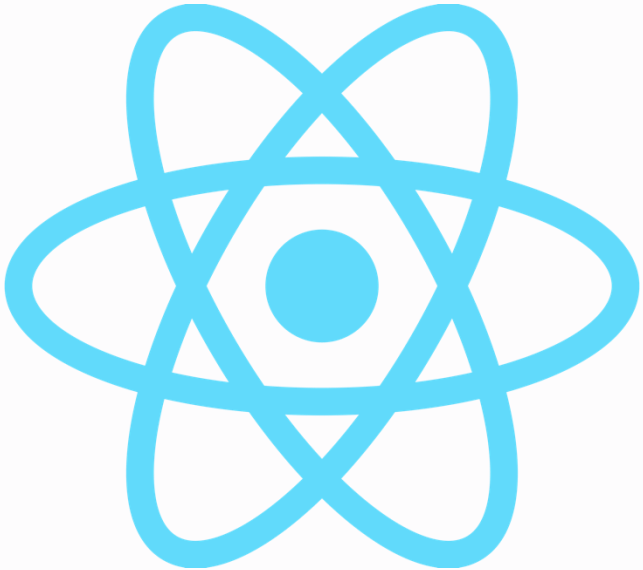
React JS



Support réalisé par Pierrick Hauguel

01

Présentation de la formation

First Day

- **Introduction and Review of Best Practices in React**
- • Source Organization
- • Use of hooks
- • Optimizations: memoization, virtual DOM
- • ErrorBoundary
- • ESLint rules
- • Strict mode

- **Advanced Patterns in React**
- • Combining hooks
- • Using useEffect and useContext to trigger actions
- • Pattern of functions as children

- **Introduction to TanStack Query**
- • Fundamentals of TanStack Query
- • Management of asynchronous data states
- • Caching, refetching, and optimizations
- • Integration with React

Second Day

- **Introduction to Zustand**
- • Basic principles of Zustand
- • Creation and management of simplified global states
- • Using Zustand in React components
- • Optimizations and best practices

- **Improving Application Performance**
- • Using React Dev Tools
- • Concurrent mode and Server Side Rendering (introduction)
- • Code splitting

Third Day

- **Advanced Testing in React**
- • Testing hooks
- • Testing components using hooks
- • Asynchronous tests
- • Advanced mocks

- **New Features in React 19**
- • New hooks
- • Changes and breaking changes
- • Review of new features

More topics ?

- Complexs forms ?
- Good practices with types ?
- How do you manage complex local data ? useState ? Something else ?
- How to override the TS namespace of React ? Which usecase ?

02

Comment tester ?

Comprendre l'objectif du test

- **Ce que vous devez tester :**
  - **Identifiez précisément l'élément à tester :** Est-ce une fonction utilitaire, un composant React, une interaction utilisateur spécifique ?
    - *Exemple :* Tester si le bouton "Envoyer" déclenche correctement la soumission du formulaire.

  - **Déterminez l'importance du test :**
    - Priorisez les tests des fonctionnalités critiques pour l'utilisateur.
    - *Exemple :* Si l'authentification est essentielle, concentrez-vous sur les tests de connexion et de déconnexion.

Identifier les types de tests appropriés

- **Choisissez le bon type de test :**
- **Tests unitaires :**
  - Pour tester des fonctions ou composants isolés.
  - *Exemple :* Vérifier que la fonction de calcul de taxes retourne le bon montant.
- **Tests d'intégration :**
  - Pour tester l'interaction entre plusieurs composants ou modules.
  - *Exemple :* Vérifier que le composant de panier met à jour le total lorsque des articles sont ajoutés.
- **Tests end-to-end (E2E) :**
  - Pour simuler le comportement réel de l'utilisateur.
  - *Exemple :* Tester le processus complet d'achat depuis la sélection du produit jusqu'au paiement.

Définir les cas de test

Scénarios d'utilisation typiques :

- **Lister les usages courants du composant ou de la fonction.**
  - *Exemple :* Pour un formulaire de contact, tester la soumission avec des données valides et invalides.
- **Entrées possibles :**
- **Considérer les cas limites et les valeurs invalides.**
  - *Exemple :* Que se passe-t-il si l'utilisateur laisse un champ obligatoire vide ou saisit un format de date incorrect ?
- **Résultats attendus :**
- **Définir clairement ce qui devrait se produire pour chaque scénario.**
  - *Exemple :* Un message d'erreur doit s'afficher si le champ email n'est pas valide.

Écrire des tests centrés sur le comportement

Bonnes pratiques :

- **Tester les effets visibles :**
  - *Solution :* Vérifiez que l'interface réagit comme prévu. Par exemple, après un clic sur "Ajouter au panier", le nombre d'articles doit augmenter.

- **Éviter les tests trop spécifiques à l'implémentation interne :**
  - *Solution :* Ne testez pas les états internes du composant qui pourraient changer lors d'une refactorisation.

Intégrer les tests dans le flux de développement

- **Questions à se poser :**
  - **À quel moment devrais-je écrire les tests ?**
    - *Solution :* Adoptez la méthode qui vous convient. Le TDD (Test Driven Development) vous oblige à écrire les tests avant le code, ce qui peut améliorer la qualité.

  - **Comment les tests s'intègrent-ils dans le processus d'intégration continue ?**
    - *Solution :* Configurez votre pipeline CI/CD pour exécuter les tests automatiquement lors des commits ou des pull requests.

Adopter une mentalité axée sur le test

Conseils pour développer cette mentalité :

- **Penser comme un utilisateur :**
  - *Solution :* Essayez de prévoir les actions et les erreurs que l'utilisateur pourrait commettre.
    - *Exemple :* Tester la navigation avec des données de session invalides.

- **Considérer les erreurs potentielles :**
  - *Solution :* Identifiez les points faibles de votre application.
    - *Exemple :* Que se passe-t-il si le serveur est indisponible ?

- **Être systématique :**
  - *Solution :* Utilisez des check-lists ou des matrices de test pour couvrir différents scénarios sans oublier de cas importants.

Revoir et améliorer continuellement les tests

Questions à se poser :

- **Mes tests sont-ils fiables et robustes ?**
  - *Solution :* Évitez les tests instables en gérant correctement les asynchronismes et en nettoyant le DOM après chaque test.

- **Sont-ils faciles à comprendre pour les autres membres de l'équipe ?**
  - *Solution :* Utilisez un langage clair et des structures de code simples.

- **Couvrent-ils suffisamment les fonctionnalités critiques de l'application ?**
  - *Solution :* Utilisez des outils de couverture de code pour identifier les zones non testées.

03

Mise en place de tests dans React

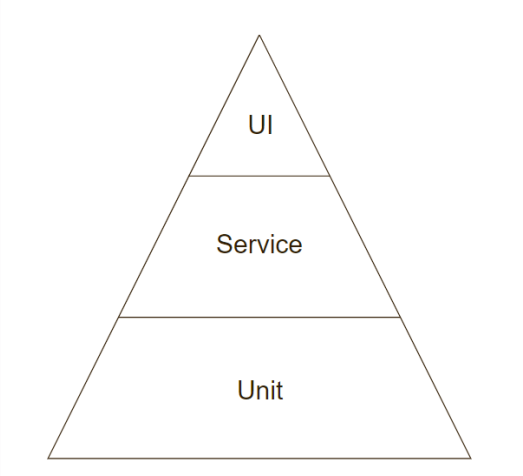
Introduction au test moderne de React

Les tests automatisés permettent de s'assurer que les fonctionnalités précédemment fonctionnelles continuent de l'être sans intervention manuelle, augmentant ainsi la confiance dans les modifications du code.

Les tests automatisés apportent confiance lors des refactorisations, servent de documentation toujours à jour et préviennent les bugs et régressions.

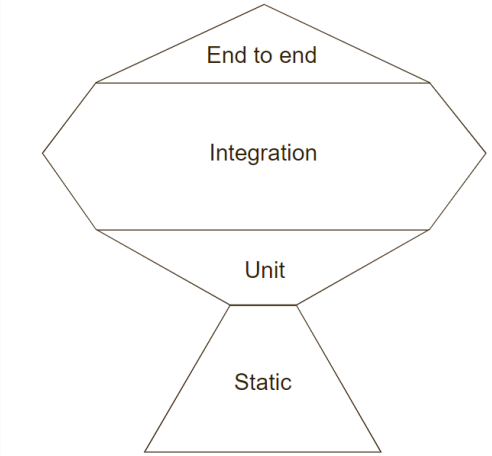
Présentation de la pyramide de tests

Explication de la pyramide des tests qui conseille une grande quantité de tests unitaires par rapport aux tests d'UI qui sont plus coûteux.



Approches alternatives de test : Le testing trophy de Kent C. Dodds

“It says that integration tests give you the biggest return on investment, so you should write more integration tests than any other kinds of tests.”



Quels outils ?

L'analyse statique permet de détecter les erreurs de syntaxe, les mauvaises pratiques et l'utilisation incorrecte des API : Formateurs de code, comme Prettier ; Les linters, comme ESLint ; Les vérificateurs de type, comme TypeScript et Flow.

Les tests unitaires vérifient que les algorithmes délicats fonctionnent correctement. Outils : Jest.

Quels outils ?

Les tests d'intégration vous donnent l'assurance que toutes les fonctionnalités de votre application fonctionnent comme prévu. Outils : Jest et Enzyme : Jest et Enzyme ou react-t**esting-library**.

Les tests de bout en bout permettent de s'assurer que votre application fonctionne comme un tout : le frontend, le backend, la base de données et tout le reste. Outils : Jest et Enzyme ou react-testing-librbrary : Cypress.

Introduction aux Tests Unitaires et d'Intégration

Tests unitaires: Validation d'une unité de code (fonction ou composant) isolément.

Tests d'intégration: Validation de l'interaction entre plusieurs modules/fonctions.

```
// Exemple d'un test unitaire
function add(a, b) {
  return a + b;
}

test('add function returns sum', () => {
  expect(add(1, 2)).toBe(3);
});

// Exemple d'un test d'intégration simulant une interaction entre API et UI
import { render, screen } from '@testing-library/react';
import App from './App';

test('displays fetched data', async () => {
  render(<App />);
  expect(await screen.findByText('Data loaded')).toBeInTheDocument();
});
```

Tests d'UI vs Tests unitaires : comparaison

Les tests d'UI sont coûteux et lents, tandis que les tests unitaires sont rapides et peu coûteux, idéaux pour tester des fonctions ou composants isolés.

```
// Exemple de test unitaire avec Jest
test('add function', () => {
  expect(add(1, 2)).toBe(3);
});
```

Tests unitaires avec Jest / vite

Jest est un outil permettant de réaliser des tests unitaires pour des algorithmes complexes ou des composants React.

```
// Test unitaire avec Jest
test('checks text content', () => {
  const { getByText } = render(<Button text="Click me!" />);
  expect(getByText(/click me/i)).toBeInTheDocument();
});
```

Tests d'intégration

Les tests d'intégration couvrent des fonctionnalités entières ou des pages, offrant un meilleur retour sur investissement que les tests unitaires.

```
// Test avec React Testing Library
test('loads items eventually', async () => {
  const { getByText } = render(<ItemsList />);
  const item = await waitForElement(() => getByText('Item 1'));
  expect(item).toBeInTheDocument();
});
```

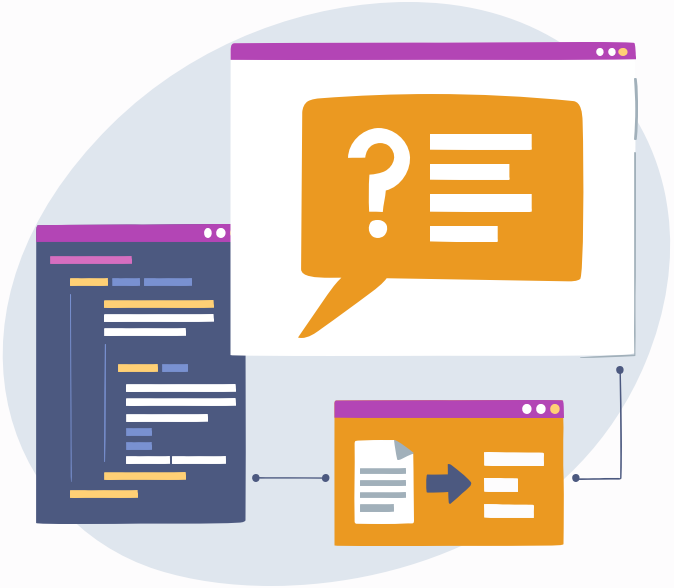
Comprendre les tests de bout en bout avec Cypress (E2E)

Cypress permet de réaliser des tests de bout en bout en simulant l'utilisation réelle de l'application dans un navigateur.

```
it('logs in successfully', () => {
  cy.visit('/login');
  cy.get('input[name=username]').type('user');
  cy.get('input[name=password]').type('password');
  cy.get('form').submit();
  cy.contains('Welcome, user!');
});
```

04

Vitest



Présentation de Vitest

- **Description:** Vitest est un framework moderne pour le test JavaScript, inspiré de Jest, rapide et conçu pour être intégré avec Vite.
- **Points forts:** Support des ESM, test en parallèle, rapidité.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './src/setupTests.js',
  },
})
```

```
npm install vite vitest @vitejs/plugin-react --save-dev
```

```
import '@testing-library/jest-dom';
```

```
{
  "scripts": {
    "test": "vitest"
  }
}
```

Premier test : qu'est-ce qu'un test ?

Un test vérifie le comportement attendu d'une fonction ou d'un bout de code.

```
import { expect, test } from 'vitest';

test('addition works', () => {
  expect(1 + 1).toBe(2); // Verifies that the result of 1 + 1 is indeed 2
});
```

Qu'est-ce qu'une assertion ?

Une assertion vérifie qu'une condition est vraie. Vitest utilise expect() pour définir les assertions.

```
expect(1 + 1).toBe(2); // This assertion checks if 1 + 1 equals 2
```

Liste des assertions dans Vitest

Liste des principales assertions utilisées dans Vitest.

```
expect(value).toBe(expected); // Asserts that value === expected
expect(value).toEqual(expected); // Asserts deep equality (for objects or arrays)
expect(value).toBeTruthy(); // Asserts that the value is truthy
expect(value).toBeFalsy(); // Asserts that the value is falsy
expect(value).toContain(item); // Asserts that the array or string contains the item
expect(value).toBeGreaterThan(number); // Asserts that the value is greater than a nu
expect(value).toThrow(); // Asserts that a function throws an error
```

Détail sur toBe vs toEqual

Différence entre toBe et toEqual. toBe vérifie l'égalité stricte (===), tandis que toEqual compare la structure des objets et tableaux.

```
expect({ a: 1 }).toEqual({ a: 1 }); // Passes, because the objects are deeply equal
expect({ a: 1 }).toBe({ a: 1 });    // Fails, because they are different object refere
```

Test d'une fonction utilitaire : multiply

Exemple de test sur une fonction utilitaire.

```
export function multiply(a, b) {
  return a * b;
}
```

```
import { multiply } from '../services/multiply';
import { expect, test } from 'vitest';

test('multiply function works correctly', () => {
  expect(multiply(2, 3)).toBe(6); // Multiplication of 2 and 3 should return 6
  expect(multiply(2, 0)).toBe(0); // Multiplying by zero should return 0
});
```

Groupement de tests avec describe

Utilisation de describe pour organiser des tests par thème ou par fonction.

```
import { describe, expect, test } from 'vitest';

describe('multiplication tests', () => {
  test('multiply positive numbers', () => {
    expect(multiply(2, 3)).toBe(6); // 2 * 3 equals 6
  });

  test('multiply by zero', () => {
    expect(multiply(2, 0)).toBe(0); // Multiplying any number by 0 returns 0
  });

  test('multiply by negative number', () => {
    expect(multiply(2, -2)).toBe(-4); // 2 * -2 equals -4
  });
});
```

Tester des fonctions asynchrones

Comment tester des fonctions asynchrones dans Vitest.

```
export async function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve('data'), 1000);
  });
}
```

```
import { fetchData } from '../services/dataService';
import { expect, test } from 'vitest';

test('fetchData returns correct data', async () => {
  const data = await fetchData();
  expect(data).toBe('data'); // Asserts that fetchData resolves to 'data'
});
```

Utilisation de beforeEach et afterEach

beforeEach permet d'exécuter du code avant chaque test, et afterEach après chaque test. Très utile pour initialiser ou nettoyer des variables.

```
let counter;

beforeEach(() => {
  counter = 0; // Initialize counter before each test
});

test('increments counter', () => {
  counter++;
  expect(counter).toBe(1); // The counter should be 1 after incrementing
});

afterEach(() => {
  // Optionally clean up after each test
});
```

Mocking des services HTTP avec Vitest

Utilisation de vi.mock() pour simuler des appels HTTP dans les tests, sans avoir à réellement appeler des services externes.

```
import axios from 'axios';
import { vi } from 'vitest';

vi.mock('axios'); // Mock axios to prevent real HTTP requests
const mockedAxios = vi.mocked(axios);

mockedAxios.get.mockResolvedValue({ data: { userId: 1 } }); // Mock the get request

test('fetches user data', async () => {
  const response = await axios.get('/user/1');
  expect(response.data.userId).toBe(1); // Expect the mocked data to be returned
});
```

Utilisation des stubs pour simuler des comportements

Les stubs permettent de remplacer temporairement une fonction pour tester différentes situations.

```
import { vi } from 'vitest';

const logStub = vi.fn(); // Stub a function

function greet() {
  logStub('Hello'); // Use the stub instead of the real implementation
}

test('log is called with correct argument', () => {
  greet();
  expect(logStub).toHaveBeenCalledWith('Hello'); // Verify that the stub was called
});
```

Mocking de fonctions asynchrones

Comment créer des mocks de fonctions asynchrones avec Vitest.

```
import { vi } from 'vitest';

const asyncFunction = vi.fn().mockResolvedValue('mocked data'); // Mock an async funct

test('async function returns mocked data', async () => {
  const result = await asyncFunction();
  expect(result).toBe('mocked data'); // Expect the mocked result
});
```

Tester des erreurs avec toThrow()

Utilisation de l'assertion toThrow() pour tester si une fonction lance une erreur

```
function throwError() {
  throw new Error('This is an error!');
}

test('function throws an error', () => {
  expect(() => throwError()).toThrow('This is an error!'); // Expect an error to be th
});
```

Gestion des tests avec des valeurs paramétrées

Utilisation de test.each() pour exécuter le même test avec différentes valeurs.

```
test.each([[1, 1, 2], [2, 2, 4], [3, 3, 6]])(
  'adds %i and %i to equal %i',
  (a, b, expected) => {
    expect(a + b).toBe(expected); // Test addition with different values
  }
);
```

Utilisation de test.only pour exécuter un test unique

test.only permet de n'exécuter qu'un seul test, utile pour le débogage.

```
test.only('runs this test only', () => {
  expect(1 + 1).toBe(2); // Only this test will run
});
```

Combiner les assertions pour tester plusieurs aspects

Exemple combinant plusieurs assertions pour vérifier différentes parties du code.

```
test('checks multiple aspects', () => {
  const user = { name: 'Bob', age: 30 };
  expect(user.name).toBe('Bob'); // Verify the name
  expect(user.age).toBeGreaterThan(20); // Verify the age is greater than 20
});
```

Mocking de modules entiers

Utilisation de vi.mock() pour simuler des modules entiers, pas seulement des fonctions spécifiques.

```
vi.mock('../utils/math', () => ({
  multiply: vi.fn(() => 10) // Mock the entire module with custom implementations
}));

import { multiply } from '../utils/math';

test('multiply function is mocked', () => {
  expect(multiply(2, 3)).toBe(10); // The mocked function returns 10 regardless of i
});
```

Écrire des tests pour des fonctions purement utilitaires

Les fonctions utilitaires peuvent être testées isolément car elles n'ont pas de dépendances extérieures.

```
function add(a, b) {
  return a + b;
}

test('add function works', () => {
  expect(add(2, 3)).toBe(5); // Test the utility function directly
});
```

Utilisation de spyOn pour observer les appels de fonction

vi.spyOn() permet de surveiller les appels à une fonction sans la remplacer.

```
const obj = {
  log: (message) => console.log(message),
};

test('spy on log function', () => {
  const spy = vi.spyOn(obj, 'log'); // Spy on the log function

  obj.log('Hello, world!');
  expect(spy).toHaveBeenCalledWith('Hello, world!'); // Check that log was called
});
```

Tester les effets secondaires

Tester des fonctions qui modifient un état externe ou un environnement, par exemple en modifiant des variables globales ou le stockage local.

```
test('modifies global variable', () => {
  global.counter = 0; // Set a global variable
  function incrementCounter() {
    global.counter++;
  }

  incrementCounter();
  expect(global.counter).toBe(1); // Check if the global variable was modified
});
```

Tests avec des timers simulés

Utilisation de vi.useFakeTimers() pour simuler des délais ou des timers dans les tests.

```
test('delays execution', () => {
  vi.useFakeTimers(); // Use fake timers for the test

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.runAllTimers(); // Fast-forward all timers

  expect(value).toBe(true); // The value should now be true after the timer has run
});
```

Contrôler le temps avec vi.advanceTimersByTime

Avancer manuellement le temps pour simuler des retards dans l'exécution du code.

```
test('advances timers by specific time', () => {
  vi.useFakeTimers();

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.advanceTimersByTime(500); // Move time forward by 500ms
  expect(value).toBe(false); // The timer hasn't completed yet

  vi.advanceTimersByTime(500); // Move time forward by another 500ms
  expect(value).toBe(true); // Now the timer should have completed
});
```

Mocking des modules externes

Utilisation de vi.mock() pour simuler des modules ou des bibliothèques externes dans les tests.

```
vi.mock('axios', () => ({
  get: vi.fn(() => Promise.resolve({ data: { id: 1 } }))
}));

import axios from 'axios';

test('fetches data with mocked axios', async () => {
  const response = await axios.get('/api/user');
  expect(response.data.id).toBe(1); // Test the mocked response
});
```

Tests de performance avec vi.measure

Utilisation de vi.measure() pour mesurer la performance d'une fonction ou d'un morceau de code.

```
test('measures performance of function', () => {
  const timeTaken = vi.measure(() => {
    let sum = 0;
    for (let i = 0; i < 1000000; i++) {
      sum += i;
    }
    return sum;
  });

  expect(timeTaken.duration).toBeLessThan(100); // Expect the function to execute
});
```

Mocking des dates avec vi.setSystemTime

Simuler des dates et des heures spécifiques pour tester des fonctions dépendantes du temps.

```
test('mocks system date', () => {
  const mockDate = new Date(2020, 1, 1);
  vi.setSystemTime(mockDate); // Set the system time to a specific date

  expect(new Date().getFullYear()).toBe(2020); // The current year should now be 2020
});
```

Mocking des événements avec des callbacks

Utilisation de mocks pour simuler des événements ou des callbacks dans les tests.

```
function onClick(callback) {
  callback('Button clicked');
}

test('calls the callback on click', () => {
  const mockCallback = vi.fn(); // Mock the callback function

  onClick(mockCallback);
  expect(mockCallback).toHaveBeenCalledWith('Button clicked'); // The callback should
});
```

Tests d'interaction entre plusieurs services

Tester l'intégration entre plusieurs services ou modules dans une application.

```
import { getUser, saveUser } from '../services/userService';

test('gets and saves user data', () => {
  const user = getUser(1);
  saveUser(user);

  expect(user.id).toBe(1); // Verify that the correct user was fetched and saved
});
```

Tests de services dépendants de l'état global

Comment tester des services qui dépendent de l'état global ou du contexte de l'application.

```
let globalState = {
  loggedIn: false,
};

function login() {
  globalState.loggedIn = true;
}

test('logs in user and updates global state', () => {
  login();
  expect(globalState.loggedIn).toBe(true); // Check if the global state was updated
});
```

Mocking d'API externes dans les services

Simuler les appels API externes dans les tests de services.

```
vi.mock('../api/externalApi', () => ({
  fetchData: vi.fn(() => Promise.resolve({ data: 'mocked data' }))
}));

import { fetchData } from '../api/externalApi';

test('fetches mocked data', async () => {
  const result = await fetchData();
  expect(result.data).toBe('mocked data'); // Test the mocked API response
});
```

Tester les retours d'erreur des services

Vérifier la gestion des erreurs dans les services en simulant des échecs.

```
function getUser(id) {
  if (id <= 0) {
    throw new Error('Invalid user ID');
  }
  return { id, name: 'John Doe' };
}

test('throws error for invalid user ID', () => {
  expect(() => getUser(-1)).toThrow('Invalid user ID'); // Expect an error to be throw
});
```

Utilisation de toMatchObject pour vérifier des objets partiels

toMatchObject() permet de vérifier partiellement un objet sans nécessiter une égalité complète.

```
const user = { id: 1, name: 'John', age: 30 };

test('matches partial object', () => {
  expect(user).toMatchObject({ id: 1, name: 'John' }); // Only checks for matching
});
```

Exécuter des Tests avec Vitest

Exécution des tests avec Vitest via la ligne de commande et en mode watch.

```
npm run test  # Exécute tous les tests
npm run test -- --watch  # Mode watch pour réexécuter automatiquement les tests
```

Exécution des Tests avec Couverture

Comment générer des rapports de couverture de tests avec Vitest pour visualiser les zones non testées du code.

```
vitest run --coverage
```

Optimisation des Tests avec Vitest

Optimisation des performances des tests en activant les tests en parallèle et le mode watch.

```
test: {
  environment: 'jsdom',
  maxThreads: 4,
  minThreads: 2,
},
```

05

React Testing Library

React Testing Library: Introduction

React Testing Library (RTL) est une bibliothèque axée sur les tests d'accessibilité et d'interaction utilisateur. Principe clé: Tester les composants comme un utilisateur interagit avec eux.

```
npm install @testing-library/react @testing-library/jest-dom
```

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

test('renders hello world', () => {
  render(<App />);
  const linkElement = screen.getByText(/hello world/i);
  expect(linkElement).toBeInTheDocument();
});
```

Installation et Configuration de Vitest et RTL

- Comment configurer un projet React pour utiliser Vitest et RTL ensemble.
- Installation des dépendances : Vitest, RTL, et JSDOM (environnement de test).

```
npm install vitest @testing-library/react jsdom --save-dev
```

```
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    globals: true,
    environment: 'jsdom',
  },
});
```

Render et screen : qu'est-ce que c'est ?

Explication des méthodes render et screen dans RTL.

```
render(<Component />); // Render the component into a virtual DOM
screen.getByText('text'); // Find an element by its text
```

Rendu des composants

Utilisation de la méthode render() de React Testing Library pour vérifier le rendu des composants.

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

test('trouve le lien learn react', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Les méthodes de recherche dans RTL : getBy, queryBy, findBy

Différences entre getBy, queryBy et findBy dans RTL.

```
// getBy throws an error if element is not found
const button = screen.getByRole('button');

// queryBy returns null if element is not found
const optionalButton = screen.queryByRole('button');

// findBy is asynchronous and waits for the element to appear
const asyncButton = await screen.findByRole('button');
```

Test des composants avec des props

Comment tester les composants qui prennent des props en entrée.

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

test('renders with props', () => {
  render(<MyComponent title="My Title" />);
  expect(screen.getByText('My Title')).toBeInTheDocument(); // Verify the prop
});
```

Test des composants enfants avec des mocks

Mocking des composants enfants pour isoler les tests des composants parents.

```
vi.mock('./ChildComponent', () => () => <div>Mocked Child</div>);

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

test('renders parent with mocked child', () => {
  render(<ParentComponent />);
  expect(screen.getByText('Mocked Child')).toBeInTheDocument(); // Ensure the mocked
});
```

Interactions utilisateur

Simuler les événements utilisateur avec fireEvent dans React Testing Library.

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

test('clique sur le bouton', () => {
  const handleClick = jest.fn();
  const { getByText } = render(<Button onClick={handleClick}>Cliquez</Button>);
  fireEvent.click(getByText(/Cliquez/i));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Utilisation de userEvent pour simuler des interactions

Utiliser userEvent pour simuler des interactions complexes.

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test('handles typing in input', async () => {
  render(<Input />);
  const input = screen.getByPlaceholderText('Type something');
  await userEvent.type(input, 'Hello, React!'); // Simulate typing in the input field
  expect(input.value).toBe('Hello, React!'); // Check if the input value is updated
});
```

Utilisation de fireEvent vs userEvent

Différences entre fireEvent et userEvent pour simuler les interactions.

```
// fireEvent triggers an immediate interaction
fireEvent.click(button);

// userEvent simulates a more realistic user interaction with delays
await userEvent.click(button);
```

Tests des entrées de formulaire

Tests des entrées de formulaire avec fireEvent.change() pour simuler la saisie de texte.

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

test('saisit le nom d\'utilisateur', () => {
  const { getByLabelText } = render(<Login />);
  const input = getByLabelText(/nom d'utilisateur/i);
  fireEvent.change(input, { target: { value: 'john_doe' } });
  expect(input.value).toBe('john_doe');
});
```

Simuler des événements de formulaire

Simulation des événements de formulaire tels que submit avec fireEvent.

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

test('soumet le formulaire', () => {
  const handleSubmit = jest.fn();
  const { getByTestId } = render(<Login onSubmit={handleSubmit} />);
  fireEvent.submit(getByTestId('login-form'));
  expect(handleSubmit).toHaveBeenCalled();
});
```

Tests des composants avec contexte

Tests des composants qui utilisent Context API en enveloppant les composants dans le fournisseur de contexte.

```
import { render } from '@testing-library/react';
import { UserProvider } from './UserContext';
import UserProfile from './UserProfile';

test('affiche le profil utilisateur', () => {
  const user = { name: 'John Doe' };
  const { getByText } = render(
    <UserProvider value={user}>
      <UserProfile />
    </UserProvider>
  );
  expect(getByText(/John Doe/i)).toBeInTheDocument();
});
```

Utilisation de act() pour des mises à jour synchrones

Utilisation de la méthode act() pour garantir que toutes les mises à jour d'état et de DOM sont appliquées avant les assertions.

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

test('met à jour le compteur', () => {
  const { getByText } = render(<Counter />);
  const button = getByText(/increment/i);

  act(() => {
    fireEvent.click(button);
  });

  expect(getByText(/count: 1/i)).toBeInTheDocument();
});
```

Tests de composants avec Redux

Tests des composants connectés à Redux en utilisant le provider de Redux.

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

test('rend le composant avec Redux', () => {
  const { getByText } = render(
    <Provider store={store}>
      <App />
    </Provider>
  );
  expect(getByText(/learn react/i)).toBeInTheDocument();
});
```

Tester une action Redux

Test d'une action simple avec Redux.

```
import counterReducer, { increment } from './counterSlice';

test('should handle increment', () => {
  const previousState = { value: 0 };
  expect(counterReducer(previousState, increment())).toEqual({ value: 1 });
});
```

Utilisation de Redux Toolkit

Introduction à Redux Toolkit pour simplifier les tests Redux.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
  },
});

export const { increment } = counterSlice.actions;
export default counterSlice.reducer;
```

```
import counterReducer, { increment } from './counterSlice';

test('should handle increment', () => {
  const previousState = { value: 0 };
  expect(counterReducer(previousState, increment())).toEqual({ value: 1 });
});
```

Tester le store configuré

Tester le comportement du store avec Redux Toolkit.

```
test('should handle store actions', () => {
  store.dispatch(increment());
  expect(store.getState().counter.value).toBe(1);
});
```

Mocking Redux State avec RTL

Comment mocker le state Redux dans les tests avec RTL.

```
import { render } from '@testing-library/react';
import { Provider } from 'react-redux';
import { store } from './store';

const renderWithRedux = (ui, { initialState, store = createStore(reducer, initialState) } = {}) => {
  return {
    ...render(<Provider store={store}>{ui}</Provider>),
    store,
  };
};
```

```
const mockStore = {
  counter: { value: 42 },
};

const { getByText } = renderWithRedux(<Counter />, { initialState: mockStore });
expect(getByText(/42/i)).toBeInTheDocument();
```

Tester les appels API avec Redux et MSW

Comment tester les effets de côté d'une action Redux avec MSW et RTL.

```
import { fetchCounter } from './counterSlice';

test('fetch counter from API', async () => {
  const result = await store.dispatch(fetchCounter());
  expect(result.payload.value).toBe(42);
});
```

Tester createAsyncThunk

Comment tester des thunks asynchrones avec Vitest.

```
import { createAsyncThunk } from '@reduxjs/toolkit';

export const fetchCounter = createAsyncThunk('counter/fetch', async () => {
  const response = await fetch('/api/counter');
  return response.json();
});
```

```
test('fetchCounter fulfills successfully', async () => {
  const result = await store.dispatch(fetchCounter());
  expect(result.type).toBe('counter/fetch/fulfilled');
  expect(result.payload.value).toBe(42);
});
```

Tester les erreurs dans les actions asynchrones

Tester les erreurs et les états d'erreur lors des appels asynchrones avec Redux.

```
test('fetchCounter rejected with error', async () => {
  server.use(
    rest.get('/api/counter', (req, res, ctx) => {
      return res(ctx.status(500));
    })
  );
  const result = await store.dispatch(fetchCounter());
  expect(result.type).toBe('counter/fetch/rejected');
});
```

Introduction à Suspense

Suspense permet de gérer le rendu asynchrone dans les composants React. Voyons comment le tester avec Vitest et RTL.

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

test('renders fallback during suspense', () => {
  const { getByText } = render(
    <Suspense fallback={<div>Loading...</div>}>
      <MyLazyComponent />
    </Suspense>
  );
  expect(getByText(/Loading/i)).toBeInTheDocument();
});
```

Exemple Complet de Test avec RTL

Tester le rendu d’un composant React et les interactions utilisateur.

```
// Button.js
const Button = ({ onClick, label }) => (
  <button onClick={onClick}>{label}</button>
);

// Button.test.js
import { render, fireEvent, screen } from '@testing-library/react';
import Button from './Button';

test('calls onClick when clicked', () => {
  const handleClick = vi.fn();
  render(<Button onClick={handleClick} label="Click Me" />);

  fireEvent.click(screen.getByText('Click Me'));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Simuler des Requêtes Asynchrones avec RTL

Gérer les fonctions asynchrones dans les tests avec async/await et waitFor.

```
test('fetches and displays data', async () => {
  global.fetch = vi.fn(() =>
    Promise.resolve({
      json: () => Promise.resolve({ message: 'Hello World' }),
    })
  );

  render(<MyComponent />);
  expect(screen.getByText(/loading/i)).toBeInTheDocument();

  await waitFor(() => expect(screen.getByText(/hello world/i)).toBeInTheDocument());
});
```

```
await waitFor(() => expect(screen.getByText('Success')).toBeInTheDocument(), {
  timeout: 2000, // ici on attend jusqu'à 2 secondes avant que le test échoue
});
```

Gestion des Erreurs dans les Tests

Simuler des erreurs dans les composants pour tester la gestion des exceptions et des erreurs asynchrones

```
const ErrorComponent = ({ fetchData }) => {
  const [error, setError] = React.useState(null);

  React.useEffect(() => {
    fetchData().catch((err) => setError(err.message));
  }, [fetchData]);

  if (error) return <div>Error: {error}</div>;
  return <div>Loading...</div>;
};

test('displays error message', async () => {
  const fetchData = vi.fn().mockRejectedValue(new Error('API Error'));

  render(<ErrorComponent fetchData={fetchData} />);
  await waitFor(() => expect(screen.getByText('Error: API Error')).toBeInTheDocument()
});
```

Test des Composants avec des Contextes

Tester des composants qui utilisent React.Context pour partager des données globales.

```
// MyContext.js
const MyContext = React.createContext();

// Component.js
const Component = () => {
  const value = React.useContext(MyContext);
  return <div>{value}</div>;
};

test('provides context value', () => {
  render(
    <MyContext.Provider value="Hello from context">
      <Component />
    </MyContext.Provider>
  );

  expect(screen.getByText('Hello from context')).toBeInTheDocument();
});
```

Test des Composants Asynchrones et des Hooks

Tester les composants qui utilisent des hooks React pour gérer des effets asynchrones.

```
const useFetchData = () => {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    fetch('/api/data')
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return data;
};

test('tests a hook with async data', async () => {
  global.fetch = vi.fn(() => Promise.resolve({ json: () => Promise.resolve({ message:

  const { result, waitForNextUpdate } = renderHook(() => useFetchData());

  await waitForNextUpdate();
  expect(result.current).toEqual({ message: 'Success' });
});
```

06

E2E avec Cypress

Introduction à Cypress

Cypress est un outil puissant pour les tests end-to-end (E2E) conçu pour les applications web modernes.

```
npm install cypress --save-dev
```

Lancer Cypress

Une fois installé, Cypress peut être lancé en mode interactif ou en mode headless.

```
npx cypress open   # Ouvre l'interface graphique de Cypress
npx cypress run    # Exécute les tests en mode headless
```

Configuration de Base de Cypress

Configuration de Cypress dans cypress.config.js pour adapter le comportement des tests (timeout, viewport, etc.).

```
// cypress.config.js
module.exports = {
  e2e: {
    baseUrl: 'http://localhost:3000',
    viewportWidth: 1280,
    viewportHeight: 720,
  },
};
```

Premier Test End-to-End (E2E)

Écrire un test simple avec Cypress qui vérifie le rendu de la page d'accueil d'une application.

```
describe('Homepage Test', () => {
  it('should load the homepage', () => {
    cy.visit('/');
    cy.contains('Welcome to My App').should('be.visible');
  });
});
```

Sélection des Éléments avec Cypress

Utiliser cy.get() pour sélectionner des éléments dans le DOM.Sélection par classe, ID, attributs, etc.

```
cy.get('.button-class').click();
cy.get('#element-id').should('have.text', 'Expected Text');
cy.get('[data-cy=submit-button]').click();
```

Assertions avec Cypress

Cypress utilise Chai pour les assertions. Tester les états des éléments, les URL, etc.

```
cy.url().should('include', '/dashboard');
cy.get('.alert').should('have.class', 'success');
cy.get('input').should('have.value', 'Cypress Test');
```

Interaction Utilisateur avec Cypress

Simuler des actions utilisateur comme click(), type(), clear().

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('button[type="submit"]').click();
```

Tests Asynchrones avec Cypress

Cypress gère automatiquement les actions asynchrones. Il attend que les éléments soient disponibles avant d'exécuter les actions suivantes.

```
cy.get('button').click();
cy.get('.loading-spinner').should('not.exist');
cy.get('.result').should('contain', 'Success');
```

Gérer les Requêtes HTTP avec cy.intercept()

Intercepter et mocker les requêtes HTTP pour simuler des réponses d'API.

```
cy.intercept('GET', '/api/data', { fixture: 'data.json' }).as('getData');
cy.visit('/');
cy.wait('@getData');
```

Utilisation des Fixtures

Les fixtures sont des fichiers JSON ou autres formats qui contiennent des données de test pour simuler des réponses de l'API.

```
cy.fixture('user.json').then((user) => {
  cy.get('input[name="username"]').type(user.username);
  cy.get('input[name="password"]').type(user.password);
});
```

Tests de Formulaires avec Cypress

Tester des formulaires en validant les inputs, soumissions et réponses.

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('input[name="password"]').type('password123');
cy.get('form').submit();
cy.get('.success-message').should('be.visible');
```

Assertions d'URL et de Navigation

Vérifier que l'URL et la navigation sont correctes après certaines actions.

```
cy.url().should('include', '/dashboard');
cy.get('a[href="/profile"]').click();
cy.url().should('include', '/profile');
```

Gestion des Cookies et Local Storage

Cypress permet d'interagir avec les cookies et le local storage pour gérer des sessions d'utilisateur.

```
cy.setCookie('session_id', '123ABC');
cy.getCookie('session_id').should('have.property', 'value', '123ABC');

cy.window().then((win) => {
  win.localStorage.setItem('token', 'authToken');
});
```

Tests avec Authentication

Gérer l'authentification avec Cypress, simuler des connexions utilisateurs et des sessions.

```
cy.request('POST', '/login', {
  username: 'user1',
  password: 'password',
}).then((response) => {
  cy.setCookie('authToken', response.body.token);
  cy.visit('/dashboard');
});
```

Intégration avec CI/CD

Intégration de Cypress dans un pipeline CI/CD comme GitHub Actions ou GitLab CI.

```yaml
name: Cypress Tests

on: [push]

jobs:
  cypress-run:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run Cypress tests
        run: npx cypress run
```

07

Mocks avec MSW

Installation de MSW

Installer MSW pour simuler les requêtes réseau dans les tests d'intégration.

```
npm install --save-dev msw
```

Initialisation du Service Worker

Générer le script du Service Worker avec la commande msw init.

```
import * as msw from 'msw';
import { setupWorker } from 'msw/browser';
import { handlers } from './handlers';

export const worker = setupWorker(...handlers);
window.msw = { worker, ...msw };
```

Démarrage du Service Worker

Démarrer le Service Worker en mode développement.

```
async function enableMocking() {
  if (process.env.NODE_ENV !== 'development') return;

  const { worker } = await import('./mocks/browser');
  return worker.start();
}

enableMocking().then(() => {
  const root = createRoot(document.getElementById('root'));
  root.render(<App />);
});
```

Handlers MSW

Créer des définitions de mocks pour les requêtes réseau.

```
import { http, HttpResponse } from 'msw';

export const handlers = [
  http.get('https://httpbin.org/anything', () => {
    return HttpResponse.json({
      args: { ingredients: ['bacon', 'tomato', 'mozzarella', 'pineapples'] }
    });
  })
];
```

08

Tests avec Storybook

Écrire votre première story

Créez votre première story pour un composant React afin de le visualiser dans Storybook.

```
import React from 'react';
import Button from './Button';

export default {
  title: 'Example/Button',
  component: Button,
};

export const Primary = () => <Button primary label="Button" />;
```

Addon Knobs pour manipuler les props

Utilisez l'addon Knobs pour interagir dynamiquement avec les propriétés de vos composants dans Storybook.

```
// Import the React library and the Button component
import React from 'react';
import Button from './Button';

// Define metadata for the story
export default {
  title: 'Example/Button', // Title in Storybook's sidebar
  component: Button,       // The component being documented
};

// Define a story named 'Primary'
export const Primary = () => <Button primary label="Button" />;
// Renders the Button component with 'primary' prop and 'Button' label
```

Addon Actions pour suivre les événements

Apprenez à utiliser l'addon Actions pour logger les actions et événements de vos composants.

```
// Import the necessary modules for knobs
import React from 'react';
import { withKnobs, text } from '@storybook/addon-knobs';
import Button from './Button';

// Add the knobs decorator to the story
export default {
  title: 'Example/Button',
  component: Button,
  decorators: [withKnobs], // Enables knobs in this story
};

// Create a story with a dynamic label
export const DynamicLabel = () => (
  <Button label={text('Label', 'Button')} />
);
// Allows you to change the 'label' prop via the knobs panel
```

Addon Controls pour manipuler les props

Utilisez l'addon Controls pour interagir avec les props de vos composants directement depuis l'interface de Storybook.

```
import React from 'react';
import { action } from '@storybook/addon-actions';
import Button from './Button';

export default {
  title: 'Example/Button',
  component: Button,
};

export const Clickable = () => (
  <Button onClick={action('button-click')} label="Click Me" />
);
```

Gestion des assets statiques

Apprenez à inclure et gérer des fichiers statiques tels que des images dans vos stories.

```
// Import an image file to use in your component
import logo from './assets/logo.png';

// Use the image in your component
const Logo = () => <img src={logo} alt="Logo" />;
// Displays the imported logo image
```

Utilisation de Storybook avec Redux

Intégrez Redux dans vos stories pour gérer l'état global de vos composants.

```
import React from 'react';
import { Provider } from 'react-redux';
import store from './store';
import TodoList from './TodoList';

export default {
  title: 'Example/TodoList',
  component: TodoList,
  decorators: [(Story) => <Provider store={store}><Story /></Provider>],
};

export const Default = () => <TodoList />;
```

09

E2E avec Playwright

Qu'est-ce que Playwright ?

Playwright est un outil de test de bout en bout qui permet de tester des applications web en simulant des interactions utilisateur réelles.

```
const { test, expect } = require('@playwright/test');

test('example test', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page).toHaveTitle('Example Domain');
});
```

Avantages de Playwright

Playwright offre une expérience de test et de débogage optimale, permet d'inspecter la page à tout moment et supporte plusieurs navigateurs.

```
const { chromium } = require('playwright');

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await browser.close();
})();
```

Playwright vs Cypress

Playwright offre une API plus cohérente, une configuration plus simple, supporte les multi-onglets et est plus rapide que Cypress.

```
const { test, expect } = require('@playwright/test');

test('compare with cypress', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page.locator('h1')).toContainText('Example Domain');
});
```

Fichier de configuration Playwright

Définir les paramètres globaux et les projets pour chaque navigateur.

```
const { defineConfig, devices } = require('@playwright/test');

module.exports = defineConfig({
  testDir: './tests',
  fullyParallel: true,
  reporter: 'html',
  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry'
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } }
  ],
  webServer: {
    command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI
```

Test de base avec Playwright

Créer un test de base pour vérifier le texte "welcome back".

```
const { test, expect } = require('@playwright/test');

test('hello world', async ({ page }) => {
  await page.goto('/');
  await expect(page.getByText('welcome back')).toBeVisible();
});
```

Requête d'éléments avec Playwright

Utiliser des sélecteurs sémantiques pour requêter les éléments DOM.

```
await page.getByRole('button', { name: 'submit' }).click();
```

Tester les interactions de base

Tester la navigation et l'interaction de base.

```
const { test, expect } = require('@playwright/test');

test('navigates to another page', async ({ page }) => {
  await page.goto('/');
  await page.getByRole('link', { name: 'remotepizza' }).click();
  await expect(page.getByRole('heading', { name: 'pizza' })).toBeVisible();
});
```

Tester les formulaires avec Playwright

Utiliser des locators pour remplir et soumettre des formulaires.

```
const { test, expect } = require('@playwright/test');

test('should show success page after submission', async ({ page }) => {
  await page.goto('/signup');
  await page.getByLabel('first name').fill('Chuck');
  await page.getByLabel('last name').fill('Norris');
  await page.getByLabel('country').selectOption({ label: 'Russia' });
  await page.getByLabel('subscribe to our newsletter').check();
  await page.getByRole('button', { name: 'sign in' }).click();
  await expect(page.getByText('thank you for signing up')).toBeVisible();
});
```

Tester les liens ouvrant dans un nouvel onglet

Vérifier l'attribut href ou obtenir le handle de la nouvelle page après avoir cliqué sur le lien.

```
const pagePromise = context.waitForEvent('page');
await page.getByRole('link', { name: 'terms and conditions' }).click();
const newPage = await pagePromise;
await expect(newPage.getByText("I'm baby")).toBeVisible();
```

10

Jest

Écrire un test simple avec Jest

Structure d'un test simple avec Jest et comment exécuter un test.

```
function sum(a, b) {
  return a + b;
}

test('additionne 1 + 2 pour obtenir 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Matchers Jest

Introduction aux différents matchers disponibles dans Jest pour vérifier les résultats des tests.

```
test('deux plus deux fait quatre', () => {
  expect(2 + 2).toBe(4);
});
```

Tests d'assertion

Utilisation des assertions de base dans Jest comme toBe, toEqual, toBeNull, toBeUndefined, et toBeDefined.

```
test('null est nul', () => {
  expect(null).toBeNull();
  expect(undefined).toBeUndefined();
  expect(1).toBeDefined();
});
```

Tests de tableaux et objets

Vérification du contenu des tableaux et des objets avec les matchers toContain et toHaveProperty.

```
test('la liste contient le mot spécifique', () => {
  const shoppingList = ['couche', 'kleenex', 'savon'];
  expect(shoppingList).toContain('savon');
});
```

Tests d'exceptions

Tester les exceptions lancées par des fonctions avec le matcher toThrow.

```
function compilesAndroidCode() {
  throw new Error('Vous utilisez la mauvaise JDK');
}

test('compilation d\'Android génère une erreur', () => {
  expect(() => compilesAndroidCode()).toThrow('Vous utilisez la mauvaise JDK');
});
```

Tests de fonctions asynchrones

Tests de fonctions asynchrones avec des callbacks, des promesses et async/await.

```
test('les données de l\'API sont des utilisateurs', async () => {
  const data = await fetchData();
  expect(data).toEqual({ users: [] });
});
```

Jest Mock Functions

Création et utilisation de fonctions mock dans Jest avec jest.fn().

```
const myMock = jest.fn();
myMock();
expect(myMock).toHaveBeenCalled();
```

```
jest.mock('axios');
const axios = require('axios');

test('mocking axios', () => {
  axios.get.mockResolvedValue({ data: 'some data' });
  // test axios.get()
});
```

Exécuter des tests conditionnels

Exécution de tests conditionnels avec test.only et test.skip, et grouper les tests avec describe.

```
test.only('ce test est le seul à être exécuté', () => {
  expect(true).toBe(true);
});

test.skip('ce test est ignoré', () => {
  expect(true).toBe(false);
});

describe('groupe de tests', () => {
  test('test A', () => {
    expect(true).toBe(true);
  });

  test('test B', () => {
    expect(true).toBe(true);
  });
});
```