JEST is a test runner that finds tests, runs the tests, determines whether the tests passes or failed & reports it back in human readable manner.

React Testing Library (RTL) 🡪 It is a testing utility that provides virtual DOM for testing React Components.

RTL provides a virtual DOM which we can use to interact with & verify the behaviour of react component. RTL is in fact a family of packages which helps testing UI components.

The CORE library is called “DOM Testing library” & RTL is simply a wrapper around the core library to test application in an easier way.

**TYPES OF TESTS**

1. Unit Tests
2. Integration Tests
3. End to End Tests

UNIT Tests focuses on testing the individual building blocks of an application such as class, function or a component.   
Each unit or building block is tested in isolation, independent of other units.  
Dependencies are mocked.

Integration Tests focuses on testing a combination of units & ensuring they work together.

End to End Tests focuses on testing the entire application flow & ensuring it works as designed from start to finish. It involves real UI, real backend DB & real services.

**Test Function**

Test (name, function, timeout)

* “name” is used to identify the test.
* “functions” contains the expectations to test.
* “timeout” specifies how long to wait before aborting the test. Default time is 5 sec.

**Explanation of default Test**

Import {render, screen} from ‘@testing-library/react’;  
Import ‘App’ from ‘./App’;  
Test(‘renders learn react link’, () => {

1. Render(<App/>);
2. Const linkElement = screen.getBYText(/learn react/i); //i means case sensitive.
3. Expect(linkElement.toBeInDocument());

});

1. Creates a virtual DOM of App Component.
2. Screen is imported from React Testing Library which contains the object to query virtual DOM.
3. Expect & Test are automatically imported from JEST.

Note:

We can also use “it” instead of “test”.  
Test.only 🡪 fit  
Test.skip 🡪xit

**WATCH MODE**

Watch Mode is an option that we can pass to JEST asking to watch files that have changed since the last commit and execute tests related only to those changed files.  
It is optimization designed to make your tests run fast regardless of how may tests you have.

test.only(“test name”, function) 🡪 will run only this test.  
test.skip(“test name”, function) 🡪 will skip this test.

**GROUPING TEST WITH JEST USING “describe”**

Syntax 🡪 describe(“name”, function)

1. “name” is group name.
2. “function” contains expectations to test.

describe(“component name”, () => {

test(“testName1”, () => {function})  
test(“testName2”, () => {function})

});

We can also use describe.only & describe.skip.  
We can nest a “describe” within a “describe”.  
We can have multiple describe in same file.  
1 test file is 1 suite.

**File Name Conventions**

Files with following suffix are allowed.

1. .test.js
2. .test.jsx
3. .spec.js
4. .spec.tsx
5. .js or .tsx files in \_\_test\_\_ folder.

**CODE COVERAGE**

It helps us to understand how much our S/W code is tested.

1. **Statement Coverage:** how many statements have been executed.
2. **Branch Coverage:** how many branches of control statements have been executed.
3. **Function Coverage:** how many functions defined have been called.
4. **Line Coverage:** how many lines of code have been tested.

Following Script has to be added in package.json.

"jest": {  
 "collectCoverage": true,  
 "collectCoverageFrom": ["src/\*\*/\*.js", "src/\*\*/\*.jsx"],  
 "coverageReporters": ["lcov", "text-summary"]  
 }

**ASSERTIONS**

Assertions are carried out using global “expect” method.

Syntax 🡪 expect(value)

Value is the value that is produced by code.

Typically we will use “expect” along with matcher function to assert something about a value.

A matcher can optionally accept an argument which is the correct expected value.

Example: expect(textElement).toBeInDocument();

Some commonly used matchers

1. .toBe()
2. .toEqual()
3. toBeCloseTo()
4. toBeGreaterThan()
5. .toBeGreaterThanOrEqual()
6. .toBeLessThan()
7. .toBeLessThanOrEqual()
8. .toMatch() 🡪 String Matcher
9. .toContain() 🡪 array method
10. .toThrow() 🡪Exception Matchers

4-7 are number matchers.

To add custom matchers we need to use another library to dev dependency called Jest-Dom.

While writing the test we should be testing for:

1. Component Rendering.
2. Component Rendering With props.
3. Component Rendering in different states.
4. Component React to Events.

What we should not be testing for:

1. Third Party Code.
2. Code that is not important from user point of view.

**How a test case is Written**

Every test we write generally involves the following basic steps.

1. Render the component.
2. Find an element rendered by the component.
3. Assert against the element found in step 2, which will pass or fail the test.

* For Step1 we use render() method.
* For Step2 we are going to use RTL Queries.
* For Step3 we use expect() method and combine it with matcher function from jest or jest\_dom.

**RTL Queries**

RTL Queries to find:-

1. getBy
2. queryBy
3. findBy

RTL queries to find multiple elements on the page:

1. getAllBy...
2. queryAllBy...
3. findAllBy...

All the above mentioned RTL Queries can be suffixed with

1. Role
2. LabelText
3. PlaceHolderText
4. Text
5. DisplayValue
6. AltText
7. Title
8. TestId

**getBy... Queries**

getBy... class of queries return matching node for a query and throws a descriptive error if no elements match or if more than 1 match is found.

**getByRole**

Queries for elements with the given role. Role refers to the aria-role(Accessible Rich Internet Application). Which provides semantic meaning to content to ensure people using assertive technologies are able to use them.

By Default, many semantic elements in HTML have a role. Button element has a button role. Anchor element has link role. H1 to H6 has heading Role. Check Boxes have checkbox role. Radio Buttons have radio role.

If we are working with elements that do not have default role then “role” attribute can be used to desired role.

For Example: To use Anchor element as button in the navbar, we can add role= “button”.

Note:

For multiple elements with same role, getByRole will give an error. TO solve this we need to provide name option

For Example: const nameElement = screen.getByRole(“textbox”, { name: “Name”});

There are multiple role options:

1. Name
2. Level
3. Hidden
4. Selected
5. Checked
6. Pressed

**getByLabelText**

It will search for label that matches the given text, then find the element associated with that element.  
Ex: const nameElement = screen.getByLabelText(‘Name’);  
expect(nameElement).toBeInTheDocument();

For multiple elements with same label text, it will give an error. To resolve this we need to give selector options.

**getByPlaceholderText**

It will search for all elements with a placeholder attribute and find one that matches the given text.

Ex: const nameElement = screen.getByPlaceholderText(“FullName”);  
 expect(nameElement).toBeInTheDocument();

**getByText**

It will search for all elements that have a text node with text content matching the given text. Typically, we would use this to find paragraph, div or span elements.

<p>All fields are mandatory</p>  
const paraElement = screen.getByText(“All fields are mandatory”);  
expect(paraElement).toBeInTheDocument();

**getByDisplayValue**

<input type= “text” id = “name” placeholder= “Fullname” value = “Shivam” onChange={()=>{}}  
const nameElement = screen.getByDisplayValue(“Shivam”);  
expect(nameElement).toBeInTheDocument();

**getByAltText**

It will return the element that has the given “alt” text.

This method only supports elements which supports alt attribute, like <img>, <input> or custom HTML elements.

Const imgElement = screen.getByAltText(“Alternative Text”);

Expect(imgElement).toBeInTheDocument()

**getByTitle**

It returns the element which has the matching “title” attribute

<span title = “close”>X</span>

const closeElement = screen.getByTitle(“close”);

expect(closeElement).toBeInTheDocument();

**getByTestId**

It returns the element which has matching “data-test id” attribute.

<div data-test id= “custom-element”>Custom HTML Element</div>

const customElement = screen.getByTestId(“custom-element”);  
expect(customElement).toBeInTheDocument();

**Priority Order For Queries(High to Low)**

1. getByRole
2. getByLabelText
3. getByPlaceHolder
4. getByText
5. getByDisplayValue
6. getByAltText
7. getByTitle
8. getByTestId

**Query Multiple Elements**

It finds multiple elements in the DOM. getAllBy... returns an array of all matching nodes for a query, and throws an error if no matching elements are found.

1. getAllByRole
2. getAllByLabelText
3. getAllByPlaceholderText
4. getAllByText
5. getAllByDIsplayValue
6. getAllByAltText
7. getAllByTitle
8. getAllByTestId

e.g. describe(“Skills”, () => {  
 const skills = [“HTML”, “CSS”, “JavaScript”];  
 test(“renders a list of skills”, ()=>{  
 render(<Skills skills = {skills}/>);  
 const listOfElements = screen.getAllByRole(“listitem”);  
 expect(listOfElements).toHaveLength(3)  
 })

});

**TextMatch**

It represents a type which can be either a

1. String
2. Regex
3. Function

Text Match as a String

<div>Hello World</div>  
screen.getByText(‘Hello World’) // full string match.  
screen.getByText(‘llo Wo’,{exact:false}) // sub string match  
screen.getByText(‘hello world’, {exact:false}) //ignore case

Text Match as Regex

Screen.getByText(/World/) // substring match  
screen.getByText(/World/i) substring match, ignore case  
screen.getByText(/^hello world$/i) // full string match ignore case

Text Match – Custom Function

(content? String, element? Element|null) => Boolean  
screen.getByText((content)=>content.startsWith(“Hello”));

**QueryBy**

It returns the matching node for a query & return null if no element matches.

(IMP) Useful for asserting an element that is not present.

Throw an error if more than 1 element is found.

**QueryAllBy**

Return an array of all matching nodes for a query and return an empty array if no element matches.

Test(“Start learning button is not rendered”, ()=>{

render(<Skills skills={skills}/>);

const startLearningButton = screen.getByRole(“button”,{name: “Start Learning”});

expect(startLearningButton).not.toBeInTheDocument();

})

**FindBy**

getBy & getAllBy class of queries:- assert if elements are present in the DOM.

queryBy & queryAllBy class of queries:- assert if elements are not present in the DOM.

findBy & findAllBy class of queries:- assert if elements are not present in the DOM to begin but make into DOM after some time. (happens automatically after given period of time)

e.g. data fetched from server will be rendered only after few seconds.

Find By returns a promise which resolves when an element is found which matches the given array. The promise is rejected if no element is found or if more than 1 element is found after adefault timeout of 1000ms.

**FindAllBy**

It returns a promise which resolves to an array of element when any element is found which makes the given query. The promise is rejected if no elements are found after a default timeout of 1000ms.

Note: We can override the timeout time.

e.g. Test(“button is eventually displayed”, async()=> {

render(<Skills skills= {skills}/>);

const startLearningButton = await screen.findByRole(“button”, {name: “start learning”}, {timeout:2000});

expect(startLearningButton).toBeInTheDocument();

});

**Manual Queries**

We can use regular query selector DOM API to find elements.

Const {container} = render(<MyComponent/>);

Const foo = container.querySelector([‘data-foo’]= “bar”);

But this is not recommended & should be used when RTL queries fails to fulfill the desired result.

**Testing Play Ground**

This extension allows us to find best query to select elements while writing tests.

**User Interactions**

Example of user Interactions: Click using mouse or keyPress using a key board.

To Test User interactions we are going to use a library called “user-events”. It is a library which simulates user interactions by dispatching the events that would have happen if the interaction took place in the browser.

**Fire Event v/s User-Event**

Fire Event is a method from RTL which is used to dispatch DOM events. However user-events are more superior. User Events simulates full interactions, which may fire multiple events and do additional checks along the way.

User Events library is installed by default.

**Pointer Interactions**

Test(“renders a count of 1 after clicking the increment button”, async ()=>{  
 User.setup() //need to call if we are dealing with user interactions  
 Render(<Counter/>);  
 Const incrementButton = screen.getByRole(“button”,{name: “interaction”})  
 await user.click(incrementButton);  
 const countElement = screen.getByRole(“heading”);  
 expect(countElement).toHaveTextContent(“1”);  
 })

Other Pointer Interactions are as follows:

1. Click()
2. dblClick()
3. tripleClick()
4. hover()
5. unhover()

**KeyBoard Interactions**

Test(“renders a count of 10 after clicking set button”, async () => {

User.setup();  
 render(<Counter/>);  
 const amountInput = screen.getByRole(“spinButton”);  
 await user.type(amountInput, “10”);  
 expect(amountInput).toHaveValue(10);  
 const setButton = screen.getByRole(“button”,{name: “set”});  
 await user.click(setButton);  
 const countElement = screen.getByRole(“heading”);  
 expect(countElement).toHaveTextContent(“10”);

})

Other Keyboard Interactions

1. User.type()
2. User.tab()
3. User.clear()
4. User.selectOptions()
5. User.deselectOptions()
6. User.upload()
7. User.copy()
8. User.cut()
9. User.paste()

**Testing Custom React Hooks**

RTL provides “renderHook” function to test custom Hooks.

Import {renderHook} from “@testing-library/react”;  
import {useCounter} from “./useCounter”;  
describe(“useCounter”, () => {  
 test(“should render initial count”, () => {  
 const {result} = renderHook(useCounter);  
 expect(result.current.count).toBe(0);  
 });  
 test(“should accept and render the same initial count”, () => {  
 const {result} = renderHook(useCounter, () => {  
 initialProps: {initialCount:10}  
 });  
 expect(result.current.count).toBe(10);  
 })  
});

**Act Utility**

Import {Act} from “@testing-library/react”;

When testing custom hooks, specifically code that causes state updates, react testing library cannot wrap them with act utility function. We have to manually import it & wrap code that causes state updates.

**Mocking Functions**

Test(“handlers are called”, async () => {  
 user.setup();  
 const incrementalHandler = jest.fn();  
 const decrementHandler = jest.fn();  
 render(  
 <CounterTwo  
 count = {0}  
 handleIncrement = {incrementHandler}  
 decrementHandler = {decrementHandler}  
 />  
 )  
 const incrementButton = screen.getByRole(“Button”,{name: “Increment”});  
 const decrementButton = screen.getByRole(“Button”, {name: “decrement”});  
 await user.click(incrementButton);  
 await user.click(decrementButton);  
 expect(incrementHandler).toHaveBeenCalledTimes(1);  
 expect(decrementHandler).toHaveBeenCalledTimes(1);  
});

**Mocking HTTP Requests**

For Mocking HTTP requests we use MockServiceWorker (MSW) library.

**Static Analysis Testing**

It is a process of verifying that whether our code meets certain expectations without actually running it.

With static testing we can:

1. Ensure consistent style & formatting
2. Check for common mistake & possible bugs.
3. Limit the complexity of code
4. Verify Type consistency

Static Analysis Testing Tools are:

* TypeScript
* Prettier
* Lint-staged
* EsLint
* Husky