Chapter 13: Time to Test

* **React Testing Library**
* Why we need Test cases?
* Test driven development

we write tesst cases even before we write our code.

**Headless browsers:**

A headless browser is just like any other browser, the only difference is we cannot see anything on the screen. Here we can say that the program actually runs in the backend and nothing can be viewed on the screen. Thus, it is known to be the one without a Head/GUI.

Just like a normal browser a Headless Browser performs all the functions like clicking links, navigating pages, downloading the document, uploading a document, etc. by performing all the instructions as per our program.

using headless browser, we can test faster.

**Unit Testing**

this is the core job of developer

**Integration Testig**

Testing it is exactly like a development, so it will take some time to get along with it.

***React testing library*** is replacement for ***enzyms***

React testing library is using ***JEST***

Jest is a delightful JavaScript Testing Framework with a focus on simplicity.

It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!

**Setup React Testing Library**

Install the library

npm install testing-library/react

also, we will require Jest with this

npm install jest

after installing jest, we will configure JEST. to do that we will create ***jest.config*** file and better way to create is

npx jest --init

***here we use npx because we just want to do it once.***

1. Yes/No for typescipt configuration.

2. Select the test environment

here we will use ***jsdom*** browser like dom.

3. Add coverage report.

we will add this as we will get the coverage of our code.

4. Provider for coverage

we will use Babel

5. Auto clear mock calls, instance, results etc before every test?

This is like a garbage collection thing and we will use this.

After doing this it will create jest.config.js for us.

now we can run the test by giving below command.

npm run test

here we will get lot of errors

because this library is not shipped along with the package after the version 28

to get that package we need to install it as below,

npm install jest-environment-jsdom

after installing the error related to version will get remove along with this we will get something like,

***test cases not found etc.***

**How we can create the test file and Tests?**

in our components folder we will create a folder named as ***\_\_tests\_\_*** and whatever files we put inside this folder jest will identify as a test files.

**How to test in JavaScript.**

as an example, we have created a file named as sum.js for which we will create the test file under *\_\_tests\_\_* folder named as sum.test.js

this is convention used for files names...

***fileName.test.js***

Writing our first test case as below

test("check the sum of 2 numbers", Function)

first argument is the name of the test

second argument will be a function, this function will be the code that this test case will execute

whenever there is a test case there will be a expect inside it, we call this as assertion also.

here we will expect from this test to call our sum function with two parameters and return the result

import {sum} from "../sum"

test("check sum of 2 positive numbers", () => {

expect(sum(2,5)).toBe(7);

});

after this if we tried to run test again we will get an error as : *Cannot use import outside a module.*

to fix this error we have to do type module. we need to take help babel in this. we need to configure our jest with babel

Configuring Babel

we will install few packages that are require

npm install babel-jest @babel/core @babel/preset-env

once installed we also need to configure it.

create babel.config.js

add below in babel.config.js

but as we already having the babelrc file we will add the required configuration.

if we copy the babel configuration to the babelrc file it shows error.

module.exports = {

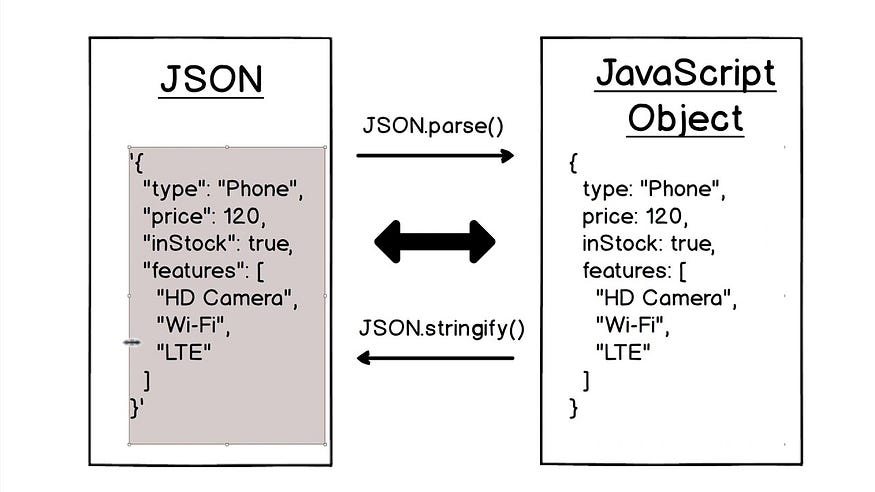
presets: [['@babel/preset-env', {targets: {node: 'current'}}]],

};

because babelrc file requires a JSON, after configuring this it will work.

**What is a difference between JSON and JS Object?**

|  |  |
| --- | --- |
| **JSON** | **JavaScript Object** |
| The key in key/value pair should be in double quotes. | The key in key/value pair can be without double quotes. |
| JSON cannot contain functions. | JavaScript objects can contain functions. |
| JSON can be created and used by other programming languages. | JavaScript objects can only be used in JavaScript. |



while doing the testing like these,

it will not run our app.

Jest will check our code-base in the conditions we give in test file.

it will not render anything on the browser.

this is because in our jest config file we have given the testEnvironment as the jsdom. which will have only few functionalities of our browser.

**JSDOM**

**JSDOM is a library which parses and interacts with assembled HTML just like a browser.** The benefit is that it isn’t actually a browser. Instead, it implements web standards like browsers do.

You can feed it some HTML, and it will parse that HTML. Then, you can inspect or modify that HTML in-memory using the normal JavaScript [DOM API](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model).

we need to code coverage in our local machine and do not want to upload into git or server. therefore we will add the /coverage in the .gitignoe file.

**Unit Testing**

Testing small components of the test.

Whenever we write test case name it should be descriptive.

e.g. test("Logo should load on rendering header")

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

import Header from "Header";

test("Logo should load on rendering header", () => {

//load the logo

const header = render(<Header />);

console.log(header);

//check if the logo is loaded

});

we don’t have cannot directly render to root as we are not having access to html root element. but Jest gives us render in which we can render anything we would like to.

here we are passing the <Header/> to the render of Jest.

after saving this and running test still we will get the error as below,

Support for the experimental syntax "jsx" isn't currently enabled.

we need to install one more package to make this understand. we will use babel to add this package.

module.exports = {

presets: ['@babel/preset-env', {targets: {node: 'current'}}],

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

};

like above we will add the configuration for jest to understand jsx.

after running the test again, we will get error as below,

Unexpected character (for the logo that we importing as png image)

this is because Jest is trying to read the image as JavaScript. Jest cannot read images so we need to create a mock or dummy images data for the test.

but who will help to tell that instead image we need to use the dummy image? it is Jest.

in package.json file we have a property named as moduleNameMapper. we will tell out Jest to take all the png images form the dummy image.

we will use configuration to do that

moduleNameMapper :{

"\\.png" : "../mocks/dummyLogo.js"

//if we want to add multiple file formats we will do something like below,

"\\.(jpg|png|svg)$ " : "../mocks/dummyLogo.js"

}

again, we will run our test cases. and again, it will be failed.

error: could not find react-redux context value, please ensure the component is wrapped in a <Provider/>

we get this error because we have subscribed to our store using useSelector in the Header component.

**we need to add the provider but how?**

we will go to our test case file and we will create a actual store. we will add provider as well as store

import { Provider } from "react-redux";

import store from "../utils/store";

test("Logo should load on rendering heade", () => {

const body = render(

<Provider store={store}>

<Header />;

</Provider>

);

console.log(header);

});

//check if the logo is loaded

after this we will run test case again and again failed...

error: useHref() may be used only in the context of a <Router> component.

this is because we have used appRouter, routerProvider and <Link/> in it. but our JestDom does not understand Link.

so along with store we need to get Link as well.

but the createBrowserRouter will not work here in JestDom because it is not a browser.

so instead of createBrowserRouter react-router-dom give us something known as ***StaticRouter***. this StaticRouter will work without browser.

import { StaticRouter} from "react-router-dom/server";

and we will provide the StaticRouter to our app by wrapping it with StaticRouter

import { Provider} from "react-redux";

import store from "../utils/store";

import { StaticRouter} from "react-router-dom/server";

test("Logo should load on rendering header", () => {

//load the logo

const header= render(

<StaticRouter>

<Provider store={store}>

<Header />;

</Provider>

</StaticRouter>

);

console.log(header);

//check if the logo is loaded

});

and we will run it again. this time it will be passed.

like this we have setup our testing env and rendered our sample test app.

now need to add logic for actual testing, what need to be expected.

2:15:00

in case of actual Dom, we will get the logo as below,

document.getElementById..

but in case of Jest we will write similerly as below

const logo = header.getAllByTestId("logo");

but as we have not added this test id in the code for logo in header component. we will add it.

<img data-testid="logo" id="logo" src=...../>

here the thing to remember is that as Dom understand the id attribute the JEST understands the data-testid attribute.

now to test we will write below command

expect(logo.src).toBe("dummyLogo.png");

and we will get error as below after running the test.

Logo should load on rendering header

this is because in we have received a array of elements byId therefore we need to write the test case as below,

expect(logo[0].src).toBe("dummyLogo.png");

after saving and runing again we will get below error.

Logo should load on rendering header

Expected: "dummyLogo.png"

Received: "http://localhost/dummy.png"

so, to pass the test case we will modify like below and re run the test.

expect(logo[0].src).toBe("http://localhost/dummy.png");

Test Passed!!!

Test Cases for Online

first we will add the test id in the online component.

<h1 data-testid="online-status">{isOnline ? "Yes" : "No"}</h1>

we will create test case as below similer to the one we did earlier for the logo.

import { Provider} from "react-redux";

import store from "../utils/store";

import { StaticRouter} from "react-router-dom/server";

test("Online status should be Yes on rendering header", () => {

//load the logo

const header= render(

<StaticRouter>

<Provider store={store}>

<Header />;

</Provider>

</StaticRouter>

);

const onlineStatus = header.getByTestId("online-status");

expect(onlineStatus.innerHTML).toBe("Yes");

});

here a change we did id instead of getting all the files we can use getbyTestId to get the single element.

This is how we can do Unit Testing by testing a signle unit as we did here by testing the header component.

2:50:00

Integration Testing

1. Testing for Search box

First lets render the search page on the home [page.

import { Provider } from "react-redux";

import store from "../utils/store";

import { StaticRouter } from "react-router-dom/server";

import Body from "../Body";

test("Search Results on Home Page", () => {

const body = render(

<StaticRouter>

<Provider store={store}>

<Body />;

</Provider>

</StaticRouter>

);

console.log(body);

});

after adding this test case and runnig we got so many errors.

Error: ReferenceError: fetch is not defined.

here fetch is a Javascript thing and so we need to mock the api calls. this happens becasue we are not running our code in the browser.

we are not having the network access to this so we cannot make a network call in our test cases(we can do it idealy but we don't have to.)

in this case we will need a mock data to make api call so we will try mock our fetch also. and we will do it like below.

global.fetch = jest.fn()

here fn() is a dummy function given by JEST, we will write what fetch actually do in api call.

as we know that the fetch will return a promise so will resolve a promise here in this function.

likewise we can reject the promise inorder to test for api failure case.

we will resolve the promise by JSON data(Whenever we make api call it will retunr a str eam of data that data we convert into the JSON data)

in resolve() we will pass the data that we have to mock.

we will get the data that api returns and we will craete a file with that data which will act a dummy data(RESTAURANT\_DATA) for our fn function.

import { Provider } from "react-redux";

import store from "../utils/store";

import { StaticRouter } from "react-router-dom/server";

import Body from "../Body";

import { RESTAURANT\_DATA } from "./data";

global.fetch = jest.fn(() => {

return Promise.resolve({

json: () => {

return Promise.resolve(RESTAURANT\_DATA);

},

});

});

test("Search Results on Home Page", () => {

const body = render(

<StaticRouter>

<Provider store={store}>

<Body />;

</Provider>

</StaticRouter>

);

console.log(body);

});

by doing this our code will automatically understand what is happening in fetch().

03:08:00

After running this also we will get the error as below,

TypeError : Cannot read properties of undefined (reading “json”)

This happen because we have created the promise but we have not returned it.

global.fetch = jest.fn(() => {

return Promise.resolve({

json: Promise.resolve(RESTAURANT\_DATA),

});

});

New Error data.json is not a function , this is because

global.fetch = jest.fn(() => {

return Promise.resolve({

json: () => Promise.resolve(RESTAURANT\_DATA),

});

});

Next we need to move the data.js file into the mocks folder that we have created earlier because testing library considers this data files as a testing file and will give an error.

We gets below, error

Unable to find an element by : [data-testid=”searchBtn”]

If we see the terminal we can see the shimmer is getting loaded and we gets this error.

So instead of searchBtn we will add shimmer for testing, for this we will add below code and we will add the the shimmer id to our shimmer in shimmer component.

<div className = “reastaurant-list” data-testid = “shimmer”>

test("Shimmer should load on Homepage", () => {

const body = render(

<StaticRouter>

<Provider store={store}>

<Body />;

</Provider>

</StaticRouter>

);

const shimmer = body.getByTestId("shimmer")

console.log(shimmer);

});

***\*NOTE:*** *Here we have to rerun so many times the test command instead of that we can achieve the HMR – Hot Module Reload by adding the scripts in the package.json file,*

*“watch-test”: “jest --watch”*

*After this if we give npm watch-test it will keep running.*

Once we have completed the shimmer test case then we need to add the expect block.

expect(shimmer).toBeInTheDocument();

In expect block we will use **.toBeInTheDocument** function from library which is known as **@testing-library/jest-dom**

.toBeInTheDocument: The jest-dom utility library provides the .toBeInTheDocument() matcher, which can be used to assert that an element is in the body of the document, or not.

And after adding the expect block with the toBeInTheDocument function we our test case will be passed.

It is basically checking whether shimmer is loaded in the document or not.

But this is not the correct way of testing.

If we check the inner html inside the terminal we can see that the shimmer component is having the shimmer cards inside of it.

expect(shimmer,innerHTML).toBeInTheDocument();

and what if we want to test whether the shimmer cards have been loaded properly or not?

We will do like below,

expect(shimmer,children).toBeInTheDocument();

by doing this we can see the children’s in the terminal window.

Next thing is after rendering the shimmer we need to wait for the content to be loaded.

and we can do this by using the **await** and **waitFor()** function given by the react testing library as below,

await waitFor(() => expect(screen.getByTestId(“search-btn”)))

here as we are doing async operation using await we need to add async in beginning

and as we have added 10 shimmer cards in the shimmer component we will check for that like below,

expect(shimmer.children.lebgth).toBe(10);

like we we cxheck for number of shimmer component we can also check the number of restaurants rendered in the body component. To do that we will follow below steps.

1. Add the test id for the body component where we are rendering the cards.

<div className="flex flex-wrap" ***data-testid = “res-list”***>

{updateSearch.map((a) => {

return (

<Link key={a.data.id} to={"/restaurant/" + a.data.id}>

<RestroCard {...a.data} key={a?.data?.id} />

</Link>

);

})}

</div>

1. Do the changes in the test page as below,

await waitFor(() => expect(body.getByTestId(“search-btn”)))

const resList = body.getByTestId("res-list ");

expect(resList.children.length).toBe(15); // here 15 because first we tried with 10 and got error saying that the expected was 10 but received 15.

Next Test case : Find the input box -> type something into it -> click on the search button.

First we will add the test id for the input box in the body component where the input box is present as below,

<input

className="search-input"

***data-testid*** = “*s****earch-input***”

type="text"

value={searchText}

placeholder="Type to Search"

onChange={(e) => updateSearchText(e.target.value.toLowerCase())}

></input>

test("Search for the string(food on homepage"), async () => {

const body = render(

<StaticRouter>

<Provider store={store}>

<Body />;

</Provider>

</StaticRouter>

);

await waitFor(() => expect(body.getByTestId(“search-btn”)));

const ***input*** = body.getByTestId(“***search-input***”);

after this we cannot type in th serach box so we need to moc that input as well, to do this we need one more component from the react testing library known as fireEvent

fireEvent[eventName]

fireEvent[eventName](node: HTMLElement, eventProperties: Object)

Convenience methods for firing DOM events.

target: When an event is dispatched on an element, the event has the subjected element on a property called target. As a convenience, if you provide a target property in the eventProperties (second argument), then those properties will be assigned to the node which is receiving the event.

We need to fire the onChange event that we have added in the input Box, we will import as below,

import {render, wiatFor, ***fireEvent***} from "@testing-library/react"

fireEvent.change(input, {

taget: {

value: "food",

},

});

Here we have added the **target** because in the searchBox we have an synthetic event as e.target.value, so we are just trying to replicate here.

All these for typing the food in the search box automatically.

Next we will fire the click event using the fireEvent as below,

Const searchBtn = bosy.getByTestId(“search-btn”)

fireEvet.click(searchBtn);

and after searching the food in the app app we are getting 3 results so we will try to add this info in the expect block as before,

const resList = body.getByTestId("res-list");

expect(resList.children.length).toBe(3);

***\*NOTE:*** *If we want to see the HTML in the terminal we will give wrong testId in the getByTestId like below,*

const resList = body.getByTestId("***wrong\_test\_id***");

Next Test case : if we click on Add our card should update

First we need to get the restaurant menu,

Here we need to moc data for the menu data of specific restaurant.

import "@testing-library/jest-dom";

import { render, wiatFor, fireEvent } from "@testing-library/react";

import { Provider } from "react-redux";

import store from "../utils/store";

import RestaurantMenu from "./RestaurantMenu";

import { StaticRouter } from "react-router-dom/server";

import { MENU\_DATA } from "./data";

global.fetch = jest.fn(() => {

return Promise.resolve({

json: () => {

return Promise.resolve(MENU\_DATA);

},

});

});

test("Add items to Cart", async () => {

const body = render(

<StaticRouter>

<Provider store={store}>

<RestaurantMenu />;

</Provider>

</StaticRouter>

);

await waitFor(() => expect(body.getByTestId("menu")));

//we need to add test id in the RestaurantMenu component and for the addItem button

//<ul data-testid= "menu"><ul/>

//<Button data-testid= "addBtn"> Add Item <Button/>

const addBtn = body.getAllByTestId("menu");

fireEvet.click(addBtn[0]);

// as we will have multiple addButons we will click on first button.

const cart =body.getByTestId("cart");

expect(cart.innerHTML).toBe("Cart - 1 item");

});