Time For Test! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-13** first. Understanding the video will make these notes way easier to understand.

> **In this episode, we are going to learn about**
>
> - What are the testcases and how to write
> - Types and importance of testing

👉 Before we start to learn today's episode:

There are many types of testing like QA and developer. But we will learn about developer testing because it's a huge domain in itself

Part-1

Types of testing:

1. Manual testing: Testing the functionality that we have developed. E.g → we have developed a search bar, manual testing is checking the search bar manually by searching the query.

> 💡 This is not a very efficient way because we can’t test every new feature in a big application. A single line can introduce bugs in our whole app because multiple components are connected to each other.

2. Automatic testing: We can write the test cases for testing the functionality. It includes

a. *Unit testing* : Write test cases for the specific part (isolated components)

b. *Integration testing*: writing testcases for the components that are connected like menu page and cart page are connected.

c. *End-to-end testing*: writing testcases from user enters into the website to user leaves the website

Install libraries:

📢 NOTE: If you are using create-react-app or vite. please ignore these installation steps because these packages already include the testing library

1. React Testing library: It is provided by react that allows to test react components

```
npm i -D @testing-library/react
```

2. jest: React testing library uses jest so we need to install it.

```
npm i -D jest
```

Since we are using Babel as a bundler so we need to install some extra libraries.

For more details check the official website: https://jestjs.io/ and https://babeljs.io/docs/usage

3. install extra babel libraries

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

4. create babel.config.js

```
const presets = [
  [
    "@babel/preset-env",
    {
      targets: {
        edge: "17",
        firefox: "60",
        chrome: "67",
        safari: "11.1",
      },
      useBuiltIns: "usage",
      corejs: "3.6.4",
    },
  ],
];

module.exports = { presets };
```

📢 NOTE: If you are following this series, then you must know parcel is using Babel. We just added babel.config file but the parcel also has a babel configuration behind it. that creates a conflict. To solve this we have to disable the parcel config. Create a .parcel.rc file

```
// .parcel.rc
{
  "extends": "@parcel/config-default",
```

```
  "transformers": {
    "*.{js,mjs,jsx,cjs,ts,tsx}": [
      "@parcel/transformer-js",
      "@parcel/transformer-react-refresh-wrap"
    ]
  }
}
```

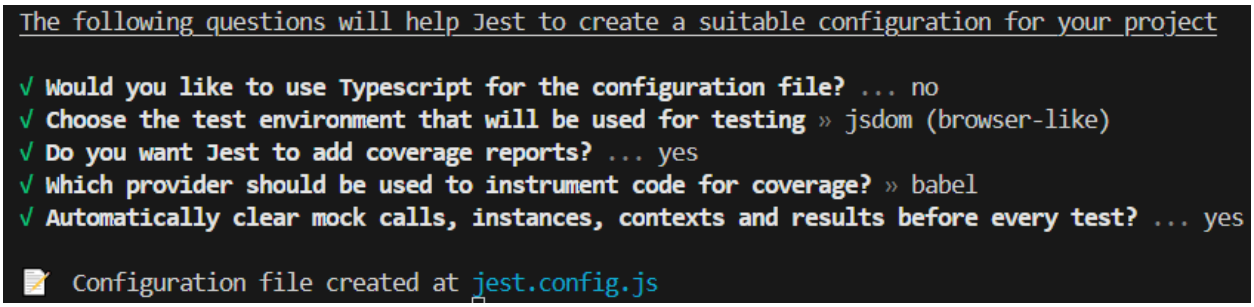
To check that you have installed successfully without an error. Try to run the testcase

`npm run test →` will give no test case found

5. Let’s configure the jest

```
npx jest --init // executing the jest package
```

After running this command, you have to select some options



💡 We are using jsdom as a test environment. When we run test cases, there is no browser or server. For running the test cases we need an environment which is jsdom

6. Install jsdom environment

```
npm install --save-dev jest-environment-jsdom
```

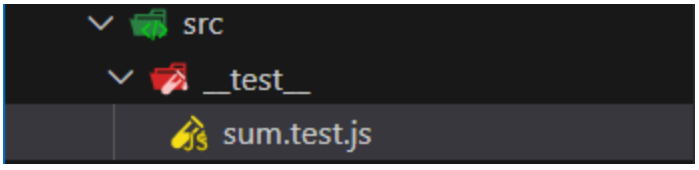
Start writing testcases:

Let’s start with writing testcases for simple function that returns the sum of two numbers

1. create a file for which we need to write the test case

```
//sum.js
export const sum  = (a,b) => {
        return a + b;
}
```

2. create a folder `__test__` and create a file in this folder `sum.test.js`



3. Let’s write out first test case

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

test("Function should calculate the sum of two numbers",() => {
```

```
        const result = sum(3,4) // call the function
        expect(result).toBe(7)  // assertion provided by jest.
}

// test will take two arg name and callback function
```

4. Run the test case

```
PS D:\Namaste_React\Namaste-food> npm run test
```

I hope now you have the overview that how testing works. So now let’s write the original test case for our project

Unit test:

Let’s write a test case to check the component is loading or not. To check any component loads or not, we need to check in the jsdom.

```
test('check component loads or not',() => {
        render(<Contactus/>   // render the component that want

    const heading = screen.getByRole("heading")  // check the sp

        expect("heading").toBeInTheDocument()   // check heading
})
```

📢 Note: If you are using parcel, you will encounter an error. The error is we haven’t enable JSX yet. So, we are not able to render the compoent that uses JSX. Let’s enable it

To enable JSX in testing environment:

a. install babel: `npm i -D @babel/preset-react`

b. set babel config:

📢 Note: Install one more library to use the dom functions `npm i -D @testing-library/jest-dom.`

Now, run the testcase. It will pass

We found the heading using “getByRole”. But we can also find using different functions like

```
const button = screen.getByText("submit") // It will find the te

expect(button).toBeInTheDocument()
```

💡 We can use `it` instead of `test` keyword

To write multiple testcases

```
describe('To test the header component',() => {

        it("Should load the header component",() => {})

        it("Should include the button",() => {})
}
```

💡 After creating testcase, git will show many files changed. We don’t need to upload the coverage folder to the github. So, add `coverage` to the `.gitignore` file

Let’s make one more test case for testing `Header` to check button is rendered or not

```
// import render and header
it("should test header compoenent",() => {
        render(<Header/>)
}
```

We will get a few Errors because we are testing the isolated component.

📢 Note: We used redux store in the header (useSelector) but the store is provided to the app component. To test the Header component, we need to provide the store to Header component as well

📢 Note: We used the Link component provided by react-router-dom. But we have provided router to the App component. So, To test the Header we have to provide the router to the Header component

```
gourav garg, 4 months ago | 1 author (gourav garg)
import { fireEvent, render, screen } from "@testing-library/react"
import Header from "../components/Header"
import { Provider } from "react-redux"  4.6k (gzipped: 1.9k)
import appStore from "../utils/appStore"
import { BrowserRouter } from "react-router-dom"  5.1k (gzipped: 2.3k)
import "@testing-library/jest-dom"

it("should render header component with a login button", () => {
    render(
        <BrowserRouter>
            <Provider store={appStore}>
                <Header/>
            </Provider>
        </BrowserRouter>
    );
    // If there are multiple buttons then can find using name showing on the button
    const LoginButton = screen.getByRole("button",{name:"Login"});
    // const LoginButton = screen.getByText("Login")
    expect(LoginButton).toBeInTheDocument();
})
```

To check the button click, you can use the *fireEvent* which behaves like an onclick method

---

Integration Testing:

💡 To run the test automatically, make a script in the package.json like “watch-test”:”jest --watch” . Now we can simple run npm run watch-test.

Let’s write the test case for the Search component. It should show the card in the body when we search some restaurant

Fetch the body component first (because search bar is in body)

```
it("should show the search button",() => {
        render(<Body/>)     // will throw an error
}
```

📢 Note: Error generated because the Body component uses the fetch function which is provided by the browser. But we are using jsdom. So we need to make a mock fetch function with mock data that will replace the original fetch function.

For creating fetch function, we need to create mock data for restaurant list. So, create a new file Mock_Data and store the data by coping the restaurant list from the api.

Create our fetch function similar to browser fetch function

```
// import mock Data
global.fetch = jest.fn(() => {
        return Promise.resolve({          //  fetch function retu
                json: () => {                 // json the promise
```

```
                        return Promise.resolve(Mock_Data)   // r
                }
        })
}
// Mock_Data will be returned in the end from the fetch functio

it("should show the search button",() => {
        render(<Body/>)    // will throw an error
}
```

Run this test, we will get the warning

> 💡 When we use the async operation, we should wrap our component inside act function. It will return a promise

Also provide the router to the component because we are using the `<Link/>`

```
import {act} from "react-dom/test-utils"
// import mock Data
global.fetch = jest.fn(() => {
        return Promise.resolve({         //  fetch function retu
                json: () => {                // json the promise
                        return Promise.resolve(Mock_Data)   // r
                }
        })
}
// Mock_Data will be returned in the end from the fetch functio

it("should show the search button",async() => {
        await act(async() => {
                render(
                    <BrowserRouter>    // because we are using L
                            <Body/>
```

```
                    </BrowserRouter>
                )
        }

        const searchBtn = screen.getByRole("Button",{name: "Sea
        expect(searchBtn).toBeInTheDocument()   // check the bu
}
```

Now, all testcase will pass successfully.

> 💡 If you don’t want to find using getByRole, We can also use getByTestId. It will always work

To use getByTestId, give the testid to the element

```
<input data-testid="searchinput" type="text"/>
```

```
const searchInput  = search.getByTestId("searchInput")
```

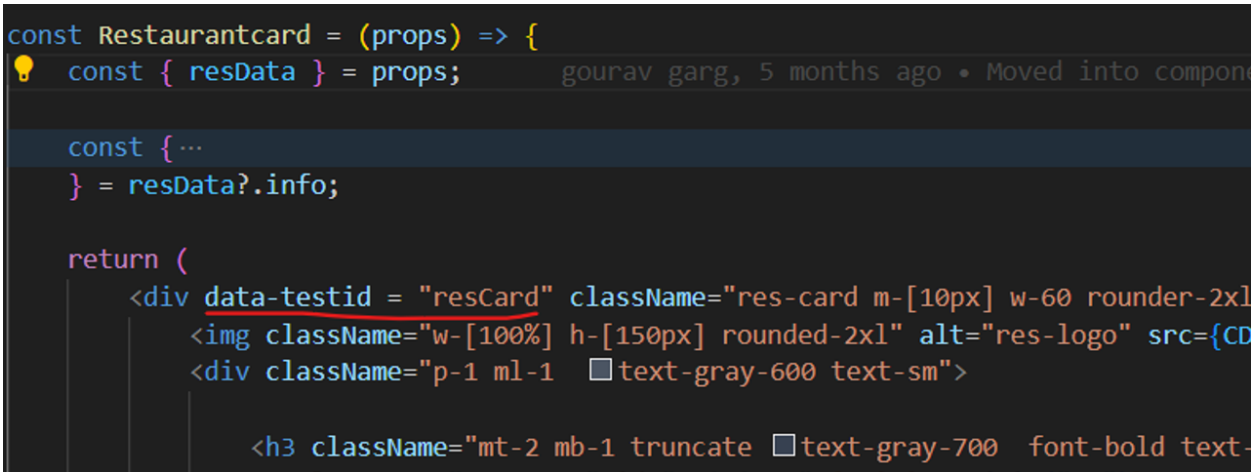
Now, test the input to get the user query to give the restaurant data.

```
const searchInput  = screen.getByTestId("searchInput")
// fireevent is provided by jest which is used to perform the ev
fireEvent.change(searchInput,{target:{value:"burger"}})    // cl

fireEvent.click(SearchButton) // click the search button
```

Now, find the restaurant card rendered on the screen after clicking the search button

💡 To get the restaurant data div, give the testid




```
const searchCards = screen.getAllByTestId("resCard")  // get the
expect(searchCards.length).toBe(2)  // expect the count that sh
```

Now, all test case will pass.

💡 If we want to write something after and before all the testcases or each testcases. jest provide functions

Final code for Integration testing

```
import { fireEvent, render, screen} from "@testing-library/react
import Body from "../components/Body"
import { BrowserRouter } from "react-router-dom"
import MockResList from "../__tests__/mocks/MockResList.json"
import { act } from "react-dom/test-utils"

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

```
        }
    })
})
// describe is used to club multiple testcases
describe("",() =>{
        // to print before all testcases
    beforeAll(() => {
        console.log("Before testcase")
    })
        // to print before each testcase
    beforeEach(() => {
        console.log("Before Each")
    })
    // to print after all testcases
    afterAll(() => {
        console.log("After testcases")
    })
    // to print after each testcase
    afterEach(() => {
        console.log("After each")
    })

    it("Should render body component with search feature",async
        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))
    })


    it("Should search ResList for burger text input",async() =>
        await act(async() =>
            render(
            <BrowserRouter>
```

```
                <Body/>
            </BrowserRouter>
        ))

        const TotalCards = screen.getAllByTestId("resCard")
        expect(TotalCards.length).toBe(18)

        const SearchButton = screen.getByRole("button",{name:"Se

        const searchInput  = screen.getByTestId("searchInput")

        fireEvent.change(searchInput,{target:{value:"burger"}})

        fireEvent.click(SearchButton)

        const searchCards = screen.getAllByTestId("resCard")

        expect(searchCards.length).toBe(2)
    })

    it("Should filter Top rated restaurant after clicking buttor

        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))

        const TotalCards = screen.getAllByTestId("resCard")
        expect(TotalCards.length).toBe(18)

        const Button = screen.getByRole("button",{name:"Top Rate

        fireEvent.click(Button)
```

```
        const searchCards = screen.getAllByTestId("resCard")

        expect(searchCards.length).toBe(6)
    })

    it("should render Username",async() => {
        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))
        const userInput = screen.getByTestId("userInput")
        expect(userInput.value).toBe("Gourav")
    })
})
```