**React Testing Library tutorial**, we will go through all the steps necessary to unit test and integration test your React components with confidence.

Jest vs React Testing Library

**React Testing Library is not an alternative to Jest**, because they need each other and every one of them has a clear task.

The .test.js / .spec.js files (or the **tests** folders) can be located at any depth under the src top level folder.

For larger projects it’s recommends to put all .test.js / .spec.js files in \_\_tests\_\_ folders under src folder

Jest - the most popular **testing framework** out there for JavaScript applications.

React uses a component-based structure, meaning a single change in one component could potentially affect others. This interdependence is why testing is so important in React. In case a component fails the test, you will be notified immediately, allowing for quick fixes and minimizing work stoppage.

In modern React, developers will not get around Jest for testing.

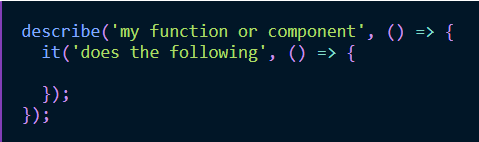
Jest being a **test runner** – means which you can run all your tests( .test.js) with: npm run test once you have set up your package.json with

"scripts": {

"test": "jest"

}

Jest is not just a **test runner** (program that runs tests). It also gives useful following **functions** to help us **write structured tests**.



Here,

describe-block is the **test suite**

it-block (which also can be named test instead of it) is the **test case**.

A test suite can have multiple test cases and a test case doesn't have to be in a test suite.

What you put into the test cases are called **assertions** (e.g. expect in Jest) which either turn out to be successful (green) or erroneous (red).

Here we have two assertions,

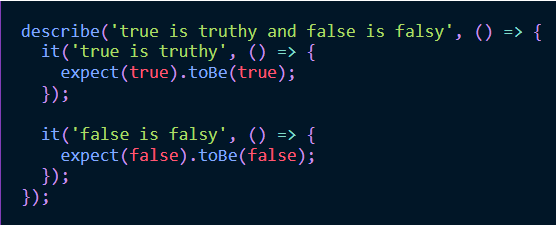


Fig: testfile.test.js file

When we run test command (i.e., npm run test) , Jest’s test runner matches all files with ***.test.js*** suffix by default.

NOTE:

1. You could configure this matching pattern and others things in a custom Jest configuration file.
2. If you are using create-react-app, Jest (and React Testing Library) comes by default with the installation.
3. If you are using a [custom React setup](https://www.robinwieruch.de/minimal-react-webpack-babel-setup/), you need to install and set up Jest (and React Testing Library) yourself.

[Jest](https://facebook.github.io/jest/) got introduced by Facebook for testing JavaScript and especially React applications. It's one of the most popular ways to test React components nowadays. Since it comes with its own test runner, you can simply call Jest from the command line to run all your tests. All your tests are defined as test suites (e.g. describe-block) and test cases (e.g. it-block or test-block).

NOTE: The Jest setup allows you to add optional configuration, to define custom npm scripts to run your Jest tests.

Jest comes with a rich API for test assertions (e.g. true to equal true). The tutorial (Jest testing in React setup)will show you how to use these test assertions for your React components and JavaScript functions. Also you will learn about Snapshot Tests to test your React components.

Webpack bundles all our JavaScript source code files into one bundle

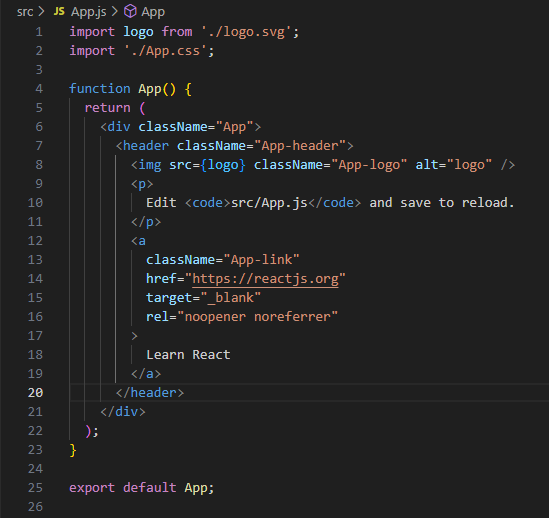
Babel enables us to use recent JavaScript features that are not supported by many browsers yet. That's why Babel is also needed for React.  
  
Jest is a test runner, which gives you the ability to run tests with Jest from the command line. In addition, Jest offers you functions for test suites, test cases, and assertions. Of course the frameworks offers more than this (e.g. spies, mocks, stubs, etc.); but essentially that's everything needed for now to understand why we need Jest in the first place.

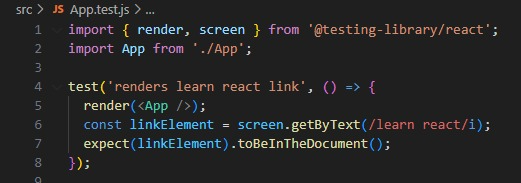
React Testing Library, in contrast to Jest, is one of the testing libraries to test React components. Another popular one in this category is Enzyme as mentioned earlier.

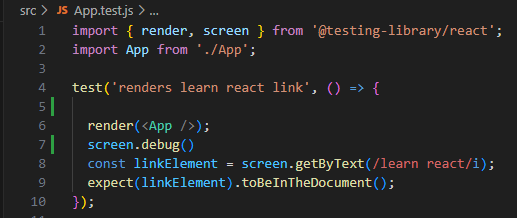
**Let see how to use React Testing Library for testing React components:**

**NOTE:** Following code content taken from React Project built by the command **npx create-react-app <project-name>**

Rendered a React component in a test with React Testing Library(RTL).

1. App component from a src/App.js file
2. App.test.js

  
Here RTL's **render( )** function takes any JSX as argument to render it as output. Afterward, you should have access to the React component in your test. To convince yourself that it's there, you can use RTL's **screen.debug( )** function:



'getByText()' search function from React Testing Library used  to find an element

that contains the text "learn react" — using a regular expression instead of a plain string.

**/learn react/:** This regular expression pattern looks for the exact phrase "learn react" in the text content of the DOM element.

**i (at the end):** This is called the "case-insensitive" flag.

Example:

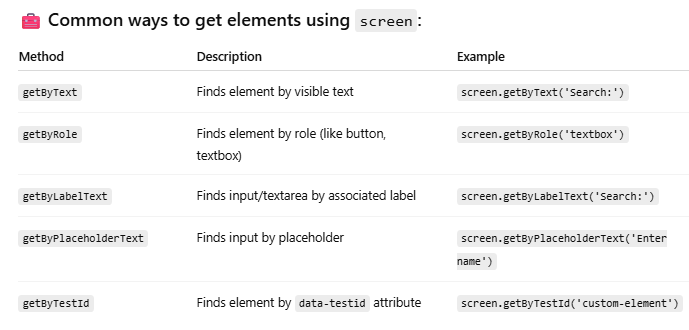
screen.getByText("Learn React"); // Exact match, case-sensitive

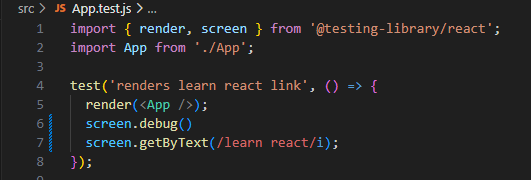
screen.getByText(/learn react/i); // Flexible match, case-insensitive

1. After running your test (**Ex**: npm test App.test.js)on the command line, you should see the HTML output of your **App component.** 
2. Whenever you write a test for a component with React Testing library, you can render the component first and then debug what's visible for RTL's renderer in the test.
3. This way, you can write your test with more confidence.

After you have rendered your React component(s), React Testing Library offers you different search functions to grab elements.

1. In **React Testing Library (RTL)**, the screen object gives you access to the **DOM elements** rendered by the render() function. It's used to query elements like buttons, inputs, text, etc.





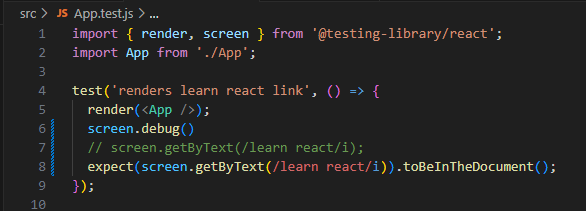
NOTE:

1. Always use RTL's debug function if you don't really know what's the rendered output of RTL's render function. After you know about the HTML structure, you can start to select elements with RTL's screen object's functions. The selected element can then be used for user interactions or assertions.
2. screen.getByText(/learn react/i); --- **Implicit Assertion** --- Not recommended

Because If '/learn react/I' element **exists**, test **passes** silently even if you **don't write expect(...)**. If element **does not exist**, test **fails(throws an error)**.

So, chala mandi expect() rayakunda getByText() ni vadataru just to confirm the element exists.

**best practice**: always use **explicit assertion with expect(...)** for better readability and clarity.

1. We will do an recommended **Explicit assertion (i.e,** expect(...).toBeInTheDocument()) that checks whether the element is in the DOM:  
   

conveniently getByText throws an error by default if the element cannot be found. This is useful for giving you a hint while writing the test that the selected element isn't there in the first place.   
NOTE:

The getByText function accepts a string or regular expression as argument, whereas a string argument is used for the exact match, a regular expression can be used for a partial match which is often more convenient.

String as argument:

expect(screen.getByText('Search:')).toBeInTheDocument();

Regular Expression as argument:

expect(screen.getByText(/Search/)).toBeInTheDocument();

The getByText function is only one of many types of **search functions** in React Testing Library. Let's see what else is there.

1. You have learned about getByText where Text is one of several search types.   
   While Text is often the common way to select elements with React Testing Library, another option is Role with getByRole.  
   The getByRole function is usually used to retrieve elements by [aria-label attributes](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques/Using_the_aria-label_attribute).
2. getByRole() mainly DOM elements ni provide chesins accessible roles base chesi find cheyyadaniki. Idi accessibility standards ni follow chestundi
3. role should be (string): Ikkada role ante HTML element yokka semantic role. For example:"button","textbox","heading","link","checkbox","radio".

Examples:

**Get a textbox by role**

render(<input type="text" placeholder="Enter name" />);

const input = screen.getByRole('textbox');

**Get a button by role**

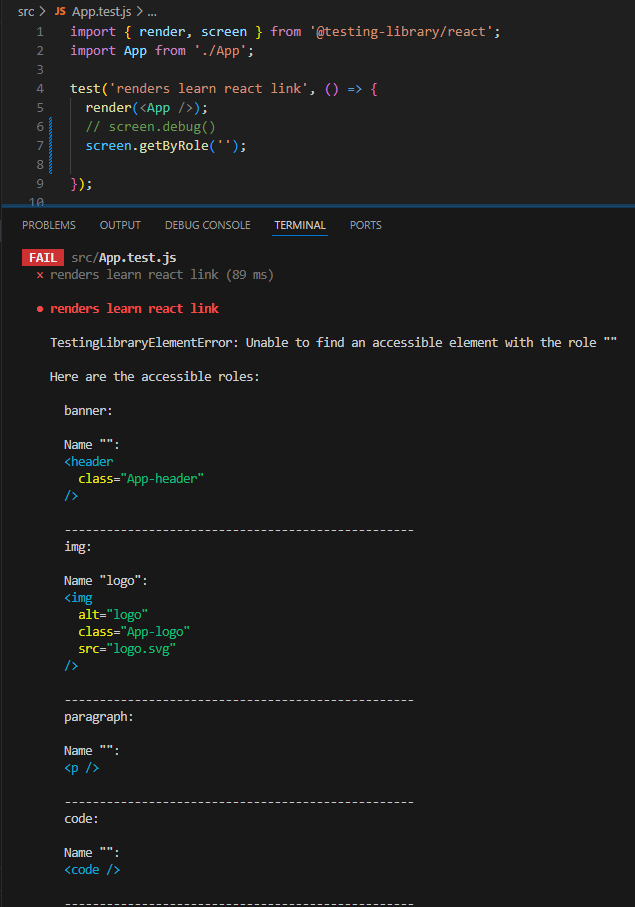
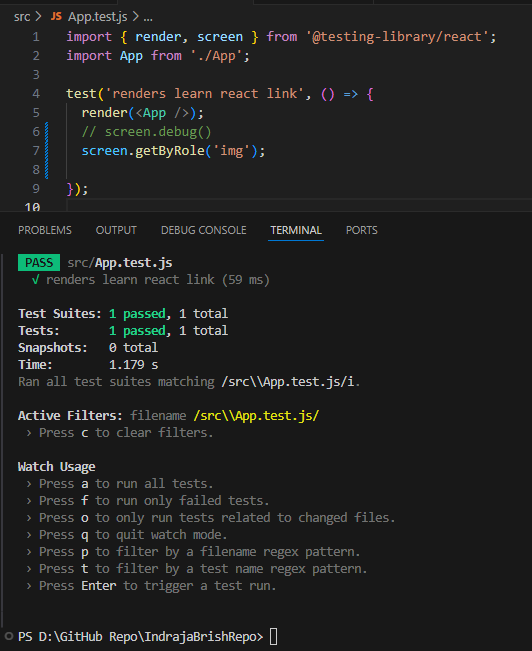
render(<button>Submit</button>);

const button = screen.getByRole('button', { name: 'Submit' });

**Heading with level**

render(<h2>Welcome</h2>);

const heading = screen.getByRole('heading', { name: 'Welcome', level: 2 });

1. However, there are also [implicit roles on HTML elements](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles) -- like **button** for a <button> element.  
   Thus you can select elements not only by visible text, but also by their accessibility role with React Testing Library.  
   The neat thing about getByRole: it shows all the selectable roles if you provide a role that isn't available in the rendered component's HTML:  
     
   

So, it isn't necessary to assign aria roles to HTML elements explicitly for the sake of testing, because the DOM already has implicit roles attached to HTML elements. This is what makes getByRole a strong contender to the getByText search function from React Testing Library.

There are other search types which are more element specific:

* **LabelText:** getByLabelText: <label for="search" />
* **PlaceholderText:** getByPlaceholderText: <input placeholder="Search" />
* **AltText:** getByAltText: <img alt="profile" />
* **DisplayValue:** getByDisplayValue: <input value="JavaScript" />

React Testing Library: Search Variants

In contrast to search types, there exist search variants as well.

One of the search variants in React Testing Library is getBy which is used for getByText or getByRole.

getBy is also the search variant which is used by default when testing React components.  
Two other search variants are queryBy and findBy; which both can get extended by the same search types that getBy has access to.

For example,

*queryBy* with all its search types:

* queryByText
* queryByRole
* queryByLabelText
* queryByPlaceholderText
* queryByAltText
* queryByDisplayValue

*findBy* with all its search types:

* findByText
* findByRole
* findByLabelText
* findByPlaceholderText
* findByAltText
* findByDisplayValue

When to use getBy and queryBy and findBy.

import { render, screen } from '@testing-library/react';

import App from './App';

test('renders App component', () => {

render(<App />);

screen.debug();

*// fails*

expect(screen.getByText(/Search/)).toBeNull();

});

});

Remember,  
getByText() used to check if the element must be present, if element not found it throws an error.

Here, expect(screen.getByText(/Searches for JavaScript/)).toBeNull();

You're using getByText() to **check if an element is NOT in the document**. But:

* getByText() **throws an error** when the element isn't found, so the test fails *before* reaching .toBeNull().
* That’s why your test is failing — the test crashes instead of passing.

Therefore, In order to assert elements which aren't there, correct approach is using queryBy instead getBy :

import { render, screen } from '@testing-library/react';

import App from './App';

test('renders App component', () => {

render(<App />);

screen.debug();

*// fails*

expect(screen.queryByText(/Search/)).toBeNull();

});

});

 queryByText(...) searches for the element.

 If it's **not found**, it **returns null** (no error).

 So now toBeNull() runs and passes ✅

So, every time you are asserting that an element isn't there, use queryBy.

### **🧠 Rule of Thumb**

* Use **getBy** when you're asserting that something **should exist**.
* Use **queryBy** when you're asserting that something **should not exist**.

### **When to use findBy?**

The findBy search variant is used for asynchronous elements which will be there eventually.

Lets take sample App.js code from https://www.robinwieruch.de/react-testing-library/

import \* as React from 'react';

function App() {

const [search, setSearch] = React.useState('');

const [user, setUser] = React.useState(null);

function handleChange(event) {

setSearch(event.target.value);

}

return (

<div>

<Search *value*={search} *onChange*={handleChange}>

Search:

</Search>

<p>Searches for {search ? search : '...'}</p>

</div>

);

}

function Search({ value, onChange, children }) {

return (

<div>

<label *htmlFor*="search">{children}</label>

<input

*id*="search"

*type*="text"

*value*={value}

*onChange*={onChange}

/>

</div>

);

}

export default App;

Above App.js test file is like

import \* as React from 'react';

import { render, screen } from '@testing-library/react';

import App from './App';

describe('App', () => {

it('renders App component', () => {

render(<App />);

screen.debug();

});

});

Fig: App.test.js  
  
If you start npm test App.test.js of your App component again, you should see the following output from the **screen.debug()** function:

<body>

<div>

<div>

<div>

<label

*for*="search"

>

Search:

</label>

<input

*id*="search"

*type*="text"

*value*=""

/>

</div>

<p>

Searches for

...

</p>

</div>

</div>

</body>

Changes made to the above App.js code to make Suitable Scenario to explain "findBy" search varient of React testing Library

APP.js Code Scenario :

After APP.js initial render, the App component fetches a user from a simulated API.

The API returns a JavaScript promise which immediately resolves with a user object, and

the component stores the user from the promise in the component's state.

The component updates and re-renders; and afterward the conditional rendering should render "Signed in as" after the component update:

**App.js:**  
import \* as React from 'react';

const getUser = () => {

  return Promise.resolve({ id: '1', name: 'Robin' });

};

function App() {

  const [search, setSearch] = React.useState('');

  const [user, setUser] = React.useState(null);

  React.useEffect(() => {

    const loadUser = async () => {

      const user = await getUser();

      setUser(user);

    };

    loadUser();

  }, []);

  function handleChange(event) {

    setSearch(event.target.value);

  }

  return (

    <div>

      {user ? <p>Signed in as {user.name}</p> : null}

      <Search value={search} onChange={handleChange}>

        Search:

      </Search>

      <p>Searches for {search ? search : '...'}</p>

    </div>

  );

}

function Search({ value, onChange, children }) {

  return (

    <div>

      <label htmlFor="search">{children}</label>

      <input

        id="search"

        type="text"

        value={value}

        onChange={onChange}

      />

    </div>

  );

}

export default App;

If we want to test the component over the stretch of its first render to its second render due to the resolved promise, we have to write an async test, because we have to wait for the promise to resolve asynchronously.

import \* as React from 'react';

import { render, screen } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', async () => {

    render(<App />);

    expect(screen.queryByText(/Signed in as/)).toBeNull();

    expect(await screen.findByText(/Signed in as/)).toBeInTheDocument();

  });

});

After its initial render, we assert that the "Signed in as" text is not there by using the queryBy instead of the getBy search variant. Then we await the new element to be found, and it will be found eventually when the promise resolves and the component re-renders again.

If you don't believe that this actually works, include these two debug functions and verify their outputs on the command line:

import \* as React from 'react';

import { render, screen } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', async () => {

    render(<App />);

    expect(screen.queryByText(/Signed in as/)).toBeNull();

    screen.debug();

    expect(await screen.findByText(/Signed in as/)).toBeInTheDocument();

    screen.debug();

  });

});

### **What about multiple elements?**

All search variants can be extended with the *All* word:

* getAllBy
* queryAllBy
* findAllBy

Whereas all of them return an array of elements and can be associated with the search types again.

### **Assertive Functions**

* Usually all these assertive functions origin from Jest/Vitest  
  However, React Testing Library extends this API with its own assertive functions like toBeInTheDocument. All these assertive functions come in an [extra package](https://github.com/testing-library/jest-dom) which are already set up for you when using create-react-app.  
  toBeDisabled
* toBeEnabled
* toBeEmpty
* toBeEmptyDOMElement
* toBeInTheDocument
* toBeInvalid
* toBeRequired
* toBeValid
* toBeVisible
* toContainElement
* toContainHTML
* toHaveAttribute
* toHaveClass
* toHaveFocus
* toHaveFormValues
* toHaveStyle
* toHaveTextContent
* toHaveValue
* toHaveDisplayValue
* toBeChecked
* toBePartiallyChecked
* toHaveDescription

## React Testing Library: Fire Event

So far, we've only tested whether an element rendered (or not) in a React component with getBy (and queryBy) and whether the re-rendered React component has a desired element (findBy). 

What about actual user interactions?

If a user types into an input field, the component may re-render, and the new value should be displayed (or used somewhere).

We can use RTL's **fireEvent** and **waitFor** functions to simulate interactions of an end user.

import \* as React from 'react';

import { render, screen, fireEvent } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', () => {

    render(<App />);

    screen.debug();

    fireEvent.change(screen.getByRole('textbox'), {

      target: { value: 'JavaScript' },

    });

    screen.debug();

  });

});

Here,

The fireEvent function takes an element (here the input field by textbox role) and an event (here an event which has the value "JavaScript").

The debug function's output should show the HTML structure before and after the event.

And you should see that the new value of the input field (here "JavaScript") gets rendered appropriately.

### **Important:** Our App component is involved in an asynchronous task; hence you may see the following warning showing up: "Warning: An update to App inside a test was not wrapped in act(...)." because **your App component updates state asynchronously** using useEffect() (due to the async getUser() function), and React Testing Library expects you to **wait** for the effects to finish **before making assertions or doing interactions** like fireEvent. **✅ How to Fix It Using async/await (Without Manually Using act())**

You can **wait for the user to appear on the screen** using findByText (which automatically handles act() internally), like this:  
import \* as React from 'react';

import { render, screen, fireEvent } from '@testing-library/react';

import App from './App';

describe('App', () => {

  //Async

  it('renders App component', async () => {

    render(<App />);

    // Wait for the async user data to load (this handles the act() warning)

    await screen.findByText(/Signed in as/);

    screen.debug();

    // Now simulate typing in the input

    fireEvent.change(screen.getByRole('textbox'), {

      target: { value: 'JavaScript' },

    });

    screen.debug();

  });

});

### 💡 Explanation

* findByText() is async and waits for the DOM to reflect changes caused by useEffect() (React state updates).
* This avoids the need for act() blocks and ensures the test is properly synced with React's rendering.

Now, we can make the assertions from before and after the event:  
import \* as React from 'react';

import { render, screen, fireEvent } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', async () => {

    render(<App />);

    // wait for the user to resolve

    await screen.findByText(/Signed in as/);

    //assertion

    expect(screen.queryByText(/Searches for JavaScript/)).toBeNull();

    //assertion

    fireEvent.change(screen.getByRole('textbox'), {

      target: { value: 'JavaScript' },

    });

    expect(screen.getByText(/Searches for JavaScript/)).toBeInTheDocument();

  });

});

As alternative, we can also literally wait for an asynchronous update to happen with React Testing Library's waitFor function:  
// In this section,

//       we have used the queryBy search variant to check whether the element isn't there before the event and

//                    the getBy search variant to check whether the element there after the event.

import \* as React from 'react';

// waitFor() is a utility from React Testing Library used to wait for asynchronous changes in the DOM to complete before running assertions.

import { render, screen, fireEvent, waitFor } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', () => {

    render(<App />);

    expect(screen.queryByText(/Searches for JavaScript/)).toBeNull();

    fireEvent.change(screen.getByRole('textbox'), {

      target: { value: 'JavaScript' },

    });

    // Means: “Wait until the text 'Searches for JavaScript appears' in the document.”

// This is necessary because: React state updates (like after a fireEvent) may not reflect immediately in the DOM. Without waitFor(), the assertion getByText might run before the DOM updates, causing test flakiness or failure.

    waitFor(() =>

      expect(

        screen.getByText(/Searches for JavaScript/)

      ).toBeInTheDocument()

    );

  });

});

NOTE: waitFor() is asynchronous, so it returns a Promise. If you don’t await it, your test might finish before the check actually runs.

import \* as React from 'react';

import { render, screen, fireEvent, waitFor } from '@testing-library/react';

import App from './App';

describe('App', () => {

  it('renders App component', async () => {

    render(<App />);

    expect(screen.queryByText(/Searches for JavaScript/)).toBeNull();

    fireEvent.change(screen.getByRole('textbox'), {

      target: { value: 'JavaScript' },

    });

    // Means: “Wait until the text 'Searches for JavaScript appears' in the document.”

    await waitFor(() =>

      expect(

        screen.getByText(/Searches for JavaScript/)

      ).toBeInTheDocument()

    );

  });

});

### **React Testing Library: User Event**

React Testing Library comes with an extended user event library which builds up on top of the fireEvent API.  
Previously we have used fireEvent to trigger user interactions; this time we will use userEvent as replacement, because the userEvent API mimics the actual browser behavior more closely than the fireEvent API.

For example, a fireEvent.change() triggers only a change event whereas userEvent.type triggers a change event, but also keyDown, keyPress, and keyUp events.

అంటే మనం అక్షరాలు టైప్ చేసినట్టు simulate చేయడం కాకుండా, ఒక్కసారిగా value మారిపోయినట్టు చూపుతుంది.

So, fireEvent.change(inputElement, { target: { value: 'Hello' } });

ఇది basically: "Input value మారిపోయింది అని React కు చెప్పటం".

userEvent.type() : ఇది **keyboard typing ను simulate చేస్తుంది.** ఇది **keyDown → keyPress → input → change → keyUp** అనే మొత్తం order లో events simulate చేస్తుంది.

fireEvent.change() – Sync behaviour

userEvent.type() – Async behaviour  
import \* as React from 'react';

import { render, screen } from '@testing-library/react';

import userEvent from '@testing-library/user-event';

import App from './App';

describe('App', () => {

  it('renders App component', async () => {

    render(<App />);

    // wait for the user to resolve

    await screen.findByText(/Signed in as/);

    expect(screen.queryByText(/Searches for JavaScript/)).toBeNull();

    await userEvent.type(screen.getByRole('textbox'), 'JavaScript');

    expect(

      screen.getByText(/Searches for JavaScript/)

    ).toBeInTheDocument();

  });

});