**Need of Testing:**

To verify whether application behaves as intended

To stay resilient against unwanted behavior when changes are made

If can be automated should be efficient on the long term

**Types of tests:**

Unit test

Integration test

E2E test

**Technologies used:**

[React Testing Library](https://testing-library.com/docs/react-testing-library/intro/)

[Jest](https://jestjs.io/docs/getting-started) as test runner

[Cypress](https://learn.cypress.io/)

**Unit test:**

**Need**:

As the application grows, it might be difficult to test all the scenarios in the application and you might miss something. Even a small change might break the application if all the major functionality is not tested properly.

That's why writing unit test cases covering all those scenarios which you're manually going through as a user is recommended.

So by executing just a single command you will be able to know if something is broken in your application or if some test is failing.

**React Testing Library:**

The React [Testing Library](https://testing-library.com/) has a set of packages that help you test UI components in a user-centric way. This means it tests based on how the user interacts with the various elements displayed on the page.

So what happens is when the user clicks any button or types in any of the input textboxes, that interaction is tested using this testing library.

So instead of the user doing this testing manually (which takes a lot of time, and the user might miss testing certain scenarios when the application grows), the testing is done by writing unit test cases and executing them by just a single command.

**Implementation:**

React Testing library comes preinstalled and preconfigured with create react app installation so we can jump in directly to the implementation part

To test a component the recommended naming standard for the test file is [component Name].test.js.

Beside that always start with a failing test so that you can always ensure the tests work.

Here we test according to the component render.

Suppose we have a component which is responsible for the login part with a condition that the log in button should only enabled once the loginid is entered, Lets call it Login component

The First test will ensure that the button is not enabled on render

Let look at the test file

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

import Login from "./Login"

import { BrowserRouter, Routes, Route } from 'react-router-dom';

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

test('On initial render, the login button should be disabled', () => {

    render(

        <BrowserRouter>

            <Routes>

            <Route path="\*" element= {<Login/>}/>

            </Routes>

        </BrowserRouter>

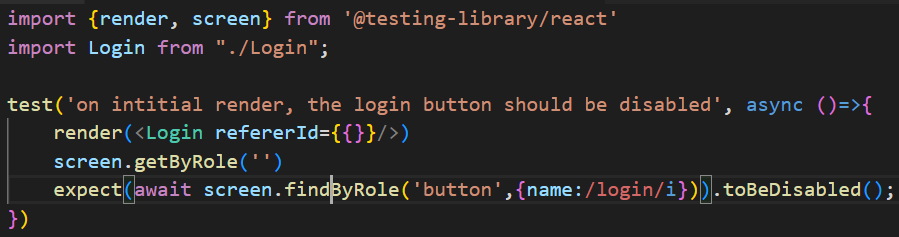
    )

    screen.getByRole('button')

    expect(screen.getByRole('button', { Name: /loginid/i })).toBeDisabled();

})

Here if the component performs asynchronous task we should use findByRole()



Now lets write another test for the button to get enabled once the login id is entered

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

import Login from "./Login"

import { BrowserRouter, Routes, Route } from 'react-router-dom';

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

test('On initial render, the login button should be disabled', () => {

    render(

        <BrowserRouter>

            <Routes>

            <Route path="\*" element= {<Login/>}/>

            </Routes>

        </BrowserRouter>

    )

    screen.getByRole('button')

    expect(screen.getByRole('button', { Name: /loginid/i })).toBeDisabled();

})

test('when loginid is entered, the login button should be enabled',()=>{

    render(

        <BrowserRouter>

            <Routes>

            <Route path="\*" element= {<Login/>}/>

            </Routes>

        </BrowserRouter>

    )

    userEvent.type(screen.getByPlaceholderText(/loginid/i),"user");

    expect(screen.getByRole('button', { Name: /loginid/i })).toBeEnabled();

})

Refer for [getByPlaceholderText](https://testing-library.com/docs/queries/byplaceholdertext/) and [userEvent](https://testing-library.com/docs/ecosystem-user-event/)

**Integration test**

Integration testing should resemble the user flow the above unit tests combined can emulate user login action.Thus the integration test can be written by combining unit tests

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

import {userEvent} from '@testing-library/userEvent'

import Login from "./Login";

test('example integration test',()=>{

    render(<Login refererId={{}}/>)

    expect(screen.getByRole('button',{name:/login/i})).toBeDisabled();

userEvent.type(screen.getByPlaceholderText(/loginid/i),"user1");

userEvent.type(screen.getByPlaceholderText(/password/i),"password");

expect(screen.getByRole('button',{name:/login/i})).toBeEnabled();

})

**End to End test**

They are the most effective type of test for testing out things that really matter like the high value features.

We will be using Cypress to perform E2E

**Setup**

install Cypress and @testing-library/cypress

npm install --save-dev cypress @testing-library/cypress

Now run

npm run cypress open

This will generate a folder named Cypress which will have the integration folder which will contain the test files

Now we add the react testing library Cypress commands for that navigate to Cypress>support>commands.js

In this file add the following line and save

import '@testing-library/cypress/add-commands'

**Implementation**

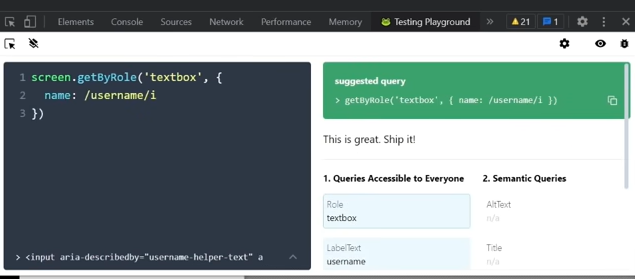
We will create a test file named blogs.cy.js Cypress>e2e folder and inside it we will write the test

We can use testing [Playground chrome extention](https://chrome.google.com/webstore/detail/testing-playground/hejbmebodbijjdhflfknehhcgaklhano?hl=en) which suggests a query when we hover over an element

Navigate to the login screen by going to the ‘/’ route and open up the Playground extention in chrome dev tools

Click on the element selector and now hover over the element you need to test and click on it

This will suggest a query as shown in the image below



Copy the suggested query and paste it to blog\_spec.js file

Now we need to make some changes in the copied value as follows

We need to use findByRole instead of getByRole here and we can emulate user actions by using various functions like type(‘string’) for typing,click()for clicking action

describe('blogs',()=>{

    it('user can login',()=>{

        // login

        cy.visit('localhost:3000/login') //visit root of our app

        cy.findByRole('textbox', { name: /username/i }).type('johndoe');

        cy.findByLabelText(/password/i).type('password');

        cy.findByRole('button', { name: /sign in/i }).click();

        //Should be able to view all the blogs

    })

})

Open up the cypress window and you will be able to see the newly added test and option to execute the test

Now the test runs and opens up a new window where you will see the behavior being triggered by cypress

**Conclusion**

Always test the most important feature

High value Features should be **End to End tested**

Writing Unit/Integration test cases for the following is recommended

Edge cases in high value features

Things that are easy to break

Basic react component testing like user interactions, conditional rendering, Utils/Hooks