* What are test cases?

Test cases in React are code snippets that ensures a React component behaves as expected. These test cases can be written using various testing frameworks, such as jest, Mocha, and Enzyme etc. The purpose of test cases is to catch bugs easly in the development process, validate that changes to the code do not break existing functionality.

* Why do we need test cases?

All the components are inter connected to each other. So, we need test cases because to ensure that our code is properly intact. Whatever we are ~~coding~~ writing passes test cases. In short, if we are adding new features, they should not break the existing ones.

Testing gives us lots of confidence that I am not breaking the existing flows. This is the care of testing.

other reasons are :—

→ Catch bugs early

→ Helps in maintain code quality

→ Improve confidence in code.

* Test Driven Development :—

In test driven development, we write test cases even before writing code. It means we will have 100 % test coverage before writing actual code.

* Different types of testing :—

- Manual testing :—

  It requires human for testing the app. It can be time consuming and prone to human error.

- Automated testing :—

  It means code tests a code. This testing doesn't require human.

- End-to-end testing :—

  It focus on verifying the complete flow of the software, from start to finish, to ensure that it behaves as expected. It requires a lot of efforts because the test cases are so huge, it covers entire user journey.

* Selenium :—

It provides a way to write tests that can simulate user interactions with a web application, such as clicking buttons, filling out forms and navigating between pages.

Selenium is commonly used for testing web applications and it supports a variety of languages, including Java, python C# and Ruby. It can be used to test applications running on different browsers, including chrome, firefox etc.

Selenium is a popular and widely used tool for automating web browser testing, and it is an essential tool for any development team working on web applications.

* Cypress :—

It is an end-to-end testing framework for web applications. It is designed to make it easy to write and run tests that simulate user interactions with a web application, such as clicking buttons, filling out forms and navigating between pages.

* Headless browsers :—

Headless browsers are browsers that are run without a graphical user interface (GUI). In other words, they are command line tools that can be used to automate web browsing and testing tasks, but they do not display a graphical interface to the user.

Example :— PhantomJS, Headless chrome, HTML Unit.

In Headless browser we can run test cases faster because it is headless, it does not have to do paint on the browser.

- Unit testing :—

It is the core job of developers. It means, we have to test small units. It is kind like a small component feature testing. A unit in software development is usually defined as the smallest testable component of the software. It could be a function, a method, or a class, depending on the programming language.

Benefits of unit testing :—

- Improved software quality and reliability
- Faster and easier bug identification and fixing
- Increase confidence in making changes to the codebase.
- Improved documentation and understanding of the code.

- Integration testing :—

Integration testing is a software testing method where individual units or components of the software are combined and tested as a group, to verify that ~~was~~ they work together as expected.

The goal of integration testing is used identify any issues or bugs that may arise from the interaction between the different units or components of the software.

other testing are :—

- function testing
- system testing
- Smoke testing
- Regression testing
- Security testing

etc... etc.... etc.....

* React Testing Library :—

React testing library is a part of the testing library. There is something known as Testing Library and this testing library offers testing for React, Angular and other frameworks also

* **Jest** :— ⟶ It is a package

Jest is a delightful javascript testing framework. If we need to test javascript code we use jest.

But React Testing library uses Jest behind the scenes.

⇒ How to set up React Testing Library

```
npm install --save-dev @testing-library/react
```

→ Now, also we have to install 'Jest'

```
npm i -D jest
```

→ Now, we will configure the jest, by creating jest.config.js file.

→ Better way of creating above file is by using

```
npx jest --init
```

we are using 'npx' not 'npm' because we want to initilize it only at once.

when we initilize it, it will help us create 'jest.config.js'
After this, there will be some questions, like this.

- would you like to use Typescript for the configuration file (y/n)
  If you using Typescript then type yes otherwise No.
- choose the test environment that will be used for testing
  jsdom (choose then press enter)
- Do you want Jest to add coverage reports? (Y/N)
  I will go with 'yes', you can do 'No' also (It is user choice)
- Which provider should be used to instrument code coverage?
  Go with 'babel'
- Automatically clear mock calls, instances, contexts and results before every test? (Y/N)
  Yes

And after this all you will see a jest.config.js file is created 🙂

→ Now we will run a test command :—

```
npm run test
```

After running this above command, we will see a error. Now its time to fix it. See what the error is saying

Error is saying :-

" As of jest 28 "jest-environment-jsdom" is no longer shipped by defaut, make sure to install separately "

It means we have to install "jest-environment-jsdom" separately, like this

→ npm i -D jest-enuvironmenment-jsdom

After running above command, run again below command

npm run test

After this, we will not get a error, but it will say that - "I have checked 30 files (can be any number) but did n't find any test cases written any of these files." like this.

30 files checked

testMatch: **/__tests__/**/*.[jt]s?(x),

**/?(*.)+(spec|test).[tj]s?(x) - 0 matches

From above, basically 'jest' try to find out test cases in our whole project because we have not written any test case yet. And it says, I am checking the **/__tests__/**/* folder but it is not there.

→ Now we will create our first test.
We will create a folder inside 'components' folder named will be __tests__.

This name is mandatary because whatever files we will create inside this, just will be consider as testing files. "__tests__" is also called as 'Dunder'.

Before moving into react testing, let's do first javascript testing because 'jest' is a javascript framework. Once we understand the JS testing, 'React' will be very easy to understand.

Let us write test cases for 'sum'. we will create a file inside '__tests__' folder, named will be sum.test.js. This is a convention that uses in the industry (.test.js) for test files.

Let us write our first test case.

→ We will write 'test()' function, then we will pass our first argment, that is name of the test ("sum of two positive number") and now it will take second argument as a function. This function will be the code that test will

sum.test.js

1st argument

Second argument

```
test("check sum of two positive number", () => {
    // test cases (expectation)
});
```

Every test case should have a expectation

→ Now we will expect something from this test. Before moving further create a file 'sum.js' for this test and write code for addition of two numbers,

Ex:—

sum.js

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

→ Now we will write expectation for test() function like this

sum.test.js

```
test("check sum of two positive number", () => {
    expect(sum(2,5)).toBe(7);
});
```

↳ It means, this will expect '7' from the sum of two number, 'expect()' takes one argument, which has to be test.

Before run this test command we have to import 'sum()' function, like this —

sum.test.js

→ import { sum } from '../sum';

Now if we run our test command by using 'npm run test' it will failed the test because we have import the file (above file) in a wrong way. It will show error like this

→ Syntax Error: Cannot use import statement outside a module

The problem is our normal javascript files does not understand this 'import'. So basically we need to take help of 'babel' in this. Now we will configure babel along with JS.

→ Search on google :- jest babel config

click on the first link. Now in the documentation click on 'Using babel'. Jest will need some babel packages. copy the below code ↓

→ npm install --save-dev babel-jest @babel/core @babel/preset-env

Now, After installing above package we have to also configure it.

It will say, configure your 'babel.config.js' file. by coping this below command.

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

Basically, Now 'babel' will make understand that there is something known as 'import', we are writing ES6, 'Babel' job is to take some code and modify it according to the config.

Now we have to create 'babel.config.js'... but instead of creating this file we have already a file known as '.babelrc'. So when we have to configure babel, there are two ways to do it.

1. babel.config.js
2. babelrc

} Both are good ways. we can use both ways

→ Now paste the above code in the '.babelrc' file and it will throw an error because below code is valid javascript code not json code.

.babelrc

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

'.babelrc' requires a json code and json and javascript code are not same. So this requires a json. So convent the code with using double quotes (" ") like this :

.babelrc

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

Now, the above code is become a json code.
Now run, npm run test command and it will passed the test case.

So this is just, we have taught till now is just javascript testing.

There will a file named 'coverage' that we need only in our machine so put it into '.gitignore'.

So, now let us try to do unit testing. It means we will test small components like individual small portion of our app.

Let us try to write test case for as soon as our header component loads.

For 'header', first of all we will 'header' and we will see what we can expect, when we load the 'header'.

→ Can we expect that our logo should be there? yes

- When my 'header' load 'logo' should load, this will be our first criteria.
- Also check when ~~our~~ we load 'header' my cart item should be zero (cart item - 0).
- One more think let us also test, our default status should be 'Online'.

Let's write three things.

→ Let us create a file name with 'Header.test.js' inside the '__tests__' folder. Now we will wright test cases in React.

First we will write 'test()' and our first test case is to load logo thats why we will write "logo should load on rendering header" inside our 'test' function.

Remember this, the test name should be descriptive because it will be ~~easty~~ easy to understand what we have write at the time of error. By reading the ~~name~~ test case name we should understand that what we are doing in the test. Make always descriptive.

First of all we will try to load our **header** and we will just load our **header** component. Then check if **logo** is loaded.

- For loading **header**, we will import our **header**.

→ `import Header from "../Header";`

This code is running inside 'jsdom' not our browser. 'jsdom' is like a small machine which is running our code. In that machine we will just load our 'header'.

We will use, which is something known as '**render**' function which will come from our testing library react, like this

Header.test.js

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

test("Logo should load on rendering header", () => {
    // Load Header.
    render()
    // check if logo is loaded.
});
```

This special 'render()' function is given by testing library. This 'render()' is like a container in which we load our header component. like this ↓

→ render(<Header />);

Whatever we will render from above code, we will keep in our 'header' variable. like this ↓

→ const header = render(<Header />);

Now, if we run, npm run test command it will through an error, that is

SyntaxError: support for the experimental syntax 'jsx' is not currently enabled.

It means, our code 'jest' does not understand 'jsx'. Now, here 'babel' will help us. So we will intall one more package just like ~~@babel/preset-env~~ @babel/preset-env. We will install @babel/preset-react. and we will add some configuration for this also i.e {"runtime": "automatic"}, like this :-

.babelrc

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

runtime means a place where our code is run. So right now our code is running in 'jsdom', so it means take the **runtime** as **automatic**.

Now install above code, like this ↓

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

After installing, let's us try to run command again npm run test and after this it will fail again, and it will say

```
SyntaxError: Unexpected character '🖼'
```

← something like this

This error is saying that I am not understanding this 'Logo' in Header.js

Header.js

```
import Logo from "../assets/img/foodCoffe.png";
```

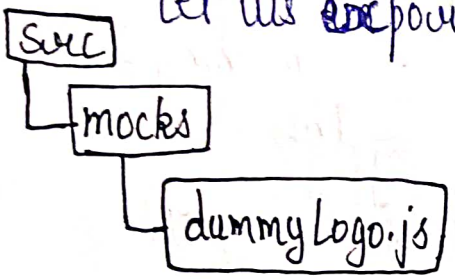
It is throwing error because it is trying to read this png image as a javascript. This 'jsdom' is trying to read this image as a javascript but it is png image. So, it means our **jsdom** does not understand **images**.

So, we have to find work-around, means in our testing when something breaks, we create a 'mock' out of it. Its like a dummy image. So now we will create a mock image (dummy image).

Let us create a mock. So we will create a mock folder on the root foolder (inside src folder). In our 'mocks' folder, we will create a dummy image. But it doesn't understand png, it understand js.

So we create dummy image, file name will be 'dummyLogo.js'. and because it is javascript we will export something from this.

Let us export dummy.png image, like this




```
export default "dummy.png";
```

Now, How our code will know that we will use 'dummy.png'. Now, here 'jest' will help us.

In the jest.config.js file, we will add configuration. So if you will see in this file, there is something known as moduleNameMapper: {}, (we have to find this function inside 'jest.config.js'). It will be on 91 Line No.

Now, very carefully read :—

'ModuleNameMapper: {}' is the place where we can tell 'jest' that all the 'png' image. take it from the dummy image. with-

Whenever we see png image any time in our app, just replace with the dummy image, we have created right now.

We will tell all my .png images over here, whatever you see, please use our dummy images like this ↓

jest.config.js

```
moduleNameMapper: {
  "\\.(jpg|png|svg)$": "../mocks/dummyLogo.js",
},
```

We have added jpg or svg type image also.

So, Now if we test it, npm run test command, we will again get a error, Oh my God!!!, again error.

This error is saying

Could not find react-redux context value; please ensure the component is wrapped in a <Provider>

It means, it is saying that in your header does not have a provider. Because in our 'Header.js' file, we are using 'useSelector()'.

See, where is our code is running?, our code is running inside 'jsdom' but it does not have a provider. That's why it is giving a error.

Let's create an actual store. So, whenever we will render our 'header' in Header.test.js file, we will provide a provider and Provider will come from react-redux library. 'Provider' also needs something that is store and this store will come from "../../utils/store".

Now, Lets see example :—

Header.test.js

```
import { Provider } from "react-redux";
import store from "../../utils/store";

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

    const header = render(
        <Provider store={store}>
            <Header />
        </Provider>
    );

});
```

Now, lets run again using command npm run test but after running again, it failed once again, (Oh my God). it given us error once again. let's see, what is the error,

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

Remember, we have something use, know as Router Provider, we have used 'Link' in Header.js file. Our 'jsdom' does not understand 'Link', it does not know where from the routing comes.

So, we have to give 'jsdom' router also. We will import 'staticRouter' which comes from the 'react-router-dom/server'. This router can work without browser.

Now, we will provider this 'staticRouter' to this app, we will just wrap all things inside our 'staticRouter'. Let's see example :—

Header.test.js →

```
import { StaticRouter } from "react-router-dom/server";
.....
.....
const header = render (
  <StaticRouter>
    <Provider store={store}>
      <Header />
    </Provider>
  </StaticRouter>
);
```

Now, let us run once again npm run test (finger crossed) and after running this command, it passed... finally it passed. Thank God! ☺

But But... but we have only pass the test case yet, we have just rendered, we have not expected something. Let us now expect something, Let's us check if the logo is loaded or not. Before starting, we can also check what is inside our header variable by doing console.log(header)

→ Now, Let us find Logo inside our header. We can find logo by various ways like 'document.getElementById' or we can do 'getElementbyTag' name.

So similarly, over here we will do something known as 'getALLByTestId'. It is best practice. It is our test id for uniqueness. We have not write the test id during development. Let's write test id for our 'Logo' in Header.js file in our img tag, such as

~~Header.test~~ Header.js

```
const Title = () => {
  <a href="/" >
    <img data-testid="logo" alt="logo" src={logo} />
  </a>
);
```

This is our test id

: Syntax

Remember always :—

→ id = " " → This is recognizes by our browser

→ data-testid = " " → This is recognizes by jest

Now we will try to find our logo by this (above test id) by writing this

header.test.js

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

↳ test id

Let's expect something by writing this below the above code.

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

Now if we run command npm run test, it will give an error which is undefined. It means, it is not able to find 'src' from above code. Because if we do console.log (logo) it will show an array and here above we are passing array, that's why it is showing error.
So we have to do like this ↓

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

Again if we run our command, it will failed again because we have not written the full url, like this.

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

Now if we run, npm run test, it will pass, test case will pass ☺

→ Now, let us try a new test case for 'online'
How we will write for 'online' a test case? think...

- What will be the first step we have to do.

We will add a test id, so that we can find out.
We will find out with the test id whether it is 'green' or 'red'.

Let us put test id—

Header.js

```
<h1 data-testid="online-status">
    {isOnline ? "green" : "red"}
</h1>
```

So we will write like this in Header.test.js file.

```
→ const onlineStatus = header.getAllByTestId("online-status");
→ expect(onlineStatus.innerHTML).toBe("green");
```

But here we have mistake something i.e. 'getAllByTestId', always written an array. So for this we have to written like (onlineStatus[0].innerHTML)

But we can use this also → 'getByTestId', in this case we ~~have~~ not ~~necessary~~ necessary to write (onlineStatus[0].)

Let see code :— Header.test.js

```
test("Online status should be green on rendering header", ()=>{
    const header = render(
      <StaticRouter>
        <Provider store={store}>
          <Header />
        </Provider>
      </StaticRouter>
    );
→   const onlineStatus = header.getByTestId("online-status");
→   expect(onlineStatus.innerHTML).toBe("green");
});
```

So Now, if you own test command, this test case will be passed  <Hurrraaayyy!!! />

→ Now, test our third test case, let us try new test case for cart have zero ("0") items. Same process, just we have done previous in all case.

↦ First we will add a test id, so that we can find out for checking whether cart has zero items our not.

Header.js

```
<span data-testid="cart">{cartItems.length}</span>
```

↳ test case id

Now let's see the code :—

```
test("Cart should have zero items on rendering header", () => {
    const header = render(
        .....
        // same as previous code
        .....
    );
    const cart = header.getByTestId("cart");
    expect(cart.innerHTML).toBe("0");
});
```

On running npm run test command, this test also will be pass.

Now, we will do Integrating testing.
Let us do integration testing for search functionality. search is a complex feature. If will do a search testing, it will cover lots of thing.

First of all, We have to load of search container, restaurent page, we will just load our body

Let's create test file name `Search.test.js`. First we will render search page. Now we will write test case, as we have written in previous test cases.

Let's us rendered our application body by importing like this

Search.test.js

→ import Body from "../Body";

But this 'Body' would not work alone. We have learnt from previous past that we should give it a 'Provider' and it will come from 'react-redux' and we will provide 'store' and import it (we have already learnt all this in previous test cases.) And we have to wrap inside 'StaticRouter' also (As we have did in previous test cases.)

Let see example: —

Search.test.js

```
test("Search results on Home page", () => {
  const body = render(
    <StaticRouter>
      <Provider store={store}>
        <Body />
      </Provider>
    </StaticRouter>
  );
});
```

Now after running, npm run test command, we will get an error (If in your Body component 'fetch' is present otherwise no error will come), Error will be

ReferenceError: fetch is not defined

Now, here 'jest' is saying that I don't know 'fetch'.

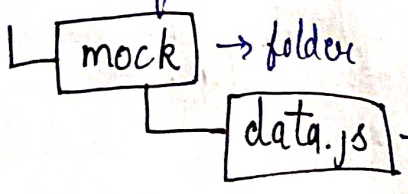
'fetch' is given by 'browser'. We will have to make our mock our API call. We are not running this code in our browser, we can't make network call in our test cases. We will use mock data to test our component.

So, we will write something known as 'global.fetch' and we will use dummy function which is given to us by 'jest' and this is known as 'jest.fn()' Basically a fetch function returns a promise. We will do a 'Promise.resolved'. We can reject the also. It means there can be do two things

- We can test for failure case or
- We can test for success case.

So, right now, we will resolve the problem. We will resolve it with 'json' data because 'fetch' returns a readable string (stream) and we convert that readable stream into a 'json'. And this 'json' again returns a **Promise** and we will do 'Promise.resolve()' in which we will pass ~~our~~ data that we have to mock.

Now, we will copy API data of the restaurent and paste it to new file named data.js (can be a another name) and this file will be inside our mock folder. Example



data.js

```
export const RESTAURENT_DATA = {
    // Paste here whole data (API)
}
```

Now, we will import this above 'data.js' in 'Search.test.js' file. We are creating our own dummy fetch. Give test-id to search component If we now, run npm test does, it will throw an error that

Unable to find an element by: [data-testid="search-btn"]

This error means, it is not able to find "search-btn" because it is firstly loading our 'shimmer'. Our app loads shimmer first, then it loads actual data.

So, it means we should first test 'shimmer' also. Give a 'test-id' to the 'shimmer'. And this will be test case for shimmer :—

Search.test.js

```
test("Shimmer should load on home page.", () => {
    const body = render(
        <StaticRouter>
            <Provider store={store}>
                <Body />
            <Provider>
        <StaticRouter>
    );
    const shimmer = body.getByTestId("shimmer");
});
```

Test id for shimmer

Everytime we have to run npm run test. and for every time it is very hectic. So we will do HMR (Hot Module Replacement) for test run. It means, we will not need to do 'npm run test' every time. just write below line Package.json file in scripts.

package.json

```
"watch-test": "jest --watch"
```

and now run npm run watch-test only for once, everytime it will run the test automatically.

Now after this, test will still complaint but this time it is complaining because of 'async-await', we are not waiting for the data to come up and this will be a warning not an error. (we have to figure out by ourself).

Now, let's expect something from the shimmer. How we will test 'shimmer'? just think!!! There is something known as 'toBeInTheDocument()', which says that it is (shimmer) there in the document or not.

Now it will say as a error that 'toBeInTheDocument is not a function. It is saying like this because this 'toBeInTheDocument' comes from another rtl library and that library is '@testing-library/jest-dom', we have to import it

search.test.js

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

Also we have to install this from the terminal, like this

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

So, yes, test case is passed.

Example:- search.test.js

```
import "@testing-library/jest-dom";
test("Shimmer should load on homepage", () => {
    const body = render(
        :
        // same as previous code
        :
    );
    const shimmer = body.getByTestId("shimmer");
    expect(shimmer).toBeInTheDocument();
});
```

But But and but this type of writing code is not a good practise We can also wrote like this in exchange of last line of above code which gives really satisfaction of passing test case ↓

```
expect(shimmer.children.length) = toBe(5);
```

"5" is because shimmer has its 5 children and this test case is surely pass. ☺

Now, let's render our restaurant page on homepage, previous was 'shimmer' Now, we have to avoid the 'shimmer'. We have to wait till our data loads.

For waiting, we have to write 'await' and 'waitFor();' this 'waitFor()' is given by 'React Testing library'. We will wait till we have 'search-btn' on our screen. And this 'waitFor' is import by "@testing-library/react"; Now because we are using 'await' we have to make our function as 'async'

Let us just grab the restaurent list. So just like we checked for '5' items in 'Shimmer', we will check for number of restaurent inside our app. Let see how many restaurents are loaded.

Now firstly we will give test id to the restaurent, like this.

Body.js

```
<div data-testid="res-list">        → test id
  {filteredRestaurent.map((restaurent) => {
      .  .  .  .  .
      .  .  .  .  .
  })}
</div>
```

Now, from above test id, we will find our restaurent list. Let's see the example :—

Search.test.js

```
test("Restaurent Page should load on home page", asyn () => {
    const body = render(
        // just like previous code
    );
→   await waitFor(() => expect(body.getByTestId("search-btn")));
→   const restList = body.getByTestId("restaurent-list")
→   expect(restlist.children.length).toBe(15);
});
```

This test case is also passed :)

→ Now. Let us find the input box, type something into it, and hit on the search-btn. Now, Let's search for the food

Firstly we will give test id to 'input' tag for searching an item in Body.js (search input tag where yours are searching the restaurent and give it to id like, 'data-testid-"search-input"

Now, we will fire an event on searching anything and there is something known as 'fireEvent' which is given by 'React Testing library'.

What is the event, we have to fire? remember we have added 'onchange()' event to that 'input' box, so that is what we need to fire.

We will do a 'change()' method with **fireEvent**,

'change()' is basically the triggering the onchange() of the input box, we have to change the 'input' and remember, for finding the value of the input box, we does 'e.target.value'. So

So basically we will create here dummy events. So we will create target and a value inside it and we passing the food inside it (by looking below code, we will understand).

We are not doing manually typing, so we have to mock that event (we will not type on the browser), our code will type for us. Now basically 'food' will be typed inside automatically.

After, we will fire an event and that is **click()** and we will see click on 'search-btn' (test-id).

In this test case, in the 'input' box, we have given the input 'food' (you will see in the code). In the expectation we will give the length of the restaurant in which 'food' string will we found. Let's see the code

Search-test.js

```
test("Search for string (food) on home page", async () => {
    const body = render(
        // same as previous test case
    );
    await waitFor(() => expect(body.getByTestId(
                        "search-btn")));
    const input = body.getByTestId("Search-input");
    fireEvent.change(input, {
        target: {
            value: "food",
        },
    });

    const searchBtn = body.getTestBy-
    const searchBtn = body.getByTestId("Search-btn");
    fireEvent.click(searchBtn);
    const restlist = body.getByTestId("rest-List");
    expect(restlist.children.length).toBe(0);
});
```

And this test case will be passed...

Let's do one more test case, for updating the menu data. First we will copy API's data of the menu data and paste into the file data.js. The data shall be in object Menu. MENU_DATA (can be another) name.

For new testing, we will create file named 'Menu.test.js' in which we will mock our menu data, We will load our 'RestaurentMenu' component. In this component we will give also test case id ie. data-testid = "menu".

We will await for our menu to load. We will also import Header component. Now, we will fire an event that will be a click event. We will also give test id to the button also of add item (i.e button) and test id will be addBtn. Now let's see the code...

Menu.test.js

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

test("Add items to cart", async () => {
    const restaurentMenu = render(
        <StaticRouter>
            <Header />
            <RestaurentMenu />
        </StaticRouter>
    );
    await waitFor(() => expect(restaurentMenu.getByTestId(
                                "menu")));
    const addBtn = restaurentMenu.getAllByTestId("addBtn");
    fireEvent.click(addBtn[0]);
    const cart = restaurentMenu.getByTestId("cart");
    expect(cart.innerHTML).toBe("1");
});
```

And that's all.....