Component Testing

Setting up Vitest and React Testing Library, your first test with render+screen, querying with getByRole/getByLabelText, jest-dom matchers, and testing conditional rendering -- with simple examples.

Intermediate 12 min

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

---

Component Testing

So far we've checked every component by clicking around in the browser by hand. That's fine for a handful of components, but as an app grows, manually re-checking every screen after every change is slow and unreliable. This lesson teaches you to verify that components work correctly AUTOMATICALLY, with code.

Setting Up Vitest and React Testing Library

We use two libraries in this course:

- **Vitest** — the tool that RUNS tests (it provides functions like `describe`, `it`, `expect`). Since it's built for Vite projects, it needs almost no extra configuration.
- **React Testing Library (RTL)** — the library that lets you MOUNT a component into a fake DOM (jsdom) and then QUERY that DOM the way a real user would see it.

To add them to a Vite project:

```
npm install -D vitest @testing-library/react @testing-library/jest-dom jsdom
```

Add a `test` block to `vite.config.js`:

```
export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
    setupFiles: ["./src/setupTests.js"],
    globals: true,
  },
});
```

`environment: "jsdom"` makes tests run against a fake DOM inside Node instead of a real browser. The setup file only needs one line:

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

This line ADDS extra assertions (like the `toBeInTheDocument()` we'll see shortly) to Vitest's `expect`.

Our First Test with render() and screen

The most basic test skeleton mounts a component into the fake DOM and checks that something we expect is there:

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

// The component under test. In a real project this usually lives in its own
// file (Counter.jsx), with the test written in a separate file
// (Counter.test.jsx) -- here we've combined the two into a single readable example.
function Counter() {
  return (
    <div>
      <p>Count: 0</p>
    </div>
  );
}

describe("Counter", () => {
  it("renders the initial count", () => {
    // render() places the component into a real DOM (jsdom, a SIMULATED browser).
    render(<Counter />);

    // screen is used to QUERY the current DOM. If getByText can't find an
    // element containing exactly this text, it fails the test IMMEDIATELY.
    expect(screen.getByText("Count: 0")).toBeInTheDocument();
  });
});
```

`describe` groups related tests together; `it` (or `test`) defines a single test case. `render(<Counter />)` mounts the component into jsdom. `screen` is used to QUERY that DOM -- `getByText` immediately fails the test if it can't find an element containing the given text.

Querying with getByRole and getByLabelText

`getByText` isn't always the best query -- RTL offers queries that are closer to how a real user (or a screen reader) PERCEIVES the page:

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

function LoginButton() {
  return <button>Log In</button>;
}

function NameField() {
  return (
    <div>
      <label htmlFor="name">Name</label>
      <input id="name" defaultValue="Ada" />
    </div>
  );
}

describe("Querying elements", () => {
  it("finds a button by its accessible role and name", () => {
    render(<LoginButton />);

    // getByRole finds elements by their ACCESSIBILITY role, not by their
    // VISIBLE text -- a <button> has the "button" role. This is the query
    // style that most closely matches how real users (and screen readers)
    // perceive the page.
    expect(screen.getByRole("button", { name: /log in/i })).toBeInTheDocument();
  });

  it("finds a form field by its connected label", () => {
    render(<NameField />);

    // getByLabelText finds the input that matches a <label htmlFor="...">
    // -- there's no need to add the input's id or a test-id.
    expect(screen.getByLabelText("Name")).toHaveValue("Ada");
  });
});
```

`getByRole("button", { name: /log in/i })` finds a `<button>` by its accessibility role and visible name -- RTL's official docs recommend `getByRole` as the PREFERRED query whenever possible. `getByLabelText("Name")` finds the input connected to `<label htmlFor="name">`, with no need to add an id or a test-id.

jest-dom Matchers

The `@testing-library/jest-dom/vitest` import we added during setup adds new DOM-specific assertions to `expect`:

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

function SubmitButton({ disabled }) {
  return <button disabled={disabled}>Submit</button>;
}

describe("SubmitButton", () => {
  it("is disabled when the disabled prop is true", () => {
    render(<SubmitButton disabled={true} />);

    // toBeDisabled/toBeEnabled/toBeInTheDocument are matchers added by
    // @testing-library/jest-dom -- they're not in plain Vitest, and are
    // installed separately in projects that use jsdom (via
    // "@testing-library/jest-dom/vitest" in setupFiles).
    expect(screen.getByRole("button", { name: /submit/i })).toBeDisabled();
  });

  it("is enabled when the disabled prop is false", () => {
    render(<SubmitButton disabled={false} />);

    expect(screen.getByRole("button", { name: /submit/i })).toBeEnabled();
  });
});
```

`toBeDisabled()` and `toBeEnabled()` check an element's `disabled` attribute; `toBeInTheDocument()` verifies whether an element exists in the DOM at all. None of these exist in plain Vitest -- they're matchers added by the jest-dom package specifically for testing the DOM.

Testing Conditional Rendering

The conditional-rendering pattern from State & Events is one of the most commonly tested scenarios -- we verify that each state shows the CORRECT text:

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

// A tested version of the conditional rendering pattern from the State & Events lesson.
function StatusMessage({ status }) {
  if (status === "loading") return <p>Loading...</p>;
  if (status === "error") return <p>Something went wrong.</p>;
  return <p>Data loaded successfully.</p>;
}

describe("StatusMessage", () => {
  it("shows a loading message", () => {
    render(<StatusMessage status="loading" />);
    expect(screen.getByText("Loading...")).toBeInTheDocument();

    // Unlike getByText, queryByText does NOT THROW an error when it can't find
    // something -- it returns null. queryBy* is used to verify that something
    // is NOT ON THE SCREEN.
    expect(screen.queryByText("Data loaded successfully.")).not.toBeInTheDocument();
  });

  it("shows an error message", () => {
    render(<StatusMessage status="error" />);
    expect(screen.getByText("Something went wrong.")).toBeInTheDocument();
  });

  it("shows the success message by default", () => {
    render(<StatusMessage status="success" />);
    expect(screen.getByText("Data loaded successfully.")).toBeInTheDocument();
  });
});
```

Three separate `it` blocks render the component with three different values of the `status` prop and check that the right message shows up each time. The first test also uses `queryByText`: unlike `getByText`, it does NOT throw if the element isn't found -- it returns `null` -- which is why `queryBy*` (not `getByText`) is used to assert that something is ABSENT from the screen.

Summary and Glossary

Vitest RUNS tests; React Testing Library lets you mount components into a fake DOM and QUERY them. `render()` mounts a component into the DOM; `screen` is used to query that DOM. `getByRole` / `getByLabelText` / `getByText` throw if they can't find an element; the `queryBy*` variants return `null` instead and are used to assert that something is ABSENT. `@testing-library/jest-dom` adds DOM-specific matchers like `toBeInTheDocument()`.

Glossary

Test Runner — The tool that discovers and runs tests and reports the results (Vitest).

jsdom — A fake DOM environment that runs inside Node and SIMULATES a real browser.

Matcher — A function chained after `expect(...)` that verifies a specific condition (`toBeInTheDocument()` , `toHaveValue()` , etc.).