(Test Case Setup)

Akshay Saini class

(Time for the Test Case in React)

- React Testing Library (Industry Standard).
  (uses Jest behind the scene)
- Earlier it was enzyme way (OID)

- Using enzyme is not good. Need to migrate to React-Testing-Library.

→ We will learn how to set up Testing Framework.

(npx create-react-app app-name)
automatically does thing for us. But we did the setup from our own. Integrated Parcel. Bable, Browserlist.
Today we will setup react testing framework right from scratch.

Why do we need test cases?

✓ When we develop an app we check the performance, maintainability etc.
✓ But for large scale application - it is having 100+ component, dependency.

✓ component are integrated with each other.
Ex: Search component (input box) affecting lot of component inside it.
✓ All are interrelated to each other.
✓ It is possible than, ~~we have~~ Someone did Restrolist component. Someone else did Search component. Somebody did 'Cart' component.

Scanned with ACE Scanner

- If we add new code, it should not break existing code. This is the fundamental of writing unit Test Case.

- Suppose there are 1000 component and we add 1 more, we should not break other component. It should be maintainable code. Unit Test gives confidence.

→ (Test Driven Development)

We write test cases even before we write our code.

- It is very good.
- If we do so, even before writing code, test case gives 100% test coverage.
- But development process becomes slow. But it ensures very good ~~com~~ code quality.

Example of Sum of two numbers: :→

```
test("Check Sum of 2 Numbers", () => {
    expect(Sum(2,3).toBe(5);
    expect(Sum(-2,3).toBe(1);
    expect(Sum().toBe("Plz check argument");
});
const sum = (2,3);
```

Write all test case to make sure sum function is ~~Expect~~ extremely efficient then write sum function.

(Jest) Diff. type of Testing:

1) // Manual Testing (Human goes into the page and check)
   - ✓ (Requires human)
   - ✓ (Not efficient)
   - ✓ Testing person will try to break the code.
   - ✓ It somebody's job.

2) Automated Testing:

- ✓ Code testing a code.
- ✓ No human involved. less error.
  - ~~Selenimum~~ (Selenium)

(3) End to End Testing:

Test the whole flow.
It will simulate, user login + Search + add item + cart + order + → Whole flow is tested.

✓ ~~It~~ Test case covers entire user journey.

✓ This part is given to QA team.

✓ (Headless Browser:) It is a kind of like a actual browser but it does not show anything. No view port involved.

✓ Testing will happen in Server.

✓ open headless browser → login → fill data → Flow → . Replacing manual testing with code.

✓ Headless Browser there will be virtual DOM everything. But browser won't push it anything. No view port. QA Job.

Scanned with ACE Scanner

4) Unit Testing: ✓ Core Job of Developers.
✓ Test when my app opens, is my logo coming properly or not.
(Dev)

✓ As a developer - it's important.
✓ Daily job of Developer.

5) Integration Testing: How do you testing the integration between component.
(Dev)

✓ If we search something and click the URL should change.

6) Regressinon Testing.
7) Smoke Testing.

(lot more)

✓ Testing exactly same like Development. It takes time. You should know it.

Let's Start diving into Testing.

- React Testing Library. (Replacement of enzyme).
- Testing Library offer testing for React
  Angular
  Vue etc.
- (JEST) - Delightful javascript testing framework with a focus on simplicity.

React Testing Library uses (JEST) behind the scene.

Scanned with ACE Scanner

React Testing library makes it efficient.

Step

① Install it in React. (React testing library + jest)

npm i --save-dev @testing/library/react

npm i -d jest

Package.json:

jest: "^29.4.1";

testing/library/react: "^13.8.1"

② (Configure jest):

I will create a jest.config file (jest --init) comman.

use npx (just execute it once) (so we should use npx)

(npm vs npx)

↳ (once I will initialize jest with config only once)

Steps: npx jest --init + Set options

1) → Typescript - N

2) → Test Environment - jsdom (browser-like)

3) → Add coverage Report Y

4) → Which provider to use → babel

5) → Automatically clear mock → Yes (garbage collection)

⑥ ↳ It will create a jest.config.js

(collectCoverage: true) etc will come.

Install jsdom +

jsdom Creation

(3) How we will run test case?

package.json

"script": {

"start": "parcel index.html"

test : "jest"

build: "parcel build index.html"

}

npm run test

after (jest 28) we need install jest-env jsdom

npm -i -d jest-environment-jsdom

(new entry in package.json)

npm run test Run again → Worked.

It checks --test folder. It's not there.

~~te~~

components

↳ --test-- folder creation under components.

- Whatever files I will write here, jest will consider it.
- Dandar (Danda test). (Google it)

(4) Create first test file case (jest will track only when it is just in the folder or pattern .test|.spec.js)

--test--

↳ sum.test.js (always write .test.js)

(some write .spec.js)

First Test case.

↙ we have Sum.js

import {Sum} from "../Sum";

Sum.test.js // Sum of 2 function + Run it to pass

test("Check Sum of 2 positive No", () => {
// function will be executed
// We have to fill our code.
// every test case should have an expectation
// we call it assertion

expect(Sum(2,5).toBe(~~5~~); 7.
// make a sum call to return '5' for example
// Sum will come from Sum file. (actual code)
// Run test // npm Run test
// one test case failed.
// Failed to pass the file.
// cannot use import statement outside module
// Normal js file won't understand import.
// jest should under 'import'.
// We will need to take help of babel in this.
// Need to add (-ve) number and other case arrived
// Modify sum function accordingly
}

5 Configure babel :

jest need babel

npm-i-d-save babel-jest @babel/core
@babel/preset-env

// Babel will make jest understand that 'import'
// We are using es6. 1 js wont understand import.
// Babel does it.

Scanned with ACE Scanner

component.spec →

sum.js
export const sum = (a,b) => a+b

classmate
Date
Page

⑥ (Bable.config.js) We need.
// We need to make our code understand

→ we have .bablerc
// 2 ways of configure – (1) .bablerc
(2) bable.config.js

✓ We used (.bablerc) to remove console.log.

✓ We will write here
(.bablerc) // (get it from offical doc of jest)

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

// JSON vs JavaScript Object difference:
// (JSON is diff than Javascript object) (Homework)

// adding " " convert it to JSON

npm run test failed.
(expected 5 recived 7)
(function gave me 7) We wrote test case wrong.

npm run test: all test case sucess.

(Where these testing happen?)

① We are not Running my App
npm run test
→ are these test running on browser? (NO)

→ It's run on a diff. environment in jest. We can't see in browser.
→ Browser would have understand import.
testEnvironment : "jsdom" // It's not browser but some browser capability.

jsdom - mini browser

- We need to tell our code that something.
- 'import' → we take help of babel to make jest understand 'import'.

- Babel will help jest to work with understanding.
- Babel ~~will~~ is a transpiler at the end.

- We got coverage Report also (collectCoverage: true) jest.config.ts

(coverage) Folder gives coverage Report.
- Coverage Report tells you how much code/test case has been covered.

(Sum.js → 100%)

(add coverage folder under gitignore)

7 [(gitignore) include it gitignore)
/coverage . parcel-cache]

8 Write basic Unit Test case.
(as soon as page load 'header').

- Load 'Header' and see.
- When I load header → Logo should load.
  → My cart item should be '0'.
  → Defaut Status "online".

a Create one more file

header.test.js

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

header.test.js

```
test("Logo should load on rendering header", () => {
    // Name should be descriptive
    // Test case Name should explain it all

    //load Header   //Check if logo is loaded
//this code is running under jsdom. i.e small
  machine to check.
// Render Header component
// const root = ReactDOM.createRoot(pickup)
                    document.getElementbyId("root");
// root.render
    (<RouterProvider />).

// Now we will use render under the testing environment.

const header = render (<Header />);
    console.log(header);
//Running test on jsdom. It's don't have 'root'.
//React does everything ~~to~~ within 'root'.
// JSDOM we just want to load Header.
// Render Function from JSDOM gives feature
  to load my component on it.
```

npm run test - failed. Syntax error. Jest does not understand JSX.

Again we need Bable.

Install. Need one more package. .babelrc

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

//We need it for JSX understanding.

npm run test. Failed again.

Scanned with ACE Scanner

It has started Reading header file.
import Logo fm "..... " ;

// It is trying to read PNG image as javaScript (JSDOM)
// Need to figure out workaround.
// Create a mock for it. Create a Dummy image for us

mocks folder
└ dummyLogo.js
export default "dummy.png"

// How will my code get to Know that we are using dummy png?

// jest will help
// all the PNG image
take it from dummy image

// (jest.config.ts)

moduleNameMapper : {
"\\.png" : "../mocks/dummyLogo.js"
(all my.png images) },
Please use my dummy image

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

(npm Run test → Failed)
→ react-redux exact value not getting.
→ what I will do?
→ I will add a provider in test case.

header.test.js

```
import { ~~CreateBrowserRouter~~ StaticRouter } from 'react-router-dom/server'
import { provider } from 'react-redux';
import { store } from "../utils/store"
```

Section A

```
const header = render (<StaticRouter><provider store={store}>
                        <Header />
                        </provider></StaticRouter>
                      );
```

//
// Router we used (Link) JSDOM won't understand it.

// Static Router will work without browser.

// How to setup all babel/preset/jest/babelrc - we have learnt.

header.test.js

→ All imports

→ test("Logo should Render", () => {

→ Section A Now we need to expect.

✓ We are getting virtual DOM object now.
✓ We are getting FiberNode // we can play with it.

→ const logo = header.getAllByTestId ("logo");
console.log(logo); // npm run test
// we got HTMLImageElement
src : 'dummy.png'

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

Header.js

<img data-testid="logo" />
add this id for testing.

const logo = header.getAllByTestId("logo");
console.log(logo[0]);
expect(logo[0].src).toBe('dummyLogo.png");

.toBe("http://localhost/dummy
(Full URL Needed).

// Testing is equally Development
// Testing is not as easy as we think of.
// It has huge learning curve.

npm run test ⟹ passed Finally.

⟹ Coverage Report comes.

✓ Now → Logo present ✓ Done
✓ DO online ✓

Header.test.js

header.js
<h1 data-test-id="online-status">

✓ Add a test id

test("online Status should be green", () => {

const onlineStatus = header.getAllByTestId("online-status")
getByTestId
(will get first id)

expect(onlineStatus.innerHTML).toBe('✅');

//Logo testing Dummy Mock.

// cart should have '0' items
// testing functionality of ux-selector also.
//It's testing hook also
~~export~~

test('cart should be '0' loading', () => {

+ Section A

const cart = header.getByTestId("cart");
expect(cart.innerHTML).toBe("0");
'Cart-0-items'

});

(npm run test ⇒ Passed)

// Magical Think for debugging a lot:
//What is rendered in the container (jsdom),
How can we see?
make a mistake "cart1"); and Run test case in console we can see the DOM rendered.

// Header component testing (Done)

Search Testing. (Integration of diff. component).
(Homework Footer Testing)

//Render search page

Scanned with ACE Scanner

```
import {Store} from "../util/Store.js";
import {Provider} from 'react-redux';
import {render} from '@testing-library/react'
import Body "../Body";
import {StaticRouter} from 'react-router-dom/server';
test("SearchResult on Homepage", () => {
//render our app's Body
const body = render(
<StaticRouter>
<Provider Store={Store}>

<Body />
</Provider>
</StaticRouter>
);
```

// fetch is not understandable.
// Fetch is browser thing
// We don't have to make API call to get data
// Mock our fetch

(6) import RESTAURENT_DATA from "./data" mock

```
global.fetch = jest.fn() => {// mock
return promise.resolve({
    json: promise.resolve(
        // mock data layer
        RESTAURENT_DATA

    )})

});
```

// code will understand what's happening inside fetch.

// creating a dummy fulbon fetch

// Creating a dummy fetch

// Fetch Returns a promise.
Test case for (Resolve) and (Reject)

→ Fetch Returns a readable stream and we convert it to json. (data.json())

✓ Create a dummy file. (data.js) under ~~Test~~ → mocks folder

(7) export const RESTAURENT_DATA = {
// Paste the API Response.

```
global.fetch = jest.fn () => {
return promise.resolve ({
        json : () => {
            return promise.resolve (RESTAURANT)
            }
        });
```

Body.ts

add

```
const Search Btn = body.getByTestId("search-btn");
console.log ($search-btn);
// our App First Load Shimmer
```

```
<button
data-testid
= "search-btn"
```

```
test("Shimmer should load on Home Page", () => {
```

See A

```
const Shimmer = body.getByTestId('Shimmer');
expect(Shimmer).toBeInTheDocument();
```

↳ comes from @testing-library/jest-dom.

```
});
```

Shimmer.js

```
data-testid = "Shimmer"
```

```
// Package.json                    // Add Hot Module Replacement for test.
    scripts:
        "watch-test" : "jest --watch"
    }
```

```
// Shimmer has Shimmer card.
   Find Children of Shimmer.
expect (Shimmer.Children.length) ~~.toBeInTheDocument~~ .toBe(10);
                                    // checking 10 card loaded.
```

test ('Restaurent Should load on homepage', async () => {

await waitFor (() => expect(~~screen~~ body. getByTestId ('search-btn'))

// const Shimmer = body. get

const restList = body. getByTestId ('rest-list');
expect (restList. ~~cons~~ children . length). toBe (15);

// We have tested reconsiliation for every thing.

// Search for 'Food' [Food] Search

test ('Search for String (Food)', ~~()~~ async () => {

[Set A]

}

Search is data-testid = 'search-input'

→ const input = body. getByTestId ('Search-input');
fireEvent. change (input, {target: {

value : "food",

// Mocking the event }});
// I am not typing
// Food will be typed for us.
const SearchBtn = body. getByTestId ('Search-btn');
fireEvent. click (SearchBtn);

const restList = body. getByTestId ('rest-list');
expect (restLis. length) . toBe (1);

// click on Add, card should update.

(Menu.test.js)

test ("update cart on Add items to cart", asyc () => {

< Header >

< RestaurentCard />

);

Rest menu.js

<ul data-testid

= "menu"

// await for menu to load.

fireEvent. and check (cart-0-item).

// Weloaded Header./ Rest Menu.

// clicked Add Button on item

// cart should be updated

The massive Session from Akshay.

Scanned with ACE Scanner