**TIME FOR TEST:**

Today we will learn how to write the testcases. Before learning test cases we will know why do we even write the testcases. What is the importance of testcases and what are the types of testing. Let us see know what are different kind of testing that a developer can do. First one is manual testing. Manual testing means as a developer when I develop some feature I will go the that page and try to test by performing the actions which were developed. But this is not very efficient because I change something in the code do I need to test all the feature in my application. We are not going to test everything in our application whenever we change something. Even if we introduce one new line in the code it might lead to any bug anywhere in our app.

The above is one way which is manual testing. There is another way which is we can write the code to test the application. Writing the test cases which will test our application automatically. What are different types of testing that we can do in our React application as a developer. There are three types of testing that we can do by writing code. **Unit Testing, Integration Testing, End-to-End Testing.**

What is Unit Testing? – We test our React Components in insolation. Suppose, we want to test our header component and not concerned about another component. I just want to test header component then we do the Unit testing for that one component. Basically, we will try to render our header component and see that it got rendered or not.

What is Integration Testing? – Testing the integration of the component. In the large application, there will be lot of components and these components talk to each other. When we search the pizza in our search bar there are lot of components collaborated to show the result. So, it is a integration of so many components. There are multiple components they are talking to each other and we will develop a flow of an action in our React Application that we will test. So, we will write the code to test the integration.

What is End to End Testing? – Testing the React Application as soon as the User lands on the website to the User leaves the website and we test the all the flows. End to End Testing requires different types of tools like Cypress, Puppeteer, Selenium. These types of tools are needed to do End to End testing.

But as a developer we are majorly concerned about the first two types of testing. In some companies, testing is part of development. Here we mainly focus on the first two types of testing which are Unit Testing and Integration Testing.

Now, we will see what libraries are needed for testing the React application. The first library we are going to see is the React Testing Library. We are going to use the React Testing Library to write our testcases in our react application. This is the more standard library for writing the testcases in React. There use to be DOM Testing Library already. This Testing Library is very old library and they have come up with the different testing libraries they are Angular Testing Library, Vue Testing Library and so on. The React Testing Library is built on the top of the DOM Testing Library. The React Testing Library will already exist in your application if you use the create-react-app. So, As we are developing our app without using the create-react-app that means there will be no Testing Library pre-installed we need to install them. We are developing like this because we will get to know that when we want to build the large scale application we need a lot of libraries to install into project. **React Testing Library uses something known as JEST. JEST is a delightful JavaScript Testing Framework with a focus on simplicity.** JEST is a different library which we use for the React Testing Library so both are different.

We will install the React testing Library into our project by entering the command **npm i -D @testing-library/react.** We will also install the JEST into our project **npm i -D jest.** In Jest, if we are using this jest with the babel then we need to install some additional dependencies. After installing the babel dependencies, we need to configure that provided in the Jest page into our project by creating the filename **babel.config.js.** If you remember, we have discussed about the babel when discussing about parcel. When we are using parcel behind the scenes we are also using the babel inside our project. Babel is a transpiler and parcel uses babel. So, when we are using jest and when we are using the babel dependencies manually. These dependencies will interfere with the parcel’s babel dependencies. There will be a conflict between babel and parcel. So, when we write the dependencies of babel it will try to override the existing babel dependencies that parcel has set for us. We will have to change the parcel behavior to accommodate babel using along with Jest.

If you want to get override with the babel in the Parcel then go to the Parcel page and search for the Jest then it will redirect to the page if you read the page then it will tell you if you want to override the babel in Parcel with the babel you are configuring for Jest. To do this, we need to make the changes in the .parcelrc file. What is this file? – This is where the parcel configuration that we will be doing. So in the Usage with the tools we have code that we need to write inside the .parcelrc file. Then whatever we setup inside the babel.config.js it will be used for babel.

Now, how can we run our test cases. Similarly, when we want to run our application we use npm start. In the package.json if we go and see inside the script there is jest assigned to the test. We can use the npm run test this will basically run our testcases. Now, we are ready to write the testcases but we need to do one more configuration which is a Jest. For this, we need to execute command jest by writing **npx jest –init**. Basically, we are just initializing the Jest. If we run this command **it will prompt that whether we want to use the Typescript for the configuration file enter N**. **Then it will prompt to select the test environment that will be used for testing it will show the option node and jsdom (browser-like)**. When we run the test cases it will not run on server or the browser then the test cases need an environment like runtime where this test cases gets executed. For this we use the jsdom. JSDOM is a library which parses and interacts with assembled HTML just like browser. All our testing code runs here in this JSDOM. If we testing our header component then our header component needed to load here in the JSDOM. We will be using the JSDOM which is a very cool thing. We will be installing the JSDOM very soon. Let us choose the JSDOM for now. **The next prompt is do you want Jest to add coverage reports then we select the YES. The next prompt is which provider should be used to instrument code for coverage select babel. Automatically clear mock calls, instances, contexts and results before every test? – select YES.**

Now there will be a file with the jest.config.js which contains whatever prompts we have included and lot more comments. There is more thing to install that is jsdom library. For instance, if we are using the Jest 28 then we need to install jest-environment-jsdom package separately. Previously, it used to be come along with the Jest. But recently it stopped shipping along with it like in Jest 28 we need to install jsdom manually.

Before going to write the test cases for our whole project. We will write the test case for normal JS code not for react to learn from basic. If we write a normal sum.js file and just run the npm run test then it will show testMatch which will have \_\_tests\_\_. This is a folder whatever files under this folder will be tracked. The one way is that we can do like this and the other way is to name the filename as filename.test.js/ filename.spec.js we can do similar for ts, jsx. Why we add two underscore behind and after the test this is known as dunnder.

How to write the test cases? – we write the function test and this function test takes two arguments first is string and the second is a callback function. The first argument which is string is the description of the test and in the second callback function we write the implementation of the test case. Now, we will test how we will test first of all we call the sum function and it takes two arguments so we give some random two numbers and store it in a variable result. I want to test whether my result is the expected result or not. **We need to write the expect(result).to Be(expected\_value) this line is known as the assertion.** If we run the test then we get passed on to the terminal. It is not mandatory to write the test cases inside the test callback function but it is always good to have the assertion in our test cases.

Now we will see how to write the test cases in the React using Unit Testing. Let’s try to write the very basic test case in React. Let’s take the contact us component which is a very basic component. We are going to use the unit testing that means we are going to test one component independent of our app in isolation. Whenever we are going to test the UI component inside react you will have to render that component on to the JS DOM first of all. How we will do that? – So, there is a method which is render comes from the @testing-library/react. Now, this render method will take a component that is going to be rendered. Now, I want to render my contact component so we will write the contact component inside this render method. As we have rendered our component on to the JS DOM. Now we can test it but what to check and how? – So, when we render our page what we can see the header of the page which is Contact Us. Let’s see whether this is rendered or not by using the testcases. There is one more method the screen, this screen is an object which comes from react js library. So, whatever renders it gives access to the screen. This screen has lot of things like if we type screen.get there will be lot of method lets use the getByRole which is heading. Now once we get this heading we will try to expect this heading and toBeInTheDocument(). Which can be written as expect(heading).toBeInComponent(). This function will finds whether my heading is there in that component or not. If we go ahead and run the test case we get the error of synax error which says that “Support for the experimental syntax ‘jsx’ isn’t currently enabled”. How to enable the jsx inside our app we need to add the @babel/preset-react. We can add this in our app by running the command **npm install -D @babel/preset-react**. We need to add this inside the babel.config.js also. If we visit that config file there is already one preset which is configured we need to add this preset also inside this. So, in this array we pass [“@babel/preset-react”, {runtime: “automatic”}]. So, why are we adding this presets? – this babel preset react is helping our testing library to convert that JSX code to HTML to read properly.

If we run test again then it will fail once again. This time the error is the toBeInTheDocument is not a function because we haven’t installed one more library which is @testing-library/jest-dom. So, now if we want to check whether there is button on my screen or not then we will write screen.getByRole(“button”) which gives button. We need to expect this button and check toBeInTheDocument. What are the roles? – There are different kinds of roles these roles are decided by the Jest or the testing library. Role can be heading, button, etc. There can be different kinds of roles.

Suppose, I want to find the field input by the placeholder text then we can do screen.getAllByPlaceholderText. In case if we want to get all the input boxes to test then we can use the getAllByRole(“textbox”) which gets all the input boxes. If we do by trying getByRole then it will throw the error because if there are multiple textboxes then we need to do getAllByRole. So, if we console log to the returned element from the screen.getAllByRole(“textbox”) we get the JSX element or react element or virtual DOM object. Whenever we write the screen this whole line is referred to as Querying. Then we can assert on to it.

Suppose our test file increase like creating 20 test cases which increases the file. Now, it will become tough to handle such type of testcases. We have to create the separate group of testcases. How do we group this test cases basically we can write something known as describe. For example, the 5 test cases are testing some part of the page, 10 test cases are testing some part of the test cases, 5 other test cases are testing the some other part of the test case. We can also have describe inside the describe block. The “test” is the name of the function for the test case. Jest says that we can write the “it” in place of test. Why is the name “it”? – Some people will read the it(“Should load input name inside the Contact component”) just for simplicity case that’s it no meaning behind it.

Let’s start writing testcases for our header component. First of all, let’s implement writing unit testing for the header component to check whether it is rendering in isolation. When header is loaded whether it is loaded with login button or not. We can also see whether the cart has 0 items or not. First, we create the Header.test.js and in this file we start writing the test cases so as usual we start by it(“Should load header component with the login button”) and the second argument as the callback function which takes render(<Header/>);. If we run npm start test then it throws the error what is the error “could not find react-redux context value”. We are getting error on the line useSelector and this comes from the react-redux. Our application is using redux but we are trying to load the header component in isolation. Where we are render in JS DOM. This JS DOM understands the JSX code but it does not understand redux code. **What we need for this is we will have to provide the redux store to our header. Even we are loading in the isolation we have to give access to store to the header**. We can do this by importing the Provider from react-redux, add the Provider tag to the Header tag inside the render method and this provider will take the store attribute for this attribute we will assign the appStore which we have developed using Redux toolkit.

Now again, if we run the npm run test we will get error again. This time the error comes from the Link tag which we used in our header. This Link comes from the react-router-dom. So, for this we basically have to provide the router to this header. For doing we need to import the BrowserRouter from react-router-dom and we have to wrap the Provider and header tag inside this BrowserRouter tag. Now it will understand this Link tag. If we again run the npm run test this time it will run.

Now, we try to write our assertion to write the assertion first we need get the login button from the screen by using the getByRole(“button”) and it returns the buttons which are inside the header component and now we write assertion which is expect(loginButton).toBeInDocument(); and if the run this test case it will execute. We can also use the getByText(“Login”). But text is not the good way Role is the good way. If there are lot of button in the header then we can pass the second parameter inside the getByRole which is name as Login.

For suppose, if you want to check only the cart without the items we can use the regex inside the getByText by adding “/Cart/” then it will check only for Cart.

Now we will test that when we click the login button it will change to logout. To do this, we write the render method and we also get the login button from screen. Now, how to simulate this when we click login it is changing to logout. To do this, we have a method which is fireEvent we get this from @testing-libraru/react which has the lot of events. From this events, we take click and we give the login button to this event. Now we need to get the logout button using the screen.getByRole. Now, we write the assertion expect statement in which we give logoutButton and we expect this to be in the Document.

Let’s start testing the Restaurant Card component. Till now, whatever we have tested it does not have the props in the component but for this card component we have props. Now, we will see how to pass the props inside the component and test it. To do this, in the render method we pass the Restaurant Card component now we need to pass props to this component which is resData which takes the resData as prop and we need to pass the data into this props how we can pass? – So , we will pass the data as mock data that means we take first data in the object and pass it as mock. For this mock we will save it in a different file. We import this data into our test file and assign it to the resData. How to check whether this data is loaded or not. We check this by taking any of the text which is loading on the screen for instance lets take name of the restaurant and assign this to it. Then we can write the assertion like expect(name). toBeInTheDocument().

Now we will see Integration testing. We will test an important flow inside our application that is search flow. When we search something in the search box then it should show that data, we will simulate everything over here. When we enter something into the search and when we click the search then it needs to filter that data for this to work we need to work with multiple components. This is known as the integration testing.

We will create a file, which is Search.test.js. In this file, we render the whole-body component which contains everything. So, let’s write the Body tag inside the render method but this is not rendering it on the browser it is rendering it on the JS DOM which is browser like it does not have all the superpower of the browser. This fetch is a super power of the browsers. So, when we try to render the Body component it is rendering it on the browser-like. So, this browser-like doesn’t have idea about the fetch. What we need to do for this situation. We need to fake this fetch we need to create a mock function for this. So, we write replace the fetch function which is unable to find in the body component by the render. Now, we will defined our own dummy function for this fetch. If we do the jest.fn() it will give us the mock fetch. This function takes the callback and here we have to mock our fetch function. Fetch function returns us the promise right so in this function we also return a promise and this promise will resolve the json which again returns a promise so this json is a key and value is the callback function which returns a promise and this promise resolves the MOCK\_DATA which we stored in the mocks. This MOCK\_DATA is the data which we copy from the fetch API which is shown in the browser copy that exact data from the browser and paste it in this file.

So what we are trying to achieve is we cannot make the network call from the test so we create this function. After this we try to run again the command npm run test and it again fails. It’s very repetitive to run the exact command so many times. To overcome we can do similar to the server which is running our application this can be done by HMR (Hot Module Replacement). We will setup the exact same setup in our application. To do this we need to setup in the script **“watch-test” : “jest --watch” in the package.json**. With the help of this command of this jest –watch there is no need to run the command npm run test. In place of this we will run **npm run watch-test**. So, if we do this after that we got an error saying that we need to wrap the body inside the act. So, whenever we are doing the fetch or the state updates then we need to wrap our component inside function. So this act function comes from the react-dom/test-utils. How to use this act function? – First we need to place the await before this act function as this function return promise and the callback function in which we are writing our render and this act function we need to make this async. This act function takes another callback function for this we need to give async and this function calls the render method. If we save the file then again we get the error and this is about the Router. We need to wrap our Body component with the BrowserRouter. Now we will get the Search button by using the getByRole present in the screen which takes button and name:”Search” as arguments. Once we get the searchBtn now we write the assertion which is expect(searchBtn).toBeInTheDocument().

Now try to write something in this input box and click the search button. Now, how to write something inside this input box. Writing something inside the input box is a event in itself right. Just like the click event for the search button. We have a onChange event for the input box. So, we will have to trigger that onChange event to update our input box. First of all, let me get my input box inside the test case by using the screen. I will get my search input by using the screen now how will I get my search input from screen by using the getByTestId. To make this work, we have to go back to where this input box is present. Now, I want to get this input box inside the my test so what I can do is I can give test id to this like **data-testid=”searchInput”** and jest will read it. Now we can find this testid in the test file by getByTestId(searchInput). This is an another way to find out the search input inside the screen. To change the input of this input text box we need to trigger the fireEvent.change(). This will take two arguments one is the searchInput which is input box we get from the body component. The second argument is the object and this object is basically simulating what we get in the onChange event of the input box in the body component. There is a callback function inside this onChange and this function takes the “e” as parameter and this “e” is given by the browser. But this “e” is not present inside this test file because there is no browser. This e.target.value will be in the browser. To simulate this in our test file we need to give the second argument as the target as key and value again is an object which contains value as key and its value is the what we want to give in the search box let’s say burger. Now I want to click the search button to do this we need to give the fireEvent.click(searchBtn). Now we need to write the assertion to test how many cards are there on the screen to do this in the screen each card is a div and to know how many div’s are there we need to go to that certain component to find the div and give the data-testid = “resCard” and if we inspect our web page and see each div for the card is having the attribute as data-testid=” resCard” and we can get this into our test file by using the const cards = screen.getAllByTestId(“resCard”). Now in the assertion we can expect the cards.length to be 4 cards. To write this, we need to write expect(cards.length).toBe(4);

There is something known as the beforeAll and beforeEach. If we want to run something before all our test cases we use the beforeAll. If we want to do something before each test case then we use the beforeEach. Similarly, we have afterAll and afterEach.

**Assignment:**

What are different types for testing?

Unit testing, Integration Testing, End to End Testing

What is Enzyme?

Enzyme is a JS testing utility for React that makes it easier to test your React Components output. You can also manipulate, traverse and in some ways simulate runtime given the output. Enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal.