Teste unitário – pega uma parte pequena da aplicação como uma função ou classe e checamos os parâmetros, retorno, integração e etc.

Nos testes vamos criar algo para simular a base de dados para não “sujarmos” a base de dados.

Esse objeto simulado se chama mock object e é utilizado para descrever um caso especial de objetos que imitam objetos reais para teste.

Não devemos confundir os Mocks com Stubs, que são objetos que auxiliam testes de ambientes.

Os Mock Objects são ainda mais utilizados no Behavior Driven Development (BDD) que se trata de uma variação do TDD e que utilizam testes com Mock Objects.

Ferramentas de auxilio para criação de Mocks

Para **Java** temos o JMockit, Mockito, EasyMock, JMock, MockCreator, MockLib e HibernateMock.

Para plataforma **.NET** temos o NMockLib, Rhino Mocks, NMock e NMock 2 TypeMock.

Para o **Ruby** temos o Mocha, RSpec e FlexMock.

Para linguagem **PHP** temos o SimpleTest, Yay! Mock, SnapTest e PHPUnit.

Para o **Delphi**, Kylix, FreePascal ou Pascal Mock.

No React vamos testar componentes para diversos testes como imagem, corpo de texto etc.

Os testes acontecem via código e o react app configura para gente com o **jest-dom**

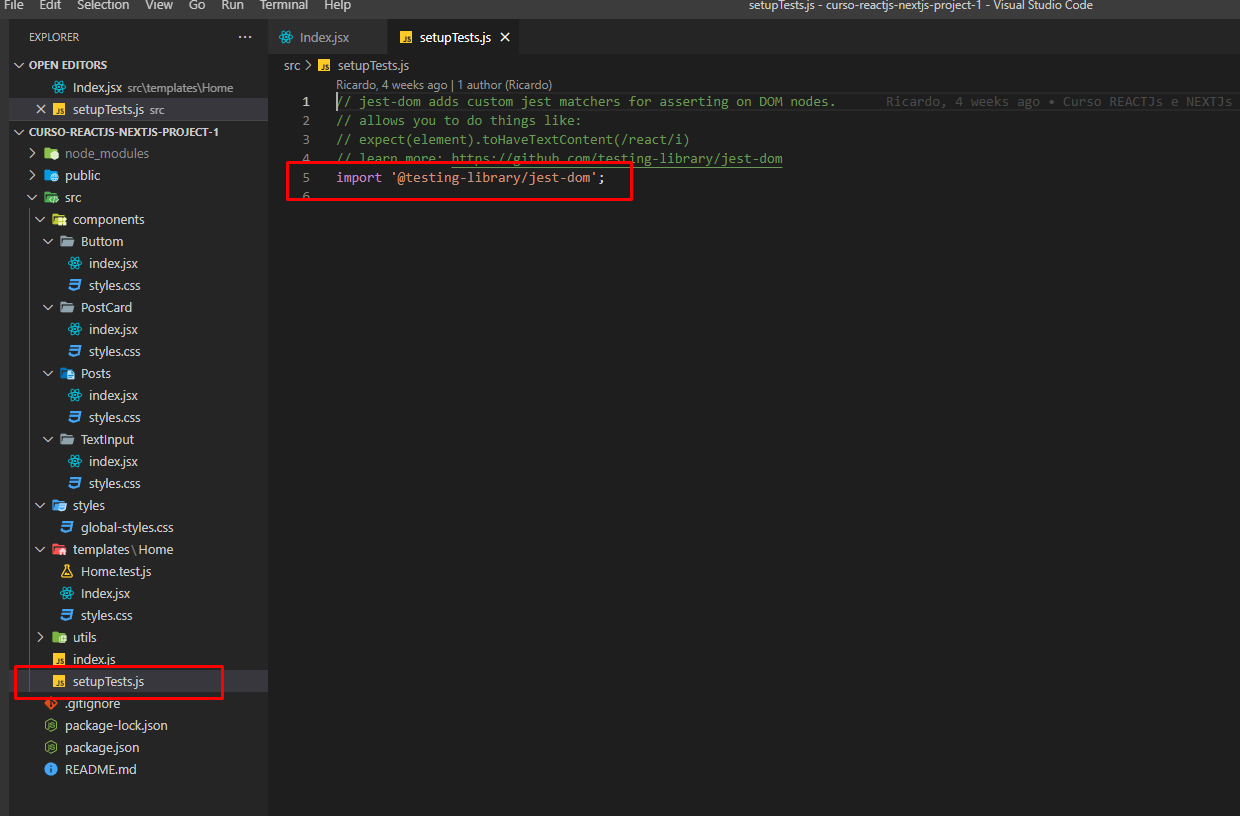
<https://github.com/testing-library/jest-dom>

<https://github.com/sapegin/jest-cheat-sheet>

O jest-dom renderiza dentro do código.

Não temos tela mas vemos a saída do código.

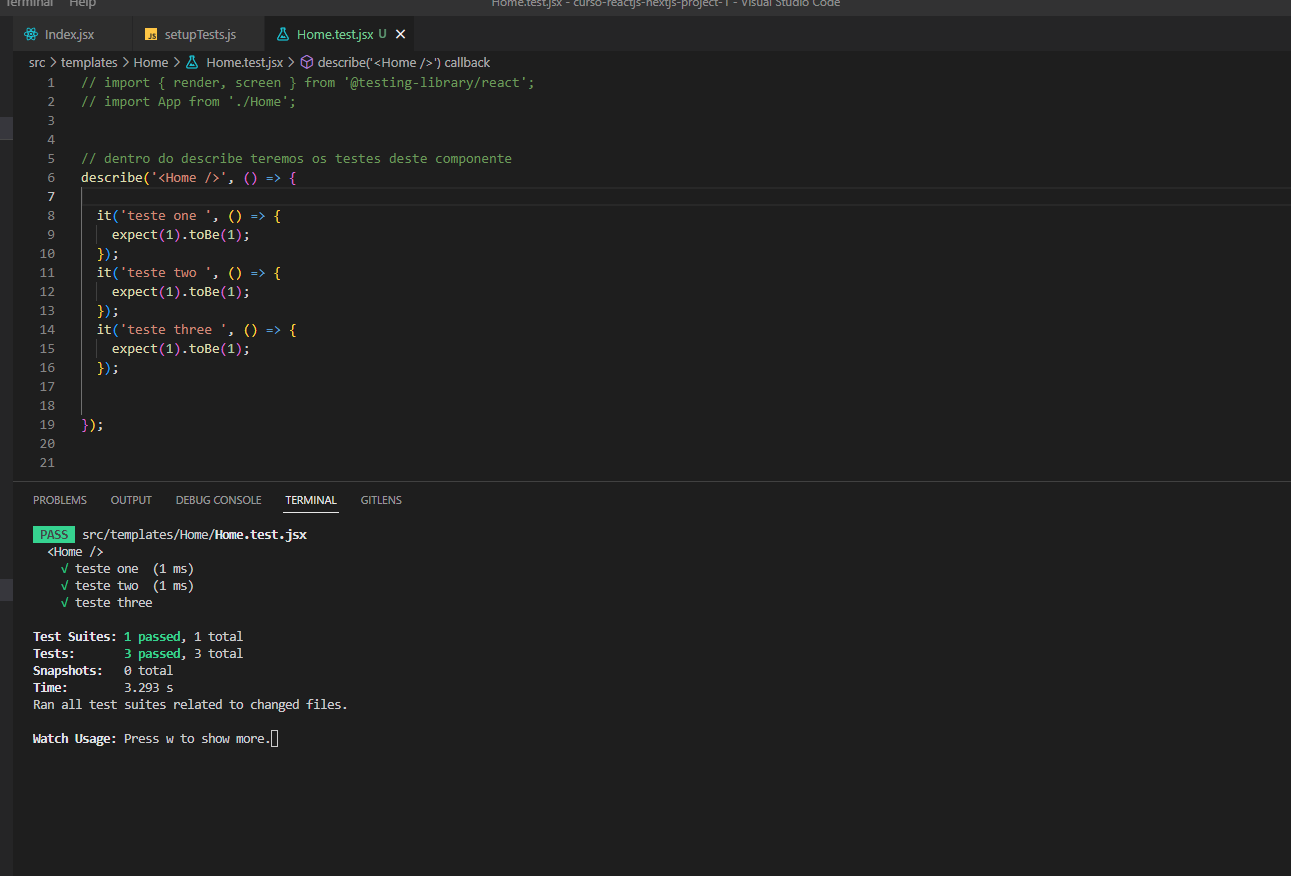
No react já temos configurada a chamada da library



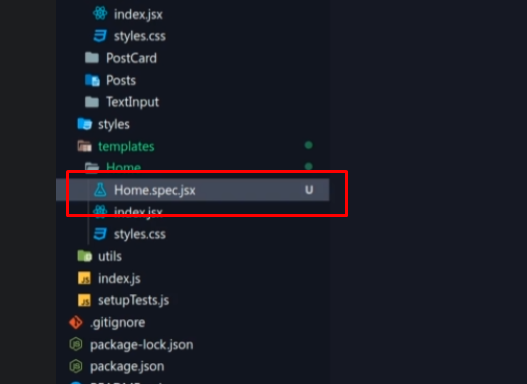
Trocamos a extensão do arquivo [Home.test.js](http://Home.test.js) para Home.test.jsx.

Colocamos os testes agrupados pelo describe e rodamos npm run test no terminal (o react já montou as configurações do test e vai rodar automaticamente a cada alteração depois de rodar uma vez).

Abaixo testes simples onde se espera receber um onde temos 1 (teste de teste).



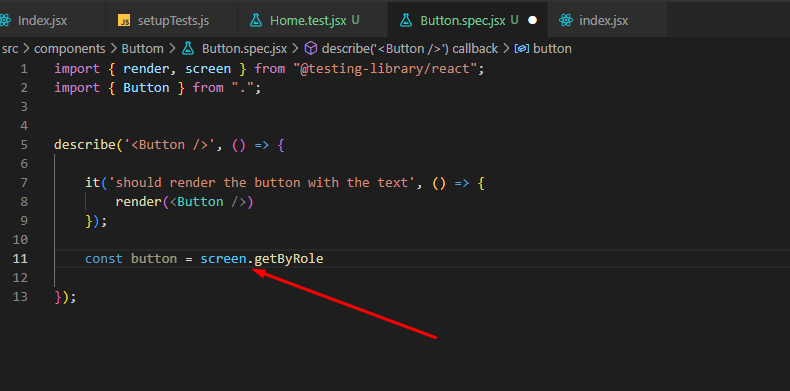
Podemos usar o nome do arquivo como Nome.spec.jsx que também vai funcionar. Testes spec são testes unitário e testes de integração são com nome test



Vamos criar um teste para o button

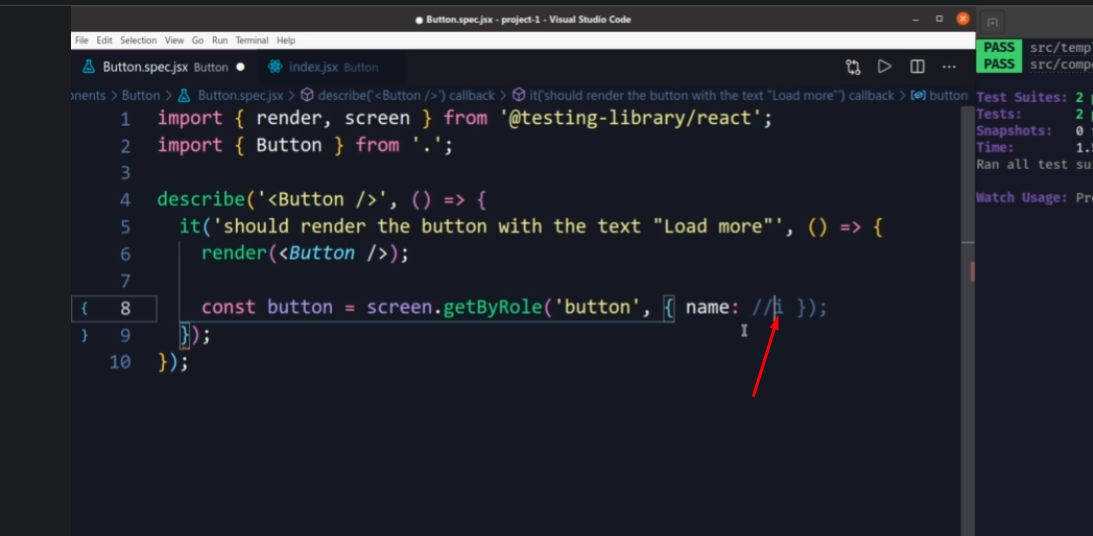
\*Podemos ter testes fora do agrupamento também que funcionam, só não estarão agrupados.

Vamos criar o describe no arquivo e importar render e screen da library de test , criar um describe e declarar uma const recebendo o screen e quando sabemos o que vai renderizar, usamos uma função de get

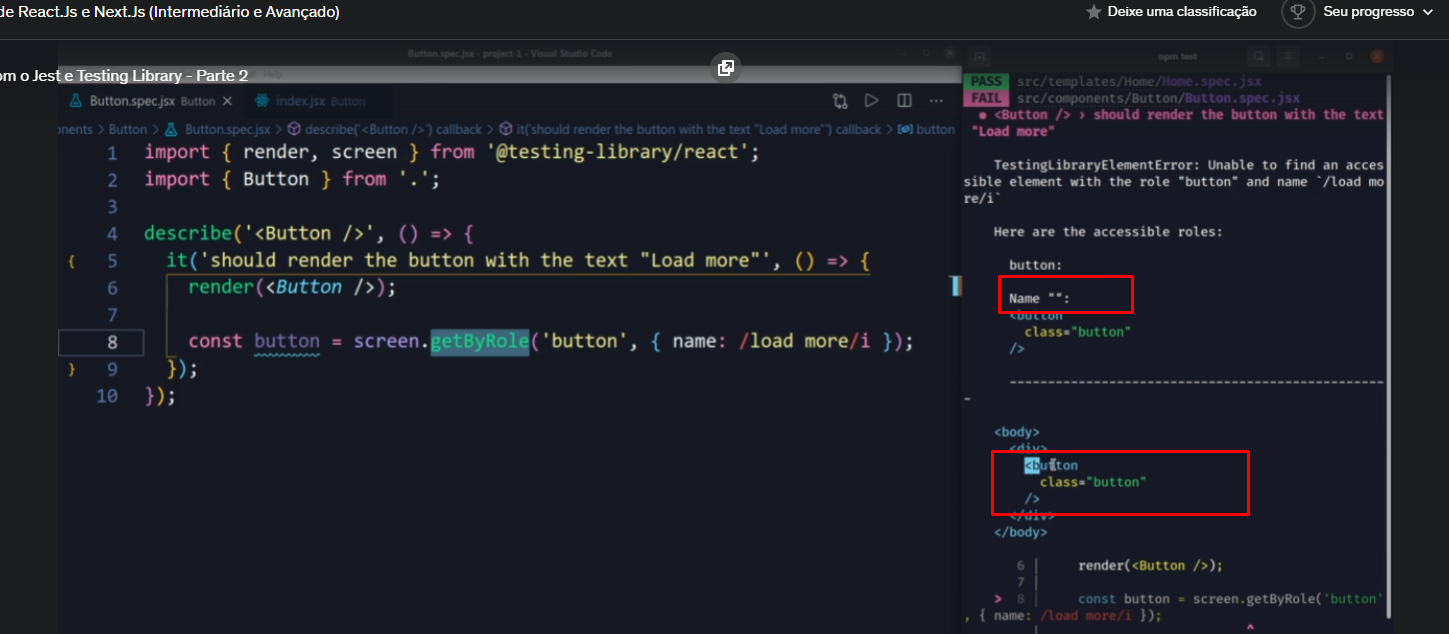


Do tipo query não levanta um erro se não encontrar o elemento:



Usamos uma expressão regular aceitando letras minúsculas que vai pegar um trecho do textop 

Neste caso da erro e o próprio sistema te avisa a razão e neste caso o botão foi renderizado sem name:



Então passamos o prop copm o text:

  it('should render the button with the text "Load more"', () => {

    render(<Button text="Load more" />);

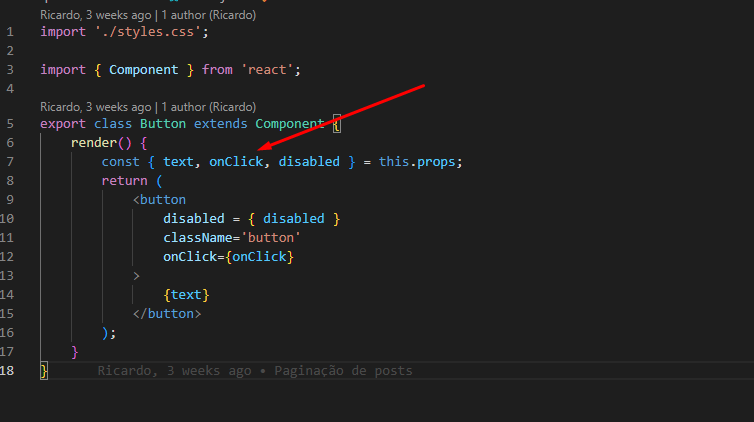
    expect.assertions(1);

    const button = screen.getByRole('button', { name: /load more/i });

    expect(button).toBeInTheDocument();

  });

No segundo teste, vamos verificar o onClick, que é a segunda prop recebida no componente:



Para simular o click, vamos criar uma função de mock (mock function) e passar no render:

 const fn = jest.fn();

    render(<Button text="Load more" onClick={fn} />);

Como é uma função do jest, ela pode nos dizer quantas vezes foi chamada:

 expect(fn).toHaveBeenCalledTimes(1);

O “1” indica quantas vezes se espera que seja chamada a função. Se deixar em branco, vai apenas verificar se foi chamada, então poderíamos colocar duas vezes o click antes com o fireEvent:

 fireEvent.click(button);

 fireEvent.click(button);

que não daira erro, porém ao colocarmos o “1”, por chamarmos 2 vezes, vai acusar o erro.

no caso, o toHaveBeenCalledTimes deve ficar abaixo do firreEvent.

Porém, para simular o evento, vamos importar o userEvento para uma checagem mais natural que vai simular a gigitação tecla por tecla (no caso do buttom, não faz diferença):

import userEvent from '@testing-library/user-event';

 userEvent.click(button);

    expect(fn).toHaveBeenCalledTimes(1);

O teste todo fica assim:

 it('should call function on button click', () => {

    const fn = jest.fn();

    render(<Button text="Load more" onClick={fn} />);

    const button = screen.getByRole('button', { name: /load more/i });

    userEvent.click(button);

    expect(fn).toHaveBeenCalledTimes(1);

  });

enquanto o index.jsx do componente buttom está assim:

import './styles.css';

import { Component } from 'react';

export class Button extends Component {

    render() {

        const { text, onClick, disabled } = this.props;

        return (

            <button

                disabled = { disabled }

                className='button'

                onClick={onClick}

            >

                {text}

            </button>

        );

    }

}

Como vemos acima, temos mais uma prop dizendo que nosso button estará disabled e podemos testar isto:

  it('should be disabled when disabled is true', () => {

    render(<Button text="Load more" disabled={true} />);

    const button = screen.getByRole('button', { name: /load more/i });

    expect(button).toBeDisabled();

  });

E o contrario também (se o disabled for false, o botão deverá estar enabled):

  it('should be enabled when disabled is false', () => {

    const fn = jest.fn();

    render(<Button text="Load more" disabled={false} onClick={fn} />);

    const button = screen.getByRole('button', { name: /load more/i });

    expect(button).toBeEnabled();

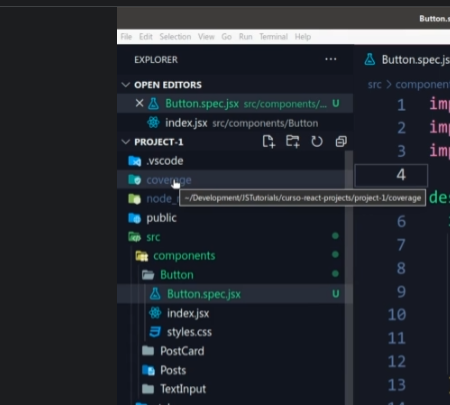
  });

rage

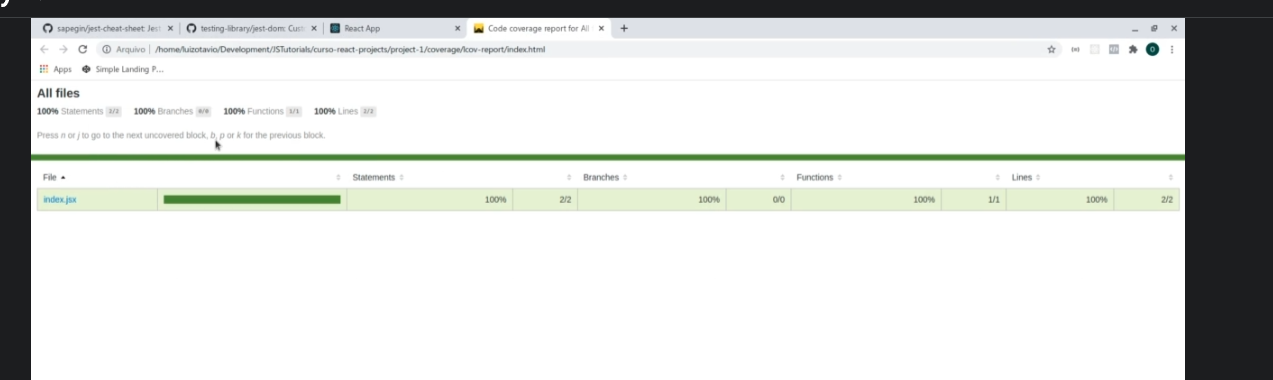
Para verificarmos se todos os testes foram checados, podemos usar o coverage que vai testar mais coisas, então usamos o:

npm test -- --coverage

Uma pasta será criada no projeto:



Dentro dela temos um index.html que podemos abrir no navegador para vermos o covered report:

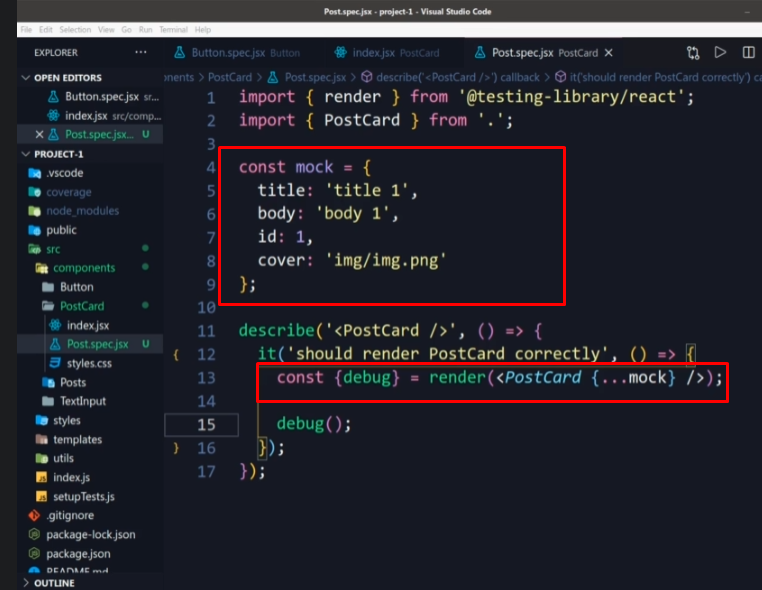


Sobre o sentido de fazer os testes é não precisar olhar no navegador após criação de um componente:

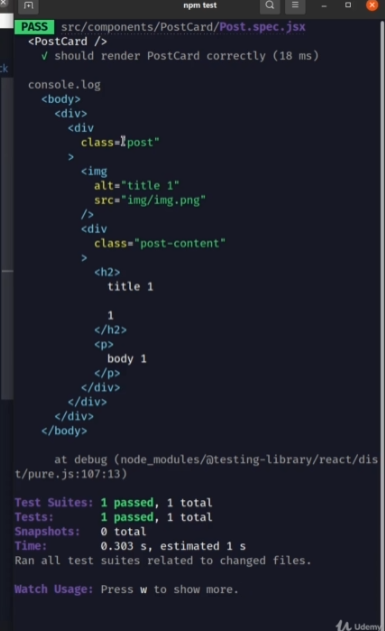
resposta do professor sobre isso:

“Testes são feitos para vc não ter que abrir o navegador e ir lá olhar... Vc não está vendo propósito porque está fazendo as duas coisas, mas vai chegar um ponto que vc testa e nem abre o navegador mais pra ver se tem ou não algo na tela... Ele agiliza bastante o seu workflow justamente por isso”.

No componente seguinte, vamos criar um mock e usar um debug para ver se está tudo ok:



Ao executarmos o teste:



Podemos usar um snapshot que fixa (faz uma imagem) do componente e se algo for alterado o test acusará.

Como fica o código simulando o que o usuário vê na tela:

import { render, screen } from '@testing-library/react';

import { PostCard } from '.';

import { postCardPropsMock } from './mock';

const props = postCardPropsMock;

describe('<PostCard />', () => {

  it('should render PostCard correctly', () => {

    render(<PostCard {...props} />);

    expect(screen.getByAltText(/title 1/i))

      .toHaveAttribute('src', 'img/img.png');

    expect(screen.getByRole('heading', { name: 'title 1 1' })).toBeInTheDocument();

    expect(screen.getByText('body 1')).toBeInTheDocument();

  });

});

O arquivo mock.js carrega o que vai preencher como teste:

export const postCardPropsMock = {

    title: 'title 1',

    body: 'body 1',

    id: 1,

    cover: 'img/img.png'

  };

Caso algo for mudado no componente, podemos fazer umn teste de snapshot

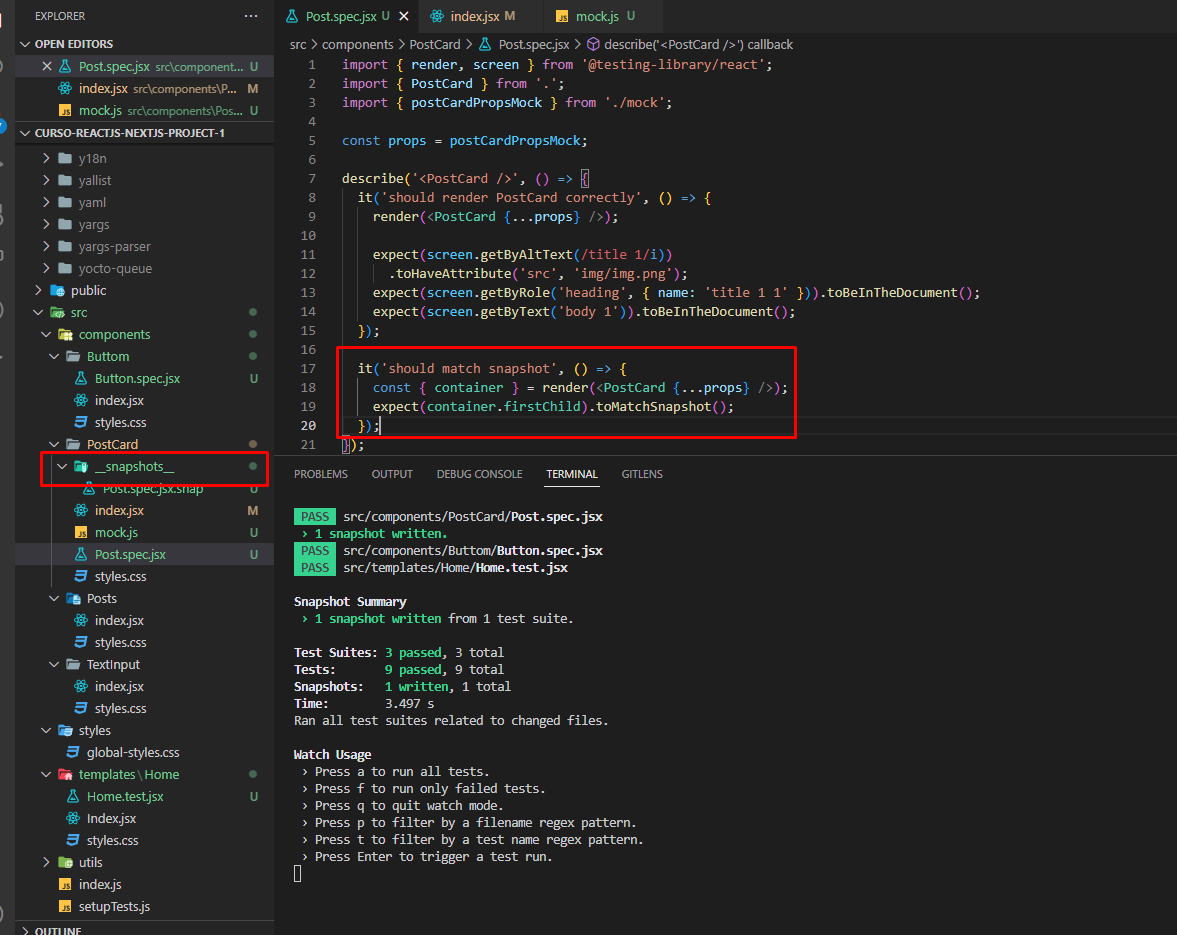
 it('should match snapshot', () => {

    const { container } = render(<PostCard {...props} />);

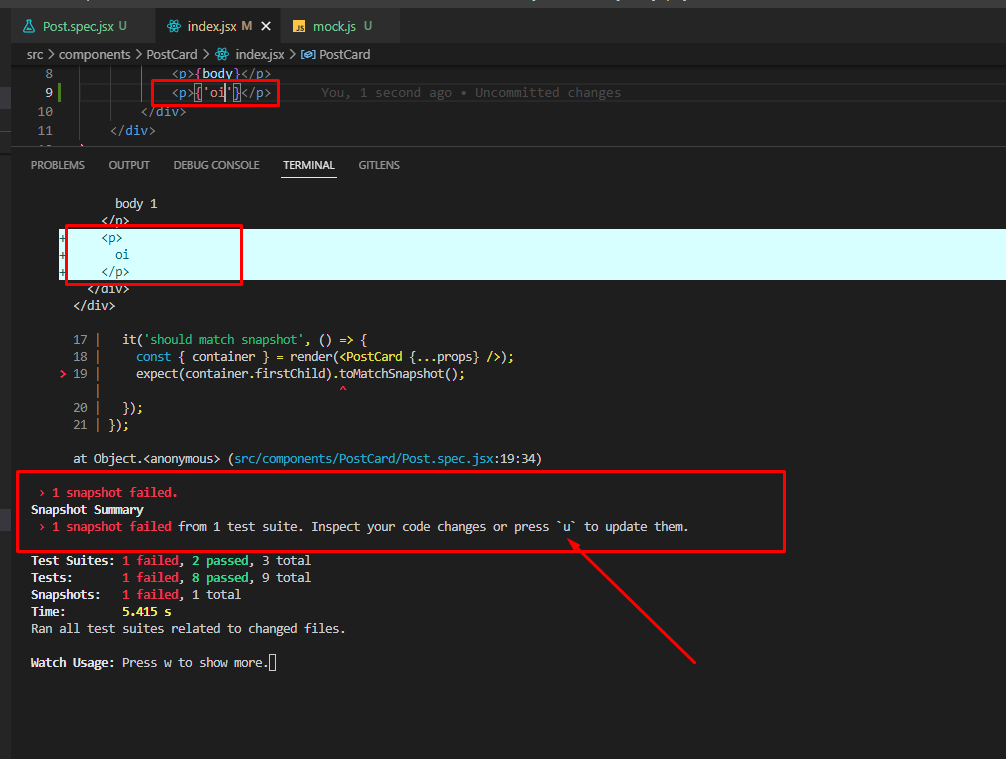
    expect(container.firstChild).toMatchSnapshot();

  });

Quando rodarmos o teste, o snapshot salva um registro do componente como ele é no momento (cria uma pasta com arquivo dentro).



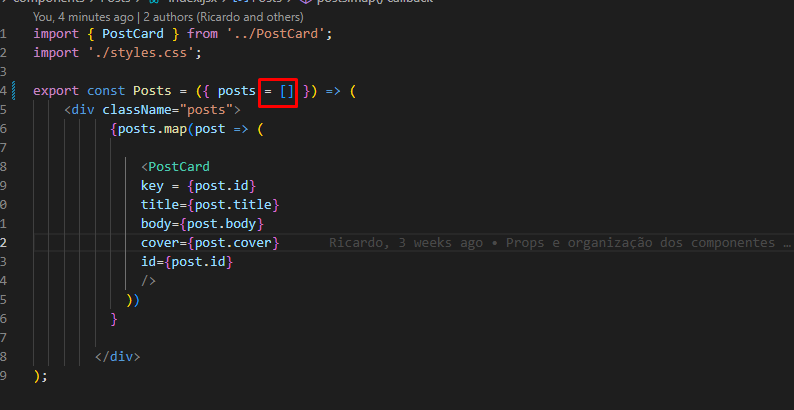
Caso haja uma diferença no snapshot para com o componente atual e mostra o que foi alterado. Caso a alteração for intencional, temos a opção de atualizar clicando “u”.



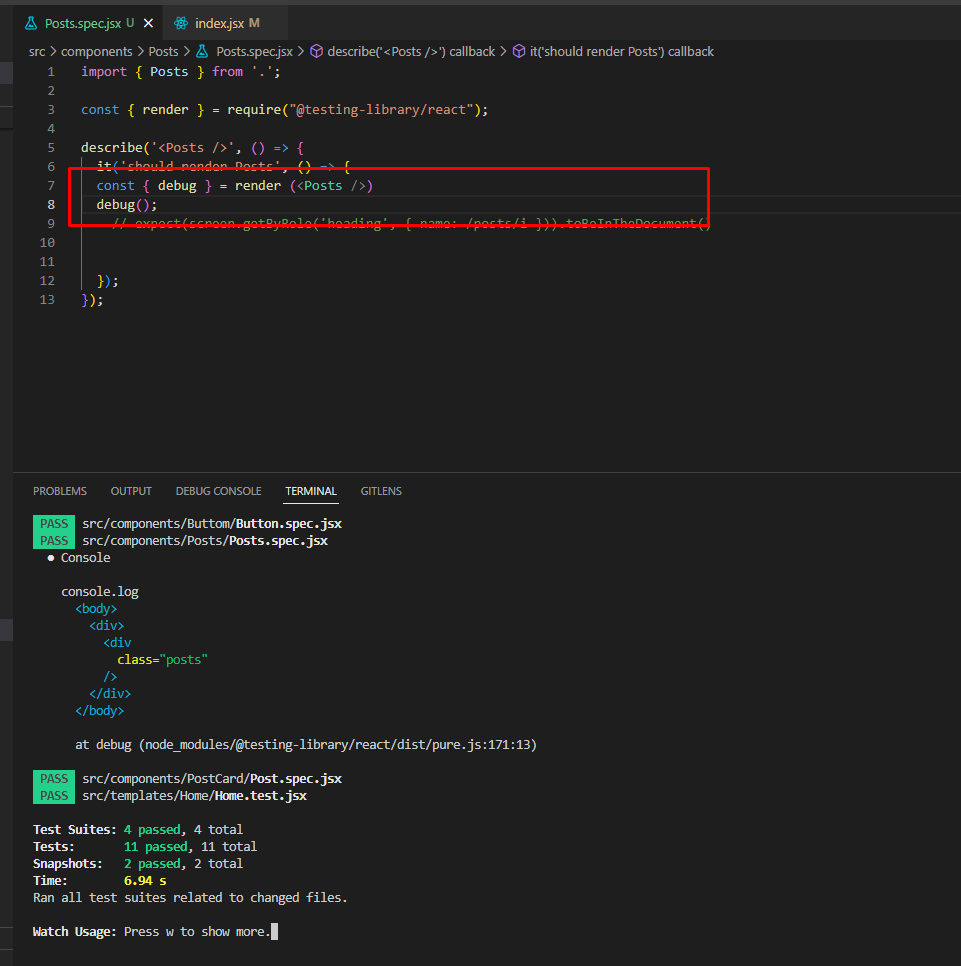
Parte 3:

Vamos criar testes para os posts.

Primeiro precisamos ajustar o componente para dizer que recebe um array:



Depois podemos criar um debug para ver o que acontece



Vamos criar mock para inserir posts criando um array de objetos:

const { render } = require("@testing-library/react");

import { Posts } from '.';

const props = {

    posts: [

      {

        id: 1,

        title: 'title 1',

        body: 'body 1',

        cover: 'img/img1.png',

      },

      {

        id: 2,

        title: 'title 2',

        body: 'body 2',

        cover: 'img/img2.png',

      },

      {

        id: 3,

        title: 'title 3',

        body: 'body 3',

        cover: 'img/img3.png',

      },

    ]

  };

describe('<Posts />', () => {

  it('should render Posts', () => {

   render (<Posts />)

  });

});

No componente de post, já incluímos o postcard dentro de posts e o postcard já foi testado anteriormente



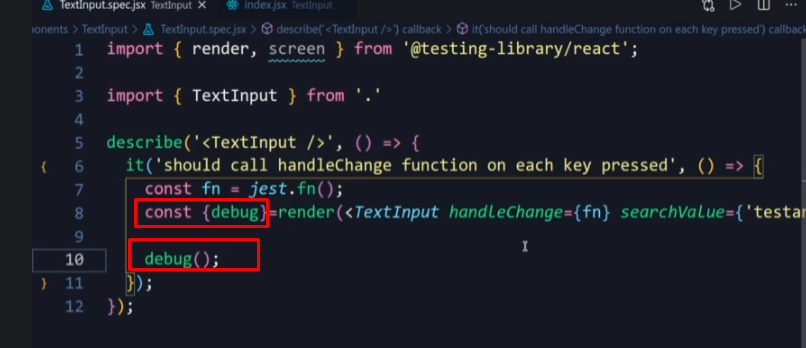
então só precisamos saber quantos posts foram renderizados e se está correto.

Vamos checar se todos os dados estão na pagina, então vamos verificar se nossas props foram enviadas, pegando a partir de uma expressão recular que encontre a palavra titulo :

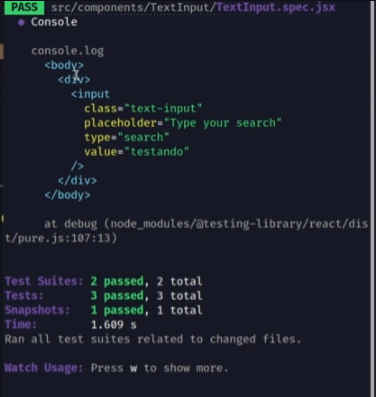


Se trocarmos para 4 e enviarmos 3, vai acusar erro.

\*Lembrando que podemos usar o Debug para vermos o que tem na tela



Retorno:



Agora vamos testar o TextInput.

Novamente criamos uma função para sabermos se ela existe e enviamos um valor para a busca

 it('should have a value of searchValue', () => {

    const fn = jest.fn();

    render(<TextInput handleChange={fn} searchValue={'testando'} />);

    const input = screen.getByPlaceholderText(/type your search/i);

    expect(input.value).toBe('testando');

  });

Testamos o placeholder e se o valor do input é o enviado.

Importamos o userEvent:

import userEvent from '@testing-library/user-event';

e vamos fazer o teste de digitação:

  it('should call handleChange function on each key pressed', () => {

    const fn = jest.fn();

    render(<TextInput handleChange={fn} />);

    const input = screen.getByPlaceholderText(/type your search/i);

    const value = 'o valor';

    userEvent.type(input, value);

    expect(input.value).toBe(value);

    expect(fn).toHaveBeenCalledTimes(value.length);

  });

Nesta linha ele digita o valor declarado anteriormente:

    userEvent.type(input, value);

Lembrando que “type” pode ser os substantivos tipo, modelo ou o verbo digitar que é o caso que vamos chamar.

Na linha abaixo vamos verificar se foi digitado o numero correto de teclas

 expect(fn).toHaveBeenCalledTimes(value.length);

Vamos retornar ao Posts e testar quando um post estiver em branco:

  it('should not render posts', () => {

    render(<Posts />);

    expect(screen.queryByRole('heading', { name: /title/i }))

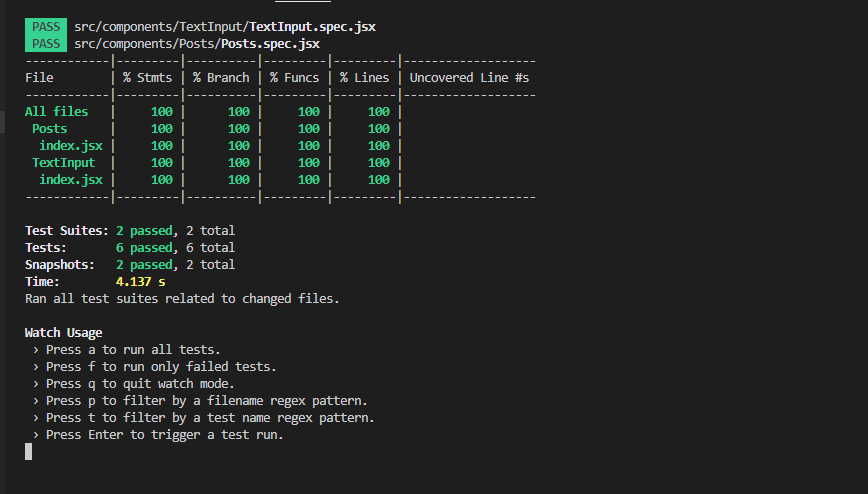
      .not.toBeInTheDocument();

  });

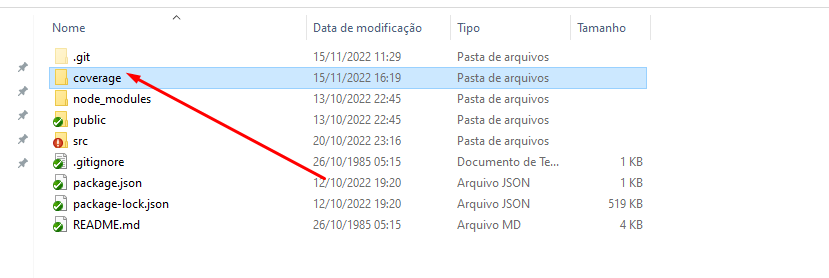
Novamente vamos testar com o coverage

npm test -- --coverage

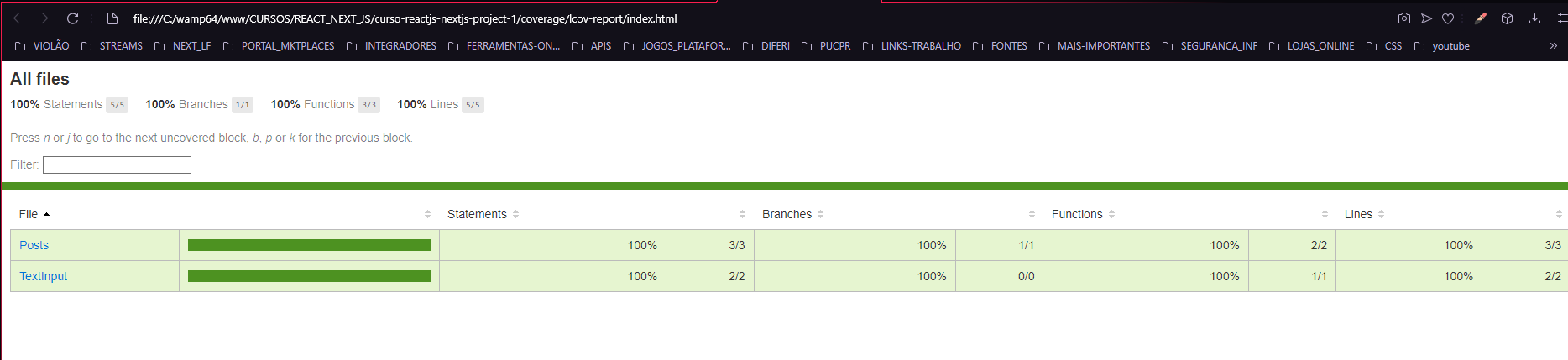
Ele retorna todos os testes



E cria um arquivo na pasta coverage



com um arquivo index.html dentro da pasta coverage\lcov-report com uma UI que podemos visualizar no navegador

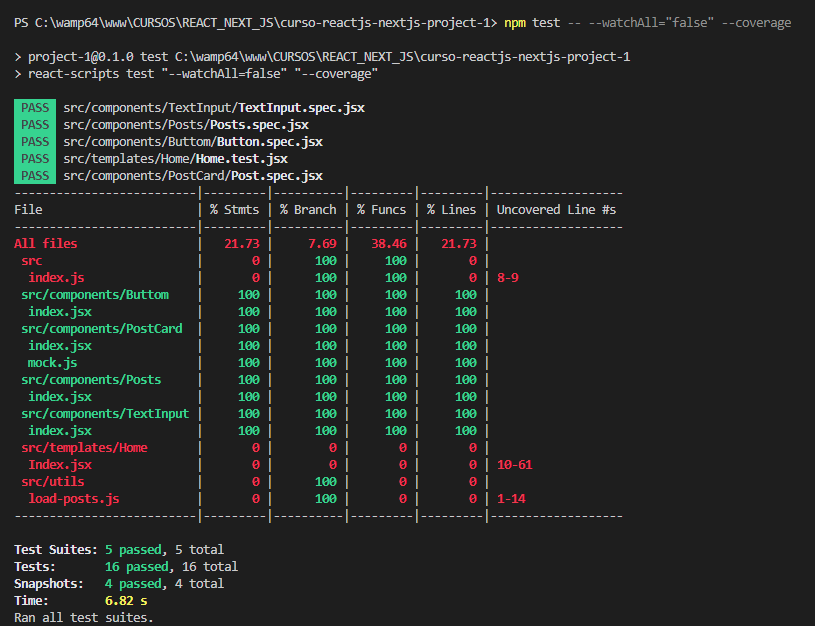




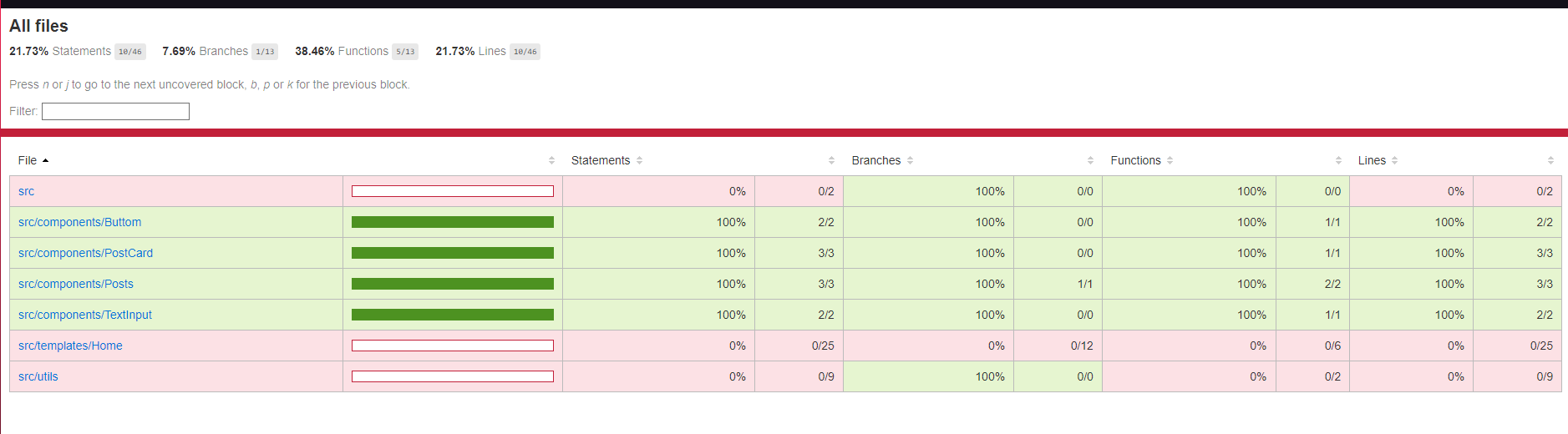
Vamos usar o comando

npm test -- --watchAll=”false” --coverage

que vai pegar todos os arquivos:



O resultado no coverage será outro:



Por exemplo, na src temos um index que nunca foi testado:

