03/11/24

Episode -13 Time for the test.

→ "A single line of code can break your whole application."

Qu What is testing?

Ans Suppose you have created a search functionality, So how you will test it, that it is working fine or not?
By searching something by your own!!
This is called manual testing, when a developer manually tests any functionality of the app.

Qu Why do we need testing?

Ans Suppose you have added / delted some code from your application, then that change affects different components which are connected to that code.

→ Now Suppose after adding some code into a component, it broke your any other component, then you will manually try to test, what went wrong, or even you will not get to know that, until you goto other component & see it is working fine or not.

→ To Solve this we have 2 options :-

1. Doing manual testing.
2. Writing test cases.

code that test our application automatically

Qu: Types of testing? (which a developers can do)

↳ 1) unit testing, 2) Integration testing 3) End to End (e2e) testing

★ Unit Testing :- Your ~~test~~ your react components in isolation.

For example :- You want to ~~test~~ your header component only & not any other component.

"Testing one unit or one component in isolation"

★ Integration Testing :- When you test how your different components are integrated / connected to each other.

How our components talk to each other.

For example :- When you search something on search button then how your Restlist component works? does it shows the Searched ~~component~~ Restaurant card, or not, How your search is integrated to your Restlist component.

★ End to End (e2e) Testing :- A Testing which starts from when a user lands on your page and do different stuffs like loging in, checking restaurants, adding it to card & etc to when the user leaves your website.

→ You will test all the flows from user landing onto your page to leaving your page.

→ For e2e testing we require different types of tools.

→ As a developer we are major concern about the 1st two types of testing i.e, Unit testing & Integration testing.

→ In Some companies you will need to do both, creating different ~~testings~~ Features & testing it.

→ We are not going to learn about e2e ~~tes~~ testing.

→ What libraries do we need to do testing :-

① → use it as dev-dependency. React testing Library ② → as a dev dependency. Jest.

→ React testing library is a part of testing library, This testing library is very old.

→ Previously we had a DOM testing library inside, testing library which was a testing library without any framework or library.

→ React testing library is build on the top of DOM testing library

→ Testing library have different testing libraries like angular testing library for testing angular project.

→ Angular testing library & all other testing libraries are build on the top of DOM testing library.

→ ☆ React testing library comes along with npx create-react-app.

→ Behind the scenes react testing library uses → Jest

→ Jest is a delightful ☆ JavaScript Testing Framework ☆ with a focus on simplicity. It is a standard to write testing using JS.

→ DOM testing library also uses Jest behind the scenes.

→ You can use Jest along with babel, react, angular etc.

→ As we are using babel so we also need to install some other dependencies and configure files, go to its website to see what all dependencies & config files are required for using Jest along with babel.

12/03/24

→ After adding all libraries you also need to configure your parcel

→ We are using parcel as a bundler into our application, internally parcel uses babel for transpilation., parcel has its own configuration for babel.

→ Now as we are using Jest along with babel if we have also added a babel configuration file to work with Jest, Now the default configuration of babel inside parcel will overrides our babel's configuration for Jest.

→ To Solve this you need to disable babel transpilation in parcel

→ To do this goto parcel document → Javascript → Babel → usage with other tools and copy past the given code their in a file .parcelrc & place it into your root folder of the project.

→ Now write Jest configuration

npx jest --init

(1) ↳ While configuring your jest use Jsdom as a testing environment When we write test cases, it doesnot run on browser, it needs a testing library to run. So we are using Jsdom as our testing environment.

→ Jsdom : It is not a browser, but it is like a browser, it have some important superpowers of browser to run test cases. We need to install this library.

→ To run test cases use command npm run test.

→ in your package.json ya under scripts, ## we have choosed "Jest" as test

```
scripts: {
    test : "jest";
}.
```

That's why we are using npm run test to start Jest to run ~~our~~ our test cases.

(2) choose babel as instrument code for coverage.

→ After completing the configuration, it will create a jest.config.js file, Similar to package.json.

★ Note :- Go to react-testing-library website, into setup is See if you are using jest version ≥ 28 you need to Seperately install `jest-environment-jsdom`

npm i -D jest-environment-jsdom == npm i --save-dev
Same as -D.

→ In older versions jest-environment-jsdom would come along with React-testing-library by default.

→ When you will do testing then you will a new folder named as coverage, add it to gitignore, it is just made to keep a coverage of all the testing files.

coverage → How many files have been covered.

17/03/24

★ Testing.

① Where should I write test cases :-

→ 1.) Inside __tests__ folder.

→ 2.) ↳ inside this folder write test cases with .js or .jt

Nameoffile.spec.js
Nameoffile.spec.ts
Nameoffile.test.js
Nameoffile.tests.ts
dot →

} Any where inside your project folder.

★ What ever files you have inside __tests__ Jest will track them

Jest will consider all the files inside __tests__ as a testing file

→ Inside your __tests__ you can name tests files as :-

sum.test.js | sumTest.js

→ But outside of your __tests__ folder you need to write

① sum.test.js / sum.tests.ts

② sum.spec.js / sum.spec.ts.

② What to write inside your test file.

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

test("It will calculate sum", () => {

    const result = sum(3,4);

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

```
export default const sum =
(a,b) => a+b
```

↑ Suppose we have this sum file and we want to test this,

So we have created a sum.test.js inside __tests__ folder,

→ Testing file Syntax

① write your test inside test method.

```
test(" ", () => { });
```

↓ description of the test

↓ A callback function, which will actually do the testing

② Inside your test method

```
const result = Sum(a,b)
```
1, 2 → calling Sum with 1, 2 & storing the sum into result

③ Assertion → checking

```
expect(result).toBe(7)
```

Now if my result will be 7 then, Sum will pass our test case, else it will fail.

④ Writing assertion into your testing file is not mandatory, if you don't write assertion then it will always pass your test case.

→ To run your test cases write command

```
npm run test
```

→ If you don't write anything inside test's call back function, then also it will pass.

→ How to write test cases for react??

* We will do || unit testing ||

→ Let's goto any page/component & see if it is getting loaded or not

* When ever you are testing any React UI component, you need to render it on |Jsdom.|

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

test ("should load contact page", () => {
  render (<Contact />); // Rendering contact page/component onto Jsdom.
  const Heading = screen.getByRole ("Heading");
  expect (Heading).toBeInTheDocument ();
});
```

Here we can render more than one component.

→ render <Contact />; When we write this command we are rendering contact component onto our Jsdom

→ But there is a problem, it will render our component as Jsx, but Jest doesn't understands Jsx, it understands html.

→ So for transpiling Jsx into html we need to install a package.

npm i -D @babel/preset-react.

→ And we also need to do some changes into our babel.config.file.

we need to add

["@babel/preset-react", {runtime: "automatic"}],

→ const Heading = Screen.getByRole ("heading");

↑ Think of Screen as something like a display onto which our component will be rendered & displayed.

→ getByRole ("heading") : We are basically trying to get a html tags/element whose role is heading.

For different html tags/elements, we have different Role, this roles are fixed by a testing library.

For h1, h2 ... h6 we have a role heading / heading ✗

For <input /> we have a role textbox.

→ let's say we have more than 1 heading into our component then use getAllByRole ("heading");

→ We can also use something like

expect (heading).not.toBe(3) → To use this methods we need to install another library.

→ expect (heading).toBeInTheDocument ();

It will basically check that heading is present into our DOM or not.

→ We have allot of different methods which we can use place of toBeInDocument for different types of testings

→ See other methods on their npm website

→ To use different methods like toBeInTheDocument and other related methods we need to install:

npm i -D @testing-library/jest-dom; ✱

→ If you write something like

const Heading = screen.getByRole("Heading");
console.log(Heading); → you will get the react element/ virtual dom
Jsx ~~element~~ / object / fibre node

→ you can also group your test cases using a describe block

```
describe("          ", () => {
    test("", () => { });
    test(" ", () => { });
});
```

✱ You can have multiple describe block and also nested describe blocks.

✱ Instead of writing test(" ", () => { });
you can also write this function as it(" ", () => { });
Both are same

✱ add something into script inside package.json

"watch-test": "jest --watch";

→ It will watch our test cases/ files, if we make any changes into our test file & safe it, it will automatically again run all the test cases.

→ So we don't need to write npm run test again & again after updating our test file.

23/03/24

→ describe function → inside describe function we can put more than on test function

```
describe("describe your test"), () => {

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

    });
    it("should dance"), () => {

    });
});
```

→ beforeAll function

```
beforeAll (() => {
    console.log("I will be called before printing/executing all test");
});
```

→ afterAll function

```
afterAll (() => {
    console.log("I will be called to after executing all test");
});
```

→ beforeEach function

```
beforeEach (() => {
    console.log("I will be called before executing each test case");
});
```

→ afterEach function

```
afterEach(() => {
  console.log("I will be called after each test case");
});
```

→ import "@testing-library/jest-dom"; → to use toBeInTheDocument(); and other related functions.

→ How to write a test case for a component which uses fetch().

↳ fetch() is given to us by browsers, not by Javascript, while testing we use Jsdom which is like a browser not actual browser, it doesn't have the power of fetch. which our browsers have.

→ we basically need to fake a fetch function, create a fake fetch function which works exactly same as fetch().

It is used to replace the fetch function by the front function globally

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

→ It will create a function. It takes a callback function ~~which we want~~

→ jest.fn(() => {}); jest is used to create a function. ~~The~~ It takes a callback function, This callback function is the actual function which we are creating.

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

fetch returns a promise i.e, we are also returning a promise

This keyword returns a resolved Promise as an object. It will return "data" as resolved promise.

```
const load = async () => {
  const data = await fetch("");
  const JSON = await data.json();
};
```

after getting the data we convert it into json which also returns a promise i.e, we are returning a promise with our final data.

Promise.resolve(5) it will return 5 as a resolved promise object.

```
global.fetch = jest.fn(() => {
  // const data = await fetch("");
  return Promise.resolve({
    // const json = await data.json();
    json: () => {
      return Promise.resolve(data);
    }
  });
});
```

→ How to write a test case if your component includes redux store and react-dom (LINK comp);

```
import {Provider} from "react-redux";
import {BrowserRouter} from "react-router-dom";
it("should render", () => {
  render(<Provider store={appStore}>
    <BrowserRouter>
      <component />
    </BrowserRouter>
  </Provider>);
});
```

→ <BrowserRouter> is a router component which we get from react-router-dom.

→ How to write a test case if your component have setState() func.

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

it("should render by component", async () => {
    await act( async () => {
        render (
            <Provider Store={appStore}>
            <BrowserRouter>
              <component />
            </BrowserRouter>
            </Provider>
        );
    });
});
```

→ How to add event listeners in test case

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

→ fireEvent is used to fire any event like click, change etc

→ render is used to render our component on jsdom

→ Screen: This method is used to check after rendering what will be rendered onto our screen.

```
it ("should renders", () => {

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

    ~~fireEvent (button).click~~

    fireEvent.click (button); // clicking on button;

  const Searchbox = screen.getByTestId ("Search");

  fireEvent.change (searchbox, {target: {value: "bisyani"}});
```

② → This is Similar to `onchange={(event) => console.log (event.target.value)}`;

→ we have 'event' in browser but not into js dom. So

② we are writing into our searchbox with target value as "bisyani".

```
});
```