User Interaction Testing

Realistic interaction simulation with user-event, testing clicks and typing, verifying form submission with vi.fn(), and testing asynchronous UI updates with findBy* -- with simple examples.

Intermediate 11 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/user-interaction-testing

---

User Interaction Testing

In Component Testing, we learned to test that a component renders CORRECTLY -- but most React apps are "static" until a user clicks something, types something, or submits a form. This lesson covers testing user INTERACTIONS.

Realistic Interaction Simulation with user-event

Alongside React Testing Library, `@testing-library/user-event` is used to simulate user interactions:

```
npm install -D @testing-library/user-event
```

RTL's own `fireEvent` API can also trigger a click/type, but `fireEvent` dispatches a single DOM event (like `click`) directly. `user-event` simulates the IN-BETWEEN steps a real user triggers while clicking/typing too (hover, focus, pointer events) -- which is why RTL's official docs now RECOMMEND `user-event` over `fireEvent`.

Testing a Click

Let's test the `Counter` component from State & Events again, this time by simulating a real click:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

describe("Counter interaction", () => {
  it("increments the count when the button is clicked", async () => {
    // userEvent.setup() creates a "user" object that behaves closer to a
    // real user's click than fireEvent (it also simulates intermediate steps
    // like hover and focus). That's why RTL now RECOMMENDS userEvent.
    const user = userEvent.setup();
    render(<Counter />);

    expect(screen.getByText("Count: 0")).toBeInTheDocument();

    // userEvent's methods are ASYNCHRONOUS -- they must always be awaited.
    await user.click(screen.getByRole("button", { name: /increment/i }));

    expect(screen.getByText("Count: 1")).toBeInTheDocument();
  });
});
```

`userEvent.setup()` creates a "user" object. This object's methods (`click`, `type`, etc.) are ALWAYS asynchronous and must be `await`ed -- forget to, and the test moves to the next line before the click finishes, checking a stale DOM state.

Testing Typing

The controlled-input pattern from Forms is tested with `user.type`:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";

function NameInput() {
  const [name, setName] = useState("");
  return (
    <div>
      <label htmlFor="name">Name</label>
      <input id="name" value={name} onChange={(e) => setName(e.target.value)} />
      <p>You typed: {name}</p>
    </div>
  );
}

describe("NameInput interaction", () => {
  it("updates the displayed text as the user types", async () => {
    const user = userEvent.setup();
    render(<NameInput />);

    const input = screen.getByLabelText("Name");

    // user.type types the given text CHARACTER BY CHARACTER -- each
    // keystroke triggers the onChange on the controlled component in a way
    // that closely resembles real typing.
    await user.type(input, "Ada");

    expect(screen.getByText("You typed: Ada")).toBeInTheDocument();
    expect(input).toHaveValue("Ada");
  });
});
```

`user.type(input, "Ada")` types the given text CHARACTER BY CHARACTER -- each keystroke triggers the controlled component's `onChange` much like typing on a real keyboard would. At the end of the test, we check both the displayed text (`getByText`) and the input's own value (`toHaveValue`).

Testing Form Submission

Filling out and submitting a form is the most common scenario where `user.type` and `user.click` are used together:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect, vi } from "vitest";

function SignupForm({ onSubmitted }) {
  const [email, setEmail] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    onSubmitted(email);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="email">Email</label>
      <input id="email" value={email} onChange={(e) => setEmail(e.target.value)} />
      <button type="submit">Sign Up</button>
    </form>
  );
}

describe("SignupForm submission", () => {
  it("calls onSubmitted with the typed email when the form is submitted", async () => {
    const user = userEvent.setup();

    // vi.fn() is a FAKE function that stands in for a real prop -- we can
    // later query what arguments it was called with and how many times.
    const handleSubmitted = vi.fn();
    render(<SignupForm onSubmitted={handleSubmitted} />);

    await user.type(screen.getByLabelText("Email"), "ada@example.com");
    await user.click(screen.getByRole("button", { name: /sign up/i }));

    expect(handleSubmitted).toHaveBeenCalledWith("ada@example.com");
    expect(handleSubmitted).toHaveBeenCalledTimes(1);
  });
});
```

`vi.fn()` creates a FAKE function that stands in for a real prop -- without any real request leaving the component, we can verify what ARGUMENTS this function was called with and how MANY times. `toHaveBeenCalledWith(...)` and `toHaveBeenCalledTimes(...)` are matchers specific to these mock functions.

Testing Asynchronous UI Updates

With the `useEffect` pattern from Hooks, a component can update ITSELF over time (like a fetch request finishing). Testing that kind of update calls for the `findBy*` queries:

```
import { useEffect, useState } from "react";
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";

// The useEffect pattern from the Hooks lesson -- the component updates its
// own state a while after mounting (in a real app this would be a fetch
// request completing; here we use setTimeout to keep it simple).
function DelayedGreeting() {
  const [ready, setReady] = useState(false);

  useEffect(() => {
    const timer = setTimeout(() => setReady(true), 50);
    return () => clearTimeout(timer);
  }, []);

  if (!ready) return <p>Loading...</p>;
  return <p>Welcome!</p>;
}

describe("DelayedGreeting async update", () => {
  it("shows loading first, then the greeting once ready", async () => {
    render(<DelayedGreeting />);

    // On the first render, "Loading..." is still shown.
    expect(screen.getByText("Loading...")).toBeInTheDocument();

    // findByText is the ASYNC version of getByText: if the element isn't
    // there right away, it doesn't throw immediately -- it keeps retrying
    // for a given period (1000ms by default). This is the right way to test
    // anything in the DOM that changes over time (fetch, timers, after an
    // animation) -- waitFor can be used for the same purpose.
    const greeting = await screen.findByText("Welcome!");
    expect(greeting).toBeInTheDocument();
  });
});
```

`getByText` (and `queryByText`) check the DOM ONLY AT THAT MOMENT -- if the element isn't there yet, the test fails. `findByText` is ASYNCHRONOUS instead: it doesn't throw if the element isn't there immediately, it retries for a set amount of time (1000ms by default) and continues once the element appears. This is the correct way to test anything that changes the DOM over time (fetch, timers, post-animation state); `waitFor(...)` can be used for the same purpose.

Summary and Glossary

`@testing-library/user-event` offers a MORE REALISTIC interaction simulation than `fireEvent`; the `click` / `type` methods on the object returned by `userEvent.setup()` must always be `await`ed. Mock functions created with `vi.fn()` are used to verify that a callback prop was called with the right arguments. Asynchronous DOM updates (that change over time) are tested with `findByText` / `waitFor` instead of `getByText`.

Glossary

user-event — A library that simulates user interactions (clicking, typing) closer to real browser behavior.

Mock Function — A fake function created with `vi.fn()` that stands in for a real function and records how it was called (arguments, call count).

Async Query — A query type, like `findBy*`, that WAITS for an element to appear in the DOM.

Practical Project

There's a real, runnable example project combining the concepts from this category (Component Testing, User Interaction Testing): **Testing Demo** -- a searchable course list and a signup form familiar from earlier categories, but this time the focus is less on the app itself and more on the real Vitest tests that verify it.

The project tests `SearchBar`, `CourseList`, and `EnrollForm` each in their own `.test.jsx` file (`getByLabelText` + `userEvent.type`, filtering with `getByText` / `queryByText`, form submission with `vi.fn()` + `findByText`), and verifies the whole thing -- wired together in `App` with lifting state up -- with a single integration test (`App.test.jsx`). You can download it and run it yourself, and read through the tests line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/testing
npm test
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, just `cd react-course-projects/projects/testing` and run `npm test`.