# Teste Unitário

Material de Referência [aws.amazon.com](https://aws.amazon.com/pt/what-is/unit-testing)

## O que é Teste Unitário?

Teste unitário é a prática de testar partes pequenas e isoladas do código, como funções ou métodos, para garantir que elas funcionem corretamente de forma independente.

## Prós

* Identificação rápida de erros e problemas em partes específicas do código.
* Facilita a manutenção e evolução do software.
* Melhora a qualidade geral do código e aumenta a confiança ao fazer mudanças.
* Permite automação, executando testes automaticamente em cada alteração do código.
* Incentiva a escrita de código modular e bem estruturado.

## Contras

* Podem exigir um tempo inicial maior para escrever e manter os testes.
* Cobrem apenas partes isoladas, não garantindo que o sistema completo funcione (precisa de outros tipos de teste).
* Testes mal escritos podem gerar falso senso de segurança.

## Exemplo

Exemplo pelo Chat GPT

/// função a ser testada

function soma(a, b)

{

return a + b;

}

/// teste unitário usando Jest

test('soma 1 + 2 é igual a 3', function ()

{

expect(soma(1, 2)).toBe(3);

});

# Teste de Integração

Material de Referência [www.ibm.com](https://www.ibm.com/br-pt/think/topics/integration-testing)

## O que é Teste de Integração

É uma abordagem de teste que junta e testa diversos componentes ou módulos da aplicação para ver se eles funcionam bem em conjunto. Verifica se se comunicam corretamente e interagem como esperado. Ideal pra empresas que lidam com o público e lançam apps novos - é fundamental para garantir uma boa primeira impressão e preservar a credibilidade da empresa.

## Prós

* Garante que diferentes partes do sistema funcionem bem juntas importante para a experiência do usuário e imagem da empresa.
* Ajuda a detectar falhas de interface, fluxo de dados e comunicação que não apareceriam em testes isolados.
* Inserido na sequência de testes, evita que erros se acumulem até os testes de sistema.

## Contras

* Empresas pequenas podem achar que nem precisam fazer testes de integração.
* É um processo mais complexo: exige identificar pontos de integração, criar cenários realistas e considerar dependências - não dá para ser rápido ou simples.

## Exemplo

Exemplo pelo Chat GPT

/// suponha que temos dois módulos:

/// math.js

export function add(a, b)

{

return a + b;

}

/// logger.js

export function log(result)

{

console.log('Resultado:', result);

}

/// teste.js (teste de integração)

import { add } from './math.js';

import { log } from './logger.js';

test('integração add + log', function ()

{

const result = add(2, 3);

expect(result).toBe(5);

/// só pra poder testar sem ficar na console

const spy = jest.spyOn(console, 'log').mockImplementation();

log(result);

expect(spy).toHaveBeenCalledWith('Resultado:', 5);

spy.mockRestore();

});

# Teste de Fumaça

Material de Referência [www.testbytes.net](https://www.testbytes.net/blog/smoke-testing-explanation-example)

## O que é

É um tipo de build verification testing, um teste rápido para checar se os recursos principais de um software estão funcionando - tipo registrarse, login, navegar, adicionat ao carrinho, pagar, que nem um “pré‑voo” da aplicação, garantindo que está estável para testes mais profundos.

## Prós

* Detecta falhas logo no início do ciclo, evitando desperdício de tempo em builds inválidas.
* Evita que testadores percam horas em algo que já está quebrado.
* Confere confiança para avançar com os testes.
* Identifica problemas de integração rapidinho.
* Executa em poucos minutos, proporcionando feedback rápido.
* Pode ajudar a identificar falhas de alta gravidade mais cedo.

## Contras

* Só cobre “funcionalidades básicas” para validar a build.
* Não testa tão profundamente.

## Exemplo

Exemplo pelo Chat GPT

test('login smoke test', async function()

{

await page.goto('...login');

await page.fill('#user', 'meuUser');

await page.fill('#pass', 'minhaSenha');

await page.click('#loginBtn');

expect(await page.url()).toContain('/dashboard');

});

test('video launch smoke test', async function()

{

await page.goto('/video');

await page.click('#playBtn');

const visible = await page.isVisible('#videoPlayer');

expect(visible).toBe(true);/// falha se der falso

});

# Teste de Caixa Branca

Material de Referência [www.browserstack.com](https://www.browserstack.com/guide/white-box-testing)

## O que é

É um método de teste onde você conhece o código por dentro - design, lógica, estrutura - e usa isso para criar casos de teste que garantam que o que está ali faz o que deveria. Testa fluxos, loops, estruturas condicionais e até vulnerabilidades internas. Pode ser feito em níveis de unidade, integração e sistema.

## Prós

* Visibilidade total do código: dá para testar caminhos específicos, garantir cobertura de instruções.
* Pode começar cedo no desenvolvimento, antes da interface existir.
* Testes são fáceis de automatizar, dando eficiência.

## Contras

* É complexo e caro: requer quem entende bem do código, e escrever/manter testes exige esforço.
* Perde o ponto de vista do usuário: por olhar “de dentro”, deixa de testar como o usuário realmente interage.
* Quando o código muda, os testes quebram e precisam ser reescritos - manutenção pesada.

## Exemplo

Exemplo pelo Chat GPT

function soma(a, b)

{

const resultado = a + b;

if (resultado > 0)

{

return 'positive: ' + resultado;

}

else

{

return 'negative: ' + resultado;

}

}

/// casos de teste em white‑box

test('soma positiva', function ()

{

expect(soma(1, 1)).toBe('positive: 2');

});

test('soma negativa', function ()

{

expect(soma(-1, -3)).toBe('negative: -4');

});

# Teste de Caixa Preta

Material de Referência [www.imperva.com](https://www.imperva.com/learn/application-security/black-box-testing)

## O que é

É o tipo de teste onde você não tem nenhuma noção do que rola por dentro do sistema. Você manda um input e observa o output. A ideia é simular como um usuário (ou invasor) externo interage com o sistema, testando tudo - UI, servidor, banco, integrações etc.

## Prós

* Não precisa saber programar ou entender o sistema.
* Sem precisar conhecer detalhes internos.
* Pode ser feito por testadores terceirizados ou crowdsourced.
* Menor chance de falso positivo.
* Testes são mais simples porque simulam ações comuns dos usuários.

## Contras

* Difícil automatizar completamente.
* Impossível cobrir todos os caminhos possíveis.
* Difícil medir a cobertura do teste.
* Quando falha, entender a causa raiz é complicado.
* Pode não refletir o ambiente real de produção (teste em escala reduzida).

## Exemplo

Exemplo pelo Chat GPT

/// função a ser testada (a gente não vê o código interno, lembrando do 'black box')

function login(username, password)

{

// ...secreto

}

/// testes (input → output esperado)

console.assert(login('admin', 'senha\_certa') === true, 'deve permitir login com credenciais válidas');

console.assert(login('admin', 'senha\_errada') === false, 'deve negar login com senha inválida');

console.log('testes de caixa preta concluídos!');