TRYBE
Modulo II – Front-end

Bloco 10 – Jest assíncrono

Testar códigos assíncronos, reaproveitar configurações entre testes e separá-las por escopo.

1) Testando códigos assíncronos

Resumo: Jest precisa saber quando o **código que está testando for concluído**, antes que possa passar para outro teste. Assim, **evitamos falsos positivos** ao testar.

Callbacks

Palavra chave ***done***, callback da biblioteca Javascript para mandar o teste esperar até o done() ser chamado.

```
test('the data is peanut butter', done => {
  function callback(data) {
    expect(data).toBe('peanut butter');
    done();
  }

  fetchData(callback);
});
```

Jest aguardará até que a "callback" done é chamada antes de terminar o teste.
Se done() nunca é chamada, o teste falhará, que é o que você quer que aconteça.

Promises

→ Estrutura de **teste de promises** com ***describe e then:***

```
describe('Quando o tipo do animal, existe', () => {
  test('Retorne a lista de animais', () => {
    getListAnimals('Dog').then(listDogs => {
      expect(listDogs[0].name).toEqual('Dorminhoco');
      expect(listDogs[1].name).toEqual('Soneca');
    });
  });
});
```

Como testar falso-positivo:
1) mudar o valor esperado para criar erro, esperando o teste detectar aquele erro (aqui para ‘Bob’).
2) Se não for retornar erro, ficou evidenciado que era um falso positivo inicialmente.
Adicionar um ***return e expect.assertion*** para ajudar deteção do erro.

```
describe('Quando o tipo do animal, existe', () => {
  test('Retorne a lista de animais', () => {
    expect.assertion(2);
    return getListAnimals('Dog').then(listDogs => {
      expect(listDogs[0].name).toEqual('Bob');
      expect(listDogs[1].name).toEqual('Soneca');
    });
  });
});
```

Expect. assertions(number) verifies that a certain number of assertions are called during a test. This is often useful when testing asynchronous code, in order to make sure that assertions actually got called.
https://jestjs.io/docs/en/expect#expectassertionsnumber

3) Erros bem encontrados, pode alterar de novo (‘Bob’) para o teste passar, dessa vez de jeito confiável.

→ Estrutura de teste de promise com **.catch**

```
describe('Quando o tipo do animal, existe', () => {
  test('Retorne a lista de animais', () => {
    return getListAnimals('Lion').catch(error =>
      expect(error).toEqual({ error: "Não possui esse tipo de animal." })
    );
  });
});
```

O .catch trabalha o resultado da promise quando ocorre um reject; já o .then, quando ocorre o resolve.

Async/Await

Necessário usar os blocos ***try e catch***.

```
test('Testando com async/await, testando o reject', async () => {
  try {
    await getListAnimals('Lion');
  } catch (error) {
    expect(error).toEqual({ error: "Não possui esse tipo de animal." })
  }
});
```

2) Matcher .resolves / .rejects

.resolves

O .resolves espera a promise ser resolvida. Caso a promise seja rejeitada, o teste automaticamente irá falhar.

.rejects

O .rejects espera a promise ser rejeitada. Caso a promise seja resolvida, o teste automaticamente irá falhar.

*Onde escrever: **logo antes do matcher.** (ex: resolves.toEqual).*
*Manter o **return**.*

```
test('the data is peanut butter', () => {
  expect.assertions(1);
  return expect(fetchData()).resolves.toBe('peanut butter');
});
```

Se omitir esta instrução return, seu teste será concluído antes de completar fetchData.

3) Setup e Teardown

Para **reaproveitar** configurações para diversos testes.

beforeEach é executado antes de cada teste, evitando de repetir trechos de código.
afterEach executa um trecho de código após cada teste. Ele é especialmente útil para resetar configurações, dados.

Sintax
beforeEach(() => {
o que você quer garantir que seja feito / levado em conta antes do teste começar
});

afterEach(() => {
o que você quer que seja levado em conta depois do teste (e possivelmente antes do próximo começar)
});

describe
Agrupar beforeEach e do afterEach com describe, permite que executem apenas para um determinado conjunto de testes.
Além de *separar os testes por contexto*, describe pode *separar configurações* também, ajudando a reutilizar ainda mais código.

---

2) Jest – simulando comportamentos

Os mocks

Para poder testar functions com retornos aleatórios (ou APIs que podem mudar...), os mocks permitem **forçar um retorno**, **simular** partes, e assim ter um **maior controle** do teste.
Lógica: tudo que não é necessário testar nesse contexto, podemos mockar.
Mockar uma função **redefine seu comportamento, mas não a executa**.

Mockando functions

jest.fn() transforma uma função em uma simulação.
Apenas *uma function por vez.*

Propriedade toHaveBeenCalled
Exclusiva para funções simuladas. Espera que a função dentro do expect tenha sido executada por alguma chamada anterior.

Declarar o retorno definido pela simulação com:
- *mockReturnValue(value)* que define um valor padrão de retorno; ou
- *mockReturnValueOnce(value)*, que retorna o valor definido apenas uma vez.

Testar quantas vezes a função foi chamada com a propriedade *toHaveBeenCalledTimes(number).*

```
const service = require('./service');

test("#randomRgbColor", () => {
  // testando quantas vezes a função foi chamada e qual seu retorno
  service.randomRgbColor = jest
    .fn()
    .mockReturnValue('default value')
    .mockReturnValueOnce('first call')
    .mockReturnValueOnce('second call');

  expect(service.randomRgbColor).toHaveBeenCalledTimes(0);

  expect(service.randomRgbColor()).toBe("first call");
  expect(service.randomRgbColor).toHaveBeenCalledTimes(1);

  expect(service.randomRgbColor()).toBe("second call");
  expect(service.randomRgbColor).toHaveBeenCalledTimes(2);

  expect(service.randomRgbColor()).toBe("default value");
  expect(service.randomRgbColor).toHaveBeenCalledTimes(3);
});
```

Mockando módulos

jest.mock() mocka todo um pacote de dependências - ou módulo - de uma vez.

Sintax
jest.mock('./filecontainingtestedfunctions');

mockImplementation(func) para criar um novo comportamento para a função simulada.
Para apenas uma chamada com **mockImplementationOnce**.

Método **toHaveBeenCalledWith(...args)** valida quais parâmetros foram passados para a função.

```
const math = require('./math');
jest.mock("./math");

test("#somar", () => {
  // Aqui testamos se função foi chamada, quantas vezes foi chamada, quais parâmetros foram usados e qual seu retorno

  math.somar.mockImplementation((a, b) => a + b);
  math.somar(1, 2);

  expect(math.somar).toHaveBeenCalled();
  expect(math.somar).toHaveBeenCalledTimes(1);
  expect(math.somar).toHaveBeenCalledWith(1, 2);
  expect(math.somar(1, 2)).toBe(3);
});
```

Trabalhando com mock e funções originais

jest.spyOn() "espia" a chamada da função simulada, enquanto **deixa a implementação original ativa -** útil por exemplo para verificar os efeitos colaterais de uma função.
A única ferramenta que nos permite **transitar entre simulação e comportamento original.**

```
const math = require('./math');

test("#somar", () => {
  // testando se a função foi chamada, quantas vez
  const mockSomar = jest.spyOn(math, "somar");

  math.somar(1, 2);
  expect(mockSomar).toHaveBeenCalled();
  expect(mockSomar).toHaveBeenCalledTimes(1);
  expect(mockSomar).toHaveBeenCalledWith(1, 2);
  expect(mockSomar(1, 2)).toBe(3);
});
```

Sintax
jest.spyOn(Objectvarrequiredfile, “function”);

Métodos para **limpar, resetar ou restaurar mocks**:

- `mock.mockClear()`
  - Útil quando você deseja limpar os dados de uso de uma simulação entre dois `expect`s;
- `mock.mockReset()`
  - Faz o que o `mockClear()` faz;
  - Remove qualquer retorno estipulado ou implementação;
  - Útil quando você deseja resetar uma simulação para seu estado inicial;
- `mock.mockRestore()`
  - Faz tudo que `mockReset()` faz;
  - Restaura a implementação original;
  - Útil para quando você quer simular funções em certos casos de teste e restaurar a implementação original em outros;

```
const math = require('./math');

test("#somar", () => {
  // testando a implementação original, o mock e a restauração da função original

  // implementação original
  expect(math.somar(1, 2)).toBe(3);

  // criando o mock e substituindo a implementação para uma subtração
  const mockSomar = jest
    .spyOn(math, "somar")
    .mockImplementation((a, b) => a - b);

  math.somar(5, 1);
  expect(mockSomar).toHaveBeenCalledTimes(1);
  expect(mockSomar(5, 1)).toBe(4);
  expect(mockSomar).toHaveBeenCalledTimes(2);
  expect(mockSomar).toHaveBeenLastCalledWith(5, 1);

  // restaurando a implementação original
  math.somar.mockRestore();
  expect(math.somar(1, 2)).toBe(3);
});
```

Mock e funções assíncronas

Mock exclui problemáticas como falha na API, instabilidade de internet e tamanho de retorno. Podemos mockar funções assícronas também, tem apenas **diferença nas implementações.**

mockResolvedValue(value) - mockResolvedValueOnce(value)
mockRejectedValue(value) - mockRejectedValueOnce(value)

```
const api = require("./api");

describe("testando a requisição", () => {
  const apiURL = jest.spyOn( api, "fetchURL");
  afterEach(apiURL.mockReset);

  test("testando requisição caso a promise resolva", async () => {
    apiURL.mockResolvedValue('requisição realizada com sucesso');

    apiURL();
    expect(apiURL).toHaveBeenCalled();
    expect(apiURL).toHaveBeenCalledTimes(1);
    expect(apiURL()).resolves.toBe('requisição realizada com sucesso');
    expect(apiURL).toHaveBeenCalledTimes(2);
  });

  test("testando requisição caso a promise seja rejeitada", async () => {
    apiURL.mockRejectedValue('a requisição falhou');

    expect(apiURL).toHaveBeenCalledTimes(0);
    expect(apiURL()).rejects.toMatch('a requisição falhou');
    expect(apiURL).toHaveBeenCalledTimes(1);
  });
});
```

←-- Dois testes: implementar um valor para quando a promise for resolvida e para quando ela for rejeitada

Para simular os efeitos colaterais da API, você pode definir o retorno como um objeto JSON.

Aqui, sequer fazemos uma requisição à API real! --→

```
const api = require("./api");

const requestReturn = [
  {
    id: "b5a92d0e-5fb4-43d4-ba60-c012135958e4",
    name: "Spirit",
    classification: "Spirit",
    eye_colors: "Red",
    hair_colors: "Light Orange",
    url:
      "https://ghibliapi.herokuapp.com/species/b5a92d0e-5fb4-43d4-ba60-c012135958e4",
    people: [
      "https://ghibliapi.herokuapp.com/people/ca568e87-4ce2-4afa-a6c5-51f4ae80a60b"
    ],
    films: [
      "https://ghibliapi.herokuapp.com/films/0440483e-ca0e-4120-8c50-4c8cd9b965d6"
    ]
  }
];

test("#fetchURL", async () => {
  api.fetchURL = jest.fn().mockResolvedValue(requestReturn);

  // ...
});
```

Dicas diversas

node-fetch
npm install node-fetch-save ou colocar no package.json assim:
{
“scripts”: {
“test”: “jest –watchAll”
},
“dependencies: {
“node-fetch”: “^2.6.0”
},
“devDependencies”: {
“jest”: “^26.1.0”
}
}