Chapter #13 - React – Testing

When we make code changes, they can introduce bugs anywhere in the application. Even a single line of code can break everything — that’s where React testing comes in.

Why Testing?

Testing helps **catch** bugs early. It helps in maintaining code quality.  
Testing improves a developer’s confidence by ensuring that they are not breaking existing functionality.

Types of Testing -

* Unit Testing - Performed by developers to test individual components or functions in isolation.
* Integration Testing - Conducted by developers to ensure that different modules or components work together correctly.
* End-to-End (E2E) Testing - Performed by developers or testers to test the complete flow of an application, from start to finish, as a user would experience it.
* Manual Testing - Done by developers or testers to manually verify that the application works as expected without automation tools.
* Automated Testing - Involves writing scripts or using tools to automatically run tests; can be performed by both developers and testers.

Unit Testing -

In **unit testing,** we test a React component in **isolation**. We focus on a specific component and verify its behavior **without worrying about other components or external dependencies.**

Integration Testing -

In integration testing, we test the interaction between components. It focuses on verifying how interrelated components work together.

For example, suppose we have a restaurant search textbox and a search button inside the Header component. When a user types something into the textbox and clicks the search button, we expect a list of restaurants to appear in the Body component.

In this case, the Header and Body components collaborate to perform the search functionality. This is where integration testing becomes important — to ensure that these components work correctly when integrated together.

End to End Testing (e2e Testing) -

End-to-End (E2E) Testing involves testing a React application from the moment a user lands on the website to the point they leave. It ensures that the entire application workflow and all functionalities behave as expected in a real-world scenario.

In E2E testing, developers or testers thoroughly test the application's features and code flow from start to finish.

Tools commonly used for End-to-End testing include Cypress and Selenium.

Manual Testing -

In **manual testing,** developers or testers verify the application's functionality **directly on the screen.** This process is **time-consuming** and **prone to human errors**, making it less efficient compared to automated testing.

Automated Testing -

In automated testing, code tests the code instead of a human manually verifying functionality on the screen.

In Unit and Integration Testing, developers write test cases in separate test files. When these test files are executed, they automatically test the application's functionality.

Behind the scenes, the code written in the test cases interacts with and validates the application code to ensure it behaves as expected.

What are test cases?

Test cases in React are code snippets that ensure a React component behaves as expected. These test cases can be written using various testing frameworks such as Jest, Enzyme, and libraries like React Testing Library (RTL).

The primary purpose of writing test cases is to catch bugs early in the development process and to validate changes made to a component without breaking existing functionality.

What is TDD (Test Driven Development)?

In **Test-Driven Development (TDD)**, we write test cases **before** writing the actual code. This approach helps ensure we have **100% test coverage** from the very beginning of development.

### Testing Tools -

* **Cypress** - A modern, developer-friendly tool for end-to-end testing, with fast execution and easy setup.
* **Selenium** - A widely-used framework for automating browsers, supporting multiple languages and browsers.
* **Headless Browsers** - Browsers without a graphical user interface, used for automated testing (e.g., Puppeteer, Playwright). (You can explore more about headless browsers online.)

### Other Types of Testing -

* **Functional Testing** - Verifies that the application functions according to the specified requirements.
* **System Testing** - **System Testing** is the phase where we test the entire, fully integrated application to ensure that all components work together correctly and meet the specified business requirements.
* **Smoke Testing** - A quick set of tests to check whether the basic functionalities of an application work, often called build verification testing.
* **Regression Testing** - Ensures that recent code changes have not adversely affected existing functionality.
* **Security Testing** - Identifies vulnerabilities and ensures that the application protects data and maintains functionality as intended.

**(You can explore these testing types in more detail online.)**

React Testing Library (RTL)

***RTL is one of the most standard libraries used for writing test cases in React.***  
  
RTL (**React Testing Library**)is widely adopted for testing React components, encouraging good testing practices by focusing on user behavior rather than implementation details.

***RTL is built on top of DOM Testing Library (DTL).***  
RTL uses **DOM Testing Library** under the hood to interact with the DOM.

***RTL is a wrapper around DTL.***  
RTL wraps DTL and adds React-specific functionality, such as rendering React components and managing React lifecycles during tests.

JEST

**Jest** is a delightful JavaScript Testing Framework commonly used with React applications.

**React Testing Library** (which is built on top of **DOM Testing Library**) works seamlessly with Jest for writing UI-focused tests.

**Jest does not use Babel internally by default**, but when you're testing modern JavaScript (like ES6+ or JSX), Babel is typically used in your setup to **transform the code** so Jest can understand it.

**📝 Note - 1**

When we create a React project using create-react-app, we can write test cases right out of the box without any additional configuration. This is because **React Testing Library (RTL)** and **Jest** come pre-installed and pre-configured with the create-react-app setup.

However, in our case, since we’ve built the application **from scratch** using the **Parcel bundler**, we need to **manually install and configure** all testing tools like **Jest** and **React Testing Library**.

Configuring RTL and JEST in our Application

**1.** Install React Testing Library with: npm i -D @testing-library/react

**2.** Install Jest with: npm i -D jest

**3.** Install Babel dependencies with: npm i -D babel-jest @babel/core @babel/preset-env

**4.** Create a babel.config.js file at the root of your project and add the following code to configure Babel with your current Node version



**5.** Until now, Parcel bundler has been using Babel with its own configuration. Now, Jest also uses Babel based on its configuration. However, Jest’s Babel setup overrides Parcel’s Babel configuration, which causes a conflict between Jest and Parcel. To resolve this, we need to adjust Parcel’s behavior to work smoothly with Jest’s Babel configuration. To do that, create a .parcelrc file in the root directory of your project with the following configuration.

{

  "extends": "@parcel/config-default",

  "transformers": {

    "\*.{js,mjs,jsx,cjs,ts,tsx}": [

      "@parcel/transformer-js",

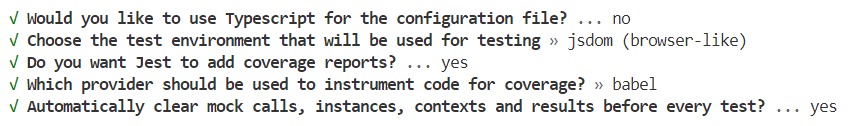
      "@parcel/transformer-react-refresh-wrap"

    ]

  }

}

**6.** Configure Jest by running: npx jest --init



**7.** If Jest version is above 28 (npx jest --version), install the JS DOM library with: npm i -D jest-environment-jsdom

**8.** Install JSX support for tests using: npm i -D @babel/preset-react

**9.** Add @babel/preset-react to your babel.config.js presets like this:

module.exports = {

  presets: [

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

    ["@babel/preset-react", { runtime: "automatic" }],

  ],

};

@babel/preset-react converts JSX code (returned by components inside render()) into plain JavaScript, allowing Jest and React Testing Library to correctly execute and understand it during tests.

**10.** Install @testing-library/jest-dom with npm i -D @testing-library/jest-dom.

**11.** Optional - Install Jest code IntelliSense typing with npm install -D @types/jest

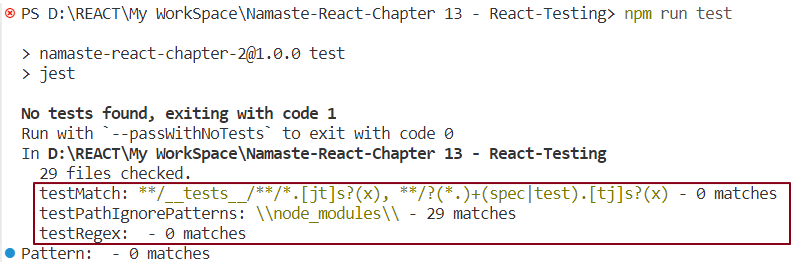
We have successfully configured RTL, Jest, Babel, and Parcel in our project.

What is JS DOM?

When we run a React application, it requires a runtime environment where the code can execute. This runtime environment is provided by the browser, where React components get rendered in the DOM.

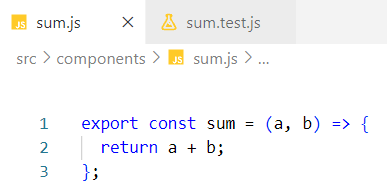
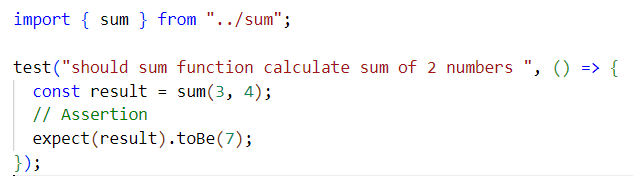
Similarly, when we run test cases, they also need an environment to execute. This runtime environment is provided by JSDOM. JSDOM is not a browser, but it simulates a browser-like environment by providing many browser features. We render React components in JSDOM in order to test them.

At this point, when we run npm run test, we might see a message in the terminal saying **No tests found**.This means Jest searched through a set of files (based on a regex pattern, e.g., 29 files) but did not find any test cases. Jest also searched inside the \_\_tests\_\_ (dunder tests) folder but still could not find any test files or test cases.



Let’s create our first test case to verify a simple JavaScript addition function.

Unit testing JavaScript addition Logic -



***Test case creation syntax:***



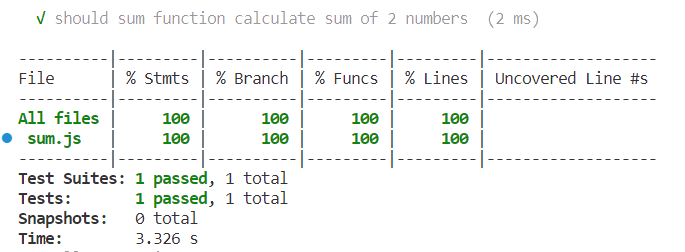
***Assertion syntax:***



Where,

* actualValue is the value returned by the function being tested.
* expectedValue is the value you expect.

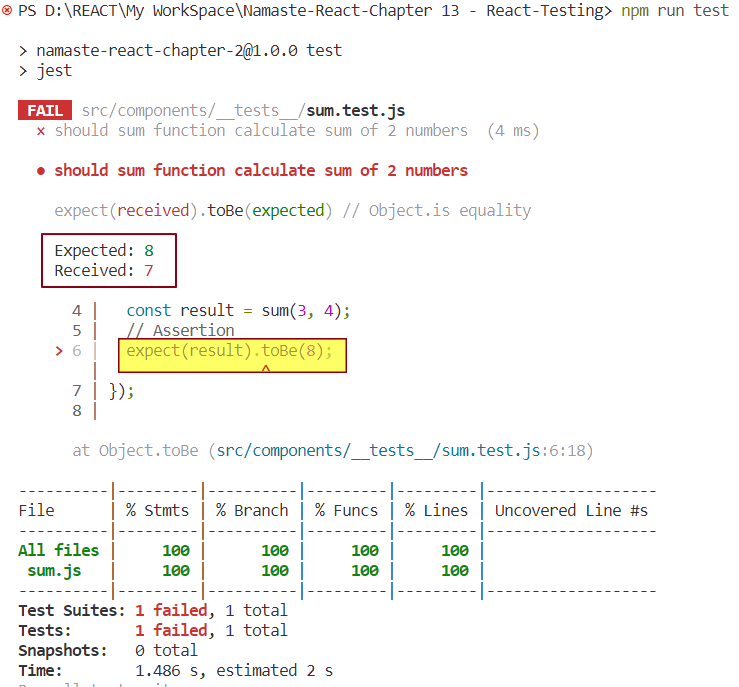
This syntax may vary depending on the type of assertion or use case. We have created a test case for the sum function. Now, let’s run the test case using: npm run test



The table above is a coverage report that shows the percentage of test cases executed.

* Test Suites represents the number of modules or test files containing test cases.
* Tests represents the total number of test cases across all modules.

The test case we wrote will fail if the expected value is anything other than 7.



The received value is the value returned by the function, while the expected value is the user-defined value.  
Note that assertions are not mandatory; without an assertion, the test case will pass by default when executed.

**📝 Note - 2**

* Test files should be named as <FileName>.test.<ext> or <FileName>.spec.<ext>.
* Test cases should be placed inside a folder named \_\_tests\_\_ so that JSDOM can automatically detect and run them. Alternatively, naming your test files with extensions like .test.ts, .spec.ts, .test.js, or .spec.js also allows the test runner to recognize and execute them.

1.Unit Testing the Contact Us React Component -

***Test case 1 -*** Here’s a simple test case to check if the header element is present in the **Contact Us** component

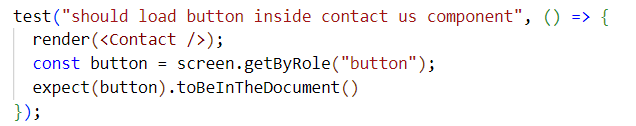




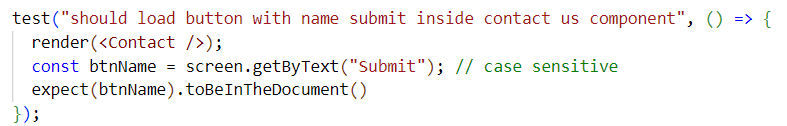
***Code Explanation:***

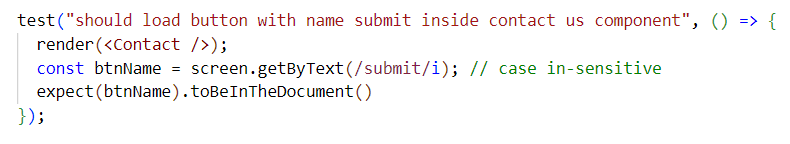
* Line 6 - render() renders the ContactUs component inside the JS DOM.
* Line 7 - When the component is rendered inside the JS DOM, the entire component tree is encapsulated within an object called screen. Using the screen object, we can access specific elements by querying their roles with getByRole.
* Line 8 - toBeInTheDocument() checks whether the element is present in the JS DOM. expect is a Jest function used for assertions, while matchers like toBeInTheDocument() are provided by @testing-library/jest-dom to extend Jest’s assertion capabilities.

***Test case 2 - Checking if the button element is present or not***



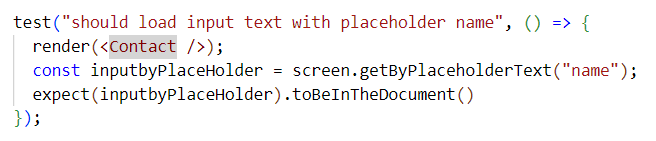
***Test case 3 - Checking if a button element with the name “Submit” is present or not***



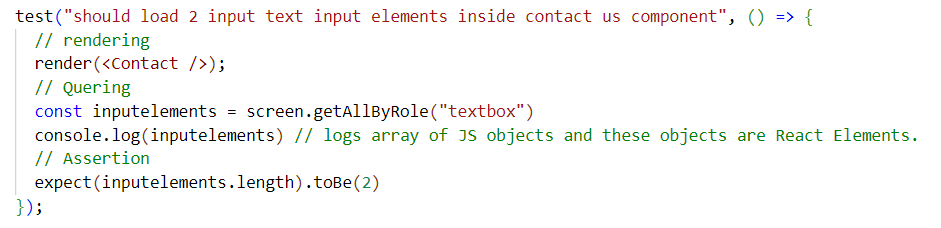


/submit/i makes the test case pass regardless of letter casing, but the text "submit" must still be present somewhere in the JS DOM for the test to succeed.

***Test case 4 – Check if an input element with a specific placeholder is present after rendering***



***Test case 5 – Verify that multiple input elements are present after rendering.***



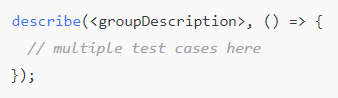
inputelements is an **array of HTMLInputElement objects** representing the actual <input> elements rendered inside the JS DOM by your Contact component.

**📝 Note - 3**

* npm run test <testFileName> runs test cases in the specified test file only.
* React elements represent the Virtual DOM elements, not the real DOM directly.
* npm run test without arguments runs all test cases across all test files in the project.

When a single test file contains many test cases, managing them can get difficult. To organize better, we group related test cases using describe()

***Syntax -***



Within the callback function body, we write the test cases that belong to the group we are creating.



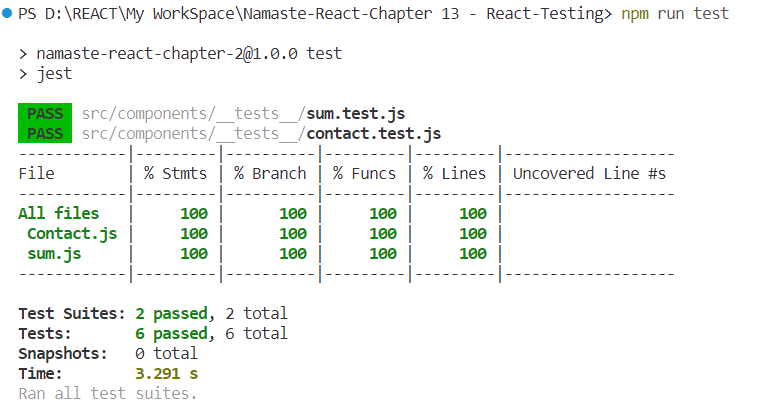
***Important points:***

* We can nest describe blocks inside other describe blocks to create subgroups.
* Instead of test, we can also use it to write test cases; it is an alias of test.

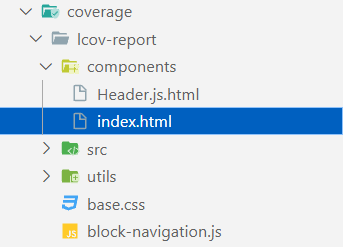
What is test coverage?

Test Coverage shows how much of your application code is tested by your test cases. It helps you know if your tests cover all important parts of the code.

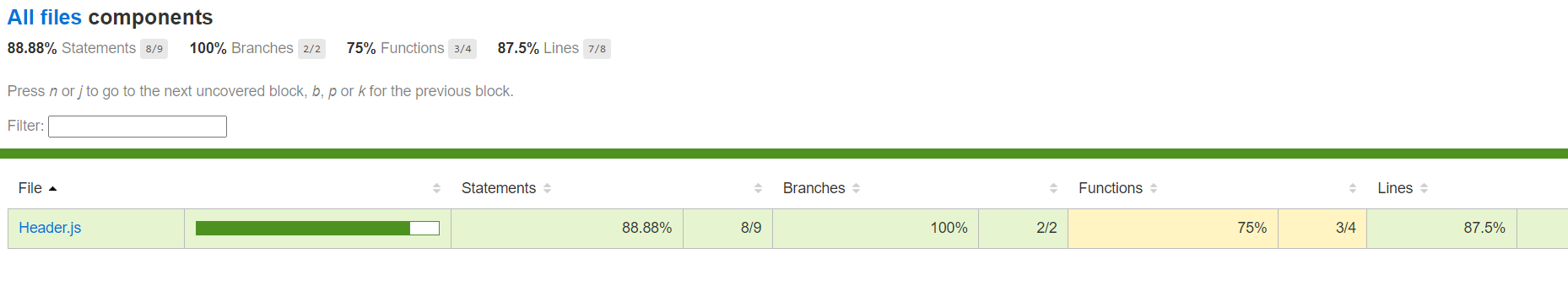
For example, if you have 10 features and 100 test cases, and 90 of those tests run successfully, your test coverage is 90%.



When you run npm run test, a **coverage** folder is created which contains reports showing which code was tested. You usually add this folder to .gitignore so it’s not pushed to Git.



Opening the index.html file inside the coverage folder in a browser gives a detailed report of what parts of your code need more testing.



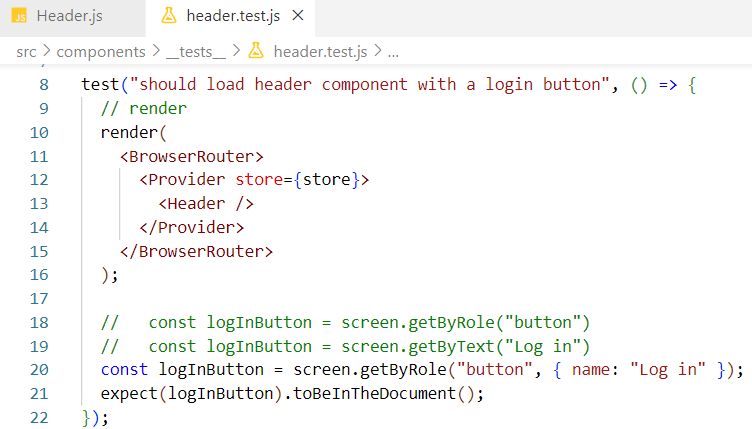
2.Unit Testing the Header React Component

JS DOM can understand JSX and standard React JavaScript code, but it does not natively understand Redux logic or React Router features.

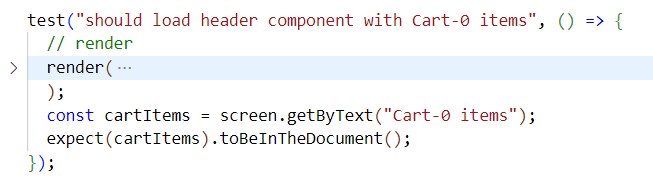
Since the Header component uses Redux for state management, when writing test cases, we must wrap the Header component with a Redux store provider in the test file.

Similarly, because the Header component uses React Router for routing, we need to wrap it with a Router provider in the test environment to properly simulate routing functionality during tests.

***Test Case 1 - Verify that the Login button is present in the Header component.***



***Test Case 2 - Verify that the Cart showing 0 items is present in the Header component***



***Test Case 3 - Verify that the Login button changes to Logout upon clicking***



***Test Case 4 - Verify that the Logout button changes to Login upon clicking***



### What is fireEvent?

fireEvent is a utility provided by **React Testing Library** that allows you to **simulate DOM events** like click, change, submit, keydown, keyup, keypress, focus, blur

A utility is simply a helper tool or function that makes it easier to perform common tasks.

In the context of testing with React Testing Library fireEvent is a utility because it provides you with convenient functions (like click, change, input, etc.) to simulate how a real user interacts with the UI.

### What is .click()?

.click() is a method from fireEvent that **simulates a user clicking** on a DOM element.

It simulates a **real user clicking the Login button** on your Header component. This triggers the onClick logic defined inside your component which updates the button label between **Login** and **Logout.**

3.Unit Testing of Restaurant Card Component

The RestaurantCard component receives data through props.

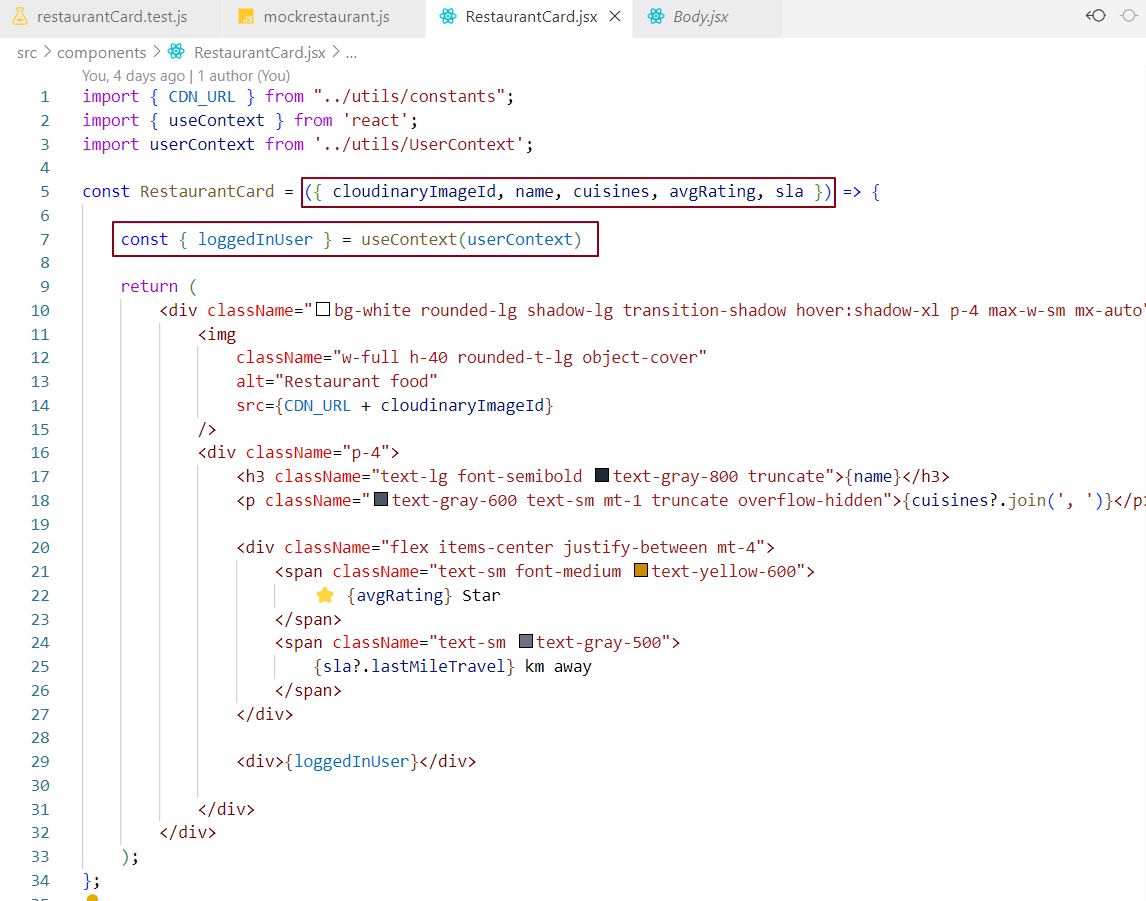
Since we're writing test cases for this component, we need to simulate that behavior by passing the appropriate data from the test file. To keep the tests clean, maintainable, and reusable, we'll create a separate **mock data file** that exports a mock restaurant object. This object should match the exact prop structure expected by the RestaurantCard component.

We'll then import and use this mock data in our test cases to verify that the component **renders the correct content into the JS DOM** based on the provided props (like name, cuisines, rating, etc.).

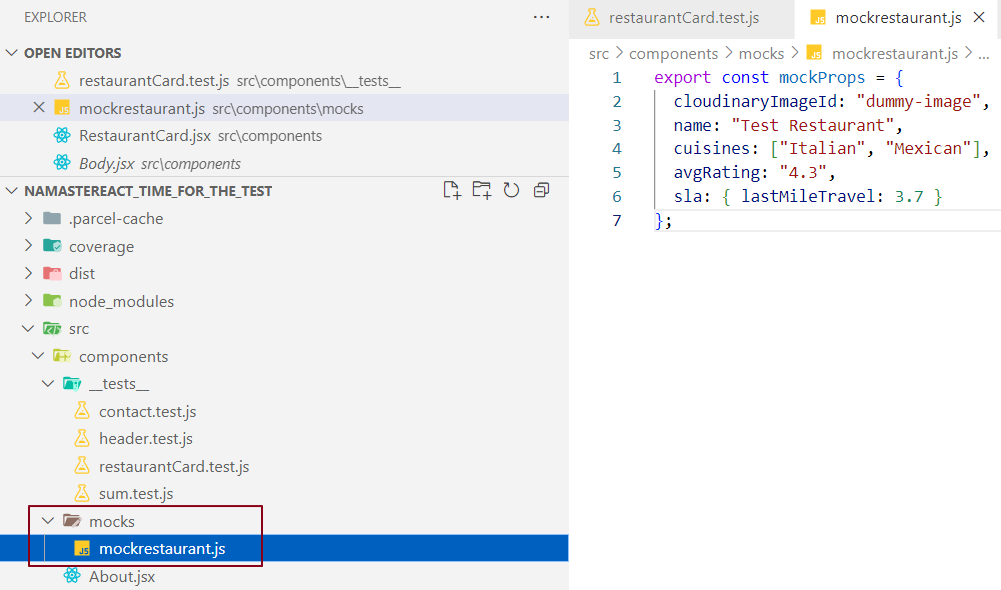
Additionally, the component uses a **React Context (**userContext**)** to display the logged-in user's name. To test this, we'll **wrap the component in a mock provider** with a hardcoded context value. This ensures we can **verify that the context value also renders properly in the JS DOM.**

By doing this, we validate both prop-based rendering and context-based rendering within the DOM.

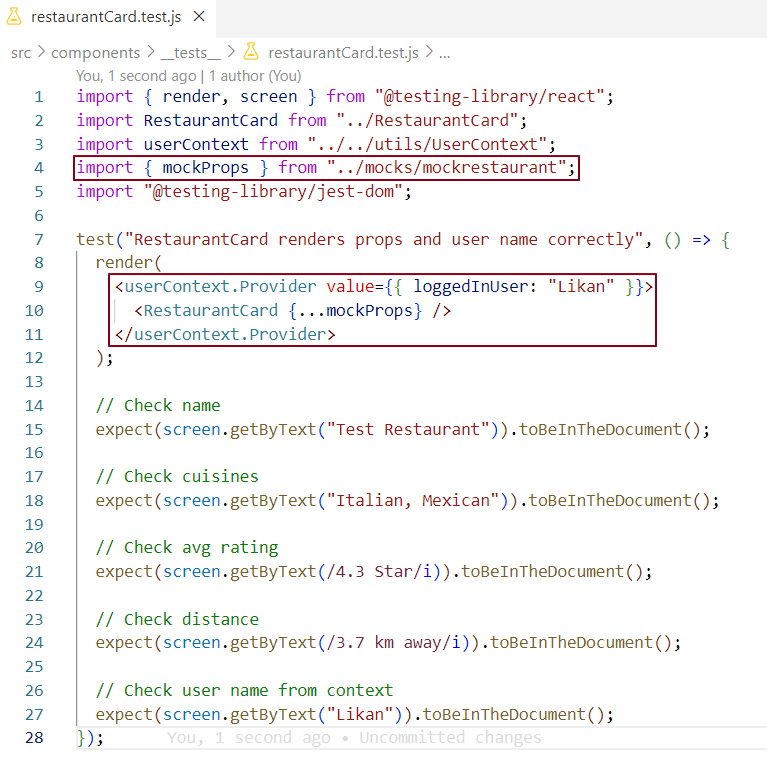
***RestaurantCard.jsx -***



***mockrestaurant.js -***



***restaurantCard.test.js -***



4. Unit Testing for a Higher-Order Component -

**🧠 How <RestaurantCard {...props?.resData?.info} /> works Internally in React ?**

***Step 1: Dummy info Object Example -***



***🔄 Step 2: What This JSX Does Internally***



This is **equivalent to writing:**



👉 So we’re passing all key-value pairs from info as individual props to RestaurantCard.

👉 We are using the **spread operator (...)** to unpack the info object while passing it as props to the RestaurantCard component. Since this is JSX (which allows us to embed JavaScript expressions), we enclose it within {} like this:

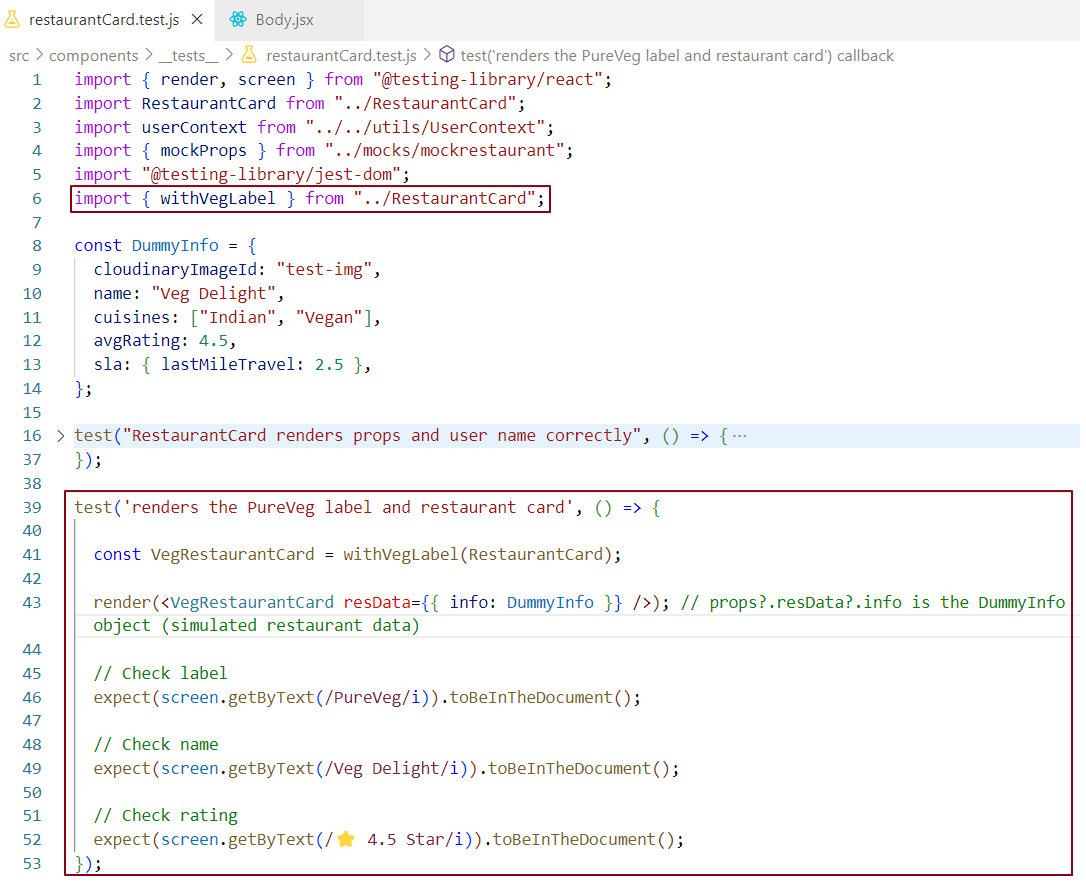


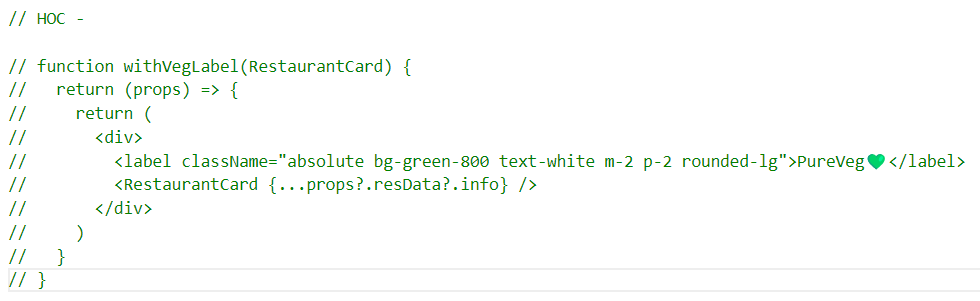
👉 Inside **JSX**, anything inside {} is treated as a **JavaScript expression.**

***Step 3: What the Component Receives via Destructuring***



Let's test our Higher Order Component now — the enhanced component named withVegLabel.





### *Code Explanation -*

In our test case, we are verifying whether the Higher-Order Component (HOC) is rendered in the DOM or not. We are also checking if the HOC contains the expected label (PureVeg💚). Additionally, we confirm that the wrapped RestaurantCard component correctly displays the restaurant’s properties in the UI.

5. Integration Testing of a React Component - Search Functionality

***Fetch API and Jest Mocking Explanation -***

The browser provides a powerful API called fetch to retrieve data from a server. The fetch function always returns a promise that resolves to a Response object. When you call .json() on this response, it returns another promise that resolves to the actual JSON data.

However, when running tests in Jest, which uses a JavaScript DOM (JS-DOM) environment, fetch is not available because JS-DOM is not a real browser and does not implement browser APIs like fetch.

Therefore, in Jest tests, we need to mock the fetch function to simulate its behavior. Mocking means creating a fake version of fetch that behaves like the real browser API, allowing us to control and test how our code handles data fetching.

In Jest, you can create a mock fetch function with -



***What is global.fetch?***

* fetch is a function normally provided by the browser to make network requests.
* In Node.js or test environments like Jest (which runs outside the browser), fetch is notdefined by default.
* global is a global object in Node.js and Jest. Assigning global.fetch means we’re defininga globalfetch function available everywhere in the test environment.

### *What is jest.fn()?*

* jest.fn() makes a **fake function** you can use in your tests.
* This fake function **remembers**:
  + How many times it was called.
  + What data (arguments) it was called with.
  + What it gave back (its return value).
* You can also **tell this fake function what to do** when someone calls it.

***global.fetch = jest.fn(); means***

* We replace the **real** fetch (which talks to the internet) with a **fake** fetch made by Jest.
* This fake fetch lets us **pretend** to get data without calling a real server.
* Because real fetch returns **promises**, our fake fetch can also return **mock promises** with fake data - so tests behave like real ones but faster and safer.

***Mocking the Fetch API with Jest to Simulate Network Requests -***



### *What’s happening here?*

* global.fetch is replaced with a Jest mock function.
* When you call fetch(), it returns a **promise** that immediately resolves to an object.
* That object has a .json() method, which itself returns a **promise** resolving to the mock data { message: "Hello from mock fetch!" }.

### *Why use a callback function inside jest.fn()?*

* jest.fn() **creates a mock function** - a fake function you can control.
* When you pass a callback like () => Promise.resolve(...) to jest.fn(), you’re telling the mock function **what to do when it is called**.
* So, **this callback is the implementation of the mock function** - it defines what the fake fetch returns every time you call it.

### *When is this callback executed?*

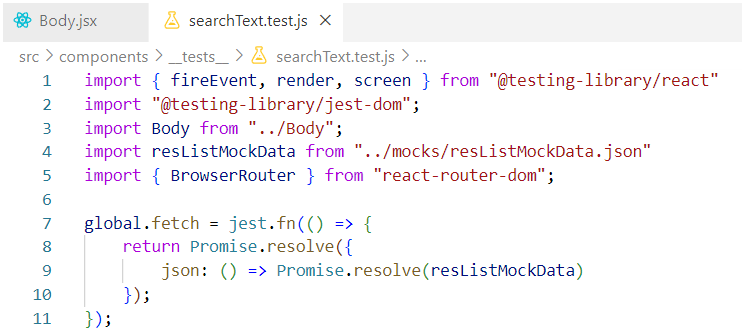
* The callback function inside jest.fn() runs **whenever the mock function is called.**
* So, when your test code calls fetch(...), the **mock fetch function runs the callback** and returns the promise you defined.
* This simulates the real fetch behavior: returning a promise that resolves to a response-like object with a .json() method.

### *In simple terms -*

* You replace fetch with a fake function (jest.fn()).
* You tell this fake function what to do **when called** - here, return a promise that resolves with mock data.
* When your code calls fetch, it triggers the callback and gets the mock data wrapped in promises.

We mock the API call **before rendering** the Body component in our test. This way, when Body runs its useEffect and calls fetch(), it uses our mocked data instead of making a real network request, ensuring consistent and reliable test results.

***Mocking fetch API call -***

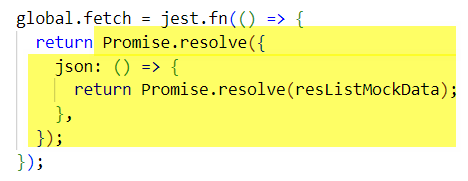
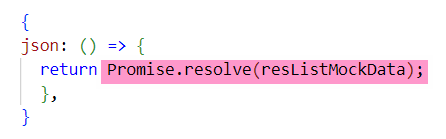


### *How Mocked fetch Works in the Test ?*

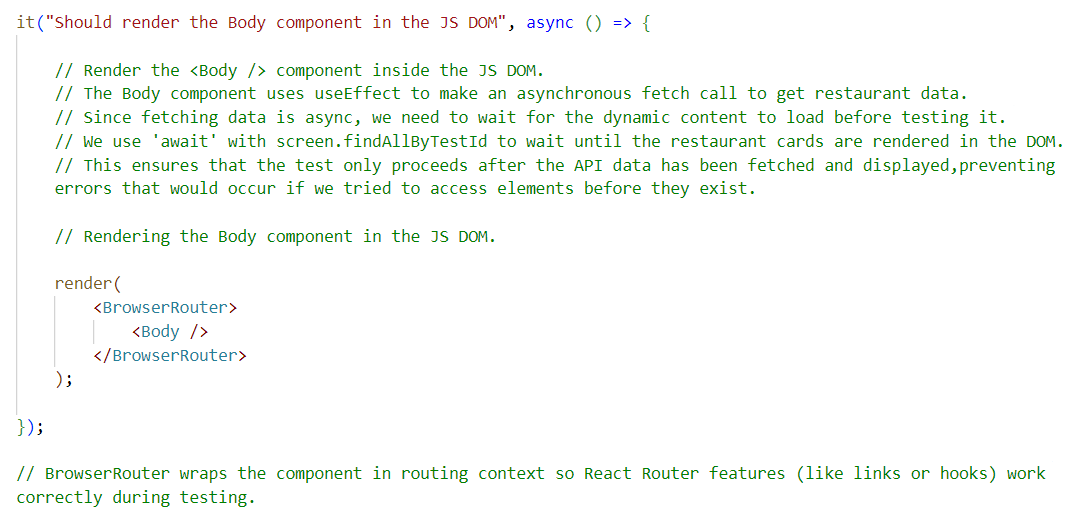
When the Body component renders inside the JS DOM, it calls fetch() to get data. Because fetch is mocked, the mock function runs instead of making an actual network request.

The mock returns a resolved promise that simulates the Response object.  
This Response object has a .json() method.

Calling .json() returns another resolved promise that resolves with the actual mock JSON data.

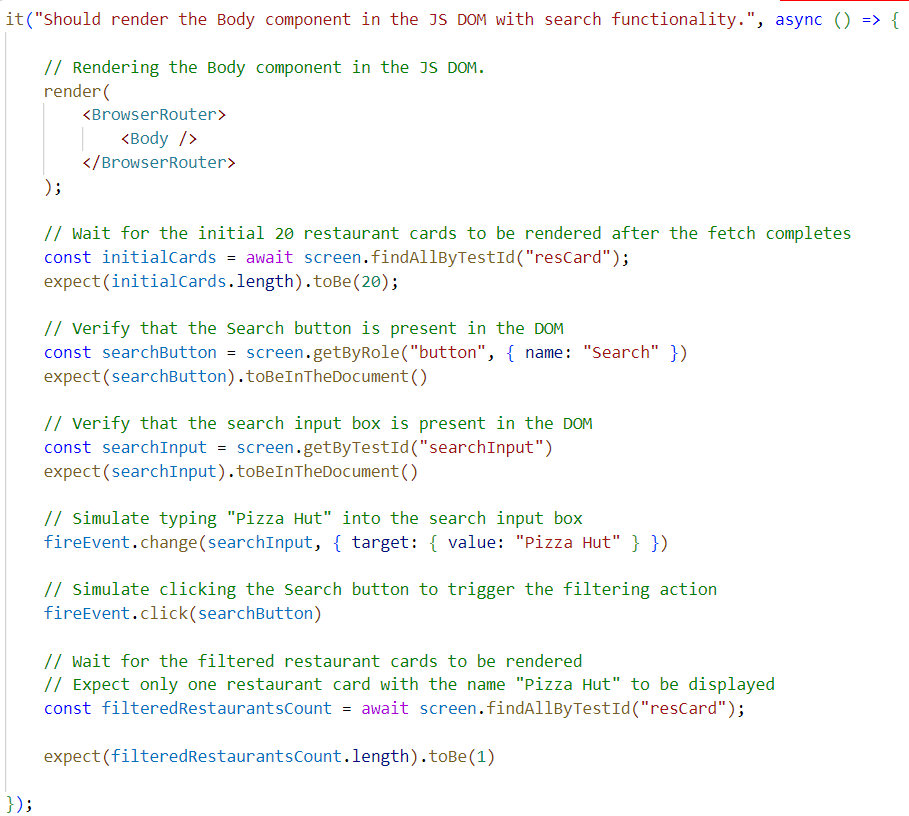


***searchText.test.js -***



### *Explanation of Mock fetch Usage in our Test -*

* When you render the Body component in the test (render(<Body />)), React runs the component’s code.
* Inside Body, there is a useEffect hook that calls fetch() to load restaurant data asynchronously.
* Since you have mocked global.fetch with jest.fn(), **this call to** fetch() **inside** Body **actually triggers the mocked fetch function**,not the real browser API.
* The mocked fetch immediately returns a resolved Promise with your mock data (resListMockData).
* This allows the component to receive the mock data just like it would receive real API data, but without making a network request.
* Because the data fetch is async, the component will update once the mock data arrives, rendering the restaurant cards.
* Your test then waits for this rendering using await screen.findAllByTestId("resCard") to ensure the dynamic data is loaded before asserting.



In this test case, we load the Body component, simulate typing text in the input box, click the search button to filter results, and verify that the filtered restaurant cards are displayed correctly in the UI. All inline comments are self-explanatory.

**📝 Note - 4**

We don’t make actual network calls in our test cases because tests don’t run in a real browser environment with internet access. Instead, they run in a simulated JS DOM, which is designed specifically for running tests.  
Therefore, testing React components doesn’t require an internet connection or a browser.

***How fireEvent.change Works in React Testing ?***

In a React application, whenever a user interacts with an input field (like typing into a textbox), an event is fired. React handles this interaction using event handlers, such as onChange, which receive an event object as an argument.

That event object typically looks like this -

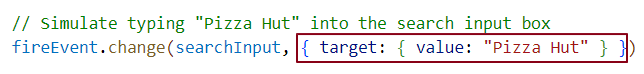


In this object -

* target refers to the input element.
* value is the current value of the input field.

***In Testing: Simulating Events with fireEvent***

* When writing unit tests using libraries like React Testing Library, we often need to simulate user actions to verify how components behave.
* To simulate a user typing into an input box, we use-



### *What This Does ?*

* This **manually triggers the** onChange **event** on the input element.
* The second argument is a **mock event object**.
* It mimics what would happen in a real browser — where the browser passes an event object to your handler.

This is equivalent to the browser calling -



So, in your component, react sees event.target.value as "Pizza Hut" — just like a real user typed it.

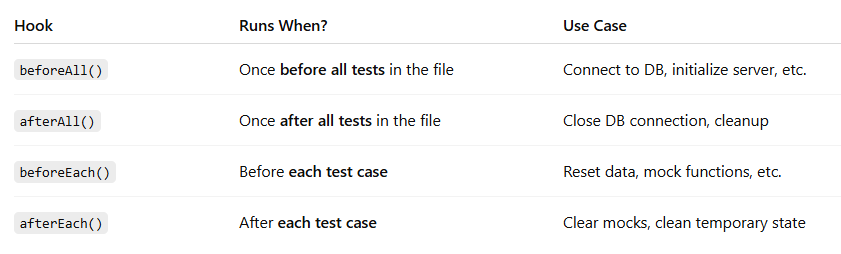
📌 ***Why This Is Useful:*** It lets you simulate user input and test how components handle and respond to events.

The test cases we wrote involve multiple components like <Body>, <SearchBar>, and <RestaurantCard>, along with systems like fetch and BrowserRouter; that’s why it’s called integration testing - it verifies how these pieces work together as a whole.

Jest Life cycle hooks

In Jest, lifecycle hooks are special functions that run code **before or after test cases**. They're typically used for **setup and teardown** tasks.

* **Setup** refers to preparing everything the test needs to run—such as creating data, initializing variables, or opening connections.
* **Teardown** means cleaning up after the test—like deleting test data, closing connections, or resetting mocks.

***Here are the main Jest lifecycle hooks***

***Jest lifecycle hooks implementation***

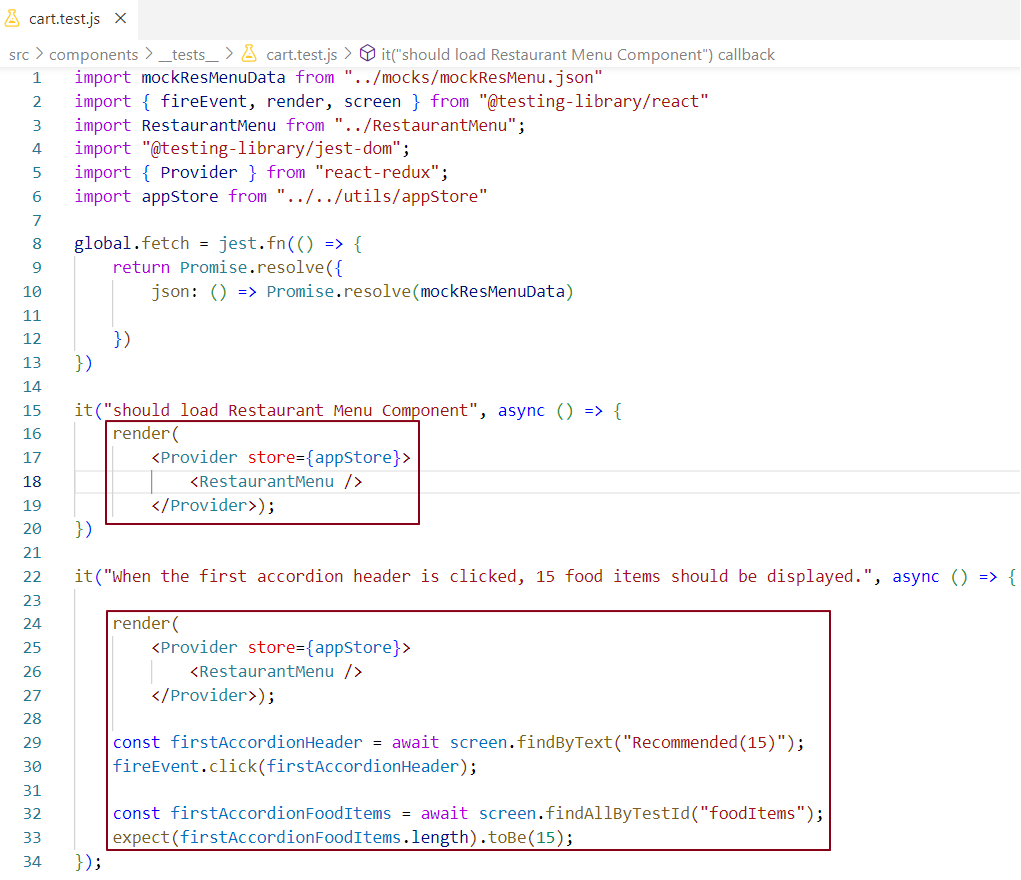


**6. Integration Testing for the "Add to Cart" Feature**

In the UI, we render the **RestaurantMenu component** when a restaurant card is selected. Inside the **RestaurantMenu component**, we display the **restaurant name** and an **accordion list** of menu categories.

***Let's mock the list of accordion categories using mock data and render them in the******JS DOM****.*

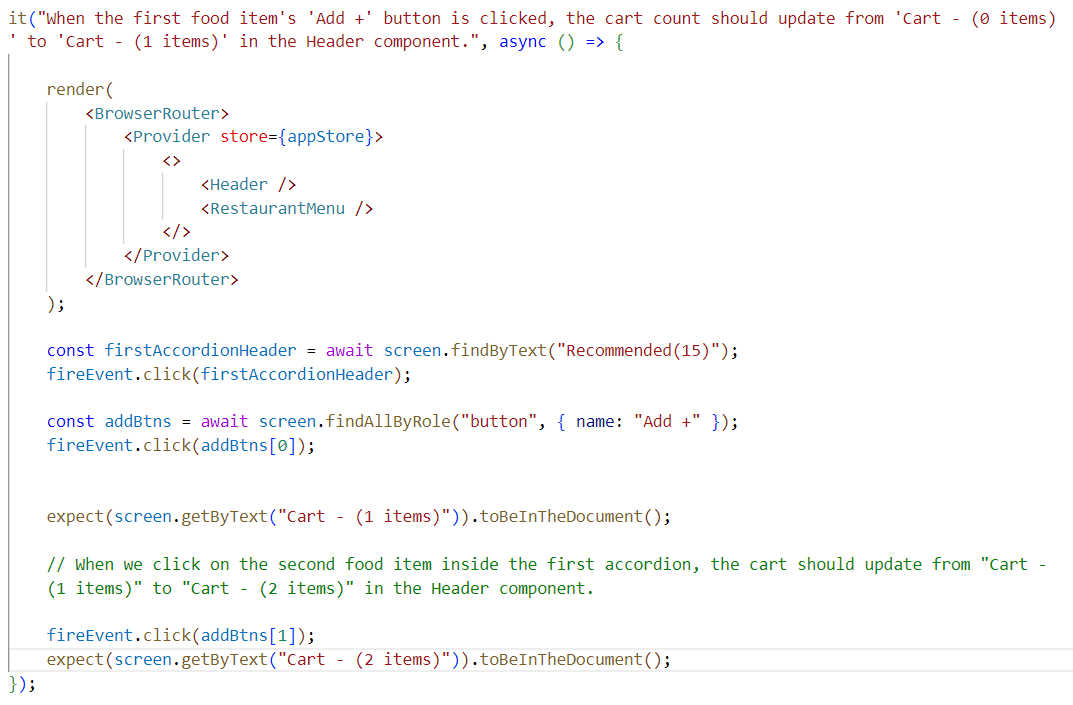
***Let's also test whether 15 food items are rendered in the JSDOM when the 'Recommended (15)' accordion is clicked.***



* getBy... does not wait for elements and runs immediately.
* If your test causes async state changes with getBy..., you must use act() to avoid warnings.
* findBy... waits for elements to appear and automatically wraps in act().
* So, use findBy... to avoid manually using act() in async tests.

The RestaurantMenu component is wrapped with the Provider component because it contains a nested ItemList child component, which uses Redux for state management. Wrapping it with Provider ensures that the Redux store is available to all child components and allows JSDOM to properly execute and test Redux-related logic.

***when the first food item's 'Add +' button is clicked, the cart count should update from 'Cart - (0 items)' to 'Cart - (1 items)' in the Header component. When we click on the second food item inside the first accordion, the cart should update from "Cart - (1 items)" to "Cart - (2 items)" in the Header component.***



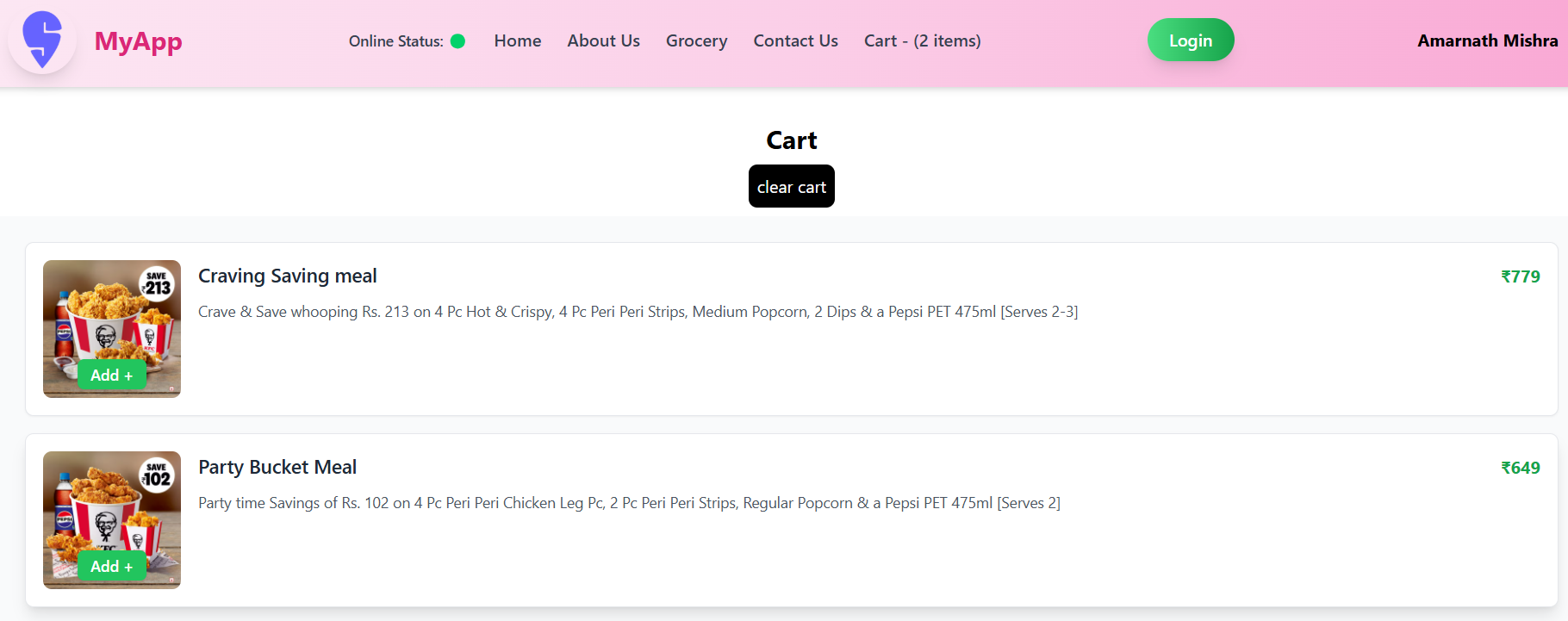
In this test case, we first expand the "Recommended(15)" accordion and select the first food item.

Then, we click on its "Add +" button, which updates the cart count in the Header component from "Cart - (0 items)" to "Cart - (1 items)".

Then, we click on its "Add +" button of the second food Item, which updates the cart count in the Header component from "Cart - (1 items)" to "Cart - (2 items) ".

To reflect this properly in JSDOM during testing, we need to render the Header component along with RestaurantMenu. Since Header uses routing features (like Link), we also wrap both components inside a BrowserRouter and Provider to ensure Redux and React Router work correctly in the test environment.

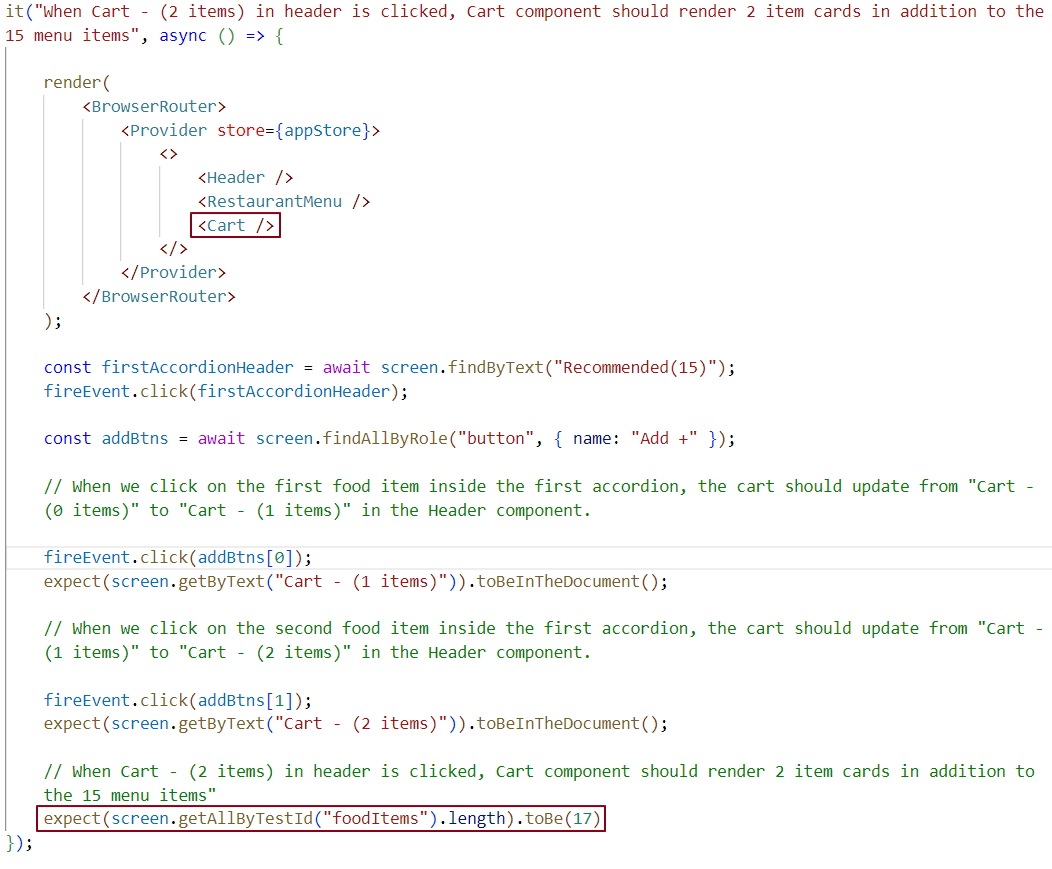
***When Cart - (2 items) in header is clicked, Cart component should render 2 item cards in addition to the 15 menu items***



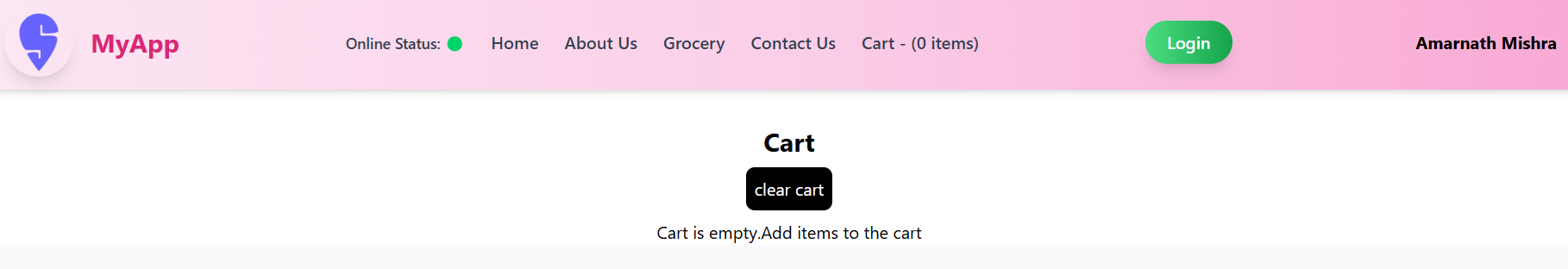
When we navigate to the cart by clicking "Cart - (2 items)" in the Header, we should see two food item cards rendered inside the Cart component. And to do that, we need to render the Cart component in JSDOM as well.

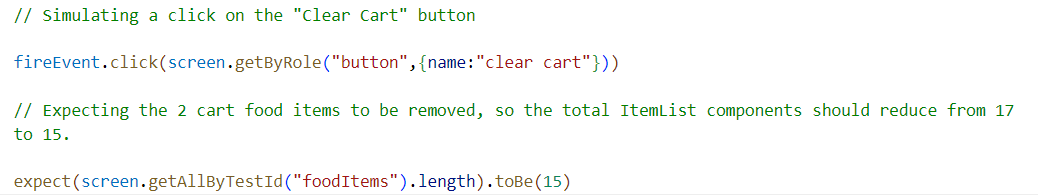
Initially, we rendered 15 food items in the RestaurantMenu through the ItemList component, and we identified them using the data-testid="foodItems".

Since the Cart component also reuses the same ItemList component (and hence the same test ID), we now expect a total of 17 food item components rendered in the DOM — 15 from the restaurant and 2 from the cart.



***When the user clicks the 'Clear Cart' button in the Cart component, it should clear all the items from the cart.***





Note- This is an **integration test** because it validates the interaction between multiple components (Header, RestaurantMenu, and Cart) and ensures that shared state (Redux store) updates and renders correctly across them.

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

In the context of React testing, **Hot Module Replacement (HMR)** refers to enabling **Jest’s watch mode**, where it automatically re-runs test cases as soon as changes are detected in test files. This helps streamline the development workflow by eliminating the need to manually run npm test after every small update.

### Why Use Watch Mode in Jest?

* Automatically detects changes in test files or related source files.
* Instantly re-runs affected test cases.
* Improves developer productivity and speeds up debugging.
* Acts like HMR for your test suite.

### How to Enable Watch Mode ?

**1. Update your** package.json by adding the following script

**2. Run the test watcher** using the command npm run watch-test

### What Happens Under the Hood?

* Jest enters **watch mode**, continuously monitoring files.
* On every detected change, it reruns **all test cases** (--watchAll) or only those affected (--watch).
* This provides a live feedback loop, much like UI HMR in development mode.

If you want Jest to run **only tests related to the changed files**, you can modify the script to: "watch-test": "jest --watch"