



Certified Tech Developer

The Ultimate Degree

Testing I

Cobertura de pruebas con Jest

El test coverage es un valor utilizado como indicador para obtener visibilidad sobre la “robustez” en un proyecto, pero no es garantía de infalibilidad. Recordemos que todo depende de qué estamos testeando y cómo. Es posible tener un 100% de cobertura, pero que nuestro código tenga bugs si lo estamos testeando mal.

¿Qué porcentaje de test coverage debemos tener en un proyecto? Es lógico pensar que se debería apuntar a un 100% de cobertura del código, pero en la realidad no suele suceder de esta forma. ¿Por qué? Cada proyecto suele tener partes de mayor y menor importancia. Garantizar un 100% de cobertura de código puede implicar dedicarle tiempo a testear partes del código repetitivas o no vitales y, posiblemente, descuidar otras que necesiten más atención. Los equipos de desarrollo suelen enfocar sus esfuerzos en las partes centrales de un proyecto y garantizar que ahí haya un porcentaje de cobertura mayor o total.

A su vez, hay varias maneras y convenciones de manejar los porcentajes de cobertura de código. Si un proyecto ya comenzó sin tests y se decide agregarlos durante el transcurso de este, se suelen poner objetivos de cobertura de código e incrementarlos entre cada sprint. Se puede comenzar con buscar alcanzar un 50% de cobertura y luego subir un 5% entre cada sprint, por ejemplo.

Otra alternativa es no bajar de cierto porcentaje. Supongamos que estamos en un proyecto en el cual ya hay un 85% de cobertura de código y el equipo se propone no bajar de un 75%, como mínimo. En un caso así, es posible elegir cuándo y dónde priorizar la resolución de una feature para poder llegar a los objetivos del sprint frente a escribir todos los tests necesarios. Muchas veces se suele diferir la escritura de estos para poder cumplir con los objetivos generales del proyecto a tiempo.

Como toda convención, hay muchísimas variantes y dependerá de la empresa, el equipo y el proyecto en el que trabajes, la metodología que se implemente y cómo la lleven a cabo.

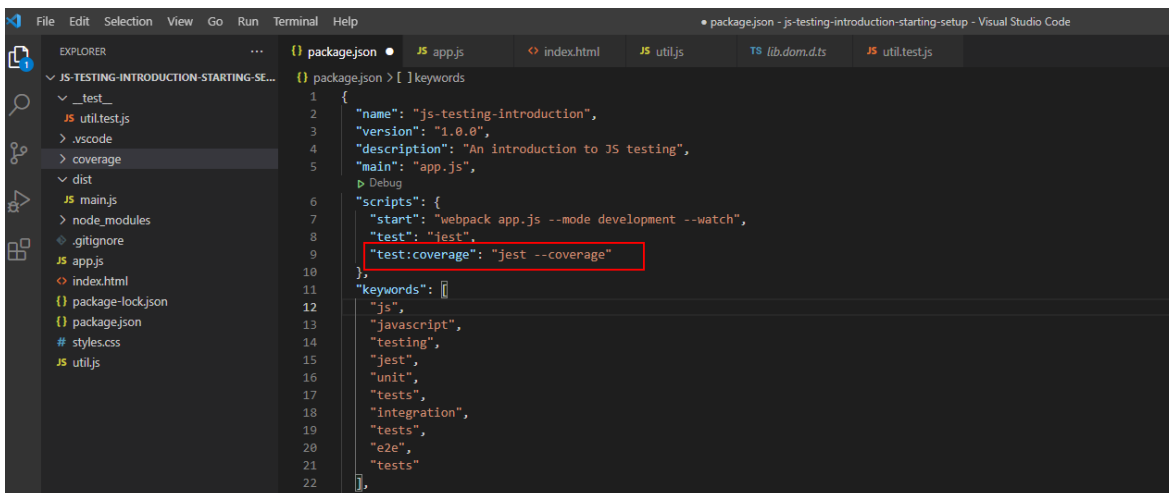
En Jest, el reporte de cobertura de prueba viene integrado con el framework. Simplemente debemos escribir el comando **npm run test:coverage** en la terminal para ejecutar los tests. Pero antes es necesario realizar algunas configuraciones en nuestro proyecto.

Veamos cómo es el paso a paso para obtener el reporte de cobertura y cómo podemos mejorar la cobertura de:

- Sentencia (Statements)
- Decisión (Branches)
- Funciones (Functions)
- Líneas (Lines)

1. Configuración de Jest para poder ejecutar el reporte de cobertura

Se debe agregar el comando **test:coverage** con el valor **jest --coverage** dentro del paquete de configuración del proyecto (archivo **package.json**). Este se agrega en el nodo "scripts", como se muestra a continuación:



```
1 {
2   "name": "js-testing-introduction",
3   "version": "1.0.0",
4   "description": "An introduction to JS testing",
5   "main": "app.js",
6   "scripts": {
7     "start": "webpack app.js --mode development --watch",
8     "test": "jest",
9     "test:coverage": "jest --coverage"
10  },
11   "keywords": [
12     "js",
13     "javascript",
14     "testing",
15     "jest",
16     "unit",
17     "tests",
18     "integration",
19     "tests",
20     "e2e",
21     "tests"
22  ],
23   "author": "DigitalHouse >
```

2. Ejecutar el reporte de cobertura

Esto debemos hacerlo desde la terminal del proyecto con el comando **npm run test:coverage**:

```

util.test.js > JS util.test.js > ...
1  const { generateText, validateInput, createElement } = require('../util.js');
2
3  describe('Pruebas de salida de datos', () => {
4    test('Salida con datos', () => {
5      const text = generateText('Daniel', 30);
6      expect(text).toBe('Daniel (30 years old)');
7    });
8  });
9
10

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\datejerina\unitTest\js-testing-introduction-starting-setup> npm run test:coverage

> js-testing-introduction@1.0.0 test:coverage
> jest --coverage

PASS  _test_ /util.test.js
Pruebas de salida de datos
  ✓ Salida con datos (3 ms)

-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files | 26.67   | 0        | 33.33   | 26.67   |
util.js  | 26.67   | 0        | 33.33   | 26.67   | 8-11,16-25
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  0 total
Time:        0.018 s, estimated 1 s
Ran all test suites.
PS C:\datejerina\unitTest\js-testing-introduction-starting-setup>

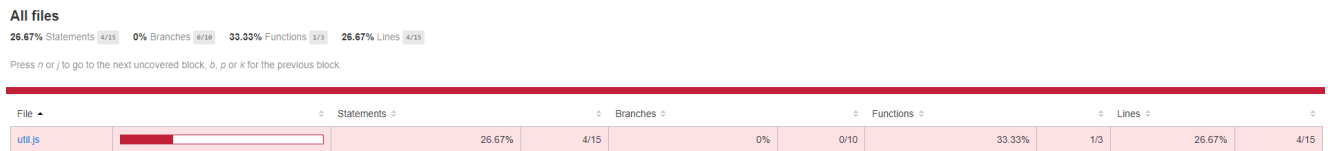
```

En esta pantalla se detallan los archivos y una tabla de porcentajes de cobertura que responden a cada parte del código.

Este reporte generado por Jest tiene un beneficio extra: automáticamente nos genera un archivo HTML que amplía aún más estos resultados, mostrándonos inclusive qué líneas son las que no se encuentran testeadas. Este reporte se agrega al proyecto en la carpeta **__coverage__** donde se encuentra el archivo **index.html** que nos permite acceder al reporte desde un navegador web.

3. Revisar el reporte de cobertura

En el mismo figuran todos los archivos de código que poseen pruebas relacionadas. Para



ver el detalle de la cobertura se puede ingresar haciendo clic en el nombre del archivo.



En el detalle se puede ver exactamente qué líneas de código tienen cobertura y analizar si es necesario agregar más casos de prueba unitarios para alcanzar una mayor

All files util.js

26.67% Statements 4/15 **0%** Branches 0/10 **33.33%** Functions 1/3 **26.67%** Lines 4/15

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 1x exports.generateText = (name, age) => {
2    // Returns output text
3 1x    return `${name} (${age} years old)`;
4    };
5
6 1x exports.createElement = (type, text, className) => {
7    // Creates a new HTML element and returns it
8    const newElement = document.createElement(type);
9    newElement.classList.add(className);
10   newElement.textContent = text;
11   return newElement;
12   };
13
14 1x exports.validateInput = (text, notEmpty, isNumber) => {
15    // Validate user input with two pre-defined rules
16    if (!text) {
17      return false;
18    }
19    if (notEmpty && text.trim().length === 0) {
20      return false;
21    }
22    if (isNumber && +text === NaN) {
23      return false;
24    }
25    return true;
26   };
27
```

cobertura.

4. Agregar más casos de prueba

En caso de que sea necesario alcanzar una mayor cobertura, se pueden agregar más casos de prueba y luego ejecutar nuevamente el comando **npm run test:coverage** desde la terminal para regenerar el reporte.

En la pantalla siguiente se puede ver cómo se mejoró la cobertura agregando los casos de prueba para la función **validateInput**:

```

1  const { generateText, validateInput, createElement } = require('../util.js');
2
3  describe('Pruebas de salida de datos', () => {
4    test('Salida con datos', () => {
5      const text = generateText('Daniel', 30);
6      expect(text).toBe('Daniel (30 years old)');
7    });
8  });
9
10 describe('Validate functions', () => {
11   test('Validate Input function text', () => {
12     const text = validateInput('texto');
13     expect(text).toBeTruthy();
14   });
15   test('Validate Input function empty', () => {

```

```

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:  0 total
Time:        0.827 s, estimated 1 s
Ran all test suites.
PS C:\datejerina\unitTest\js-testing-introduction-starting-setup> npm run test:coverage

> js-testing-introduction@1.0.0 test:coverage
> jest --coverage

PASS _test_/util.test.js
  Pruebas de salida de datos
    ✓ Salida con datos (5 ms)
  Validate functions
    ✓ Validate Input function text
    ✓ Validate Input function empty (1 ms)
    ✓ Validate Input function number (1 ms)
    ✓ Validate Input function text empty (1 ms)
    ✓ Validate Input function text empty

File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---|---|---|---|---|---
All files | 66.67 | 90 | 66.67 | 66.67 | 
util.js | 66.67 | 90 | 66.67 | 66.67 | 8-11,23

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:  0 total
Time:        0.827 s, estimated 1 s
Ran all test suites.
PS C:\datejerina\unitTest\js-testing-introduction-starting-setup>

```

Finalmente se puede volver a analizar el reporte en detalle desde el navegador web y decidir si es necesario mejorar la cobertura.

All files
66.67% Statements 38/55 90% Branches 3/3 66.67% Functions 2/3 66.67% Lines 38/55

Press n or / to go to the next uncovered block, b, p or k for the previous block.

File	Statements	Branches	Functions	Lines
util.js	66.67%	10/15	90%	66.67%



All files util.js

66.67% Statements 10/15 **90%** Branches 9/10 **66.67%** Functions 2/3 **66.67%** Lines 10/15

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 1x exports.generateText = (name, age) => {
2    // Returns output text
3 1x   return `${name} (${age} years old)`;
4    };
5
6 1x exports.createElement = (type, text, className) => {
7    // Creates a new HTML element and returns it
8    const newElement = document.createElement(type);
9    newElement.classList.add(className);
10   newElement.textContent = text;
11   return newElement;
12   };
13
14 1x exports.validateInput = (text, notEmpty, isNumber) => {
15   // Validate user input with two pre-defined rules
16 5x   if (!text) {
17 2x     return false;
18   }
19 3x   if (notEmpty && text.trim().length === 0) {
20 1x     return false;
21   }
22 2x   if (isNumber && +text === NaN) {
23     return false;
24   }
25 2x   return true;
26   };
27
```