

Chapter #13 - React – Testing

When we make any code changes, it may introduce bugs anywhere within the application. Even a single line of code can mess up everything, that's where React Testing comes in.

Why Testing?

Testing helps catches bugs early. Testing helps in maintaining code quality.

Testing improves developer's confidence by ensuring that he is not breaking the existing code functionality.

Types of Testing

- * Unit Testing (developer)
- * Integration Testing (developer)
- * End to End Testing (developer or tester)
- * Manual Testing (developer or tester)
- * Automated Testing (developer or tester)

Unit Testing

In unit testing we test our react component in isolation. We take a specific react component and test that component in isolation without worrying about other components.

Integration Testing

In integration testing we test integration of components. In integration testing inter related components are tested. Let's say we have a **restaurant search textbox** and a **search** button in header component. If we type anything in the search textbox and click on search button we expect to see list of restaurants in body component. In this case, two components such as header component and body component are collaborated each other to perform search functionality. This is where Integration testing comes in.

End to End Testing

Testing a react application as soon as user lands on website to user leaves the website is End to End Testing. In End to End testing developers test the application functionalities and code flow from the beginning to end thoroughly. Cypress, selenium are the tools used to perform end to end testing.

Manual Testing

In manual Testing, developers or testers test the application functionality in screen. This is a very time-consuming process and prone to human errors.

Automated Testing -

In automated Testing, **code tests the code** instead of human testing the code functionality in screen. In Unit & Integration testing we developer write code in test files and by running these test files, application functionalities are tested, Behind the scenes the code written in test cases tests application code.

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**, enzyme and library such as **RTL** React testing library. The purpose of writing test cases is to catch bugs early in the development process, validate the changes made in a component without breaking the existing functionality.

What is TDD (Test Driven Development)?

In TDD we write test cases even before writing the actual code which means we will have 100% test coverage before writing actual code.

Testing Tools?

Cypress, Selenium, Headless browser ([Explore Online](#))

Other Types of testing

Functional Testing, System Testing, Smoke Testing, Regression Testing, Security Testing ([Explore Online](#))

React Testing Library (RTL)

RTL is one of the most standard libraries used for writing test cases in react.

RTL is built on top of **DOM Testing Library**. (**DTL**)

RTL is a wrapper around DTL. DTL is one of the subsets of **Testing Library**.

JEST

JEST is delightful JavaScript Testing Framework.

RTL uses jest behind the scenes.

When we execute the command **create-react-app** and create a project we can write test cases right from the start without configuring & installing test libraries and frameworks. Because RTL & Jest are already shipped with create react app. But in our case, we have to configure from the very start because we have created our application from scratch without relying on create-react-app.

Configuring RTL and JEST in our Application

1. Install RTL using cmd `npm i -D @testing-library/react`
2. Install Jest using cmd `npm i -D jest`
3. Install Babel dependencies using cmd `npm install -D babel-jest @babel/core @babel/preset-env`
4. Configure babel with current node version and to do that create a file `babel.config.js` in project root level and paste below code there.

```
module.exports = {presets: [['@babel/preset-env', {targets: {node: 'current'}}]],};
```

5. configure parcel config file to disable default babel transpilation. Up to this point Parcel bundler has been using babel. Parcel has its own babel configuration and now jest is using babel as per Jest configuration. Jest-babel configuration is overriding parcel-babel configuration. This will create a conflict between Jest and Parcel. We will have to change Parcel behavior to use babel along with jest and to do that create `.parcelrc` under root directory with following configuration.

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

6. Configure Jest using cmd `npx jest --init`
7. If jest version is > 28 Install JS DOM library using cmd `npm i -D jest-environment-jsdom`
8. Install `@babel/preset-react` using `npm i -D @babel/preset-react` to make JSX work in test cases.
9. Include `@babel/preset-react` inside babel config.

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

`@babel/preset-react` is helping our testing library to convert JSX code to html so that it will read properly

10. Install `@testing-library/jest-dom` using `npm i -D @testing-library/jest-dom`
11. **Optional** - Jest code Intellisense - `npm install @types/jest`

We have successfully configured RTL, JEST, babel & parcel in our project

What is JS DOM?

When we run react application they need a run time environment where the code gets executed. This runtime environment is given by browser. In browser Dom react component gets rendered.

Similarly, when we run test cases, they need an environment for execution. This Runtime environment is given by JS DOM. JS DOM is not a browser, but it like a browser which gives features of browser. We render react components in JS DOM in order to test those components.

At this moment when we execute `npm run test` we get a message in the terminal saying No tests found. This means JEST tries to search across the (regEx) 29 files (number may vary) and did not find test cases written anywhere. Jest also tried searching testcases written in files inside `__tests__` (dunder test folder) but could not find them.

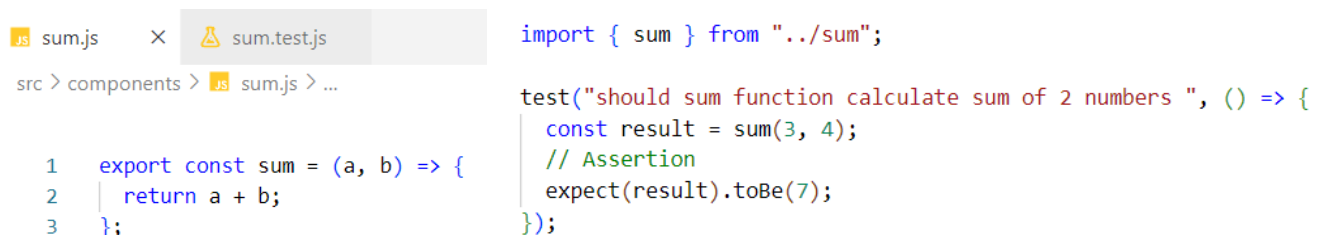
```
PS D:\REACT\My Workspace\Namaste-React-Chapter 13 - React-Testing> npm run test

> namaste-react-chapter-2@1.0.0 test
> jest

No tests found, exiting with code 1
Run with `--passWithNoTests` to exit with code 0
In D:\REACT\My Workspace\Namaste-React-Chapter 13 - React-Testing
29 files checked.
testMatch: **/__tests__/**/*.[jt]s?(x), **/?(*.)+(spec|test).[jt]s?(x) - 0 matches
testPathIgnorePatterns: \\node_modules\\ - 29 matches
testRegex: - 0 matches
Pattern: - 0 matches
```

Let's create our first test case for testing a JavaScript addition logic.

Unit testing JavaScript addition Logic -



```
sum.js  X  sum.test.js
src > components > sum.js > ...

1  export const sum = (a, b) => {
2    |   return a + b;
3  };

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

test("should sum function calculate sum of 2 numbers ", () => {
  const result = sum(3, 4);
  // Assertion
  expect(result).toBe(7);
});
```

Test case creation syntax: `test(<description>, callback)`

Assertion syntax: `expect(<actualValue>).toBe(<expectedValue>)` where actualValue is function returned Value and expectedValue is user given value. This syntax may vary depending on use cases.

We have created a test case for sum. Let's execute this test case using `npm run test`.

✓ should sum function calculate sum of 2 numbers (2 ms)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
• sum.js	100	100	100	100	
Test Suites: 1 passed, 1 total					
Tests: 1 passed, 1 total					
Snapshots: 0 total					
Time: 3.326 s					

The above tabular representation is coverage report which demonstrates the percentage of test cases executed. Test suites represents Number of modules or Test files where test cases are written. Tests represents Total number of test cases across all the modules

The above test case will fail if the expected value is other than 7.

⊗ PS D:\REACT\My Workspace\Namaste-React-Chapter 13 - React-Testing> npm run test

```
> namaste-react-chapter-2@1.0.0 test
> jest
```

FAIL src/components/__tests__/sum.test.js

✗ should sum function calculate sum of 2 numbers (4 ms)

• should sum function calculate sum of 2 numbers

expect(received).toBe(expected) // Object.is equality

Expected: 8
Received: 7

```
4 |   const result = sum(3, 4);
5 |   // Assertion
> 6 |   expect(result).toBe(8);
    |                     ^
7 | });
8 |
```

at Object.toBe (src/components/__tests__/sum.test.js:6:18)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
sum.js	100	100	100	100	
Test Suites: 1 failed, 1 total					
Tests: 1 failed, 1 total					
Snapshots: 0 total					
Time: 1.486 s, estimated 2 s					

Received value is function returned value whereas Expected value is user defined value.

Assertion is not mandatory. Without assertion test case get passed on execution.

Note - Naming of a test file - <FileName>.test.<ext> or <FileName>.spec.<ext>

Unit testing Contact us react component –

Test case 1 – (checking whether header element is present or not)

```
contact.test.js Contact.js X
src > components > JS Contact.js > Contact
1  const Contact = () => {
2    return (
3      <>
4        <div className="font-bold text-3xl p-4 m-4">
5          <h1>Contact us page</h1>
6        </div>
7        <form>
8          <input
9            type="text"
10           className="border border-black p-2 m-2"
11           placeholder="name"
12         />
13         <input
14           type="text"
15           className="border border-black p-2 m-2"
16           placeholder="message"
17         />
18         <button className="border border-black p-2 m-2 bg-gray-200 rounded-lg">
19           Submit
20         </button>
21       </form>
22     </>
23   );
24 };
25
26 export default Contact;
```

```
contact.test.js X JS Contact.js
src > components > _tests_ > contact.test.js > ...
1  import { render, screen } from "@testing-library/react";
2  import Contact from "../Contact";
3  import "@testing-library/jest-dom";
4
5  test("should load contact us component", () => {
6    render(<Contact />);
7    const heading = screen.getByRole("heading");
8    expect(heading).toBeInTheDocument();
9  });
```

Code Explanation -

Line #6 – render () renders the contact component in JS-DOM.

Line #7 - screen is the object given by RTL which stores the JSDOM loaded component. Using screen object, we can get access to individual component elements. getByRole gets elements based on their Role.

Line #8 - toBeInTheDocument() represents whether the component element is present inside the JS-DOM or not. expect is the method given by JEST framework which does assertion operation.

Test case 2 – (checking whether button element is present or not)

```
test("should load button inside contact us component", () => {  
  render(<Contact />);  
  const button = screen.getByRole("button");  
  expect(button).toBeInTheDocument()  
});
```

Test case 3 – (checking whether button element with name submit is present or not)

```
test("should load button with name submit inside contact us component", () => {  
  render(<Contact />);  
  const btnName = screen.getByText("Submit"); // case sensitive  
  expect(btnName).toBeInTheDocument()  
});  
  
test("should load button with name submit inside contact us component", () => {  
  render(<Contact />);  
  const btnName = screen.getByText(/submit/i); // case in-sensitive  
  expect(btnName).toBeInTheDocument()  
});
```

Test case 4 – Checking whether input element with a specific placeholder name exists or not.

```
test("should load input text with placeholder name", () => {  
  render(<Contact />);  
  const inputbyPlaceholder = screen.getByPlaceholderText("name");  
  expect(inputbyPlaceholder).toBeInTheDocument()  
});
```

Test case 5– Checking whether multiple input elements exist or not.

```
test("should load 2 input text input elements inside contact us component", () => {  
  // rendering  
  render(<Contact />);  
  // Querying  
  const inputelements = screen.getAllByRole("textbox")  
  console.log(inputelements) // logs array of JS objects and these objects are React Elements.  
  // Assertion  
  expect(inputelements.length).toBe(2)  
});
```

Notes -

npm run test <testFileName> executes a specific test File test cases

npm run test executes all test cases across all test files

When we have lots of test cases in a single test File its difficult to manage all these test cases, so we can create small group of test cases.

We use `describe()` for grouping test cases

Syntax – `describe(<groupDescription>, callback)`

With in the callback function body we write number of test cases for which we are creating the group.

```
contact.test.js X Contact.js
src > components > __tests__ > contact.test.js > ...
 1  import { render, screen } from "@testing-library/react";
 2  import Contact from "../Contact";
 3  import "@testing-library/jest-dom";
 4
 5  describe("contactus component test cases", () => {
 6 >   test("should load contact us component", () => { ...
10   });
11
12 >   test("should load button inside contact us component", () => { ...
16   });
17
18 >   test("should load button with name submit inside contact us component", () => { ...
22   });
23
24 >   test("should load input text with placeholder name", () => { ...
28   });
29
30 >   test("should load 2 input text input elements inside contact us component", () => { ...
37   });
38  });
```

Important points -

- * We can have described inside describe to create sub groups.
- * While writing test cases instead of **test** we can also use **it**.it is alias of **test**

What is Test Coverage?

Test Coverage is a technique which determines whether the test cases are actually covering the application code and how much code is exercised when we run the test cases. If there are 10 requirements and 100 test cases are created and if 90 test cases are executed then test coverage is 90%. Below is an example of test coverage report

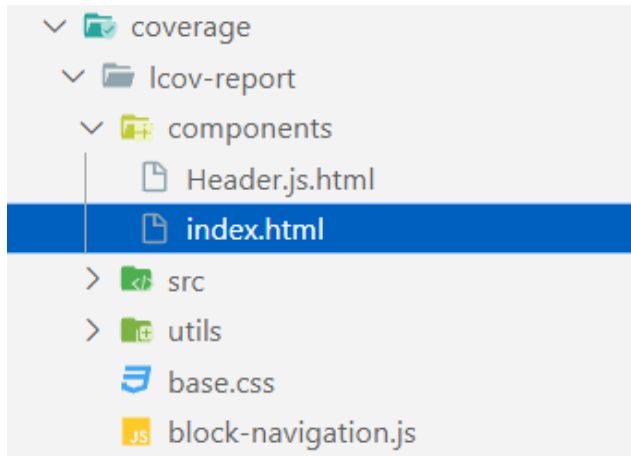
- PS D:\REACT\My Workspace\Namaste-React-Chapter 13 - React-Testing> `npm run test`

```
> namaste-react-chapter-2@1.0.0 test
> jest
```

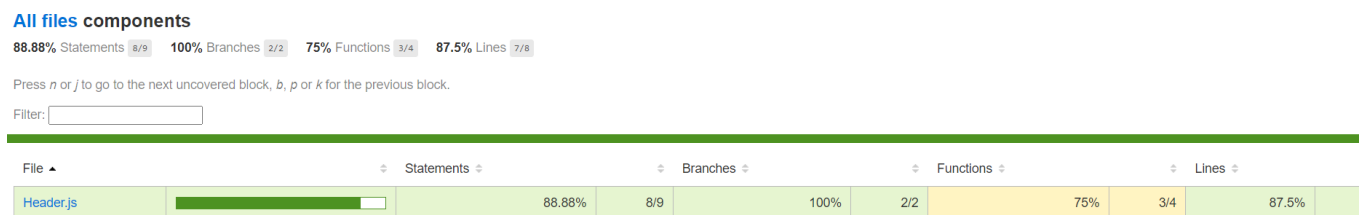
PASS src/components/__tests__/sum.test.js					
PASS src/components/__tests__/contact.test.js					
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
Contact.js	100	100	100	100	
sum.js	100	100	100	100	

```
Test Suites: 2 passed, 2 total
Tests:       6 passed, 6 total
Snapshots:  0 total
Time:        3.291 s
Ran all test suites.
```


When we execute the command `npm run test` it also creates a coverage folder which will have test coverage related information useful for developers to identify the coverage percentage. We don't need this folder to deploy into GIT. That's why we put this folder in `gitignore`.



Under Coverage Folder we will have `index.html`. If we open this file in browser it will give a detailed summary of the test coverage percentage. Also, it will give what test cases need to be written where



Unit testing Header react component

JS DOM understand JSX React code JavaScript code but does not understand `redux` code. In header component we have used `redux` functionalities. while writing test cases we have to provide `redux` store to the header component in test file.

JS DOM also does not understand React Router DOM features. While writing test cases we need to provide router to the header component in test file.

Test case 1 – To check whether login button exist in the component or not.

```
JS Header.js  header.test.js X
src > components > __tests__ > header.test.js > ...
8 test("should load header component with a login button", () => {
9   // render
10  render(
11    <BrowserRouter>
12      <Provider store={store}>
13        <Header />
14      </Provider>
15    </BrowserRouter>
16  );
17
18  // const logInButton = screen.getByRole("button")
19  // const logInButton = screen.getByText("Log in")
20  const logInButton = screen.getByRole("button", { name: "Log in" });
21  expect(logInButton).toBeInTheDocument();
22  });
```

Test case 2 – To check whether Cart-0 items exist in the component or not.

```
test("should load header component with Cart-0 items", () => {
  // render
  > render(...)
  );
  const cartItems = screen.getByText("Cart-0 items");
  expect(cartItems).toBeInTheDocument();
});
```

Test case 3 – Login button should change into Logout on button click

```
test("should change Log in button to log out on click", () => {
  // render
  render(
    <BrowserRouter>
      <Provider store={store}>
        <Header />
      </Provider>
    </BrowserRouter>
  );

  // get the Login button
  const logInButton = screen.getByRole("button", { name: "Log in" });

  // fire the click event of Login button
  fireEvent.click(logInButton)

  // get the Logout button
  const logOutButton = screen.getByRole("button", { name: "Log out" });

  // Assertion
  expect(logOutButton).toBeInTheDocument();
});
```

Integration testing of a react component – (search functionality)

Browser has a powerful API `fetch` which is responsible for fetching the data from the server. But `fetch` always returns a promise which gets resolved when json data is available. This is not the actual data that we want, this is a json data which returns another promise and the promise gets resolved when the actual data is available. This is how `fetch` returns the data.

But our testing code, runs inside JS-DOM and JS-DOM is not a browser. JS DOM does not understand `fetch`. So, we will have to write our own `fetch` implementation. We will have to mock our own `fetch` function, whose behavior will be similar to the browser `fetch` function. This is called mocking a Function.

Our mock function returns a promise which gets resolved when json data is available. This json data returns another promise which gets resolved when the actual data is available.

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

This returns a promise which gets resolved when a special JavaScript object is available. Below is that special JS object.

```
{  
  json: () => {  
    return Promise.resolve(resListMockData);  
  },  
}
```

This JS object has a property `json` which is a method and it returns another promise that gets resolved when the actual data is available. Ultimately, we are returning the actual data from our mock `fetch` function.

Note - We don't need internet and browser to test React test cases

```

search.test.js ×  JS RestaurantCards.js  JS Body.js
src > components > __tests__ > search.test.js > test("should search function for 'burger' food work")
1  const { render, screen, fireEvent } = require("@testing-library/react");
2  import { act } from "react-dom/test-utils";
3  import Body from "../../components/Body";
4  import resListMockData from "../../components/mocks/mockResListData.json";
5  import { BrowserRouter } from "react-router-dom";
6  import "@testing-library/jest-dom";
7
8  global.fetch = jest.fn(() => {
9      return Promise.resolve({
10         json: () => {
11             return Promise.resolve(resListMockData);
12         },
13     });
14 });
15
16 test("should search function for 'burger' food work", async () => {
17     await act(async () => {
18         render(
19             <BrowserRouter>
20                 <Body />
21             </BrowserRouter>
22         );
23     });
24
25     const searchBtn = screen.getByRole("button", { name: "Search" });
26
27     expect(searchBtn).toBeInTheDocument();
28
29     const searchInput = screen.getByTestId("searchInput");
30
31     fireEvent.change(searchInput, { target: { value: "burger" } });
32
33     fireEvent.click(searchBtn);
34
35     const searchedRestaurnat = screen.getAllByTestId("resCard");
36
37     expect(searchedRestaurnat.length).toBe(2);
38 });

```

code explanation-

1. In Line #8, we are defining our own fetch function inside JS-DOM global object. We are creating our mock fetch function using `jest.fn()`
2. In Line #25, we are getting the search button from the JS DOM.
3. In Line #27, we are verifying whether the search button exists within the JS DOM or not
4. In Line #29, we are getting searchinput from the JS DOM. JEST will find the input search text box by its data-testid
5. In Line#31, JEST types burger in the search input inside JS DOM.

6. In Line #31, target attribute is similar to synthetic event `e` passed inside callback function of `onChange` event of browser and value is similar to `e.target.Value`

7. In Line #33, JEST clicks the search button

8. In Line #35, we fetch all restaurant list by their `data-testid` and store it in a variable

9. In Line #37 we verify whether the fetched restaurant cards quantity is similar to 2 or not

If all these conditions satisfy, test case is passed.

beforeAll, beforeEach, afterAll, afterEach –

These methods are very helpful for clean-up task. These methods get executed depending on when the test cases are executing.

```
contact.test.js ×
src > components > _tests_ > contact.test.js > describe("contactus component test cases") callback
1  import { render, screen } from "@testing-library/react";
2  import Contact from "../Contact";
3  import "@testing-library/jest-dom";
4
5  describe("contactus component test cases", () => {
6    | beforeAll(() => {
7    |   | console.log("gets executed before all test cases execution");
8    |   | });
9
10   | beforeEach(() => {
11   |   | console.log("gets executed before each test case execution");
12   |   | });
13
14   | afterAll(() => {
15   |   | console.log("gets executed after all test cases execution");
16   |   | });
17
18   | afterEach(() => {
19   |   | console.log("gets executed after each test case execution");
20   |   | });
21
22   > test("should load contact us component", () => { ...
26   |   | });
27
28   > test("should load button inside contact us component", () => { ...
32   |   | });
33
34   > test("should load button with name submit inside contact us component", () => { ...
38   |   | });
39
40   > test("should load input text with placeholder name", () => { ...
44   |   | });
45
46   > test("should load 2 input text input elements inside contact us component", () => { ...
53   |   | });
54   | });
```

How to Achieve HMR (Hot Module Replacement) in React testing?

HMR allows JEST to watch over test files where test case related changes are detected so that Jest automatically runs all the test cases across these files. We don't have to manually execute `npm run test` after every change in test files. To achieve this configuration, we add `--watchAll` switch to our test script command.

```
"scripts": {  
  "start": "parcel ./index.html",  
  "build": "parcel build index.html",  
  "test": "jest --watchAll"  
},
```