Test Writing Process:

1. Pick one out Component to test all by itself.
2. Make a test file for the component if one does not exist
3. Decide what the important parts of the Component are
4. Write a test to make sure each parts works as expected
5. Run tests at the command line.

Test Code Format:

* First Import the Packages of jest
* Second import the component which we want to test
* Use a test function method to test the case

Sample:

Import {render,screen} from ‘@testing-library/react; package import

Import User from ‘@testing-library/user-event’; package import

Import userForm from ‘./useForm’; component path

test(‘statement’,method call);

test('it shows two input and button',()=>{

    //render the component

    render(<UserForm/>)

    //Manipulate the component or Find an element in it

    const inputs=screen.getAllByRole('textbox');

    const button=screen.getByRole('button');

    //Assertion-Make sure the component is doing what we expect to do

    expect(inputs).toHaveLength(2);

    expect(button).toBeInTheDocument();

})

ARIA Role:

1. ARIA Roles clarify the purpose of an HTML element
2. Traditionally used by screen readers-softwares to help people understanfd the content on the screen
3. Many HTML elements have an ‘implicit ’ ,or automatically assigned, role.
4. Elements can be assigned manually assigned a role.Even trained engineer do this incorrectly.

Heading =>h1 to h6 tags

List => ul and li

Button =>button

Link => **a** tag

Textbox => input , type=”text”

Matchers from React Testing Library:

1. Expect(element).toBeInTheDocument() => Make sure element is present on the page
2. Expect(element).toBeEnabled() =>Makes sure an element (like an input) is not disabled
3. Expect(element).toHaveClass() =>Makes sure an element has a class name
4. Expect(element).toHaveTextContent() =>Make sure an element has some particular text
5. Expect(element).toHaveClass() => Make sure an input,select or textarea has a value.

User.click(element) =>Simulates clicking on the provided element

User.keyboard(‘asdf’) =>Simulates typing ‘asdf’

User.keyboard(‘{Enter}’) =>Simulates pressing the Enter key.

Sample Test-2

test('it calls onUserAdd when the form is submitted', () => {

    //NOT THE BEST IMPLEMENTATION

    const argList=[];

    const callback=(...args) => {

        argList.push(args);

    }

    //Try to render the component

    render(<UserForm onUserAdd={callback}/>)

    //Find the two inputs

// const input = screen.getAllByRole('textbox') //but here i have a two input that for name and email soi destructure that

const [nameInput,emailInput] = screen.getAllByRole('textbox');

    //Simulates typing in a name

user.click(nameInput);

user.keyboard('santhosh');

    //Simulates typing in a email

    user.click(emailInput);

    user.keyboard('santhosh@gmail.com');

    //Find the button

const button =screen.getByRole('button');

    //Simulating clicking the button

    user.click(button);

    //Assertion to make sure the "onUserAdd" gets called with name and email

    expect(argList).toHaveLength(1);

    expect(argList[0][0]).toEqual({name:'santhosh',email:'santhosh@gmail.com'});

})

MOCK FUNCTION:

1. Mock meaning Not Real
2. It is a fake function that doesn’t do anything
3. Record whenever its gets called and the arguments with that called
4. Used very oftenwhen we need to make sure a components calls a callback

**Line by Line Explanation:**

test('it calls onUserAdd when the form is submitted', () => {

This is a test case that verifies whether the onUserAdd function is called with the correct arguments when the form is submitted.

const mock = jest.fn();

Here, we create a mock function using the jest.fn() method. A mock function is a special kind of function that allows us to spy on how it is used, including how many times it is called and with what arguments. We will use this mock function to verify that onUserAdd is called with the correct arguments.

render(<UserForm onUserAdd={mock} />)

This renders the UserForm component and passes the mock function as the onUserAdd prop. This means that instead of using the actual onUserAdd function, the mock function will be called when the form is submitted.

const [nameInput, emailInput] = screen.getAllByRole('textbox');

This finds the two text input fields in the rendered UserForm component and assigns them to the nameInput and emailInput variables using array destructuring.

user.click(nameInput);

user.keyboard('santhosh');

This simulates a user clicking on the nameInput text field and typing the value "santhosh".

expect(mock).toHaveBeenCalled();

This verifies that the mock function has been called at least once. If the onUserAdd function is called with the correct arguments, the mock function should be called at least once as well.

expect(mock).toHaveBeenCalledWith({name:'santhosh',email:'santhosh@gmail.com'});

This verifies that the mock function has been called with the correct arguments, namely an object with name set to "santhosh" and email set to "[santhosh@gmail.com](mailto:santhosh@gmail.com)". If onUserAdd is called with the correct arguments, the mock function should be called with the same arguments as well.

Element and Role for table:

* Thead => rowgroup
* Tbody => rowgroup
* Tr => row
* Th => columnheader
* Td =>cell

Difference between Query selector and Data-testid:

The main difference between **container.querySelector** and **data-testid** is the approach to element selection. **container.querySelector** is more flexible and can handle complex queries, but it is more susceptible to breaking if the DOM structure changes. **data-testid** is more explicit and less likely to break, but it requires you to add extra attributes to your HTML.

To find the email and username in the table-cell and also in webpage:

test('should render name and email',()=>{

const users=[

    {name: 'santhosh',email: 'santhosh@gmail.com'},

    {name: 'ranjith',email: 'ranjith@gmail.com'}

];

render(<UserList users={users} />)

for(let user of users) {

    const name=screen.getByRole('cell', {name:user.name})

    const email=screen.getByRole('cell', {name:user.email})

    expect(name).toBeInTheDocument();

    expect(email).toBeInTheDocument();

}

});