Episode 13 - Time for the Test

It is going to cover how to test your app & write test cases for it

Q Why do we need test cases?
We write test cases to prevent new features to break existing code

Test driven development
We write tests even before we write code. It is good but it makes development very slow

Different type of Testing:

1) Manual Testing → Human testing done on each component

2) Automated Testing → Testing done through softwares like Selenium to automate the testing process

Testing done by Developer

3) E2E Testing → Flow way testing, covers entire user Journey (Q/A Team)
Headless Browser - Kind of like an actual browser without UI
Replacing the manual testing with code

Testing the entire flow when a user lands on a website till it leaves the website. It will test all the actions user is doing after landing the website until it leaves in one go.

Tools - Cypress, Puppeteer, Selenium

4) Unit Testing → Core job of developers. Testing components in Isolation or testing just a part of a large app. Testing 1 unit or component or a small part of app. Eg- Header

5) Integration Testing → Testing integration between components. Where many components are involved
Eg: Search Filter

Libraries / Tools to Write Test Cases

1) **React Testing Library -**
This is built on top of DOM Testing Library. React Testing Library adds more React features in DOM Testing Library.
DOM Testing Library is a base library for all others Testing Libraries like React Testing Library, Native Testing Library, Angular Testing Library etc.

Create React App already provides testing file (RTL) in the app.
RTL uses Jest. It is a JavaScript Testing Framework. Jest uses Babel

Setting up Testing in app:-

→ **Install React Testing Library**
```
npm i -D @testing-library/react
```

→ **Install Jest**
```
npm i -D jest
```

Jest used with Babel needs additional dependencies

→ **Using Babel, Install dependencies (Jest Website)**
```
npm install --save-dev babel-jest @babel/core @babel/preset-env
```

→ **Config Babel**
In babel.config.js,

```
// Code
  module.exports = {
      presets: [[`@babel/preset-env`, { target: { node: `current` } }]],
  };
```

Parcel's Babel Configuration clashing with new Babel configuration. How to fix it?
Parcel is using Babel internally and has its own configuration but to use jest we have created a new Babel configuration. This will create a conflict that which babel configuration should be used.

Parcel gives an option to disable its in-built Babel Configuration.

→ How to disable Babel Configuration of Parcel:

1) Create .parcelrc file

2) Write the following code in that file

```
{
  "extends": "@parcel/config-default",
  "transformers": {
      "*.{js,mjs,jsx,cjs,ts,tsx}": [
        "@parcel/transformer-js",
        "@parcel/transformer-react-refresh-wrap"
      ]
    }
}
```

Or

Copy the code from parceljs.org → JavaScript → Babel → Usage with other tools

Now it will use Babel Config of babel.config.js file

→ Run Test Cases

```
npm run test
```

→ Test Configuration

```
npx jest --init
```

No
jsdom (browser-like)
Yes
Babel
Yes

Now jest.config.js will be formed

When we run test cases there is no browser running they need a runtime so we use jsdom. It is a library that provides the feature of a browser and helps to run test cases.

Js DOM Installation

If Test's version is 28 or higher then js-environment-jsdom must be installed separately

// Code

```
npm install --save-dev jest-environment-jsdom
```

✯ Basic Testing

→ Install VScode-icons extension

→ Create a folder in src
__tests__ [__ __ (2 underscores front & back)] → called Dunder

Or a file
1) filename.test.js (or)
2) filename.test.ts (or)
3) filename.spec.js (or)
4) filename.spect.ts

Eg:-
In sum.js,

```
//Code
export const sum = (a,b) => {
    return a+b;
}
```

In __tests__ → sum.test.js,

```
//Code
import {sum} from "../sum";

test("Sum function should calculate the sum of two numbers", () => {
    const result = sum(3,4);

    //Assertion
    expect(result).toBe(7);
});
```

Run command in terminal to test:
npm run test

✯ React Testing (Unit Testing)

```
//Code
import {render, screen} from "@testing-library/react";
import Contact from "../Contact";
import "@testing-library/jest-dom";

test("Should load contact us component", () => {
    render(<Contact />);

    //Querying
    const heading = screen.getByRole("heading");

    //Assertion
    expect(heading).toBeInTheDocument();
});
```

→ render() in test — to load a UI component on JS dom. To test a UI we have to load it on Jsdom. This method comes from react testing library

→ screen in test - to get the information about the Rendered UI. It give access to different methods to find specific elements, or information.

→ expect() in test - to check if the passed value is present or not. It give access of different methods to check where that value is present. But we need an external library. Eg. InTheDocument using toBeInTheDocument() method.

Errors :-

1) Syntax Error: Jsx

We are passing Jsx but it is not enabled in our test cases

→ To enable js, we have to install @babel/present-react

// Code

```
npm i -D babel/preset-react
```

→ Include it in Babel.config.js as well

```
module.exports = {
    presets: [
        ["@babel/preset-env", {targets: {node: "current"}}],
        ["@babel/preset-react", {runtime: "automatic"}],
    ],
};
```

↓

Not adding this might throw errors as it will take some default value

Babel/present-react is helping our testing library to convert the Jsx code to HTML so that it can read properly

2) TypeError: toBeInTheDocument is not a function

→ Install @testing-library/jest-dom

```
npm install -D @testing-library/jest-dom
```

It will give alot of different methods along with expect method in test cases

☆ More React Examples

1) Button:

```
test("Should load button in contact component", () => {
    render(<Contact />);
    const button = screen.getByRole("button");
    expect(button).toBeInTheDocument();
});
```

2) Through text:

```
test("Should load text in contact component", () => {
    render(<Contact/>);
    const buttonName = screen.getByText("Submit");
    expect(buttonName).toBeInTheDocument();
    },
  );
```

3) Multiple Elements:

```
test("Should load 2 input boxes on the contact component", () => {
    render(<Contact/>);

    // Querying
    const inputBoxes = screen.getAllByRole("textBox");

    // Assertion
    expect(inputBoxes.length).toBe(2);
  },
);
```

Difference between getByRole & getAllByRole

getByRole() will expect only 1 element if multiple elements are passed then it will throw an error.

Whereas getAllByRole() will expect 1 or more elements. It will get the element in Jsx. It returns an array of objects. These objects are HTML input element. It is virtual dom react element.

Jsx → React Element → Object

☆ Grouping Test cases

Grouping similar working test cases into a single block which is "describe"

Eg

```
describe("Contact Us Page Test Cases", () => {
    test("Should load Contact Us", () => {
      render(<Contact/>);
      const buttonName = screen.getByText("Submit");
      expect(buttonName).toBeInTheDocument();
      },
    );

    test("Should load button in contact component", () => {
      render(<Contact/>);
      const button = screen.getByRole("button");
      expect(button).toBeInTheDocument();
    });
  }
);
```

→ Instead of test() we can use it(). "it" is another name for "test"

→ Add coverage folder to git-ignore

✲ Unit Testing

//Code

```
import {Provider} from 'react-redux';
import {render, screen} from "@testing-library/react";
import appStore from "../../utils/appstore";
import Header from "../Header";
import {BrowserRouter} from "react-router-dom";
import "@testing-library/jest-dom";

it("Should render Header Component with a login button", () => {
    render(
        <BrowserRouter>
          <Provider store={appStore}>
            <Header />
          </Provider>
        </BrowserRouter>
      );

    const loginButton = screen.getByRole("button", {name: "login"});

    expect(loginButton).toBeInTheDocument();
})
```

→ In Header component, useSelector is used which is imported from react-redux. It is not JSX or JavaScript so it will throw error. To prevent it we have to provider the store to Header component in test case.

→ In Header component, <Link> tag is imported from react-router-dom. It is not a part of React so it will throw error. To prevent it we have to pass <Browser Router> to Header component

→ If there are multiple buttons in the component and we only want to select a specific button or element then we can pass additional info in screen method like name of the button (element)

→ You can pass regex in screen.getByText(/..../);

→ Test case of Login Toggle Button :-

```
it("Should change Login Button to Logout on Click", () => {
    render(
      <BrowserRouter>
        <Provider store={appStore}>
          <Header/>
        </Provider>
      </Browser>
    );

    const loginButton = screen.getByRole("button", {name: "Login"});

    fireEvent.click(loginButton);        → Firing an event in testing
    const logoutButton = Screen.getByRole("button", {name: "Logout"});
    expect(logoutButton).toBeInTheDocument();
});
```

→ Test case to render a component with props :-

```
import {render, screen} from "@testing-library/react";
import RestaurantCard from "../RestaurantCard";
import MOCK_DATA from "./mocks/resCardMock.json";
import "@testing-library/jest-dom";

it("Should render RestaurantCard component with props Data", () => {
    render(<RestaurantCard resData={MOCK_DATA} />);

    const name = screen.getByText("Leon's - Burgers & Wings (Leon Grill)");

    expect(name).toBeInTheDocument();
});
```

→ MOCK_DATA = a file where the actual data passed for a single Restaurant Card is present

★ Integration Testing

→ Test cases for Search Restaurants

```
import {render, screen} from "@testing-library/react";
import {act} from "react-dom/test-utils";
import Body from "../Body";
import MOCK_DATA from "../mocks/mockResListData.json";
import {BrowserRouter} from "react-router-dom";
import "@testing-library/jest-dom";

global.fetch = jest.fn(() => {
      return Promise.resolve({
            json: () => {
                return Promise.resolve(MOCK_DATA)
            }
        });
```

```
});

it("Should Search ResList for burger text input", async () => {
    await act(async () =>
        render(
            <BrowserRouter>
                <Body />
            </BrowserRouter>
        )
    );

    const cardsBeforeSearch = Screen.getAllByTestId("resCard");

    expect(cardBeforeSearch.length).toBe(20);

    const searchBtn = screen.getByRole("button", { name: "Search" });

    const searchInput = screen.getByTestId("Search Input");

    fireEvent.change(SearchInput, { target : { value : "burger" } });
                                  ↳ updating the value of event(e) coming from browser

    fireEvent.click(searchBtn);

    const cardsAfterSearch = screen.getAllByTestId("resCard");

    expect(cardsAfterSearch.length).toBe(4);
});
```

→ Faking Fetch (Mock Fetch Function)

Fetch is a browser operation and testing is done on jsdom which has limited features of browser. Testing is not done on actual browser so we can't make a network call during testing. Therefore, we have to make a fetch like function

→ global.fetch → accessing fetch variable in global object

→ jest.fn() → to create a new function that works similar to fetch (mock fetch function)

It takes a callback function where a promise is returned. On fulfilling the promise, it will resolve & return json() where json() again returns a promise & on fulfilling it returns the data. In our case, MOCK_DATA.

Eg: Actual Fetch

```
const res = await fetch("URL")
const data = await res.json()
```

→ MOCK_DATA – Data from the actual API copied & pasted in a file to use it in mock fetch function

→ Automatic running tests (using HMR)

Add In package.json,
"watch-tests : " jest --watch"

npm run watch-test

To automatically run test on saving file

→ act() – comes from "react-dom/test-utils". It is used whenever we try to test a fetch or any state updates. act() returns a promise so we have to use async await outside. It will take a callback function which is async

→ <BrowserRouter> – Link tag is imported from react-router-dom that's why we are wrapping our component in <BrowserRouter>

→ Test 1 – Checking the number of all the cards before updating

→ Updating – Firing changeEvent on searchInput and adding the value "burger" in searchInput.

Firing click Event on search button to search the cards with updating value in input

→ Test 2 - Checking the number of cards after firing events for search

→ Test Id
1) In JSX, data-testid = "Search Input"

2) data-testid = "resCard"
It will give testid to all the cards present. When we get elements by test id it will give an array of objects of all the React Elements with that id and we can get the length of that array & check.

→ Test cases to get Top Rated Restaurants
Similar to previous one

// Code
```
it ("Should filter Top Rated Restaurants", async () => {
    await act (async () =>
        render (
            <BrowserRouter>
                <Body />
            </BrowserRouter>
        )
    );
```

```
const cardBeforeFilter = screen.getAllByTestId("resCard");

expect(cardsBeforeFilter.length).toBe(20);

const topRatedBtn = screen.getByRole("button", { name: "Top Rated Restaurants" });

fireEvent.click(topRatedBtn);

const cardsAfterFilter = screen.getAllByTestId("resCard");

expect(cardsAfterFilter.length).toBe(13);
});
```

✱ Different Methods or Helper Functions :-

1) **beforeAll()** → It will run before running all the test cases. Used for clean up, log, test something before running any test cases

Eg

```
beforeAll(() => {
    console.log("BeforeAll");
});
```

2) **beforeEach()** → It will run before running each test case. Before every test case

Eg

```
beforeEach(() => {
    console.log("Before Each");
```

Used for cleaning up before every test case

3) **AfterAll()** → It will run after completing all the test cases

Eg

```
afterAll(() => {
    console.log("After All");
});
```

4) **AfterEach()** → It will run after every test case

Eg

```
afterEach(() => {
    console.log("After Each");
});
```

☆ Coverage File

Coverage → Icon-report → index.html

Right click on index.html & click open with Live Server

It will give a coverage report where it will tell you exactly where it has tested & where it has not.