Frontend Testing

A Complete 9-Chapter Web Development Course

Topics covered:

The Testing Pyramid & Jest/Vitest Fundamentals

The Testing Library Philosophy & Component Testing

Hooks, State & Mocking Dependencies

Testing Across Frameworks · Integration & Coverage

Exercises: 27 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · centered on React Testing Library

Single standalone course · pairs with the site's React/Vue/Angular/Svelte courses

Table of Contents

01 Why Test Frontend Code? The Testing Pyramid & Types of Tests

02 Jest & Vitest Fundamentals

03 The Testing Library Philosophy

04 Testing React Components

05 Testing Hooks & State

06 Mocking Dependencies

07 Testing Across Frameworks

08 Integration & Coverage

09 Capstone: Testing a Real Feature

CHAPTER 1 OF 9

Why Test Frontend Code? The Testing Pyramid & Types of Tests

COURSE 1 · CH 1

Why Test Frontend Code? The Testing Pyramid & Types of Tests

What makes UI code harder to test than a plain function — and the vocabulary this whole course builds on

A pure backend function is easy to test: give it an input, check the output. A UI component is judged by something less direct — what a real person can *see* and *do* with it. This chapter lays out why that difference matters, and introduces the vocabulary — the testing pyramid, and the four common types of tests — that the rest of this course is built on.

Why Frontend Code Needs Its Own Testing Approach

A component doesn't just return a value — it renders DOM, responds to clicks and typing, and often waits on something asynchronous (an API call resolving, a loading spinner disappearing) before the "real" result even appears. Correctness for a UI component means: does the right thing appear on screen, and does it respond the way a real user would expect when they interact with it? That's a fundamentally different question than "does this function return the right number," and it needs its own set of tools and habits — the subject of this entire course.

The Testing Pyramid

A classic model for thinking about a test suite's shape, from the bottom up:

LAYER	SPEED	REALISM	HOW MANY
Unit tests	Fastest	Lowest — isolated logic only	Most
Component / Integration tests	Moderate	Medium — real rendering, simulated interaction	Fewer
End-to-End (E2E) tests	Slowest	Highest — a real browser, a full user journey	Fewest

The shape is a trade-off, not an accident: tests closer to the real user experience are more convincing when they pass, but slower to run and more expensive to write — which is exactly why there are usually far fewer of

them.

The Four Types of Tests, Concretely

1. Unit Tests

A single function or utility, no DOM involved at all — e.g. testing that `formatCurrency(1050)` returns `"$10.50"`.

2. Component Tests

Rendering *one* component in isolation, simulating interaction, asserting on what appears — this course's primary focus, starting Chapter 3.

3. Integration Tests

Multiple pieces working together — a form, its submit handler, and a list that updates in response, all exercised in one test.

4. End-to-End (E2E) Tests

A real (or real-like) browser driving a full user journey — login, add an item, checkout. Tools: Playwright, Cypress — explicitly **out of scope** for this course.

This Course's Toolchain

Chapter 2 introduces **Jest** and **Vitest** as the test runners underneath everything. Chapters 3–5 build up **React Testing Library**, the tool this course spends the most time on, for component-level testing. Chapter 6 covers mocking dependencies with **MSW (Mock Service Worker)**. Chapter 7 steps back to compare Vue Test Utils, Angular's TestBed, and Svelte Testing Library — since this site already has complete courses for all three frameworks.

THE PYRAMID ISN'T A STRICT LAW

Kent C. Dodds — the creator of Testing Library, this course's central tool — has argued for a different shape entirely: the "testing trophy," which weights *component/integration* tests more heavily than pure unit tests, precisely because modern component-testing tools have made them cheap and highly confidence-inspiring. The pyramid is a useful starting mental model, not a rule this course enforces rigidly.

THE #1 BEGINNER MISTAKE: TESTING IMPLEMENTATION, NOT BEHAVIOR

Asserting that a component's internal state variable holds a specific value, or that a specific CSS class was applied, tests *how* the component happens to be built right now — not what a real user actually experiences. These tests break constantly during harmless refactors, even when the app still works perfectly. Chapter 3's entire philosophy exists to fix this exact habit before it forms.

Coding Challenges

CHALLENGE 1

For each of the following, classify it as a unit, component, integration, or E2E test, and explain your reasoning: (a) testing a validateEmail(string) function, (b) testing that a Button component renders its label text, (c) testing a full checkout flow across three pages in a real browser.

View solution

CHALLENGE 2

Write a short paragraph explaining why a test that asserts wrapper.find('.is-active').length === 1 (checking a CSS class directly) is more fragile than a test that asserts the user can see text confirming an item is selected.

View solution

CHALLENGE 3

Explain, in your own words, the trade-off the testing pyramid describes — specifically, why a team wouldn't just write nothing but E2E tests, given that they're the most realistic.

View solution

CHAPTER 1 QUICK REFERENCE

- **Unit test** — isolated logic, no DOM, fastest and most numerous
- **Component test** — one component rendered and interacted with in isolation (this course's main focus)
- **Integration test** — multiple components/pieces exercised together
- **E2E test** — a real browser, a full user journey, slowest and fewest (out of scope here — Playwright/Cypress territory)
- The testing pyramid trades realism for speed as you go up — no single layer is "the right one" on its own
- The "testing trophy" (Kent C. Dodds) weights component/integration tests more heavily — a legitimate alternative shape, not a contradiction
- Test **behavior a user can observe**, not internal implementation details — the habit Chapter 3 builds on directly
- **Next chapter:** Jest & Vitest Fundamentals — test runner basics, assertions, and setup/teardown

CHAPTER 2 OF 9

Jest & Vitest Fundamentals

COURSE 1 · CH 2

Jest & Vitest Fundamentals

The test runner mechanics underneath every test this course writes, before the DOM ever enters the picture

Chapter 1 established *why* and *what* to test. This chapter is the *how* — the test runner mechanics that every later chapter builds on, kept deliberately DOM-free for now so the focus stays on the runner itself.

Anatomy of a Test File

```
describe("add", () => {
  it("adds two positive numbers", () => {
    expect(add(2, 3)).toBe(5);
  });
});
```

`describe` groups related tests under a label. `it` (or `test` — the two are interchangeable) defines one individual test case. `expect(...)` wraps a value so a **matcher** like `.toBe(...)` can assert something about it.

Common Matchers

```
expect(5).toBe(5);                       // strict equality — primitives
expect({ name: "Ada" }).toEqual({ name: "Ada" });  // deep equality — objects/arrays
expect("hello").toBeTruthy();
expect([1, 2, 3]).toContain(2);
expect(() => riskyCall()).toThrow();
```

The `toBe` vs. `toEqual` distinction trips up almost everyone early on — covered concretely below.

Setup and Teardown

```
let counter;

beforeEach(() => {
  counter = 0;  // fresh state before EVERY test — no leakage between tests
});

it("increments", () => {
  counter++;
  expect(counter).toBe(1);
});
```

`beforeEach` / `afterEach` run before or after every single test; `beforeAll` / `afterAll` run once for the whole `describe` block. Resetting shared state before each test keeps tests **isolated** — one test's leftover state should never affect another, a concern that becomes far more important once mocking enters the picture.

Basic Mocking with jest.fn() / vi.fn()

A quick preview of Chapter 6's deeper territory — here, just the raw mechanic: a mock function records how it was called, without needing any real implementation behind it.

```
const onSubmit = jest.fn();

handleFormSubmit({ email: "a@example.com" }, onSubmit);

expect(onSubmit).toHaveBeenCalled();
expect(onSubmit).toHaveBeenCalledWith({ email: "a@example.com" });
expect(onSubmit).toHaveBeenCalledTimes(1);
```

Vitest: Jest's Vite-Native Sibling

Vitest was built specifically for Vite-based projects — faster, native ESM, no separate Babel/webpack transform step — and deliberately copied Jest's API almost one-to-one. `describe`, `it`, and `expect` work identically; the one real naming difference in the basics is `jest.fn()` becoming `vi.fn()`. This course uses Jest's naming by convention (it remains the more common baseline), but everything transfers directly to Vitest with that single substitution.

PIECE	JEST	VITEST
Grouping / test case / assertion	**describe / it / expect**	**Identical**
Mock function	**jest.fn()**	**vi.fn()**
Config file	**jest.config.js**	**vite.config.ts (test block)**

beforeEach / afterEach

Runs before/after every individual test in the block — the default choice for resetting shared state.

beforeAll / afterAll

Runs once for the entire `describe` block — for expensive setup that's safe to share across tests.

WATCH MODE IS THE DEFAULT WORKFLOW

Both `jest --watch` and plain `vitest` (which watches by default) re-run only the tests affected by your latest change, instantly. This is what makes Chapter 1's "unit tests are fast" point actually pleasant in practice — write a small change, glance at the terminal, know within a second whether it broke anything.

TOBE VS. TOEQUAL: THE CLASSIC FIRST MISTAKE

`expect([1, 2, 3]).toBe([1, 2, 3])` **fails** — even though the arrays "look the same." `toBe` checks reference identity (are these the exact same object in memory), and two separately-created arrays never share a reference. Use `toEqual` for comparing the contents of objects and arrays; reserve `toBe` for primitives (numbers, strings, booleans) where reference and value are the same thing.

Coding Challenges

CHALLENGE 1

Write a describe/it test file for a function multiply(a, b), covering a positive-number case and a case multiplying by zero.

View solution

CHALLENGE 2

Write a test using beforeEach to reset an array-based "cart" to empty before each test, then verify that adding one item results in a cart of length 1 — proving the reset actually runs between tests, not just once.

View solution

CHALLENGE 3

Write a test using jest.fn() to verify that a function processItems(items, callback) calls its callback once per item, with each item as the argument — using toHaveBeenCalledTimes and toHaveBeenCalledWith.

View solution

CHAPTER 2 QUICK REFERENCE

- **describe(name, fn)** — groups related tests; **it/test(name, fn)** — one test case
- **expect(value).matcher(...)** — toBe (strict/reference equality), toEqual (deep equality), toBeTruthy/toBeFalsy, toContain, toThrow
- **beforeEach/afterEach** — run per test; **beforeAll/afterAll** — run once per describe block
- **jest.fn() / vi.fn()** — a mock function; toHaveBeenCalled/toHaveBeenCalledWith/toHaveBeenCalledTimes inspect its calls
- **Vitest** — Jest's Vite-native sibling; near-identical API, jest.fn() → vi.fn() is the main naming change
- toBe checks reference identity; use toEqual for objects/arrays, or a confusing failure is guaranteed
- Watch mode (default in Vitest, `--watch` in Jest) is the everyday workflow, not an optional extra
- **Next chapter:** The Testing Library Philosophy — testing like a user, not like the implementation

CHAPTER 3 OF 9

The Testing Library Philosophy

COURSE 1 · CH 3

The Testing Library Philosophy

The tool built specifically around Chapter 1's "test behavior, not implementation" principle

Chapter 1 named the fix; Chapter 2 built the raw test-runner mechanics. This chapter introduces the actual tool built around that philosophy — **React Testing Library (RTL)** — and the philosophy itself, before any real component interaction begins in Chapter 4.

The Guiding Principle

Testing Library's own tagline, from creator Kent C. Dodds, states the whole philosophy in one line: "*The more your tests resemble the way your software is used, the more confidence they can give you.*" Concretely, that means testing through the same interface a real user has — what's visible on screen, what can be clicked or typed into — never through a component's internal React state, props, or methods.

render(): Mounting a Component Into a Test DOM

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

render(<Greeting name="Ada" />);
expect(screen.getByText("Hello, Ada")).toBeInTheDocument();
```

`render(...)` mounts a component into a lightweight, jsdom-based test DOM and returns the `screen` object, used to query it. Older tools like **Enzyme** — React's previous, now-deprecated testing library, still referenced in a lot of older tutorials — deliberately exposed internals like `.state()` and `.instance()`. RTL deliberately does **not** provide any way to reach into a component's internals at all. That's a design choice, not a missing feature.

Querying by Role, Text, and Label

```
screen.getByRole("button", { name: "Submit" });
screen.getByText("Welcome back");
screen.getByLabelText("Email");
```

Every one of these queries something a real user (or assistive technology) actually perceives — an ARIA role, visible text, a form label — rather than a CSS class or internal prop, directly resolving Chapter 1's CSS-class gotcha. This connects concretely to `web-accessibility1`'s ARIA material: writing accessible markup in the first place is what makes a component queryable by role or label at all. Accessibility and testability reinforce each other by construction, not by coincidence.

The getBy / queryBy / findBy Trio

PREFIX	IF NOT FOUND	USE WHEN
`getBy...`	Throws immediately	The element should already exist right now
`queryBy...`	Returns null	Asserting something does NOT exist
`findBy...`	Returns a rejected Promise (after retrying)	Waiting for async content to appear

The Query Priority Guide

Testing Library's own documented, recommended order — not just this course's opinion:

1. **getByRole** — first choice; matches how assistive technology and most users actually perceive an element
2. **getByLabelText / getByPlaceholderText / getByText** — strong alternatives when a role query doesn't fit
3. **getByTestId** — an explicit *last resort*, an escape hatch for when nothing else works, not a default habit

render() + screen

Mounts a component into a jsdom test environment; `screen` is the object every query is called on.

No Internals Exposed

Unlike Enzyme, RTL provides no way to reach a component's state/instance directly — by design.

getByRole First

The top of the official priority order — the query closest to how a real user or screen reader perceives the page.

getByTestId Last

Invisible to real users and assistive tech alike — a deliberate escape hatch, not a first choice.

HARD TO QUERY = OFTEN A REAL ACCESSIBILITY SIGNAL

If a component is genuinely hard to query with `getByRole` or `getByLabelText`, that's frequently a real sign the markup itself isn't accessible enough — a missing label, a wrong or missing ARIA role. RTL's design turns "write

testable code" and "write accessible code" into nearly the same habit, a direct, practical payoff of the site's own `web-accessibility1` course.

REACHING FOR GETBYTESTID BY DEFAULT DEFEATS THE POINT

A `data-testid` attribute is invisible to real users and assistive technology alike — over-relying on it produces tests that pass even when a real user genuinely couldn't perceive or interact with the element the same way. It's not wrong to use occasionally, but it should be a deliberate last resort, not a habit — a common, real mistake among teams migrating from Enzyme's more implementation-focused style.

Coding Challenges

CHALLENGE 1

For a component rendering a "Save" button, write the getByRole query that would find it, and explain why getByRole('button', {name: 'Save'}) is preferred over getByText('Save').

View solution

CHALLENGE 2

Explain which query prefix (getBy, queryBy, or findBy) you'd use for each: (a) confirming an error message is NOT shown before form submission, (b) confirming a submit button exists immediately after rendering, (c) confirming a "Success!" message appears after an async save completes.

View solution

CHALLENGE 3

A component renders a div with class "error-message" containing the text "Invalid email" — but no ARIA role and no associated label. Explain why this is both a testing problem and an accessibility problem, and what change would fix both at once.

View solution

CHAPTER 3 QUICK REFERENCE

- **Guiding principle:** test the way a real user uses the software, never internal implementation details
- **render(<Component />)** — mounts into a jsdom test DOM; returns/enables the `screen` query object
- RTL deliberately exposes **no** way to reach component internals (state/instance) — unlike the older, deprecated Enzyme

- **Query priority:** getByRole first, then getByLabelText/getByPlaceholderText/getByText, getByTestId as a last resort only
- **getBy*** — throws if missing; **queryBy*** — returns null if missing; **findBy*** — async, waits for appearance
- Hard-to-query markup is often a real accessibility gap, not just a testing inconvenience
- **Next chapter:** Testing React Components — rendering, firing events, conditional rendering, and snapshot testing's pitfalls

CHAPTER 4 OF 9

Testing React Components

COURSE 1 · CH 4

Testing React Components

Putting Chapter 3's philosophy into practice — rendering, firing real interactions, and the truth about snapshot testing

Chapter 3 established the philosophy and the query system. This chapter puts both into practice — rendering real components, simulating genuine user interaction, and asserting on what actually changes on screen.

Rendering and Asserting on Output

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

it("greets the given name", () => {
  render(<Greeting name="Ada" />);
  expect(screen.getByText("Hello, Ada!")).toBeInTheDocument();
});
```

Firing Events: fireEvent vs. userEvent

`fireEvent` dispatches a single, low-level DOM event directly. `userEvent` (from `@testing-library/user-event`) simulates the full, realistic *sequence* of events a real browser fires for that interaction — typing one character involves `keydown`, `keypress`, `input`, and `keyup` in order; a click involves pointer and focus events too. Testing Library's own current recommendation: prefer `userEvent` whenever possible, since it more closely resembles real usage — directly extending Chapter 3's guiding principle.

```
// Preferred — realistic event sequence
await userEvent.click(screen.getByRole("button", { name: "Submit" }));
await userEvent.type(screen.getByLabelText("Email"), "ada@example.com");

// Lower-level — dispatches exactly one raw event
fireEvent.change(screen.getByLabelText("Email"), { target: { value: "ada@example.com" } });
```

fireEvent vs. userEvent

fireEvent

Dispatches exactly one raw DOM event. Fast, but can miss real browser behavior (e.g. skipping focus changes).

userEvent (preferred)

Simulates the full, realistic event sequence a real user's action would trigger — closer to actual usage.

Asserting on Conditional Rendering

Combining Chapter 3's query trio with real interaction — confirming something is absent, triggering a change, then confirming it appears:

```
it("shows an error only after an invalid submission", async () => {
  render(<SignupForm />);

  expect(screen.queryByText("Email is required")).not.toBeInTheDocument();

  await userEvent.click(screen.getByRole("button", { name: "Sign Up" }));

  expect(await screen.findByText("Email is required")).toBeInTheDocument();
});
```

Snapshot Testing: What It Is

`expect(container).toMatchSnapshot()` captures the entire rendered output into a stored file — future runs diff against it and flag *any* difference. The appeal is real: fast to write, catches unintended changes. The real problem is just as real: a snapshot captures **everything**, including implementation details Chapter 1 and 3 have been warning about all along. A tiny, intentional style tweak can produce a giant diff that gets rubber-stamped without real review — a well-known failure mode called **"snapshot blindness."**

When Snapshots Are (and Aren't) a Good Fit

Targeted Assertions vs. Full Snapshots

Targeted getBy/queryBy Assertions

Specific, meaningful, and resistant to unrelated changes — the right default for behavior.

Full Snapshots

Broad, brittle, and prone to blind "update and move on" fixes — reserve for small, stable, presentational components only.

fireEvent

One raw DOM event — useful for low-level cases userEvent doesn't cover.

userEvent (preferred)

A realistic multi-event sequence matching actual browser behavior — the default choice.

Conditional Rendering Pattern

queryBy to confirm absence → interact → getBy/findBy to confirm appearance.

Snapshot Pitfall

Captures everything, including irrelevant details — easy to blindly "update" without real review.

USEREVENT METHODS ARE ASYNC

Modern `userEvent` (v14+) methods return Promises, mirroring real browser event timing — always `await userEvent.click(...)` / `await userEvent.type(...)`. Forgetting `await` doesn't throw an error; it just silently lets the test move on before the interaction has actually finished, producing confusing, intermittent failures.

BLINDLY PRESSING "UPDATE SNAPSHOT" DEFEATS THE WHOLE POINT

Running `--updateSnapshot` (or pressing `u` in watch mode) without actually reading the diff turns a snapshot test into a rubber stamp — it approves whatever the new output happens to be, intentional bug or not. A snapshot test only provides value if a real human actually looks at what changed before accepting it as correct.

Coding Challenges

CHALLENGE 1

Write a test for a Counter component that starts at 0 and increments when a button labeled "Increment" is clicked, using userEvent and getByRole/getByText, confirming the displayed count changes from "0" to "1".

View solution

CHALLENGE 2

Write a test for a PasswordField component that shows a "Passwords do not match" message only when a confirm-password input differs from the password input, using queryByText to confirm the message's absence and presence at the right moments.

View solution

CHALLENGE 3

Explain why a snapshot test of a large, frequently-edited form component is more likely to cause "snapshot blindness" than a snapshot test of a small, rarely-changed Icon component — and propose a more targeted alternative test for the form component.

View solution

CHAPTER 4 QUICK REFERENCE

- **fireEvent** — dispatches one raw DOM event; **userEvent** (preferred) — simulates a realistic multi-event sequence
- **await** every userEvent call — its methods are async, mirroring real browser timing
- **Conditional rendering pattern:** queryBy (confirm absence) → interact → getBy/findBy (confirm appearance)
- **toMatchSnapshot()** — captures full rendered output; diffs on every future run
- Snapshots capture implementation details too — small/stable/presentational components are the right fit, not large or frequently-changing ones
- Blindly accepting a snapshot update without reading the diff defeats the test's entire purpose
- **Next chapter:** Testing Hooks & State — renderHook, useState/useEffect, controlled forms, and waiting for async updates

CHAPTER 5 OF 9

Testing Hooks & State

COURSE 1 · CH 5

Testing Hooks & State

Testing logic with no rendered output of its own, and components whose behavior depends on timing

Chapter 4 tested components as black boxes through their rendered output. This chapter covers two more specific situations: testing a **custom hook** in isolation — which has no JSX of its own at all — and testing components whose correctness depends on `useState` / `useEffect` timing, including controlled forms and asynchronous updates.

Testing Custom Hooks with renderHook

A custom hook like `useCounter` isn't a component — `render()` and `screen` don't apply. `renderHook` mounts a hook inside a minimal, invisible test component behind the scenes, returning a `result` object whose `.current` holds the hook's live return value:

```
import { renderHook, act } from "@testing-library/react";

it("increments the count", () => {
  const { result } = renderHook(() => useCounter());

  expect(result.current.count).toBe(0);

  act(() => {
    result.current.increment();
  });

  expect(result.current.count).toBe(1);
});
```

`act(...)` tells React "a state update just happened here — flush it and re-render before continuing." Without it, an assertion immediately after calling an updater function can read a **stale** value from before React has actually processed the update.

Testing Components That Use useState

A controlled input's displayed value depends on component state that changes as the user types — combining Chapter 4's `userEvent.type` with a query that reads the input's current value:

```
it("updates the input as the user types", async () => {
  render(<SearchBox />);
  const input = screen.getByLabelText("Search");

  await userEvent.type(input, "react testing");

  expect(input).toHaveValue("react testing");
});
```

Testing useEffect Timing

A component that fetches data in `useEffect` doesn't show its result synchronously — this is exactly where Chapter 3's `findBy` and the more general `waitFor` matter. `findBy*` is a convenience wrapper built around *one* query; `waitFor(callback)` is general-purpose — it re-runs an arbitrary assertion callback until it passes or times out, useful when a check can't be expressed as a single query.

```
// Simple case — one query eventually succeeds
expect(await screen.findByText("Ada Lovelace")).toBeInTheDocument();

// More complex — two conditions checked together, not expressible as one query
await waitFor(() => {
  expect(fetchUser).toHaveBeenCalledTimes(1);
  expect(screen.queryByText("Loading...")).not.toBeInTheDocument();
});
```

Controlled Forms End to End

Combining everything: typing into multiple controlled inputs, submitting, and confirming an async result:

```
it("submits the form and shows a success message", async () => {
  render(<ProfileForm />);

  await userEvent.type(screen.getByLabelText("Name"), "Ada");
  await userEvent.type(screen.getByLabelText("Email"), "ada@example.com");
  await userEvent.click(screen.getByRole("button", { name: "Save" }));

  expect(await screen.findByText("Profile saved!")).toBeInTheDocument();
});
```

renderHook

act()

Mounts a custom hook with no JSX of its own; `result.current` holds its live return value.

Flushes a state update before assertions continue — required when calling an updater outside RTL's own built-in helpers.

findBy* (simple)

A single query, built-in retry — the right choice when one thing needs to eventually appear.

waitFor (flexible)

Wraps an arbitrary assertion callback — the right choice when multiple conditions need checking together.

MODERN RTL AUTO-WRAPS MOST THINGS IN ACT()

Older tutorials show explicit `act(...)` calls everywhere — but `render`, `fireEvent`, and `userEvent` now wrap themselves in `act` internally. Explicit `act()` is mainly needed for `renderHook`'s own updater calls (as above) or when triggering a state change through something entirely outside RTL's built-in helpers.

FORGETTING AWAIT ON FINDBY/WAITFOR PRODUCES FLAKY, NOT BROKEN, TESTS

Both `findBy*` and `waitFor` return Promises. Forgetting `await` doesn't throw a clear error — it lets the test's assertions run against the wrong moment in time, before the update has actually finished. The result is a test that sometimes passes and sometimes fails depending on timing — a genuinely nasty class of bug to diagnose, and the exact same category of mistake as Chapter 4's forgotten-`await`-on-userEvent gotcha, just showing up again here.

Coding Challenges

CHALLENGE 1

Write a renderHook test for a custom hook useToggle() that returns { value, toggle }, starting at false, confirming that calling toggle() once sets value to true and calling it again sets it back to false.

View solution

CHALLENGE 2

Write a test for a component that fetches a list of items on mount (in useEffect) and shows "Loading..." until they arrive. Use findByText to confirm an item eventually appears, and queryByText to confirm "Loading..." is gone by then.

View solution

CHALLENGE 3

Explain why forgetting await on a findByText call can make a test pass sometimes and fail other times, rather than failing consistently and obviously — referencing what the test actually does if the Promise is never awaited.

📄 View solution

CHAPTER 5 QUICK REFERENCE

- **renderHook(() => useX())** — mounts a custom hook; `result.current` holds its live value
- **act(() => { ... })** — flushes a state update; needed for renderHook updater calls, largely automatic elsewhere now
- **toHaveValue(...)** — asserts a controlled input's current value
- **findBy*** — single query, built-in retry, for one thing appearing eventually
- **waitFor(callback)** — general-purpose retry for arbitrary/multiple assertions together
- Modern render/fireEvent/userEvent auto-wrap in act() — explicit act() is the exception now, not the rule
- Forgetting await on findBy/waitFor produces flaky tests, not clear errors — the same risk class as Chapter 4's userEvent gotcha
- **Next chapter:** Mocking Dependencies — mocking fetch/axios, Mock Service Worker (MSW), timers, and context providers

CHAPTER 6 OF 9

Mocking Dependencies

COURSE 1 · CH 6

Mocking Dependencies

Where the data in Chapter 5's async tests actually comes from — and why a real network call has no place in a test suite

Chapter 5 tested components that fetch data, using `findBy` / `waitFor` to wait for it — but never addressed where that data actually came from during the test. This chapter closes that gap.

The Problem With Real Network Calls in Tests

A real network call in a test is slow, depends on an external service actually being up, and makes it hard to deliberately trigger error cases (a 500 response, a timeout) on demand. Unit and component tests are supposed to be fast and deterministic — directly the speed argument behind Chapter 1's testing pyramid — and a real network dependency violates both.

Mocking fetch/axios Directly

```
global.fetch = jest.fn().mockResolvedValue({
  json: async () => ({ name: "Ada" }),
});
```

This works, but couples the test tightly to *how* the request is made (specifically `fetch`, or specifically `axios`) rather than *what* the request and response actually are. Swapping the underlying HTTP client later means rewriting every test that mocked it this way.

Mock Service Worker (MSW): The Modern Preferred Approach

MSW intercepts requests at the **network level** rather than mocking the `fetch` / `axios` function itself — the component's real request code runs completely unmodified; only the server's response is faked. This is genuinely more realistic, a direct extension of Chapter 3's guiding principle applied to network behavior, and resilient to swapping HTTP libraries later.

```
import { http, HttpResponse } from "msw";
import { setupServer } from "msw/node";

const server = setupServer(
  http.get("/api/user", () => HttpResponse.json({ name: "Ada" }))
);

beforeAll(() => server.listen());
afterAll(() => server.close());
```

The `beforeAll` / `afterAll` lifecycle is Chapter 2's setup/teardown hooks, now managing a mock server's lifetime.

Simulating Error Responses

MSW makes deliberately testing error states trivial — override a handler for one test only:

```
it("shows an error message when the request fails", async () => {
  server.use(
    http.get("/api/user", () => HttpResponse.error())
  );

  render(<UserProfile />);

  expect(await screen.findByText("Could not load profile")).toBeInTheDocument();
});
```

Mocking Timers

Components using `setTimeout` / `setInterval` — a debounced search box, a toast that auto-dismisses — would otherwise force tests to actually wait real time, or behave flakily. `jest.useFakeTimers()` lets a test advance time instantly:

```
it("dismisses the toast after 3 seconds", () => {
  jest.useFakeTimers();
  render(<Toast message="Saved!" />);

  expect(screen.getByText("Saved!")).toBeInTheDocument();

  act(() => jest.advanceTimersByTime(3000));

  expect(screen.queryByText("Saved!")).not.toBeInTheDocument();
});
```

Mocking Context Providers

A component consuming React Context (e.g. `useAuth()`) needs that context available to render sensibly — wrapping it in a lightweight, test-only provider with preset fake values avoids needing a real login flow or real backend:

```
function renderWithProviders(ui, { authValue = { user: { name: "Ada" } } } = {}) {
  return render(
    <AuthContext.Provider value={authValue}>{ui}</AuthContext.Provider>
  );
}

renderWithProviders(<Dashboard />);
```

A reusable helper like this is worth introducing now — Chapters 8–9's more realistic feature tests will lean on it directly.

Manual fetch/axios Mocking vs. MSW

Manual Mocking

Replaces the HTTP client function itself — couples the test to *how* requests are made.

MSW (preferred)

Intercepts at the network level — the real request code runs unmodified; only the server's response is faked.

MSW Handlers

Define what a URL returns; override per-test with server.use() for error cases.

Fake Timers

jest.useFakeTimers() + advanceTimersByTime() — instant, deterministic time-based tests.

renderWithProviders

A test-only wrapper supplying fake context values instead of a real provider.

On-Demand Error Simulation

Something a real network call can't easily offer — a genuine testing advantage, not just a workaround.

MSW'S HANDLERS WORK BEYOND TESTS TOO

Because MSW intercepts at the network level, the same mock handlers can power actual local development — running the app against MSW instead of a real backend — not just automated tests. One set of fake API responses, two genuinely different uses.

FORGETTING TO RESET MSW HANDLERS LEAKS BETWEEN TESTS

A handler overridden with `server.use(...)` for one error-case test stays overridden for every test that runs afterward, unless explicitly reset — typically via `afterEach(() => server.resetHandlers())`. Forgetting this produces confusing failures in later, seemingly unrelated tests that unexpectedly hit the error response meant for a different test

entirely. This is Chapter 2's test-isolation principle (resetting shared state in `beforeEach`), now applying specifically to network mocks.

Coding Challenges

CHALLENGE 1

Set up an MSW handler for GET /api/products returning a list of two products, and write a test confirming both product names appear on screen after a ProductList component mounts.

View solution

CHALLENGE 2

Write a test using jest.useFakeTimers() and jest.advanceTimersByTime() for a component that shows "Are you still there?" after 60 seconds of inactivity — confirming the message is absent immediately after render and present after advancing time.

View solution

CHALLENGE 3

Explain, with a concrete example, how forgetting server.resetHandlers() in afterEach could cause a test that has nothing to do with errors to fail unexpectedly, if an earlier test in the same file used server.use() to simulate a failed request.

View solution

CHAPTER 6 QUICK REFERENCE

- Real network calls in tests are slow, flaky, and can't easily simulate error cases on demand
- **Manual fetch/axios mocking** — couples tests to how requests are made; works, but fragile to client swaps
- **MSW** (preferred) — intercepts at the network level; real request code runs unmodified
- **server.use(...)** — override a handler for one test (e.g. to simulate an error); reset with `server.resetHandlers()` in afterEach
- **jest.useFakeTimers() + advanceTimersByTime()** — instant, deterministic testing of setTimeout/setInterval-based behavior
- **renderWithProviders** — a reusable helper wrapping render() with fake context values instead of real providers
- Forgetting to reset MSW handlers leaks state between tests — the same isolation risk Chapter 2's beforeEach was built to prevent

- **Next chapter:** Testing Across Frameworks — Vue Test Utils, Angular's TestBed, and Svelte Testing Library

CHAPTER 7 OF 9

Testing Across Frameworks

COURSE 1 · CH 7

Testing Across Frameworks

Vue Test Utils, Angular's TestBed, and Svelte Testing Library — the same philosophy, four different APIs

Chapters 2–6 built deep, React-specific fluency with Testing Library. This chapter steps back: the same underlying philosophy — test like a user, not the implementation — shows up in Vue, Angular, and Svelte's own testing tools too, at varying distances from RTL's exact API. This site already has complete courses for all three frameworks; this chapter connects to them directly rather than re-teaching them.

Vue Test Utils

Vue's own official testing library, conceptually the closest of the three to RTL — `mount()` renders a component into a test environment, returning a `wrapper` to query. Vue's own community has increasingly adopted `@testing-library/vue`, a Testing-Library-flavored wrapper providing the exact same `getByRole` / `getByText` query style used throughout this course:

```
// @testing-library/vue — same query style as React
import { render, screen } from "@testing-library/vue";

render(Greeting, { props: { name: "Ada" } });
expect(screen.getByRole("heading")).toHaveTextContent("Hello, Ada");

// vs. raw Vue Test Utils — CSS selectors, the exact anti-pattern from Ch.1/3
const wrapper = mount(Greeting, { props: { name: "Ada" } });
expect(wrapper.find(".greeting-heading").text()).toBe("Hello, Ada");
```

Angular's TestBed

A fundamentally more ceremonial approach — Angular's dependency-injection-driven architecture means testing a component typically requires configuring a `TestBed` module (declaring the component, its dependencies, mock services) before it can even be instantiated:

```
  await TestBed.configureTestingModule({
    declarations: [GreetingComponent],
  }).compileComponents();

  const fixture = TestBed.createComponent(GreetingComponent);
  fixture.componentInstance.name = "Ada";
  fixture.detectChanges();  // Angular's change detection doesn't run automatically in a test
```

This is noticeably more setup than RTL's single `render()` call — a direct consequence of Angular's DI-heavy architecture, not a difference in testing philosophy. `@testing-library/angular` exists specifically to wrap this ceremony behind a more familiar `render()` + `screen` API.

Svelte Testing Library

`@testing-library/svelte` is the closest of all three to React's version — Svelte's compile-time component model needs neither Angular's DI ceremony nor Vue's separate wrapper API. `render()`, `screen`, and `fireEvent` / `userEvent` work almost identically to every example in Chapters 3–6.

The Pattern Across All Four

React's virtual DOM, Vue's reactive templates, Angular's DI-plus-zone-based change detection, and Svelte's compile-time approach are genuinely different component models under the hood. Despite that, every major framework's testing ecosystem has converged, to varying degrees, on the same Testing-Library-popularized idea: query by role/text/label, fire realistic events, assert on user-visible outcomes. Chapter 3's philosophy isn't a React quirk — it's closer to an industry-wide norm for component-based UI testing.

FRAMEWORK	NATIVE TOOL	TESTING-LIBRARY FLAVOR	SETUP CEREMONY
React	—	`@testing-library/react`	Minimal — `render()`
Vue	Vue Test Utils	`@testing-library/vue`	Minimal
Angular	TestBed	`@testing-library/angular`	Heavier — DI module setup
Svelte	—	`@testing-library/svelte`	Minimal

Vue

Vue Test Utils (wrapper.find) or @testing-library/vue (getByRole) — both common in real codebases.

Angular

TestBed + fixture.detectChanges() natively; @testing-library/angular hides the ceremony.

Svelte

The Constant

@testing-library/svelte — nearly identical to React's API, thanks to Svelte's simpler compiled model.

Query by role/text/label, fire realistic events, assert on outcomes — true across all four ecosystems.

THE PRACTICAL TAKEAWAY

Knowing RTL from this course makes Vue's and Svelte's Testing-Library variants a very short hop — same queries, same events, same philosophy. Angular's `TestBed` ceremony is the one genuinely different thing to learn, and it's rooted entirely in Angular's own DI architecture (the site's Angular course covers this architecture directly), not in a different idea about what makes a good test.

DON'T ASSUME EVERY CODEBASE USES THE TESTING-LIBRARY FLAVOR

Plenty of existing Vue and Angular code — and plenty of tutorials — still use raw Vue Test Utils' `wrapper.find('.class')` style or bare Angular `TestBed` + `fixture.debugElement.query(By.css(...))`. Recognizing both styles matters when reading real, existing test suites, not just when writing new ones from scratch.

Coding Challenges

CHALLENGE 1

Rewrite this chapter's raw Vue Test Utils example (wrapper.find('.greeting-heading')) using @testing-library/vue's getByRole instead, and explain what's gained by the rewrite.

View solution

CHALLENGE 2

Explain why Angular's TestBed setup is more involved than React's render() call — specifically, what Angular architectural feature (covered in this site's own Angular course) makes that extra ceremony necessary rather than accidental.

View solution

CHALLENGE 3

In your own words, explain why "component-based UI testing has converged on one philosophy across frameworks" is a stronger, more useful claim than "React Testing Library is a good tool" — referencing what this convergence tells you about testing skills you build in one framework.

View solution

CHAPTER 7 QUICK REFERENCE

- **Vue** — Vue Test Utils (wrapper.find) or @testing-library/vue (getByRole) — both used in real codebases
- **Angular** — TestBed.configureTestingModule + fixture.detectChanges(), or @testing-library/angular to hide the ceremony
- **Svelte** — @testing-library/svelte, nearly identical to React's API
- Angular's extra setup ceremony comes from its DI architecture, not a different testing philosophy
- The constant across all four: query by role/text/label, fire realistic events, assert on user-visible outcomes
- Real codebases mix raw native-tool style and Testing-Library style — recognize both
- **Next chapter:** Integration & Coverage — testing multiple components together, coverage thresholds, and accessibility testing with jest-axe

CHAPTER 8 OF 9

Integration & Coverage

COURSE 1 · CH 8

Integration & Coverage

Testing real component wiring, what coverage numbers actually mean, and closing the loop with accessibility

Chapter 1's testing pyramid named "integration tests" as its middle layer, but Chapters 2–7 have stayed mostly at the component level. This chapter finally writes a genuine integration test, then turns to two different topics: what code coverage actually measures (and where it misleads), and the accessibility-testing tool that closes the loop back to `web-accessibility1`.

From Component Tests to Integration Tests

A component test renders *one* component, often with mocked props/children. An integration test renders a parent alongside its **real** children, exercising how they actually communicate — using Chapter 6's MSW to fake the underlying API, since the point is exercising the real component wiring, not a real network:

```
it("updates results when a search is submitted", async () => {
  render(<SearchPage />);  // contains a REAL SearchForm + a REAL SearchResults

  await userEvent.type(screen.getByLabelText("Search"), "laptop");
  await userEvent.click(screen.getByRole("button", { name: "Search" }));

  expect(await screen.findByText("3 results for \"laptop\"")).toBeInTheDocument();
});
```

Neither `SearchForm` nor `SearchResults` is mocked — the test proves they genuinely work together, not just individually.

Code Coverage: What It Measures

`jest --coverage` reports what percentage of lines, branches, functions, and statements were *executed* during the test run — useful as a rough signal for finding obviously untested code, not as proof of correctness.

Why 100% Coverage Can Be Misleading

A line being *executed* says nothing about whether it was actually *asserted on correctly*. Consider a genuinely hollow test:

```
it("renders", () => {
  render(<UserProfile user={{ name: "Ada" }} />);
  // no assertions at all — yet every line UserProfile touches now counts as "covered"
});
```

This test contributes real coverage percentage while catching precisely zero bugs. 100% coverage is neither necessary nor sufficient for a genuinely well-tested codebase — coverage finds *obviously* neglected code (0% on an entire file is a real red flag), it doesn't validate that the tests which do exist are meaningful.

Setting Sensible Coverage Thresholds

Jest's `coverageThreshold` config option fails the test run if coverage drops below a set percentage — a practical floor (commonly 70–80%) that catches obviously-neglected code, not a target chased for its own sake. A team mandate of "100% coverage or the PR is blocked" actively incentivizes exactly the hollow test shown above.

Accessibility Testing with jest-axe

`jest-axe` runs automated accessibility checks — missing labels, invalid ARIA usage, and more — against rendered output, closing the loop directly to `web-accessibility1`:

```
import { axe, toHaveNoViolations } from "jest-axe";
expect.extend(toHaveNoViolations);

it("has no accessibility violations", async () => {
  const { container } = render(<SignupForm />);
  const results = await axe(container);
  expect(results).toHaveNoViolations();
});
```

Automated checks like `axe` catch only a **subset** of real accessibility problems — structural/markup issues, not things like whether an error message makes logical sense in the order a screen reader announces it. Manual testing, exactly as `web-accessibility1` covers, is still necessary — `axe` is a floor, not a substitute.

Coverage as a Signal vs. Coverage as a Target

As a Signal

Flags obviously untested files/branches — a useful early-warning tool.

As a Target

Chasing 100% actively rewards hollow, assertion-free tests that inflate the number without catching bugs.

Integration Test

A parent rendered with its real children, exercising actual communication between them.

Coverage Report

What percentage of code ran during tests — a signal for gaps, not a correctness proof.

coverageThreshold

A practical floor (e.g. 70–80%) that fails builds below it — not a 100% mandate.

jest-axe

Automated, structural accessibility checks — a floor, not a replacement for manual review.

ACCESSIBILITY TESTING TRANSFERS ACROSS FRAMEWORKS TOO

`axe-core` (the engine behind `jest-axe`) has bindings across the ecosystem — `cypress-axe`, framework-agnostic `axe-core` usage in Vue/Angular/Svelte test suites. The same automated-accessibility-as-a-floor approach transfers across frameworks, exactly like Chapter 7's core testing philosophy did.

A PASSING JEST-AXE CHECK DOESN'T MEAN "THIS IS ACCESSIBLE"

`axe` only catches automatically detectable issues — missing alt text, invalid ARIA, certain contrast problems. It cannot catch whether an error message's reading order actually makes sense to a screen reader user, or whether a keyboard-only flow is genuinely usable end to end. Treating a green `jest-axe` result as "done" directly contradicts `web-accessibility1`'s own point: automated tools are a floor, never a ceiling.

Coding Challenges

CHALLENGE 1

Write an integration test for a TodoApp component containing a real AddTodoForm and a real TodoList: typing a new todo and submitting should make it appear in the list, using no mocks for either child component.

View solution

CHALLENGE 2

Explain, with a concrete example different from this chapter's own, how a test could achieve 100% line coverage of a function while still failing to catch an obvious bug in that function's logic.

View solution

CHALLENGE 3

Write a jest-axe test for a component, then describe one realistic accessibility problem it would NOT catch, and briefly explain why axe is structurally unable to detect it.

View solution

CHAPTER 8 QUICK REFERENCE

- **Integration test** — a parent rendered with its real children, no mocking of the components themselves
- `jest --coverage` — reports line/branch/function/statement execution, not correctness
- 100% coverage is neither necessary nor sufficient — a hollow, assertion-free test still counts toward it
- **coverageThreshold** — set a practical floor (70-80%), not a 100% mandate that incentivizes hollow tests
- **jest-axe** — `axe(container)` + `toHaveNoViolations()` for automated accessibility checks
- Automated accessibility checks are a floor — manual testing (per web-accessibility1) is still required
- Both the coverage-as-target trap and the axe-is-enough trap share one lesson: automated numbers are a signal, not a substitute for real review
- **Next chapter:** Capstone — a complete test suite for a real feature, combining every tool from this course

CHAPTER 9 OF 9

Capstone: Testing a Real Feature

COURSE 1 · CH 9

Capstone: Testing a Real Feature

A complete test suite for a LoginForm, using every tool this course has built up

One feature, one complete test suite, every chapter's tool making an appearance: rendering, real user interaction, async mocking, conditional rendering, and an accessibility check.

The Feature Under Test

`LoginForm`: an email field and a password field, both required. Submitting calls `POST /api/login`. While the request is pending, the button reads "Signing in...". A failed login (401) shows an error message. A successful login shows a welcome message.

Setting Up MSW for the Capstone

```
const server = setupServer(
  http.post("/api/login", () => HttpResponse.json({ name: "Ada" }))
);

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());  // Ch.6's leaked-handler gotcha, avoided
afterAll(() => server.close());
```

The Full Test Suite

```
describe("LoginForm", () => {
  it("renders labeled email and password fields", () => {  // Ch.3
    render(<LoginForm />);
    expect(screen.getByLabelText("Email")).toBeInTheDocument();
    expect(screen.getByLabelText("Password")).toBeInTheDocument();
  });

  it("shows a validation message when submitted empty", async () => {  // Ch.4
    render(<LoginForm />);
    await userEvent.click(screen.getByRole("button", { name: "Log In" }));
    expect(await screen.findByText("Email and password are required")).toBeInTheDocument();
  });

  it("shows a pending state, then a welcome message, on success", async () => {  // Ch.5 + Ch.6
    render(<LoginForm />);
    await userEvent.type(screen.getByLabelText("Email"), "ada@example.com");
    await userEvent.type(screen.getByLabelText("Password"), "hunter2");
    await userEvent.click(screen.getByRole("button", { name: "Log In" }));

    expect(screen.getByRole("button", { name: "Signing in..." })).toBeInTheDocument();
    expect(await screen.findByText("Welcome, Ada!")).toBeInTheDocument();
  });

  it("shows an error message on a failed login", async () => {  // Ch.6
    server.use(http.post("/api/login", () => new HttpResponse(null, { status: 401 })));
    render(<LoginForm />);
    await userEvent.type(screen.getByLabelText("Email"), "ada@example.com");
    await userEvent.type(screen.getByLabelText("Password"), "wrong");
    await userEvent.click(screen.getByRole("button", { name: "Log In" }));

    expect(await screen.findByText("Invalid email or password")).toBeInTheDocument();
  });

  it("has no accessibility violations", async () => {  // Ch.8
    const { container } = render(<LoginForm />);
    expect(await axe(container)).toHaveNoViolations();
  });
});
```

What This Capstone Demonstrates

TEST	CHAPTER'S TOOL APPLIED
Labeled fields render	**Ch.3 — query by role/label**
Empty-submission validation	**Ch.4 — userEvent + conditional rendering**
Pending → success flow	**Ch.5 + Ch.6 — findBy/async + MSW**

TEST	CHAPTER'S TOOL APPLIED
Failed-login error	**Ch.6 — server.use() error override**
Accessibility check	**Ch.8 — jest-axe**

Chapter 2's runner mechanics (`describe` / `it` / `expect`) sit underneath every single one of these, and Chapter 7's point stands too — this exact same suite, translated to `@testing-library/vue` or `@testing-library/svelte`, would look nearly identical.

A GOOD TEST SUITE READS LIKE A SPECIFICATION

A new developer could read this suite's five test names — with no access to `LoginForm`'s actual implementation — and know exactly what it's supposed to do: labeled fields, required-field validation, a pending state, a success path, a failure path, and an accessibility baseline. That's the real, practical payoff of everything this course has built toward: tests that document behavior, not just verify it.

HONEST SCOPE: THIS ISN'T AN E2E TEST

This suite tests `LoginForm` in relative isolation — no real routing, no real backend, no verification that a successful login actually navigates to a dashboard page. That's Chapter 1's E2E layer, deliberately out of this course's scope from the very first chapter. A real production app would layer Playwright/Cypress tests on top of exactly this kind of component-level suite, not instead of it.

Coding Challenges

CHALLENGE 1

Add a sixth test to this suite: confirm that the "Log In" button is disabled while the login request is pending, to prevent a double-submit, using the same MSW setup as the success-path test.

View solution

CHALLENGE 2

This suite's empty-submission test doesn't use MSW at all. Explain why that's correct — what makes that specific behavior different from the login-success and login-failure tests that do need it.

View solution

CHALLENGE 3

Write a short reflection: pick one habit from this course (e.g. query priority, queryBy for absence, resetting MSW handlers, coverage skepticism) that you think you'd have gotten wrong without this course, and explain what would have gone wrong in practice.

View solution

CHAPTER 9 QUICK REFERENCE

- A complete feature test suite combines: rendering + query priority (Ch.3), userEvent + conditional rendering (Ch.4), async findBy (Ch.5), MSW + error simulation (Ch.6), and jest-axe (Ch.8)
- Ch.2's describe/it/expect mechanics underpin every test regardless of what's being tested
- Ch.7's cross-framework convergence means this same suite structure transfers to Vue/Svelte with minimal changes
- A well-named test suite documents behavior — readable as a spec, not just a pass/fail gate
- This suite is intentionally not an E2E test — Chapter 1 scoped that layer out from day one

✅ Frontend Testing Complete — 9 / 9 chapters

From the testing pyramid through Jest/Vitest mechanics, the Testing Library philosophy, real component and hook testing, mocking, cross-framework testing, coverage honesty, and accessibility — to a complete, realistic feature test suite. Every habit this course built is meant to transfer directly to real code, in React or otherwise.