# Aula 6 - Testes

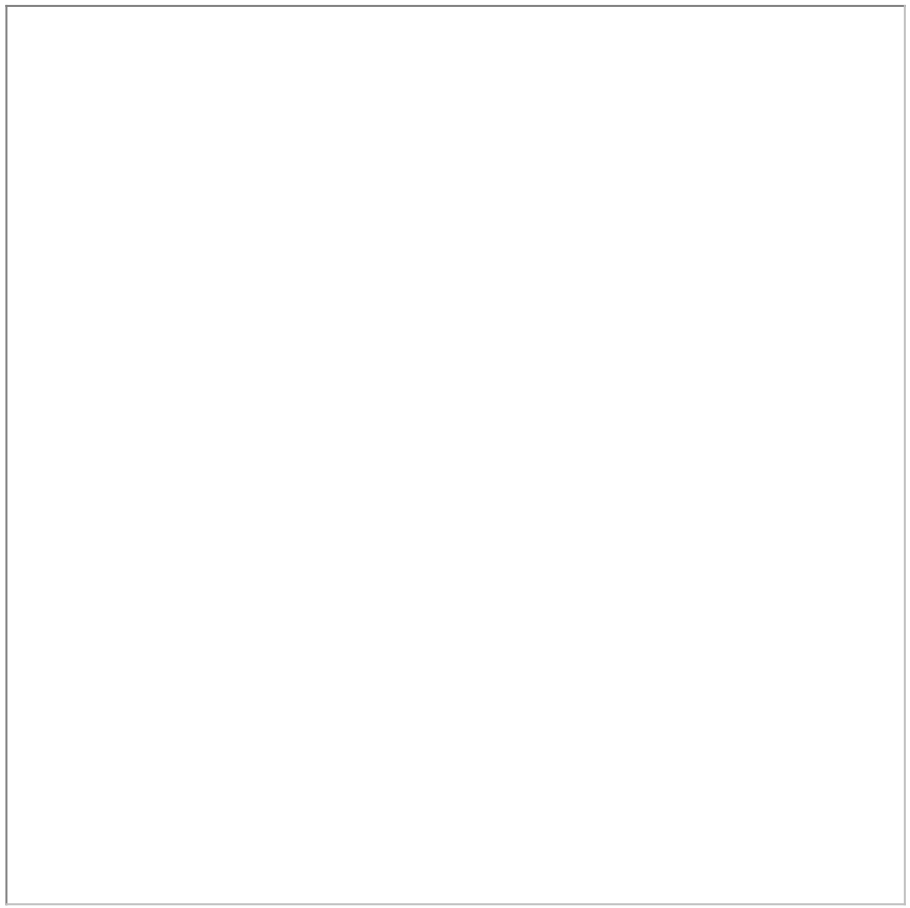
**Docupedia Export**

Author:Ferro Alisson (CtP/ETS)
Date:22-Aug-2023 15:16

# Table of Contents

Para concluir o nosso treinamento iremos ver agora um poucos sobre os testes e como implementa-los.
O que são testes de software?
Testes de software são uma atividade fundamental no processo de desenvolvimento de software, que visa verificar e validar se um sistema ou aplicativo funciona conforme o esperado. Essa prática envolve a execução de casos de teste específicos com o objetivo de identificar defeitos, erros, falhas ou inconsistências no software, garantindo que ele esteja livre de problemas antes de ser lançado para uso.
Existem diversos tipos de teste de software, cada um com um objetivo específico e foco em determinados aspectos do sistema. Como por exemplo:

- Teste de Unidade (Unit Testing): Realizado para verificar se cada unidade individual de código (como funções ou métodos) funciona corretamente de forma isolada.
- Teste de Integração (Integration Testing): Verifica se as unidades de código funcionam de forma integrada, ou seja, se elas se comunicam e trabalham corretamente em conjunto.
- Teste de Sistema (System Testing): Testa o sistema completo como uma entidade única, verificando se todas as partes integradas funcionam conforme o esperado.
- Teste de Aceitação (Acceptance Testing): Realizado pelos usuários finais ou stakeholders para validar se o sistema atende aos requisitos e expectativas definidos para ele.
- Teste Funcional (Functional Testing): Verifica se as funcionalidades do sistema estão de acordo com os requisitos definidos e se ele executa as tarefas esperadas.
- Teste de Desempenho (Performance Testing): Avalia o desempenho do sistema em termos de velocidade, resposta, eficiência e escalabilidade.
- Teste de Segurança (Security Testing): Verifica a segurança do sistema, identificando vulnerabilidades e garantindo que ele proteja os dados e informações confidenciais.
- Teste de Usabilidade (Usability Testing): Avalia a facilidade de uso e a experiência do usuário do sistema.
- Teste de Regressão (Regression Testing): Realizado após modificações no sistema para garantir que as alterações não tenham impactado negativamente funcionalidades já existentes.
- Teste de Carga (Load Testing): Avalia o comportamento do sistema sob carga e verifica como ele se comporta em situações de alta demanda.
- Teste de Estresse (Stress Testing): Avalia os limites e capacidades do sistema, submetendo-o a condições extremas e verificando como ele responde.



Assim temos a pirâmide de testes, onde a maioria dos testes deveriam ser Unit Testing por se tratarem de menor custo e maior velocidade, acima temos o integration testing, que são testes mais custosos se comparar com unit e um pouco mais lentos, e por fim testes end to end, que são testes de interface completo, onde testa-se a interface junto com API e banco de dados, que são testes mais caros para fazer.

Nesse treinamento iremos abordar Unit Testing.
Para inicial vamos utilizar o Jest, um framework de testes de testes de código aberto, desenvolvida pelo Facebook, que é especialmente projetada para testar aplicações em JavaScript. Ele é amplamente utilizado na comunidade de desenvolvedores para testar aplicações frontend, bibliotecas, componentes React, entre outros projetos que utilizam JavaScript

Instalando o Jest

Para instalar o jest na nossa aplicação é bem simples.

```
npm i --dev jest
```

Vamos utilizar o --dev pois o jest é nossa dependencia de desenvolvimento.

Para iniciar, vamos alterar o nosso package.json e substituir a linha do script de teste.

```
"test": "jest --testEnvironment=node --watchAll --verbose --coverage",
```

Agora vamos criar um arquivo de uma calculadora para iniciarmos. Como se trata de testar métodos do nosso sistema, será um unit test.
Vamos criar um arquivo em **src/commom/calc.js**

```
function Soma(a,b){
    return a + b;
}

module.exports = { Soma }
```

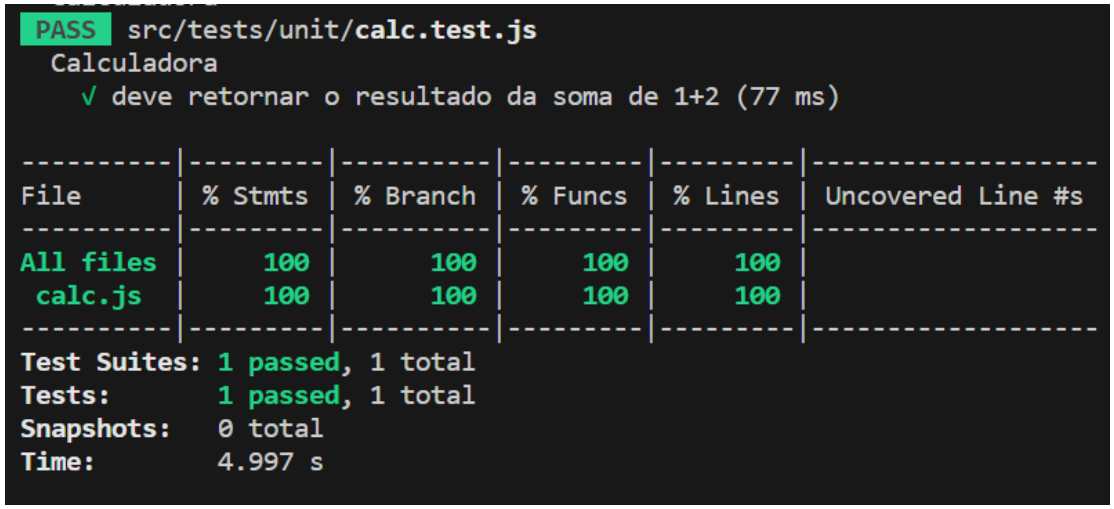
Como é um calculo simples, podemos verificar facilmente que está correto, mas podemos testar quaisquer métodos que achamos necessário, desde cálculos simples até cálculos mais complexos.
Para criar um teste vamos criar um arquivo em **src/tests/unit/calc.test.js**

```
const { Soma } = require("../../commom/calc")

describe('Calculadora', () => {
    it('deve retornar o resultado da soma de 1+2', () => {
        const res = Soma(1,2);
        expect(res).toBe(3);
    })
})
```

Agora para rodar o teste basta no terminar rodarmos o comando

```
npm run test
```

E assim será rodado nosso primeiro caso de teste



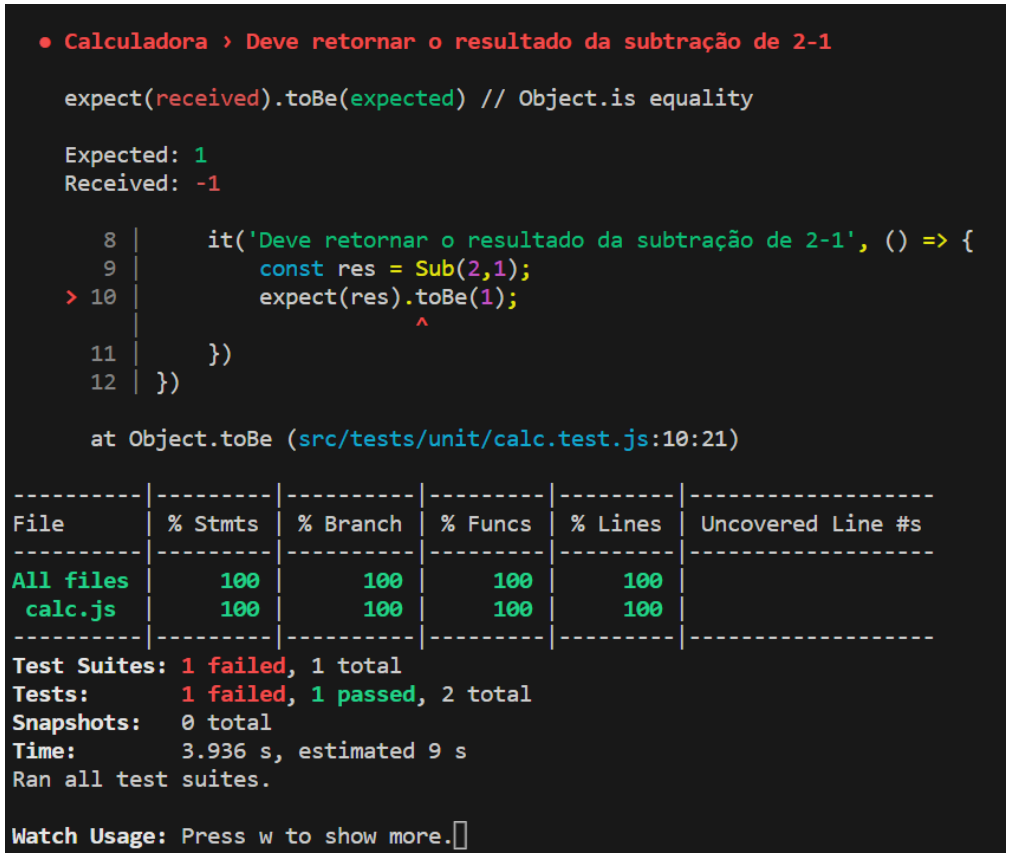

Podemos ver que esta passando 1 teste do total de 1 teste.
Agora vamos criar mais uma função só que para subtrair.

```
function Soma(a,b){
    return a + b;
}

function Sub(a,b){
    return a-b;
}

module.exports = { Soma, Sub }
```

E rodando os teste temos como resultado

```
  Calculadora
    √ deve retornar o resultado da soma de 1+2 (72 ms)
    √ Deve retornar o resultado da subtração de 2-1 (2 ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 calc.js  |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        9.098 s
Ran all test suites.
```

Caso alguém por ventura fosse alterar o código e sem perceber deixasse return b-a na função Sub, perderíamos um tempo para que conseguisse descobrir onde estaria o erro, e as vezes poderia ir para produção esse bug.
Sabemos que a-b é diferente de b-a, mas lembrando que poderia ser um calculo muito mais complexo e que não ficaria tão óbvio assim ao olhar já encontrar o bug
Vamos fazer essa alteração e ver o que acontece




Nosso resultado diz que 1 teste falhou e que esperava receber 1 e recebeu -1 da função Sub. Sendo assim, nosso bug já esta mais evidente, sabemos que foi um bug na função Sub.
Vamos voltar a função como estava anteriormente.

# 1 Desafio 1: Termine a calculadora com as funções multiplicação e divisão

# 2 Desafio 2: com base no código abaixo, faça testes unitários para o código e descubra onde esta o erro na função e refatore para todos os testes passarem.

```
function validaCpf(cpf){
    if(!cpf) return false;

    const cpfSplit = cpf.replaceAll('.', '').replace('-','')
    if(cpfSplit.length != 11) return false

    var isSequencial = false
    for(let i = 1; i<cpfSplit.length; i++){
        if(cpfSplit[i] == cpfSplit[i-1])
            isSequencial = true
    }

    if(isSequencial) return false

    var somaDig1 = 0;
    for(let i=0; i<cpfSplit.length-2; i++){
        somaDig1 += (Number(cpfSplit[i])*(10 - i));
    }

    if((11 - (somaDig1 % 11)) != cpfSplit[9]) return false

    var somaDig2 = 0;
    for(let i=0; i<cpfSplit.length-1; i++){
        somaDig2 += Number(cpfSplit[i])*(11 - i);
    }

    if((11-(somaDig2 %11)) != cpfSplit[10]) return false

    return true
}
```

```
module.exports = { validaCpf }
```