React : tester son application

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

React.js

Les tests unitaires avec **React.js** : les outils

- **Jest** : framework de test généraliste qui fournit l'infrastructure de test
- **React Testing Library** (**RTL**) : spécifiquement conçu pour tester l'interface utilisateur des composants **React** d'une manière centrée sur l'utilisateur.

React.js

Jest

- Exécution des tests : `it`, `test`, `describe`
- Assertions : `expect`
- Setup et Teardown : `beforeEach`, `afterEach`, `beforeAll` et `afterAll`

React.js

React Testing Library

- Rendu et Sélection des Éléments : `getByText`, `getByRole`, `getByLabelTex`...
- Interactions Utilisateur : `fireEvent` et `userEvent`
- Nettoyage Automatique : `cleanup`

React

Pour créer le squelette d'une application React

```
npx create-react-app react-test
```

React

Pour créer le squelette d'une application React

```
npx create-react-app react-test
```

Pour lancer le projet

```
npm start
```

React.js

Contenu initial de `App.js`

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

React.js

Remarque

Si vous avez créé le projet avec **CRA**, **Jest** et **RTL** ont également été installés. Donc, aucune installation n'est nécessaire.

React.js

Principales fonctions de **Jest**

- `Describe` : utilisé pour regrouper et décrire des tests similaires
- `it` : représente un cas de test individuel.
- `beforeEach` : exécution de code avant chaque test
- `afterEach` : exécution de code après chaque test
- `beforeAll` : exécution de code avant tous les tests
- `afterAll` : exécution de code après tous les tests

React.js

Commençons par préparer les deux cas de tests suivants (vides) dans `App.test.js`

```
describe('Test d\'App.js', () => {

  it('description du premier test', () => {

  });

  it('description du deuxième test', () => {

  });

});
```

React

Lançons les deux tests vides avec la commande suivante

```
npm test
```

React

Lançons les deux tests vides avec la commande suivante

```
npm test
```


Résultat

```
PASS src/App.test.js
 Test d'App.js
   √ description du premier test (1 ms)
   √ description du deuxième test

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
```

React.js

Pour lancer un seul test dans un `describe`, on utilise `only`

```
describe('Test d\'App.js', () => {

  it.only('description du premier test', () => {

  });

  it('description du deuxième test', () => {

  });

});
```

React

Pour lancer les tests d'un fichier en particulier

```
npm test chemin/vers/composant.test.js
```

React.js

Ajoutons les fonctions `beforeAll`, `beforeEach`, `afterEach` **et** `afterAll`

```
describe('Test d\'App.js', () => {
  beforeAll(() => {
    console.log('Avant tous les tests');
  });
  beforeEach(() => {
    console.log('Avant chaque test');
  });
  afterEach(() => {
    console.log('Après chaque test');
  });
  afterAll(() => {
    console.log('Arès tous les tests');
  });
  it('description du premier test', () => {

  });
  it('description du deuxième test', () => {

  });
});
```

React.js

Relancer les tests et vérifier que

- ils passent tous
- `beforeEach` a été exécuté deux fois avant chaque test
- `afterEach` a été exécuté deux fois après chaque test
- `beforeAll` a été exécuté avant tous les tests
- `afterAll` a été exécuté après tous les tests

React.js

Assertion

`expect` : permet de vérifier que les valeurs testées correspondent aux attentes

React.js

`Expect` comporte les matchers de comparaison suivants (Première partie)

- `toBe` : pour vérifier l'égalité stricte (`===` du JavaScript)
- `toEqual` : pour vérifier l'égalité non-stricte
- `toContain` : pour vérifier qu'un `Array` ou une `string` contient un élément
- `toBeDefined` : pour vérifier si un objet est défini
- `toBeNull` : pour vérifier si la valeur est nulle
- `toBeTruthy` : pour vérifier si la valeur est `true`
- `toBeFalsy` : pour vérifier si la valeur est `false`

React.js

`Expect` comporte les matchers de comparaison suivants (Deuxième partie)

- `toHaveBeenCalled` : pour vérifier si une méthode a été appelée
- `toHaveBeenCalledWith` : pour vérifier si une méthode a été appelée avec des paramètres d'une certaine valeur
- `toBeGreaterThan` : pour vérifier si une valeur est supérieure à une autre
- `toBeLessThan` : pour vérifier si une valeur est inférieure à une autre
- `toBeCloseTo` : pour vérifier si deux nombres sont proches (utile pour les comparaisons de nombres flottants)
- `toHaveLength` : pour vérifier la longueur d'un `Array` ou d'une `string`
- `toMatch` : pour vérifier si une `string` correspond à une expression régulière
- `toThrow` : pour vérifier si une fonction lance une erreur
- `toHaveProperty` : pour vérifier si un objet a une propriété spécifique avec une certaine valeur

React.js

Utilisons les assertions et les matchers pour tester la logique du composant `App` sans le Rendu

```
import App from './App';

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

});
```

React.js

Utilisons les assertions et les matchers pour tester la logique du composant `App` sans le Rendu

```
import App from './App';

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

});
```


Explication

- On Vérifie que le composant `App` est bien une fonction.
- on utilise le matcher `toBe` avec l'assertion `expect`.

React

Testez et vérifiez le résultat suivant

```
PASS src/App.test.js
 Test d'App.js
   √ App est une fonction (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

React.js

Remarque

Pour tester le rendu d'un composant, il faut utiliser **RTL.**

React.js

Utilisons `render()` de `RTL` pour monter le composant dans une ”fausse” version du DOM pour permettre les tests

```
import App from './App';
import { render } from '@testing-library/react'

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
  });

});
```

React.js

Utilisons `render()` de `RTL` pour monter le composant dans une ”fausse” version du DOM pour permettre les tests

```
import App from './App';
import { render } from '@testing-library/react'

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
  });

});
```

Un warning peut s’afficher en raison de l’utilisation implicite de `act` dans `render` de **React Testing Library**.

React.js

Importons ensuite `screen` **et affichons toutes les** `query functions` **fournies par** `React Testing Library`

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    console.log(screen);
  });

});
```

React.js

3 types de **Query function**

- `getBy` et `getAllBy` (Requête immédiate)
  - `getBy` : retourne un seul élément. Lance une erreur si l'élément n'est pas trouvé.
  - `getAllBy` : retourne tous les éléments qui correspondent. Lance une erreur si aucun élément n'est trouvé.
- `queryBy` et `queryAllBy` (Requête sans erreur)
  - `queryBy` : retourne un seul élément ou `null` s'il n'est pas trouvé. Ne lance pas d'erreur.
  - `queryAllBy` : retourne tous les éléments ou un tableau vide s'ils ne sont pas trouvés. Ne lance pas d'erreur.
- `findBy` et `findAllBy` (Requête asynchrone)
  - `findBy` : retourne une promesse qui résout un seul élément (utile pour des rendus asynchrones). Lance une erreur si l'élément n'est pas trouvé après un timeout.
  - `findAllBy` : retourne une promesse qui résout tous les éléments correspondant au sélecteur (utile pour des rendus asynchrones multiples).

React.js

Principales méthodes `getBy` et `getAllBy`

- `getByText` et `getAllByText` : recherche des éléments par leur texte visible.
- `getByLabelText` et `getAllByLabelText` : recherche des éléments associés à un `label`.
- `getByPlaceholderText` et `getAllByPlaceholderText` : recherche des éléments par leur texte d'espace réservé (placeholder).
- `getByRole` et `getAllByRole` : recherche des éléments par leur rôle **ARIA** (par exemple : `button`, `heading`, `textbox`).
- `getByAltText` et `getAllByAltText` : recherche des éléments par leur texte alternatif (utile pour les images avec l'attribut `alt`).
- `getByTitle` et `getAllByTitle` : recherche des éléments par leur attribut `title`.
- ...

React.js

Quand utiliser chaque méthode ?

- Utilisez `getBy` et `getAllBy` lorsque vous vous attendez à ce que l'élément soit présent immédiatement.
- Utilisez `queryBy` et `queryAllBy` lorsque vous souhaitez vérifier la présence ou l'absence d'un élément dans le DOM sans faire échouer le test si l'élément est introuvable.
- Utilisez `findBy` et `findAllBy` lorsque l'élément pourrait apparaître après un rendu asynchrone, comme une requête **HTTP**.

React.js

Exemple avec `getByText()`

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const linkElement = screen.getByText(/learn react/i);
    expect(linkElement).toBeInTheDocument();
    expect(linkElement).toHaveAttribute('href', 'https://reactjs.org');
    expect(linkElement).toHaveTextContent('Learn React')
  });

});
```

React.js

`Expect` avec **Jest-DOM** comporte les matchers suivants (Première partie)

- `toBeInTheDocument` : pour vérifier si un élément est présent dans le **DOM**
- `toHaveTextContent` : pour vérifier si un élément contient un texte spécifique
- `toHaveAttribute` : pour vérifier si un élément possède un attribut avec une certaine valeur
- `toHaveClass` : pour vérifier si un élément possède une ou plusieurs classes **CSS** spécifiques
- `toHaveStyle` : pour vérifier si un élément possède un style **CSS** spécifique
- `toBeVisible` : pour vérifier si un élément est visible (par exemple, pas caché par **CSS**)
- `toBeEmptyDOMElement` : pour vérifier si un élément ne contient aucun enfant
- `toBeDisabled` : pour vérifier si un élément est désactivé (comme un bouton)

React.js

`Expect` avec **Jest-DOM** comporte les matchers suivants (Deuxième partie)

- `toBeEnabled` : pour vérifier si un élément est activé
- `toBeChecked` : pour vérifier si une case à cocher ou un bouton radio est coché
- `toHaveFocus` : pour vérifier si un élément a le focus
- `toContainElement` : pour vérifier si un élément contient un autre élément
- `toContainHTML` : pour vérifier si un élément contient un certain **HTML**
- `toHaveValue` : pour vérifier la valeur d'un élément de formulaire (comme un champ de saisie)
- `toHaveAccessibleDescription` : pour vérifier si un élément a une description accessible
- `toHaveAccessibleName` : pour vérifier si un élément a un nom accessible

React.js

Exemple avec `getByAltText()`

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const logoElement = screen.getByAltText(/logo/i);
    expect(logoElement).toBeInTheDocument();
  });

});
```

React.js

Exemple avec `getByRole()` (`banner` : le rôle de la balise `header`)

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const roleElement = screen.getByRole(/banner/i);
    expect(roleElement).toBeInTheDocument();
  });

});
```

React.js

Exemple avec une méthode `getAllByText()`

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const links = screen.getAllByText(/learn react/i);
    for (const link of links) {
      expect(link).toBeInTheDocument();
    }
  });

});
```

React.js

On peut utiliser la fonction `screen.debug()` pour visualiser l'état actuel du DOM

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    screen.debug();
    const links = screen.getAllByText(/learn react/i);
    for (const link of links) {
      expect(link).toBeInTheDocument();
    }
  });

});
```

React.js

Ou pour visualiser une partie du DOM (plusieurs éléments)

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const links = screen.getAllByText(/learn react/i);
    screen.debug(links);
    for (const link of links) {
      expect(link).toBeInTheDocument();
    }
  });

});
```

React.js

Ou pour visualiser une partie du DOM (un seul élément)

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

describe('Test d\'App.js', () => {

  test('App est une fonction', () => {
    expect(typeof App).toBe('function');
  });

  test('Tester le rendu', () => {
    render(<App />);
    const links = screen.getAllByText(/learn react/i);
    for (const link of links) {
      screen.debug(link);
      expect(link).toBeInTheDocument();
    }
  });

});
```

React

Créons le composant `Compteur` **dans** `src/components`

```
import { useState } from 'react';

export default function Compteur() {
    let [counter, setCounter] = useState(0);

    const incrementer = () => {
        setCounter(counter + 1)
    }
    const decrementer = () => {
        setCounter(counter - 1)
    }
    return (
        <div>
            <h1>Compteur </h1>
            <button onClick={decrementer}>-</button>
            {counter}
            <button onClick={incrementer}>+</button>
        </div>
    )
}
```

React

Créons également le fichier de test `Compteur.test.js` avec le contenu initial suivant

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

describe('Test de Compteur.js', () => {

});
```

React

Ajoutons un premier test pour vérifier la valeur initiale du compteur

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

describe('Test de Compteur.js', () => {

    test('affiche le compteur initial à 0', () => {
        render(<Compteur />);
        expect(screen.getByText(/0/)).toBeInTheDocument();
    });

});
```

React

Utilisons `fireEvent` pour simuler des clics sur les deux boutons

```
import { render, screen, fireEvent } from '@testing-library/react';
import Compteur from './Compteur';

describe('Compteur Component', () => {
    test('affiche le compteur initial à 0', () => {
        render(<Compteur />);
        expect(screen.getByText(/0/)).toBeInTheDocument();
    });

    test('incrémente le compteur quand le bouton + est cliqué', () => {
        render(<Compteur />);

        const incrementButton = screen.getByText('+');
        fireEvent.click(incrementButton);

        expect(screen.getByText(/1/)).toBeInTheDocument();
    });

    test('décrémente le compteur quand le bouton - est cliqué', () => {
        render(<Compteur />);

        const decrementButton = screen.getByText('-');
        fireEvent.click(decrementButton);

        expect(screen.getByText(/-1/)).toBeInTheDocument();
    });
});
```

React.js

Question

Et si on avait besoin de simuler un évènement plus complexe (comme la saisie d'une valeur et la valeur saisie) ?

React.js

Question

Et si on avait besoin de simuler un évènement plus complexe (comme la saisie d'une valeur et la valeur saisie) ?

Réponse

Il faut utiliser `userEvent`.

React

ajoutons la modification du `pas` au composant `Compteur`

```
import { useRef, useState } from 'react';

export default function Compteur() {
    let [counter, setCounter] = useState(0);
    const pas = useRef(1);
    const incrementer = () => {
        const nouvelleValeur = counter + Number(pas.current.value);
        setCounter(nouvelleValeur);
    };

    const decrementer = () => {
        const nouvelleValeur = counter - Number(pas.current.value);
        setCounter(nouvelleValeur);
    };
    return (
        <div>
            <h1>Compteur </h1>
            <div>
                <input type='number' ref={pas} defaultValue={pas.current}/>
            </div>
            <button onClick={decrementer}>-</button>
            {counter}
            <button onClick={incrementer}>+</button>
        </div>
    )
}
```

React

Utilisons `userEvent` pour simuler la saisie de la valeur `5` dans l'`input:number`

```
test('incrémente le compteur avec un pas personnalisé', async () => {
    render(<Compteur />);

    const inputPas = screen.getByRole('spinbutton');
    const incrementButton = screen.getByText('+');

    await userEvent.clear(inputPas);
    await userEvent.type(inputPas, '5');

    await userEvent.click(incrementButton);

    expect(screen.getByText(/5/)).toBeInTheDocument();
  });
```

React

Utilisons `userEvent` pour simuler la saisie de la valeur `5` dans l'`input:number`

```
test('incrémente le compteur avec un pas personnalisé', async () => {
    render(<Compteur />);

    const inputPas = screen.getByRole('spinbutton');
    const incrementButton = screen.getByText('+');

    await userEvent.clear(inputPas);
    await userEvent.type(inputPas, '5');

    await userEvent.click(incrementButton);

    expect(screen.getByText(/5/)).toBeInTheDocument();
  });
```

Et l'import

```
import userEvent from '@testing-library/user-event';
```

React.js

Explication

- `screen.getByRole('spinbutton')` : permet de trouver l'`input` de type `number`
- `userEvent.clear(inputPas)` : efface la valeur par défaut de l'`input`
- `userEvent.type(inputPas, '5')` : saisit la valeur `5` dans le champ
- `userEvent.click(incrementButton)` : simule un clic sur le bouton d'incrémentation

React.js

Créons un fichier `db.json`, à la racine du projet, qui servira de base de données

```
{
    "personnes": [
        {
            "age": 45,
            "nom": "Wick",
            "prenom": "John",
            "id": 1
        },
        {
            "age": 40,
            "nom": "Dalton",
            "prenom": "Jack",
            "id": 2
        }
    ]
}
```

React.js

Pour pouvoir envoyer des requêtes HTTP, il faut démarrer json-server sur le port `5555` en précisant le nom de la base de données (`db.json`)

```
npx json-server -p 5555 db.json
```

React.js

Pour pouvoir envoyer des requêtes HTTP, il faut démarrer json-server sur le port `5555` en précisant le nom de la base de données (`db.json`)

```
npx json-server -p 5555 db.json
```

Résultat (parmi toutes les lignes affichées)

```
Endpoints:
http://localhost:5555/personnes
```

React.js

Créons le composant `Personne.js` **dans** `src/components`

```
import axios from 'axios'
import { useEffect, useState } from "react";

export default function Personne() {
    const [personnes, setPersonnes] = useState([]);
    useEffect(() => {
        axios
            .get('http://localhost:5555/personnes')
            .then(response => setPersonnes(response.data))
    }, []);
    return (
        <div>
            <h1>Gestion de personnes</h1>
            <ul>
                {
                    personnes && personnes.map((elt) =>
                        <li key={elt.id}>
                            {elt.nom} {elt.prenom}
                        </li>
                    )
                }
            </ul>
        </div>
    );
}
```

React.js

Réponse

Lancez le projet et vérifiez que la liste des personnes définies dans `db.json` s'affiche correctement en consultant le composant `Personne`.

React

Créons également le fichier de test `Personne.test.js` avec le contenu initial suivant

```
import Personne from './Personne';
import { render, screen, waitfor } from '@testing-library/react
    '
import axios from 'axios';

describe('Personne component', () => {

});
```

React

Mockons ensuite l'API Axios

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {

});
```

React

Mockons ensuite l'API Axios

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {

});
```


En cas de problème avec les mocks, modifiez la commande de test dans la section `scripts` de `package.json`

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --transformIgnorePatterns \"node_modules/(?!@codemirror)/\"",
  "eject": "react-scripts eject"
},
```

React

Préparons le test et le mock des données de la réponse de l'API

```
import { render, screen, waitFor } from '@testing-library/react
    '
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
    });
});
```

React

Mockons également l'appel à la méthode `get` d'`axios`

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);
    });
});
```

React

Rendons le composant avant de tester son contenu

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);

        render(<Personne />);
    });
});
```

React

Commençons par vérifier que le titre est affiché

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);

        render(<Personne />);

        expect(screen.getByText('Gestion de personnes')).toBeInTheDocument();
    });
});
```

React

Vérifions aussi que notre ressource a été appelée au moins une fois

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);

        render(<Personne />);

        expect(screen.getByText('Gestion de personnes')).toBeInTheDocument();
        expect(axios.get).toHaveBeenCalledWith('http://localhost:5555/personnes');
    });
});
```

Utilisons `waitFor` pour attendre que les données soient affichées après l'appel API

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);

        render(<Personne />);

        expect(screen.getByText('Gestion de personnes')).toBeInTheDocument();
        expect(axios.get).toHaveBeenCalledWith('http://localhost:5555/personnes');

        await waitFor(() => {
            expect(screen.getByText('Doe John')).toBeInTheDocument();
        });
    });
});
```

Utilisons `waitFor` pour attendre que les données soient affichées après l'appel API

```
import { render, screen, waitFor } from '@testing-library/react'
import Personne from './Personne';
import axios from 'axios';

jest.mock('axios');

describe('Personne component', () => {
    it('doit retourner la liste des personnes', async () => {
        const mockData = {
            data: [
                { id: 1, nom: 'Doe', prenom: 'John' },
                { id: 2, nom: 'Smith', prenom: 'Jane' },
            ],
        };
        axios.get.mockResolvedValueOnce(mockData);

        render(<Personne />);

        expect(screen.getByText('Gestion de personnes')).toBeInTheDocument();
        expect(axios.get).toHaveBeenCalledWith('http://localhost:5555/personnes');

        await waitFor(() => {
            expect(screen.getByText('Doe John')).toBeInTheDocument();
        });
    });
});
```

Sans `waitFor`, le test échoue car les éléments n'ont pas été mis à jour dans le **DOM**.