Testes em Next.js com Jest

Objetivos do Treinamento

Tipos de Testes

Compreender os diferentes tipos de testes aplicáveis ao Next.js e quando utilizar cada um

Configuração Jest

Configurar Jest em projetos Next.js com App Router e TypeScript de forma profissional

Testes Unitários

Criar testes unitários eficientes para funções e lógica de negócio

Testes de Componentes

Testar componentes React usando Testing Library com boas práticas

Server Components

Testar Server Components e Server Actions do Next.js 13+

Boas Práticas

Aplicar estratégias de cobertura e uso de mocks para testes robustos

Agenda do Treinamento

1

Introdução aos Testes

5 minutos — Fundamentos e importância dos testes em aplicações Next.js

2

Configuração do Jest

10 minutos — Setup completo do ambiente de testes com TypeScript

3

Testes Unitários

5 minutos — Testando funções puras e lógica de negócio

Testes de Componentes

4

10 minutos — Testing Library e testes de interface do usuário

Server Components e Actions

5

5 minutos — Estratégias para testar recursos server-side do Next.js

6

Boas Práticas e Cobertura

5 minutos — Mocks, cobertura de código e qualidade de testes

O que Testar no Next.js

Em aplicações Next.js modernas, diversos elementos precisam de cobertura de testes para garantir qualidade e confiabilidade. É essencial identificar o que realmente importa testar.

Funções Utilitárias

Lógica pura e transformações de dados

Componentes Client

Componentes React com "use client"

Componentes Server

Server Components do Next.js 13+

Server Actions

Funções server-side com "use server"

Hooks Customizados

Custom hooks com lógica reutilizável

API Interna

Rotas em app/api e handlers

Regras de Negócio

Validações e lógica crítica do domínio

Tipos de Teste

Diferentes tipos de testes servem propósitos distintos na pirâmide de testes. Conhecer cada categoria ajuda a construir uma estratégia de qualidade eficiente.

2
3

4

1 E2E

2 Integração

3 Componentes

4 Unitários

Testes Unitários

Testam funções e métodos isoladamente. Rápidos de executar e fáceis de manter. Formam a base da pirâmide.

Testes de Componentes (UI)

Verificam renderização e interações de componentes React. Usam Testing Library para simular o comportamento do usuário.

Testes de Integração

Validam a comunicação entre múltiplos módulos. Testam fluxos completos e integração com APIs.

Testes E2E

Simulam jornadas reais do usuário. Utilizam ferramentas como Playwright ou Cypress para validar a aplicação completa.

Por que usar Jest

Jest se consolidou como o framework de testes mais popular para aplicações JavaScript e TypeScript. Sua integração nativa com React e Next.js, combinada com um ecossistema maduro, torna-o a escolha ideal para projetos modernos.

Desenvolvido pelo Facebook, Jest oferece uma experiência de desenvolvimento excepcional com recursos avançados de mocking, cobertura de código integrada e execução paralela de testes.

Configuração Simples

Setup mínimo e zero-config

Suporte Node + React

Funciona perfeitamente com ambos

Mocks Poderosos

Sistema de mocking robusto

Alto Desempenho

Execução rápida e paralela

Testing Library

Integração perfeita

Configurando Jest — Passo 1

Instalação das Dependências

O primeiro passo é instalar todas as bibliotecas necessárias para configurar o ambiente de testes. Este comando instala o Jest, suporte para TypeScript, ambiente JSDOM para simular o DOM, e as bibliotecas da Testing Library para testes de componentes React.

```
npm i -D jest @types/jest ts-jest jest-environment-jsdom \
  @testing-library/react @testing-library/jest-dom
```

jest + @types/jest

Framework de testes e tipagens TypeScript

ts-jest

Transformador para executar testes TypeScript

jest-environment-jsdom

Simula ambiente de navegador para testes

@testing-library/react

Utilitários para testar componentes React

@testing-library/jest-dom

Matchers customizados para assertions do DOM

Configurando Jest — Passo 2

Arquivo jest.config.js

Crie o arquivo de configuração principal do Jest na raiz do projeto. Este arquivo define como o Jest deve processar os arquivos, resolver imports e preparar o ambiente de testes.

```
module.exports = {
  preset: "ts-jest",
  testEnvironment: "jest-environment-jsdom",
  moduleNameMapper: {
    "^@/(.*)$": "<rootDir>/$1",
  },
  setupFilesAfterEnv: ["<rootDir>/jest.setup.ts"],
};
```

preset: "ts-jest"

Configura Jest para processar TypeScript automaticamente

testEnvironment

Define JSDOM como ambiente para simular navegador

moduleNameMapper

Mapeia alias @/ para resolver imports corretamente

setupFilesAfterEnv

Arquivo executado antes de cada teste para configurações globais

Configurando Jest — Passo 3

Arquivo jest.setup.ts

Este arquivo é executado antes de cada teste e serve para configurações globais. Aqui importamos os matchers customizados da Testing Library, que adicionam assertions úteis como `toBeInTheDocument()` e `toHaveTextContent()`.

O arquivo jest.setup.ts centraliza todas as configurações que devem estar disponíveis em todos os testes, evitando importações repetidas.

```
import '@testing-library/jest-dom';
```

Configurando Jest — Passo 4

Ajuste no tsconfig.json

Para que o TypeScript reconheça os alias de importação (como @/), é necessário configurar o mapeamento de paths no tsconfig.json. Isso garante que tanto o compilador TypeScript quanto o Jest resolvam os imports corretamente.

```
{
```

Importante: Se o seu projeto já possui outras

Rodando Testes

Executar Testes

Use o comando abaixo para executar todos os testes do projeto. O Jest irá buscar automaticamente por arquivos com extensões `.test.ts`, `.test.tsx`, `.spec.ts` ou `.spec.tsx`.

```
npm test
```

No modo watch, Jest reexecuta automaticamente os testes relacionados aos arquivos modificados:

```
npm test -- --watch
```

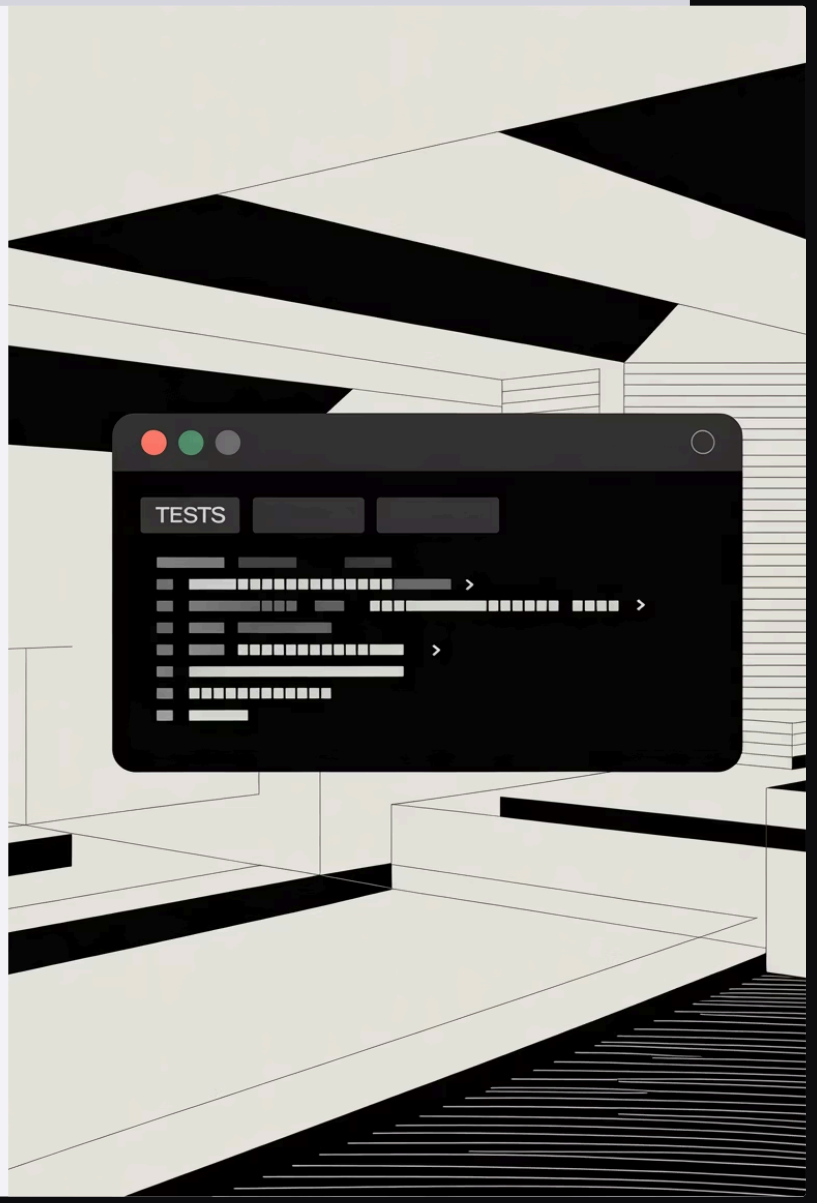
Cobertura de Código

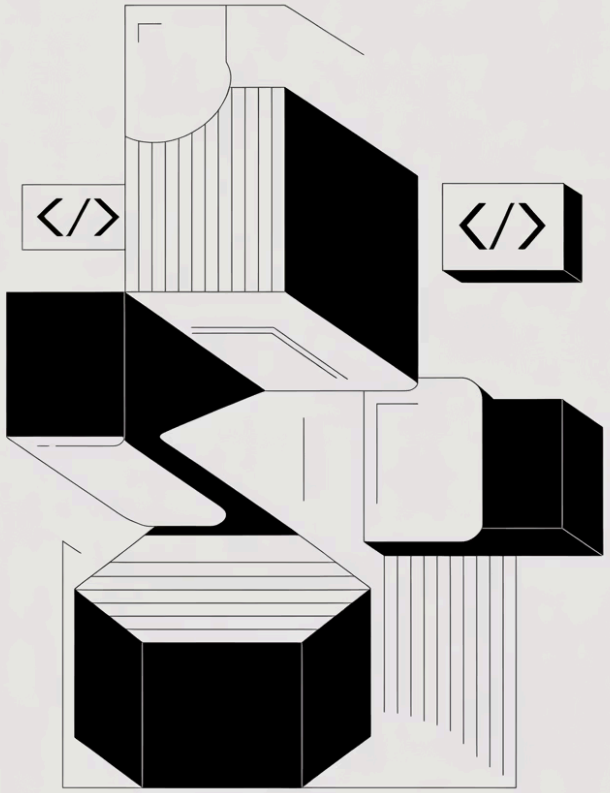
Para gerar relatórios de cobertura de código, execute o comando abaixo. Será criada uma pasta `coverage/` com relatórios detalhados em HTML.

```
npm run test:coverage
```

Adicione este script no `package.json`:

```
"scripts": {
 "test": "jest",
 "test:coverage": "jest --coverage"
}
```





Testes Unitários

Testes unitários focam em testar funções e lógica de negócio de forma isolada. São rápidos, confiáveis e formam a base da pirâmide de testes. Vamos criar uma função simples para demonstrar.

Função soma.ts

Uma função pura que recebe dois números e retorna sua soma. Funções puras são ideais para testes unitários pois não possuem efeitos colaterais.

```
export function soma(a:
number, b: number) {
  return a + b;
}
```

Características

- Função pura
- Sem dependências externas
- Fácil de testar
- Resultado previsível

Teste Unitário — Exemplo

Arquivo soma.test.ts

Testes unitários seguem uma estrutura simples: organize os testes em blocos `describe` e escreva casos de teste com `it` ou `test`. Use `expect` para fazer assertions sobre o resultado.

```
import { soma } from './soma';

describe("soma()", () => {
  it("soma dois números", () => {
    expect(soma(2, 3)).toBe(5);
  });
});
```

Import

Importa a função a ser testada

Describe

Agrupa testes relacionados

It/Test

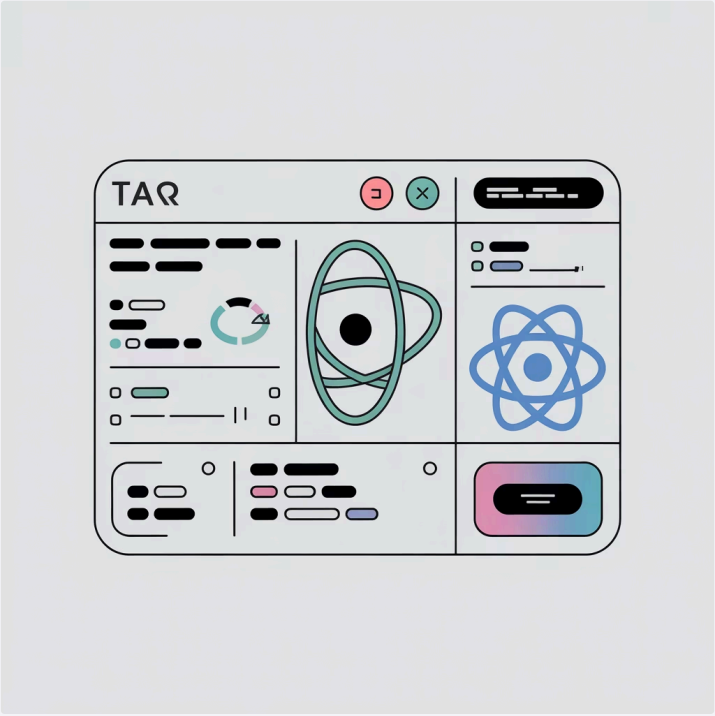
Define um caso de teste específico

Expect

Faz a assertion do resultado esperado

Boas práticas: escreva múltiplos casos de teste para cobrir cenários diferentes, incluindo casos extremos e validações de erro.

Testes de Componentes



Ferramentas Principais

Jest

Test runner e assertions

Testing Library

Renderização e queries

JSDOM

Ambiente DOM simulado

O que Testamos

Testes de componentes verificam a renderização e comportamento de componentes React do ponto de vista do usuário. A Testing Library incentiva boas práticas ao focar em como o usuário interage com a interface.

Renderização

Verifica se elementos aparecem corretamente na tela

Interações do Usuário

Simula clicks, digitação e outros eventos

Estado e Props

Valida comportamento baseado em estado e propriedades

Componente Button — Exemplo

Arquivo Button.tsx

Um componente Client simples que renderiza um botão clicável. A diretiva `"use client"` indica que este componente será executado no navegador e pode usar hooks e eventos.

```
"use client";

export function Button({ onClick, children }) {
  return <button onClick={onClick}>{children}</button>;
}
```

Este componente é um exemplo perfeito para testar interações do usuário. Ele recebe uma função `onClick` como prop e executa essa função quando o usuário clica no botão.

Teste do Button

Arquivo Button.test.tsx

Este teste verifica se o callback `onClick` é chamado quando o usuário clica no botão. Usamos `jest.fn()` para criar uma função mock que rastreia suas chamadas.

```
import { render, screen, fireEvent } from "@testing-library/react";
import { Button } from "./Button";

test("clicar no botão", () => {
  const handleClick = jest.fn();
  render(<Button onClick={handleClick}>Clique</Button>);
  fireEvent.click(screen.getByText("Clique"));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

01

Criar Mock

jest.fn() cria uma função rastreável

02

Renderizar

render() monta o componente no DOM virtual

03

Interagir

fireEvent.click() simula click do usuário

04

Verificar

expect() valida que o callback foi chamado

Componente com Estado

Arquivo Counter.tsx

Um contador que utiliza o hook `useState` para gerenciar estado local. Este exemplo demonstra como testar componentes que possuem estado interno e atualizam a UI em resposta a interações.

```
"use client";
import { useState } from "react";

export function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <p data-testid="count">{count}</p>
      <button onClick={() => setCount(count + 1)}>
        Incrementar
      </button>
    </>
  );
}
```

data-testid: Atributo especial usado pela Testing Library para selecionar elementos de forma confiável nos testes, sem depender de texto ou classes CSS.

Teste do Counter

Arquivo Counter.test.tsx

Este teste valida que o estado do componente é atualizado corretamente quando o usuário interage com ele. Verificamos se o contador incrementa ao clicar no botão.

```
import { render, screen, fireEvent } from "@testing-library/react";
import { Counter } from "./Counter";

test("incrementa contador", () => {
  render(<Counter />);
  fireEvent.click(screen.getByText("Incrementar"));
  expect(screen.getByTestId("count")).toHaveTextContent("1");
});
```

A Testing Library re-renderiza automaticamente o componente após cada interação, permitindo verificar as mudanças no DOM. Use `getByTestId` para selecionar elementos de forma confiável.

Para testar múltiplos cliques ou comportamentos assíncronos, você pode encadear múltiplas interações e assertions no mesmo teste.

Testando Server Components

Estratégia de Teste

Server Components do Next.js 13+ executam no servidor e retornam JSX. Para testá-los, precisamos mockar funções assíncronas e aguardar a resolução antes de renderizar.

Exemplo de Server Component

```
export default async function Page() {
  const data = await getUsers();
  return <div>{data.length} usuários</div>;
}
```

Mock de Funções Assíncronas

Use jest.mock() para substituir chamadas de API ou banco de dados

Await do Componente

Aguarde a resolução do componente antes de renderizar

Render com Await

Use render(await Component()) para testar o resultado final

Server Components não podem usar hooks ou event handlers. Eles são ideais para buscar dados e renderizar conteúdo no servidor.

Recursos Adicionais

Documentação do Jest

Referência completa da API, guias e exemplos práticos para dominar o framework de testes

Testing Library

Boas práticas e queries para testar componentes React focando na experiência do usuário

Next.js Testing Guide

Documentação oficial sobre testes em Next.js, incluindo Server Components e App Router

---

Próximos Passos

Continue praticando com testes de integração, explorando mocks avançados e implementando testes E2E com Playwright. A cobertura de testes é uma jornada contínua de aprendizado e refinamento.

Começar a Testar | Ver Exemplos Completos