

# Teste de Software - Atividade 1

## Tutorial para atividade 1

O tutorial faz referência a pergunta "How do you test for the non-existence of an element using jest and react-testing-library?" encontrada originalmente no [Link no Stackoverflow](#).

**Docente:** Glauco de Figueiredo Carneiro

**Integrantes do grupo:**

- Edgar de Souza Dias
- Isaac Levi Lira de Oliveira
- José Matheus Ribeiro dos Santos
- Leonardo Alexandre de Souza Barreto
- Ulisses de Jesus Cavalcante

Todos os integrantes do grupo participaram da escolha da pergunta, criação do ambiente, codificação da solução verificada e da análise das demais resposta. Na criação do tutorial e do vídeo, cada integrante ficou responsável por uma parte do trabalho e abaixo o primeiro e último nome do integrante e a etapa que ele desenvolveu: Edgar Dias (Etapa 1), Isaac Oliveira (Etapa 2 - Análise das demais respostas), José Santos (Etapa 2 - Descrição do problema), Leonardo Barreto (Etapa 2 - Análise das demais respostas) e Ulisses Cavalcante (Etapa 2 - Solução do problema a partir da resposta escolhida).

Link do vídeo criado pelos integrantes: [https://drive.google.com/file/d/1fZeLze4-ZYxrS6tDc5VVa3Llu9XVK0I/\\_view?usp=sharing](https://drive.google.com/file/d/1fZeLze4-ZYxrS6tDc5VVa3Llu9XVK0I/_view?usp=sharing)

Nome do repositório no Github: Teste\_Software\_2025\_Jose\_Santos

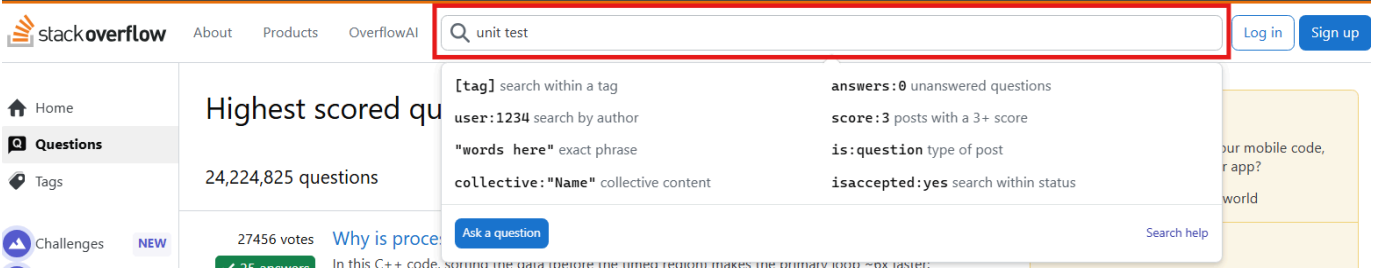
Link do repositório no Github: ([https://github.com/jmatheus21/Teste\\_Software\\_2025\\_Santos\\_Jose](https://github.com/jmatheus21/Teste_Software_2025_Santos_Jose))

## Etapa 1

Para este tutorial, consideraremos o [Stack Overflow](#) como nossa biblioteca de perguntas e respostas sobre desenvolvimento e programação. Nesse sentido, assumindo o objetivo de encontrar uma pergunta sobre um problema relacionado a testes de unidade com pelo menos uma resposta aceita e no mínimo 400 votos, seguiremos os passos descritos a seguir.

## Elaboração da busca

Inicialmente, após acessar a página de perguntas da biblioteca, utilizaremos a barra de busca para inserir os termos relacionados ao conteúdo desejado. Considerando o nosso objetivo, utilizaremos “*unit*” e “*test*” para encontrar resultados acerca de testes unitários, como demonstrado na figura abaixo.



Além dos termos, utilizaremos dois operadores para que a busca traga resultados específicos que satisfaçam os requisitos desejados.

Operador	Valor	Descrição
hasaccepted	yes	Retorna apenas postagens que possuam respostas aceitas
score	400	Retorna apenas postagens que tenham pelo menos 400 votos

Após unir os termos e operadores, a consulta final deve ser construída da seguinte forma:

Q

unit test hasaccepted:yes score:400

## Filtrando resultados

Com os resultados da busca em mãos, utilizaremos um filtro para ordená-los em ordem decrescente de pontuação. Para isso, clicaremos em “More” e, em seguida, em “Score”, como destacado abaixo.

Relevance

Newest 1 More

Active

2 Score

Após aplicar o filtro, escolha uma das perguntas listadas. Como foi mencionado anteriormente, para este trabalho, escolhemos a pergunta “[How do you test for the non-existence of an element using jest and react-testing-library?](#)”.

## Etapa 2

### Descrição do problema

O problema do usuário consiste em descobrir como fazer um teste unitário que verifique que a partir do estado de propriedades ou eventos um componente não está sendo renderizado em uma aplicação baseada na biblioteca `ReactJS` usando os frameworks `Jest` e a biblioteca `React-Testing-Library`.

Durante a questão ele cita sobre o uso dos métodos `getByText` e `getByTestId` que lançam erros imediatamente caso o elemento não seja encontrado. Como um erro é lançado, a execução do teste é interrompida no ponto em que a função falha. O `expect`, usado para verificar a não existência do elemento nunca é alcançado, e o teste falha com uma mensagem de erro indicando que o elemento não pôde ser encontrado, em vez de ser confirmado que ele não existe.

Portanto, para a criação do cenário da questão, foi criado um componente de formulário com 3 (três) campos, "tipo", "habilidades" e "formação". O campo de "tipo" é uma caixa de seleção ( `select` ) e os demais são campos de entrada ( `inputs` ). O campo de seleção permite a escolha entre as opções de "Professor" e "Funcionário". A alteração nesse campo provoca mudanças de renderização nos demais campos. Quando a opção "Professor" é selecionada, o campo de "formação" é renderizado e o de "habilidades" não é, ao mesmo passo que, quando a opção "Funcionário" é selecionado, o campo de "habilidades" é exibido e o de "formação" não é.

Com algumas configurações omitidas para maior clareza, segue abaixo a configuração do cenário:

```
const App = () => {
  const [tipo, setTipo] = useState("P");

  return (
    <form>
      <div>
        <label htmlFor="tipo">Tipo: </label>
        <select id="tipo" onChange={(e) => setTipo(e.target.value)}>
          <option value="P">Professor</option>
          <option value="F">Funcionário</option>
        </select>
      </div>
      {tipo == "P" ? (
        <div>
          <label htmlFor="formacao">Formação: </label>
          <input
            id="formacao"
            type="text"
          />
        </div>
      ) : null}
    </form>
  )
}
```

```

        placeholder="Digite a formação do professor"
      />
    </div>
  ) : (
    <div>
      <label htmlFor="habilidades">Habilidades: </label>
      <input
        id="habilidades"
        type="text"
        placeholder="Digite as habilidades do funcionário"
      />
    </div>
  )}
</form>
);
};

```

É importante notar que o problema não consiste em verificar se o elemento não está visível para o usuário, mas sim se ele existe ou não no DOM. Portanto, um cenário em que um elemento fica invisível (por exemplo, usando `display: none` na estilização) devido à alteração de uma propriedade ou evento não consiste no objetivo da pergunta do usuário.

Para execução do ambiente de teste, após a clonagem do repositório, instale as bibliotecas com o comando no terminal `npm install` e rode os testes da solução com `npm test tests/App.test.jsx`. Caso queira visualizar a tela do formulário, execute `npm run dev`.

## Solução do problema a partir da resposta escolhida

Como já foi mencionado na descrição do problema, o autor da pergunta cita que tentou utilizar os métodos `getByText` e `getByTestId` para verificar a ausência de elementos. Entretanto, tais métodos lançam uma exceção quando o elemento não é encontrado, o que faz com que a execução do teste seja interrompida imediatamente, ou seja, o `expect` que serviria para verificar a não existência do elemento nunca seria alcançado.

Dessa maneira, a solução adotada para esse problema, foi a utilização do método `queryBy`, que, diferentemente do `getBy`, quando o elemento não é encontrado, retorna **null**, o que é ideal para o nosso caso, que é de verificar a não existência de um elemento, já que podemos tratar isso, visto que a execução do nosso teste não será interrompida de forma abrupta devido a uma exceção, caso o elemento não tenha sido encontrado, como é o caso do `getBy`.

Vale ressaltar que haverá dois casos de teste: um para verificar a renderização do campo "Habilidades" e outro para o campo "Formação". Como já mencionado, o campo "Habilidades" deve ser exibido apenas quando o tipo selecionado for Funcionário, e o campo "Formação" apenas quando o tipo selecionado for Professor.

A seguir, vamos entender como essa solução foi aplicada, linha por linha, no código:

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

Importa funções da *Testing Library*, como o `render`, que renderiza o componente *React* em um ambiente de teste, e o `screen`, que permite acessar elementos renderizados (como `getByLabelText` e `queryByText`)

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

Adiciona *matchers* extras, que são responsáveis pela verificação de elementos no *DOM*, como o `toBeNull()` e `toBeInTheDocument()`;

```
import userEvent from "@testing-library/user-event";
```

Permite simular interações reais do usuário, como clicar, digitar e selecionar opções em campos

```
<select>
```

```
import App from "../src/App";
```

Importa o componente `App`, que será o alvo dos testes

```
describe("App Component", () => {
```

Define um bloco de testes relacionado ao componente `App`, é uma forma de agrupar testes que se referem ao mesmo comportamento ou funcionalidade

### 1º caso de teste - Verificação da renderização do campo “Habilidades”

```
it("verifica a renderização do campo 'Habilidades' utilizando os métodos queryByText e toBeNull", async () => {
```

Define um teste com essa descrição, usa o `async` porque usa interações assíncronas (com `userEvent`)

```
render(<App/>);
```

Renderiza o componente `App`, criando um *DOM* virtual que será utilizado no teste

```
const tipoSelect = screen.getByLabelText(/Tipo:/i);
```

Seleciona o `<select>` que está associado a um `<label>` com o texto “Tipo:”, ou seja, aqui é como se o usuário estivesse acessando o `<select>` referente ao tipo

```
await userEvent.selectOptions(tipoSelect, 'P');
```

Simula o usuário selecionando a opção ‘P’ (Professor) no campo Tipo

```
expect(tipoSelect.value).toBe('P');
```

Garante que o valor selecionado seja realmente ‘P’, ou seja, há confirmação de que o tipo de usuário selecionado é um professor

```
const campoHabilidadesAntesDaMudancaDeTipo = screen.queryByText(/Habilidades:/i);
```

Aqui é verificado se o campo “Habilidades” foi renderizado na tela. Como estamos usando o `queryBy`, caso o elemento seja encontrado, ele retorna o elemento: caso contrário, é retornado **null**. Nesse caso, o valor retornado deve ser **null**, pois o campo “Habilidades” não deve ser renderizado quando o tipo selecionado é um professor

```
expect(campoHabilidades).toBeNull();
```

Confirma que o campo “Habilidades” não foi renderizado

```
await userEvent.selectOptions(tipoSelect, 'F');
```

Nesse momento, simulamos a troca do tipo de usuário para um funcionário, para verificar se o campo “Habilidades” será renderizado

```
expect(tipoSelect.value).toBe('F');
```

Garante que o valor selecionado seja realmente 'F', ou seja, há confirmação de que o tipo de usuário selecionado é um funcionário

```
const campoHabilidadesDepoisDaMudancaDeTipo = screen.queryByText(/Habilidades:/i);
```

Aqui é verificado se o campo “Habilidades” foi renderizado. Agora, ele deve ser renderizado, pois esse campo é referente ao usuário do tipo funcionário

```
expect(campoHabilidadesDepoisDaMudancaDeTipo).not.toBeNull();
```

Confirma que o campo “Habilidades” foi renderizado através da negação do `toBeNull()`, ou seja, como esse campo foi encontrado, o `queryBy` irá retornar o elemento; sendo assim, não será nulo

## 2º caso de teste - Verificação da renderização do campo "Formação"

```
it("verifica a renderização do campo 'Formação' utilizando os métodos queryByText e toBeInTheDocument", async () => {
```

Define um teste com essa descrição, usa o `async` porque usa interações assíncronas (com `userEvent`)

```
render(<App/>);
```

Renderiza o componente `App`, criando um *DOM* virtual que será utilizado no teste

```
const tipoSelect = screen.getByLabelText(/Tipo:/i);
```

Seleciona o `<select>` que está associado a um `<label>` com o texto “Tipo:”, ou seja, aqui é como se o usuário estivesse acessando o `<select>` referente ao tipo

```
await userEvent.selectOptions(tipoSelect, 'F');
```

Simula o usuário selecionando a opção 'F' (Funcionário) no campo Tipo

```
expect(tipoSelect.value).toBe('F');
```

Garante que o valor selecionado seja realmente 'F', ou seja, há confirmação de que o tipo de usuário selecionado é um funcionário

```
const campoFormacao = screen.queryByText(/Formação:/i);
```

Aqui é verificado se o campo “Formação” foi renderizado na tela. Como estamos usando o `queryBy`, caso o elemento seja encontrado, ele retorna o elemento: caso contrário, é retornado **null**. Nesse caso, o valor retornado deve ser **null**, pois o campo “Formação” não deve ser renderizado quando o tipo selecionado é um funcionário

```
expect(campoFormacao).not.toBeInTheDocument();
```

Confirma que o campo “Formação” não foi renderizado através da negação do método `toBeInTheDocument()`, que verifica se um determinado elemento se encontra no *DOM*

```
await userEvent.selectOptions(tipoSelect, 'P');
```

Nesse momento, simulamos a troca do tipo de usuário para um professor, para verificar se o campo “Formação” será renderizado

```
expect(tipoSelect.value).toBe('P');
```

Garante que o valor selecionado seja realmente 'P', ou seja, há confirmação de que o tipo de usuário selecionado é um professor

```
const campoFormacaoDepoisDaMudancaDeTipo = screen.queryByText(/Formação:/i);
```

Aqui é verificado se o campo “Formação” foi renderizado. Agora, ele deve ser renderizado, pois esse campo é referente ao usuário do tipo professor

```
expect(campoFormacaoDepoisDaMudancaDeTipo).toBeInTheDocument();
```

Confirma que o campo “Formação” foi renderizado através do método `toBeInTheDocument()`, que, como já foi mencionado anteriormente, verifica se um elemento está presente no *DOM*. Nesse caso, esse campo deverá ser renderizado, pois faz referência a um usuário do tipo professor

## Análise das demais respostas

O processo para escolher as demais alternativas de respostas para este documento não foi o número de votos, e sim o quão significativa a sua análise será para o entendimento do problema do autor da pergunta.

### 1. Use `queryBy` / `queryAllBy`. - 104 Votos

▲

Use `queryBy` / `queryAllBy`.

104

▼

As you say, `getBy*` and `getAllBy*` throw an error if nothing is found.

However, the equivalent methods `queryBy*` and `queryAllBy*` instead return `null` or `[]`:

🔖

🕒

**queryBy**

`queryBy*` queries return the first matching node for a query, and return `null` if no elements match. This is useful for asserting an element that is not present. This throws if more than one match is found (use `queryAllBy` instead).

**queryAllBy**


`queryAllBy*` queries return an array of all matching nodes for a query, and return an empty array (`[]`) if no elements match.

<https://testing-library.com/docs/dom-testing-library/api-queries#queryby>.

So for the specific two you mentioned, you'd instead use `queryByText` and `queryById`, but these work for all queries, not just those two.

Share Improve this answer Follow

answered Nov 4, 2019 at 12:27

 **Sam**  
6,670 ● 6 ● 48 ● 71

Esta resposta está em segundo lugar dentre as escolhidas, porém alguns usuários acreditam que ela deveria ser a resposta aceita. Ela tem um caráter menos prático e mais conceitual, explicando que a função que o autor da pergunta utilizou, a `getBy` lança um error se o objeto que ele quer observar na página não foi encontrado, e então ofereceu as alternativas `queryBy` e `queryAllBy`.

### 2. `GetBy*` throws an error when not finding an elements, so you can check for that - 38 Votos

38

getBy\* throws an error when not finding an elements, so you can check for that


```
expect(() => getByText('your text')).toThrow('Unable to find an element');
```

Share

Improve this answer

Follow

answered Jun 18, 2020 at 10:50



Gabriel Vasile

2,368 ● 1 ● 17 ● 28

7

This can be pretty error-prone. Error throws are used for debugging purposes and not to for verification.  
– Milen Gardev Jan 6, 2021 at 17:03

2

@RahulMahadik because expect(...) will call the anonymous function when it's ready to detect/catch a thrown error. If you omit the anonymous arrow function, you will immediately call a throwing function as the code executes, which immediately breaks out of execution. – Ian MacFarlane Aug 23, 2022 at 1:00

Add a comment

Esta resposta está tecnicamente correta, mas não responde a pergunta do usuário e nem explica como pode resolvê-la. Ele apenas atesta que a função `getBy` lança um erro quando não encontra os elementos que está procurando e utiliza a função de depuração para o propósito de verificação, o que não é uma boa prática e pode levar a erros.

### 3. Another solution: you could also use a try/catch block - 13 Votos

13

Another solution: you could also use a try/catch block

```
expect.assertions(1)
try {
  // if the element is found, the following expect will fail the test
  expect(getByTestId('your-test-id')).not.toBeVisible();
} catch (error) {
  // otherwise, the expect will throw, and the following expect will pass the test
  expect(true).toBeTruthy();
}
```

Este usuário sugere uma abordagem com `try / catch`, essa alternativa é desnecessariamente complexa, uma vez que o `react-testing-library` oferece funções simples e diretas para verificar a não existência de elementos. Além disso, o uso de `try / catch` pode levar a múltiplas expectativas sendo executadas, o que pode causar confusão e erros.

### 4. The default behavior of queryByRole is to find exactly one element. If not, it throws an error. So if you catch an error, this means the current query finds 0 element - 8 Votos



8



The default behavior of `queryByRole` is to find exactly one element. If not, it throws an error. So if you catch an error, this means the current query finds 0 element

```
expect(
  ()=>screen.getByRole('button')
).toThrow()
```

`getByRole` returns 'null', if it does not find anything

```
expect(screen.queryByRole('button')).toEqual((null))
```

`findByRole` runs asynchronously, so it returns a `Promise`. If it does not find an element, it rejects the promise. If you are using this, you need to run `async` callback

```
test("testing", async () => {
  let nonExist = false;
  try {
    await screen.findByRole("button");
  } catch (error) {
    nonExist = true;
  }
  expect(nonExist).toEqual(true);
});
```

[Share](#) [Improve this answer](#) [Follow](#)

answered Feb 1, 2023 at 6:20



Yilmaz

50.4k ● 19 ● 222 ● 273

[Add a comment](#)

Esta resposta é a mais errônea entre as alternativas. Ele explica que o comportamento do `queryByRole` é encontrar exatamente um elemento, e, se não, lança um erro, e que o `getByRole` retorna null caso não encontre nada. A explicação das funções está invertida, o que pode induzir ao erro qualquer pessoa sem o conhecimento necessário. Além de utilizar métodos desnecessariamente complicados como as funções `expect()` e `toThrow()`.