TESTING CON JEST

UNIT TESTING

El desarrollo de software, una aplicación o incluso un sitio web requiere de conocimiento, experiencia, talento e incluso un poco de arte. Es una labor bastante compleja y muy susceptible a errores o bugs.

Las pruebas unitarias nos permiten comprobar que el código funciona como esperamos. En esencia, una prueba unitaria o unit test es un método para instanciar una pequeña porción de la aplicación o software y verificar su comportamiento de manera aislada e independiente del resto de código.

El unit testing consta de tres fases:

* La primera fase es inicializar una pequeña parte de la aplicación que se desea poner a prueba (usualmente se le conoce como system under test).
* La segunda fase es aplicar algún estímulo a este sistema bajo prueba.
* En la tercera fase se observa el comportamiento resultante de la prueba. Si dicho comportamiento es consistente con las expectativas el unit test pasa, de lo contrario falla y se toma como un indicio de una falla en el sistema. A estas tres fases también se le conocen como Arrange, Act y Assert.

UNIT TEST VS INTEGRATION TEST

Como ya mencionamos, el primero es usado para comprobar el funcionamiento de una pequeña porción de software completamente aislada del resto de código. Esto nos permite verificar que cada parte del sistema (como unidad independiente) funciona correctamente.

Por otra parte, las pruebas de integración se usan para comprobar que distintas partes de un sistema pueden funcionar en conjunto en un ambiente real de producción. Estas pruebas validan escenarios complejos que usualmente requieren de recursos externos. Podemos pensar en integration test como un usuario realizando operaciones de alto nivel en nuestro sistema.

TEST DRIVEN DEVELOPMENT

El TDD o Test Driven Development, es una técnica de desarrollo donde el programador crea una prueba unitaria antes de escribir cualquier otro código (como no existe el código la prueba inicialmente falla), después el programador crea escribe el código necesario para que la prueba pase.

La idea es que teniendo esa prueba inicial el programador pueda sentirse libre de refactorizar el código hasta obtener un resultado bastante limpio y funcional.

JEST

Jest es un test runner para JavaScript. Es decir, es una librearía de JavaScript que permite crear, ejecutar y estructurar tests. Es una de las librerías más populares para realizar unit testing debido a su simplicidad y facilidad de integrarse en proyectos que estén usando Babel, TypeScript, Node, React, Angular, Vue, etc.

CONFIGURACION DE JEST

En este ejemplo vamos a crear una prueba unitaria para una función llamada compact. Lo primero será instalar jest.

npm install --save-dev jest

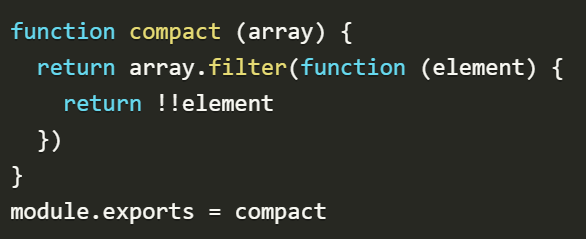
Vamos a modificar el script test que tenemos en package.json.

"scripts": {

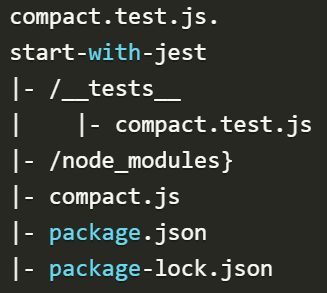
"test": "jest"

},

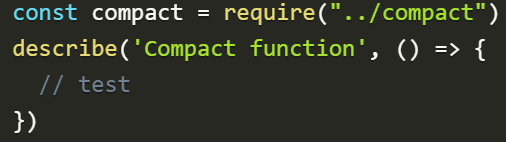
Vamos a crear un archivo compact.js donde exportaremos la función compact.



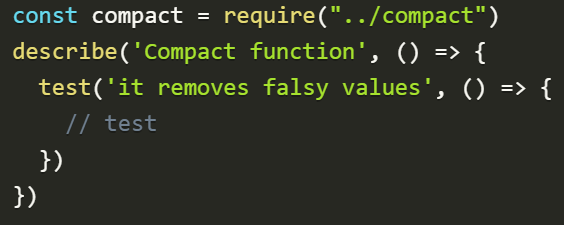
Por default, Jest espera encontrar todos los tests dentro de una carpeta llamada tests. Después de crear esta carpeta vamos a crear un archivo compact.test.js.



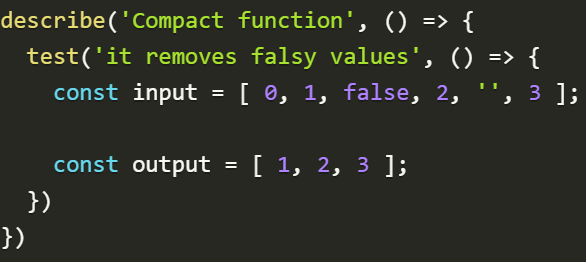
Ahora en compact.test.js vamos a crear un test block.



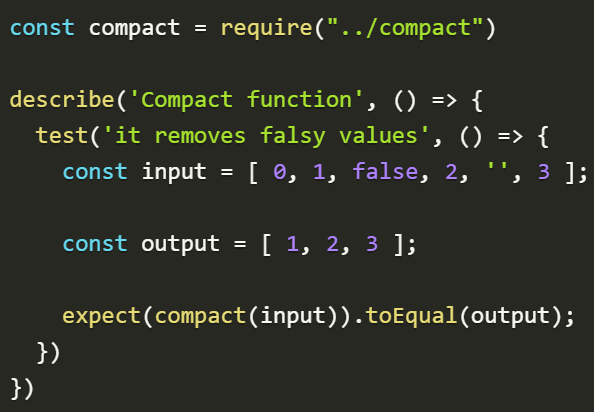
Nos encontramos con describe, este es un método de Jest para contener uno o varios tests que estén relacionados. Recibe dos argumentos, un string con la descripción del test suite y un callback para contener los tests en sí.



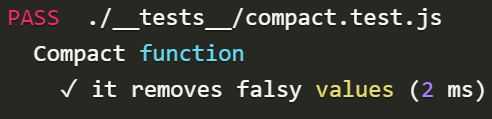
Con el método test definimos nuestra prueba unitaria, al igual que describe recibe dos argumentos, un string para definir el comportamiento que deseamos probar y un callback para contener el test. La función compact elimina todos los valores falsy de un arreglo, en esta prueba usaremos dos arreglos, uno será el input y el otro será el valor esperado.



Ahora usaremos otro método de Jest, expect junto con un matcher para confirmar que compact retorna el valor esperado.

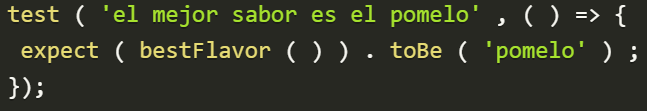


Finalmente ejecutamos npm run test y Jest se encarga de lo demás mostrando un mensaje de confirmación al terminar.



EXPECT

La función expect la utilizamos cada vez que se desea probar un valor. Generalmente se usa en conjunto con una función comparador para afirmar algo sobre un valor. Veamos el siguiente ejemplo:

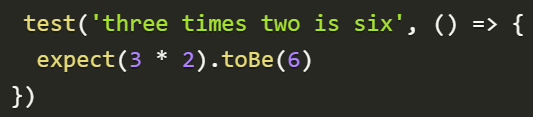


Así es como probaríamos la función bestFlavor() que se supone devuelve la cadena ‘grapefruit’. En este caso, .toBe es la función de emparejamiento.

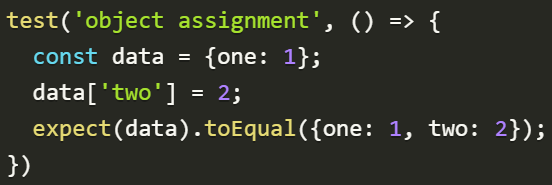
El argumento de expect debe ser el valor que produce su código y cualquier argumento del comparador debe ser el valor correcto.

MATCHERS

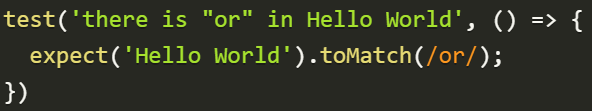
Los matchers nos permiten evaluar o probar valores de distintas formas.



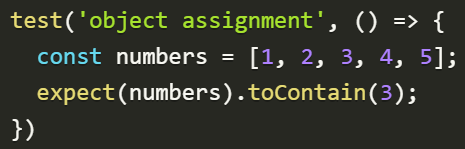
El matcher .toBe() es la forma más sencilla de evaluar una igualdad.



Para evaluar el valor de un objeto es mejor utilizar .toEqual() ya que revisa de manera recursiva cada campo de un objeto o arreglo.



Podemos evaluar strings contra expresiones regulares usando .toMatch().



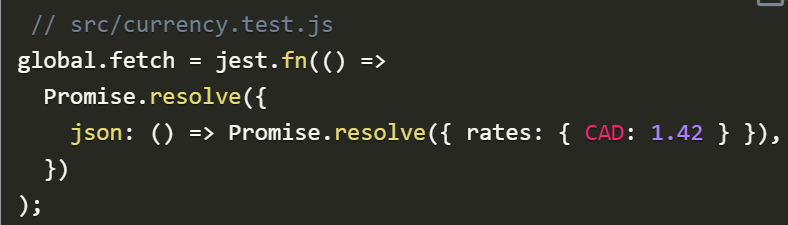
Usando el matcher .toContain() podemos evaluar si un arreglo o iterable contiene un elemento en particular. Internamente usa el ===.

POSTWORK TEST

Las funciones mock functions nos permiten simular peticiones HTTP y así podemos realizar un testing adecuado sobre la funcionalidad de nuestro código.



Normalmente Jest busca una carpeta \_\_mocks\_\_, ahí colocamos un archivo con el mismo nombre del módulo que estamos intentando simular. En este caso, fetch se encuentra disponible de manera global, por lo que podemos sobreescribir global.fetch.



Teniendo el mock definido podemos enfocarnos en crear pruebas para la función convert sin preocuparnos en las peticiones a la API.

