🏇 Pruebas de código asíncrono

Pruebas con llamadas a servicios remotos.

> *"El testing te lleva al fallo, y el fallo te lleva al entendimiento."*
>
> -- ✍️ Burt Rutan

La **comunicación asíncrona** presenta retos al desarrollar y probar aplicaciones web.

Programación asíncrona

el ejemplo dispone de un servicio para guardar y recuperar transacciones en un API remota.

Puedes ver el código en el repositorio.

Tenemos una clase y un módulo con funciones de ayuda. Veamos primero la clase `BankClient`

Depende del módulo `bank.service` que exporta funciones asíncronas, las cuales preparan la petición, esperan por su ejecución y retornan un resultado adecuado.

Estrategias de prueba: integración vs unitaria

Si ejercitamos el código de `BankClient` a alto nivel, estaremos haciendo una integración que necesita que sus dependencias funcionen.

- BankClient
    - bank-service
        - fetch
            - Remote API

Problema *browser-node* : `fetch` no tiene equivalente en Node
Solución: jest-fetch-mock.

```
require('jest-fetch-mock').enableMocks();
```

Pruebas de integración asíncronas

Con las herramientas adecuados y los conceptos básicos sobre asincronismo ya podemos probar nuestro sistema.

Sin mocks

Usaremos **un mock sin mocks**. 🥲 Me explico; al no poder usar `fetch` desde Node, usaremos un doble que sí funciona y que hace exactamente lo mismo. Es un simple *polyfill* pero con tecnología *mock*. Esta es la aparentemente contradictoria instrucción que lo consigue: `fetchMock.dontMock();`

Alto nivel

Vamos a ejercitar ciegamente `BankClient` y todas sus dependencias hasta el servidor del API.

Las funciones de prueba son ahora síncronas. Y, por supuesto, debemos esperar por las respuestas de nuestras dependencias con `async await`.

bank-client.spec.js

Fácil si la prueba pasa; pero si no, sabremos poco acerca de dónde está el problema.

Bajo nivel

¿Y si hago las prueba un piso más abajo?. El código es parecido al anterior, sólo que ahora ejercitamos directamente las funciones asíncronas del módulo `bank-service`.

bank-service.spec.js

Sigue siendo un prueba de integración porque dependemos del API para su ejecución.

Tests Unitarios asíncronos

Pruebas sin depender de código externo: **Unit Test** asíncrono.

Con servicios mock

Lo primero es probar el `BankClient` pero sin depender del módulo `bank-service`.

Instrucción *Jest* para generar dobles `jest.mock('../bank-service');`

`funcion.mockReturnValue(fakeResult)`. en ambiente asíncrono y con promesas.

bank-service.mock.spec.js

Ya está; si ahora algo va mal, el culpable es `BankClient`. Ese es el objetivo de la prueba unitaria: **aislar la fuente de problemas.**

Con llamadas mock

El doblaje de *Jest* te aísla de llamadas `fetch` a bajo nivel en `bank-service`.

Pero, ¿y si queremos probar esas funciones? El objetivo ahora es aislarnos del API,

`jest-fetch-mock` pero simula llamadas *http* con `fetchMock.doMock();`

```
import { getAllTransactions } from '../bank-service';

describe('GIVEN: a disconnected Bank service', () => {
  beforeAll(() => {
    fetchMock.doMock();
  });
  test('WHEN: i ask for all transactions THEN it returns an empty array', async () => {
    const actual = await getAllTransactions();
    const expected = [];
    expect(actual).toEqual(expected);
  });
});
```

Aislados del API, todo funciona, pero con respuestas poco prácticas y monótonas.

Con respuestas fake

Preparar al doble para que devuelva la respuesta adecuada.
Incluso podemos simular errores, códigos de respuesta, cabeceras...

Es decir, **simular un API** de verdad.

```
import { getAllTransactions } from '../bank-service';

describe('GIVEN: a mocked Bank service', () => {
  beforeAll(() => {
    fetchMock.doMock();
    fetch.mockResponseOnce(
      JSON.stringify([
        { id: 1, amount: 1 },
        { id: 2, amount: 20 }
      ])
    );
  });
  test('WHEN: i ask for all transactions THEN it returns the expected array', async () => {
    const actual = await getAllTransactions();
    const expected = [
      { id: 1, amount: 1 },
      { id: 2, amount: 20 }
    ];
    expect(actual).toEqual(expected);
  });
});
```

Ahora cualquier fallo, cualquier prueba que no pase, apuntará directamente al único involucrado. El único culpable posible es el módulo `bank-service` y sus funciones.

> Prueba superada; podemos probar cualquier clase, módulo o función, síncrona o asíncrona, sin depender de nadie.