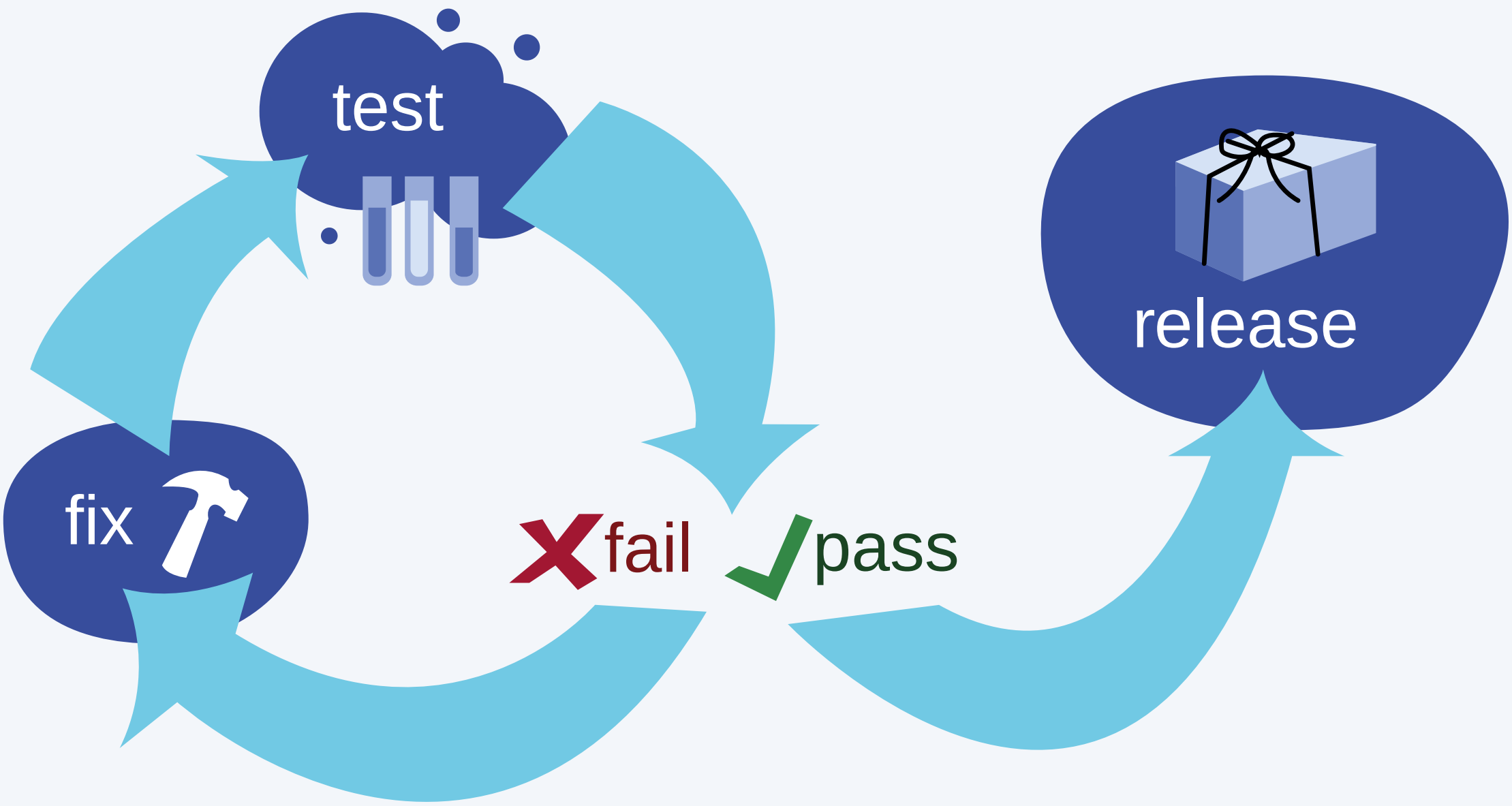
Day 28

Testing ReactJS Applications



@oluwakemi Oluwadahunsi

Introduction

As your codebase expands, small errors and edge cases you don’t expect can cascade into larger failures. **Bugs** lead to bad user experience. One way to prevent fragile programming is to test your code before releasing it into the wild.

Why Testing Is Important

We're humans, and humans make mistakes. Testing is important because it helps you uncover these mistakes and verifies that your code is working.

Perhaps even more importantly, testing ensures that your code continues to work in the future as you add new features, refactor the existing ones, or upgrade major dependencies of your project.

There is more value in testing than you might realize. One of the best ways to fix a bug in your code is to write a failing test that exposes it.

Then when you fix the bug and re-run the test, if it passes it means the bug is fixed, never reintroduced into the code base.

@oluwakemi Oluwadahunsi

Testing ReactJS Applications

Testing in React involves verifying that your components, hooks, state management, and other functionalities behave as expected.

This ensures your application runs correctly under different conditions and that new changes don’t break existing functionality.

When making changes or adding new features, tests ensure that existing functionality isn’t unintentionally broken.

Tests can also serve as documentation for new people joining your team. For people who have never seen a codebase before, reading tests can help them understand how the existing code works.

@oluwakemi Oluwadahunsi

Types of Tests

- **Unit Testing:** Focuses on testing individual components or functions to ensure that they work in isolation. For example, testing whether a button component renders correctly.
- **Component Tests:** Similar to unit tests but focused on testing individual React components. This ensures that components render correctly and interact properly with other components.
- **Integration Tests:** Tests the interaction between different components and parts of your application. For example, testing whether a form component correctly interacts with a state management solution like Redux.
- **End-to-End (E2E) Testing:** Tests the entire application by simulating user interactions in a real browser. For example, testing whether a user can log in, navigate to a specific page, or fill out a form.

@oluwakemi Oluwadahunsi

- **Snapshot Tests:** Captures the rendered output of a component and compares it to a stored “snapshot.” If the component's output changes, the test fails.

Popular Testing Libraries in React

1. Jest:

The Jest testing library is a popular JavaScript testing framework.

It is widely used for testing JavaScript applications, particularly those built with React, although it can be used to test any JavaScript project.

Jest provides an **all-in-one** solution for **unit testing**, i**ntegration testing**, and even **snapshot testing**, allowing developers to write tests that verify the correctness of their code in a structured and easy-to-understand manner.

For example:

```
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();
});
```

In this example, a simple Jest test is written using the **@testing-library/react** to test the **App** component in a React application. The goal of this test is to check if a specific text, "**learn react,**" is rendered correctly in the App component.

Let’s further break this test down:

- **render:** This function comes from **@testing-library/react** and is used to render the React component in a simulated DOM environment during the test. This lets you test the component as it would appear in the browser.

@oluwakemi Oluwadahunsi

- **screen:** This is an object provided by @testing-library/react that contains **methods** for querying elements on the rendered page (in the virtual DOM). You can use it to search for elements based on their text, roles, labels, etc.
- **test('renders learn react link', () => { ... }):** This defines a test. The **first argument** is a string description of what the test is checking (in this case, "**renders learn react link**"), and the **second argument** is the **test function** where the actual test logic resides.
- **render(<App />):** This renders the **App** component in a virtual DOM.
- **screen.getByText(/learn react/i):** This line searches the rendered DOM for an element that contains text matching the regular expression **/learn react/i.** The **/i** flag makes the search case-insensitive, meaning it will match "Learn React," "learn react," or any other variation in capitalization.

@oluwakemi Oluwadahunsi

- **expect(linkElement).toBeInTheDocument():** This is an **assertion**. It checks if the linkElement (the result of screen.getByText()) is actually present in the DOM. If it is, the test passes. If the text "learn react" is not found in the DOM, the test will fail.

2. React Testing Library (RTL):

React Testing Library (RTL) is a popular testing utility that is built on top of **DOM Testing Library.** It is designed specifically for testing React components in a way that closely mimics how **users interact** with your application.

RTL focuses on **testing components' behavior** and **interactions** rather than their internal implementation details. This makes tests more reliable, less prone to breakage, and easier to maintain.

It provides simple utilities to interact with DOM elements, making it easy to write readable tests.

@oluwakemi Oluwadahunsi

Key Benefits of React Testing Library:

- **User-Centric Testing:** RTL encourages testing the app in the same way a user would interact with it, such as by querying for elements by their text, labels, roles, or other accessible attributes, rather than focusing on implementation details like class names or component state.
- **Less Fragile Tests:** Since RTL tests are more aligned with user behavior and interaction, they are less likely to break with minor changes to the implementation (like refactoring, renaming internal variables, etc.).
- **Minimal API:** RTL has a simple and minimal API, focusing on providing helpers to find elements on the page and interacting with them.

@oluwakemi Oluwadahunsi

- **Improved Accessibility:** RTL encourages you to use accessibility-first selectors (like g**etByRole** and **getByLabelText**), which can lead to better accessibility practices in your app.

- **Works Well with Jest:** RTL works seamlessly with **Jest**, the default testing framework in React projects, providing built-in assertions and testing utilities.

Example:

```
1 import { render, fireEvent } from '@testing-library/react';
2 import Button from './Button';
3
4 test('calls onClick prop when clicked', () ⇒ {
5   const handleClick = jest.fn();
6   const { getByText } = render(<Button onClick={handleClick}>Click Me</Button>);
7   fireEvent.click(getByText(/click me/i));
8   expect(handleClick).toHaveBeenCalledTimes(1);
9 });
```

This test checks if the **Button** component's **onClick** prop is called correctly when the button is clicked.

@oluwakemi Oluwadahunsi

- **jest.fn():** This creates a mock function, handleClick, which will be used to simulate the onClick prop. It allows you to track how many times the function was called.

- **render(<Button onClick={handleClick}>Click Me</Button>):** This renders the Button component with the onClick prop set to the handleClick mock function. The button displays the text "Click Me".

- **fireEvent.click(getByText(/click me/i)):** This simulates a click event on the button by finding it using its text ("Click Me"). The /click me/i is a case-insensitive regular expression that matches the button text.

- **expect(handleClick).toHaveBeenCalledTimes(1):** This assertion checks if the handleClick function (the onClick handler) was called exactly once after the button was clicked.

@oluwakemi Oluwadahunsi

3. Cypress:

Cypress is an **end-to-end** testing framework. It runs tests in a **real browser environment** and provides utilities to interact with the DOM, making it ideal for testing the entire flow of your application.

Example of a cypress test:

```
describe('My First Test', () => {
  it('Visits the Kitchen Sink', () => {
    cy.visit('https://example.cypress.io');
    cy.contains('type').click();
    cy.url().should('include', '/commands/actions');
  });
});
```

This Cypress test checks that when visiting 'https://example.cypress.io', the text "**type**" can be clicked, and that clicking it changes the **URL** to include '**/commands/actions**'. It simulates real user interactions with the website and ensures the expected navigation occurs.

@oluwakemi Oluwadahunsi

- The **describe block** defines a test suite. It groups related tests together under the label "**My First Test.**" This makes it easier to organize tests logically.
- The **it block** defines an individual test case. In this case, the test is named "**Visits the Kitchen Sink.**" It describes what this specific test is supposed to do.
- **cy.visit('https://example.cypress.io'):** This command tells Cypress to open the URL '**https://example.cypress.io**'. It's simulating a user visiting this page in a browser.
- **cy.contains('type').click():** Cypress looks for an element that contains the text '**type**' and clicks on it. This simulates a user clicking on a button or link with that text.
- **cy.url().should('include', '/commands/actions'):** This is an **assertion**. It checks that after the click action, the URL contains the string '**/commands/actions**', verifying that the user has been navigated to the correct page.

@oluwakemi Oluwadahunsi

4. Enzyme:

Enzyme, developed by Airbnb, allows you to manipulate, traverse, and interact with React components, making it useful for both shallow rendering and full DOM rendering tests.

Example of Enzyme test:

```
import { shallow } from 'enzyme';
import App from './App';

it('renders welcome message', () => {
  const wrapper = shallow(<App />);
  expect(wrapper.contains(<h1>Welcome to React</h1>)).toEqual(true);
});
```

This **Enzyme** test checks whether the **App** component renders an **<h1>** element with the text "**Welcome to React.**" If the text is present, the test passes; otherwise, it fails.

- **it:** Defines a test case, in this case, checking if the **App** component renders a welcome message.

@oluwakemi Oluwadahunsi

- **shallow(<App />):** Shallow-renders the App component and returns a wrapper object. This wrapper can be used to interact with and inspect the rendered component.
- **wrapper.contains(<h1>Welcome to React</h1>):** This checks if the shallow-rendered App component contains an <h1> element with the text "Welcome to React".
- **toEqual(true):** This asserts that the **<h1>** element is found and equals **true**, meaning the h1 tag with the message is correctly rendered in the **App component**.

How To Integrate testing in React using RTL & Jest

We will be testing our Mood Tracker Application as an example to explain this topic.

Here is a Step-by-Step Guide to Set Up and Use React Testing Library in a React project:

Step 1: Install Required Packages

First, let's install the necessary dependencies. Since we are using Vite, run the following command in your project directory:

“npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event jest-environment-jsdom jest @babel/preset-react babel-jest”

If you're using **Create React App**, **Jest** is already included, so you only need to install the testing library packages:

“npm install --save-dev @testing-library/react @testing-library/jest-dom @testing-library/user-event jest-environment-jsdom @babel/preset-react babel-jest”

Step 2: Configure Jest (if not using Create React App):

If you are using the CRA project, you don’t need this step as it has already been done, but since we used Vite, do this:

Create a `**jest.config.js**` file in your project root and add this:

```
export default {
  testEnvironment: 'jest-environment-jsdom',
  setupFilesAfterEnv: ['@testing-library/jest-dom'],
  moduleNameMapper: {
    '\\.(css|less|scss|sass)$': 'identity-obj-proxy',
  },
  transform: {
    '^.+\\.(js|jsx|ts|tsx)$': 'babel-jest',
  },
};
```

Also create a **.babelrc** file in the root of your project (if you don't have one yet) and add the following content:

```
{
  "presets": [
    "@babel/preset-env",
    "@babel/preset-react"
  ]
}
```

Step 3: Update package.json:

Add the following script to your `**package.json**` file, in the **scripts**: "test": "node --experimental-vm-modules node_modules/jest/bin/jest.js"

```
"scripts": {
  "test": "node --experimental-vm-modules node_modules/jest/bin/jest.js",
  "dev": "vite",
```

Step 4: Creating test files

Create a folder named "__tests__" in src folder.
Then, create a file in this folder named "**MoodSelector.test.jsx**"

@oluwakemi Oluwadahunsi

MoodSelector.test.jsx

```jsx
import React from "react";
import { render, screen, fireEvent } from "@testing-library/react";
import "@testing-library/jest-dom";
import { MoodContext } from "../contexts/MoodContext";
import MoodSelector from "../components/MoodSelector";
import { describe, it, expect, jest} from "@jest/globals";


const renderMoodSelectorWithContext = (contextValue) => {
  return render(
    <MoodContext.Provider value={contextValue}>
      <MoodSelector />
    </MoodContext.Provider>
  );
};

describe("MoodSelector Component", () => {
  const addMoodEntryMock = jest.fn();
  const updateCurrentMoodEntryMock = jest.fn();
  const getAllMoodsMock = jest.fn(() => [
    { label: "Happy", emoji: "😄", color: "#FFD700" },
    { label: "Sad", emoji: "😢", color: "#4169E1" },
  ]);
  const mockContextValue = {
    addMoodEntry: addMoodEntryMock,
    getAllMoods: getAllMoodsMock,
    currentMoodEntry: null,
    updateCurrentMoodEntry: updateCurrentMoodEntryMock,
  };

  it("renders mood buttons", () => {
    renderMoodSelectorWithContext(mockContextValue);

    // Check if mood buttons are rendered
    expect(screen.getByText("How are you feeling?")).toBeInTheDocument();
    expect(screen.getByText("Happy")).toBeInTheDocument();
    expect(screen.getByText("Sad")).toBeInTheDocument();
  });

  it("updates the current mood on button click", () => {
    renderMoodSelectorWithContext(mockContextValue);

    // Simulate selecting a mood
    const happyButton = screen.getByText("Happy");
    fireEvent.click(happyButton);

    // Ensure updateCurrentMoodEntry was called with the correct data
    expect(updateCurrentMoodEntryMock).toHaveBeenCalledWith({
      mood: { label: "Happy", emoji: "😄", color: "#FFD700" },
      date: expect.any(String),
      activities: [],
      note: "",
    });
  });
});
```

In the test component above, we have written 2 Test cases:

- **Renders Mood Buttons:** Ensures that the mood buttons are rendered correctly.

- **Mood Selection:** Simulates a user clicking on a mood and checks that the **updateCurrentMoodEntry** function from MoodContext is called with the correct mood data.

Now, run “**npm test**” in your terminal. You should have:

```
kemil@DESKTOP-QKVDT62 MINGW64 /c/Desktop/My-Projects/my-vite-app (main)
> my-vite-app@0.0.0 test
> node --experimental-vm-modules node_modules/jest/bin/jest.js

 PASS  src/__tests__/MoodSelector.test.jsx
  MoodSelector Component
    √ renders mood buttons (45 ms)
    √ updates the current mood on button click (17 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        3.215 s
Ran all test suites.
```

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app