the editor shows a function definition: 'function suma(a, b){ return a + b; }' followed by 'module.exports = suma;'. The file name 'suma.js' is visible in the tab at the top of the editor.

```
src > JS suma.js > ...  
1  function suma(a, b){  
2      return a + b  
3  }  
4  
5  module.exports = suma;
```

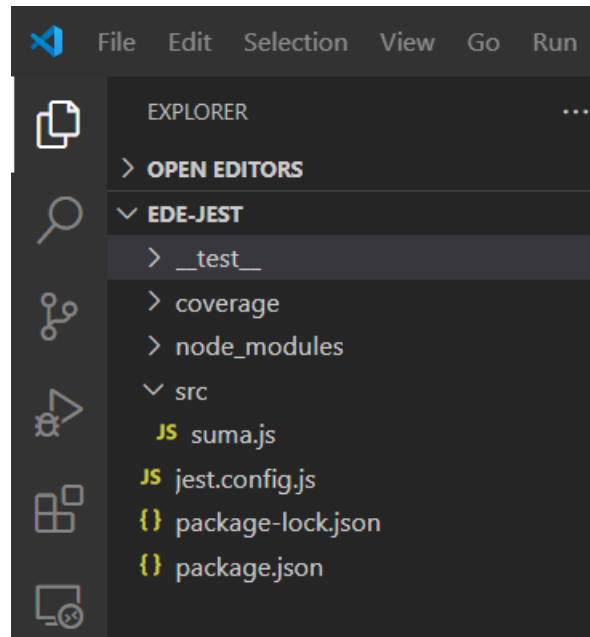
- La expresión “***module.exports = suma***” permite exportar la función “***suma(a, b)***” para recogerla y poderla invocar desde el fichero de prueba unitaria.

Diseño del caso de prueba

- Dado el **fragmento de código**, podemos diseñar varios casos de prueba. Como siempre empezaremos con el más sencillo:
 - **Identificador:** 1
 - **Nombre:** sumar números enteros positivos.
 - **Descripción:** se le pasarán dos parámetros de entrada en la llamada a la función que realiza la suma.
 - **Precondición:** los valores de los parámetros serán 3 y 2.
 - **Pasos:** llamada a la función correspondiente.
 - **Postcondición:** la función debe devolver el valor 5 para resultar una prueba satisfactoria, en caso contrario resultará una prueba fallida.
 - **Informe:** a realizar una vez finalizada la prueba unitaria.

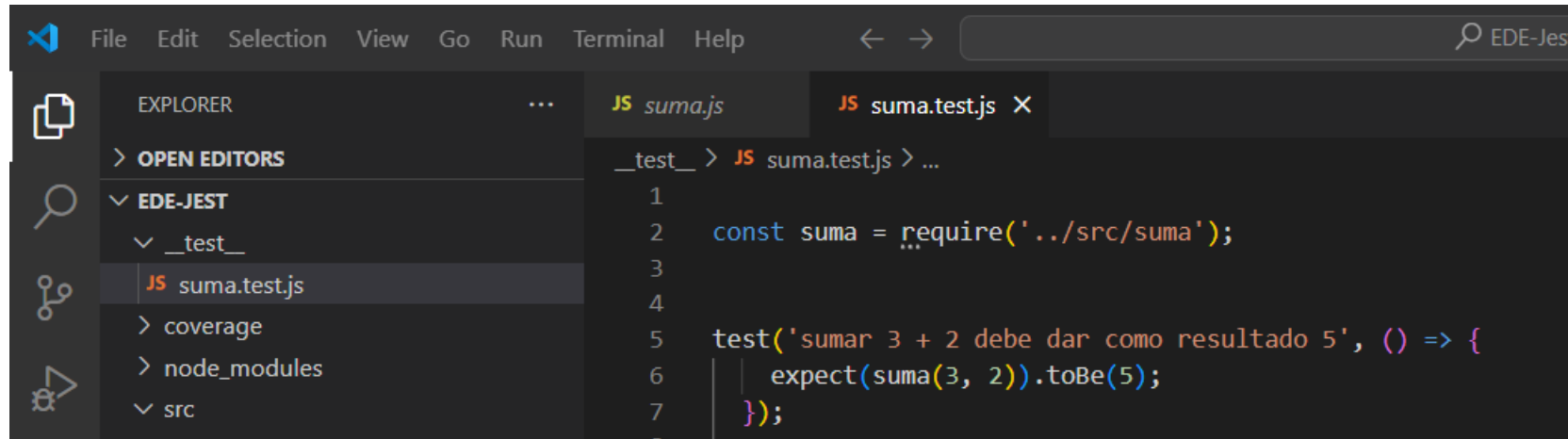
Desarrollo de la prueba unitaria

- Crearemos una carpeta en nuestro proyecto donde almacenaremos los scripts de pruebas unitarias, denominada **por convención: __test__**



Desarrollo de la prueba unitaria

- Dentro de ella podemos crear nuestro primer fichero de **prueba unitaria**. La denominación debe tener la siguiente estructura **por convención**: **suma.test.js**



```
1
2 const suma = require('../src/suma');
3
4
5 test('sumar 3 + 2 debe dar como resultado 5', () => {
6   expect(suma(3, 2)).toBe(5);
7 });
```

- La expresión ***“const suma = require('../src/suma');”*** permite importar lo exportado previamente en el fichero ***“suma”*** y asignarlo a la constante ***“suma”***.

Ejecución de la prueba unitaria

- Ejecutamos el siguiente comando desde la ventana del terminal: ***npm test***

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

• PS C:\EDE-Jest> npm test

> ede-jest@1.0.0 test
> jest

PASS  __test__/suma.test.js
  ✓ sumar 3 + 2 debe dar como resultado 5 (8 ms)

-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100 |    100   |    100   |    100   |
  suma.js |    100 |    100   |    100   |    100   |
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.284 s
Ran all test suites.
PS C:\EDE-Jest> 
```

Informe de la prueba unitaria

- El **resultado** de la ejecución **coincide con lo esperado**, con lo que la prueba ha resultado ser **satisfactoria**.



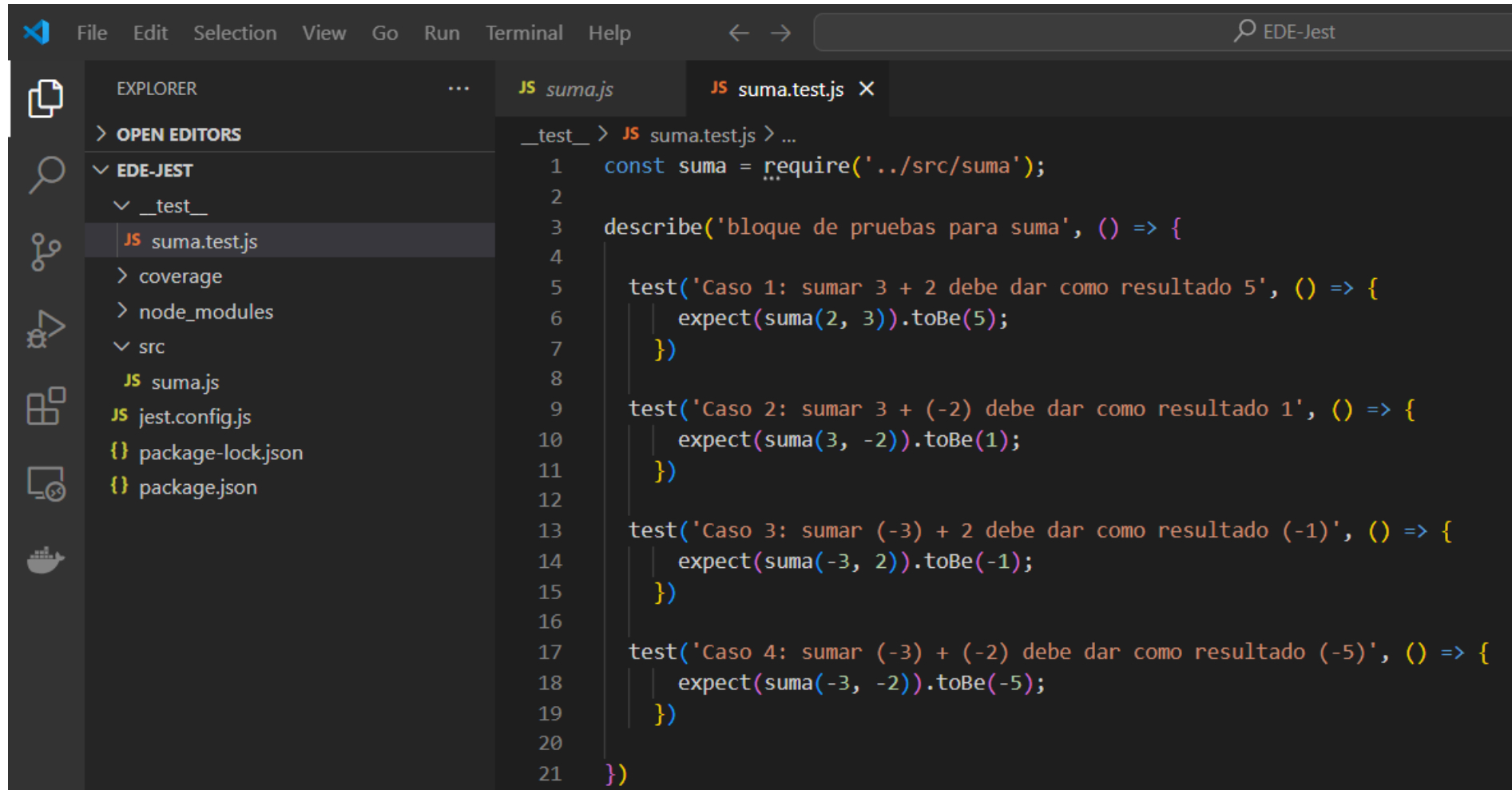
- *Es el momento de reflexionar... ¿si las pruebas son satisfactorias es completamente seguro que el código es totalmente correcto y que su ejecución nunca fallará?*



Desarrollo de la prueba unitaria

- Podemos diseñar y desarrollar **varios casos de pruebas unitarias para testear un fragmento de código o función** en un mismo fichero de pruebas.
- En nuestro sencillo ejemplo, hemos desarrollado el caso de prueba 1, en el que sumamos 2 números enteros positivos y obtenemos como resultado otro entero positivo. Podemos incrementar los casos de prueba:
 - **Caso 2:** sumamos 1 entero positivo y otro negativo. El resultado debe ser un entero positivo.
 - **Caso 3:** sumamos 1 entero positivo y otro negativo. El resultado debe ser un entero negativo.
 - **Caso 4:** sumamos 2 enteros positivo y otro negativo. El resultado debe ser un entero positivo.

Desarrollo de la prueba unitaria



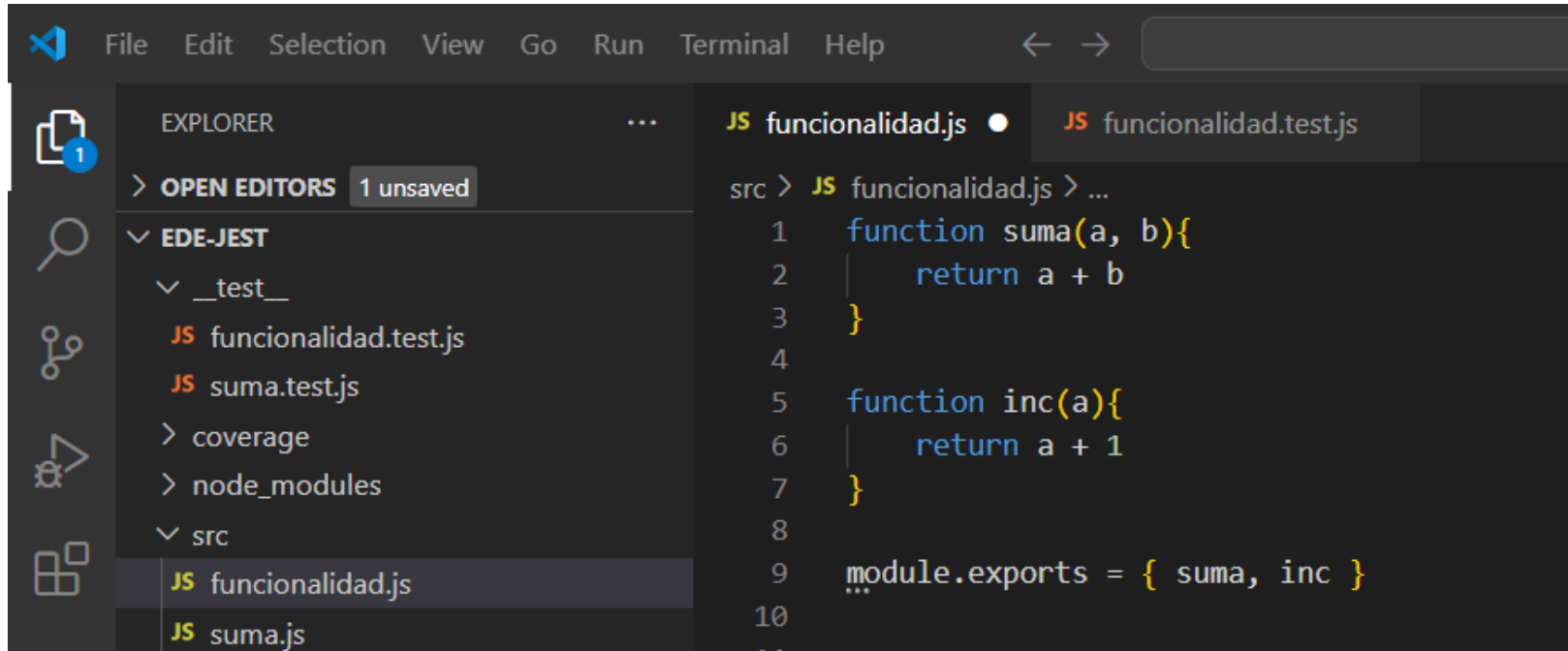
The screenshot shows the VS Code editor interface. The Explorer sidebar on the left displays the project structure with folders like 'coverage', 'node_modules', and 'src'. The 'src' folder is expanded, showing 'suma.js', 'jest.config.js', 'package-lock.json', and 'package.json'. The 'EDE-JEST' folder is also visible. The main editor area shows the 'suma.test.js' file with the following code:

```
__test__ > JS suma.test.js > ...
1  const suma = require('../src/suma');
2
3  describe('bloque de pruebas para suma', () => {
4
5      test('Caso 1: sumar 3 + 2 debe dar como resultado 5', () => {
6          expect(suma(2, 3)).toBe(5);
7      })
8
9      test('Caso 2: sumar 3 + (-2) debe dar como resultado 1', () => {
10         expect(suma(3, -2)).toBe(1);
11     })
12
13     test('Caso 3: sumar (-3) + 2 debe dar como resultado (-1)', () => {
14         expect(suma(-3, 2)).toBe(-1);
15     })
16
17     test('Caso 4: sumar (-3) + (-2) debe dar como resultado (-5)', () => {
18         expect(suma(-3, -2)).toBe(-5);
19     })
20
21 })
```

Desarrollo de la prueba unitaria

- Si un fichero con funcionalidad del proyecto consta de varias funciones, también podemos diseñar y desarrollar **varios casos de pruebas unitarias para testear diferentes funciones** en un mismo fichero de pruebas.
- Ampliando un poco nuestro sencillo ejemplo, podemos generar un nuevo fichero de funcionalidad del proyecto con varias funciones. Por ejemplo: **funcionalidad.js**

Desarrollo de la prueba unitaria

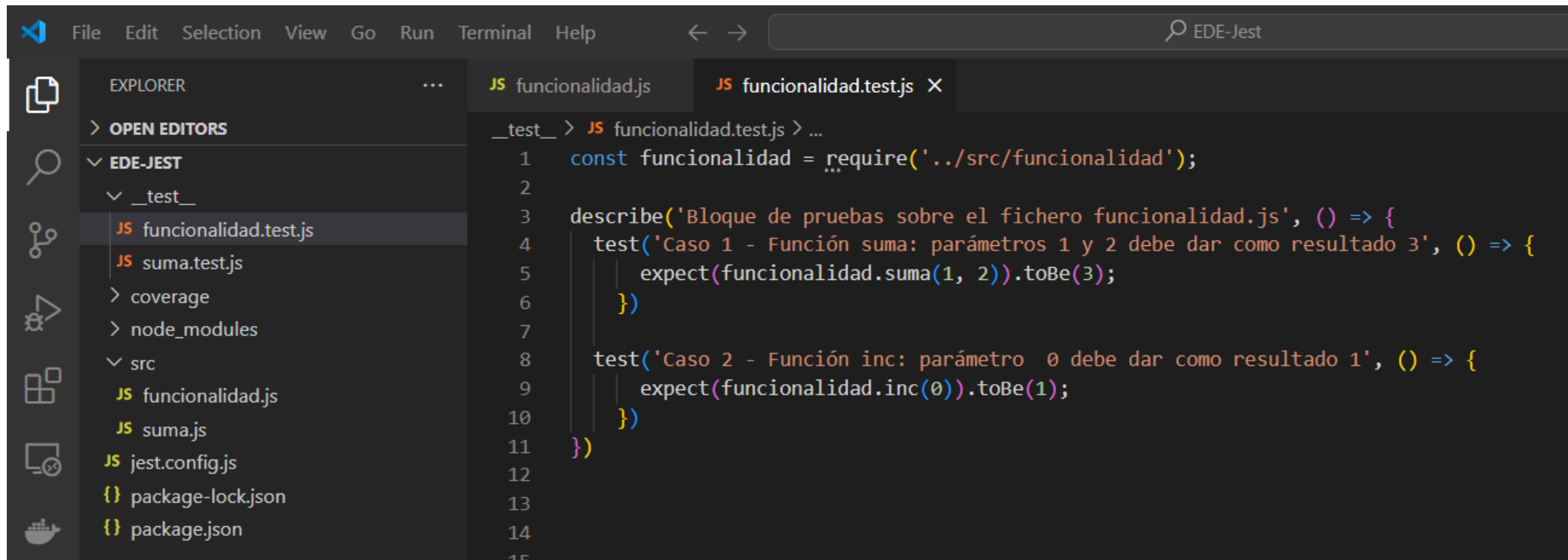


The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays a project structure with folders like 'EDE-JEST', '__test__', 'coverage', 'node_modules', and 'src'. The 'src' folder is expanded, showing 'funcionalidad.js' and 'suma.js'. The 'funcionalidad.test.js' file is selected in the 'EDE-JEST' folder. The main editor area shows the content of 'funcionalidad.js', which defines two functions: 'suma(a, b)' and 'inc(a)', and exports them as 'module.exports = { suma, inc }'.

```
src > JS funcionalidad.js > ...
1  function suma(a, b){
2      |   return a + b
3  }
4
5  function inc(a){
6      |   return a + 1
7  }
8
9  module.exports = { suma, inc }
10
11
```

- La expresión ***“module.exports = { suma, inc }”*** permite exportar las funciones ***“suma(a, b)”*** y ***“inc(a)”*** para recogerlas y poderlas invocar desde el fichero de prueba unitaria.

Desarrollo de la prueba unitaria



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar is open, showing a project structure with a folder named `_test_` containing `funcionalidad.test.js` and `suma.test.js`. The main editor area displays the content of `funcionalidad.test.js`. The code is as follows:

```
1 const funcionalidad = require('../src/funcionalidad');
2
3 describe('Bloque de pruebas sobre el fichero funcionalidad.js', () => {
4   test('Caso 1 - Función suma: parámetros 1 y 2 debe dar como resultado 3', () => {
5     expect(funcionalidad.suma(1, 2)).toBe(3);
6   })
7
8   test('Caso 2 - Función inc: parámetro 0 debe dar como resultado 1', () => {
9     expect(funcionalidad.inc(0)).toBe(1);
10  })
11 })
```

- La expresión ***“const funcionalidad = require('../src/funcionalidad');”*** permite importar lo exportado previamente en el fichero ***“funcionalidad”*** y asignarlo a la constante ***“funcionalidad”***.

Ejecución de la prueba unitaria

- Al ejecutar el comando para lanzar las pruebas, **se lanzarán todas las del proyecto.**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● PS C:\EDE-Jest> npm test

> ede-jest@1.0.0 test
> jest

PASS  __test__/suma.test.js
PASS  __test__/funcionalidad.test.js

-----|-----|-----|-----|-----|-----|
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----|
All files    |    100 |    100 |    100 |    100 |
funcionalidad.js |    100 |    100 |    100 |    100 |
suma.js     |    100 |    100 |    100 |    100 |
-----|-----|-----|-----|-----|

Test Suites: 2 passed, 2 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        0.922 s, estimated 1 s
Ran all test suites.
○ PS C:\EDE-Jest> 
```

Informe de la prueba unitaria

- Los **resultados** de la ejecución de todas las pruebas **coinciden con lo esperado**, con lo que **la fase de pruebas** ha resultado ser **satisfactoria**.



- *Sólo con que una de las pruebas hubiera sido fallida, el resultado global de la fase de pruebas habría sido fallido.*