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.

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

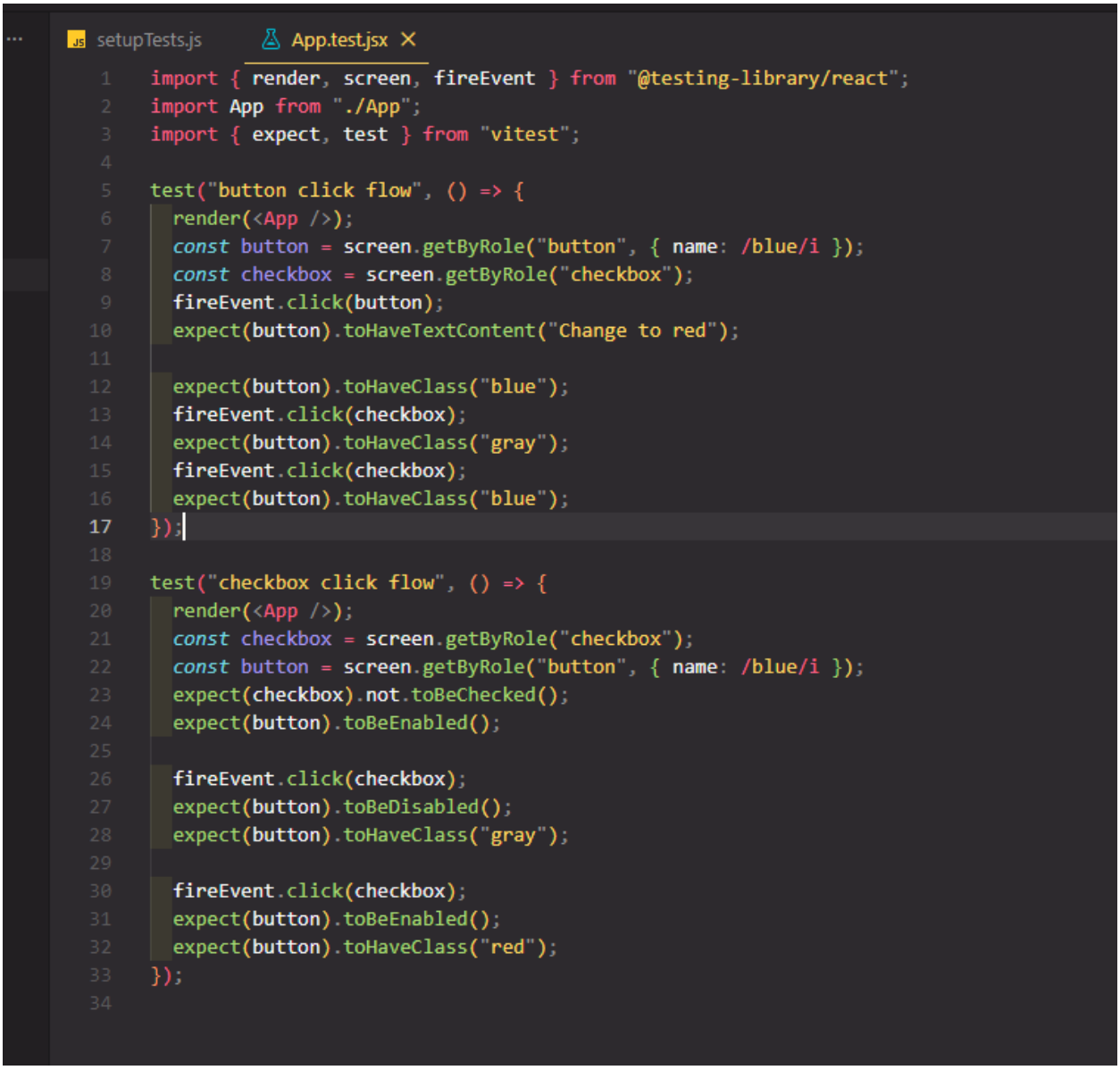
- Simulates **user actions** such as clicks
- Used here to mimic button clicks and checkbox toggling

expect

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

---

- Code snippet:



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.