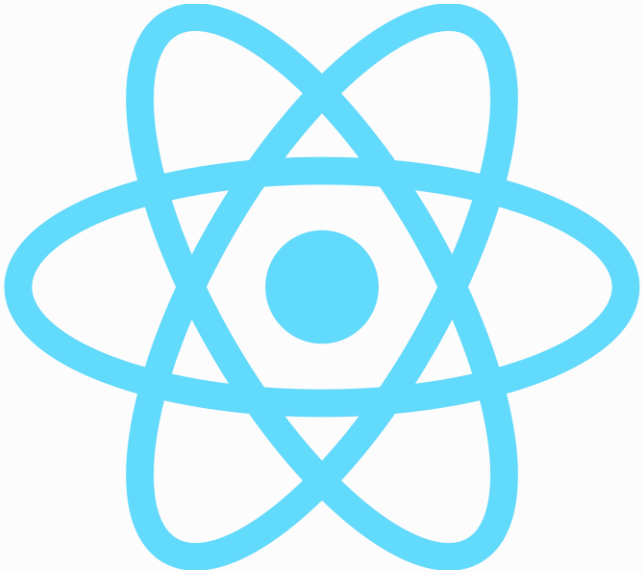
React JS



Support réalisé par Pierrick Hauguel

01

Les règles ESLINT à connaître

Configuration de base ESLint pour TypeScript

Configuration de base dans .eslintrc.json :

```
{
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended"
  ]
}
```

Règle @typescript-eslint/explicit-function-return-type

Description : Oblige à définir le type de retour des fonctions.
Avantage : Clarifie les attentes et évite les erreurs imprévues.

```
// Mauvais
function add(a: number, b: number) {
  return a + b;
}

// Correct
function add(a: number, b: number): number {
  return a + b;
}
```

Règle @typescript-eslint/no-explicit-any

Description : Interdit l'utilisation du type any.

Avantage : Encourage l'utilisation de types spécifiques pour éviter les erreurs au moment de l'exécution.

```
// Mauvais
function processData(data: any): void {
  console.log(data);
}

// Correct
function processData(data: string | number): void {
  console.log(data);
}
```

Règle @typescript-eslint/no-unused-vars

Description : Empêche la déclaration de variables inutilisées.
Avantage : Aide à maintenir un code propre et évite la confusion.

```
// Mauvais
const name: string = "John Doe";
const age: number = 30; // Variable inutilisée

// Correct
const name: string = "John Doe";
console.log(name);
```

Règle @typescript-eslint/consistent-type-definitions

Description : Force l'utilisation soit de interface soit de type pour les types d'objets.
Avantage : Assure une cohérence dans la définition des types.

```
// Choisir l'uniformité : Interface ou Type

// Mauvais : Mélange des deux
type UserType = {
  name: string;
  age: number;
};

interface UserInterface {
  name: string;
  age: number;
}

// Correct : Utiliser uniquement les interfaces ou les types
interface User {
  name: string;
  age: number;
}
```

Règle @typescript-eslint/no-inferrable-types

Description : Évite la redondance dans les déclarations de types qui peuvent être inférés.
Avantage : Simplifie le code et améliore sa lisibilité.

```
// Mauvais
const isActive: boolean = true;

// Correct
const isActive = true; // Type inféré automatiquement comme boolean
```

Règle @typescript-eslint/explicit-module-boundary-types

Description : Oblige à spécifier les types des arguments et des retours pour les fonctions exportées.
Avantage : Améliore la maintenabilité des API publiques.

```
// Mauvais
export function fetchData(url) {
  return fetch(url).then(res => res.json());
}

// Correct
export function fetchData(url: string): Promise<any> {
  return fetch(url).then(res => res.json());
}
```

Règle @typescript-eslint/no-non-null-assertion

Description : Interdit l'utilisation de l'opérateur ! pour les assertions non-nulles.
Avantage : Évite les erreurs liées à des valeurs nulles ou indéfinies.

```
// Mauvais
const userName: string | null = getUserName();
console.log(userName!.toUpperCase()); // Potentiellement dangereux

// Correct
const userName: string | null = getUserName();
if (userName) {
  console.log(userName.toUpperCase());
}
```

Règle @typescript-eslint/ban-types

Description : Interdit l'utilisation de certains types comme Object, Function, etc.
Avantage : Encourage l'utilisation de types plus spécifiques et sûrs.

```
// Mauvais
function logData(data: Object): void {
  console.log(data);
}

// Correct
function logData(data: Record<string, unknown>): void {
  console.log(data);
}
```

@typescript-eslint/no-empty-function

Description : Interdit les fonctions vides sans commentaires expliquant pourquoi elles sont vides.
Avantage : Encourage un code plus expressif et utile.

```
// Mauvais
function doNothing(): void {}

// Correct
function doNothing(): void {
  // Cette fonction est intentionnellement vide
}
```

Liste de toutes les règles

// @typescript-eslint/explicit-function-return-type
// @typescript-eslint/no-explicit-any
// @typescript-eslint/no-unused-vars
// @typescript-eslint/consistent-type-definitions
// @typescript-eslint/no-inferrable-types
// @typescript-eslint/explicit-module-boundary-types
// @typescript-eslint/no-non-null-assertion
// @typescript-eslint/ban-types
// @typescript-eslint/no-empty-function

02

Le Strict Mode

Qu'est-ce que le Strict Mode?

Définition: Le Strict Mode est un outil pour détecter les problèmes potentiels dans une application React.
Activation: Il est activé via le composant <React.StrictMode>.
Objectif: Aider les développeurs à identifier les problèmes de performance, les usages non sécurisés, et les pratiques obsolètes.

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Rendus en double: Pourquoi ?

Détection des effets secondaires: Le rendu en double permet d'identifier les effets de bord imprévus. Par exemple, si une opération mutative est effectuée dans un composant ou un hook, le double rendu peut révéler un comportement inattendu.

Test de l'idempotence: S'assurer que les composants n'ont pas d'effets de bord visibles et qu'ils sont "idempotents" (produisent le même résultat à chaque exécution).

Débogage des mises à jour de l'état: Le rendu en double aide à s'assurer que les mises à jour d'état sont correctement gérées sans introduire de bugs invisibles en mode production.

Avertissements et Dépréciations

APIs dépréciées: Le Strict Mode émet des avertissements pour les méthodes React qui seront bientôt obsolètes. Cela aide les développeurs à anticiper les changements futurs de l'API.

Legacy String Refs et findDOMNode: Il affiche des avertissements lorsque ces anciennes pratiques sont utilisées, car elles seront supprimées dans les futures versions de React.

Comment le Strict Mode aide-t-il à détecter les bugs?

Avertissement sur les effets non sécurisés: Le Strict Mode peut révéler des erreurs telles que les effets non sécurisés (par exemple, modifier l'état après un effet de bord) qui pourraient ne pas être évidentes sans le double rendu.

Exemple de bug subtil: Par exemple, une fonction asynchrone qui n'est pas correctement nettoyée peut entraîner des erreurs que le Strict Mode mettra en évidence.

03

Le concurrent mode

Introduction au Concurrent Mode en React

Le Concurrent Mode est une nouvelle fonctionnalité expérimentale de React qui permet à l'interface utilisateur de rester réactive même lorsque l'application effectue des tâches lourdes. Il permet de rendre les composants sans bloquer le fil d'exécution principal, améliorant ainsi l'expérience utilisateur.

Pourquoi le Concurrent Mode ?

Le Concurrent Mode répond au besoin d'interfaces utilisateur fluides et réactives, en particulier dans les applications complexes où le rendu peut être long. Il permet d'optimiser les performances en laissant React interrompre et reprendre le travail de rendu en fonction de la priorité.

Suspense pour le rendu asynchrone

Suspense est un composant React utilisé pour gérer le rendu asynchrone. Il permet d'afficher un fallback (comme un spinner) pendant que le contenu principal est en cours de chargement.

```
const MyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Chargement...</div>}>
      <MyComponent />
    </Suspense>
  );
}
```

Transition d'état avec useTransition

useTransition permet de marquer certaines mises à jour d'état comme de faible priorité, améliorant ainsi la réactivité de l'interface utilisateur.

```
const [isPending, startTransition] = useTransition();

function handleClick() {
  startTransition(() => {
    // Action de faible priorité
    setState(newValue);
  });
}
```

Utilisation de useDeferredValue

useDeferredValue permet de différer une valeur jusqu'à ce que le système soit prêt à la rendre, en maintenant la réactivité de l'interface pour les autres interactions.

```
const deferredValue = useDeferredValue(value);

return <ExpensiveComponent value={deferredValue} />;
```

04

Les

Hooks avancés

Le Hook useReducer

useReducer est une alternative à useState, idéale pour gérer des états plus complexes avec une logique d'état local basée sur des actions.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Accès aux Éléments DOM avec useRef

useRef permet de conserver une référence mutable à travers les re-renders du composant, souvent utilisé pour accéder à un élément DOM directement.

```
const myRef = useRef(initialValue);
```

useMemo

useMemo est un hook qui permet de mémoriser une valeur calculée et de l'empêcher d'être recalculée à chaque rendu du composant. Cela peut être utile pour les valeurs qui sont coûteuses à calculer, comme les fonctions complexes ou les appels à des API.

```
const ProductList = () => {
  const products = [
    { name: "Produit 1", price: 10 },
    { name: "Produit 2", price: 20 },
    { name: "Produit 3", price: 30 },
  ];

  const memoizedTotalPrices = useMemo(() => {
    const totalPrices = products.map(product => product.price);
    return totalPrices;
  }, [products]);

  return (
    <ul>
      {products.map((product, index) => (
        <li key={product.name}>
          {product.name} - {memoizedTotalPrices[index]}
        </li>
      ))}
    </ul>
  );
};
```

memo

```
const Button = memo(({ onClick }) => {
  const [isHovered, setIsHovered] = useState(false);

  const handleMouseEnter = () => {
    setIsHovered(true);
  };

  const handleMouseLeave = () => {
    setIsHovered(false);
  };

  return (
    <button
      onMouseEnter={handleMouseEnter}
      onMouseLeave={handleMouseLeave}
      style={{
        backgroundColor: isHovered ? "red" : "blue",
      }}
      onClick={onClick}
    >
      Bouton
    </button>
  );
});
```

memo est une fonction qui permet d'encapsuler un composant fonctionnel et de l'empêcher de se rendre inutilement. Cela est utile pour les composants purs, qui ne devraient se rendre que si leurs props ou leur état changent.

useCallback

useCallback est un hook qui permet de mémoriser une fonction et de l'empêcher d'être recréée à chaque rendu du composant. Cela peut être utile pour les callbacks qui sont transmis aux composants enfants, car cela évite de les recréer à chaque fois que le composant parent se rend.

```
const CommentList = () => {
  const [sortBy, setSortBy] = useState("date");

  const memoizedSortFn = useCallback((comments, sortBy) => {
    // Fonction de tri
    return comments.sort((a, b) => {
      if (sortBy === "date") {
        return a.date - b.date;
      } else {
        return a.author.localeCompare(b.author);
      }
    });
  }, [sortBy]);

  const sortedComments = memoizedSortFn(comments, sortBy);

  return (
    <ul>
      {sortedComments.map(comment => (
        <li key={comment.id}>
          {comment.author} - {comment.date}
        </li>
      ))}
    </ul>
  );
};
```

05

Redux et Zustand

Introduction à Redux

Redux est une bibliothèque de gestion d'état prévisible pour applications JavaScript. Elle aide à écrire des applications qui se comportent de manière consistante, tournent dans différents environnements (client, serveur et natif), et sont faciles à tester.

Principes fondamentaux de Redux

Redux repose sur quelques principes clés : l'état global de l'application est stocké dans un objet arbre unique au sein d'un seul store. L'état est en lecture seule. Les changements d'état sont effectués en envoyant des actions. Les réducteurs sont des fonctions pures qui prennent l'état précédent et une action, et retournent le nouvel état.

Configuration initiale de Redux

```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);
```

Pour commencer avec Redux, installez redux et react-redux. Ensuite, créez un store Redux et fournissez-le à votre application via le Provider de React-Redux.

Actions

Les actions sont des objets JavaScript qui envoient des données de votre application vers votre store. Elles sont la seule source d'informations pour le store.

```
const addAction = { type: 'ADD', payload: 1 };
```

Reducers

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'ADD':
      return state + action.payload;
    default:
      return state;
  }
}
```

Les reducers spécifient comment l'état de l'application change en réponse aux actions envoyées au store. Rappelez-vous que les actions décrivent le fait que quelque chose s'est passé, mais ne spécifient pas comment l'état de l'application change.

useSelector et useDispatch

useSelector permet d'accéder à l'état du store, tandis que useDispatch vous donne accès à la fonction dispatch pour envoyer des actions.

```
import { useSelector, useDispatch } from 'react-redux';

function Counter() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch({ type: 'ADD', payload: 1 })}>
        Increment
      </button>
      <span>{count}</span>
    </div>
  );
}
```

Store et gestion de l'état

```
import { combineReducers, createStore } from 'redux';
import counterReducer from './reducers/counterReducer';

const rootReducer = combineReducers({
  counter: counterReducer
});

const store = createStore(rootReducer);
```

- Le store de Redux sert de conteneur pour l'état global de votre application. Utilisez combineReducers pour diviser l'état et la logique de réduction en plusieurs fonctions gérant des parties indépendantes de l'état.

Middleware Redux

Les middlewares offrent un point d'extension entre l'envoi d'une action et le moment où elle atteint le réducteur. Utilisez redux-thunk pour gérer la logique asynchrone.

```
import { applyMiddleware, createStore } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

Introduction à Redux Toolkit

Le Redux Toolkit (RTK) est un ensemble d'outils visant à simplifier le code Redux, encourager les bonnes pratiques et améliorer la dévelopabilité avec Redux. Il offre des utilitaires pour simplifier la configuration du store, la définition des reducers, la gestion de la logique asynchrone, et plus encore.

Avantages par rapport à Redux standard

RTK réduit la quantité de code boilerplate nécessaire pour configurer un store Redux, simplifie la gestion des actions et des reducers avec createSlice, automatise la création d'actions, et facilite la gestion de la logique asynchrone avec createAsyncThunk.

Installation et configuration

- Pour démarrer avec RTK, installez le paquet @reduxjs/toolkit ainsi que react-redux si vous travaillez avec React.

Configurer le Store avec configureStore

configureStore simplifie la configuration du store en incluant automatiquement des middlewares comme Redux Thunk et en activant les outils de développement Redux.

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

export const store = configureStore({
  reducer: {
    // Reducers vont ici
  },
});
```

Création de Slice avec createSlice

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

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

export const { increment, decrement } = counterSlice.actions;
```

- createSlice permet de regrouper les reducers et les actions correspondantes dans un seul objet, simplifiant ainsi la gestion de l'état.

Gestion des effets secondaires avec createAsyncThunk

createAsyncThunk simplifie la gestion des opérations asynchrones en encapsulant la logique asynchrone et le traitement des états de la requête (loading, success, error) dans une seule fonction.

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

export const fetchUserData = createAsyncThunk(
  'users/fetchById',
  async (userId, thunkAPI) => {
    const response = await fetch(`https://api.example.com/users/
    return await response.json();
  }
);
```

Utilisation de useDispatch et useSelector avec Redux Toolkit et React

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './slices/counterSlice';

function CounterComponent() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <span>{count}</span>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}
```

- RTK fonctionne de manière transparente avec les hooks useDispatch et useSelector de react-redux, facilitant l'accès à l'état et la dispatch d'actions dans les composants React.

Introduction à Zustand

Zustand est une petite bibliothèque de gestion d'état pour React qui offre une approche plus simple et plus directe que Redux. Avec Zustand, la création de stores globaux pour gérer l'état est simplifiée et ne nécessite pas de boilerplate ou de middleware supplémentaire.

Philosophie et avantages de Zustand

- Zustand se concentre sur une API minimaliste et un hook personnalisé pour accéder au store. Les avantages incluent une configuration facile, pas de dépendance à Redux ou Context API, et une intégration naturelle avec les hooks de React.

Installation de Zustand

L'installation de Zustand est simple et directe, en utilisant npm ou yarn. Ceci installe la dernière version de Zustand dans votre projet.

```
npm install zustand
// ou
yarn add zustand
```

Création d'un store avec Zustand

```
import create from 'zustand';

const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 })),
  decrement: () => set(state => ({ count: state.count - 1 })),
}));
```

La création d'un store dans Zustand se fait en utilisant la fonction create. Vous pouvez y définir l'état initial du store et les actions qui manipuleront cet état. Zustand permet de créer des stores sans l'overhead traditionnellement associé à Redux.

Gestion de l'état avec des hooks

Zustand utilise des hooks pour accéder et manipuler l'état du store. Cela rend l'intégration avec les composants fonctionnels React naturelle et efficace.

```
function Counter() {
  const { count, increment, decrement } = useStore();
  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  );
}
```

Gestion des effets secondaires

```
import create from 'zustand';
import { useEffect } from 'react';

const useStore = create(set => ({
  data: null,
  fetchData: async () => {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    set({ data });
  },
}));

// Dans un composant
const Component = () => {
  const { data, fetchData } = useStore();
  useEffect(() => {
    fetchData();
  }, [fetchData]);

  return <div>{data && <p>{data.someField}</p>}</div>;
};
```

Zustand permet de gérer des effets secondaires en utilisant des actions ou en réagissant à des changements d'état spécifiques à l'aide de middlewares ou de l'API native React.useEffect.

Sélecteurs et abonnements

Zustand permet de sélectionner une partie de l'état lors de l'utilisation du hook du store, réduisant ainsi les re-renders inutiles. Les abonnements aux changements d'état sont également possibles pour une gestion fine des mises à jour.

```
const count = useStore(state => state.count);
```

Exemples d'utilisation avancée de Zustand

```
import create from 'zustand';
import { persist } from 'zustand/middleware';

const useStore = create(persist(set => ({
  user: null,
  setUser: user => set({ user }),
}), {
  name: 'user-settings', // nom de la clé du local storage
}));
```

Zustand supporte des cas d'utilisation avancés comme le partage de l'état entre différents composants, la persistance de l'état dans le local storage, et l'intégration avec des outils de débogage.

Exercice !

Remplacez la centralisation d’état par zustand !

Pour aller plus loin…

Daishi Kato : https://twitter.com/dai_shi

Vidéo de Jack Herrington : https://www.youtube.com/watch?v=sqTPGMipjHk

06

Les Contexts

Introduction au Contexte dans React

Le Contexte permet de partager des valeurs facilement entre plusieurs composants, sans nécessiter de passer explicitement une prop à chaque niveau de l'arbre des composants. Il est idéal pour des données dites "globales" telles que le thème, les préférences utilisateur, etc.

Création et Utilisation du Contexte

La mise en place d'un contexte commence par React.createContext(), qui retourne un objet contenant un composant Provider et Consumer. Le Provider permet de fournir une valeur de contexte à tous les composants enfants qui l'encapsulent.

```
const MyContext = React.createContext(defaultValue);
```

Fournir des Valeurs avec le Provider

```
<MyContext.Provider value={/* some value */}>
  {/* child components */}
</MyContext.Provider>
```

Le composant Provider sert à fournir une valeur de contexte à l'arbre des composants, permettant ainsi aux composants enfants d'accéder à ces données sans passer par une prop.

Accéder aux Valeurs de Contexte

Les valeurs de contexte sont accessibles aux composants enfants via le composant Consumer, le hook useContext dans les composants fonctionnels, ou MyContext.Consumer et static contextType dans les composants de classe.

```
const value = useContext(MyContext);
```

Meilleures Pratiques avec le Contexte

- **Restriction d'Usage :** Utilisez le contexte pour des données globales qui changent rarement.
- **Mise à Jour du Contexte :** Optimisez les mises à jour pour éviter les rendus inutiles.
- **Composition de Contextes :** Utilisez plusieurs contextes pour séparer les préoccupations sans recourir excessivement au "prop drilling".

Exemple

```
const ThemeContext = createContext({
  theme: "light",
  toggleTheme: () => {},
});

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

const MyComponent = () => {
  const { theme } = useContext(ThemeContext);

  return (
    <div>
      The current theme is: {theme}
    </div>
  );
};
```

Exercice !

Centralisez vos appels grâce à un context.

07

Stylisation et CSS en React

Approches de la Stylisation dans React

React offre plusieurs méthodes pour styliser les composants, du CSS traditionnel à des approches plus modernes et dynamiques, jouant un rôle crucial dans la création d'interfaces utilisateur attrayantes et cohérentes.

Utiliser le CSS Traditionnel dans React

L'importation de feuilles de style CSS est la méthode la plus directe et familière pour appliquer des styles, permettant une intégration facile des styles globaux ou spécifiques à un composant.

```
import './App.css';
```

Styled-components

```
const Button = styled.button`
  background-color: blue;
  color: white;
`;
```

Styled-components permet de créer des composants stylisés en utilisant des littéraux de gabarit tagués, intégrant étroitement styles et logique de composants pour un theming dynamique et une réutilisation facilitée.

Appliquer des Styles Inline

```jsx
// Exemple d'application de styles inline dans un composant React
const dynamicStyles = {
  backgroundColor: 'green',
  color: 'white',
};

function MyComponent() {
  return <div style={dynamicStyles}>Hello World</div>;
}
```

L'utilisation de styles inline via l'attribut style permet d'appliquer des styles dynamiques directement sur des éléments, utile pour des ajustements rapides ou des styles conditionnels.

Explorer Tailwind CSS et Emotion

Des bibliothèques comme Tailwind CSS et Emotion offrent des approches alternatives pour la stylisation, combinant les avantages de la modularité, de la réutilisabilité et de l'encapsulation des styles.

07.1

Les Styled-components

Introduction aux composants stylisés

- Les composants stylisés permettent de créer des composants réutilisables avec des styles encapsulés.
- Ils offrent une alternative aux approches CSS traditionnelles, avec une syntaxe plus concise et une meilleure séparation des préoccupations

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Cliquez-moi</Button>
    </div>
  );
};
```

Définition et avantages des composants stylisés

```
const Button = styled.button`
  background-color: ${props => props.color};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button color="red">Bouton rouge</Button>
      <Button color="blue">Bouton bleu</Button>
    </div>
  );
};
```

- **Avantages:**
  - Amélioration de la lisbilité du code.
  - Réutilisation des styles.
  - Meilleure séparation des préoccupations.
  - Facilité de maintenance.

Créer et utiliser des composants stylisés

Créer un composant stylisé avec styled. Utiliser le composant comme n'importe quel autre composant React.

```
const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Cliquez-moi</Button>
    </div>
  );
};
```

Utilisation de props et de variantes

```
const Button = styled.button`
  background-color: ${props => props.color || 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;

  ${props => props.primary && `
    background-color: green;
  `}
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Bouton bleu</Button>
      <Button color="red">Bouton rouge</Button>
      <Button primary>Bouton principal</Button>
    </div>
  );
};
```

- Les props permettent de personnaliser les styles des composants stylisés.
- Les variantes facilitent la création de variations de style pour un composant.

Héritage de styles et composition de composants

- L'héritage de styles permet de réutiliser les styles d'un composant parent dans un composant enfant.
- La composition de composants permet de créer des composants plus complexes en combinant des composants plus simples.

```
const Container = styled.div`
  padding: 20px;
  border: 1px solid #ddd;
`;

const Title = styled.h2`
  margin-bottom: 10px;
`;

const Content = styled.p`
`;

const MyComponent = () => {
  return (
    <Container>
      <Title>Titre</Title>
      <Content>Contenu du composant</Content>
    </Container>
  );
};
```

Gestion des styles dynamiques

```
const Button = styled.button`
  background-color: ${props => props.isHovered ? 'red' : 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  const [isHovered, setIsHovered] = useState(false);

  return (
    <div>
      <Button isHovered={isHovered} onMouseEnter={() => setIsHovered(true)}>
        Bouton avec style dynamique
      </Button>
    </div>
  );
};
```

- Les styles dynamiques permettent de générer des styles en fonction de l'état du composant ou des props.
- Ils peuvent être utilisés pour créer des interfaces utilisateur plus réactives et interactives.

Animations et transitions

- Les animations et les transitions permettent de rendre les interfaces utilisateur plus fluides et interactives.
- Elles peuvent être utilisées pour attirer l'attention de l'utilisateur ou pour améliorer l'expérience utilisateur.

```
const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;

  transition: all 0.2s ease-in-out;

  &:hover {
    background-color: red;
  }
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Hover Me</Button>
    </div>
  );
};
```

07.2

Le Package Emotion

Introduction à Emotion

```
import styled from '@emotion/styled';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );
};
```

- Emotion est une bibliothèque JavaScript pour créer des styles CSS dynamiques et réutilisables.
- Elle offre une alternative aux solutions CSS traditionnelles, avec une syntaxe plus concise et une meilleure séparation des préoccupations.

Définition et avantages d'Emotion

```
const Button = styled.button`
  background-color: ${props => props.color};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button color="red">Red Button</Button>
      <Button color="blue">Blue Button</Button>
    </div>
  );
};
```

- Amélioration de la lisbilité du code.
- Réutilisation des styles.
- Meilleure séparation des préoccupations.
- Facilité de maintenance.

Fonctionnement et concepts clés d'Emotion

```
const Button = styled.button`
  background-color: ${props => props.theme.colors.primary};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Button with theme</Button>
    </div>
  );
};
```

Emotion injecte des styles CSS dans le DOM via des attributs data-emotion.
Les styles sont générés de manière unique pour chaque composant.

Créer et utiliser des styles avec Emotion

Création de composants stylisés avec styled d'Emotion.
Utilisation des composants stylisés comme des composants React classiques.

```
import styled from '@emotion/styled';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );
};
```

Syntaxe de base avec @emotion/core

```
import { css } from '@emotion/core';

const buttonStyles = css`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

<button className={buttonStyles}>Click me</button>
```

@emotion/core est le package principal d'Emotion.
Il fournit les fonctionnalités de base pour la création de styles.
Syntaxe de base pour définir des styles avec css:

Utilisation de props et de variantes

```
const Button = styled.button`
  background-color: ${props => props.color || 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;

  ${props => props.primary && `
    background-color: green;
  `}
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Blue Button</Button>
      <Button color="red">Red Button</Button>
      <Button primary>Main Button</Button>
    </div>
  );
};
```

- **Props:**
  - Permettent de personnaliser les styles des composants stylisés.
- **Variantes:**
  - Facilité la création de variations de style pour un composant.

08

Utilisation de React Hook Form

Introduction à React Hook Form

React Hook Form est une bibliothèque légère pour gérer les formulaires dans React, en se basant sur les hooks pour simplifier le processus. Elle offre des performances optimisées et une syntaxe simple pour une meilleure expérience de développement.

Utilisation des Hooks dans React Hook Form

React Hook Form utilise des hooks comme useForm, useFieldArray, et useWatch pour gérer l'état et le comportement des formulaires, offrant ainsi une approche minimaliste et performante.

```
import { useForm } from 'react-hook-form';

const { register, handleSubmit } = useForm();
```

Configuration de Base

```
npm install react-hook-form
```

```
import { useForm } from 'react-hook-form';

const { register, handleSubmit } = useForm();
```

Pour commencer à utiliser React Hook Form, installez-le dans votre projet React et initialisez l'état du formulaire à l'aide du hook useForm.

Enregistrement des Champs et Gestion des Soumissions

Utilisez register pour enregistrer les champs du formulaire et handleSubmit pour gérer la soumission du formulaire et exécuter une fonction de rappel.

```
<input {...register("email")} />

const onSubmit = (data) => console.log(data);

<form onSubmit={handleSubmit(onSubmit)}>
```

Performance Optimisée et Syntaxe Légère

React Hook Form offre une performance optimisée en réduisant les rerenders inutiles et une syntaxe légère qui simplifie la gestion des formulaires dans les applications React.

09.1

Utilisation de Tanstack Query

Démarrer avec Tanstack Quer

```jsx
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const App = () => {
  const queryClient = new QueryClient();

  return (
    <QueryClientProvider client={queryClient}>
      <MyComponent />
    </QueryClientProvider>
  );
};
```

Ce module vous guide à travers l'installation et la configuration de Tanstack Query v5 dans votre projet React. Vous apprendrez à installer les packages nécessaires, à configurer la bibliothèque et à comprendre les concepts fondamentaux.

Comprendre les concepts clés de Tanstack Query

Ce module explore les concepts fondamentaux de Tanstack Query, tels que les requêtes, les mutations, l'état des données, l'invalidation du cache et les hooks de base.

```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error occurred.</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    ))}
  </ul>
);
```

Gérer les erreurs et les chargements avec Tanstack Query

```
const { data, status, error, isFetching } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error has occurred: {error.message}</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    ))}
  </ul>
);
```

Ce module vous montre comment gérer les erreurs et les chargements dans vos applications Tanstack Query. Vous apprendrez à afficher des messages d'erreur, à gérer les re-tentatives et à utiliser des statuts de chargement.

Optimiser les performances avec Tanstack Query

Ce module vous montre comment optimiser les performances de vos applications Tanstack Query. Vous apprendrez à utiliser le cache de données, à effectuer des requêtes sélectives et à mettre en place la pagination.

```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
  options: {
    cacheTime: 10000, // Cache data for 10 seconds
    staleTime: 5000, // Allow data to be stale for 5 seconds
  },
});
```

Pour aller plus loin

TkDodo: https://twitter.com/TkDodo
Le compte du concurrent : (SWR)
https://twitter.com/huozhi

09.2

Utilisation avancée de Tanstack Query

Introduction aux Mutations avec Tanstack Query

Les mutations sont utilisées dans Tanstack Query pour effectuer des modifications de données comme les insertions, mises à jour, ou suppressions.

```
import { useMutation } from '@tanstack/react-query';

const createItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItem() {
  const mutation = useMutation(createItem);
  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add Item</button>;
}
```

Base des Mutations

Apprendre à utiliser useMutation pour créer, mettre à jour ou supprimer des données.

```
import { useMutation } from '@tanstack/react-query';

const updateItem = async (item) => {
  return fetch(`/api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  }).then(res => res.json());
};

function EditItem({ item }) {
  const { mutate } = useMutation(updateItem, {
    onSuccess: () => alert('Item updated successfully!')
  });

  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
}
```

Gestion des États de Mutation

Gestion des états de mutation pour afficher les retours visuels appropriés tels que les chargements, les succès et les erreurs.

```
import { useMutation } from '@tanstack/react-query';

const deleteItem = id => fetch(`/api/items/${id}`, { method: 'DELETE' });

function DeleteItem({ itemId }) {
  const { mutate, isLoading, isError, error } = useMutation(() => deleteItem(itemId), {
    onSuccess: () => alert('Item deleted successfully!'),
    onError: () => alert('Error deleting item!')
  });

  if (isLoading) return <div>Deleting...</div>;
  if (isError) return <div>Error: {error.message}</div>;

  return <button onClick={() => mutate()}>Delete Item</button>;
}
```

Synchronisation avec le Cache après une Mutation

Comment utiliser les mutations pour non seulement modifier les données, mais aussi synchroniser ces modifications avec le cache local.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

const addItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItemToList() {
  const queryClient = useQueryClient();
  const mutation = useMutation(addItem, {
    onSuccess: () => {
      queryClient.invalidateQueries(['items']);
    }
  });

  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add to List</button>;
}
```

Utilisation des Rollbacks en cas d'Échec

Implémenter des rollbacks pour restaurer l'état précédent en cas d'échec de la mutation.

```
const updateItem = async (item) => {
  const response = await fetch(`/api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  if (!response.ok) {
    throw new Error('Failed to update item');
  }
  return response.json();
};

function UpdateItem({ item }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateItem, {
    onMutate: async (newItem) => {
      await queryClient.cancelQueries(['items']);
      const previousItems = queryClient.getQueryData(['items']);
      queryClient.setQueryData(['items'], old => old.map(d => d.id === newItem.id ? newItem : d));
      return { previousItems };
    },
    onError: (err, newItem, context) => {
      queryClient.setQueryData(['items'], context.previousItems);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['items']);
    }
  });

  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
```

Requêtes paginées

Les requêtes paginées sont utilisées pour charger les données par page, ce qui réduit la charge sur le serveur et améliore l'expérience utilisateur en chargeant les données progressivement.

```
const fetchProjects = async (page = 0) => {
  const response = await fetch(`/api/projects?page=${page}`);
  return response.json();
};

function PaginatedProjects() {
  const [page, setPage] = useState(0);
  const { data: projects, isLoading, isError, error } = useQuery({
    queryKey: ['projects', page],
    queryFn: () => fetchProjects(page),
    keepPreviousData: true,
  });

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

  return (
    <div>
      {projects.map(project => <div key={project.id}>{project.name}</div>)}
      <button onClick={() => setPage(old => old + 1)}>Next Page</button>
    </div>
  );
}
```

Introduction à useInfiniteQuery

Configurer useInfiniteQuery nécessite de spécifier comment récupérer les données et comment identifier la prochaine page à charger.

```
import { useInfiniteQuery } from '@tanstack/react-query';

const fetchPosts = async ({ pageParam = 1 }) => {
  const response = await fetch(`/api/posts?page=${pageParam}`);
  return response.json();
};

function InfinitePosts() {
  const { data, fetchNextPage, hasNextPage } = useInfiniteQuery(['posts'], fetchPosts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? undefined
  });

  return (
    <div>
      {data.pages.map((page, i) => (
        <React.Fragment key={i}>
          {page.data.map(post => <p key={post.id}>{post.title}</p>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={() => fetchNextPage()}>Load More</button>}
    </div>
  );
}
```

Configurer useInfiniteQuery

Configurer useInfiniteQuery nécessite de spécifier comment récupérer les données et comment identifier la prochaine page à charger.

```
function Products() {
  const {
    data,
    error,
    fetchNextPage,
    hasNextPage,
    isLoading,
    isFetchingNextPage
  } = useInfiniteQuery(['products'], fetchProducts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? false
  });

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>An error occurred: {error.message}</div>;

  return (
    <div>
      {data.pages.map((page, index) => (
        <React.Fragment key={index}>
          {page.items.map(item => <div key={item.id}>{item.name}</div>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={fetchNextPage}>Load More</button>}
      {isFetchingNextPage && <div>Loading more...</div>}
    </div>
```

Requêtes simultanées

Utilisation du hook useQueries pour exécuter plusieurs requêtes simultanément. Cela est utile pour charger plusieurs données indépendantes en même temps.

```
import { useQueries } from '@tanstack/react-query';

function FetchMultipleData() {
  const results = useQueries({
    queries: [
      { queryKey: ['user', 1], queryFn: () => fetch('/api/user/1').then(res => res.json()) },
      { queryKey: ['post', 1], queryFn: () => fetch('/api/post/1').then(res => res.json()) }
    ]
  });

  return (
    <div>
      <h1>User: {results[0].data?.name}</h1>
      <h2>First Post: {results[1].data?.title}</h2>
    </div>
  );
}
```

Configurer useQueries pour des requêtes multiples

useQueries permet de lancer plusieurs requêtes simultanées. Chaque requête est indépendante mais configurée dans un seul appel de hook, ce qui simplifie la gestion de plusieurs sources de données.

```
const fetchResource = (resource) => fetch(`/api/${resource}`).then(res => res.json());

function Resources() {
  const resourceNames = ['users', 'posts', 'comments'];
  const queryResults = useQueries({
    queries: resourceNames.map(name => ({
      queryKey: [name],
      queryFn: () => fetchResource(name),
      staleTime: 5000
    }))
  });

  if (queryResults.some(query => query.isLoading)) return <div>Loading...</div>;

  return (
    <div>
      {queryResults.map((result, idx) => (
        <div key={idx}>
          <h3>{resourceNames[idx]}</h3>
          <ul>
            {result.data.map(item => <li key={item.id}>{item.title || item.name}</li>)}
          </ul>
        </div>
      ))}
```

Exemple simple avec useQueries

Utilisation de useQueries pour charger des données de différents endpoints API simultanément.

```
const fetchById = (type, id) => fetch(`/api/${type}/${id}`).then(res => res.json());

function FetchMultipleIds() {
  const ids = [1, 2, 3];
  const result = useQueries({
    queries: ids.map(id => ({
      queryKey: ['data', id],
      queryFn: () => fetchById('data', id)
    }))
  });

  return (
    <div>
      {result.map((res, index) => (
        <div key={index}>
          <h3>Data {ids[index]}</h3>
          <p>{res.data?.detail}</p>
        </div>
      ))}
    </div>
  );
}
```

Optimisation des requêtes simultanées

Stratégies pour optimiser les performances et l'efficacité des requêtes simultanées, comme le partage du cache et la réduction des appels réseau.

```
function OptimizedMultipleQueries() {
  const types = ['profile', 'settings', 'notifications'];
  const queries = useQueries({
    queries: types.map(type => ({
      queryKey: [type],
      queryFn: () => fetchData(type),
      staleTime: 10000, // Use a longer stale time to reduce refetching
      cacheTime: 300000 // Keep data in cache longer
    }))
  });

  return (
    <div>
      {queries.map((query, index) => (
        <div key={index}>
          <h3>{types[index]}</h3>
          {query.isLoading ? <p>Loading...</p> : <p>{query.data?.name}</p>}
        </div>
      ))}
    </div>
  );
}
```

Gestion des dépendances entre requêtes

Gérer les dépendances entre les requêtes avec useQuery pour s'assurer que certaines données sont chargées avant d'autres.

```
import { useQuery } from '@tanstack/react-query';

const getUser = () => fetch('/api/user').then(res => res.json());
const getUserPosts = userId => fetch(`/api/users/${userId}/posts`).then(res => res.json());

function DependentQueries() {
  const userQuery = useQuery(['user'], getUser);
  const postsQuery = useQuery(['user', 'posts'], () => getUserPosts(userQuery.data.id), {
    enabled: !!userQuery.data // Only run this query if the user data is available
  });

  return (
    <div>
      {userQuery.isLoading ? <p>Loading user...</p> : <h1>{userQuery.data.name}</h1>}
      {postsQuery.isLoading ? <p>Loading posts...</p> : postsQuery.data.map(post => <p key={post.id}>{post.title}</p>)}
    </div>
  );
}
```

Préchargement de données

Présentation de la fonctionnalité de préchargement dans Tanstack Query pour améliorer l'expérience utilisateur en chargeant les données avant qu'elles ne soient nécessaires.

```
import { useQuery, QueryClient } from '@tanstack/react-query';

const queryClient = new QueryClient();
const fetchProductDetails = id => fetch(`/api/products/${id}`).then(res => res.json());

// Préchargement des données d'un produit
queryClient.prefetchQuery(['product', 1], () => fetchProductDetails(1));

function ProductDetails() {
  const { data: product } = useQuery(['product', 1], () => fetchProductDetails(1));

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
    </div>
  );
}
```

Préchargement de données

Présentation de la fonctionnalité de préchargement dans Tanstack Query pour améliorer l'expérience utilisateur en chargeant les données avant qu'elles ne soient nécessaires.

Invalidation sélective du cache

L'invalidation sélective permet de rafraîchir certaines données dans le cache sans affecter les autres, ce qui est crucial pour maintenir les données synchronisées avec les serveurs.

```
import { useQueryClient, useQuery } from '@tanstack/react-query';

const fetchUser = id => fetch(`/api/users/${id}`).then(res => res.json());

function UserProfile({ userId }) {
  const queryClient = useQueryClient();
  useQuery(['user', userId], () => fetchUser(userId), {
    onSuccess: () => {
      // Invalider et rafraîchir les dépendances
      queryClient.invalidateQueries(['posts', userId]);
    }
  });

  return <div>User Profile</div>;
}
```

Invalidation sélective du cache

L'invalidation sélective permet de rafraîchir certaines données dans le cache sans affecter les autres, ce qui est crucial pour maintenir les données synchronisées avec les serveurs.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

const updateUser = userData => fetch('/api/user', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(userData)
});

function UpdateUser() {
  const queryClient = useQueryClient();
  const mutation = useMutation(updateUser, {
    onSuccess: () => {
      // Invalider uniquement les requêtes liées à l'utilisateur mis à jour
      queryClient.invalidateQueries(['userProfile']);
    }
  });

  return <button onClick={() => mutation.mutate({ id: 1, name: 'Jane Doe' })}>Update User</button>;
}
```

Introduction aux Optimistic Updates

Les Optimistic Updates sont une stratégie pour améliorer l'expérience utilisateur dans les applications web interactives, en appliquant immédiatement les modifications présumées sans attendre la confirmation du serveur.

```
function UpdateUserComponent({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateUser, {
    onMutate: async newUser => {
      await queryClient.cancelQueries(['user', newUser.id]);
      const previousUser = queryClient.getQueryData(['user', newUser.id]);
      queryClient.setQueryData(['user', newUser.id], newUser);
      return { previousUser };
    },
    onError: (err, newUser, context) => {
      queryClient.setQueryData(['user', newUser.id], context.previousUser);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['user', user.id]);
    }
  });

  return (
    <button onClick={() => mutate({ ...user, name: 'Updated Name' })}>
      Update Name
    </button>
  );
}
```

Exemple pratique d'Optimistic Update

Démonstration d'un Optimistic Update avec un exemple où un utilisateur peut activer ou désactiver une fonctionnalité de notification instantanément.

```
const toggleNotification = async (userId, enabled) => {
  return fetch(`/api/users/${userId}/notification`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ enabled })
  }).then(res => res.json());
};

function NotificationToggle({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(() => toggleNotification(user.id, !user.notificationEnabled), {
    onMutate: async newSetting => {
      await queryClient.cancelQueries(['user', user.id]);
      const previousUser = queryClient.getQueryData(['user', user.id]);
      queryClient.setQueryData(['user', user.id], {
        ...user,
        notificationEnabled: !user.notificationEnabled
      });
      return { previousUser };
    },
    onError: (err, newSetting, context) => {
      queryClient.setQueryData(['user', user.id], context.previousUser);
    }
  });

  return (
    <button onClick={() => mutate()}>
      {user.notificationEnabled ? 'Disable' : 'Enable'} Notifications
    </button>
  );
}
```

Principes de base du caching

Le caching avec Tanstack Query permet de stocker les résultats de requêtes et de les réutiliser pour améliorer la rapidité des chargements de données et réduire la charge sur les serveurs backend.

```
import { useQuery } from '@tanstack/react-query';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataComponent() {
  const { data } = useQuery(['data'], fetchData, {
    cacheTime: 5 * 60 * 1000, // Cache les données pendant 5 minutes
  });

  return <div>{data.title}</div>;
}
```

Configurer le caching pour des performances optimisées

Configuration du cache pour maximiser les performances, en définissant des politiques de durée de vie du cache et en personnalisant les stratégies de récupération des données.

```
import { useQuery, useQueryClient } from '@tanstack/react-query';

const fetchProduct = id => fetch(`/api/products/${id}`).then(res => res.json());

function Product({ productId }) {
  const queryClient = useQueryClient();
  const { data: product } = useQuery(['product', productId], () => fetchProduct(productId), {
    onSuccess: () => {
      // Précharger les données relatives dès que le produit est chargé
      queryClient.prefetchQuery(['reviews', productId], () => fetch(`/api/products/${productId}/reviews`));
    }
  });

  return <div>{product.name}</div>;
}
```

Utilisation de Suspense avec React Query

Explication de l'intégration de React Suspense avec React Query pour gérer les états de chargement de manière plus déclarative et homogène.

```
import { useQuery } from '@tanstack/react-query';
import { Suspense } from 'react';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataLoader() {
  const { data } = useQuery(['data'], fetchData, {
    suspense: true // Activer Suspense
  });

  return <div>{data.title}</div>;
}

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <DataLoader />
    </Suspense>
  );
}
```

Intégration de Suspense avec les composants React

Techniques pour intégrer Suspense avec les composants React, facilitant la gestion des dépendances de données et la manipulation des états de chargement.

```
import { useQuery } from '@tanstack/react-query';
import { Suspense, useState } from 'react';

const fetchDetails = id => fetch(`/api/details/${id}`).then(res => res.json());

function DetailsComponent({ id }) {
  const { data: details } = useQuery(['details', id], () => fetchDetails(id), {
    suspense: true
  });

  return <div>{details.description}</div>;
}

function DetailsLoader() {
  const [id, setId] = useState(1);

  return (
    <div>
      <button onClick={() => setId(id + 1)}>Next</button>
      <Suspense fallback={<div>Loading details...</div>}>
        <DetailsComponent id={id} />
      </Suspense>
    </div>
  );
}
```

10

Mise en place de tests dans React

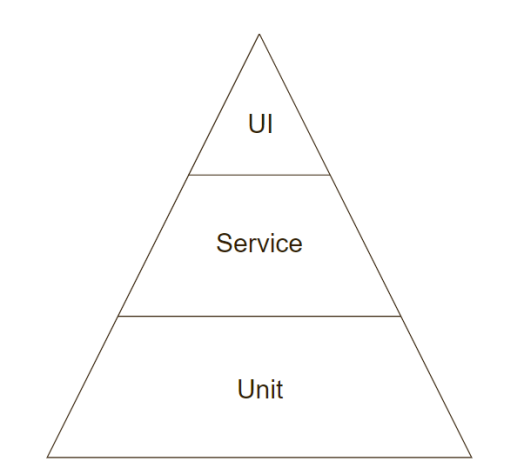
Introduction au test moderne de React

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.

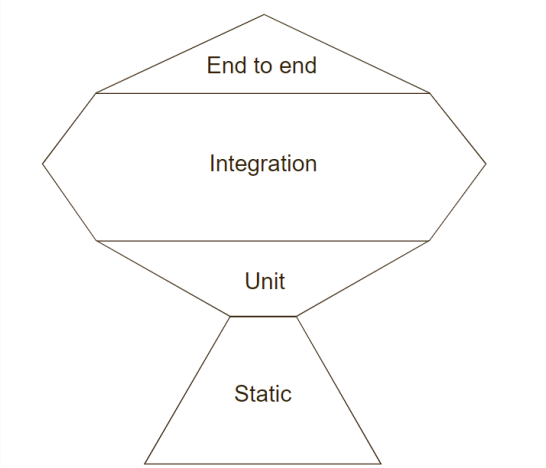
Présentation de la pyramide de tests

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-testing-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.

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);
});
```

Introduction aux tests unitaires avec Jest

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 : maximiser le retour sur investissement

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!');
});
```

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);
  });
});
```

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();
});
```

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);
});
```

Snapshots

Création et mise à jour des snapshots avec Jest pour les composants React.

```
import renderer from 'react-test-renderer';
import App from './App';

test('crée un snapshot de App', () => {
  const tree = renderer.create(<App />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

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();
});
```

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();
});
```

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();
});
```

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
```

Installation de MSW

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

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

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'] }
    });
  })
];
```

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 />);
});
```

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();
```

Tester les requêtes réseau avec MSW

Utiliser MSW pour simuler les réponses des API dans les tests.

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

test('load ingredients asynchronously', async ({ page }) => {
  await page.goto('/remote-pizza');
  await expect(page.getByText('bacon')).toBeHidden();
  await page.getByRole('button', { name: 'cook' }).click();
  await expect(page.getByText('bacon')).toBeVisible();
  await expect(page.getByRole('button', { name: 'cook' })).toBeDisabled();
});
```

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();
});
```

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"
  }
}
```

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',
  },
});
```

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
```

Écrire un Test Unitaire Basique avec Vitest

Exemple de test unitaire pour une fonction simple.

```
// utils.js
export function capitalize(str) {
  return str[0].toUpperCase() + str.slice(1);
}

// utils.test.js
import { describe, it, expect } from 'vitest';
import { capitalize } from './utils';

describe('capitalize function', () => {
  it('should capitalize the first letter', () => {
    expect(capitalize('test')).toBe('Test');
  });

  it('should handle empty strings', () => {
    expect(capitalize('')).toBe('');
  });
});
```

Mocking avec Vitest

Comment utiliser les fonctions de mock de Vitest pour simuler des comportements (mock de modules, fonctions, timers).

```
import { vi } from 'vitest';
import { fetchData } from './api';

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

test('mocks API call', async () => {
  const data = await fetchData();
  expect(data).toEqual({ data: 'mocked data' });
});
```

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
});
```

Utilisation de screen dans RTL

Introduction à l'API screen pour interagir avec le DOM généré dans les tests.

```
render(<Button label="Submit" />);
expect(screen.getByText('Submit')).toBeInTheDocument();
```

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' });
});
```

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,
},
```

Tests Paramétrés pour Cas Multiples

Utiliser les tests paramétrés avec Vitest pour éviter la duplication des tests pour des cas similaires.

```
it.each([
  [1, 1, 2],
  [2, 2, 4],
  [3, 3, 6]
])('adds %i and %i to get %i', (a, b, expected) => {
  expect(add(a, b)).toBe(expected);
});
```

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
```