Relatório de Testes com o Framework Jest

1. Descrição do Framework

Jest é um framework de testes em JavaScript desenvolvido e mantido pela Meta (anteriormente Facebook), amplamente adotado como padrão na indústria para projetos JavaScript e TypeScript. Sua popularidade deriva de uma filosofia que combina simplicidade, performance e um conjunto robusto de funcionalidades integradas, minimizando a necessidade de configuração e de bibliotecas externas.

O principal objetivo do Jest é fornecer uma experiência de teste "out-of-the-box". Ele inclui, em uma única instalação, um executor de testes (test runner), uma biblioteca de asserções (assertions) e um framework de mocks, eliminando a complexidade de integrar múltiplas ferramentas.

Principais Características:

- **Configuração Mínima:** Jest é projetado para funcionar na maioria dos projetos JavaScript sem a necessidade de arquivos de configuração extensos. Ele detecta automaticamente os arquivos de teste no projeto.
- **Assertions Integradas:** Oferece uma API de asserções rica e expressiva através da função `expect()`, com uma vasta gama de "matchers" (`.toBe()`, `.toEqual()`, `.toHaveBeenCalled()`, etc.) que tornam os testes legíveis e fáceis de escrever.
- **Mocks e Spies:** Possui um poderoso sistema de mocking integrado que permite isolar o código sob teste. É possível mockar módulos inteiros, classes ou funções individuais, além de espionar chamadas de função para verificar seu comportamento.
- **Relatórios de Cobertura de Código:** Gera relatórios detalhados de cobertura de código nativamente, identificando partes do código que não foram testadas e ajudando a garantir a qualidade do software.
- **Execução em Paralelo:** Para otimizar a performance, o Jest executa testes em processos paralelos e isolados, reduzindo significativamente o tempo total de execução da suíte de testes.

A estrutura de um teste em Jest é organizada e intuitiva, geralmente utilizando blocos `describe` para agrupar testes relacionados e blocos `test` (ou `it`) para definir os casos de teste individuais.

```
// Estrutura básica de um teste Jest
describe("Grupo de testes para a Calculadora", () => {
  // Bloco executado antes de cada teste deste grupo
  beforeEach(() => {
    // Ex: inicializar uma nova instância da calculadora
  });

  test("deve somar dois números positivos corretamente", () => {
    // Arrange (Preparação)
    const num1 = 2;
    const num2 = 3;

    // Act (Execução)
    const resultado = minhaFuncaoDeSoma(num1, num2);

    // Assert (Verificação)
    expect(resultado).toBe(5);
  });
});
```

---

2. Categorização do Framework

O Jest pode ser classificado sob múltiplas perspectivas de teste:

i) Técnicas de Teste

Sob a perspectiva de técnicas, o Jest é primariamente uma ferramenta para **Testes de Caixa Branca**.

- **Argumento:** Os testes escritos com Jest têm acesso direto ao código-fonte, incluindo classes, módulos e funções. Os desenvolvedores criam testes que conhecem a estrutura interna do software, permitindo testar caminhos lógicos específicos (branches) dentro de uma função. Por exemplo, ao testar a função de divisão, criamos casos específicos para a condição `if (divisor === 0)`, o que demonstra conhecimento da implementação interna.

ii) Níveis de Teste

Considerando os níveis de teste, o Jest se destaca principalmente em:

- **Testes de Unidade:** Este é o ponto mais forte do Jest. Sua arquitetura, com mocks integrados e um executor rápido, é ideal para isolar e testar a menor parte funcional do código (uma função ou uma classe) de forma independente. O projeto demonstra isso perfeitamente, com um arquivo de teste dedicado (`Calculadora.test.js`) que verifica a classe de lógica de negócio em total isolamento.
- **Testes de Integração:** Embora seja excelente para testes de unidade, o Jest também pode ser utilizado para testes de integração, verificando a interação entre diferentes módulos.

iii) Tipos de Teste

Em relação aos tipos de teste, o Jest suporta principalmente:

- **Testes Funcionais:** Os 24 testes implementados para a classe `Calculadora` são funcionais. Eles verificam se uma funcionalidade específica (soma, subtração, etc.) se comporta conforme o esperado, validando as saídas com base em um conjunto de entradas. Exemplo: `expect(calculadora.sumar(2, 3)).toBe(5)`.
- **Testes de Regressão:** A suíte de testes como um todo funciona como uma rede de segurança contra regressões. Ao executar os 24 testes após cada alteração no código, garantimos que as funcionalidades existentes não foram quebradas, prevenindo a reintrodução de bugs.

---

3. Forma de Instalação/Integração do Framework

A integração do Jest em um projeto JavaScript é simples e direta.

Passo 1: Pré-requisitos Certifique-se de ter o Node.js e o npm (ou Yarn) instalados no seu ambiente de desenvolvimento.

Passo 2: Instalação das Dependências No terminal, na raiz do projeto, execute o seguinte comando para instalar o Jest:

```
npm install --save-dev jest jest-environment-jsdom
```

(Nota: `jest-environment-jsdom` é mantido para compatibilidade, embora os testes de lógica pura não interajam com o DOM).

Passo 3: Configuração no `package.json` Adicione a seção `jest` ao seu arquivo `package.json` para configurar o ambiente de teste e a coleta de cobertura de código.

```
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watchAll",
    "test:coverage": "jest --coverage"
  },
  "devDependencies": {
    "jest": "^29.7.0",
    "jest-environment-jsdom": "^29.7.0"
  },
  "jest": {
    "testEnvironment": "jsdom",
    "collectCoverageFrom": ["Calculadora.js"]
  }
}
```

Passo 4: Execução dos Testes Com a configuração pronta, utilize os scripts definidos no `package.json` para rodar os testes:

```
# Executar todos os testes uma única vez
npm test

# Executar testes com o relatório de cobertura
npm run test:coverage
```

---

4. Estratégias para Derivação de Casos de Teste

Para derivar os casos de teste da classe `Calculadora`, foram aplicadas as estratégias de **Particionamento de Equivalência** e **Análise de Valor Limite**. Demonstraremos o processo usando como exemplo a função `dividir(numerador, denominador)`.

Contexto: A função de divisão possui comportamentos distintos dependendo dos valores de entrada, especialmente em relação ao zero.

Passo 1: Identificação das Partições de Equivalência

Identificamos as classes de entradas que deveriam produzir resultados similares. Para numerador e denominador, as partições são:

- Números positivos
- Números negativos
- Zero

Passo 2: Análise de Valor Limite e Casos Especiais

A análise de valor limite foca nos "limites" das partições e em casos especiais que podem levar a erros. Para a divisão, o valor mais crítico é o **zero**.

- **Limite:** Denominador igual a zero (divisão por zero).
- **Caso Especial 1:** Numerador igual a zero.
- **Caso Especial 2:** Numerador e denominador iguais a zero.

Passo 3: Derivação e Combinação dos Casos de Teste

Combinando as partições e os casos especiais, derivamos os seguintes casos de teste:

1. **Cenário: Divisão Padrão (Positivo / Positivo)**
    - **Partição:** Numerador Positivo, Denominador Positivo.
    - **Caso de Teste:** `dividir(10, 2)`
    - **Resultado Esperado:** `5`
2. **Cenário: Divisão com Resultado Negativo (Negativo / Positivo)**
    - **Partição:** Numerador Negativo, Denominador Positivo.
    - **Caso de Teste:** `dividir(-10, 2)`
    - **Resultado Esperado:** `-5`
3. **Cenário: Divisão por Zero (Valor Limite)**
    - **Partição:** Qualquer Numerador, Denominador Zero.
    - **Caso de Teste 1:** `dividir(5, 0)` -> `Infinity`
    - **Caso de Teste 2:** `dividir(-5, 0)` -> `-Infinity`
4. **Cenário: Zero Dividido por um Número (Caso Especial)**
    - **Partição:** Numerador Zero, Denominador Não-Zero.
    - **Caso de Teste:** `dividir(0, 5)`
    - **Resultado Esperado:** `0`
5. **Cenário: Zero Dividido por Zero (Caso Especial)**
    - **Partição:** Numerador Zero, Denominador Zero.
    - **Caso de Teste:** `dividir(0, 0)`
    - **Resultado Esperado:** `NaN` (Not a Number)

Passo 4: Implementação dos Casos de Teste em Jest

Os casos derivados foram traduzidos diretamente para testes no arquivo `Calculadora.test.js`, validando cada comportamento esperado.

```
// Código extraído do relatório original
describe("Operação de Divisão", () => {
    test("deve dividir dois números positivos", () => {
        expect(calculadora.dividir(10, 2)).toBe(5); // Cenário 1
    });

    test("deve retornar infinito ao dividir por zero", () => {
        expect(calculadora.dividir(5, 0)).toBe(Infinity); // Cenário 3
        expect(calculadora.dividir(-5, 0)).toBe(-Infinity); // Cenário 3
    });

    test("deve retornar NaN ao dividir zero por zero", () => {
        expect(calculadora.dividir(0, 0)).toBeNaN(); // Cenário 5
    });

    test("deve retornar 0 ao dividir zero por um número", () => {
        expect(calculadora.dividir(0, 5)).toBe(0); // Cenário 4
    });
});
```