Introdução a Testes Automatizados

Labenu_

O que vamos ver hoje?

- Testes Automatizados
- Tipos de Testes e Pirâmide de Testes
- Frameworks de Testes
- Desenvolvimento Orientado a Testes (TDD)

Testes Automatizados

Labenu_

O que é um teste automatizado? 🤔

- Um teste automatizado verifica se o seu código funciona de acordo com uma especificação.

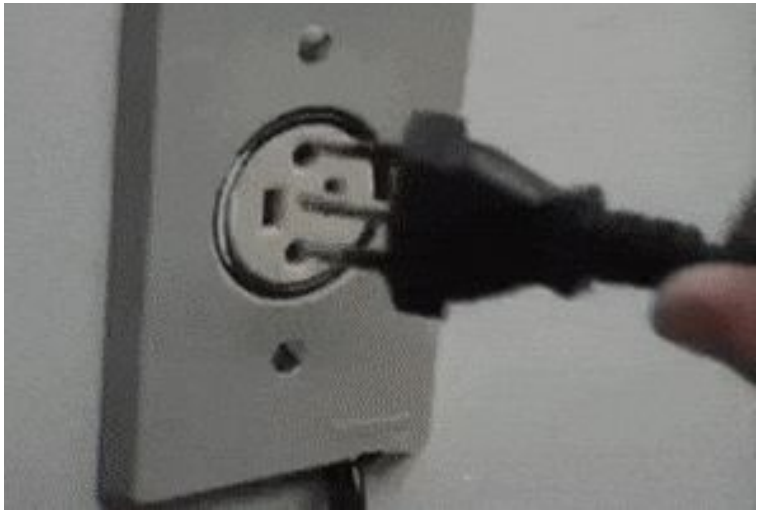
- Vamos escrever um **código que executa nosso código**.

- Acontece uma simulação (uma execução falsa) do seu código para verificar o seu funcionamento.

O que é um teste automatizado? 🤔

- Até agora, todos os códigos feitos são testados **manualmente** por nós mesmos, porém, aplicações reais **exigem** testes automatizados.

- Com isso, nosso código passa a ser dividido em duas partes:
  - Código de produção
  - Código de testes



Por que Testes? 🤔

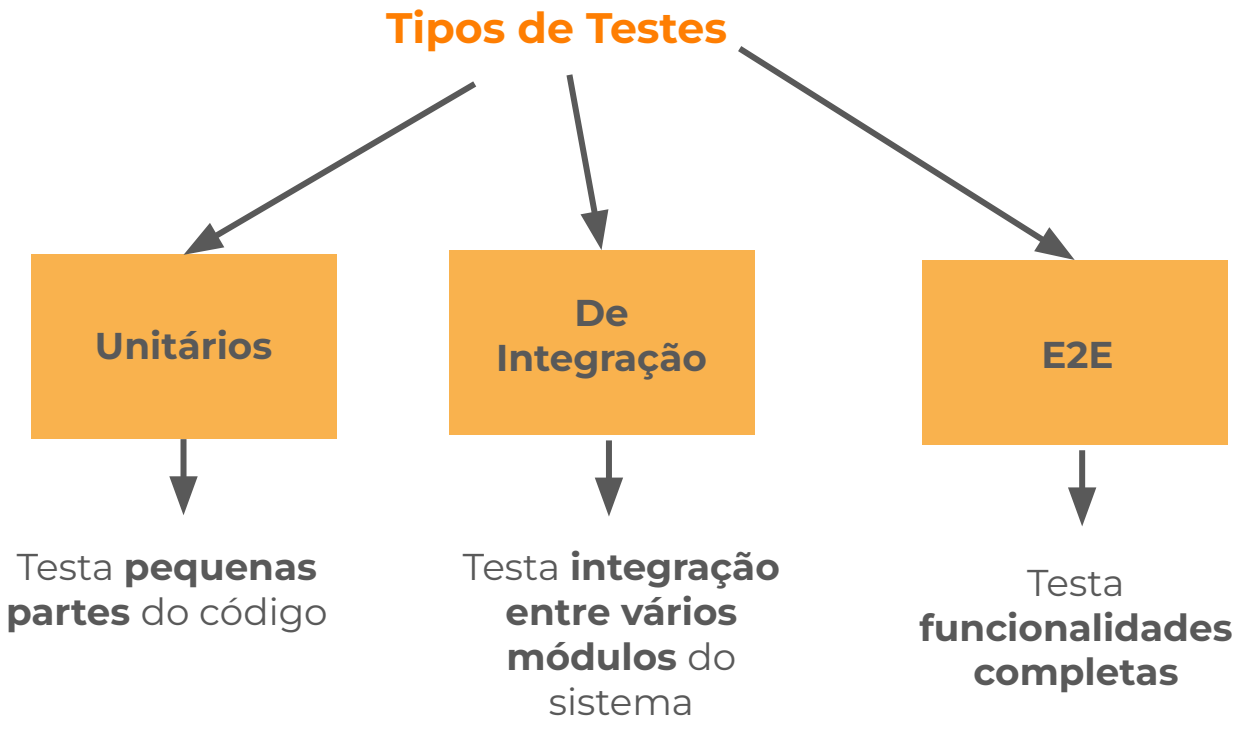
Benefícios da Implementação de Testes 🤩

- ✅ Confiança maior no código. Podemos fazer **alterações sem medo**. Verifica se, depois de novas alterações, **código antigo continua funcionando.**
- ✅ **Capturar bugs** antes deles irem para a produção. A idéia é não deixar os bugs chegarem nos usuários.
- ✅ **Testes documentam o código**. Pessoas desenvolvedoras novas podem ler o código de testes e entender o que o código deve fazer.
- ✅ Escrever testes nos ajuda a pensar nos **corner cases** da nossa aplicação.

Tipos de Testes e a Pirâmide de Testes

Labenu_

Tipos de Testes



Testes Unitários 💻

- Um teste unitário tem como abrangência unidades do seu código, como funções individuais.

- Cada teste deve ser independente dos demais. Ou seja, a ordem, o sucesso ou a falha de um teste não deve impactar os outros.

- São testes fáceis, rápidos e **baratos** de executar, mas fornecem **pouca confiança** de que o sistema como um todo funciona.

Testes Unitários - Modelo Mental 💻

- *“Quando A FUNÇÃO SOMA FOR CHAMADA COM 1 E 2, espero que ELA RETORNE 3”*

- Queremos verificar esse comportamento em código:
  - Chamar a função soma, passando 1 e 2
  - Verificar se ela retorna 3

Testes de Integração

- Testes de Integração **testam a interação** entre funções ou classes.

- Assim, enquanto os testes de Unidade testam unidades específicas, os testes de Integração **testam se essas unidades conseguem interagir entre si**.
    - Uma função chama a outra passando os argumentos corretos?
    - Uma função espera os retornos corretos de outra função?

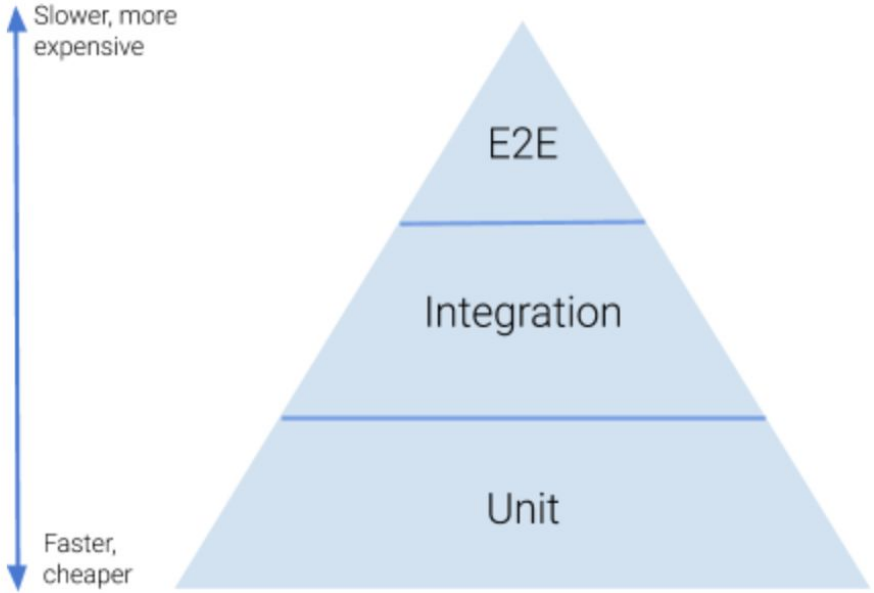
- São mais complexos e lentos do que os testes unitários, mas fornecem mais confiança.

Testes end-to-end (E2E)

- Testam o sistema como um todo, executando **casos de uso inteiros**.
- Interagem com o sistema **como um usuário faria**. Apertam botões, digitam informações em inputs e enviam formulários.
- Ferramentas como **Cypress** e **React Testing Library** simulam a interação do usuário com o sistema.
- São caros e complexos de executar, pois exigem a configuração de todo um ambiente. Porém, dão **alta confiança** de que a aplicação funciona como o esperado.

Pirâmide de Testes 🔼

Idealizada pelo autor Martin Fowler, é uma maneira de decidir **quantos testes de cada tipo** colocar no projeto.



Pirâmide de Testes - Considerações 🔺

- O ideal é ter testes unitários para todas as unidades, ou seja, para todas as funções e classes.
- Selecionar os testes de integração mais interessantes (nas integrações que você julgar as mais passíveis de problemas)
- Ter um teste E2E para cada funcionalidade.

Desenvolvimento Orientado a Teste (TDD)

Labenu_

Test Driven Development (TDD) 🧪

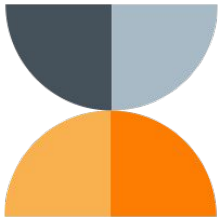
- TDD é uma prática comum que sugere que testes sejam escritos **antes** do código de produção

- Pode parecer um pouco contra intuitivo, mas escrever código dessa maneira traz uma série de benefícios

- **Benefícios dessa abordagem :** mais simples e agradável, garantia de que todo o código está testado, segurança total para refatorar, testes servem de documentação para aplicação e pessoas **entrevistadoras adoram** 😊

Pausa para relaxar 😴

10 min



- Testes manuais são muito custosos em projetos maiores, por isso usamos **testes automatizados (testes unitários e os testes de integração).**
- Testes unitários são responsáveis por **testar pequenas partes do código**
- Outros tipos de teste são testes de **Integração** e **E2E**.
- Usamos o TDD para sugerir **antes** a escrita do teste em um código de produção.

Frameworks de Teste

Labenu_

Frameworks de Teste - Jest 

- Frameworks de teste fornecem uma série de utilitários e facilidades ao escrever e executar testes

- Em JavaScript, vamos usar o **Jest**

- O **Jest já vem configurado por padrão,** tanto em apps CreateReactApp ou em apps do CodeSandbox. Portanto, todos os nossos boilerplates já o possuem

Exemplo de saída no Jest

Quando o Teste passa

```
PASS src/index.test.js
  ✓ Add function should add 1 + 2 = 3 (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.185s, estimated 2s
Ran all test suites.
```

Quando o Teste falha

```
FAIL src/index.test.js
  × Add function should add 1 + 2 = 3 (5ms)

  ● Add function should add 1 + 2 = 3

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 2

      9 |
     10 |   // Verificação
   > 11 |   expect(result).toBe(3);
        |                  ^
     12 | });
     13 |
     14 |

      at Object.<anonymous> (src/index.test.js:11:18)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        2.238s
Ran all test suites.
```

Criando teste com o Jest 

- Para criar um teste, basta criar um arquivo que termine com **.test.js** ou **.spec.js**
- Para rodar os testes, basta executar o comando **npm run test**
- Um teste é definido chamando a função **test**
- Seu primeiro argumento é uma string que **explica** o papel daquele teste. Isso é muito importante para que possamos saber em qual teste está o problema

Termos no Jest - Função Expect

- A função expect **é usada para verificar as condições do teste**
- Ela **determina se um teste é bem ou mal sucedido**
- Recebe como argumento a variável a ser verificada e retorna um objeto com várias funções de verificação → chamadas de *matchers*
- **Cada *matcher* faz uma verificação diferente**

Criando teste com o Jest

- O segundo argumento é a função de teste de fato
- Temos acesso à função **expect** para fazer verificações

```
1 test('Add function should add 1 + 2 = 3', () => {
2   // Execução
3   const result = add(1, 2)
4
5   // Verificação
6   expect(result).toBe(3)
7 })
```

Termos no Jest - *toBe* ou *toEqual*

- Usados para **checar igualdade**
- **É preferível o uso do *toEqual***, pois ele checa igualdade de arrays e objetos corretamente (não somente por referência)
- **Teste é bem sucedido se o valor do *expect* e do *matcher* são iguais**

Termos no Jest - *toContain* 

- Usado para **checar se um array contém um valor**
- Teste é bem sucedido se o valor do *matcher* é encontrado dentro do array passado para o *expect*

Termos no Jest - *toMatchObject*

- Usado para **checar se um objeto tem certas propriedades**
- Teste é bem sucedido se o objeto passado para o matcher é **um subconjunto das propriedades** do objeto passado para o *expect*

Termos no Jest - *Matchers - toHaveLength*

- Usado para **checar se um array tem um determinado valor**
- Teste é bem sucedido se o array passado para o *expect* possui o tamanho passado para o *matcher*

Termos no Jest - *Matchers - Matchers - toBeGreaterThen / LessThen*

- Usados para **checar se o valor é maior ou menor que um número específico**

- Teste é bem sucedido se o número passado para o *expect* for maior/menor que o passado para o *matcher*

Termos no Jest - not

- **.not.**
  - Permite fazer uma validação "negativa": se o valor não é outro; se o array não contém um elemento; etc

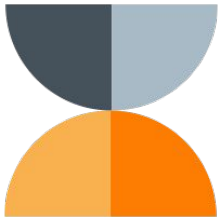
```
describe("Testing validateEmptyProperties", () => {
  test("Testing not", async () => {
    const result =  testingFunction();

    expect(result).not.toBe(0);
    expect(result).not.toEqual(1);
    expect(result).not.toContain("teste");

  });
});
```

Pausa para relaxar 😴

5 min



- Para nos a**uxiliar a escrever e executar testes, usamos frameworks de teste**. Em JS, vamos usar o Jest.
- O Jest possui a função **expect**, que facilita a verificação de diversos tipos de valores.
- Ela é usada com ***matchers***, que determinam se o teste é bem ou mal sucedido.

Exercício 1

- Escreva testes para verificar se função que retorna maior número de um array funciona

- Escrever testes para verificar se função que remove números duplicados de um array funciona

Resumo

Labenu_

Resumo - Teoria 📓

- Um teste serve para **verificar** se algo **funciona de acordo** com uma **especificação**
- Testes unitários são responsáveis por **testar pequenas partes do código**
- Para nos **auxiliar a escrever e executar testes, usamos frameworks de teste.** Em JS, vamos usar o Jest.

Resumo - Matchers do Jest 📘

toMatchObject	Testa se há propriedades no objeto
toBe / toEqual	Verifica igualdade
toContain	Testa se há um valor no array
toHaveLength	Verifica comprimento do array
toBeGreaterThan	Testa se um valor é maior
toLessThan	Testa se um valor é menor

Dúvidas? 🧐

Labenu_

Obrigado(a)!