**React Testing Library - Beginner to Advanced Guide**

**Table of Contents**

1. Introduction
2. Setup with Vite
3. Basic Component Testing
4. Todo App Test Case
5. Event Handling
6. State and Props Testing
7. API Testing
8. Async Operations and waitFor
9. Using act() for State Updates
10. Testing Conditional Rendering
11. Best Practices

**1. Introduction**

React Testing Library (RTL) focuses on testing the UI from the user’s perspective. It avoids testing implementation details, promoting more reliable tests.

**2. Setup with Vite**

Install necessary packages:

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

Update vite.config.js:

import { defineConfig } from 'vite';

import react from '@vitejs/plugin-react';

export default defineConfig({

plugins: [react()],

test: {

environment: 'jsdom',

globals: true,

setupFiles: './src/setupTests.js'

},

});

Create src/setupTests.js:

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

**3. Basic Component Testing**

**Greeting.jsx**

export default function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

**Greeting.test.jsx**

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

import Greeting from './Greeting';

test('renders greeting message', () => {

render(<Greeting name="Deepak" />);

expect(screen.getByText('Hello, Deepak!')).toBeInTheDocument();

});

**4. Todo App Test Case**

**Todo.jsx**

import { useState } from 'react';

export default function Todo() {

const [todos, setTodos] = useState([]);

const [input, setInput] = useState('');

const addTodo = () => {

if (input.trim()) {

setTodos([...todos, input]);

setInput('');

}

};

return (

<div>

<input

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="Add todo"

/>

<button onClick={addTodo}>Add</button>

<ul>

{todos.map((todo, i) => <li key={i}>{todo}</li>)}

</ul>

</div>

);

}

**Todo.test.jsx**

import { render, screen, fireEvent } from '@testing-library/react';

import Todo from './Todo';

test('adds a todo item', () => {

render(<Todo />);

const input = screen.getByPlaceholderText('Add todo');

const button = screen.getByText('Add');

fireEvent.change(input, { target: { value: 'Buy milk' } });

fireEvent.click(button);

expect(screen.getByText('Buy milk')).toBeInTheDocument();

});

**5. Event Handling**

Use fireEvent or userEvent:

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

import userEvent from '@testing-library/user-event';

await userEvent.type(input, 'New Task');

await userEvent.click(button);

**6. State and Props Testing**

**Counter.jsx**

export default function Counter({ initial = 0 }) {

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

return (

<>

<p>Count: {count}</p>

<button onClick={() => setCount(c => c + 1)}>Increment</button>

</>

);

}

**Counter.test.jsx**

render(<Counter initial={5} />);

expect(screen.getByText('Count: 5')).toBeInTheDocument();

fireEvent.click(screen.getByText('Increment'));

expect(screen.getByText('Count: 6')).toBeInTheDocument();

**7. API Testing**

**User.jsx**

import { useEffect, useState } from 'react';

export default function User() {

const [user, setUser] = useState(null);

useEffect(() => {

fetch('/api/user')

.then(res => res.json())

.then(setUser);

}, []);

if (!user) return <p>Loading...</p>;

return <h1>{user.name}</h1>;

}

**User.test.jsx**

global.fetch = vi.fn(() =>

Promise.resolve({ json: () => Promise.resolve({ name: 'Deepak' }) })

);

test('renders user data', async () => {

render(<User />);

expect(screen.getByText('Loading...')).toBeInTheDocument();

const name = await screen.findByText('Deepak');

expect(name).toBeInTheDocument();

});

**8. Async Operations and waitFor**

await waitFor(() => {

expect(screen.getByText('Done!')).toBeInTheDocument();

});

**9. Using act() for State Updates**

import { act } from 'react-dom/test-utils';

await act(async () => {

await promiseFunction(); // e.g., setTimeout, async update

});

**10. Testing Conditional Rendering**

**Message.jsx**

function Message({ isLoggedIn }) {

return <p>{isLoggedIn ? 'Welcome' : 'Please log in'}</p>;

}

**Message.test.jsx**

render(<Message isLoggedIn={true} />);

expect(screen.getByText('Welcome')).toBeInTheDocument();

**11. Best Practices**

* Prefer user queries (getByRole, getByText, findByText)
* Use userEvent for real interaction simulation
* Mock API calls to avoid external dependencies
* Avoid testing internal state, focus on visible behavior
* Organize tests near components or in \_\_tests\_\_ folders

✅ This guide can be extended with coverage reports, mocking modules, context API testing, and error boundaries. Let me know if you want those additions!