

03/11/24

Episode - 13

Time for the test.

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

Ques: 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.

Ques: Why do we need testing?

Ans: Suppose you have added / deleted some code from your application, then that change affect 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

Ques: Types of testing? (which a developer can do)

- 1) unit testing,
- 2) integration testing
- 3) End to End (e2e) testing

**\* Unit Testing :-** You test your ~~real~~ 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 <sup>Restaurants card</sup> component or not, How if 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 logging in, checking restaurants, adding it to card etc. till 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 care major concern about the first two types of testing i.e, Unit testing & Integration testing.

→ In some companies you will need to do both, creating different <sup>features</sup> tests & testing it.

→ We are not going to learn about e2e testing.

→ So this part is left for you to explore it.

- What libraries do we need to do testing :-
- ① → use it as dev-dependency.
  - ② → as a dev dependency.
- React testing library is a part of ~~the~~ 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 used 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 website dependencies & config files are required for using Jest along with babel.

12/08/24

- 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 override our babel's configuration for Jest.
- To solve this you need to disable babel transpilation in parcel
  - To do this go to parcel document → Javascript → Babel → usage with other tools and copy past the given code of theirs in a file → parcelrc.js 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 does not 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 you under scripts, if we have choosed "Jest" as test framework then we will write the code with

```
scripts: {
  "start": "npm run test" and the script will look like
  "test": "Jest"
}.
```

That's why we are using `npm run test` to start Jest to run over 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:- Cho to `react-testing-library` website, into setup is  
to see if you are using Jest version  $\geq 28$  you need to  
separately install `Jest-environment-jsdom`

`npm i -D jest-environment-jsdom == npm i -D save-dev`

→ In older versions `Jest-environment-jsdom` would come along with `react-testing-library` by default.

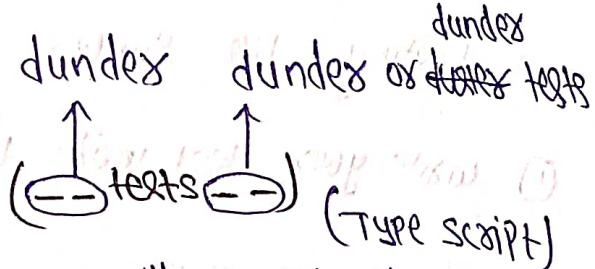
→ When you will do testing then you will a new folder named as `coverage`, add it to `git ignore`, it is just made to keep a coverage of all the testing files.

`coverage` → How many files have been covered.

17/03/24

## \* Testing.

- ① Where should I write test cases :-
- 1) Inside --tests-- folder:  
↳ Inside this folder write test cases with `js` or `ts`
  - 2) Name of file `spec.js`  
Name of file `spec.ts`  
Name of file `test.js`  
Name of file `tests.ts`  
dot → ↳ Anywhere inside your project folder



- \* What ever file you have inside `--tests--` Test will track them  
Test will consider all the files inside `--tests--` as a testing file
- Inside your `--tests--` you can name test file as :-  
`SumTest.js` / `SumTest.ts`
  - But outside of your `--tests--` folder you need to write as :-

- NP ① `Sum.test.js` / `Sum.test.ts` will be considered as test
- ② `Sum.spec.js` / `Sum.spec.ts` without tests

- ② What to write inside your test file.

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

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

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

```
const result = sum (3,4);
```

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

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

So we have created a `Sum.test.js`  
inside `--tests--` folder,

## → Testing file Syntax

- ① Write your test inside test method.

```
test('it', () => {});
```

description of the test

A callback function, which will actually do the testing

- ② Inside your test method

```
const result = sum(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 use command

```
npm run test
```

anything

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

→ How to write test cases for React ??

\* We will do unit testing.

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

\* Whenever you are testing any React UI component, you need to render it on jsdom.

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

```
test ("should load contact page", () => {
```

Here we can render more renders (<Contact />); // Rendering contact page component onto jsdom.

```
const Heading = screen.getByRole ("heading");
```

```
expect (Heading).toBeInTheDocument ();
```

```
});
```

→ 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 test doesn't understand JSX, it understands html.

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

```
npm i -> @babel/preset-react.
```

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

We need to add

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

→ correct 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 elements, we have different role, these roles are fixed by a testing library.

For `h1, h2 ... hs` we have a role `heading / Heading X`

For `<input type="text">` we have a role `textbox`.

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

→ We can also use something like expect (`heading`), not tobe (`3`)

→ expect (`heading`). tobeInTheDocument (`1`);

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");`  
`const heading = screen.getByRole("heading");` → you will get the <sup>virtual dom</sup> ~~React element~~/  
~~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 file, if we make any changes into our test file & save 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 one test function.

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

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

→ beforeAlt function

beforeAll ( ) => {

```
console.log ("I will be called before printing /executing all test");
```

۱۰

→ after all function

AfterAll ( ) => {

```
console.log ("I will be called by after executing all test");
```

319

→ beforeEach function

beforeEach(() => {

```
control.log("I will be called before executing each test case");
```

{ }<sup>o</sup>

→ afterEach function

afterEach(() => {

    console.log("I will be called after each test call");

});

→ import "@testing-library/jest-dom"; → to usetoBeInTheDocument(),

contains() and other related functions.

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

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

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

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

~~fetch~~.global.fetch =

Jest.fn(() =>

    return Promise.resolve({

        → It will create a function.

        → It takes a callback function  
        which we want

        return JSON.stringify({});

    });

    return Promise.resolve(data);

},

});

→ 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 fetch // fetch returns a promise
        .then(res => res.json()) // i.e., we are also returning a
        .then(data => data) // promise
    }
  })
})

```

return Promise.resolve(data);

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(s) it will return s 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),
    }
  })
})

```

It's a good practice to use await in the code.

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

```

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

```

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

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

```
import { act } from "react-dom/test-utils";
it("should render by component", () => {
  await act(() => {
    render(
      <Provider store={appStore}>
        <Router>
          <Component />
          <BrowserRouter>
            <Provider>
              <div>
                <div>
                  <div>
                    <div>
                      <div>
                        <div>
                          <div>
                            <div>
                              <div>
                                <div>
                                  <div>
                                    <div>
                                      <div>
                                        <div>
                                          <div>
                                            <div>
                                              <div>
                                                <div>
                                                  <div>
                                                    <div>
                                                      <div>
                                                        <div>
                                                          <div>
                                                            <div>
                                                              <div>
                                                                <div>
                                                                  <div>
                                                                    <div>
                                                                      <div>
                                                                        <div>
                                                                          <div>
                                                                            <div>
                                                                              <div>
                                                                                <div>
                                                                                  <div>
                                                                                    <div>
                                                                                      <div>
                                                                                        <div>
              </div>
            </Provider>
          </div>
        </Router>
      </Provider>
    );
  });
});
```

→ How to add event listeners in test code

```
import { fireEvent } from "@testing-library/react";
```

→ 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 render", () => {

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

fireEvent(button).click

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

const searchbox = screen.getByText("Search");

(2) → fireEvent.change(searchbox, {target: {value: "biryani"}});

→ 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 "biryani".

});

# ReGEX

Regular expressions (regex) in JavaScript are patterns used to match character combinations in strings. Here's a simple explanation along with the syntax:

## 1. What is Regex?

- Regex is a sequence of characters that forms a search pattern. It's like a super-powered find-and-replace tool for text.

## 2. Syntax:

- /pattern flags

- The pattern is the regex itself, enclosed between forward slashes ('/'). It's what you are searching for.
- Flags are optional and modify how the pattern is matched. Some common flags are `g` (global match), `i` (case-insensitive), `m` (multiline), etc.

## 3. Examples:

- /hello/ matches "hello" in a string.

- /[0-9]+/ matches one or more digits in a string.

- /^start/ matches "start" only at the beginning of a string (^ is an anchor for start).

- /end\$/ matches "end" only at the end of a string (\$) is an anchor for end).

## 4. Using Regex in JavaScript:

- JavaScript provides a `RegExp` object to work with regex. You can create a regex pattern using `new RegExp( pattern , flags )` or simply use regex literals like `/pattern/`.

## 5. Example Usage:

```
const str = 'Hello World';
```

```
const pattern = /hello/i; // Case-insensitive match for "hello"
```

```
const result = pattern.test(str); // Checks if the pattern matches the string
```

```
console.log(result); // Output: true
```

In essence, regex in JavaScript is a powerful tool for string manipulation and searching based on specific patterns.

## Promise.resolve()

Promise.resolve is a method in JavaScript that creates a new Promise object that is already resolved with a given value. It is commonly used to wrap synchronous values or values that are already promises to ensure consistent handling of asynchronous operations.

Here's why Promise.resolve is used:

1. Wrapping Synchronous Values: If you have a synchronous value that you want to treat as a promise, you can use `Promise.resolve` to wrap it in a promise.

```
const value = 42;  
const promise = Promise.resolve(value);
```

2. Handling Asynchronous Operations: Sometimes, you may have functions that return promises, and you want to ensure they always return a promise, regardless of whether the operation is synchronous or asynchronous.

```
function asyncOperation() {  
  return Promise.resolve(someAsyncFunction());  
}
```

3. Creating a Resolved Promise: If you need a promise that is already resolved with a specific value, you can use `Promise.resolve` instead of manually creating a new promise.

```
const resolvedPromise = Promise.resolve('Resolved Value');
```

Overall, `Promise.resolve` simplifies working with promises by providing a consistent way to handle both synchronous and asynchronous values within the context of promises.