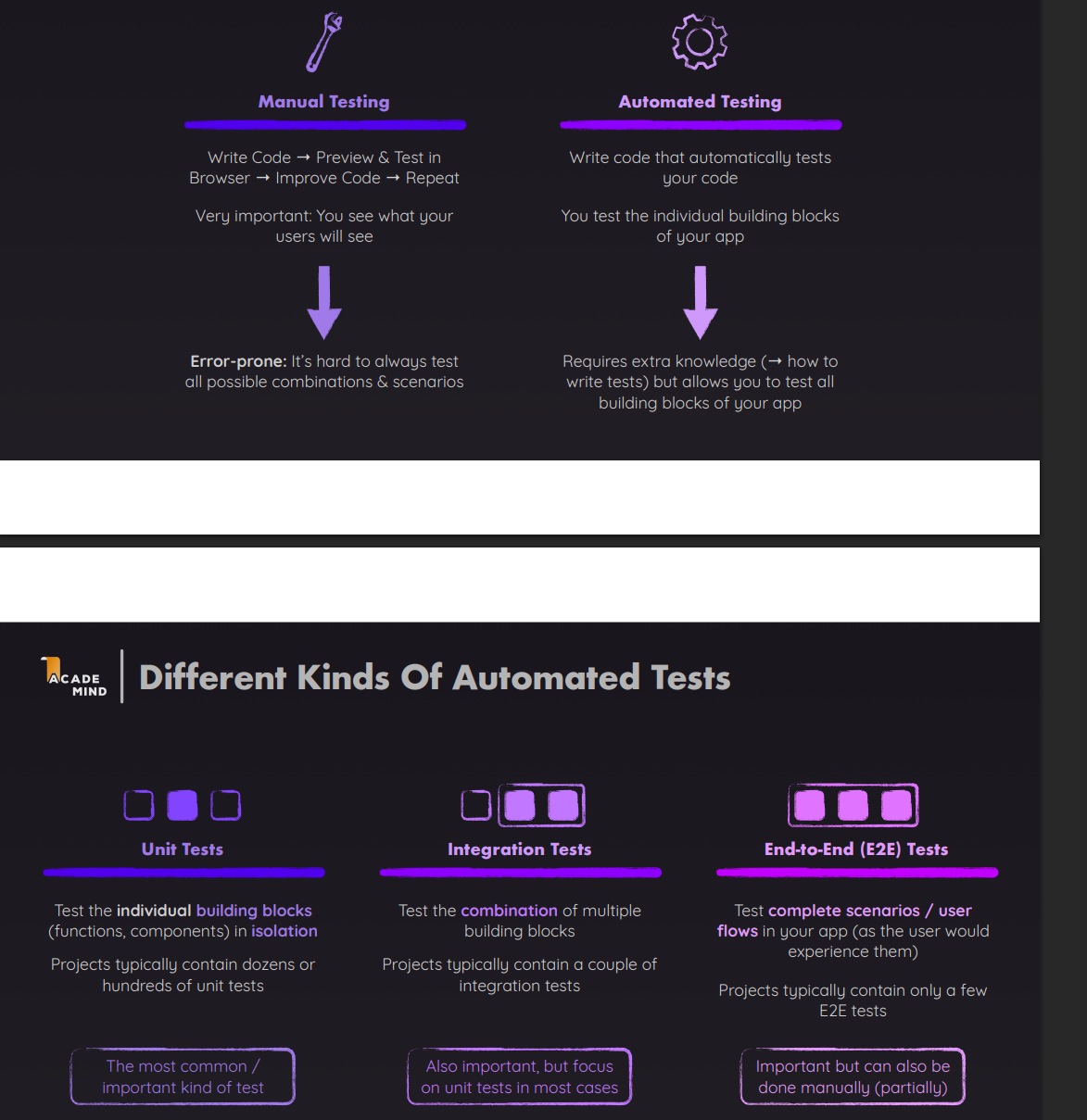
* 

Vitest: Test Runner

React Testing Library: Tool to interact with DOM

Jest: Test Framework /test runner

**📝 Cheatsheet to remember**

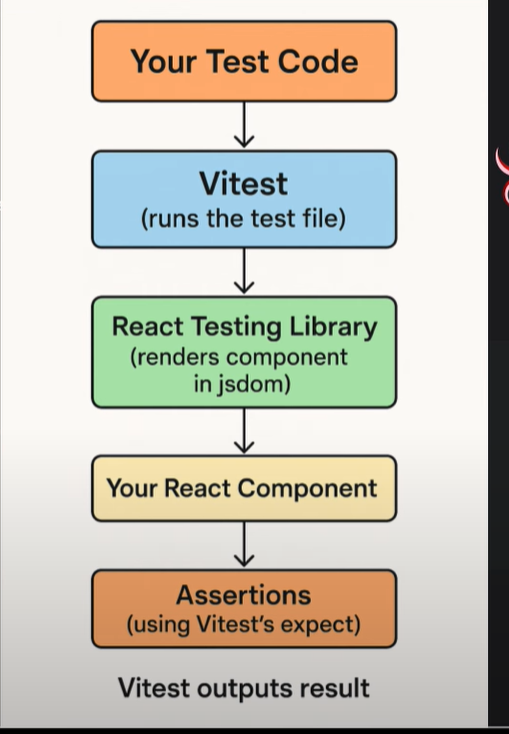
Here’s an easy mnemonic:

**Jest** = **test & expect**

Think: Jest is the “judge” checking results.

**RTL** = **render & screen**

Think: RTL is your “eyes” looking at the DOM.

* 
* RTL cheatsheet : <https://testing-library.com/docs/react-testing-library/cheatsheet/>
* Vitest and RTL flow: they both work together in react testing
* 

jsdom → then they give you a **fake browser-like environment**.

jsdom is a **JavaScript implementation of the browser DOM APIs** (document, window, elements, events).

It runs inside Node.js (so you can test without a real browser).

It simulates the DOM in **memory** (not on a screen).

* **screen in RTL (React Testing Library)**

**What it is:**

screen is just a **querying API** provided by RTL.

Example: screen.getByText("Login").

It queries the **DOM rendered in memory**.

The actual DOM is created by **jsdom** (a Node.js DOM implementation).

So when you do render(<MyComponent />), React renders into a **virtual DOM tree** provided by jsdom, and screen queries it.

* It.skip => skips the testcase and run others

It.only => runs only that testcase skips others

* Npm test->cra

Npm run test -> vite projects

* getBy\* ->throws error if not found -. Used when some button or something is mandatory to be present in dom

queryby-> throws null thrown if not found found -. Used when some button or something is optional to be present in dom

findby-> only for async ele -> returns promise so must be used with await and declare func as async

* prefer userevent than fireevent as userevent focuses more on user interactivity(clicks/types like user) ,fireevent is like machine doing it.
* In context provider,redux testing, enclose app/rendering component with context/redux provider and pass values/redux store to context provider,then it will not throw error

With RTL, to test Redux or Context-based state, you **wrap the component in the appropriate Provider** (<Provider store={store}> for Redux or <MyContext.Provider> for Context). Then simulate user actions with userEvent and assert UI updates.

* In routing,if some compoenent wrapped with router, if u want to test thsat ,wrap/enclose that component in test.js file too in render

To test routing in React, wrap components in **MemoryRouter(react-router-dom)**. Use initialEntries to simulate starting routes, and userEvent.click on links to simulate navigation. Assertions are then made on the expected route content.

* Act & waitfor() fro async

A screenshot of a questionnaire

AI-generated content may be incorrect.

A close-up of a white background

AI-generated content may be incorrect.

* Renderhook to test custom hooks

// useCounter.js

import { useState } from "react";

export function useCounter(initialValue = 0) {

const [count, setCount] = useState(initialValue);

const increment = () => setCount((c) => c + 1);

return { count, increment };

}

**Test with renderHook**

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

import { useCounter } from "./useCounter";

test("should increment counter", () => {

const { result } = renderHook(() => useCounter(0));

// initial value

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

// update state inside act

act(() => {

result.current.increment();

});

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

});

👉 result.current gives you access to the hook’s return value.  
👉 act() is needed when you trigger state updates.

**🔹 Testing Hooks with Context / Redux**

You can pass a **wrapper** to renderHook to provide context (just like render in RTL).

**Example with Context**

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

import { UserProvider, useUser } from "./UserContext";

test("updates user via context hook", () => {

const wrapper = ({ children }) => <UserProvider>{children}</UserProvider>;

const { result } = renderHook(() => useUser(), { wrapper });

expect(result.current.user).toBe(null);

act(() => {

result.current.setUser({ name: "Alisha" });

});

expect(result.current.user.name).toBe("Alisha");

});

**🔹 Difference: render vs renderHook**

render → for testing **components** (UI + DOM).

renderHook → for testing **custom hooks directly** (logic only).

**🔹 Interview One-Liner**

renderHook is used to test custom hooks directly. It gives access to the hook’s return value (result.current). If the hook depends on context or providers, you pass a wrapper to wrap the hook inside those providers.