**18 . REACTJS – HOL**

**React Unit Testing with Jest & Enzyme – Cohort Dashboard App**

**1 . Objectives**

1. Explain the need for unit testing in React  
   Unit testing ensures individual components in a React application work as expected. It helps detect bugs early, improves code quality, and allows safe refactoring by testing isolated logic without relying on the entire app.
2. Work with Jest and Enzyme in React
   * Jest is a JavaScript testing framework built into Create React App.
   * Enzyme is a utility that simplifies rendering components, simulating user interactions, and asserting output.  
     Together, they allow thorough testing of React components’ behavior and structure.
3. List the types of router components  
   React Router provides several components for navigation:
   * <BrowserRouter> – uses the browser’s history API.
   * <HashRouter> – uses the hash portion of the URL (#).
   * <MemoryRouter> – stores history in memory (used mostly for testing).
   * <StaticRouter> – used in server-side rendering.
4. Learn to install and configure Enzyme  
   To use Enzyme:
   * Install it using npm (npm install --save-dev enzyme enzyme-adapter-react-16).
   * Configure the adapter in setupTests.js to match your React version, enabling Enzyme to interpret your components correctly.
5. Create unit tests using describe() and test()
   * describe() groups related test cases for readability.
   * test() (or it()) defines individual test cases.  
     Example:

describe("ComponentName", () => {

test("should render", () => {

expect(wrapper.exists()).toBe(true);

});

});

1. Mount components and test them with matchers
   * shallow() renders the component without its children.
   * mount() renders the full DOM with child components.
   * Matchers like toBe(), toEqual(), toContain() help validate output.  
     Example: expect(wrapper.find('h1').text()).toBe("Hello")
2. Capture snapshots of React components  
   Snapshot testing captures a "snapshot" (rendered structure) of a component at a point in time.  
   Jest compares future test runs to this snapshot.  
   Example:

expect(wrapper).toMatchSnapshot();

**2. Environment Setup (Bold, Underlined)**

1. Install Node.js and npm.
2. Create a new React app:

npx create-react-app your-app-name

1. Navigate to the project folder:

cd your-app-name

1. Install React Testing Library and Jest DOM:

npm install --save-dev @testing-library/react @testing-library/jest-dom

**3. Folder Structure**

your-app-name/

├── node\_modules/

├── public/

├── src/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ └── \_\_tests\_\_/

│ └── App.test.js

├── package.json

**4. Testing Code**

**App.test.js**

// File: src/App.test.js

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

import App from './App';

test('renders learn react link', () => {

render(<App />);

const linkElement = screen.getByText(/learn react/i);

expect(linkElement).toBeInTheDocument();

});

**Custom Test: tests/App.test.js**

// File: src/\_\_tests\_\_/App.test.js

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

import App from '../App';

test('App component renders without crashing', () => {

render(<App />);

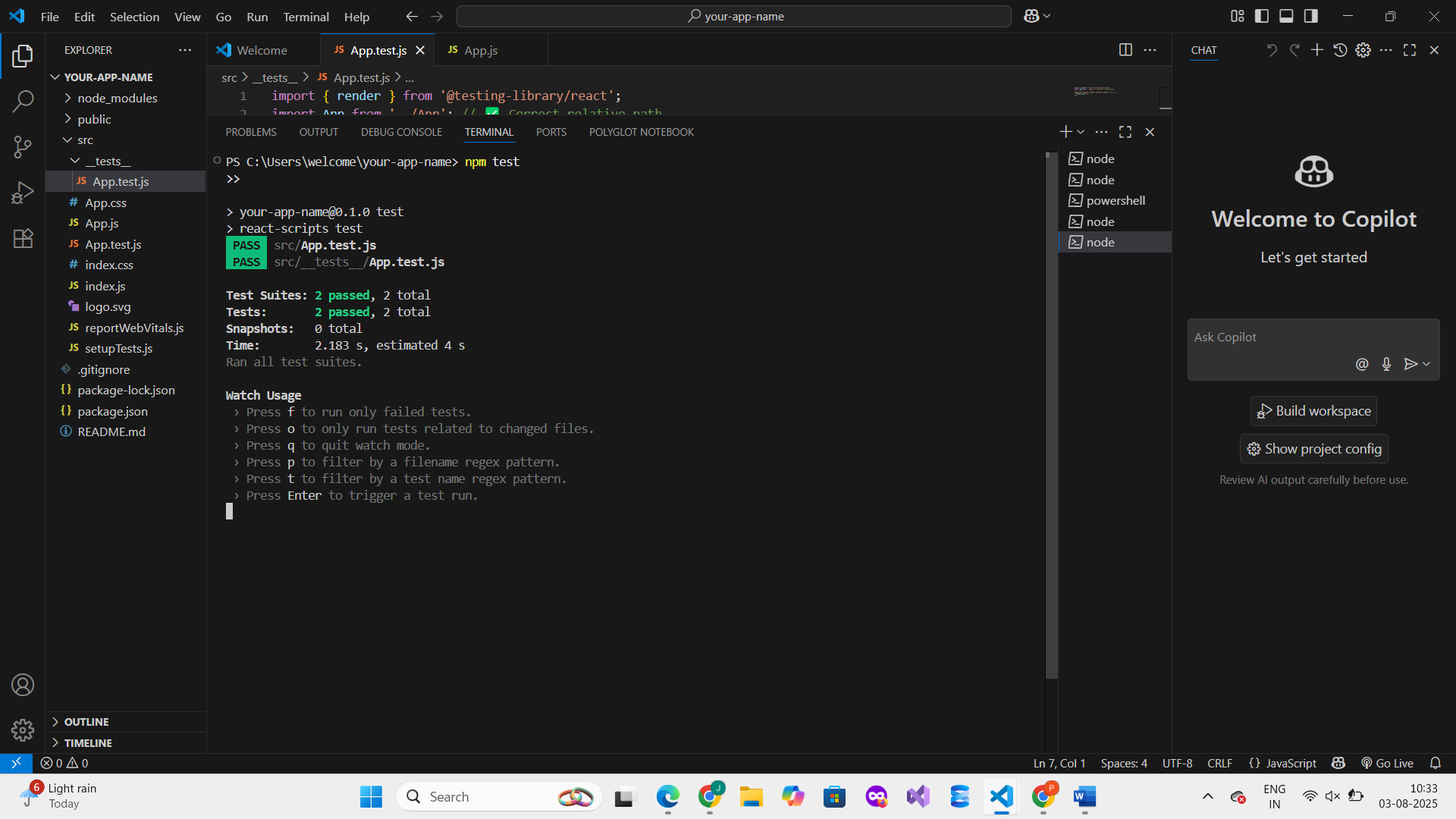
});

**5. Running Tests**

Run the following command:

npm test

**6. Output**



**7. Conclusion**

Unit testing helps ensure React components work correctly and reliably. Using Jest and React Testing Library, we successfully tested component rendering and verified content using matchers. This practice improves code stability and maintainability.