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 suing 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 transpiration. 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

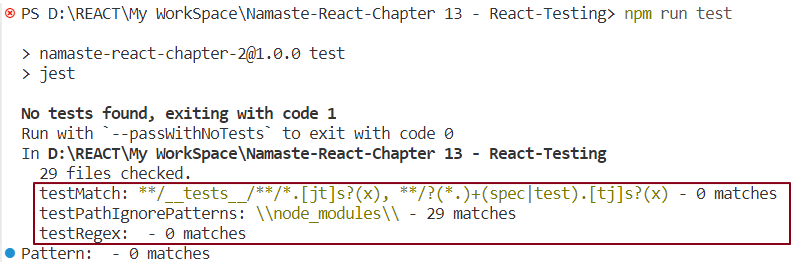
We have successfully configured RTL, JEST, babel & parcel in our project

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.

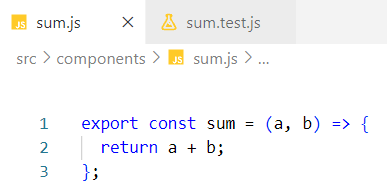
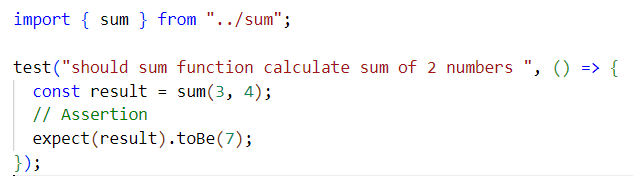
Similarly, when we run test cases, they need an environment for execution. This Runtime environment is given by JS DOM.JS DOM is not a browser, but it like a browser which gives features of browser. We render react components in JS DOM in order to test those components.

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.



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

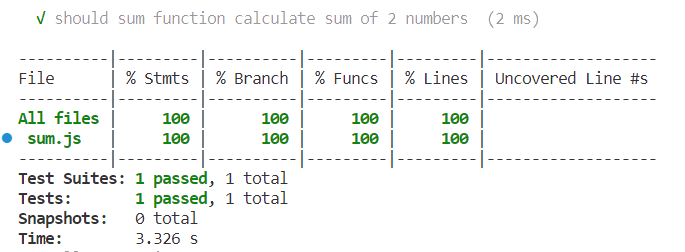
Unit testing JavaScript addition Logic -



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

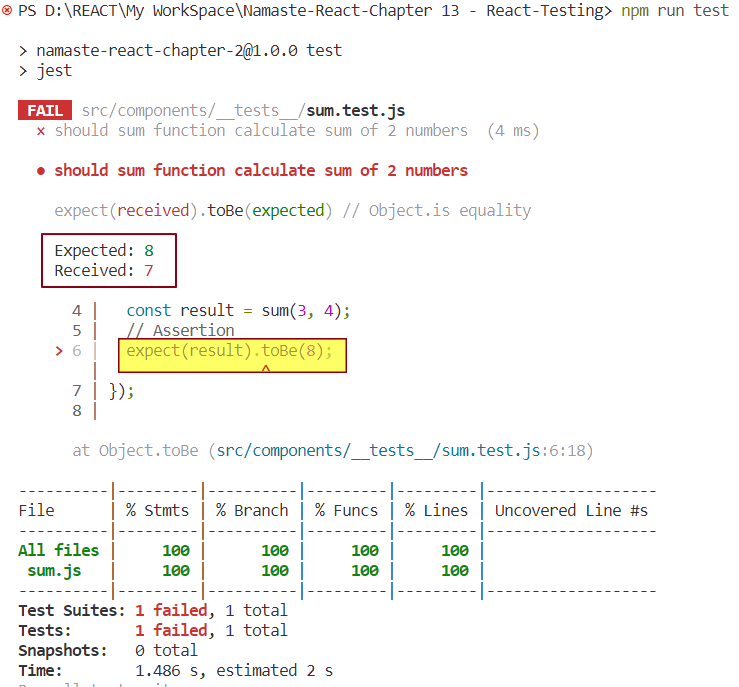
Assertion syntax: expect(<actualValue>).toBe(<expectedValue>) where actualValue is function returned Value and expectedValue is user given value. This syntax may vary depending on use cases.

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



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.



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)





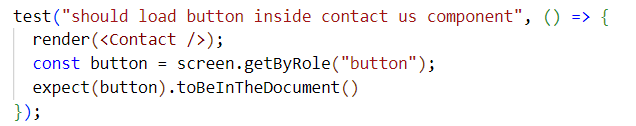
Code Explanation -

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

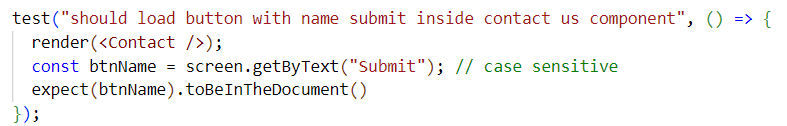
Line #7 - screen is the object given by RTL which stores the JSDOM loaded component. Using screen object, we can get access to individual component elements. getByRole gets elements based on their Role.

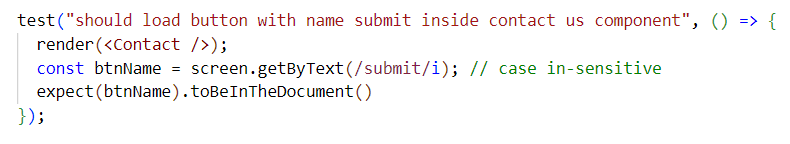
Line #8 - toBeInTheDocument() represents whether the component element is present inside the JS-DOM or not. expect is the method given by JEST framework which does assertion operation.

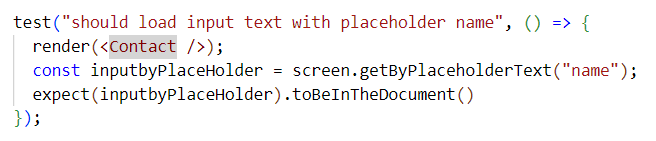
Test case 2 – (checking whether button element is present or not)



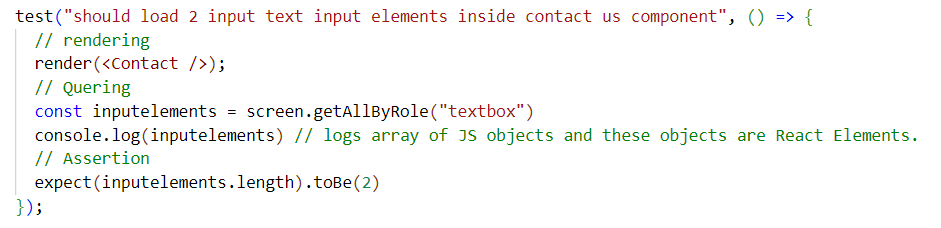
Test case 3 – (checking whether button element with name submit is present or not)



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



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



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(<groupDesciption>, callback)

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



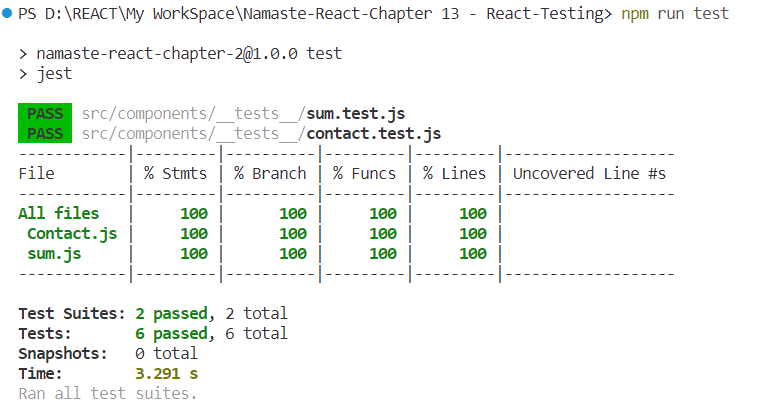
Important points -

\* We can have described inside describe to create sub groups.

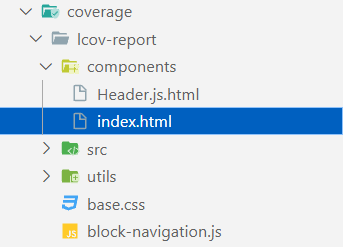
\* While writing test cases instead of **test** we can also use **it**.**it** is alias of **test**

What is Test Coverage?

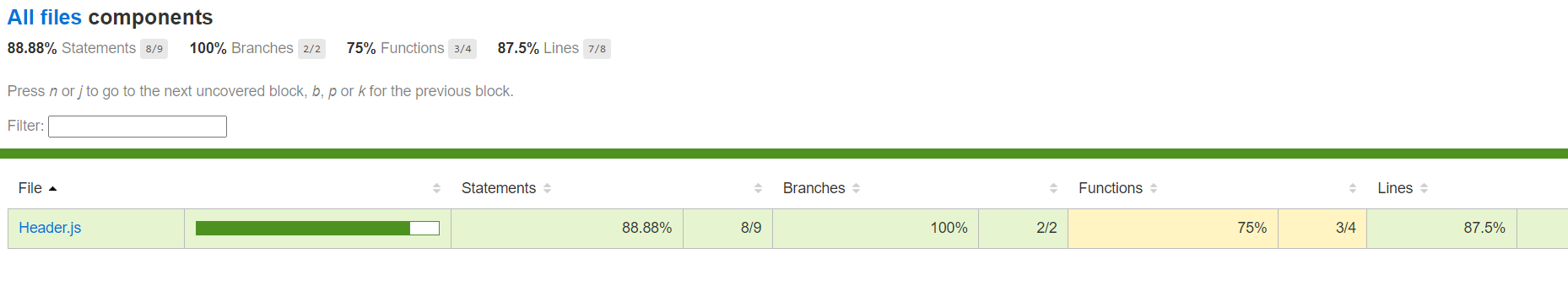
Test Coverage is a technique which determines whether the test cases are actually covering the application code and how much code is exercised when we run the test cases. If there are 10 requirements and 100 test cases are created and if 90 test cases are executed then test coverage is 90%. Below is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



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

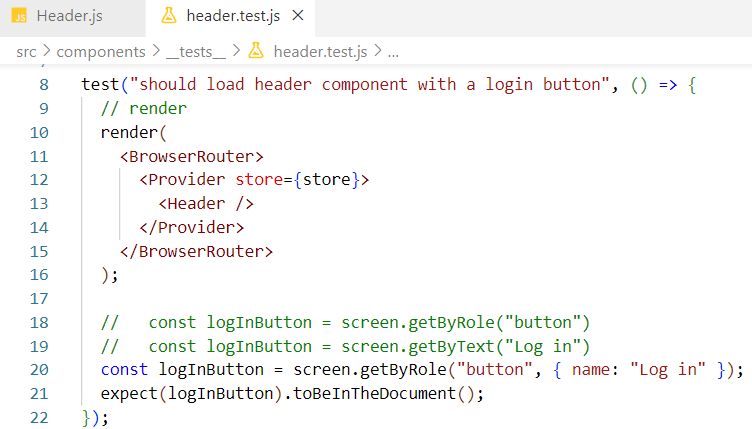


Unit testing Header react component

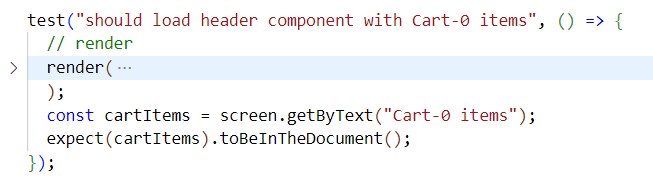
JS DOM understand JSX React code JavaScript code but does not understand redux code. In header component we have used redux functionalities. while writing test cases we have to provide redux store to the header component in test file.

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.



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



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

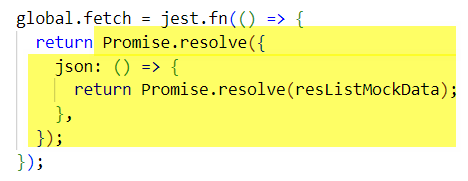


Integration testing of a react component – (search functionality)

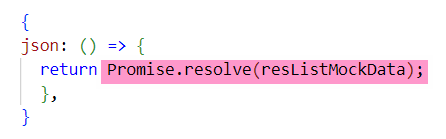
Browser has a powerful API fetch which is responsible for fetching the data from the server. But fetch always returns a promise which gets resolved when json data is available. This is not the actual data that we want, this is a json data which returns another promise and the promise gets resolved when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



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



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



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

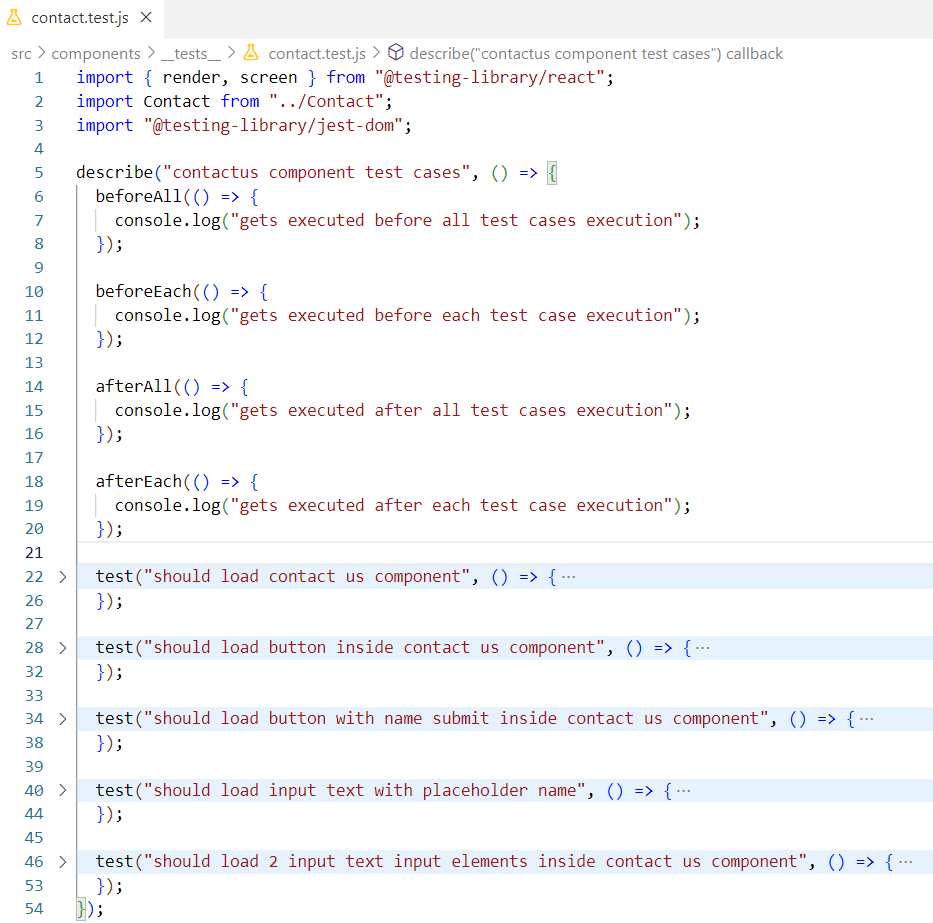
**8.**In Line #35, we fetch all restaurant list by their data-testid and store it in a variable

**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.



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.

