* RTL vs Jest (Testing Library).....

1) Jest

  * Testing Framework: Jest is a JavaScript testing framework developed by Facebook.
    It's designed to be a complete solution for testing JavaScript code, including React applications.

  * General Focuses on JavaScript testing....

  * Emphasis on functional and unit tests....

  * Built-in mocking capabilities...

  * Provides a wide range of utilities for testing JavaScript code

2) RTL(React Testing Library):-

  * Testing Utilities: RTL, on the other hand, is not a testing framework but a set of utilities for testing React components
    It encourages testing components in a way that closely resembles how users interact with the application.

  * Focuses on React Component testing....

  * Emphasis on user-centric and integration tests...

  * Supports manual mocking; can be used with Jest....

  * Focuses more on component interaction and behavior testing....

  * Provides utilities specifically designed for testing React components...

  * Encourages testing based on user behavior and interactions....

  Note :- Jest provides an excellent testing framework and can be enhanced with RTL to improve the testing of React components
      especially when focusing on user interactions and behaviors.

* React Testing

  * Package.json

    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",

  Note :- This Above all Library Covers Unit and Integration Test Cases....

  * To Test a Particular File....

  ---> npm test src/App.test.js

  //Syntax for a Function...

  #1
```
test("testing for sum function", ()=>{
  expect(sum(10,20)).toBe(30);
});
```

  #2
```
test("testing for Sum Function With Different method", ()=> {

  let a=10;
  let b=20;
  let output=30
  expect(sum(a,b)).toBe(output)
});
```

  * React Testing Structure....

```
    * What needs to import....
    * How to render the component....
    * How tests working with RTL....
    * How test finding UI elements....



    * Important Things to import in App.test.js

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

      * render :- It is used to render a Components....
      * screen :-

      * Can we Test Multiple Components in a Single File...?
      => We Can Test Multiple Componnets in a Single Test File...


*   Writing First React Test

    *  Make New Test function...
    *  Write test case for check text on screen...
    *  Write test for case-insensitive text...
    *  Test title for image...
    *  Write multiple expect in the same test function....


*   Test Input Box

    * Make Input box in App Component
    * Write Test case Function
    * The test Input box is present or not.

    * Test input box.

      * name ,
      * Placeholder,
      * Id,
      * Value,
      * Type,


* Test Case Run Options....

    * How to run specific test files...
    * What is watch Mode...?     ----------> (IMP)
    * How to run the failed test case....  -----> (IMP)
    * How to run call test cases...        -----> (IMP)
    * How to quit watch mode...     ------> (IMP)
    * How to filter test files for run....   ----> (IMP)
    * How to Filter Test case....


*   Test Grouping With Describe

    * What is Describe?
    * How to make testcases Group?
    * Run test case with Describe?
    * Skip in Describe?
    * Only in Describe...
    * Nested Describe...


    Syntax for describe :-

    describe("Name of Test Case ", () =>
      {
        Write Your Test Cases....
      }
    )


    *    Example :-

    describe("API test case group", ()=> {
```

```
test("api case 1 ",()=> {
  render(<InputText/>);
  let checkInput1 = screen.getByRole("textbox");
  expect(checkInput1).toHaveAttribute("name","username")
})
test("api case 2 ",()=> {
  render(<InputText/>);
  let checkInput1 = screen.getByRole("textbox");
  expect(checkInput1).toHaveAttribute("name","username")
})
test("api case 3 ",()=> {
  render(<InputText/>);
  let checkInput1 = screen.getByRole("textbox");
  expect(checkInput1).toHaveAttribute("name","username")
})
})


* To Skip any of The Test Case....

  // To Skip Test Case....

describe("Name of Test Case ", () =>
  {
    Write Your Test Cases....
  }
)


// To Execute This Test Cases Only....

  describe.only("Name of Test Case ", () =>
  {
    Write Your Test Cases....
  }
)


// To Skip This Group of  Test Cases Only....

  describe.skip("Name of Test Case ", () =>
  {
    Write Your Test Cases....
  }
)

// Nested Describe

describe("",()=>{

    test.skip("",()=>{

    })

    describe("",()=>{

      test("",()=>{


      })

    })
})



# Test OnChange Event with Input Box...

  * Make Input box in the component.
  * Define state and use with on change event.
  * Import Component in Test File.
  * Write Code for test case.
  * Run test case.

  describe("UI Testing for OnChange Event",()=>{

    test("Input Box Testing",()=>{
```

```
    render(<Input/>);
    let inputtest = screen.getByRole("textbox");
    fireEvent.change(inputtest,{target:{value:"a"}});
    expect(inputtest.value).toBe("a");
  });
});
```

* Test Click Event with Button...

  * Make Button and State in the Component...
  * Update state with button click event...
  * Import Component in test file...
  * Write code for test click event...
  * Run Test Case....

```
import { fireEvent, render , screen } from "@testing-library/react";
import Button from "../Components/Button";

describe(" UI Testing Button",()=>{

  test("test case on the button",()=>{
    render(<Button/>);
    const TestButton = screen.getByRole("button");
    fireEvent.click(TestButton);
    expect(screen.getByText("Welcome World ReactJs World Atharva Deelip Deshmukh")).toBeInTheDocument();
  })
})
```

* File and Folder naming Convention....

  * What file name we can use for test case file?
  * Folder name for testing files...
  * Run test case with naming convention

  Example :- test_name.test.js

  Different Naming Convention....

  * file_name.test.js
  * fie_name.spec.js
  * file_name.spec.jsx
  * __test__ ---> Folder Name
  * Inside __tests__ Folder We can Write Simply text.js (no need to write text.test.js)....

* Before and After Hooks....

  ---> They all are Simply Jest Hooks and not a React Hooks....

  * use of before and after hooks...
  * beforeAll and beforeEach....
  * AfterAll and afterEach...
  * Example....

  Uses :-

  1) When we Want to Run Any Of the Function...
  2) To Clean the Database...
  3) To set a Environment...
  4) for Uni Testing if we want to set the constant....

  * beforeAll :- It will Run Only Once all the Cases...

  * beforeEach :- It will Run as Many Test Cases are Present...

  * AfterAll :- When All the Test Cases are Runned then also it will run once...

* afterEach :- When All the Test Cases are Runned then also it will run as Many Test Cases are Present...

* SnapShot Testing....

  * What is SnapShot testing...?
  * Example ?
  * When this is usefull?
  * How to update snapshots?

  * What is SnapShot Testing....?

  * Snapshot tests are useful when you want to make sure your UI does not change unexpectedly.

```
import { render , screen } from "@testing-library/react";
import Data from "../Components/Data";

describe("UI testing",()=>
{
  test("testing a Data",()=>
  {
    let Data1=render(<Data/>);
    expect(Data1).toMatchSnapshot();
  })
})
```

* Important Points for Testing

  * What we Should Test ?
  * What things we should not test ?
  * Important Points....?

  * What we Should Test ?

    * Testing component rendering
    * UI Elements that we write
    * Functions which we write ---> To Check a Validation....
    * API Testing
    * Event Testing ----> Button and Input Testing and Many More....
    * Props and States
    * UI Condition testing | UI State Testing...

  * Avoid Testing for

    * External UI Library code...
    * No Need to test default function of JS and React...
    * Sometimes we should mock function rather than testing it in details...

    * Important Points

      * Do not write snapshots in starting of the project...
      * Run test case after completing your functionality...
      * Make a standard for code coverage...

  * Class Component Method Testing

    * Make Class Component....
    * Install React test renderer....
    * Test Class Component Method....

* Generally We Should Not Do a Function Testing Because We are reside with the Output...

* Test Render Package Creates the Instance of that Class Components....

To Install the react-test-renderer using the Below Following Command....

```
> npm i react-test-renderer
```

In the Testing Component Testing File Import Below Package

```
> import renderer from 'react-test-renderer';
```

```
import Users from "../Components/UsersClassComp";
import renderer from 'react-test-renderer';

test("Class Components method testing",()=>{

const componentData = renderer.create(<Users/>).getInstance();
expect(componentData.getUserList()).toMatch("user List")

})
```

* Functional Component Method Testing...
  * Discuss Possible case for method testing
  * Define the button, click event and method...
  * Test method with event...
  * Test method without event....

* First method to Test a Functional Component....

UserFunctionalComp.js

```
import React, { useState } from 'react'

function UsersFunctionalComp() {

const [data,SetData] = useState("");

const handleData = () => {
  SetData("Hello World");
}
  return (
  <div>
    <h1>Testing a Functional Components</h1>
    <button  data-testid="btn1"  onClick={handleData}>Update Data</button>
    <h2>{data}</h2>
  </div>
  )
}

export default UsersFunctionalComp;
```

UserFunctionalComp.test.js

```
import { fireEvent, render, screen } from "@testing-library/react"
import UsersFunctionalComp from "../Components/UsersFunctionalComp";

describe("UI Testing",()=> {
  test("Functional Components Testing",()=>{
    render(<UsersFunctionalComp/>);
    const btn = screen.getByTestId("btn1");
    fireEvent.click(btn);
    expect(screen.getByText("Hello World")).toBeInTheDocument();
```

```
    })
  })
```

> Developer's Standared Method...

* Second method to Test a Functional Component....

> Take a particular Function in Different Component then Access Those Component in UserFunctionalComp.test.js
  and then Test that Component Which Contain a Particular Function...

> handle.js

```
const handleData1 = () =>
{
  console.log("Atharva Deshmukh");
  return "Atharva Deshmukh";
}

export default handleData1;
```

> UserFunctionalComp.test.js

```
import { fireEvent, render, screen } from "@testing-library/react"
import handleData1 from "../Components/handle";
describe("UI Testing 2",()=>{

test("Functional Component Test 3",()=>{

  expect(handleData1()).toMatch("Atharva Deshmukh");

})

})
```

* RTL Query ---> (IMP)...

  * What is RTL Query ?
  * Why need RTL Query ?
  * Steps in Testing UI ?
  * How RTL Query finds elements ?
  * Type of RTL Queries....

  * RTL Query :- RTL Query is Used to find Our UI Elements So that We Can Test That Elements...

  Steps in Testing UI

  * Render Component...
  * Find Element and action...
  * Assertions...

  * How RTL Find Elements

  * By Element Type
  * By Element name
  * By Element id
  * By Test id

  * Type of RTL Quries

    * Find Single Element

      * getBy

```
      * queryBy
      * findBy

    * Find Multiple Elements

      * getAllBy
      * queryAllBy
      * findAllBy



#

* getByRole Query  -----> Most Used Query....



  * What is the Role in getByRole ?
  * What is semantic elements ?

    * Button,heading tags and table are semantic element??????...
    * Div and span are not semantic elements

  * Test textbox with getByRole

    * text box present or not....
    * text box value...
    * text box disbaled or not...


  * Test button with getByRole....


    * getByRole Comes in the Catgory of getBy....

    * What is the Role in getByRole ?

    >   In Our UI There are Many Semantic Tags Where Each of Them Role is Already Defined....


    * What is Semantic Tags ?

    > Semantic tags are those tags Which Tells Themself and Browser and us that What are there Specific Working...

    > button
    > heading tags
    > table


    * Example of getByRole Testing on TextBox

    FirstInput.js

    import React from 'react'

    export default function First() {
      return (
      <div >
        <h1>getByRole</h1>
        <input type="text" />
      </div>
      )
    }

    FirstInput.test.js


    import { render, screen } from "@testing-library/react"
    import First from "../Components/getByRole/FirstInput";

    describe ("UI Testing Part",()=>{

      test("Testing Input Box ", ()=> {
        render(<First/>);
        let input=screen.getByRole("textbox");
        expect(input).toBeInTheDocument();
      })
```

```
})


* Note :- To Set a Value in a Input TextBox We Can Give Through Two Approach

  * 1) Pass the defaultValue="Atharva Deshmukh";
  * 2) Pass a Value Through OnChange Using Different Function....


Type To Check Whether It is a Input TextBox is Disbaled or Not...

Input.js

import React from 'react'

export default function First() {
  return (
  <div >
    <h1>getByRole</h1>
    <input type="text" defaultValue={"Atharva"} disabled/>
  </div>
  )


Input.test.js

import { render, screen } from "@testing-library/react"
import First from "../Components/getByRole/FirstInput";

describe ("UI Testing Part",()=>{

  test("Testing Input Box ", ()=> {
    render(<First/>);
    let input=screen.getByRole("textbox");
    expect(input).toBeInTheDocument();
    // expect(input).toHaveValue("Enter You Name :- ");
  })

})


Button.test.js

    test ("Button Testing test-1",()=>
  {
    render(<First/>);
    let btn=screen.getByRole("button");
    fireEvent.click(btn);
    expect(btn).toBeInTheDocument();      // To Check a Button is Present or Not...
    expect(screen.getByText("Button UI Testing")).toBeInTheDocument();   // To Check Whether a Text is printed on Scre

  })

* Types in expect

* toBeInTheDocument();

  * To Check Whether My InputFields are Present or Not....

  * expect(input).toBeInTheDocument();

* toHaveValue();   ----> It Contains a Value those Same value or Not....

  * expect(input).toHaveValue("Enter You Name :- ");

* toBeDisabled(); ----> To Check Whether Button is Enabled or Disbaled....

  * expect(input).toBeDisabled();


* toHaveAttribute("key","value");

  * To Check Whether The key value pair are Present in the Code or not...

  *   expect(input).toHaveAttribute("name","username");
```

```
#
* Multiple elements with Role Custom Role....

  * Multiple elements with the same role issue...
  * Multiple buttons with role...
  * Multiple Input box with role...
  * Custom Role....



  * Multiple elements with the same role issue...

    Solution :-


    Button.js

    import React from "react";

    function Box() {
      return (
      <div>
        <button>Click 1</button>
        <button>Click 2</button>
      </div>
      )
    }

    export default Box;



    // *  It Will Show the Error  TestingLibraryElementError: Found multiple elements with the role "button"....


    Button.test.js

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

    describe("UI  Testing",()=>{
      test("Button Testing test 1",()=>{
        render(<Box/>);
        let btn=screen.getByRole("button");
        let btn1=screen.getByRole("button");
        expect(btn).toBeInTheDocument();
        expect(btn1).toBeInTheDocument();
      })
    })


    // Solution for Above Error...

    * We Can use Multiple Attributes in getByRole("button",{name:"Click 2"});


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

    describe("UI  Testing",()=>{
      test("Button Testing test 1",()=>{
        render(<Box/>);
        let btn=screen.getByRole("button",{name:"Click 1"});
        let btn1=screen.getByRole("button",{name:"Click 2"});
        expect(btn).toBeInTheDocument();
        expect(btn1).toBeInTheDocument();
      })
    })



    Multiple Input TextBox in Same test Check....
```

```
Input.js

import React from 'react'

function Box1() {
  return (
  <div>
    <label htmlFor="input1">User Name</label>
    <input type="text" id='input1'/>
    <label htmlFor="input2">User Name 2</label>
    <input type="text" id='input2'/>
  </div>
  )
}

export default Box1;


Input.test.js


import { render , screen } from "@testing-library/react"
import Box1 from "./Box1";

describe("UI Testing ",()=>{
test('Multiple InputBox Testing', () => {

render(<Box1/>);
let input1=screen.getByRole("textbox",{name:"User Name"});
let input2=screen.getByRole("textbox",{name:"User Name 2"});

expect(input1).toBeInTheDocument();
expect(input2).toBeInTheDocument();
})

})


// How to Use Custom Role.... -----> (IMP)....


* How to Test a Non Semantic Elements....



Error :- TestingLibraryElementError: Unable to find an accessible element with the role ""...


  div.js

  <div>
    <h1>Hello Atharva It Your Time You Achive Something in Life..</h1>
  </div>

  div.test.js

  const dv1=screen.getByRole("");
        expect(dv1).toBeInTheDocument();


  // To Slove Above Error....  Define Manually Role in  div.js....


  div.js

  <div role='dummy'>  ----> Define manually Role.... in Html File...

    <h1>Hello Atharva It Your Time You Achive Something in Life..</h1>
  </div>



  div.test.js

  const dv1=screen.getByRole("dummy");
```

```
        expect(dv1).toBeInTheDocument();
```


* RTL Query : getAllByRole

* React Testing Using Typescript

  * Test Driven Development :- * Write a First test and to Pass That Case We Should Write Code later...

  * Concept :- Props Object With a Name....

  First.tsx

```
import { render, screen } from "@testing-library/react"
import Second from "./Second";

describe("UI Testing",()=>
{
  test("Text Testing Second",()=> {

    render(<Second name='Atharva'/>);
    let Element=screen.getByText("Hello Atharva");
    expect(Element).toBeInTheDocument();
  })
})
```

  First.test.tsx

```
import React from 'react'
type GreetProps = {
  name?: string
}

export default function Second(props:GreetProps) {
  return (
  <div>
  <h1>Hello {props.name}</h1>
  </div>
  )
}
```

Jest 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...

  * An Optimization designed to make your tests run fast regardless

of How many tests you have....

* Note :- In ReactJs With Typescript We Can use it with the replace of test...
They Both Are Declared Globally...

Syntax :-

```
describe("",()=>{
  it("",()=>{

  })
  it("",()=>{

  })
})
```

* To Replicate test.only we Can Use

```
test.only("",()=>{

})
```

replace it with

```
fit("",()=>{

})
```

and to Exculde a test use

```
xit("",()=>{

})
```

Code Coverage

* A metric that can help you understand how much of your Software code is tested...

* Statement Coverage :- How many of the statements in the Software code have been executed

* Branches Coverage:- How many of the Branches of the Controll Structures(if statements for instance) have been executed

* function coverage :- How many of the functions defined have been called and finally..

* Line Coverage: How many of lines of source code have been tested...

* Important Command To See Whole Coverage

Add in Package.json

```
* "scripts": {
    "test:coverage": "react-scripts test --coverage --watchAll"
  }
```

> Run The Command in CMD

* npm run test:coverage

```
    * Add in Package.json for the Whole Coverage

    Case-1 :- If we Want to Access to Test  one Folder files...

    "test:coverage": "react-scripts test --coverage --watchAll --collectCoverageFrom='src/Components/**/*.{ts,tsx}"

    or

    "test:coverage": "react-scripts test --coverage --watchAll --collectCoverageFrom='!src/components/**/*.{types,stories,


    (IMP)

    * CoverageThreshold :- With Jest it is Possible to Specify Minimum threshold Inforcement for Coverage Reports...


      "jest":{
      "coverageThreshold":{
        "global":{
          "branches": 100,
          "functions": 100,
          "lines": 100,
          "statements": 100
        }
      }
      }


      * Assertions

      * When writing tests, we often need to check that values
        meet certain conditions...

      * Assertions decide if a test passes or fails...

      * expect(value)

      * The argument should be the value that your code produces...
      * Typically, you will use expect along with a "matcher
        function to assert something about a value...
      * A Matcher can optionally accept an arguement which is the correct expected value...



  #
  * What is Test...?

    > Test Component renders...
    > Test Component renders with props...
    > Test Component renders in different States...
    > Test Component reacts to events...

  * What not to test?

    > Implementation details....
    > Third Party Code...
    > Code That is not Important from a User Point of view...



# RTL Queries

  * 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

    To render the component, we use the render method from RTL

    * For assertion, we use expect passing in a value and combine it with a matcher
        function from jest or jest-dom..
```

* Queries are the Methods that testing Library provides to find elements on the page..

  * To Find a Single element on the page, We Have

    * getBy...
    * queryBy...
    * findBy...

  * To find multiple elements on the page, we Have

    * getAllBy...
    * queryAllBy...
    * FindAllBy...

    The Suffix can be one of Role, LabelText PlaceHolderText, text, DisplayValue, AltText, Title and Finally TestId...

    * getBy.. class of queries return the matching node for a query, and throw a descriptive error if no elements match
      or if more than one match is found...

    * The Suffix can be one of Role, LabelText PlaceHolderText, text, DisplayValue, AltText, Title and Finally TestId...

    * By default, many semantic elements in HTML have role....

    * Button element has a button role, anchor element has link role, h1 to h6
      elements have a heading role, checkboxes have a checkbox role, radio buttons have a radio role and so on...

    * If you are Working with elements that do not have a default role or if you
      want to specify a different role, the role attribute can be used to add the desired role..

    * To use an anchor element as a button in the navbar, you can add role='button'...

    * getByRole

    * getByRole Options ----> (IMP)

      * It is UseFull When Multiple Elements Have Same Role....

      1) name
      2) level
      3) hidden
      4) selected
      5) checked
      6) pressed

    Third.js

```
import React from 'react'

export default function Third() {
  return (
  <div>
    <form action="">
      <div>
        <label htmlFor="name">Name</label>
        <input type="text" id='name'/>
      </div>
      <div>
        <label htmlFor="job-location">Job Location</label>
        <select value="">
          <option value="">Select a Country</option>
          <option value="US">United States</option>
          <option value="GB">United Kingdom</option>
          <option value="CA">Canada</option>
          <option value="AU">Australia</option>
          <option value="FR">France</option>
          <option value="DE">Germany</option>
        </select>
      </div>
      <div>
        <label>
          <input type="checkbox" id='terms' /> I agree to the terms and conditions
        </label>
      </div>
```

```
        <button>Submit</button>
      </form>
    </div>
    )
  }
```

Third.test.js

```
import { render , screen} from "@testing-library/react";
import Third from "./Third";

describe('Application', () => {
  test('Renders Correctly', () => {

    render(<Third/>);
    let Input=screen.getByRole("textbox");
    expect(Input).toBeInTheDocument();

    let  jobLocationElement = screen.getByRole("combobox");
    expect(jobLocationElement).toBeInTheDocument();

    const termsElement = screen.getByRole("checkbox");
    expect(termsElement).toBeInTheDocument();

    const submitButtonElement = screen.getByRole("button");
    expect(submitButtonElement).toBeInTheDocument();
  })

})
```

* To Test a Multiple TextBox....

Third.js

```
<div>
  <label htmlFor="name">Name</label>
  <input type="text" id='name'/>
</div>
<div>
  <label htmlFor="bio">Bio</label>
  <textarea name="bio" id="bio"></textarea>
</div>
```

Third.test.js

```
let Input=screen.getByRole("textbox" , {name:"Name"});
expect(Input).toBeInTheDocument();

let BioName=screen.getByRole("textbox" , {name:"Bio"});
expect(BioName).toBeInTheDocument();
```

* To Test a Multiple Heading...

Third.js

```
  <h1>Job Application Form</h1>
<h2>Section 1</h2>
```

Third.test.js

```
let Head = screen.getByRole("heading" , {name:"Job Application Form"});
expect(Head).toBeInTheDocument();

let Head1 = screen.getByRole("heading" , {name:"Section 1"});
expect(Head1).toBeInTheDocument();
```

```
  * Note :- To Test Multiple Same Field With Different Name Assign to Them...

    * We Can Differentiate Heading By There Level Also...

    example :- level:2;



#
* getByLabelText

  * getByLabelText will search for the label that matches the given text, then find
      the element associated with the label....


  Third.js

  <label htmlFor="name">Name</label>
        <input type="text" id="name" />


  Third.test.js


  let nameElement = screen.getByLabelText('Name');
  expect(nameElement).toBeInTheDocument();


  (IMP)

  Note :- * Use Selector Also To Differntiate Between Input and textarea or any...

  Third.js

    <div>
    <label htmlFor="name">Name</label   ----> (*)
    <input type="text" id="name" />
  </div>


  Third.js

  <div>
    <label htmlFor="job-location">Name</label> ---> (*)
    <select value="">
      <option value="">Select a Country</option>
      <option value="US">United States</option>
      <option value="GB">United Kingdom</option>
      <option value="CA">Canada</option>
      <option value="AU">Australia</option>
      <option value="FR">France</option>
      <option value="DE">Germany</option>
    </select>
  </div>



  Third.test.js

  let Input=screen.getByRole("textbox" , {name:"Name"});
  expect(Input).toBeInTheDocument();


  let nameElement = screen.getByLabelText('Name',{selector:"input"});
  expect(nameElement).toBeInTheDocument();



#
* getByPlaceholderText

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

#
* getByText

Fourth.js

```
<p>This Section is Mandatory</p>
```

Fourth.text.js

```
let paragraphText = screen.getByText("This Section is Mandatory");
expect(paragraphText).toBeInTheDocument();
```

#
* getByDisplayValue

* getByDisplayValue returns the input, textarea, or select element that has the matching display value...

Fifth.tsx

```
<div>
  <label htmlFor="name">Name</label>
  <input type="text" id="name" placeholder="Fullname" value="Atharva" onChange={()=> {}}/>
</div>

// OnChange Handler is Used to Remove the Warning....
```

FiFth.test.tsx

```
let nameElement2=screen.getByDisplayValue("Atharva");
expect(nameElement2).toBeInTheDocument();
```

#
* getByAltText

* getByAltText will return the element that has the given alt text...
* This method only supports elements which accept an alt attribute like <img>,
<input>,<area> or custom HTML elements...

Sixth.js

```
<img src="" alt="A Natures Image" />
```

Sixth.test.js

```
let ImageElement=screen.getByAltText("A Natures Image");
expect(ImageElement).toBeInTheDocument();
```

#
* getByTitle

* getByTitle returns the element that has the matching title attribute...

Seventh.js

```
<span title="close"></span>
```

Seventh.test.js

```
let closeElement=screen.getByTitle("close");
    expect(closeElement).toBeInTheDocument();
```

#
* getByTestId

* getByTestId returns the element that has the matching data-testid attribute...

Eight.js

```
<div data-testid="custom-element">
  Custom HTML element
</div>
```

Eight.test.js

```
let CustomElement = screen.getByTestId("custom-element");
    expect(CustomElement).toBeInTheDocument();
```

#
* Priority Order for Queries

* Note :- Your test should resemble how users interact with your code
  (Component , page) as much as possible....

1) getByRole

2) getByLabelText

3) getByPlaceholderText

4) getByText ---> Outside a Form we can Simply find Elements like div , span , paragraph ....

5) getByDisplayValue

6) getByAltText :- When Your Element is one Which supports all text such Image , area , Input or any Custom Element...

7) getByTitle

8) getByTestId

#
* RTL getAllBy Queries

* Find multiple elements in the DOM...

* getAllBy returns an array of all matching nodes for a query, and throws
  an error if no elements match...

* getAllByRole

* getAllByLabelText

* getAllByPlaceholderText

* getAllByText

* getAllByDisplayValue

* getAllByAltText

* getAllByTitle

* getAllByTestId

#
* Query Multiple Elements

> skills.types.ts

```
export type SkillsProps = {
  skills: string[];
}
```

> skills.tsx

```tsx
import { SkillsProps} from "./skills.types";
import React from 'react'

export const Skills = (props:SkillsProps) => {
  const {skills} = props;
  return (
  <>
    <ul>
      {skills.map((skill)=>{
        return <li key={skill}>{skill}</li>;
      })}
    </ul>
  </>
  )
}
```

> skills.test.tsx

```tsx
import { render , screen } from "@testing-library/react";
import { Skills } from "./skills";

describe('Skills', () => {
  const skills = ["HTML","CSS","JavaScript"];

  test('renders Correctly', () => {
    render(<Skills skills={skills} />);
    const listElement = screen.getByRole("list");
    expect(listElement).toBeInTheDocument();
  })

  test("renders a list of skills",()=>{
    render(<Skills skills={skills} />);
    const listItemElements = screen.getAllByRole("listItem");
    expect(listItemElements).toHavelength(skills.length);
  })
})
```

#
* Text Match

  * TextMatch represents a Type which can be either a

  * String...
  * regex...
  * function...

  * String

    Hello World

    screen.getByText("Hello World"); // full String match...

    screen.getByText('llo Worl', {exact:false}) // substring match...

    screen.getByText('hello world',{exact:false}) // ignore case

  * TextMatch - regex

    Hello World

    screen.getByText(/World/) // substring match

    screen.getByText(/word/i) // substring match, ignore case...

    screen.getByText(/^hello world$/i) // full string match, ignore Case...

  * TextMatch - custom function

  (content?: string, element?: Element | null) => boolean

```
    <div>Hello World </div>

    screen.getByText((content)=> content.startsWith('Hello'))


  #
  * queryBy



    Tenth.types.js

    export type SkillsProps = {
    skills: string[];
    }

     Tenth.js

    import { SkillsProps} from "./skills.types";
    import React from 'react'

    export const Skills = (props:SkillsProps) => {
      const {skills} = props;
      return (
      <>
        <ul>
          {skills.map((skill)=>{
            return <li key={skill}>{skill}</li>;
          })}
        </ul>
      </>
      )
    }


    Tenth.test.js


    import { render , screen } from "@testing-library/react";
    import { Skills } from "./skills";

    describe('Skills', () => {
      const skills = ["HTML","CSS","JavaScript"];

      test('renders Correctly', () => {
        render(<Skills skills={skills} />);
        const listElement = screen.getByRole("list");
        expect(listElement).toBeInTheDocument();
      })

      test("renders a list of skills",()=>{
        render(<Skills skills={skills} />);
        const listItemElements = screen.getAllByRole("listItem");
        expect(listItemElements).toHavelength(skills);
      })
    })




  Testing a Linear Gradient....


  // MyComponent.tsx
  import React from 'react';

  const MyComponent = () => {
    return (
      <div style={{ backgroundImage: 'linear-gradient(to right, red, blue)' }}>
        This is my component
      </div>
    );
  };
```

```
// MyComponent.test.tsx
import React from 'react';
import { render } from '@testing-library/react';
import { expect } from '@jest/globals';
import MyComponent from './MyComponent';

describe('MyComponent', () => {
  it('should render the expected linear gradient', () => {
    const { container } = render(<MyComponent />);
    expect(container).toMatchInlineSnapshot(); // Capture the rendered component
  });
});
```

Mocking: Mock the component that provides the text or the function that generates it. Update the mock data to simulate the te

```
        // "@testing-library/jest-dom": "^6.4.2",
```