Here's a **complete cheatsheet** for **Vitest** and **React Testing Library (RTL)** — perfect for React developers who want to test their components effectively. It covers:

- ✅ Installation
- ⚙️ How they work under the hood
- 🧪 Core testing functions (`describe` , `it` , `expect`)
- 🔍 `screen` & queries `waitFor` , `act` , and async testing

✅ 1. INSTALLATION

Install Vitest & React Testing Library (for React + Vite project):

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

Optional:

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

`vite.config.ts` **setup:**

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
 plugins: [react()],
 test: {
 environment: 'jsdom', // important for DOM APIs
 globals: true, // for global describe/it/expect etc.
 setupFiles: './setupTests.ts', // for jest-dom etc.
 },
})
```

`setupTests.ts` :

```
import '@testing-library/jest-dom'
```

2. HOW VITEST + RTL WORK TOGETHER UNDER THE

HOOD Vitest:

Runs tests in Node.js using `jsdom` to simulate the DOM.

Supports ESM, fast re-runs, type-safe (TS support), and snapshot testing.

It compiles your tests like Vite and runs in a Vite-like dev environment.

RTL:

Focuses on testing **components from the user's perspective**.

Queries DOM elements like a user would (e.g., by role, label, placeholder).

Encourages **real user interactions** instead of testing internal component logic.

?? 3. CORE TEST FUNCTIONS

`describe` :

Groups related tests.

```
describe('Button Component', () => {
 it('should render correctly', () => {})
})
```

`it` **or** `test` :

Defines a single test case.

```
it('should render a button', () => {
 // test logic
})
```

`expect` :

Assertions to validate output.

```
expect(screen.getByRole('button')).toBeInTheDocument()
```

```
render(<MyComponent />)
screen.getByRole('button')
```

4. SCREEN & QUERY TYPES

`screen` :

Access the DOM rendered by RTL without needing destructuring.

Query Types:

Query Type	Sync	Async	Throws on not found	Multiple Match Error
`getBy...`				✅
`queryBy...`				✅
`findBy...`			(timeout)	✅

Common Queries:

Query	Description
`getByRole`	Accessible role (e.g., button, heading)
`getByLabelText`	Associated label (for input forms)
`getByPlaceholderText`	Placeholder value
`getByText`	Visible text content

`getByTestId`	Uses `data-testid` attribute
`getByAltText`	For images
`getAllBy...`	Returns multiple elements
`queryBy...`	Returns null if not found (no error)

5. ASYNC TESTING UTILS

`waitFor` :

Waits for a condition to be true (e.g., after state update, fetch).

```
await waitFor(() => {
 expect(screen.getByText('Loaded')).toBeInTheDocument()
})
```

`findBy...` :

Shortcut for `await waitFor(() => getBy...)`

```
const element = await screen.findByText('Loaded')
```

`act` :

Wraps updates to components (like user events or timers). Normally **handled automatically**, but use it when needed:

```
await act(async () => {
 fireEvent.click(button)
})
```

6. COMMON UTILITIES

`render()` :

Renders a React component into the testing DOM.

```
render(<MyComponent />)
```

`fireEvent` (low-level) or `userEvent` (high-level):

```
fireEvent.click(button)
```

```
await userEvent.type(input, 'Hello')
      await userEvent.click(button)
```

✅ 7. EXAMPLE TEST CASES

✅ 1. Simple Button Render

```
it('renders the button', () => {
 render(<button>Click Me</button>)
 expect(screen.getByText('Click Me')).toBeInTheDocument()
})
```

✅ 2. Input with user typing

```
it('accepts input text', async () => {
 render(<input placeholder="Name" />)
 const input = screen.getByPlaceholderText('Name')
 await userEvent.type(input, 'John')
 expect(input).toHaveValue('John')
})
```

✅ 3. Button Click updates state

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

```
}

it('increments count on click', async () => {
 render(<Counter />)
 const button = screen.getByText('Click')
 await userEvent.click(button)
 expect(screen.getByText('Count: 1')).toBeInTheDocument()
})
```

✅ 4. Async fetch result

```
function FetchData() {
 const [data, setData] = useState('')
 useEffect(() => {
 fetch('/api').then(res => res.text()).then(setData)
 }, [])
 return <div>{data}</div>
}

global.fetch = vi.fn(() =>
 Promise.resolve({ text: () => Promise.resolve('Hello') }) ) as any

it('renders fetched data', async () => {
 render(<FetchData />)
 expect(await screen.findByText('Hello')).toBeInTheDocument()
})
```

🔧 8. MOCKING MODULES & FUNCTIONS Mocking fetch:

```
global.fetch = vi.fn(() =>
 Promise.resolve({ json: () => Promise.resolve({ name: 'John' }) })
) as any
```

Mocking module:

```
vi.mock('./api', () => ({
 getUser: vi.fn(() => Promise.resolve({ id: 1 })),
}))
```

9. CLEANUP

Automatic cleanup is handled by RTL, but you can also use:

```
import { cleanup } from '@testing-library/react'

afterEach(() => {
 cleanup()
})
```

10. DEBUGGING

Debug the DOM

```
screen.debug()
```

🧪 11. SNAPSHOT TESTING (optional)

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

it('matches snapshot', () => {
 const { asFragment } = render(<Component />)
 expect(asFragment()).toMatchSnapshot()
})
```

12. BEST PRACTICES

Prefer `getByRole` , `getByLabelText` over `getByTestId` .

Simulate real user interactions using `userEvent` .

Don't test implementation details (avoid testing `useState` , `useEffect`).

Always assert what's visible or accessible to users.

Here is a **complete table of all query types** provided by **React Testing Library**, along with their **variants** and **what they return**.

🧪 React Testing Library Query Methods

Base Query	Variant	Returns	Throws on Not Found	Throws on Multiple	Async
`getBy`	`getByText`	Element	✅ Yes	✅ Yes	❌ No

Base Query	Variant	Returns	Throws on Not Found	Throws on Multiple	Async
	`getByRole`	Element	✅ Yes	✅ Yes	❌ No
	`getByLabelText`	Element	✅ Yes	✅ Yes	❌ No
	`getByPlaceholderText`	Element	✅ Yes	✅ Yes	❌ No
	`getByAltText`	Element	✅ Yes	✅ Yes	❌ No
	`getByDisplayValue`	Elem	✅ Yes	✅ Yes	❌ No
	`getByTitle`	Element	✅ Yes	✅ Yes	❌ No
	`getByTestId`	Element (`data-testid`	✅ Yes	✅ Yes	❌ No
`getAllBy`	`getAllByText`	Array of elements	✅ Yes	❌ No	❌ No
	`getAllByRole`	Array of elements	✅ Yes	❌ No	❌ No
	`getAllByLabelTe`	Array of elements	✅ Yes	❌ No	❌ No
	...	Same variants as `getBy*`	✅ Yes	❌ No	❌ No

`queryBy`	`queryByText`	Element or `null`	❌ No	✅ Yes	❌ No
	`queryByRole`	Element or `null`	❌ No	✅ Yes	❌ No
	...	Same variants as `getBy*`	❌ No	✅ Yes	❌ No
`queryAllBy`	`queryAllByText`	Array (possibly empty)	❌ No	❌ No	❌ No

Base Query	**Variant**	**Returns**	**Throws on Not Found**	**Throws on Multiple**	**Async**
	`queryAllByRole`	Array (possibly empty)	❌ No	❌ No	❌ No
`findBy`	`findByText`	`Promise<Element>`	✅ (after timeout)	✅ Yes	✅ Yes
	`findByRole`	`Promise<Element>`	✅ (after timeout)	✅ Yes	✅ Yes
	...	Same variants as `getB`	✅ Yes	✅ Yes	✅ Yes
`findAllBy`	`findAllByText`	`Promise<Array<Element>>`	✅ Yes	❌ No	✅ Yes
	`findAllByRole`	`Promise<Array<Element>>`	✅ Yes	❌ No	✅ Yes

Summary of Base Queries:

Prefix	**Use When**
`getBy*`	

	You expect **exactly one match**. Fails on 0 or multiple matches.
`getAllBy*`	You expect **multiple matches**. Fails on 0 match.
`queryBy*`	You expect **0 or 1 match**. Returns `null` if none.
`queryAllBy*`	You expect **0 or more matches**. Returns an empty array if none.
`findBy*`	You expect **1 match after an async update** (e.g., after fetch).
`findAllBy*`	You expect **multiple matches after an async update**.