

**Lesson 7 – React Application Testing**

**Activity 12**

**Why Testing?**

* Testing is an essential aspect of software development that focuses on ensuring software products meet the required quality standards. It involves running a program or system with the purpose of discovering errors or confirming that it meets specified requirements. By conducting thorough testing, developers can validate the functionality and reliability of their software, thereby enhancing its overall quality.
* Testing plays a crucial role in MERN applications as it is vital for ensuring the reliability, functionality, and stability of your software. By conducting thorough testing, you can identify bugs early in the development process, maintain code quality, and build confidence in your application's behavior.

**Types of Testing in MERN stack**

In the MERN stack, there are several types of testing that can be performed to ensure the quality and reliability of the application. Here are some common types of testing in the MERN stack:

* **Unit Testing:** Unit testing involves testing individual components or functions in isolation to ensure they work as expected. Tools like Jest or Mocha are commonly used for unit testing in the MERN stack.
* **Integration Testing:** Integration testing focuses on testing how different components or services in the application work together. It ensures that the interactions between different parts of the application are functioning correctly. Tools like Jest or Mocha, along with Chai for assertions, can be used for integration testing in the MERN stack.
* **End-to-End Testing:** End-to-end testing involves testing the entire application flow, from the user interface to the backend, to ensure that all components work together seamlessly. Tools like Cypress or Selenium can be used for end-to-end testing in the MERN stack.
* **Functional Testing:** Functional testing verifies that the application functions correctly and meets the specified requirements. It focuses on testing the application's features and user interactions. Tools like Jest or Mocha can be used for functional testing in the MERN stack.
* **Performance Testing:** Performance testing is conducted to evaluate the application's performance under different conditions, such as high user loads or heavy data processing. It helps identify bottlenecks and optimize the application's performance. Tools like Apache JMeter or Artillery can be used for performance testing in the MERN stack.
* **Security Testing:** Security testing is essential to identify vulnerabilities and ensure the application is secure against potential threats. It involves testing for common security issues like cross-site scripting (XSS) or SQL injection. Various security testing tools and practices can be employed in the MERN stack

**Jest and React Testing Library Overview**

* Jest and React Testing Library are widely used tools for testing React applications. Together, they provide a comprehensive testing framework for verifying React components and application logic. **React Testing Library** is a lightweight library built on top of Jest for testing React components. Its primary goal is to test the behavior of components in a way similar to how users interact with them.
* This code is a unit test written using **Jest** and **React Testing Library** for a Counter component. Here's a breakdown of what each part of the test does:

**Common React Testing Library Functions:**

1. **render**:
   * Renders a React component for testing.
2. **screen**:
   * Provides access to the rendered DOM for querying elements.
3. **Query Methods**:
   * getByText: Finds elements by their text content.
   * getByTestId: Finds elements with the data-testid attribute.
   * getByRole: Finds elements by their ARIA roles. ARIA(Accessible Rich Internet Applications)
   * findBy... and queryBy... provide asynchronous or non-throwing alternatives.
4. **User Events**:
   * React Testing Library integrates with the @testing-library/user-event library to simulate user interactions like typing, clicking, etc.

**Activity**

1. Create a Counter component in your React app.

import React, { useState } from 'react'

function Counter({ initialCount}) {

    const [count, setCounter] = useState(initialCount)

    const increment = () => {

        setCounter((prev) => prev + 1)

    }

    const decrement = () => {

        setCounter((prev) => prev - 1);

    }

    const restart = () => {

        setCounter(0)

     }

    const switchSigns = () => {

        setCounter((prev) => prev \* -1)

    }

    return (

      <div>

        <h3 data-testid="count">{count}</h3>

        <button data-testid="increment" onClick={increment}>

         Increment

        </button>

        <button data-testid="decrement" onClick={decrement}>

          Decrement

        </button>

        <button data-testid="restart" onClick={restart}>

          Restart

        </button>

        <button data-testid="switchsign" onClick={switchSigns}>

          Switch Sign

        </button>

      </div>

    );

}

export default Counter

**Conducting Unit Test for Counter Component**

To unit test the Counter component, you can use Jest along with React Testing Library. Below are the steps to test the component:

**Step 1: Install Dependencies**

First, make sure you have Jest and React Testing Library installed. You can install them if you haven't already:

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

**Step 2: Create a Test File and write the test case**

Create a test file for your component, typically in the src folder, with the name Counter.test.js. Write a test case for the component.

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

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

import "@testing-library/jest-dom/extend-expect";

import Counter from "./Counter";

describe(Counter, () => {

    it("counter displays correct initial count",() => {

        render(<Counter initialCount={5} />)

        const h3element = screen.getByTestId("count")

        expect(h3element).toHaveTextContent(5)

    })

})

Key Concepts and Explanation for the test case

1. describe(Counter, () => { ... }):
   * The describe block is used to group related tests. Here, it indicates that all the tests inside this block pertain to the Counter component.
2. it("counter displays correct initial count", () => { ... }):
   * The it block defines a single test case.
   * The string "counter displays correct initial count" is the description of what this test is verifying.
3. render(<Counter initialCount={5} />):
   * This renders the Counter component with the prop initialCount set to 5 for testing purposes.
   * React Testing Library’s render function creates a virtual DOM for the component under test, enabling interaction and inspection.
4. screen.getByTestId("count"):
   * Retrieves a DOM element with the data-testid attribute value of "count".
   * This is commonly used in tests for selecting elements explicitly for verification purposes.
5. expect(h3element).toHaveTextContent(5):
   * This assertion verifies that the h3element (retrieved from the DOM) contains the text content 5.
   * If the component correctly initializes and displays the count based on the initialCount prop, the test will pass.

**Step 3: Create other test cases**

it("increments the count when the increment button is clicked", () => {

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

const h3element = screen.getByTestId("count");

const btnelement = screen.getByTestId("increment");

fireEvent.click(btnelement);

expect(h3element).toHaveTextContent(6);

});

it("decrements the count when the decrement button is clicked", () => {

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

const h3element = screen.getByTestId("count");

const btnelement = screen.getByTestId("decrement");

fireEvent.click(btnelement);

expect(h3element).toHaveTextContent(4);

});

it("resets the count to 0 when the restart button is clicked", () => {

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

const h3element = screen.getByTestId("count");

const btnelement = screen.getByTestId("restart");

fireEvent.click(btnelement);

expect(h3element).toHaveTextContent(0);

});

it("switches the sign of the count when the switch sign button is clicked", () => {

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

const h3element = screen.getByTestId("count");

const btnelement = screen.getByTestId("switchsign");

fireEvent.click(btnelement);

expect(h3element).toHaveTextContent(-5);

});