React Testing

1. Introduction

The Series is intended to teach me how to write automated tests for a react application

In this course we will learn react testing with Jest and React Testing Library

Let’s begin with a brief introduction to testing

Une image contenant texte

Description générée automatiquement

For that we typically rely on manual testing

Une image contenant texte

Description générée automatiquement

Now what is a solution to these Drawbacks of manual testing

So the solution is Automated Testing .

Une image contenant texte

Description générée automatiquement

So now we will see the course structure

Une image contenant texte

Description générée automatiquement

1. Jest and react testing Library
2. Jest

* Jest is a JavaScript testing framework
* Jest is a test runner that finds tests, runs the tests, determines whether the tests passed or failed and reports it back in a human readable manner

1. React testing Library

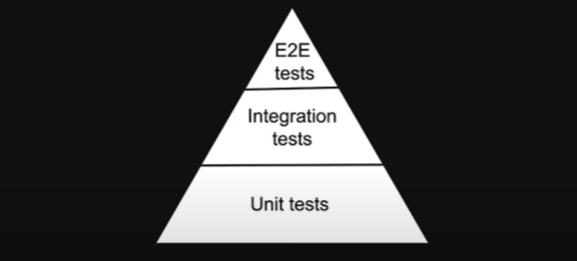
* JavaScript testing utility that provides virtual DOM for testing react components.
* React Testing Library provides a virtual DOM which we can use to interact with and verify the behavior of a react component.
* Testing Library is infact a family of packages which helps test Ul components
* The core library is called DOM Testing library and RTL is simply a wrapper around this core library to test react applications in an easier way.

**Now we will be learning about the two libraries as we progress through the series but I felt it was essential to set some context as to what the two libraries are and to clearly state that one is not an alternative to the other we will using both of them together**

1. Jest and react testing Library

Types of tests

* There are three types of tests.
* **Unit test** 
  + Focus is on testing the individual building blocks of an application such as a class or a function or a component
  + Each unit or building block is tested in isolation, independent of other units Dependencies are mocked
  + Run in a short amount of time and make it very easy to pinpoint failures
  + Relatively easier to write and maintain N
* **Integration Test** 
  + Focus is on testing a combination of units and ensuring they work together
  + Take longer than unit tests
* **E2E Tests**
  + Focus is on testing the entire application flow and ensuring it works as designed from start to finish
  + Involves in a real Ul, a real backend database, real services etc
  + Take the longest as they cover the most amount of code
  + Have a cost implication as you interact with real APIs that may charge based on the number of requests
* Testing pyramid.



The bulk of our test are unit test at the bottom of the pyramid as you move up the pyramid our test get larger but at the same time the number of tests get smaller , unit test are the easiest to write and maintain but the E2E tests give me the most confidence as they loosely resemble a user testing our application .

**What sort of tests are we writing in the series :**

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

create-react-app

When we create react app using npx create-react-app MyApp it will install automatically Jest and React-testing-library

Test

* 1. Run a test

If we will run a Test we run

* + **npm test**
  1. Anatomy of a test and create our first automated test

Une image contenant texte

Description générée automatiquement

**Our first Test:**

* **We use the combination of Jest and React testing library This is the anatomy of a test in a react projet**
* **Both test and expect methods from jest which create react app globally provides in every test**
* **What we will see in this example:**
  + **Greet should render the text hello and if a name is passed into the component It should render hello followed by the name**

**Greet.tsx**

export const Greet = () => {

  return <div>Hello</div>;

};

Greet.test.tsx

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

import { Greet } from "./Greet"

test('Greet renders correctly',()=>{

    render(<Greet/>)

    // render hels us to create a virtual dom of the componenet

    const textElement= screen.getByText('Hello')

    //

    expect(textElement).toBeInTheDocument()

    // we expect textElement to be in the document

})

Une image contenant texte

Description générée automatiquement

* 1. RedGreen Testing

Une image contenant texte

Description générée automatiquement

In the TDD we pass a redTest to a greenTest

So we will create our test at first :

Greet.test.tsx

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

import { Greet } from "./Greet"

test('Greet renders correctly',()=>{

    render(<Greet/>)

    // render hels us to create a virtual dom of the componenet

    const textElement= screen.getByText('Hello')

    //

    expect(textElement).toBeInTheDocument()

    // we expect textElement to be in the document

})

test('Greet renders with a name',()=>{

    render(<Greet name='Vishwas'/>)

    const textElement= screen.getByText('Hello Vishwas')

    expect(textElement).toBeInTheDocument()

})

Second step we create a software code and we will refctor the code

Greet.tsx

type GreetProps = {

  name?: string;

};

export const Greet = (props: GreetProps) => {

  return <div>Hello {props.name}</div>;

};

Resume

Une image contenant texte

Description générée automatiquement

1. Jest
2. Filtering test

* Watch Mode

Une image contenant texte

Description générée automatiquement

Watch mode is default in react testing (When we run npm start we will just see the test in our last commit we can’t see all the test (for optimization ) so if we want to see all the test we need to press a in our terminal)

* Filtring Test

Une image contenant texte

Description générée automatiquement

* Only And Skip filtering
  + Only test method

// It will test only this test

// We use only when we working on a test in a file that contains

// Multiple Test

test.only('Greet renders with a name',()=>{

    render(<Greet name='Vishwas'/>)

    const textElement= screen.getByText('Hello Vishwas')

    expect(textElement).toBeInTheDocument()

})

* + Skip test method

// Jest It will skip that test

test.skip('Greet renders with a name skip',()=>{

    render(<Greet name='Vishwas'/>)

    const textElement= screen.getByText('Hello Vishwas')

    expect(textElement).toBeInTheDocument()

})

* Summarize
  + Watch mode : Run test that have change since the last commit
  + Run Test only in change files
  + Run test by file name
  + Run Test By Test name
  + Only test method
  + Skip Test method

1. Grouping Test

In Jest testing framework for React applications, a describe block is used to group a set of related tests together. This makes it easier to organize and understand the purpose of the tests being written. Each describe block can contain one or more test cases, written using it blocks, which specify the individual test scenarios. The describe block is useful for breaking down a complex test suite into smaller, manageable chunks.

Une image contenant texte

Description générée automatiquement

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

import { Greet } from "./Greet";

describe("Greet", () => {

  test("renders correctly", () => {

    render(<Greet />);

    const textElement = screen.getByText("Hello");

    expect(textElement).toBeInTheDocument();

  });

  test("render a name", () => {

    render(<Greet name="Vishwas" />);

    const textElement = screen.getByText("Hello Vishwas");

    expect(textElement).toBeInTheDocument();

  });

});

describe("Nested", () => {

test("renders  a name", () => {

  render(<Greet name="Vishwas" />);

  const textElement = screen.getByText("Hello Vishwas");

  expect(textElement).toBeInTheDocument();

});

})

Une image contenant texte

Description générée automatiquement

1. Filename Convention

Une image contenant texte

Description générée automatiquement

1. Code coverage

Une image contenant texte

Description générée automatiquement

* We will configure our package.json

"scripts": {

    "start": "react-scripts start",

    "build": "react-scripts build",

    "test": "react-scripts test",

    "eject": "react-scripts eject",

    "coverage":"react-scripts test --coverage --watchAll --collectCoverageFrom='src/Component/\*\*/\*.{ts,tsx}' --collectCoverageFrom='!src/Component/\*\*/\*.{types,stories,constants,test,spec}.{ts,tsx}'"

  },

* + Explanation

react-scripts test --coverage

found the coverage test related to files changed since last commit

--watchAll

To watch all the coverage file test

--collectCoverageFrom='src/Component/\*\*/\*.{ts,tsx}

Within component Folder collect coverage from files that end with .ts or .tsx

--collectCoverageFrom='!src/Component/\*\*/\*.{types,stories,constants,test,spec}.{ts,tsx}

Ignoring coverage Within component from files end with. {types or stories or constants or test or spec}.{tx or tsx}

* Run the script
  + npm run coverage
* Coverage threshold

With jest it is possible to specify a minimum threshold Enforcement for coverage reports If thresholds aren’t met Jest will fail

So we will add the following jest configuration in package.json

"jest": {

    "coverageThreshold": {

      "global": {

        "branches": 80,

        "functions": 80,

        "lines": 80,

        "statements": -10

      }

    }

  }

With this configuration Jest will fail if there is less than 80 percent branch line and function coverage or if there are more than 10 uncovered statements

Greet.test.tsx

test("renders correctly", () => {

    render(<Greet />);

    const textElement = screen.getByText("Hello");

    expect(textElement).toBeInTheDocument();

  });

Greet.tsx

export type GreetProps = {

  name?: string;

};

export const Greet = (props: GreetProps) => {

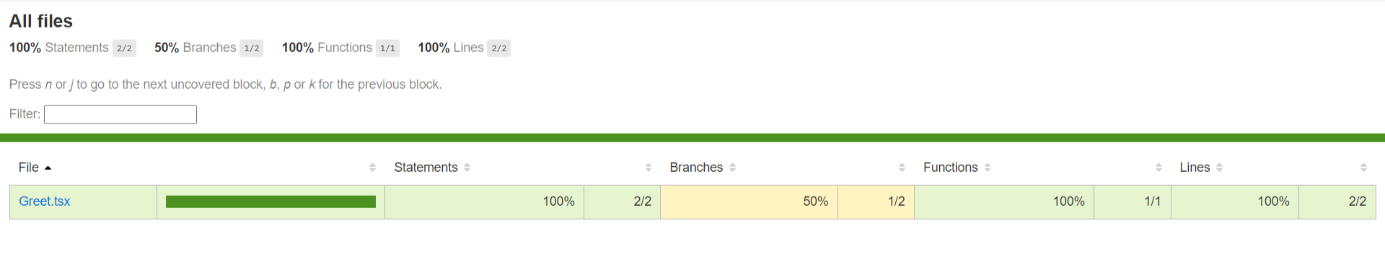
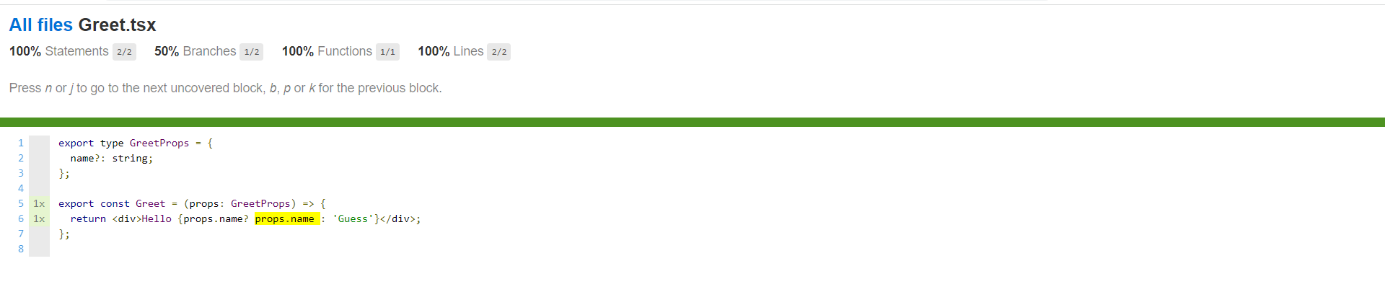
  return <div>Hello {props.name? props.name : 'Guess'}</div>;

};

Une image contenant texte, tableau de points

Description générée automatiquement

Jest also generate an html file of our coverage report (In coverage report folder there is a index.html), So we se the same report plus some css applied.

* Summarize

Code coverage is a metric that can help understand how much of the software code is tested we learned about the jest coverage option option to generate the report we also used the watch all flag to generate coverage for all test in the project we were also specified which files we should or should not collect code coverage from using the collect coverage from flag ,finally we learned hoe to set coverage threshold using the jest configuration in package.json just will fail if the coverage does not meet the threshold requirements ,code coverage is very important from a CI/CD point

1. Assertions

"Assertion" in Jest refers to the act of checking if a value or expression matches the expected result. Jest provides a set of built-in assertions, such as expect, that can be used to test the output of your code. The expect method takes an actual value as an argument, and you can chain methods to it to specify the expected result. If the actual value does not match the expected result, Jest will throw an error and fail the test.

Here's an example of an assertion in Jest:

const add = (a, b) => a + b;

test('adds 1 + 2 to equal 3', () => {

const result = add(1, 2);

expect(result).toBe(3);

});

In this example, the expect method is used to check if the result of add(1, 2) is equal to 3. If it is, the test will pass. If not, the test will fail and an error will be thrown.

1. Section summary

Une image contenant texte

Description générée automatiquement

1. React testing library.
2. What to test and what to not test

* What to test
  + Test component renders
  + Test component renders with props
  + Test component renders in different states
    - For example if we have a navbar the login button should render if the user is logged out and not nogged in
  + Test component reacts to events
    - This is applicable to components like buttons and form controls which allow user interaction.
* What not to test
  + Implementation details
    - We want testing the behavior and not that behavior is implemented that also make refactoring easier.
  + Third party code
    - We should testing our code and not the code we are consuming from an external library for example if we use material UI we don’t have to test the button component or the tabs components from an UI
  + Code that is not important from a user point of view
    - For example if we written a utility function that display date in a user-friendly format we don’t have to test if the function was called by the component instead we can directly test if the date was re-render in the expected format

1. RTL Queries

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

2.1. getByRole

2.1.1 getByRole Definition

Une image contenant texte

Description générée automatiquement

**getByRole** is a method provided by the React Testing Library that helps you find an element in your React component based on its accessibility role. This is useful when you want to test that a particular element is correctly marked up with the correct accessibility role, or when you want to find an element to interact with it as part of a test.

Here's an example of how you might use **getByRole** in a test:

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

const MyComponent = () => (

  <button aria-label="click me">Click me</button>

);

test('My component has a button with aria-label', () => {

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

  const button = getByRole(container, 'button');

  expect(button).toHaveAttribute('aria-label', 'click me');

});

In this example, the **render** method from the React Testing Library is used to render the **MyComponent** component. The **getByRole** method is then used to find a button element within the rendered component, and the **toHaveAttribute** method from Jest is used to check that the button has an **aria-label** attribute with the value "click me".

Pour voir tous les Role pour tous les tags ils faut visiter ce lien : <https://testing-library.com/docs/queries/byrole>

Application.test.tsx

describe("Applicfation",()=>{

    // We test if the input Element is present or not

    test('render correctly',()=>{

        render(<Application/>);

        const nameElement = screen.getByRole('textbox', {

            name: 'Name',

          })

          expect(nameElement).toBeInTheDocument()

          // We test if the select dropdown Element is present or not

          const jobLocationElement= screen.getByRole('combobox')

          expect(jobLocationElement).toBeInTheDocument()

        //   We test if the checkboks Element is present or not

        const termsElement = screen.getByRole('checkbox')

        expect(termsElement).toBeInTheDocument()

        // Test if the Button Element is present or not

        const submitButtonElement=screen.getByRole("button")

        expect(submitButtonElement).toBeInTheDocument()

    } )

})

* + 1. GetByRole Option

Une image contenant texte

Description générée automatiquement

Application.tsx

export const Application = () => {

    return (

      <>

        <h1>Job application form</h1>

        <h2>Section 1</h2>

        <p>All fields are mandatory</p>

        <span title="close">X</span>

        <img src="https://via.placeholder.com/150" alt="a person with a laptop" />

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

        <form>

          <div>

            <label htmlFor="name">Name</label>

            <input

              type="text"

              id="name"

              placeholder="Fullname"

              value="Vishwas"

              onChange={() => {}}

            />

            <label htmlFor="bio">Bio</label>

            <input

              type="text"

              id="bio"

              placeholder="Fullname"

              value="Vishwas"

              onChange={() => {}}

            />

          </div>

          <div>

            <label htmlFor="job-location">Job location</label>

            <select id="job-location">

              <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="IN">India</option>

              <option value="AU">Australia</option>

            </select>

          </div>

          <div>

            <label>

              <input type="checkbox" id="terms" /> I agree to the terms and

              conditions

            </label>

          </div>

          <button disabled>Submit</button>

        </form>

      </>

    )

  }

Application.test.tsx

describe("Applicfation",()=>{

    test('render correctly',()=>{

        render(<Application/>);

         // We test if the input (with label Name <label htmlFor="name">Name</label>) Element is present or not

        //  We test if we have a heading (from h1 o h6)

         const h1Heading = screen.getByRole("heading",{

            level:1,

            name:'Job application form'

         })

         expect(h1Heading).toBeInTheDocument()

         const h2Heading = screen.getByRole("heading",{

            level:2,

            name:'Section 1'

         })

         expect(h2Heading).toBeInTheDocument()

         const nameElement = screen.getByRole('textbox', {

            name: 'Name',

          })

          expect(nameElement).toBeInTheDocument()

           // We test if the input (with label Bio <label htmlFor="bio">Bio</label>) Element is present or not

          const bioElement = screen.getByRole('textbox', {

            name: 'Bio',

          })

          expect(bioElement).toBeInTheDocument()

          // We test if the select dropdown Element is present or not

          const jobLocationElement= screen.getByRole('combobox')

          expect(jobLocationElement).toBeInTheDocument()

        //   We test if the checkboks Element is present or not

        const termsElement = screen.getByRole('checkbox')

        expect(termsElement).toBeInTheDocument()

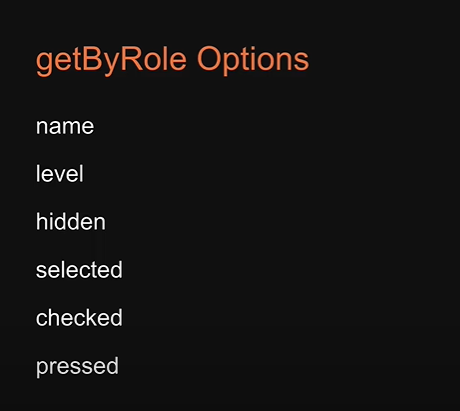
        // Test if the Button Element is present or not

        const submitButtonElement=screen.getByRole("button")

        expect(submitButtonElement).toBeInTheDocument()

    } )

})



* 1. getByLabelText

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

<label htmlFor="name">Name</label>

            <input

              type="text"

              id="name"

              placeholder="Fullname"

              value="Vishwas"

              onChange={() => {}}

            />

<div>

            <label htmlFor="job-location">Name</label>

            <select id="job-location">

              <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="IN">India</option>

              <option value="AU">Australia</option>

            </select>

          </div>

<div>

            <label>

              <input type="checkbox" id="terms" /> I agree to the terms and

              conditions

            </label>

          </div>

Application.test.tsx

{/\*  GetByLabelText \*/}

    // const nameElement2 = screen.getByLabelText("Name");

    // expect(nameElement2).toBeInTheDocument();

    // if we have many Element with the same label we can add the selector

    const nameElement2Test = screen.getByLabelText("Name", {

      selector: "input",

    });

    expect(nameElement2Test).toBeInTheDocument();

    const termsElement2 = screen.getByLabelText(

      "I agree to the terms and conditions"

    );

    expect(termsElement2).toBeInTheDocument();

  });

});

* 1. getByPlaceholderText

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

<input

              type="text"

              id="name"

              placeholder="FullName"

              value="Vishwas"

              onChange={() => {}}

            />

Application.test.tsx

{/\* getByPlaceholderText \*/}

    const nameElement3=screen.getByPlaceholderText("FullName")

    expect(nameElement3).toBeInTheDocument()

* 1. getByText

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

<p>All fields are mandatory</p>

Application.test.tsx

{/\*getByText \*/}

    const paragraphElement=screen.getByText("All fields are mandatory")

    expect(paragraphElement).toBeInTheDocument()

* 1. getByDisplayValue

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

<input

              type="text"

              id="name"

              placeholder="FullName"

              value="Vishwase"

              onChange={() => {}}

            />

Application.test.tsx

 {/\*getByDisplayValue \*/}

  const nameElement4=screen.getByDisplayValue('Vishwase')

  expect(nameElement4).toBeInTheDocument()

* 1. getByAltText

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

        <img src="https://via.placeholder.com/150" alt="a person with a laptop" />

Application.test.tsx

        <img src="https://via.placeholder.com/150" alt="a person with a laptop" />

* 1. getBtTitle

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

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

Application.test.tsx

{/\* getByTitle \*/}

const closeElement=screen.getByTitle("close")

expect(closeElement).toBeInTheDocument()

* 1. getByTestId

Une image contenant texte

Description générée automatiquement

Applicaton.tsx

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

Application.test.tsx

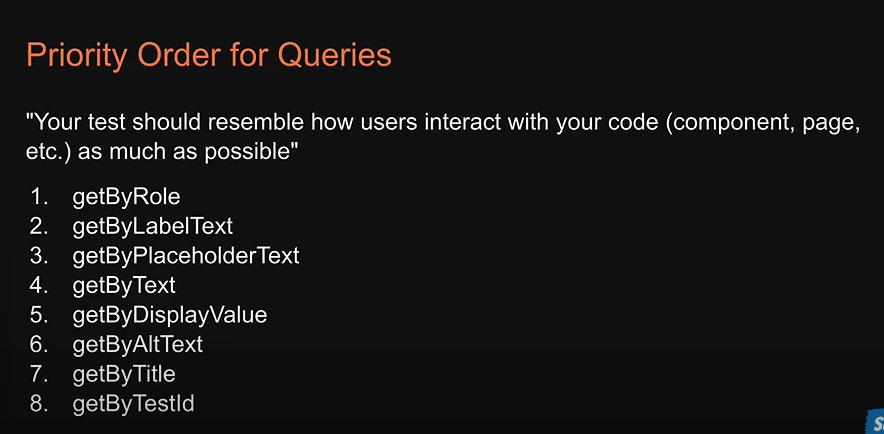
{/\*getByTestId\*/}

const customElement=screen.getByTestId('custom-element')

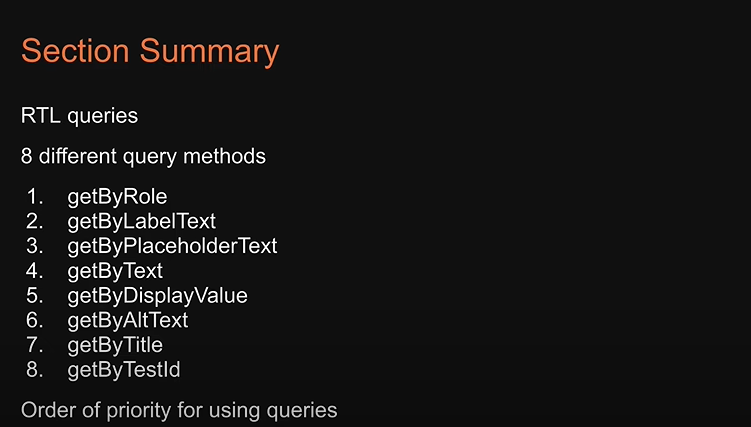
expect(customElement).toBeInTheDocument()

* 1. Priority order for queries And section Summary

If we can’t test our component with the first queries we passed to the next



This query helps us to find Element in the Virtual Dom



* 1. Query multiple elements

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Skills.types.ts

export type SkillsProps = {

    skills: string[]

  }

Skills.tsx

import { SkillsProps } from './skills.types'

export const Skills = (props: SkillsProps) => {

  const { skills } = props

  return (

    <>

      <ul>

        {skills.map((skill) => {

          return <li key={skill}>{skill}</li>

        })}

      </ul>

    </>

  )

}

Skills.test.tsx

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

import { Skills } from "./Skills";

describe("Skills",()=>{

    const skills=["HTML","CSS","JAVASCRIPT"]

    test('render 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)

    })

})

* 1. TextMatch

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

1. queryBy

if we want to test that a certain element is not rendered in the dom

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Skills.tsx

import { useState, useEffect } from 'react'

import { SkillsProps } from './skills.types'

export const Skills = (props: SkillsProps) => {

  const { skills } = props

  const [isLoggedIn, setIsLoggedIn] = useState(false)

  useEffect(() => {

    setTimeout(() => {

      setIsLoggedIn(true)

    }, 1001)

  }, [])

  return (

    <>

      <ul>

        {skills.map((skill) => {

          return <li key={skill}>{skill}</li>

        })}

      </ul>

      {isLoggedIn ? (

        <button>Start learning</button>

      ) : (

        <button onClick={() => setIsLoggedIn(!isLoggedIn)}>Login</button>

      )}

    </>

  )

}

Une image contenant texte

Description générée automatiquement

because the start learning button is not in the dom

Solution:

test("start learning button in not rendred",()=>{

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

        const startLearningButton=screen.getByRole("button",{

            name:"Start learning",

        });

        expect(startLearningButton).not.toBeInTheDocument()

    })

4.FindBy

Une image contenant texte

Description générée automatiquement

Skills.tsx

import { useState, useEffect } from 'react'

import { SkillsProps } from './skills.types'

export const Skills = (props: SkillsProps) => {

  const { skills } = props

  const [isLoggedIn, setIsLoggedIn] = useState(false)

  useEffect(() => {

    setTimeout(() => {

      setIsLoggedIn(true)

      console.log("test")

    }, 1500)

  }, [])

  return (

    <>

      <ul>

        {skills.map((skill) => {

          return <li key={skill}>{skill}</li>

        })}

      </ul>

      {isLoggedIn ? (

        <button onClick={() => setIsLoggedIn(!isLoggedIn)}>Start learning</button>

      ) : (

        <button onClick={() => setIsLoggedIn(!isLoggedIn)}>Login</button>

      )}

    </>

  )

}

Une image contenant texte

Description générée automatiquement



skills.test.tsx

test("start learning button is eventually displayed", async () => {

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

    const startLearningButton = await screen.findByRole(

      "button",

      {

        name: "Start learning",

      },

      {

        timeout: 2000,

// 1000ms default value

      }

    );

    expect(startLearningButton).toBeInTheDocument();

  });

1. Debugging

Screen..debug()🡺we see the dom trees printed in the terminal

Une image contenant texte

Description générée automatiquement

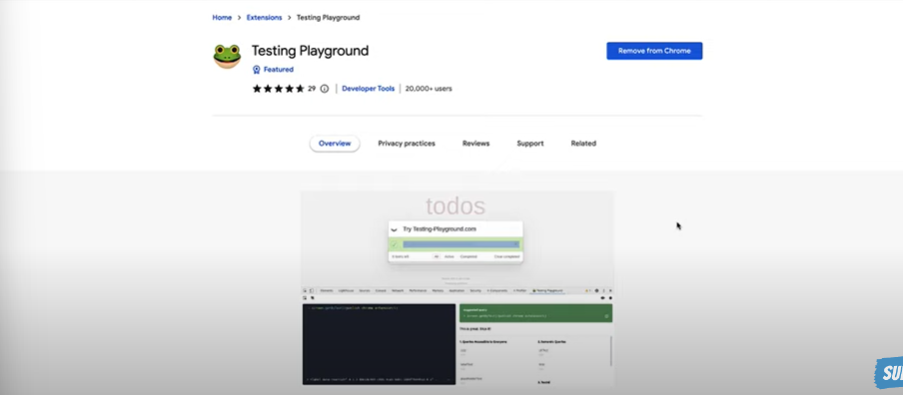
logRoles()🡺Helpful for finding ways to query the dom under test with getByRoll method. Print out a list of all the implicit aria rules within the dom tree.

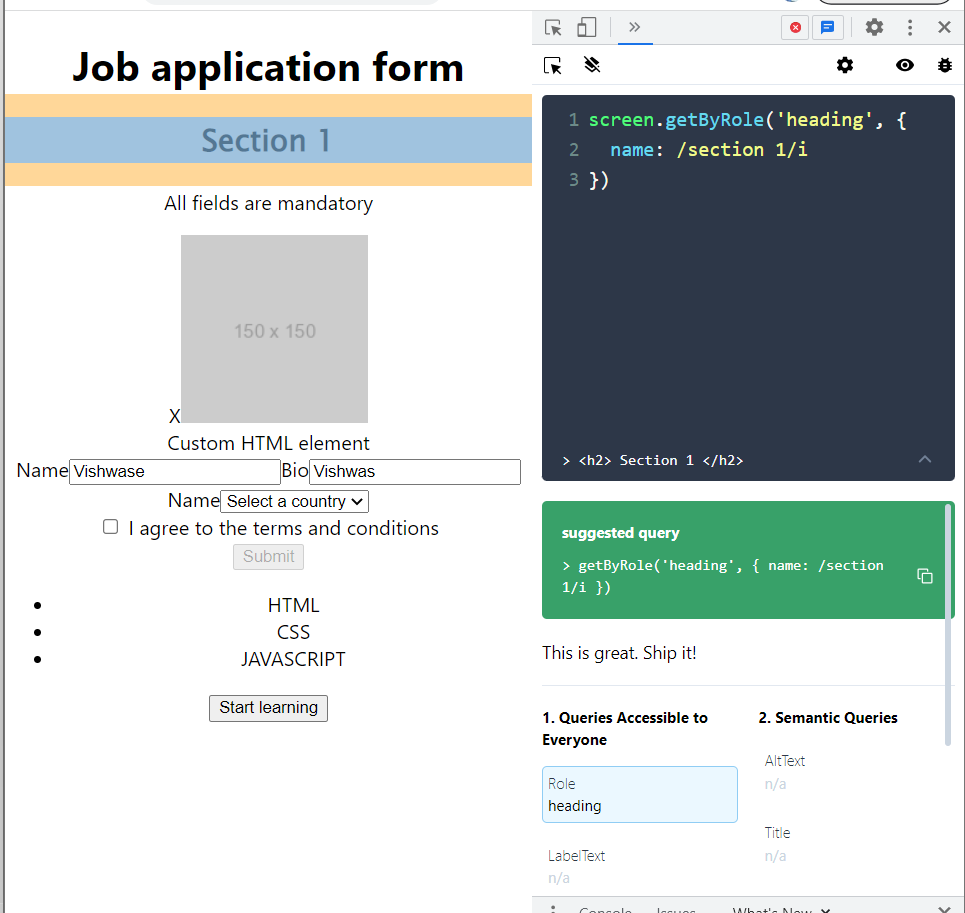
Une image contenant texte, moniteur, capture d’écran, écran

Description générée automatiquement

1. Testing Playground

Testing playground is an extension. That helps to find the best queries to select element when working with the testing library ,it encouraging good testing practices.





1. User interactions
   1. Introduction

User interaction is installed by default when we create our React app but we will install the latest version **npm i @testing-library/user-event@latest**

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

* 1. Pointer interactions

Une image contenant texte

Description générée automatiquement

const incrementButton = screen.getByRole("button", {

      name: "Increment",

    });

    await user.click(incrementButton);

    const countElement = screen.getByRole("heading");

    expect(countElement).toHaveTextContent("1");

Une image contenant texte

Description générée automatiquement

Counter.tsx

import { useState } from "react";

export const Counter = () => {

  const [count, setCount] = useState(0);

  const [amount, setAmount] = useState(0);

  return (

    <div>

      <h1>{count}</h1>

      <button onClick={() => setCount((count) => count + 1)}>Increment</button>

    </div>

  );

};

Counter.test.tsx

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

import { Counter } from "./Counter";

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

test("renders a count of 1 after clicking the increment button", async () => {

    user.setup();

    render(<Counter />);

    const incrementButton = screen.getByRole("button", {

      name: "Increment",

    });

    await user.click(incrementButton);

    const countElement = screen.getByRole("heading");

    expect(countElement).toHaveTextContent("1");

  });

  test("renders a count of 10 after clicking the increment button twice", async () => {

    user.setup();

    render(<Counter />);

    const incrementButton = screen.getByRole("button", {

      name: "Increment",

    });

    for (let i = 0; i < 10; i++) {

      await user.click(incrementButton);

    }

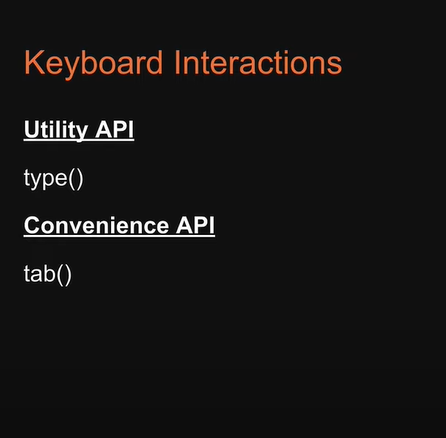
    const countElement = screen.getByRole("heading");

    expect(countElement).toHaveTextContent("10");

  });

});

* 1. Keyboard interaction



Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Counter.test.tsx

test("renders a count of 10 after clicking the 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")

  })

  test("elements are focused in theb right order",async ()=>{

     user.setup()

     render(<Counter/>)

     const amountInput= screen.getByRole("spinbutton")

     const setButton=screen.getByRole("button",{name:"Set"})

     const incrementButton=screen.getByRole("button",{name:"Increment"})

     await user.tab()

     expect(incrementButton).toHaveFocus();

     await user.tab()

     expect(amountInput).toHaveFocus();

     await user.tab()

     expect(setButton).toHaveFocus();

  })

* 1. Summary

Une image contenant texte

Description générée automatiquement

1. Test component wrapper in providers and test custom hooks
   1. Test component wrapper in providers

Sometimes component need a context provider to work as expected when it comes to test such components we can rely on the wrapper option that can be passed into the render function we specify our provider as its value which will allow us to test the individual component without having to rely on the entire component tree where the context provider is present

App.tsx

<AppProviders>

        <div className=”App”>

          <MuiMode />

        </div>

      </AppProviders>

AppProviders.tsx

import { ThemeProvider, createTheme } from ‘@mui/material/styles’

import CssBaseline from ‘@mui/material/CssBaseline’

const theme = createTheme({

  palette: {

    mode: ‘dark’,

  },

})

export const AppProviders = ({ children }: { children: React.ReactNode }) => {

  return (

    <ThemeProvider theme={theme}>

      <CssBaseline />

      {children}

    </ThemeProvider>

  )

}

MuiMode.tsx

import { useTheme } from ‘@mui/material/styles’

import { Typography } from ‘@mui/material’

export const MuiMode = () => {

  const theme = useTheme()

  return (

    <>

      <Typography component=”h1”>{`${theme.palette.mode} mode`}</Typography>

    </>

  )

}

Mui-mode.test.tsx

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

import { MuiMode } from “./MuiMode”;

import { AppProviders } from “../provider/AppProviders”;

describe(“MuiMode”, () => {

  test(“renders text correctly”, () => {

    render(<MuiMode />,{

        wrapper:AppProviders

    });

    const headingElement = screen.getByRole(“heading”);

    expect(headingElement).toHaveTextContent(“dark mode”);

  }) ;

}) ;

* 1. Custom Render Functions

The context provider wraps only a few components however for something like a theme provider or redux store provider it make sense to have the wrapper at the top level and available for every component and this also save us from having to specify the wrapper option in every test , **now the way to achieve a single wrapper across all tests is by writing a custom render function** .

Custom render Function🡺

Test-utils.tsx

import { ReactElement } from 'react'

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

import { AppProviders } from './Component/provider/AppProviders'

const customRender = (

  ui: ReactElement,

  options?: Omit<RenderOptions, 'wrapper'>

) => render(ui, { wrapper: AppProviders, ...options })

export \* from '@testing-library/react'

export { customRender as render }

mui-mode.test.tsx

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

import { MuiMode } from "./MuiMode";

import { render } from "../../test-utils";

describe("MuiMode", () => {

  test("renders text correctly", () => {

    render(<MuiMode />);

    const headingElement = screen.getByRole("heading");

    expect(headingElement).toHaveTextContent("dark mode");

  });

});

* 1. Custom React Hooks

For testing react hooks we do not rely on render and screen from react testing library instead we rely on render hook which returns a result which in turn contains a current property that has all the returns values of the custom hook and if you have to invert the hook you can pass in an options object to render hook with the key called initial props for our example we’ve used a counter hook to verify if the initial count renders

userCounter.types.ts

export type UseCounterProps = {

    initialCount?: number

  }

useCounter.tsx

import { useState } from 'react'

import { UseCounterProps } from './userCounter.types'

export const useCounter = ({ initialCount = 0 }: UseCounterProps = {}) => {

  const [count, setCount] = useState(initialCount)

  const increment = () => setCount(count + 1)

  const decrement = () => setCount(count - 1)

  return { count, increment, decrement }

}

useCounter.test.tsx

import { renderHook } from "@testing-library/react"

import { useCounter } from "./useCounter"

describe("useCounter",()=>{

    test("should render the 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)

    })

})

* 1. Act utility

In the context of the React Testing Library, "act" is a utility function provided by the React testing framework that helps ensure that updates to the user interface are properly processed and rendered before the test continues.

When you use React to update the user interface, those updates are typically asynchronous. This means that when you run tests, React may not have finished updating the interface by the time the test finishes running. As a result, you may end up with test failures or unexpected behavior.

To avoid this, you can use the "act" utility function provided by the React testing framework. "act" lets you explicitly wait for updates to the user interface to be processed and rendered before your test continues. This helps ensure that your tests accurately reflect the state of the user interface.

You should wrap any code that updates the user interface within the "act" function when you are testing your React components. This includes rendering components, clicking buttons, typing into input fields, and other user interactions that trigger updates to the user interface.

useCounter.test.tsx

import { renderHook } from "@testing-library/react";

import { useCounter } from "./useCounter";

import { act } from "react-dom/test-utils";

describe("useCounter", () => {

  test("should increment and decrement the count", () => {

    const { result } = renderHook(useCounter);

    act(() => result.current.increment());

    expect(result.current.count).toBe(1);

    act(() => result.current.decrement());

    expect(result.current.count).toBe(0);

  });

  test("should decrement the count", () => {

    const { result } = renderHook(useCounter);

    act(() => result.current.decrement());

    expect(result.current.count).toBe(-1);

  });

});

Une image contenant texte

Description générée automatiquement

1. Mocking
   1. Mocking Function

In Jest, a mocking function is a utility that allows you to create a "fake" version of a module, function, or object, which can be used in place of the real one during testing. By doing this, you can isolate the code you are testing and control the behavior of its dependencies, making it easier to write reliable and predictable tests.

Mocking functions can be especially useful when testing code that has external dependencies, such as APIs or libraries that make network requests. By using a mock function instead of the actual dependency, you can test your code in isolation, without worrying about external factors that may affect the outcome of your tests.

CounterTwo.types.ts

export type CounterTwoProps = {

    count: number

    handleIncrement?: () => void

    handleDecrement?: () => void

  }

CounterTwo.tsx

import { CounterTwoProps } from './CounterTwo.types'

export const CounterTwo = (props: CounterTwoProps) => {

  return (

    <div>

      <h1>Counter Two</h1>

      <p>{props.count}</p>

      {props.handleIncrement && (

        <button onClick={props.handleIncrement}>Increment</button>

      )}

      {props.handleDecrement && (

        <button onClick={props.handleDecrement}>Decrement</button>

      )}

    </div>

  )

}

CounterTwo.test.tsx

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

import { CounterTwo } from "./CounterTwo";

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

describe("CounterTwo", () => {

  test("renders correctly", () => {

    render(<CounterTwo count={0} />);

    const textElement = screen.getByText("Counter Two");

    expect(textElement).toBeInTheDocument();

  });

  test("handlers are called", async () => {

    user.setup();

    const incrementHandler = jest.fn();

    const decrementHandler = jest.fn();

    render(

      <CounterTwo

        count={0}

        handleIncrement={incrementHandler}

        handleDecrement={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)

  });

});

* 1. Mocking http requests

When testing code that makes HTTP requests, it's often useful to mock those requests so that your tests can run quickly and reliably without depending on external APIs or resources.

MSW (Mock Service Worker) is a popular library for mocking HTTP requests in JavaScript. It provides a powerful and flexible API for intercepting and mocking HTTP requests, allowing you to easily test your code without relying on external APIs or services.

users.tsx

import { useState, useEffect } from 'react'

export const Users = () => {

  const [users, setUsers] = useState<string[]>([])

  const [error, setError] = useState<string | null>(null)

  useEffect(() => {

    fetch('https://jsonplaceholder.typicode.com/users')

      .then((res) => res.json())

      .then((data) => setUsers(data.map((user: { name: string }) => user.name)))

      .catch(() => setError('Error fetching users'))

  }, [])

  return (

    <div>

      <h1>Users</h1>

      {error && <p>{error}</p>}

      <ul>

        {users.map((user) => (

          <li key={user}>{user}</li>

        ))}

      </ul>

    </div>

  )

}

8.2.1. MSW Setup

Install

npm install msw --save-dev

server.ts

// src/mocks/server.js

import { setupServer } from 'msw/node'

import { handlers } from './handlers'

// This configures a request mocking server with the given request handlers.

export const server = setupServer(...handlers)

handlers.ts

import {rest} from 'msw'

export const handlers =[

    rest.get('https://jsonplaceholder.typicode.com/users',(req,res,ctx)=>{

        return res(

            ctx.status(200),

            ctx.json([

                {name:"Bruce wayne"},

                {name:"Bruce wayne"},

                {name:"Bruce wayne"}

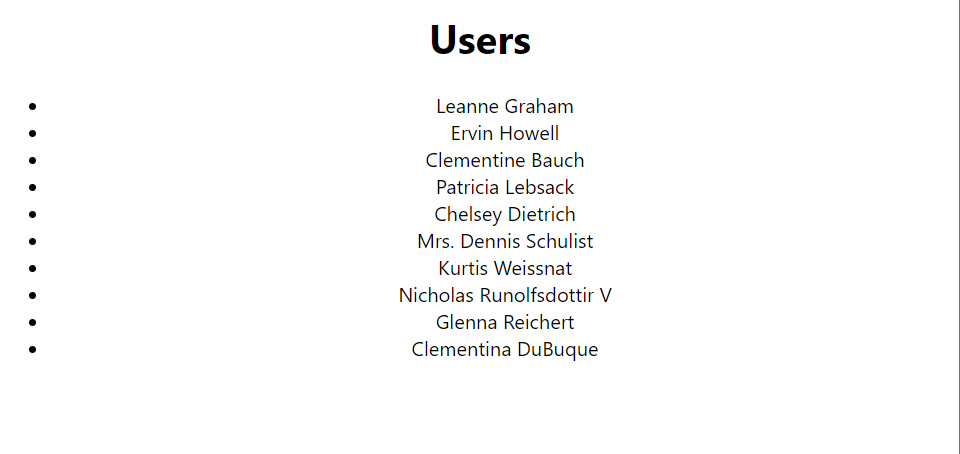
            ])

        )

    })

]

* + 1. Testing with MSW



Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

We need to configure our Msw in our app so we need to change the file setup test

Setuo.Test.ts

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

* + 1. MSW Error Handling

We ca reset handlers per test to ensure error responses are handled by our test we can response with a status of 500 and assert against the right element present in the dom

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

import { Users } from "./users"

import { server } from "../../mocks/server"

import { rest } from "msw"

describe('Users', () => {

 test('renders error',async ()=>{

        server.use(

            rest.get(

                "https://jsonplaceholder.typicode.com/users",

                (req,res,ctx)=>{

                    return res(ctx.status(500))

                }

            )

        )

        render(<Users/>)

        const error =await screen.findByText("Error fetching users")

        expect(error).toBeInTheDocument()

    })

})

* + 1. Summary

Une image contenant texte

Description générée automatiquement

1. Static Analysis Testing

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

* 1. Eslint

ESLint is a popular open-source static code analysis tool that is used to analyze and identify potential problems or errors in JavaScript and TypeScript code. It helps developers to maintain a consistent coding style and avoid common mistakes by enforcing coding standards and best practices.

ESLint can be customized to match the coding style and conventions of a specific project or team, which makes it a useful tool for maintaining code quality and consistency. It can be configured to flag issues such as syntax errors, unused variables, undefined variables, and other common coding mistakes.

ESLint is highly extensible and can be configured to support a variety of coding styles and frameworks, including React, Angular, Vue.js, and Node.js. It also supports a range of plugins and rule sets, which allow developers to extend and customize the set of rules used to check their code.

In summary, ESLint is a valuable tool for ensuring code quality and consistency in JavaScript and TypeScript projects. By enforcing coding standards and best practices, it helps developers to write cleaner, more maintainable code that is less prone to errors and bugs.

**Create react app install Eslint by default.**

Une image contenant texte

Description générée automatiquement

* eslint-plugin-jest-dom

**eslint-plugin-jest-dom** is a plugin for ESLint that provides additional rules for working with Jest DOM, a set of custom matchers for testing web applications. Jest DOM provides a set of matchers that are specifically designed for testing web applications, such as **toHaveTextContent** and **toHaveAttribute**. These matchers make it easier to write test cases that are more readable and maintainable.

The **eslint-plugin-jest-dom** plugin provides additional rules for ESLint that help enforce best practices when working with Jest DOM. For example, the **prefer-checked** rule can be used to enforce the use of the **checked** attribute instead of the **defaultChecked** property when testing form inputs. This helps to ensure that tests are consistent and more easily maintainable.

Other rules provided by the plugin include **prefer-enabled-disabled**, which enforces the use of the **disabled** attribute instead of the **defaultDisabled** property, and **prefer-focus**, which enforces the use of the **focus** method instead of the **document.activeElement** property.

By using the **eslint-plugin-jest-dom** plugin, developers can ensure that their test code follows best practices when working with Jest DOM. This can lead to more maintainable and easier-to-read test cases, which in turn can help improve the quality and reliability of the code being tested.

* + Install
    - npm install --save-dev eslint-plugin-jest-dom

Package.json

"eslintConfig": {

    "extends": [

      "react-app",

      "react-app/jest",

      "plugin:jest-dom/recommended"

    ]

  },

Une image contenant texte

Description générée automatiquement

* 1. Prettier

Install Prettier formatted code from vsCode it will do.

* 1. Husky

Husky is a popular open-source tool that allows developers to define Git hooks using simple configuration files. Git hooks are scripts that are executed before or after specific Git commands, such as **commit**, **push**, or **merge**. They can be used to automate tasks such as running tests, linting code, or formatting files.

Husky provides an easy-to-use interface for defining Git hooks using JavaScript or shell scripts. It integrates with popular JavaScript tools such as ESLint, Prettier, and Jest, making it easy to run these tools automatically when specific Git commands are executed.

One of the key benefits of using Husky is that it helps to enforce consistent development practices across a team. By running automated tasks such as linting and testing before code is committed or pushed, it helps to catch errors and maintain code quality.

Husky can also be used to customize Git workflows by defining custom hooks that execute scripts based on specific events or conditions. For example, a team could define a custom hook that runs a deployment script when code is pushed to a specific branch.

In summary, Husky is a powerful tool for automating common development tasks and enforcing best practices in Git workflows. By using Husky, developers can save time and improve the quality of their code, while ensuring that their workflows are consistent and maintainable.

* 1. Lint-staged

**lint-staged** is an open-source tool that is used to run linters and other automated tasks on staged files before they are committed to a Git repository. Staged files are the files that have been modified and are ready to be committed using the Git **git add** command.

The **lint-staged** tool allows developers to define a set of tasks that are executed on each staged file. These tasks can include linting code, formatting code, or running tests. **lint-staged** uses Git hooks to ensure that these tasks are run only on staged files, rather than on the entire codebase.

One of the key benefits of using **lint-staged** is that it helps to catch errors and enforce coding standards before code is committed to a repository. By running linters and other automated tasks on staged files, developers can ensure that the code being committed is clean, readable, and maintainable. This can help to improve the quality of the codebase and reduce the number of bugs and errors.

**lint-staged** can be easily integrated with popular JavaScript tools such as ESLint, Prettier, and Jest, making it easy to run these tools automatically when staged files are committed. It also supports parallel processing, which can help to speed up the execution of tasks on large codebases.

In summary, **lint-staged** is a powerful tool for enforcing coding standards and best practices in Git workflows. By automating common development tasks and running them on staged files, it helps to catch errors and improve code quality before code is committed to a repository.

1. Course Summary
2. Summary

**First Section**

Une image contenant texte

Description générée automatiquement

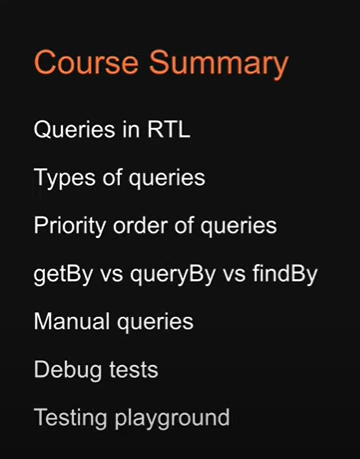
**Second section**

**Third section**

Une image contenant texte

Description générée automatiquement

Fourth section

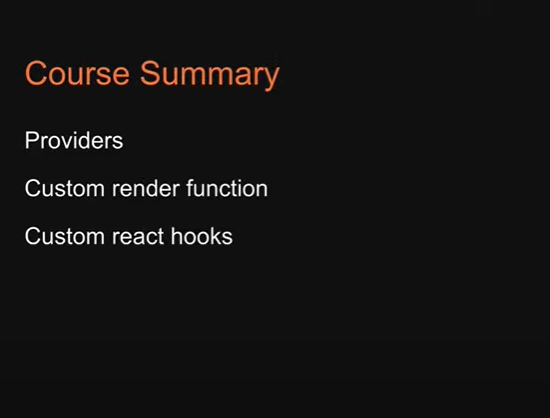


Five sections

Une image contenant texte

Description générée automatiquement

Sixe section



Seven section

Une image contenant texte

Description générée automatiquement

Last section

