React Testing Library

1. Introduction
   1. Definition RTL

La bibliothèque de test React Testing Library est une bibliothèque qui fournit des utilitaires et des méthodes pour faciliter le test des composants React. Elle se concentre sur le test du comportement réel des composants, en simulant les interactions utilisateur et en effectuant des assertions sur le rendu et le fonctionnement du composant dans le contexte de l'application.

* 1. Pourquoi on fait des tests automatises

les tests automatisés sont essentiels pour maintenir un code de qualité, faciliter la détection précoce des erreurs, garantir le bon fonctionnement des fonctionnalités et permettre une maintenance efficace du code au fil du temps. Ils jouent un rôle clé dans l'amélioration de la fiabilité et de la robustesse des applications logicielles.

* 1. Difference entre Jest et RTL

1. Jest :

* Jest est un framework de test JavaScript.
* Jest est un exécuteur de tests qui trouve les tests, les exécute, détermine s'ils ont réussi ou échoué, et rapporte les résultats de manière lisible par les humains.

1. React Testing Library :

* Utilitaire de test JavaScript qui fournit un DOM virtuel pour tester les composants React.
* React Testing Library fournit un DOM virtuel avec lequel nous pouvons interagir et vérifier le comportement d'un composant React.
* Testing Library est en fait une famille de packages qui aide à tester les composants UI.
* La bibliothèque principale s'appelle DOM Testing Library et RTL est simplement une surcouche de cette bibliothèque principale pour tester les applications React de manière plus facile.
  1. Les 3 types de test

Une image contenant triangle, cône

Description générée automatiquement

Les tests unitaires, les tests d'intégration et les tests E2E (End-to-End) sont souvent représentés sous la forme d'une pyramide de tests pour illustrer leur répartition et leur niveau d'effort. Voici une brève explication de chaque type de test dans le contexte de la pyramide des tests :

1. Tests unitaires : Les tests unitaires se situent à la base de la pyramide. Ils sont axés sur la validation du comportement précis d'une unité de code isolée, telle qu'une fonction ou une classe. Les tests unitaires sont généralement écrits et exécutés par les développeurs. Ils sont rapides à exécuter et fournissent un retour rapide sur les erreurs de logique ou de comportement spécifique d'une unité de code. Ils sont nombreux et relativement peu coûteux à maintenir.
2. Tests d'intégration : Les tests d'intégration se situent au niveau intermédiaire de la pyramide. Ils vérifient la communication et l'intégration entre plusieurs composants ou modules d'un système. Les tests d'intégration valident les interactions entre différentes parties du système, telles que les couches de l'application ou les services tiers. Ils sont moins nombreux que les tests unitaires, mais plus coûteux à écrire et à maintenir car ils nécessitent une configuration plus complexe.
3. Tests E2E (End-to-End) : Les tests E2E se situent au sommet de la pyramide. Ils valident le comportement global de l'application dans son environnement réel, en simulant les interactions utilisateur réelles. Les tests E2E couvrent des scénarios complets et vérifient que l'application fonctionne correctement de bout en bout. Ils sont moins nombreux mais plus coûteux à écrire et à maintenir en raison de leur exécution plus lente et de leur dépendance à l'égard de l'interface utilisateur.

La pyramide des tests suggère que la majorité des tests devraient être des tests unitaires, suivis des tests d'intégration, tandis que les tests E2E devraient être moins nombreux mais tout de même présents pour valider l'application dans son ensemble. Cette approche permet d'obtenir une couverture efficace des différents niveaux de test, en favorisant des tests rapides, fiables et maintenables.

* 1. Test Driven Developement (TDD) Or RedGreenTesting

Le TDD (Test-Driven Development) est une méthode de développement logiciel qui consiste à écrire les tests avant d'écrire le code de production. Il suit un cycle itératif : écrire un test, voir le test échouer, écrire le code minimal pour que le test réussisse, puis refactoriser si nécessaire. Le TDD vise à améliorer la qualité du code, à favoriser la conception modulaire et à faciliter la maintenance à long terme.

1. Jest
   1. Filtering test
      * Watch mode

Le mode Watch en Jest permet à Jest de surveiller les fichiers et de réexécuter automatiquement les tests lorsqu'ils sont modifiés.

* + - Filtring Test

Une image contenant texte

Description générée automatiquement

* + - 1. Only et skip

Only

// 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

// 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()

})

* 1. Filename Convention
* Files with .test.js or .test.tsx or .spec.js or .spec.tsx.tsx suffix
* Files with .js ir .tsx suffix in \_\_tests\_\_ foldes

Recommendation is to always put your tests next to the code they are testing so that relative imports are shorter.

* 1. Code coverage

Une métrique qui peut aider à comprendre à quel point mon code logiciel est testé :

* Couverture des instructions (Statement coverage) : combien d'instructions du code logiciel ont été exécutées.
* Couverture des branches (Branches coverage) : combien de branches de la structure de contrôle (par exemple, les instructions "if") ont été exécutées.
* Couverture des fonctions (Function coverage) : combien de fonctions définies ont été appelées.
* Couverture des lignes (Line coverage) : combien de lignes du code source ont été testées.

**Configiration**

"scripts": {

    "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}

**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

**html file of our coverage**

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.

Une image contenant texte, nombre, ligne, capture d’écran

Description générée automatiquementUne image contenant texte, capture d’écran, ligne

Description générée automatiquement

**Resume**

La couverture de code est une métrique qui peut aider à comprendre dans quelle mesure le code logiciel est testé. Nous avons appris l'option de couverture de code dans Jest pour générer le rapport. Nous avons également utilisé l'indicateur "watch all" pour générer la couverture pour tous les tests du projet. Nous avons spécifié les fichiers à inclure ou exclure de la collecte de la couverture de code en utilisant l'indicateur "collect coverage from". Enfin, nous avons appris comment définir les seuils de couverture en utilisant la configuration de Jest dans le fichier package.json. Jest échouera si la couverture ne satisfait pas les exigences du seuil. La couverture de code est très importante d'un point de vue CI/CD.

1. React testing Library
   1. **Quoi tester et quoi ne pas tester**
      * 1. Quoi tester
      * 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.

* 1. Quoi ne pas tester

o Détails de l'implémentation : Nous voulons tester le comportement et non la façon dont le comportement est implémenté, ce qui facilite également la refactorisation.

o Code tiers : Nous devrions tester notre code et non le code que nous consommons à partir d'une bibliothèque externe. Par exemple, si nous utilisons Material UI, nous n'avons pas besoin de tester le composant de bouton ou les composants d'onglets de l'interface utilisateur.

o Code qui n'est pas important du point de vue de l'utilisateur : Par exemple, si nous avons écrit une fonction utilitaire qui affiche une date dans un format convivial pour l'utilisateur, nous n'avons pas besoin de tester si la fonction a été appelée par le composant. Au lieu de cela, nous pouvons directement tester si la date a été rendue dans le format attendu.

* 1. **RTL QUERIES**

1. **Définition**

Les requêtes (queries) sont les méthodes que Testing Library met à votre disposition pour trouver des éléments sur la page.

1. **La difference entre getBy et findBy et queryBy en ReactTesting Library**

La différence entre **getBy**, **findBy** et **queryBy** dans React Testing Library réside dans leur comportement lorsqu'aucun élément correspondant n'est trouvé :

1. **getBy** : Les requêtes commençant par **getBy** sont utilisées pour sélectionner un élément du DOM qui est censé être présent dans le rendu. Si aucun élément correspondant n'est trouvé, une erreur est levée, indiquant que l'élément recherché est manquant. Cela signifie que **getBy** est utilisé pour rechercher des éléments qui doivent absolument être présents pour que le test réussisse.
2. **findBy** : Les requêtes commençant par **findBy** sont utilisées pour rechercher des éléments asynchrones ou des éléments qui peuvent être rendus de manière asynchrone dans le DOM. Elles renvoient une promesse qui se résout lorsqu'un élément correspondant est trouvé dans le DOM, ou rejette une erreur si aucun élément n'est trouvé dans le délai imparti. Cela permet d'effectuer des tests asynchrones et d'attendre que les éléments soient rendus avant d'effectuer des assertions.
3. **queryBy** : Les requêtes commençant par **queryBy** sont utilisées pour sélectionner un élément du DOM, mais elles renvoient **null** au lieu de lever une erreur si aucun élément correspondant n'est trouvé. Cela permet de vérifier la présence facultative d'éléments dans le rendu. Si l'élément est trouvé, il peut être utilisé pour effectuer des assertions. Si aucun élément n'est trouvé, **queryBy** renvoie **null**, ce qui permet de gérer les cas où la présence de l'élément est facultative.

En résumé, **getBy** est utilisé pour les éléments qui doivent absolument être présents, **findBy** est utilisé pour les éléments rendus de manière asynchrone, et **queryBy** est utilisé pour les éléments dont la présence est facultative.

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

1. **Priority**

Based on [the Guiding Principles](https://testing-library.com/docs/guiding-principles), your test should resemble how users interact with your code (component, page, etc.) as much as possible. With this in mind, we recommend this order of priority:

1. **Queries Accessible to Everyone** Queries that reflect the experience of visual/mouse users as well as those that use assistive technology.
   1. getByRole: permet de sélectionner un élément du DOM en fonction de son rôle,
   2. getByLabelText: en RTL est une méthode qui permet de sélectionner un élément de formulaire en fonction de son étiquette associée(label).
   3. getByPlaceholderText: en RTL est une méthode qui permet de sélectionner un élément de formulaire en fonction de son attribut "placeholder" dans le DOM.
   4. getByText: recherchera l'élément qui a un nœud texte avec un contenu correspondant au texte donné. Généralement, vous l'utiliseriez pour trouver des éléments de paragraphe, de div ou de span.
   5. getByDisplayValue: renvoie l'élément **<input>**, **<textarea>** ou **<select>** qui a une valeur affichée correspondante.
2. **Semantic Queries** HTML5 and ARIA compliant selectors. Note that the user experience of interacting with these attributes varies greatly across browsers and assistive technology.
   1. getByAltText: renverra l'élément qui a le texte alternatif donné. Cette méthode prend uniquement en charge les éléments qui acceptent un attribut alt comme <img>, <input>, <area> ou du HTML personnalisé.
   2. getByTitle: getByTitle renvoie l'élément qui a un attribut title correspondant.
3. **Test IDs**
   1. getByTestId: getByTestId renvoie l'élément qui a un attribut data-testid correspondant.
4. **Utilisation des Queries**

**screen**

En RTL (React Testing Library), l'objet **screen** est une utilité qui regroupe plusieurs méthodes de requête (queries) pour trouver des éléments dans le DOM. Il facilite l'accès aux méthodes de requête sans avoir à les importer individuellement.

1. **Queies**
   1. **ByRole**

La méthode **getByRole** en RTL permet de sélectionner un élément du DOM en fonction de son rôle. Voici un exemple d'utilisation :

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

Cette ligne de code sélectionne un élément de type bouton dans le DOM et le stocke dans la variable **buttonElement**.

expect(buttonElement).toBeInTheDocument();

Cette ligne de code vérifie si le bouton présent dans le DOM

1. **ByRoleOptions**

La méthode getByRole en RTL offre la possibilité d'utiliser des options pour filtrer les éléments en fonction de leur rôle et d'autres attributs. Voici comment utiliser les options avec getByRole :

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

      level: 1,

      name: "Job application form",

    });

Cette ligne de code sélectionne un élément de type heading avec level H1 et name : Job application form dans le DOM et le stocke dans la variable **h1Heading**.

Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement

* 1. getByLabelText

**getByLabelText** en RTL est une méthode qui permet de sélectionner un élément de formulaire en fonction de son étiquette associée(label).

 // 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();

options de sélection supplémentaires, telles que **{ selector: 'input' }** pour filtrer les éléments de type **<input>**.

* 1. **getByPlaceholderText**

**getByPlaceholderText** en RTL est une méthode qui permet de sélectionner un élément de formulaire en fonction de son attribut "placeholder" dans le DOM.

const nameElement3 = screen.getByPlaceholderText("FullName");

    expect(nameElement3).toBeInTheDocument();

* 1. **getByText**

**getByText** recherchera l'élément qui a un nœud texte avec un contenu correspondant au texte donné. Généralement, vous l'utiliseriez pour trouver des éléments de paragraphe, de div ou de span.

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

    expect(paragraphElement).toBeInTheDocument();

* 1. **getByDisplayValue**

**getByDisplayValue** renvoie l'élément **<input>**, **<textarea>** ou **<select>** qui a une valeur affichée correspondante.

 const nameElement4=screen.getByDisplayValue('Vishwase')

  expect(nameElement4).toBeInTheDocument()

* 1. **getByAltText**

**getByAltText** renverra l'élément qui a le texte alternatif donné. Cette méthode prend uniquement en charge les éléments qui acceptent un attribut **alt** comme **<img>**, **<input>**, **<area>** ou du HTML personnalisé.

 const imageElement=screen.getByAltText('a person with a laptop')

 expect(imageElement).toBeInTheDocument()

* 1. **getByTitle**

**getByTitle** renvoie l'élément qui a un attribut **title** correspondant.

const closeElement = screen.getByTitle("close");

    expect(closeElement).toBeInTheDocument();

* 1. **getByTestId**

**getByTestId** renvoie l'élément qui a un attribut **data-testid** correspondant.

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

expect(customElement).toBeInTheDocument()

1. **l'ordre de priorité recommandé pour les requêtes (queries) en React Testing Library**

Si nous ne pouvons pas tester notre composant avec les premières requêtes, nous pouvons passer aux suivantes dans l'ordre de priorité.

ByRole=>ByLabelText=>ByPlaceHolderText=>ByText=>ByDisplayValue=>ByAltText=>ByTitle=>ByTestId

1. **Query multiple elements**

La recherche de plusieurs éléments correspondants à l'aide de méthodes de requête dans React Testing Library.

<ul>

        {skills.map((skill) => {

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

        })}

      </ul>

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 automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

1. **QueryBy**

L'utilisation de **queryBy** est utile lorsque vous vous attendez à ce que l'élément puisse ne pas être présent dans le DOM, et vous voulez vérifier sa présence de manière non intrusive sans provoquer d'échec de test.

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**



**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**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.queryByRole("button", {

      name: "Start learning",

    });

    expect(startLearningButton).not.toBeInTheDocument();

  });

1. **FindBy**

L'utilisation de **findBy** est utile lorsque vous vous attendez à ce que l'élément puisse ne pas être immédiatement présent dans le DOM, par exemple lorsqu'il est chargé de manière asynchrone ou lorsqu'il dépend de certaines interactions utilisateur. Cela vous permet d'attendre que l'élément recherché apparaisse dans le DOM avant de poursuivre l'exécution du test.

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**



  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>

      )}

    </>

  )

}

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()** est une méthode fournie par React Testing Library dans Jest qui permet d'afficher une représentation textuelle de l'état actuel du DOM rendu par votre composant.

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

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

  render(<MyComponent />);

  // Affiche le rendu actuel du DOM

  screen.debug();

});

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

* 1. **Testing Playground**

L'extension "Testing Playground" est un outil développé pour les navigateurs Chrome et Firefox qui permet de faciliter le processus de test en proposant une interface conviviale pour écrire, exécuter et déboguer des tests unitaires JavaScript.

Une image contenant texte, logiciel, Système d’exploitation, Page web

Description générée automatiquementUne image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

* 1. **User interaction**
     + 1. **Introduction**
     + **Définition**

RTL encourage les tests basés sur le comportement de l'utilisateur plutôt que sur les détails d'implémentation. Cela signifie que les tests doivent simuler les interactions de l'utilisateur avec le composant et vérifier les résultats attendus en fonction de ces interactions.

* + - **userEvent**

La bibliothèque "user-event" est une extension de React Testing Library (RTL) qui facilite encore plus la simulation d'interactions utilisateur dans les tests. Elle fournit des méthodes supplémentaires pour simuler des interactions utilisateur réalistes, telles que la saisie de texte, la sélection d'options, la coche de cases à cocher, etc.

* + - **Differrence entre fireEvent et userEvent**
* **fireEvent** : C'est une fonction fournie par RTL qui permet de déclencher des événements sur les éléments DOM. Elle simule des interactions basiques telles que les clics, les changements de valeur, les soumissions de formulaire, etc. Cependant, elle ne simule pas les interactions utilisateur de manière réaliste. Par exemple, lors de la saisie de texte avec **fireEvent.change(inputElement, { target: { value: 'valeur' } })**, aucun événement de frappe n'est généré pour chaque caractère.
* **userEvent** : C'est une extension de RTL qui fournit des méthodes supplémentaires pour simuler des interactions utilisateur réalistes. Elle vise à reproduire le comportement réel de l'utilisateur lors de l'interaction avec l'interface utilisateur. Par exemple, lors de la saisie de texte avec **userEvent.type(inputElement, 'valeur')**, des événements de frappe sont générés pour chaque caractère, ce qui permet de tester de manière plus précise le comportement du composant.

En résumé, **fireEvent** est plus simple et basique, tandis que **userEvent** fournit des méthodes plus avancées et réalistes pour simuler les interactions utilisateur. Dans la plupart des cas, il est recommandé d'utiliser **userEvent** pour des tests plus réalistes et expressifs. Cependant, il peut être nécessaire d'utiliser **fireEvent** dans certaines situations spécifiques où les interactions de bas niveau sont nécessaires.

* + - 1. **Pointer interaction**

Lorsque vous utilisez la bibliothèque **user-event**, vous disposez de plusieurs méthodes pour simuler différentes interactions utilisateur dans vos tests. Voici quelques-unes des interactions les plus couramment utilisées :

* **userEvent.click(element)**: Simule un clic sur un élément.
* **userEvent.type(element, text)**: Simule la saisie de texte dans un élément, caractère par caractère.
* **userEvent.clear(element)**: Simule la suppression du contenu d'un élément de formulaire.
* **userEvent.selectOptions(element, values)**: Simule la sélection d'options dans un élément de liste déroulante (**<select>**).
* **userEvent.hover(element)**: Simule le survol d'un élément avec la souris.
* **userEvent.tab()**: Simule l'appui sur la touche Tab du clavier.
* **userEvent.keyboard('{key}')**: Simule l'appui sur une touche spécifique duclavier. Une image contenant texte

  Description générée automatiquement

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>

  );

};

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

      name: "Increment",

    });

    await user.click(incrementButton);

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

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

* + - 1. **Keybord Interaction**

L'interaction avec le clavier est une partie essentielle des tests d'interface utilisateur. Lorsqu'il s'agit d'interagir avec des éléments à l'aide du clavier dans les tests avec React Testing Library (RTL)

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

     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();

  })

Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

* + - 1. **Les interaction les plus utilise avec user-event**
* **userEvent.click**
* **userEvent.** **dblClick**
* **userEvent.** **hover**
* **userEvent.** **unhover**
* **userEvent.** **selectOptions**
* **userEvent.** **deselectOptions**
* **userEvent.type**
* **userEvent.keyboard**
* **userEvent.upload**
* **userEvent.clear**
* **userEvent.tab**
* **userEvent.paste**
  1. **Test component wrapper in providers**
     + 1. 1ere façon

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

    });

// or

// render(

    //   <AppProviders>

    //     <div className="App">

    //       <MuiMode />

    //     </div>

    //   </AppProviders>

    // );

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

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

  }) ;

}) ;

* + - 1. **Custom render**

La méthode "custom render" vous permet de créer une fonction de rendu personnalisée qui enveloppe votre composant dans les fournisseurs requis. Cela vous permet de réutiliser facilement cette fonction de rendu personnalisée dans vos cas de test.

Le fournisseur de contexte ne enveloppe que quelques composants, cependant, pour des éléments tels qu'un fournisseur de thème ou un fournisseur de magasin Redux, il est logique d'avoir le wrapper au niveau supérieur et disponible pour chaque composant. Cela nous évite également d'avoir à spécifier l'option de wrapper dans chaque test. Maintenant, la façon d'obtenir un seul wrapper pour tous les tests est d'écrire une fonction de rendu personnalisée.

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 & act** 
     + 1. **Custom React Hooks**

Pour tester les hooks React, nous ne nous appuyons pas sur la fonction render et l'objet screen de la bibliothèque de tests React. À la place, nous utilisons la fonction renderHook, qui renvoie un résultat contenant une propriété current. Cette dernière contient toutes les valeurs de retour du hook personnalisé. Si vous devez inverser le hook, vous pouvez passer un objet d'options à la fonction renderHook, avec une clé appelée initialProps.

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 }

}

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**

Dans React Testing Library, la fonction **act** est un utilitaire fourni par React lui-même. Elle est utilisée pour gérer les opérations asynchrones lors des tests. La fonction **act** permet de s'assurer que toutes les mises à jour d'état et les effets secondaires sont correctement exécutés avant de continuer le test.

Lorsque vous effectuez des opérations asynchrones telles que des appels à une API, des délais (**setTimeout**, **setInterval**), ou des mises à jour d'état asynchrones, vous devez envelopper ces opérations dans la fonction **act**. Cela garantit que les modifications d'état sont correctement propagées et que les composants réagissent comme prévu.

En utilisant **act**, vous pouvez attendre la résolution des opérations asynchrones avant de vérifier les résultats du test ou d'interagir avec les composants.

  act(() => {

    // rendu des composants

  });

  // exécution des assertions

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);

  });

* 1. **Mock & MSW**
     + 1. **Mock**

**Sur my jest doc y’as tous se qu’il faut avoir sur les mock**

Les mock functions sont des fonctions simulées qui remplacent les fonctions réelles lors des tests. Elles permettent de contrôler le comportement des fonctions simulées et de vérifier leur utilisation dans les tests. Les mock functions peuvent enregistrer les appels qui leur sont faits, définir des valeurs de retour simulées et effectuer d'autres actions spécifiques pour les tests.

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

    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. **MSW**

Pour les tests unitaires et fonctionnels ont n’utilise pas les apis réelles.

MSW (Mock Service Worker) est une bibliothèque de simulation de service worker utilisée pour intercepter et gérer les requêtes réseau dans les tests JavaScript. Elle permet de créer des mocks de requêtes HTTP, de simuler des réponses de serveur et de contrôler le comportement du réseau dans un environnement de test. Cela facilite la création de tests unitaires et d'intégration pour les applications qui effectuent des appels API.

1. Setup

Une image contenant texte

Description générée automatiquement

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

1. Msw Error Handling

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. **Static analysis testing**
     + 1. **Definition**

Le Static Analysis Testing, également connu sous le nom de test d'analyse statique, est une méthode de test qui analyse le code source d'un programme sans l'exécuter. Il vise à détecter les erreurs, les bugs potentiels et les violations des bonnes pratiques de programmation en se basant sur l'analyse statique du code. Cela permet d'identifier les problèmes avant même l'exécution du programme, améliorant ainsi la qualité et la fiabilité du code.

* Static analysis tools
  + Typescript
  + ESlint
  + Prettier
  + Husky
  + **Lint-staged**
    - 1. **ESlint**

ESLint est un outil d'analyse statique du code JavaScript largement utilisé. Il permet de détecter les erreurs de syntaxe, les problèmes de style, les pratiques non recommandées et les erreurs potentielles dans le code JavaScript. Il fournit des règles configurables et personnalisables pour maintenir la cohérence et la qualité du code. ESLint peut être intégré aux pipelines de développement pour assurer une meilleure qualité du code et faciliter la détection précoce des erreurs.

* + - 1. **Prettier**

Prettier est un outil de formatage de code qui permet d'automatiser et d'uniformiser la mise en forme du code source. Il prend en charge de nombreux langages de programmation, dont JavaScript, CSS, HTML, et bien d'autres. Prettier analyse le code et le reformate automatiquement en suivant des règles prédéfinies pour l'indentation, les espaces, les sauts de ligne, etc. Cela permet de maintenir un code propre et lisible, en évitant les discussions sur le style de formatage au sein de l'équipe de développement.

Install Prettier formatted code from vsCode it will do.

* + - 1. **Husky**

Husky est un outil qui permet d'ajouter des hooks de pré-commit et de pré-push à un projet Git. Il facilite l'exécution de scripts ou de commandes spécifiques avant chaque commit ou push, ce qui permet d'automatiser certaines tâches telles que l'exécution des tests unitaires, la vérification de la syntaxe du code ou le formatage du code. Husky s'intègre facilement aux projets existants et offre une configuration simple pour définir les hooks à exécuter.

* + - 1. **Lint-Staged**

Lint-staged est un outil qui permet d'exécuter des linters ou des validateurs de code uniquement sur les fichiers modifiés lors d'un commit. Il s'intègre généralement avec Husky et permet de définir des règles spécifiques pour chaque type de fichier. Lorsqu'un commit est effectué, lint-staged sélectionne les fichiers concernés et exécute les linters configurés pour vérifier leur conformité. Cela permet d'optimiser le processus de validation du code en ne vérifiant que les parties modifiées, ce qui peut être particulièrement utile dans les grands projets avec beaucoup de fichiers.