

Chapter 2 — Testing User Interaction with React Testing Library

Purpose of These Tests

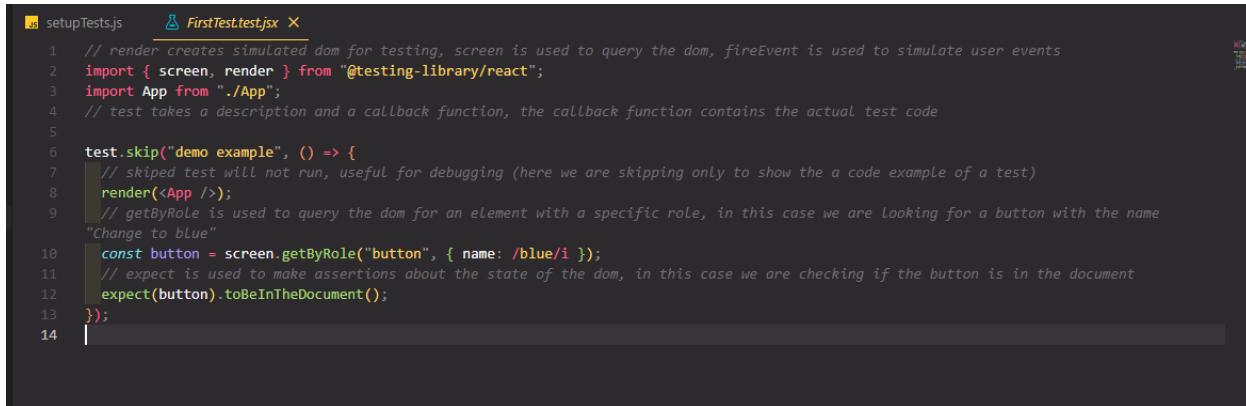
The goal of these tests is to **verify user behavior**, not React implementation details.

We want to ensure that:

- The UI responds correctly when users **click the button**
- The UI updates properly when users **toggle the checkbox**
- The visual state (color, disabled state) always matches what the user expects

These tests follow **React Testing Library philosophy**:

Test what the user can see and do — not internal state or functions.



```
setupTests.js  FirstTest.test.jsx X
1 // render creates simulated dom for testing, screen is used to query the dom, fireEvent is used to simulate user events
2 import { screen, render } from "@testing-library/react";
3 import App from "./App";
4 // test takes a description and a callback function, the callback function contains the actual test code
5
6 test.skip("demo example", () => {
7   // skiped test will not run, useful for debugging (here we are skipping only to show the a code example of a test)
8   render(<App />);
9   // getByRole is used to query the dom for an element with a specific role, in this case we are looking for a button with the name
10  "Change to blue"
11  const button = screen.getByRole("button", { name: /blue/i });
12  // expect is used to make assertions about the state of the dom, in this case we are checking if the button is in the document
13  expect(button).toBeInTheDocument();
14});
```

Tools Used in This Test File

render

- Renders the <App /> component into a **simulated DOM**
- Allows us to interact with the component as if it were in a browser

screen

- Used to query elements from the rendered DOM
- Queries are based on **accessibility**, not implementation

fireEvent

- Simulates **user actions** such as clicks
- Used here to mimic button clicks and checkbox toggling

expect

- Used to assert what should be true about the UI after each interaction
-

- Code snippet:

```

...  setupTests.js      App.test.jsx X
      1  import { render, screen, fireEvent } from "@testing-library/react";
      2  import App from "./App";
      3  import { expect, test } from "vitest";
      4
      5  test("button click flow", () => {
      6    render(<App />);
      7    const button = screen.getByRole("button", { name: /blue/i });
      8    const checkbox = screen.getByRole("checkbox");
      9    fireEvent.click(button);
     10   expect(button).toHaveTextContent("Change to red");
     11
     12   expect(button).toHaveClass("blue");
     13   fireEvent.click(checkbox);
     14   expect(button).toHaveClass("gray");
     15   fireEvent.click(checkbox);
     16   expect(button).toHaveClass("blue");
     17 });
     18
     19 test("checkbox click flow", () => {
     20   render(<App />);
     21   const checkbox = screen.getByRole("checkbox");
     22   const button = screen.getByRole("button", { name: /blue/i });
     23   expect(checkbox).not.toBeChecked();
     24   expect(button).toBeEnabled();
     25
     26   fireEvent.click(checkbox);
     27   expect(button).toBeDisabled();
     28   expect(button).toHaveClass("gray");
     29
     30   fireEvent.click(checkbox);
     31   expect(button).toBeEnabled();
     32   expect(button).toHaveClass("red");
     33 });
     34

```

Test 1 — Button Click Flow

Test Description

This test verifies how the **button behaves when clicked**, and how its behavior changes when the checkbox is toggled.

Step 1: Render the Application

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

- Creates a simulated DOM
 - The component is now ready for interaction
-

Step 2: Query Elements the Way a User Would

```
const button = screen.getByRole("button", { name: /blue/i });
```

```
const checkbox = screen.getByRole("checkbox");
```

Why this is important:

- getByRole reflects how assistive technologies find elements
 - No reliance on class names, IDs, or test IDs
-

Step 3: Simulate User Interaction

```
fireEvent.click(button);
```

- Simulates a real user clicking the button
 - We do not check React state
 - We only care about visible effects
-

Step 4: Verify UI Behavior When Checkbox Is Toggled

```
fireEvent.click(checkbox);
```

```
expect(button).toHaveClass("gray");
```

This confirms:

- The button becomes disabled
 - The UI reflects the disabled state visually
-

Step 5: Re-enable the Button

```
fireEvent.click(checkbox);  
  
expect(button).toHaveClass("blue");
```

This confirms:

- The button returns to its previous color
 - The UI correctly restores state from a user perspective
-

Why Some Assertions Are Commented Out

The commented assertions show **alternative checks** that could be tested:

- Initial color
- Text changes
- Intermediate states

They are commented to:

- Keep the test focused
 - Avoid over-testing the same behavior
 - Demonstrate different assertion possibilities
-

Test 2 — Checkbox Click Flow

Test Description

This test focuses exclusively on the **checkbox behavior** and how it affects the button.

Step 1: Render and Query Elements

```
render(<App />);

const checkbox = screen.getByRole("checkbox");

const button = screen.getByRole("button", { name: /blue/i });
```

Step 2: Assert Initial State

```
expect(checkbox).not.toBeChecked();

expect(button).toBeEnabled();
```

This verifies:

- The checkbox starts unchecked
 - The button is clickable by default
-

Step 3: Disable the Button via Checkbox

```
fireEvent.click(checkbox);

expect(button).toBeDisabled();

expect(button).toHaveClass("gray");
```

This confirms:

- User interaction disables the button
 - Visual feedback matches functional state
-

Step 4: Re-enable the Button

```
fireEvent.click(checkbox);

expect(button).toBeEnabled();

expect(button).toHaveClass("red");
```

This ensures:

- The UI fully recovers

- State transitions are correct from the user's perspective
-

Why These Are Good React Testing Library Tests

These tests are strong because they:

- Query elements by accessibility
- Simulate real user actions
- Assert visible behavior
- Do NOT test internal state
- Do NOT mock implementation details

They remain **stable even if internal code is refactored**, as long as the user experience stays the same.

