TABLE DES MATIERES

INTRODUCTION

React est une bibliothèque JavaScript populaire utilisée pour créer des interfaces utilisateur (UI) dynamiques et interactives (React nous permet d'écrire du HTML dynamique). Elle permet de développer des applications pour diverses plateformes, telles que le web, le mobile, et même des expériences en réalité augmentée ou virtuelle, tout en restant centré sur la réutilisation des **composants**.

Lors de la création d'une application web avec React, on écrit du code en JavaScript et en **JSX** (JavaScript XML), qui est une syntaxe permettant de combiner JavaScript avec du *HTML-like*. Ce code est ensuite transformé (transpilé) par **Babel** en un format JavaScript que les navigateurs peuvent interpréter directement. Ce processus permet aux développeurs de profiter des avantages de JSX tout en produisant un code compatible avec les standards des navigateurs. L'étape clé ici est la transpilation. Le code JSX écrit par le développeur est transformé en JavaScript "*pur*" grâce à un outil comme Babel. Ensuite, ce JavaScript est injecté dans la page HTML via des scripts, et les composants React interagissent avec le DOM virtuel pour optimiser la performance et la mise à jour de la page.

Pour le développement mobile, **React Native** permet de réutiliser des concepts de React tout en ciblant des plateformes natives comme iOS et Android. Le code écrit en React Native utilise du JavaScript et des composants spécifiques, mais au lieu d'être transformé en HTML/CSS, il est compilé en code natif. Les composants React Native sont traduits en éléments d'interface utilisateur natifs, comme les boutons, les listes, et autres éléments typiques de chaque plateforme. Le JavaScript de React Native est interprété par un moteur JavaScript qui communique avec le code natif de chaque plateforme (via un pont "bridge"), permettant à l'application d'interagir avec les composants natifs.

Dans le domaine de la réalité augmentée (AR) et de la réalité virtuelle (VR), des Frameworks comme **React 360** (anciennement React VR) permettent de créer des expériences immersives. Le code React est alors converti en WebGL, une API qui permet de générer des graphiques 3D dans le navigateur. Ce code graphique est optimisé pour rendre des environnements VR ou AR. Le JavaScript est interprété par le moteur WebGL du navigateur, qui gère le rendu des éléments 3D et leur interaction avec les périphériques de réalité virtuelle ou augmentée.

CRÉER UN PROJET WEB REACT

Il existe plusieurs outils qui facilitent la configuration d'un projet React pour le Web.

1. Utiliser **Create React App** (CRA)

Create React App est un serveur de développement standard qui simplifie la création d'un projet React avec peu de configuration. Il met en place une structure de base avec tout ce qu'il faut pour commencer à développer directement. Les étapes pour créer un projet React avec CRA sont les suivantes :

1. Installe **Node.js** et **npm** (ou **yarn** si tu préfères).
2. Ouvre un terminal et exécute la commande suivante pour créer ton projet :
   `npx create-react-app nom-du-projet`
   Cela télécharge les dépendances et crée une structure de base pour ton application.
3. Accède à ton projet : `cd nom-du-projet`
4. Lance le serveur de développement : `npm start`

Cela démarre un serveur local et tu peux voir ton application en accédant à `http://localhost:3000` ou `http://127.0.0.1:3000` dans ton navigateur. Create React App s'occupe de toutes les configurations (Webpack, Babel, etc.) en arrière-plan pour te permettre de te concentrer sur le développement.

2. Utiliser **Vite**

Vite est un serveur de développement plus récent, rapide et léger, souvent préféré pour ses temps de démarrage et de construction rapides par rapport à CRA. Il utilise également une configuration par défaut basée sur **ES Modules**. Les étapes pour créer un projet React avec vite sont les suivantes :

- Assure-toi d'avoir **Node.js** et **npm** installés.
- Ouvre un terminal et exécute la commande suivante pour créer ton projet avec Vite :
  `npm create vite@latest nom-du-projet -- --template react`
  Cette commande crée un projet React en utilisant Vite.

- Accède au répertoire du projet : `cd nom-du-projet`
- Installe les dépendances : `npm install`
- Lance le serveur de développement : `npm run dev`

STRUCTURE MINIMALE D'UN PROJET REACT

Une fois le projet créé, on se retrouve avec un dossier qui contient la structure de base d'un projet React. Tu retrouveras généralement les mêmes fichiers dans un projet démarré avec Vite ou avec Create React App (CRA), avec quelques variations mineures :

```
nom-du-projet/
├── node_modules/
├── public/
│   ├── favicon.ico
│   ├── index.html
│   ├── manifest.json
│   └── robots.txt
├── src/
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   └── logo.svg
├── .gitignore
├── package.json
├── package-lock.json
├── README.md
└── yarn.lock  (si tu utilises Yarn au lieu de npm)
```

Chaque fichier et dossier a un rôle spécifique, et je vais te les expliquer un par un :

1. **Dossier** `public/`

   Le dossier `public/` contient les fichiers statiques qui ne sont pas transformés par **Webpack**. Voici les principaux fichiers à l'intérieur de ce dossier :

   - `favicon.ico` : Fichier d'icône qui s'affiche dans l'onglet du navigateur à côté du titre de la page. Tu peux remplacer cette icône par celle de ton projet.
   - `index.html` : C'est le point d'entrée de toute application React dans le navigateur. Ce fichier HTML est servi tel quel à chaque fois qu'un client se connecte à votre site. React injecte ensuite les composants à l'intérieur de l'élément `<div id="root"></div>` qui s'y trouve. Ce fichier contient généralement les métadonnées de la page (balise `<head>`), les liens vers les feuilles de style, et le conteneur où l'application React sera rendue.
   - `manifest.json` : Ce fichier définit la configuration pour les Progressive Web Apps (PWA). Il contient des informations comme le nom de ton application, les icônes, et les paramètres d'affichage pour les appareils mobiles.
   - `robots.txt` : Ce fichier informe les moteurs de recherche (comme Google) des pages à ignorer lorsqu'ils explorent et indexent ton site.

2. **Dossier** `src/`

   Le dossier `src/` est le cœur du projet React, contenant tout le code source de ton application. C'est ici que tu développeras tes composants, tes fichiers CSS, et ta logique métier.

   - `App.js` : Le composant principal de l'application. C'est ici que tu définis la structure et le rendu de l'application. Il s'agit souvent du point de départ pour ajouter des composants ou modifier l'interface utilisateur.
   - `App.css` : Le fichier de styles associé à App.js. Il contient les styles spécifiques à l'application ou à certains composants. Tu peux le modifier ou le remplacer si tu utilises une autre approche pour les styles (comme SCSS ou des styled-components).
   - `App.test.js` : Ce fichier contient des tests unitaires pour le composant `App.js`. *Jest*, le framework de test, l'utilise pour s'assurer que ton composant fonctionne comme attendu. Par défaut, Create React App est livré avec un environnement de test prêt à l'emploi.

- **index.js**: C'est le point d'entrée de l'application JavaScript. Ce fichier rend l'application React dans le DOM du navigateur. Il utilise généralement le code suivant :
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```
Le composant `<App />` est rendu à l'intérieur de l'élément avec l'`id root` dans `index.html`.
- **index.css**: Un fichier CSS global pour l'application. Il est généralement utilisé pour définir les styles de base, comme la mise en page globale, les polices, et les couleurs.
- **logo.svg** : Une icône SVG (Scalable Vector Graphics) utilisée par défaut dans le composant `App.js`. Tu peux la remplacer ou supprimer si tu n'en as pas besoin.

3. **Fichiers à la racine du projet**
   - **.gitignore** : Ce fichier liste les fichiers et dossiers à ignorer par Git, comme `node_modules/` ou les fichiers générés automatiquement. Il est important pour éviter de versionner des fichiers inutiles dans ton dépôt Git.
   - **package.json** : Ce fichier décrit les métadonnées de ton projet et les dépendances nécessaires. Il contient aussi les scripts utiles comme `npm start` pour démarrer l'application, ou `npm test` pour lancer les tests. Voici quelques sections importantes :
     - `"dependencies"` : Liste des packages dont ton projet a besoin (React, ReactDOM, etc.).
     - `"scripts"` : Commandes pour automatiser certaines tâches comme le lancement du serveur de développement, la compilation, etc.
   - **package-lock.json** : Ce fichier assure que les mêmes versions de dépendances sont installées chez tous les développeurs travaillant sur le projet. Il garantit la cohérence des versions de modules.
   - **README.md** : Un fichier de documentation (*markdown*) qui fournit des informations sur ton projet, comme comment démarrer l'application, les commandes disponibles, et une description générale. C'est généralement le premier fichier que les gens consultent lorsqu'ils découvrent ton projet.
4. **Dossier `node_modules/`**

Ce dossier contient toutes les dépendances du projet installées via **npm** ou **yarn**. Il est automatiquement généré lorsque tu exécutes **`npm install`** ou **`yarn install`**. Tu n'as pas besoin de modifier ce dossier directement.

Cette structure de base est celle générée automatiquement par Create React App, mais elle peut être personnalisée au fur et à mesure que ton projet évolue. Tu pourras ajouter des dossiers supplémentaires pour organiser tes composants (`components/`), tes services (`services/`), tes images (`assets/`), ou encore tes contextes (`contexts/`) en fonction des besoins de ton application.

Avec Vite, la structure est assez similaire, mais il y a quelques différences :
1. Le fichier d'entrée de l'application React est souvent nommé **`main.jsx`** au lieu de `index.js`.
2. Vite est plus léger, donc il peut y avoir moins de fichiers et de configurations par défaut.

A chaque fois que tu veux reprendre le développement de ton application, il suffit de te rendre dans le dossier de ton projet et taper la commande **`npm run dev`**.

DÉPLOYER UNE APPLICATION WEB REACT

En réalité, React n'est rien d'autre qu'un moyen qui permet de créer des sites web avec JavaScript plus vite qu'on ne le ferait si on devait se lancer uniquement avec du JavaScript pur et des API Web. Lorsqu'on a fini d'écrire son application React, on convertit toute notre application en html, CSS et JavaScript traditionnel, format adéquat pour la majorité des navigateurs. Cette conversion se passe en quatre étapes :
1. La **transpilation** : Elle consiste à traduire tout notre code React en un JavaScript traditionnel. Pour se faire, on utilise des outils comme Babel.
2. Le **bundling** : Elle consiste à grouper les fichiers JavaScript en un seul ou en fichier suffisamment petits pour ne pas gêner le rendu des pages. On utilise des outils comme WebPack.

3. La **minification** : Elle consiste à enlever les éléments inutiles de notre code source afin d'en réduire la taille le plus possible.
4. L'**optimisation** des assets : les images, le CSS, les vidéos… sont aussi optimisés.

Effectuer ces tâches manuellement est très fastidieux. Voilà pourquoi il existe des outils comme Vite ou Create React App. Il ne s'agit pas simplement de serveurs de développement (un logiciel qui nous permet de visualiser notre application pendant qu'on est encore en train de l'écrire), mais aussi d'outils de *tests* et surtout, de *build*. Ces outils étant eux même écrit en JavaScript, ils ont besoin d'un moteur JavaScript pour fonctionner. Voilà pourquoi vous devez avoir Node.js, qui est un serveur JavaScript, pendant que vous êtes en train de développer votre application React. Une fois en format build, plus besoin, vous avez une structure ordinaire qui ne contient que du HTML, CSS et du Javascript simple.

On tape la commande **`npm run build`** pour convertir un projet React. Cela crée un dossier `build/` qui sera utilisé en production comme notre frontend.

CHAPITRE 1. CRÉER UN ÉLÉMENT

La chose la plus fondamentale qu'on peut créer en React est un *élément*. Il s'agit d'un morceau de code décrivant un élément HTML. On utilise la fonction `React.createElement(type,props,enfants)` pour créer un élément où :

1. ***Type*** : spécifie l'élément que l'on veut créer. Il s'agit d'un `string` qui représente un élément HTML (`'div'`, `'h1'`, etc.) ou d'un *composant* React.
2. ***Propriétés*** : Autrement appelé *props*, il s'agit d'un objet contenant les propriétés (attributs) à appliquer à l'élément qu'on compte créer. Il peut s'agir d'attributs HTML (`id`, `className`, etc.) ou de propriétés propres à React.
3. ***Enfants*** : Après les deux premiers paramètres, tous les paramètres qui suivent s'agissent des ''*enfants*'' de l'élément. Un enfant peut s'agir du contenu de la balise HTML (s'il ne consiste qu'en une chaine de caractères) ou d'un autre élément créé avec `React.createElement()`.

(ex : Voici un exemple d'un élément simple avec aucune propriété et un seul enfant :
`React.createElement('h1', null, "mon titre");`, son équivalent en HTML est `<h1>mon titre</h1>`.
Voici autre exemple d'un élément avec d'autres éléments enfants à l'intérieur :

```
React.createElement('div', {className: 'titre'}, "Liste",
  React.createElement('ul', null,
    React.createElement('li', null, "item 1"),
    React.createElement('li', null, "item 2")
  )
);
```

, son équivalent en HTML est

```
<div class="titre">
  Liste
  <ul>
    <li>item 1</li>
    <li>item 2</li>
  </ul>
</div>
```

).

APPLIQUER DU CSS A DU CODE REACT

Pour lier et appliquer du CSS à un élément React créé avec `React.createElement()`, on peut procéder de deux façons principales : en utilisant des classes CSS ou des styles en ligne.

1. **Utilisation de classes CSS**

   Si vous avez un fichier CSS externe, vous pouvez ajouter des classes CSS à votre élément via la propriété `className` (ex :

   Dans le fichier `App.css` :

   ```
   .mon-style {
     color: blue;
     font-size: 20px;
   }
   ```

Ensuite, dans le fichier **App.js** :

```
import React from 'react';
import './App.css'; //Importation du fichier CSS

const App = () => {
  return React.createElement(
    'div', {className: 'mon-style'}, //Ajout de la classe CSS
    'Hello, world!'
  );
};

export default App;).
```

2. **Utilisation de styles en ligne**

On peut également appliquer des styles en ligne en passant un objet à la propriété style. L'objet doit contenir les propriétés CSS sous forme *camelCase* (ex :
Dans le fichier **App.js** :

```
import React from 'react';

const App = () => {
  return React.createElement(
    'div',
    {style: {color: 'blue', fontSize: '20px'} /*Styles en ligne*/},
    'Hello, world!'
  );
};

export default App;).
```

CHAPITRE 2. LE LANGAGE JSX

Utiliser `React.createElement()` à chaque fois qu'on veut créer un élément est une tâche qui devient dure, au fur et à mesure que l'on doit créer des éléments complexes. Pour pallier à ce problème, React a proposé une syntaxe plus simple : Le JSX. Le JSX (*JavaScript XML*) est une syntaxe utilisée avec React qui permet de mélanger JavaScript et une syntaxe qui ressemble à du HTML pour créer des éléments d'interface utilisateur de manière déclarative. JSX est très proche du HTML, mais il s'exécute en JavaScript, permettant d'ajouter des fonctionnalités dynamiques directement dans l'interface.

Pour créer un élément JSX, on peut simplement écrire des balises HTML-like directement dans le code JavaScript (ex : `const élément = <h1>Bonjour, tout le monde!</h1>;`). Dans cet exemple, nous avons créé un élément `<h1>` avec du texte à l'intérieur. L'élément est ensuite stocké dans une constante `élément` pour être utilisée ultérieurement.

Il est possible d'ajouter des commentaires dans un élément JSX. Cependant, comme le JSX se trouve dans un fichier JavaScript, les commentaires sont légèrement différents de ceux utilisés dans HTML. Ils doivent être encadrés par des accolades pour indiquer qu'il s'agit d'une expression JavaScript (ex :
`{/*Ceci est un commentaire en JSX*/}`
`const élément = <div>{/*Ceci est un autre commentaire en JSX*/}</div>;`).

Une fois un élément JSX créé, il peut être rendu à l'écran à partir de n'importe quel fichier grâce à la fonction `ReactDOM.render()`. Cette fonction prend deux arguments : l'élément JSX à rendre et un conteneur DOM défini dans `index.html`, dans lequel l'élément sera rendu (ex :

```
ReactDOM.render(
  <h1>Bonjour, React!</h1>, document.getElementById('root')
);
```

). Ici, l'élément `<h1>` est rendu dans un élément du DOM de `index.html` ayant l'ID `id="root"`.

Comme pour les éléments HTML, vous pouvez ajouter des propriétés (*props*) à un élément JSX. Les props sont des attributs passés à des composants ou des éléments JSX pour leur fournir des informations

supplémentaires (ex : `const élément = <img src="./image.png" alt="Une image"/>;`). Dans cet exemple, l'élément `<img>` reçoit les props `src` et `alt`.

APPLIQUER DU CSS A UN ELEMENT JSX

On peut ajouter du CSS à un élément JSX en utilisant l'attribut `className` (au lieu de `class` utilisé en HTML), ou bien on peut appliquer des styles en ligne avec l'attribut `style` (ex :
Dans le fichier **`quelconque.css`** :

```
.titre {
  color: blue;
  font-size: 20px;
}
```

Ensuite, dans un fichier **`quelconque.js`** :

```
import './quelconque.css'; //Importation du fichier CSS

const élément = <div className="titre">Mon titre stylé</div>;
```

Avec style en ligne,
`const élément = <h1 style={{color:'blue', fontSize:'20px'}}>Titre stylé</h1>;`). Ici, on injecte un objet JavaScript avec les propriétés CSS écrits en *camelCase*.

INJECTER DES DONNEES DANS LE CONTENU D'UN ELEMENT JSX

JSX permet d'injecter des données dynamiques (comme des `string`, des `number`, des tableaux ou des objets) directement dans le rendu des éléments. Les expressions JavaScript doivent être placées entre des accolades `{}` dans les balises JSX :

- Injecter un `string` (ex : `const élément = <h1>{'Bonjour, ' + name}!</h1>;`).
- Injecter un `number` (ex : `const élément = <p>{2024}</p>;`).
- Injecter un tableau (ex :
```
const nombres = [1, 2, 3, 4, 5];
const élément = (
  <ul>
    {nombres.map((nombre) =>
      (<li key={nombre}>{nombre}</li>))
    }
  </ul>
);
```).
- Injecter un objet (ex :
```
const utilisateur = {firstName: 'Christian', lastName: 'Kaykaykay'};
const élément = <h1>{utilisateur.firstName} {utilisateur.lastName}</h1>;
```).

Dans tous ces exemples, des expressions JavaScript sont insérées directement dans le code JSX en utilisant les accolades.

CHAPITRE 3. LES COMPOSANTS

En HTML, créer un site web consiste à décrire chaque page par un fichier HTML. En React, par contre, on ne décrit pas directement les pages web, mais des petites parties qui en constituent chacune : les composants. Une application en React est un agrégat de composants. Ces derniers permettent de diviser l'interface utilisateur en morceaux indépendants, réutilisables et modulaires. Un composant est une fonction JavaScript qui prend un objet en entrée (appelé `props`) et retourne un élément React décrivant une partie de l'interface utilisateur.

Le composant `App`, défini dans `App.js` et rendu dans `index.html` via `index.js`, est généralement le composant principal d'une application React. Pensez-y comme l'équivalent de la fonction `main()` dans d'autres langages. On y définit toute la logique de notre application. Mais puisque définir la logique dans toute une

application dans une seule fonction peut la rendre longue, on la décompose en d'autres composants au gré de notre désir.
Un composant peut consister à un élément aussi simple qu'un bouton à une page Web entière. Il existe plusieurs moyens de créer un composant mais la manière la plus moderne consiste à définir une fonction qui retourne du JSX (ex :

```
const nomComposant = () => {
  return (<p>Contenu JSX</p>);
};).
```

Pour expliquer de quoi un composant s'agit réellement, considérons un exemple où on veut créer la page suivante :

Si on découpe la page en composants, on peut identifier la page comme le composant `App`. L'en-tête (encerclé en vert) sera défini par un composant `Header`, le corps de la page (en jaune) est défini par le composant `Main` et contient deux composants : `ListeTech` (en mauve) et `Carte` (en bleu). Le pied de page est à son tour défini par le composant `Footer` (en rouge). Créons notre page :

Dans le fichier **`Header.js`** :

```
import {stylesHeader} from './Header.css';

const Header = () => {
  return (
    <header style={stylesHeader}> //Utilisation du fichier CSS
      <div className='mon-header'>
        <h1>Welcome to 30 Days Of React</h1>
        <h2>Getting Started React</h2>
        <h3>JavaScript Library</h3>
        <p>Asabeneh Yetayeh</p>
        <small>Oct 3, 2020</small>
      </div>
    </header>
  );
}

export default Header;
```

Dans le fichier **`Main.js`** :

```
Import {MainCss} from './Main.css';
import asabenehImage from './images/asabeneh.jpg';

//Composant Carte
const Carte = () => (
  <div className='user-card' style={MainCss}>
    <img src={asabenehImage} alt='asabeneh image' />
    <h2>Asabeneh Yetayeh</h2>
  </div>
);

//Composant ListeTech
const ListeTech = () => {
  const techs = ['HTML', 'CSS', 'JavaScript'];
  const techsFormattés = techs.map((tech) => <li key={tech}>{tech}</li>);
  return techsFormattés;
};

//Composant Main
const Main = () => (
  <main>
    <div className='main-wrapper' style={MainCss}>
      <p>Prerequisite to get started react.js:</p>
      <ul>
        <ListeTech /> //Utilisation du composant ListeTech
      </ul>
      <Carte /> //Utilisation du composant Carte
    </div>
  </main>
);

export default Main;
```

Dans le fichier **Footer.js** :

```
Import {FooterCss} from './Footer.css';

const Footer = () => (
  <footer>
    <div className='footer-wrapper' src={FooterCss}>
      <p>Copyright 2020</p>
    </div>
  </footer>
);

export default Footer;
```

Dans le fichier **App.js** :

```
import React from 'react';
import Header from './Header.js';
import Main from './Main.js';
import Footer from './Footer.js';


const App = () => (
  <div className='app'>
    <Header /> //Utilisation du composant Header
    <Main /> //Utilisation du composant Main
    <Footer /> //Utilisation du composant Footer
  </div>
```

```
);

export default App;
```

Dans le fichier **index.js** :

```
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Ce code est une application React simple qui utilise des composants pour créer une structure de page avec un en-tête, un contenu principal et un pied de page. Chaque composant correspond à une partie spécifique de l'interface utilisateur. Voici une explication détaillée :

- Composant `Header` (**`Header.js`**) : Le fichier **`Header.js`** exporte un composant React nommé `Header` qui affiche un en-tête contenant des balises HTML `<h1>`, `<h2>`, `<h3>`, `<p>`, et `<small>`. Le style est importé depuis un fichier CSS nommé **`Header.css`**, et appliqué à l'élément HTML `<header>` via `style={stylesHeader}`.
- Composant `Main` (**`Main.js`**) : Il contient deux sous-composants : `ListeTech` et `Carte`.
    - `Carte` : Affiche une carte d'utilisateur avec une image et un nom. L'image d'Asabeneh est importée depuis le fichier local `./images/asabeneh.jpg`, et le style est appliqué à partir de **`Main.css`**.
    - `ListeTech` : Génère une liste non ordonnée d'éléments (HTML, CSS, JavaScript). Les éléments sont stockés dans un tableau et affichés dynamiquement à l'aide de la méthode `map()`.
    - `Main` : Le composant principal qui inclut les deux composants précédents (`ListeTech` et `Carte`), tout en affichant une phrase introductive. Le style provient de **`Main.css`**.
- Composant `Footer` (**`Footer.js`**) : Le composant `Footer` affiche un pied de page simple contenant un texte de copyright. Le style est appliqué à partir du fichier **`Footer.css`**.
- Composant principal `App` (**`App.js`**) : Le composant `App` est la structure principale de l'application. Il importe les trois composants créés (`Header`, `Main`, `Footer`) et les assemble dans une `<div>` avec la classe app. Ce composant sert de conteneur pour l'ensemble de la page.
- Rendu final (**`index.js`**) : Le fichier `index.js` est le point d'entrée de l'application. Il utilise `ReactDOM.createRoot()` pour créer la racine de l'application et rendre le composant `App` à l'intérieur d'un élément HTML avec l'ID `id="root"`. C'est ici que l'application React démarre.

Ce projet React divise l'interface en plusieurs composants, chacun avec ses propres fonctionnalités et styles. Cela rend le code réutilisable et plus facile à maintenir.

CHAPITRE 4. LES PROPS

Une props (propriété) est un mécanisme fondamental dans React permettant de passer des données d'un composant parent à un composant enfant. C'est un objet immuable qui permet à un composant de recevoir des informations dynamiques. Une props aide à rendre un composant réutilisable et permet de gérer sa communication avec d'autres composants.

Dans un *composant fonctionnel*, les props sont passées en argument à la fonction (ex :

```
function Bienvenue(props){
  return <h1>Bienvenue, {props.nom}!</h1>;
}
```

). Dans cet exemple, `props.nom` est utilisé pour afficher une valeur dynamique dans le composant. Un cas d'utilisation à partir d'un composant parent ressemblerait à `<Bienvenue name="Christian"/>`.

L'objet `props` peut contenir différents types de données (ex :

```
function Info(props){
  return (
    <div>
      <p>Nom : {props.name}</p> {/* String */}
      <p>Âge : {props.age}</p> {/* Number */}
```

```
      <p>Est adulte : {props.isAdult ? "Oui" : "Non"}</p> {/* Boolean */}
      <p>Liste d'amis : {props.friends.join(", ")}</p> {/* Tableau */}
      <p>Détails : {props.details.location}</p> {/* Objet */}
      <button onClick={props.cliqué}>Cliquez-moi</button> {/* Fonction */}
    </div>
  );
}

const details = {location: 'Paris'};
const friends = ['Alice', 'Bob', 'Charlie'];

//utilisation
ReactDOM.render(
  <Info
    name="John"
    age={30}
    isAdult={true}
    friends={friends}
    details={details}
    cliqué={() => alert('Bouton cliqué !')}
  />,
  document.getElementById('root')
);).
```

DESTRUCTURER L'OBJET PROPS

Il est courant de déstructurer `props` pour simplifier l'accès aux propriétés, plutôt que d'écrire `props.nom` à chaque fois (ex :

```
function Bienvenue({nom, age}){
  return (
    <div>
      <h1>Bienvenue, {nom}!</h1>
      <p>Âge : {age}</p>
    </div>
  );
}

//utilisation
ReactDOM.render(
  <Bienvenue nom="Alice" age={25} />,
  document.getElementById('root'));).
```

Dans cet exemple, on extrait directement `nom` et `age` des `props`, ce qui rend le code plus propre et plus lisible.

VALIDER LES PROPRIETES DE PROPS

React offre un mécanisme pour valider les props via `PropTypes`. Cela permet de s'assurer que les props passées à un composant sont du bon type et aident à détecter les erreurs pendant le développement. Pour utiliser `PropTypes`, vous devez les importer depuis le package `'prop-types'` : **`npm install prop-types`** (ex :

```
import PropTypes from 'prop-types';

function Profil({name, age, isAdmin}){
  return (
    <div>
      <h1>{name}</h1>
      <p>Âge : {age}</p>
      <p>Admin : {isAdmin ? "Oui" : "Non"}</p>
    </div>
```

```
  );
}
//On affecte un objet à Profil
Profil.propTypes = {
  name: PropTypes.string.isRequired,  //name doit être une chaîne obligatoire
  age: PropTypes.number,              //age doit être un nombre
  isAdmin: PropTypes.bool,            //isAdmin doit être un booléen
};

//Rendre le composant avec des props correctes
ReactDOM.render(
  <Profil name="Alice" age={30} isAdmin={true} />,
  document.getElementById('root')
);
```

). Si les props ne respectent pas les types définis dans `PropTypes`, une erreur s'affichera dans la console pendant le développement (Voir https://fr.legacy.reactjs.org/docs/typechecking-with-proptypes.html).

DEFINIR UNE VALEUR PAR DEFAUT POUR LES PROPS

`defaultProps` permet de définir des valeurs par défaut pour les props lorsqu'elles ne sont pas fournies par le composant parent. Cela garantit que le composant dispose de toutes les données dont il a besoin, même si certaines props sont manquantes (ex :

```
function Profil({nom, age, isAdmin}){
  return (
    <div>
      <h1>{nom}</h1>
      <p>Âge : {age}</p>
      <p>Admin : {isAdmin ? "Oui" : "Non"}</p>
    </div>
  );
}

Profil.defaultProps = {
  age: 18,                //Si l'age n'est pas fourni, il sera 18 par défaut
  isAdmin: false,         //Si isAdmin n'est pas fourni, il sera false par défaut
};

ReactDOM.render(
  <Profil nom="Alice" />,
  document.getElementById('root')
);
```

). Dans cet exemple, même si la prop `age` et `isAdmin` ne sont pas passées, le composant utilise les valeurs par défaut spécifiées dans `defaultProps`.

CHAPITRE 5 : MAPPER DES TABLEAUX EN REACT

En React, il est fréquent de travailler avec des tableaux (arrays) pour afficher des listes d'éléments dynamiques. La méthode JavaScript `map()` est souvent utilisée pour itérer sur ces tableaux et créer un élément JSX pour chaque élément du tableau. Lorsqu'on rend une liste d'éléments en React, chaque élément doit avoir une **clé** (`key`) unique pour que React puisse identifier quels éléments ont changé, ont été ajoutés ou supprimés.

Voyons comment on peut utiliser `map()` pour rendre des listes d'éléments à partir de tableaux de nombres, de tableaux imbriqués, ou d'objets.

MAPPER UN TABLEAU DE NOMBRES

Si vous avez un tableau de nombres, vous pouvez utiliser `map()` pour créer un élément pour chaque nombre (ex :

```
const nombres = [1, 2, 3, 4, 5];
function ListeNombres(){
  return (
    <ul>
      {nombres.map((nombre) => (
        <li key={nombre.toString()}>{nombre}</li>
      ))}
    </ul>
  );
}
```

). Dans cet exemple, le tableau nombres contient des entiers. On utilise la méthode `map()` pour créer un élément `<li>` pour chaque nombre dans le tableau. La clé (`key`) est importante : ici, la valeur du nombre est utilisée comme clé unique, car chaque nombre est unique dans le tableau.

MAPPER UN TABLEAU MULTIDIMENSIONNEL

Si vous avez un tableau de tableaux, vous pouvez également utiliser `map()` pour parcourir ces sous-tableaux et créer des éléments JSX imbriqués.

```
const tableaux = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];
function ListeTableaux(){
  return (
    <div>
      {tableaux.map((tableau, index) => (
        <ul key={index}>
          {tableau.map((item) => (
            <li key={item.toString()}>{item}</li>
          ))}
        </ul>
      ))}
    </div>
  );
}
```

). Dans cet exemple, le tableau tableaux contient des sous-tableaux. Le premier `map()` itère sur chaque sous-tableau pour créer une liste `<ul>`. Le deuxième `map()` à l'intérieur du premier crée un `<li>` pour chaque élément dans le sous-tableau. La clé pour chaque sous-liste est `index` (l'indice du tableau), car chaque sous-tableau est unique. Pour chaque élément du sous-tableau, on utilise la valeur de l'élément comme clé.

MAPPER UN TABLEAU D'OBJETS

Il est fréquent d'avoir un tableau d'objets en React. Vous pouvez utiliser `map()` pour accéder aux propriétés des objets et les rendre dans le JSX (ex :

```
const utilisateurs = [
  {id: 1, nom: 'Alice', age: 25},
  {id: 2, nom: 'Bob', age: 30},
];
function ListeUtilisateurs(){
  return (
    <ul>
      {utilisateurs.map((utilisateur) => (
        <li key={utilisateur.id}> {utilisateur.nom} - {utilisateur.age} ans </li>
      ))}
    </ul>
  );
}
```

). Dans cet exemple, le tableau utilisateurs contient des objets avec des propriétés comme `id`, `nom`, et `age`. On utilise `map()` pour créer une liste `<li>` pour chaque utilisateur, en affichant son nom et son âge. La clé

est basée sur l'identifiant unique (`id`) de chaque objet, ce qui est une pratique recommandée lorsque les objets ont des identifiants uniques.

Les clés (`key`) sont cruciales pour les performances et le bon fonctionnement des listes en React. Elles permettent à React d'identifier les éléments de la liste et d'appliquer efficacement les modifications lors du ré-rendu des composants. Voici pourquoi les clés sont importantes :

1. Si les éléments d'une liste changent (ajoutés, supprimés, réordonnés), React utilise les clés pour déterminer quels éléments doivent être mis à jour, réutilisés ou supprimés.
2. Une clé doit être unique parmi les éléments frères d'une liste, mais elle n'a pas besoin d'être unique dans l'ensemble de l'application.
3. Une clé incorrecte ou manquante peut entraîner des comportements inattendus lors de la mise à jour des listes, notamment des éléments qui ne sont pas mis à jour correctement.

Voici quelques bonnes pratiques à adopter avec les clés :

1. Utilisez des identifiants uniques présents dans vos données (comme `id`) lorsque c'est possible.
2. Évitez d'utiliser l'index du tableau (`index`) comme clé si l'ordre de la liste est susceptible de changer, car cela peut entraîner des ré-rendus inefficaces.
3. Ne réutilisez pas des clés à travers différentes listes.

CHAPITRE 6 : LES COMPOSANTS DE CLASSE EN REACT

Les **composants de classe** (ou *class components*) étaient l'une des premières façons de créer des composants dans React avant l'introduction des **hooks** en 2018. Même si les composants fonctionnels sont maintenant largement utilisés grâce aux hooks, il est toujours important de comprendre les composants de classe, car ils sont encore présents dans de nombreuses bases de code existantes et possèdent des particularités propres, notamment la gestion de *l'état* (**state**) et des méthodes de cycle de vie.

CREATION D'UN COMPOSANT DE CLASSE

Un composant de classe est une classe JavaScript qui étend `React.Component`. Cette classe doit implémenter une méthode obligatoire appelée `render()`, qui retourne du JSX (ou `null`). Contrairement aux composants fonctionnels, les composants de classe peuvent gérer leur propre état avec `this.state` et réagir à des événements du cycle de vie du composant (ex :

```
class MonComposant extends React.Component {
  render(){
    return <h1>Bonjour, {this.props.nom}!</h1>;
  }
}
```

). Dans cet exemple, `MonComposant` est une classe qui étend `React.Component`. La méthode `render()` retourne un élément JSX qui affiche par exemple, "Bonjour, Alice !". Le composant reçoit une prop `nom`, accessible via `this.props`. L'utilisation est la même qu'avec les composants fonctionnels.

GESTION DE L'ETAT (STATE) DANS LES COMPOSANTS DE CLASSE

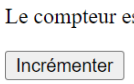
L'une des caractéristiques distinctives des composants de classe est la possibilité de gérer un état interne avec `this.state`. Cet état est un objet qui peut être utilisé pour stocker des informations dynamiques propres au composant. Lorsqu'un état est mis à jour via `this.setState()`, le composant est automatiquement ré-rendu (ex :

```
class Compteur extends React.Component {
  constructor (props){
    super(props);
    this.state = {count: 0};
  }
  incrementer = () => {this.setState((prevState) => ({count: prevState.count + 1}));};

  render(){
    return (
      <div>
        <p>Le compteur est à : {this.state.count}</p>
```

```
        <button onClick={this.incrementer}>Incrémenter</button>
      </div>
    );
  }
}).
```

Le composant `Compteur` initialise l'état dans le constructeur avec `this.state={count: 0}`. La méthode `incrementer()` met à jour l'état en appelant `this.setState()`, ce qui entraîne un ré-rendu du composant. Dans la méthode `render()`, on affiche la valeur actuelle du compteur avec `this.state.count` et un bouton pour l'incrémenter. A chaque fois que vous cliquez sur le bouton « Incrémenter », le compteur sera incrémenté et ce, grâce à l'état interne qu'on a mis.



METHODES DE CYCLE DE VIE DANS LES COMPOSANTS DE CLASSE

Les composants de classe ont plusieurs *méthodes de cycle de vie* qui permettent de réagir à des événements spécifiques dans la vie d'un composant, comme sa création, son montage, sa mise à jour, ou sa destruction. Voici quelques-unes des méthodes de cycle de vie les plus utilisées :

1. `componentDidMount()` : appelée après que le composant est monté dans le DOM.
2. `componentDidUpdate()` : appelée après une mise à jour (par exemple, lorsqu'une prop ou un état a changé).
3. `componentWillUnmount()` : appelée juste avant que le composant soit démonté du DOM et détruit.

(ex :

```
class CompteurAvecCycleDeVie extends React.Component {
  constructor(props){
    super(props);
    this.state = {count: 0};
  }
  componentDidMount(){
    console.log("Le composant est monté !");
  }
  componentDidUpdate(prevProps, prevState){
    if (prevState.count !== this.state.count){
      console.log("Le compteur a été mis à jour !");
    }
  }
  componentWillUnmount(){
    console.log("Le composant va être démonté.");
  }
  incrementer = () => {
    this.setState({count: this.state.count + 1});
  };

  render(){
    return (
      <div>
        <p>Le compteur est à : {this.state.count}</p>
        <button onClick={this.incrementer}>Incrémenter</button>
      </div>
    );
  }
}).
```

COMPOSANTS AVEC OU SANS ETAT (STATEFUL VS STATELESS)

Composants avec état (*stateful components*) : Ces composants possèdent et gèrent leur propre état interne, comme vu précédement. Ils peuvent rendre dynamiquement en fonction des changements d'état.

Composants sans état (*stateless components*) : Ces composants ne gèrent pas d'état interne. Ils sont généralement utilisés pour afficher du contenu basé uniquement sur les props. Avant les hooks, les composants fonctionnels étaient souvent considérés comme des composants sans état, bien qu'ils puissent maintenant gérer l'état avec `useState()`.

Composants de présentation (*presentation components*) : Ces composants se concentrent sur l'affichage et ne gèrent généralement pas l'état. Ils reçoivent toutes leurs données via des props et sont responsables de l'UI. Ce sont des composants "dumb" car ils ne contiennent pas de logique complexe.

Composants conteneurs (*container components*) : Ces composants sont responsables de la logique d'affaires et de la gestion de l'état. Ils sont souvent appelés composants intelligents (*smart components*) car ils contiennent des interactions complexes avec l'état, les actions, et parfois les API.

CHAPITRE 7 : LES ÉTATS (STATE) EN REACT

En React, l'état (ou state) est une fonctionnalité essentielle qui permet de gérer des données dynamiques dans un composant. Contrairement aux props, qui sont immuables et passées d'un composant parent à un composant enfant, l'état est propre à chaque composant et peut être modifié de manière interne pour refléter des changements dans l'interface utilisateur.

L'état est un objet JavaScript qui représente une partie des données qu'un composant contrôle et qui peut changer au cours de la vie du composant. Lorsqu'un état change, React ré-rend automatiquement le composant pour refléter ce changement. Cela permet de rendre l'interface utilisateur interactive et réactive aux actions de l'utilisateur. Par exemple, un état pourrait stocker des informations comme le contenu d'un champ de saisie, le nombre de clics sur un bouton, si un utilisateur est connecté ou non, etc.

DEFINIR L'ETAT DANS LES COMPOSANTS

Il existe **deux** types de composants principaux en React : les composants de classe et les composants fonctionnels. Ils gèrent tous deux l'état de manière différente.

1. L'état dans les composants de classe : dans un composant de classe, l'état est initialisé dans le constructeur de la classe avec `this.state`, et les mises à jour de l'état se font en utilisant la méthode `this.setState()` (ex :

```
class Compteur extends React.Component {
  constructor(props){
    super(props);
    this.state = {count: 0}; //Initialisation de l'état
  }
  //Méthode pour mettre à jour l'état
  incrementer = () => {
    this.setState({count: this.state.count + 1});
  };
  render(){
    return (
      <div>
        <p>Le compteur est à : {this.state.count}</p>
        <button onClick={this.incrementer}>Incrémenter</button>
      </div>
    );
  }
```

}). L'état initial (`count`) est défini à `0` dans le constructeur. La méthode `incrementer()` utilise `this.setState()` pour modifier l'état et ré-rendre le composant.

2. L'état dans les composants fonctionnels avec les **hooks** : Avant l'introduction des hooks dans React 16.8, seuls les composants de classe pouvaient gérer un état. Les hooks ont introduit la possibilité pour les composants fonctionnels de gérer l'état grâce à la fonction `useState()` (ex :

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

```
function Compteur(){
  const [count, setCount] = useState(0); //Déclaration d'un état avec useState
  return (
    <div>
      <p>Le compteur est à : {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrémenter</button>
    </div>
  );
}
```

). L'état `count` est initialisé à `0` grâce à `useState(0)`. La fonction `setCount()` permet de mettre à jour l'état et de ré-rendre le composant.

GERER L'ETAT COMPLEXE

Si l'état devient plus complexe, vous pouvez gérer plusieurs morceaux d'état avec des objets ou des structures imbriquées (ex :
Dans les composants de classe :

```
class ProfilUtilisateur extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      nom: 'Alice',
      age: 25,
      ville: 'Paris'
    };
  }
  updateAge = () => {
    this.setState({age: this.state.age + 1});
  };
  render(){
    return (
      <div>
        <p>Nom : {this.state.nom}</p>
        <p>Âge : {this.state.age}</p>
        <p>Ville : {this.state.ville}</p>
        <button onClick={this.updateAge}>Vieillir d'un an</button>
      </div>
    );
  }
}
```

Dans les composants fonctionnels avec `useState()` :

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

function ProfilUtilisateur(){
  const [profil, setProfil] = useState({
    nom: 'Alice',
    age: 25,
    ville: 'Paris'
  });
  const updateAge = () => {
    setProfil((prevProfil) => ({
      ...prevProfil,
      age: prevProfil.age + 1,
    }));
  };
  return (
    <div>
      <p>Nom : {profil.nom}</p>
      <p>Âge : {profil.age}</p>
```

```
      <p>Ville : {profil.ville}</p>
      <button onClick={updateAge}>Vieillir d'un an</button>
    </div>
  );
}).
```

LIFTING STATE UP (REMONTER L'ETAT VERS UN COMPOSANT PARENT)

Dans une application, il est souvent nécessaire de partager des informations entre plusieurs composants. Une des façons de le faire est de remonter l'état dans un composant parent (le créer là bas), puis de le passer aux composants enfants sous forme de props (ex :

```
function Parent(){
  const [message, setMessage] = useState('Hello from Parent!');

  return (
    <div>
      <Enfant message={message} />
    </div>
  );
}

function Enfant({message}){
  return <p>{message}</p>;
}).
```

Dans cet exemple, le composant `Parent` gère l'état `message` et le transmet au composant `Enfant` via une prop.

CHAPITRE 8 : LE RENDU CONDITIONNEL EN REACT

Le rendu conditionnel en React consiste à afficher certains éléments d'interface en fonction de conditions spécifiques. Cela est essentiel pour créer des interfaces dynamiques et réactives. En React, il existe plusieurs manières de réaliser un rendu conditionnel, notamment en utilisant des instructions if-else, l'opérateur ternaire et l'opérateur logique `&&`. Je vais t'expliquer chacun de ces mécanismes en détail.

UTILISATION DE IF-ELSE

Une des manières les plus classiques de réaliser un rendu conditionnel est d'utiliser une structure if-else dans la fonction de rendu de votre composant. Cette approche est simple et lisible, mais ne peut pas être directement utilisée dans le code JSX, car le JSX ne supporte pas directement les instructions conditionnelles (ex :

```
function Exemple(props){
  if(props.estConnecté){return <h1>Bienvenue !</h1>;}
  else {return <h1>Veuillez-vous connecter.</h1>;}
}).
```

Dans cet exemple, en fonction de la valeur de `props.estConnecté`, soit le message "Bienvenue !" est affiché si l'utilisateur est connecté, soit le message "Veuillez-vous connecter." est affiché sinon.

UTILISATION DE L'OPERATEUR TERNAIRE

L'opérateur ternaire est un raccourci efficace pour écrire des conditions directement dans le JSX. Il fonctionne en testant une condition : si elle est vraie, il renvoie un élément, sinon il en renvoie un autre. C'est une approche plus concise que if-else (ex :

```
function Exemple(props){
  return (
    <div>
      {props.estConnecté ? <h1>Bienvenue !</h1> : <h1>Veuillez-vous connecter.</h1>}
    </div>
  );
}).
```

Dans cet exemple, l'opérateur ternaire vérifie la valeur de `props.estConnecté`. Si elle est vraie, le message de bienvenue est affiché, sinon le message demandant de se connecter s'affiche.

UTILISATION DE L'OPERATEUR LOGIQUE &&

En JavaScript, l'opérateur logique `&&` renvoie la deuxième partie de l'expression uniquement si la première partie est vraie. Cet opérateur est souvent utilisé en React pour rendre un élément uniquement si une condition est remplie (ex :

```
function Exemple(props){
  return (<div>{props.estConnecté && <h1>Bienvenue !</h1>}</div>);
}
```

). Dans cet exemple, si `props.estConnecté` est `true`, le message "Bienvenue !" est affiché. Sinon, rien ne sera rendu à cet endroit.

COMBINAISON DES OPERATEURS

Il est également possible de combiner ces différentes méthodes pour gérer des conditions plus complexes. Par exemple, on peut combiner un `&&` avec un opérateur ternaire pour gérer plusieurs états conditionnels dans un seul bloc JSX (ex :

```
function Exemple(props){
  return (
    <div>
      {props.iestConnecté ? (
        <h1>Bienvenue !</h1>
      ) : (
        props.afficherBtnConn && <button>Connexion</button>
      )}
    </div>
  );
}
```

). Dans cet exemple, si l'utilisateur est connecté, le message de bienvenue est affiché. Sinon, si la condition `props.afficherBtnConn` est vraie, un bouton de connexion est affiché.

CHAPITRE 9 : LES EVENEMENTS EN REACT

En React, les événements fonctionnent de manière très similaire à ceux du DOM dans le navigateur, mais avec quelques différences importantes et des fonctionnalités supplémentaires. Les événements en React sont des interactions déclenchées par l'utilisateur ou d'autres actions, comme des clics de bouton, des mouvements de souris, des changements dans un champ de formulaire, etc. React offre un modèle d'événements synthétiques, appelé **Synthetic Events**, pour gérer les événements de manière uniforme sur tous les navigateurs.

- Les événements en React sont nommés en **camelCase** (ex : `onClick` au lieu de `onclick` en HTML).
- Les événements React utilisent des fonctions au lieu de chaînes de caractères (ex :
  Dans React, on écrit : `<button onClick={handleClick}>Cliquez ici</button>`
  En HTML, cela ressemblerait à : `<button onclick="handleClick()">Cliquez ici</button>`).

Voici quelques événements les plus fréquemment utilisés en React :

- **Souris :**
  - `onClick` : Déclenché lorsqu'un élément est cliqué.
  - `onDoubleClick` : Déclenché lors d'un double clic.
  - `onMouseEnter` et `onMouseLeave` : Détecte lorsque la souris entre ou quitte un élément.
- **Clavier :**
  - `onKeyDown` : Déclenché lorsqu'une touche est enfoncée.
  - `onKeyUp` : Déclenché lorsqu'une touche est relâchée.
  - `onKeyPress` : Déclenché lorsqu'une touche est pressée et qu'elle produit une valeur (par exemple, une lettre ou un chiffre).
- **Formulaires :**
  - `onChange` : Déclenché lorsqu'une valeur de formulaire change, comme dans un champ texte.

- `onSubmit` : Déclenché lorsqu'un formulaire est soumis.
- `onFocus` et `onBlur` : Détecte lorsque les champs de formulaire gagnent ou perdent le focus.

Pour une liste exhaustive, voir https://fr.legacy.reactjs.org/docs/events.html (ex :

```
function App(){
  function handleClick(){
    alert('Bouton cliqué !');
  }
  return (<button onClick={handleClick}>Cliquez ici</button>);
}).
```

PASSER DES PARAMETRES A UN EVENT HANDLER

Il est parfois nécessaire de passer des paramètres supplémentaires à une fonction d'événement. Pour ce faire, vous pouvez utiliser une fonction fléchée (ex :

```
function App(){
  function handleClick(param){
    alert('Vous avez cliqué sur ' + param);
  }
  return (<button onClick={() => handleClick('le bouton')}>Cliquez ici</button>);
}).
```

React gère un objet d'événement synthétique appelé `SyntheticEvent`. Cet objet est une abstraction qui combine les événements du navigateur natif tout en les standardisant. En voici les propriétés :

- `target` : Référence à l'élément ayant déclenché l'événement.
- `type` : Le type d'événement (par exemple, `'click'`, `'keydown'`).
- `preventDefault()` : Empêche le comportement par défaut de l'événement (par exemple, empêcher un formulaire de se soumettre).
- `stopPropagation()` : Empêche la propagation de l'événement aux autres éléments.

(ex :

```
function Formulaire(){
  function handleSubmit(event){
    event.preventDefault(); //Empêche l'actualisation de la page lors de la soumission
    console.log('Formulaire soumis');
  }

  return (
    <form onSubmit={handleSubmit}> //appelle handlesubmit()
      <button type="submit">Soumettre</button>
    </form>
  );
}).
```

Dans cet exemple, lorsque le formulaire est soumis, la méthode `preventDefault()` empêche le comportement par défaut (l'actualisation de la page). Vous voyez qu'on a défini la fonction `handleSubmit()` avec un paramètre `event`, mais on l'a invoqué sans. En JavaScript, le premier argument passé à un *event handler* est l'objet lié à cet évènement, et JavaScript le passe automatiquement à la fonction.

EVENEMENTS IMBRIQUES

Dans certains cas, vous pouvez avoir des événements imbriqués (comme des boutons à l'intérieur de divs). React suit la phase de **propagation** (ou **bubbling**) des événements, où un événement se propage du plus profond des éléments imbriqués vers l'extérieur.
Si vous voulez empêcher un événement de se propager aux éléments parents, vous pouvez utiliser la méthode `stopPropagation()` (ex :

```
function App(){
  function handleClickDiv(){
    console.log('Div cliquée');
  }
```

```
  function handleClickButton(event){
    event.stopPropagation(); //Empêche la propagation de l'événement au div parent
    console.log('Bouton cliqué');
  }
  return (
    <div onClick={handleClickDiv}>
      <button onClick={handleClickButton}>Cliquez ici</button>
    </div>
  );
```

}). Dans cet exemple, lorsque le bouton est cliqué, la propagation est stoppée grâce à `stopPropagation()`, donc la fonction de clic sur la div parent n'est pas exécutée.

EVENEMENTS ET EXECUTION ASYNCHRONE

Les événements en React peuvent aussi interagir avec des actions asynchrones comme des appels API. Par exemple, après avoir cliqué sur un bouton, vous pouvez déclencher une requête pour récupérer des données depuis un serveur (ex :

```
function App(){
  async function handleClick(){
    const reponse = await fetch('https://api.exemple.com/data');
    const data = await reponse.json();
    console.log(data);
  }
  return (
    <button onClick={handleClick}>Récupérer les données</button>
  );
}
```

). Dans cet exemple, un clic sur le bouton déclenche une requête asynchrone pour récupérer des données depuis une API.

CHAPITRE 10 : LES FORMULAIRES EN REACT

Les formulaires en React sont un aspect important des interfaces utilisateurs, permettant de capturer et de gérer les données utilisateur. La gestion des formulaires dans React diffère des formulaires classiques en HTML, car React se base sur le **concept d'état** (state) et de **composants contrôlés** pour suivre et mettre à jour les valeurs des champs du formulaire.

COMPOSANTS CONTROLES

Dans un composant contrôlé, l'état du formulaire est entièrement géré par React via le state. Chaque champ du formulaire (comme les `<input>`, `<textarea>`, et `<select>`) a sa propre valeur qui est gérée par le state du composant. Chaque fois que l'utilisateur saisit quelque chose dans le champ, l'état est mis à jour, et React ré-rend le champ avec la nouvelle valeur (ex :

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

function FormulaireControllé(){
  const [inputValue, setInputValue] = useState('');

  function handleChange(event){
    setInputValue(event.target.value);
  }

  function handleSubmit(event){
    event.preventDefault();
    alert(`Le texte soumis est : ${inputValue}`);
  }
  return (
    <form onSubmit={handleSubmit}>
```

```
      <label>
        Saisir un texte :
        <input type="text" value={inputValue} onChange={handleChange} />
      </label>
      <button type="submit">Envoyer</button>
    </form>
  );
```

}). Le champ `<input>` est contrôlé, car sa valeur est gérée par l'état (`inputValue`). `onChange` est utilisé pour mettre à jour l'état à chaque modification de l'utilisateur. Lors de la soumission du formulaire, la fonction `handleSubmit()` est appelée, où nous utilisons `preventDefault()` pour éviter l'actualisation de la page.

COMPOSANTS NON CONTROLES

Les composants non contrôlés sont plus proches du comportement natif des éléments HTML. Ici, la valeur des champs du formulaire n'est pas gérée directement par React via le state, mais vous accédez aux valeurs des champs uniquement via les références (`ref`) (ex :

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

function FormulaireNonControllé(){
  const inputRef = useRef(null);

  function handleSubmit(event){
    event.preventDefault();
    alert(`Le texte soumis est : ${inputRef.current.value}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Saisir un texte :
        <input type="text" ref={inputRef} />
      </label>
      <button type="submit">Envoyer</button>
    </form>
  );
```

}). La valeur de l'input n'est pas gérée via l'état de React. On utilise une référence (`ref`) pour accéder directement à la valeur de l'élément DOM. L'utilisation des références est utile pour des cas où vous ne souhaitez pas ré-rendre le composant à chaque modification.

Avantages des composants non contrôlés :

- **Moins de gestion d'état** : Utile pour des cas simples où vous n'avez pas besoin de gérer ou de manipuler la valeur de l'entrée en temps réel.
- **Accès direct au DOM** : Permet d'accéder directement à un élément DOM, utile dans des scénarios spécifiques comme des intégrations avec des bibliothèques non React.

GERER PLUSIEURS CHAMPS DANS UN FORMULAIRE EST FACILE EN REACT.

Pour gérer plusieurs champs d'un formulaire, on peut stocker tous les champs dans un seul objet d'état et utiliser le nom de chaque champ comme clé (ex :

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

function FormulaireMultichamps(){
  const [formData, setFormData] = useState({nom: '', mail: ''});

  function handleChange(event){
    const {name, value} = event.target;
    setFormData({...formData, [name]: value});
  }
```

```
function handleSubmit(event){
  event.preventDefault();
  alert(`Nom: ${formData.nom}, Email: ${formData.mail}`);
}

return (
  <form onSubmit={handleSubmit}>
    <label>
      Nom :
      <input type="text" name="nom" value={formData.nom} onChange={handleChange} />
    </label>
    <br />
    <label>
      Email :
      <input type="email" name="mail" value={formData.mail} onChange={handleChange} />
    </label>
    <br />
    <button type="submit">Envoyer</button>
  </form>
);
```

}). Chaque champ du formulaire est lié à une clé dans l'objet `formData`. `handleChange()` met à jour le champ correspondant dans `formData` en utilisant la propriété `name` de l'élément.

AUTRES ELEMENTS DE FORMULAIRE

Certains éléments de formulaire comme les boutons radio, les cases à cocher et les listes déroulantes (select) nécessitent une attention particulière :

1. **Cases à cocher (Checkbox) :**
   Les cases à cocher utilisent la propriété `checked` plutôt que `value` pour indiquer leur état (coché ou non) (ex :

```
function CheckboxForm(){
  const [estChecké, setEstChecké] = useState(false);

  function handleChange(event){
    setEstChecké(event.target.checked);
  }

  return (
    <form>
      <label>
        Accepter les conditions :
        <input type="checkbox" checked={estChecké} onChange={handleChange} />
      </label>
    </form>
  );
}).
```

2. **Boutons radio (Radio Buttons) :**
   Les boutons radio sont souvent utilisés en groupes, et seul un bouton dans le groupe peut être sélectionné à la fois (ex :

```
function RadioFormulaire(){
  const [optionSelectionnée, setOptionSelectionnée] = useState('option1');

  function handleChange(event){
    setOptionSelectionnée(event.target.value);
  }
  return (
    <form>
```

```
          <label>
            Option 1 :
            <input type="radio" value="option1"
              checked={optionSelectionnée==='option1'} onChange={handleChange} />
          </label>
          <label>
            Option 2 :
            <input type="radio" value="option2"
              checked={optionSelectionnée==='option2'} onChange={handleChange} />
          </label>
        </form>
      );
    }).
```

3. **Listes déroulantes (`<select>`) :**
   Les listes déroulantes utilisent value pour indiquer l'option sélectionnée (ex :

```
function FormulaireSelect(){
  const [valeurSelectionnée, setValeurSelectionnée] = useState('option1');

  function handleChange(event){
    setValeurSelectionnée(event.target.value);
  }

  return (
    <form>
      <label>
        Choisir une option :
        <select value={valeurSelectionnée} onChange={handleChange}>
          <option value="option1">Option 1</option>
          <option value="option2">Option 2</option>
        </select>
      </label>
    </form>
  );
}).
```

VALIDATION DES FORMULAIRES EN REACT

La validation des formulaires peut être effectuée de manière simple en vérifiant les valeurs des champs avant la soumission. Vous pouvez également ajouter des messages d'erreur dynamiques en fonction des conditions de validation (ex :

```
function FormulaireValidable(){
  const [email, setEmail] = useState('');
  const [erreur, setErreur] = useState('');

  function handleChange(event){
    setEmail(event.target.value);
  }

  function handleSubmit(event){
    event.preventDefault();
    if (!email.includes('@')){
      setErreur('Adresse email invalide');
    } else {
      setErreur('');
      alert('Formulaire soumis avec succès');
    }
  }
```

```
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Email :
        <input type="email" value={email} onChange={handleChange} />
      </label>
      <button type="submit">Envoyer</button>
      {erreur && <p style={{ color: 'red' }}>{erreur}</p>}
    </form>
  );
}).
```

CHAPITRE 11 : LE CYCLE DE VIE D'UN COMPOSANT

La **durée de vie d'un composant** en React, également appelée **cycle de vie d'un composant** (component lifecycle), fait référence aux différentes étapes par lesquelles un composant passe depuis sa création jusqu'à sa destruction. Ces étapes sont particulièrement importantes dans les composants basés sur les **classes**, mais les **hooks** dans les composants fonctionnels permettent aussi de gérer le cycle de vie.
Les composants React passent généralement par trois phases principales : *Montage* (Mounting), *Mise à jour* (Updating), *Démontage* (Unmounting).

GESTION DU CYCLE DE VIE DES COMPOSANTS DE CLASSE

Les composants de classe React comportent des méthodes spécifiques qu'on doit créer, qui sont appelées à différents moments du cycle de vie du composant.

1. **Montage (Mounting)** : C'est la phase où un composant est créé et inséré dans le DOM pour la première fois. Les méthodes clés sont :
   - `constructor()` : Méthode appelée avant que le composant ne soit monté. C'est ici que l'état initial (state) est défini et que les props peuvent être initialisés.
   - `static getDerivedStateFromProps()` : Utilisée rarement, cette méthode permet de mettre à jour l'état interne en fonction des changements de props avant le montage ou la mise à jour. Elle remplace les méthodes obsolètes comme `componentWillMount()`.
   - `render()` : Méthode obligatoire, qui retourne les éléments React à afficher dans l'interface utilisateur.
   - `componentDidMount()` : Méthode appelée juste après que le composant a été monté dans le DOM. C'est un endroit idéal pour effectuer des opérations nécessitant l'accès au DOM, comme des requêtes réseau, l'ajout de listeners ou la configuration de timers.

   (ex :
   ```
   class MonComposant extends React.Component {
     constructor(props){
       super(props);
       this.state = {count: 0};
     }
     componentDidMount(){
       //Par exemple, une requête API ou un timer
       console.log("Composant monté !");
     }
     render(){
       return <div>{this.state.count}</div>;
     }
   }).
   ```
2. **Mise à jour (Updating)** : Cette phase est déclenchée lorsque les props ou l'état (state) d'un composant changent, entraînant un nouveau rendu (re-render). Plusieurs méthodes sont utilisées dans cette phase :
   - `static getDerivedStateFromProps()` : Appelée à chaque mise à jour de props, comme pendant la phase de montage. Elle permet de synchroniser l'état avec les props.

- `shouldComponentUpdate()` : Utilisée pour contrôler si le composant doit être re-rendu ou non. Par défaut, React ré-renderise un composant à chaque changement de props ou d'état, mais cette méthode peut être utilisée pour optimiser les performances en évitant des re-render inutiles.
- `render()` : Le composant est ré-rendu en fonction des nouvelles props ou du nouvel état.
- `getSnapshotBeforeUpdate()` : Appelée juste avant la mise à jour réelle du DOM. Cette méthode est utilisée pour capturer des informations du DOM (comme la position de défilement) avant qu'il ne soit modifié.
- `componentDidUpdate()` : Appelée après chaque mise à jour, c'est l'endroit idéal pour effectuer des actions qui dépendent du fait que le DOM ait été mis à jour, comme déclencher de nouvelles requêtes réseau si des props spécifiques ont changé.

(ex :
```
class MonComposant extends React.Component {
  componentDidUpdate(prevProps, prevState){
    if (this.state.count !== prevState.count){
      console.log("Le composant a été mis à jour !");
    }
  }
  render(){
    return <div>{this.state.count}</div>;
  }
}).
```

3. **Démontage (Unmounting)** : Cette phase est déclenchée lorsque le composant est retiré du DOM. C'est une étape importante pour éviter les fuites de mémoire, en particulier lorsque vous avez des écouteurs d'événements ou des timers à nettoyer.
   - `componentWillUnmount()` : Méthode appelée juste avant que le composant ne soit démonté et retiré du DOM. Vous pouvez y effectuer des opérations de nettoyage, comme retirer des écouteurs d'événements ou stopper des timers.

(ex :
```
class MonComposant extends React.Component {
  componentWillUnmount(){
    console.log("Le composant va être démonté");
    //Effectuer le nettoyage, par exemple enlever un listener ou un timer
  }

  render(){
    return <div>Composant</div>;
  }
}).
```

GESTION DU CYCLE DE VIE DES COMPOSANTS FONCTIONNELS

Avec l'introduction des **hooks** dans React 16.8, la gestion du cycle de vie dans les composants fonctionnels a été simplifiée. Vous pouvez utiliser des hooks comme `useEffect()` et `useState()` pour gérer différentes phases du cycle de vie dans un composant fonctionnel. Le hook `useEffect()` permet de gérer plusieurs aspects du cycle de vie (`componentDidMount()`, `componentDidUpdate()`, et `componentWillUnmount()`) dans un composant fonctionnel.

- **Montage** : Le premier argument de `useEffect()` est une fonction qui s'exécute après le rendu initial, équivalent à `componentDidMount()`.
- **Mise à jour** : La fonction dans `useEffect()` est appelée après chaque mise à jour, sauf si vous spécifiez des dépendances dans un tableau en deuxième argument.
- **Démontage** : Vous pouvez retourner une fonction de nettoyage dans `useEffect()`, qui sera exécutée avant le démontage, équivalent à `componentWillUnmount()`.

(ex :
```
import {useState, useEffect} from 'react';

function MyFunctionalComponent(){
  const [count, setCount] = useState(0);
```

```
  useEffect(() => {
    console.log("Composant monté"); //Exécuté après le premier rendu (montage)

    return () => { //considérée comme componentWillUnmount() par useEffect, voilà prkw
      console.log("Composant va être démonté"); //elle est exécutée avant le démontage
    };
  }, []); //Tableau de dépendances vide => exécuté uniquement au montage et démontage

  useEffect(() => {
    //Exécuté après chaque mise à jour du compteur (count)
    console.log("Compteur mis à jour : ", count);
  }, [count]); //Exécuté uniquement lorsque "count" change

  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrémenter</button>
    </div>
  );
}).
```

CHAPITRE 12 : REACT-ROUTER (VERSION 6)

Dans le monde du développement frontend, créer des interfaces dynamiques et interactives est essentiel pour offrir une expérience utilisateur fluide. Une partie cruciale de cette expérience est la navigation entre différentes vues ou "pages" dans une application web. Avec React, une bibliothèque JavaScript populaire pour construire des interfaces utilisateur, React Router est l'outil standard pour implémenter cette navigation. C'est une bibliothèque standard pour la gestion des routes dans les applications React. Contrairement à un site web classique où chaque clic sur un lien `<a>` charge une nouvelle page à partir du serveur, les applications React sont souvent des Single Page Applications (SPA). Cela signifie qu'au lieu de recharger la page à chaque changement, les différentes vues sont gérées par JavaScript localement, ce qui permet de rendre les transitions plus fluides et rapides. React Router permet donc de définir des chemins (routes) qui correspondent à différentes vues ou composants dans votre application. Grâce à cela, vous pouvez construire une application qui simule la navigation d'une application multipage, tout en restant une application à page unique. Il faut cependant l'installer en tapant : **`npm install react-router-dom`**.

CREER DES ROUTES

Dans une application React classique, il n'y a qu'une seule page HTML qui est chargée. Cependant, lorsque l'utilisateur clique sur un lien, il s'attend à ce que l'URL change et qu'une nouvelle "page" apparaisse. **React Router** permet cela sans rechargement du navigateur, en utilisant le concept de **routes**. Chaque route associe un chemin d'URL à un composant spécifique à afficher. Une route en React Router associe un chemin d'URL, non pas à une page, mais à un composant spécifique. On encapsule les différentes vues, donc routes d'un composant dans `BrowserRouter` (ex :

```
import BrowserRouter from 'react-router-dom';

function monComposant(){
  return (
    <BrowserRouter basename="/monComposant" >
      {/*le reste de votre composant va ici*/}
    </BrowserRouter>
  );
}).
```

Dans cet exemple, le composant `BrowserRouter` encapsule toute la vue du composant pour y permettre la gestion des routes. Cela ne change rien de visuel, mais permet à votre composant de gérer des routes et écouter les changements d'URL.
La props `basename` permet de spécifier une URL de base, à partir de laquelle toutes les routes de ce composant vont partir (ex : `"/monComposant/Vue1"`). Elle est optionnelle, mais recommandée.

Une fois que vous avez wrappé votre composant dans un `BrowserRouter`, on peut désormais y **créer** les routes en question. Pour ça, on utilise des composants `Route`, qui lient chaque URL à un composant spécifique. Ces composants doivent à leur tour être wrappés dans un `Routes` qui permet de ne rendre qu'une route à la fois. Le composant `Route` de React Router est l'un des éléments essentiels pour créer une navigation conditionnelle et basée sur les chemins dans les applications React.

1. **Création d'une route avec `createBrowserRouter()` :**
   (ex :
```
import {BrowserRouter, Routes, Route} from 'react-router-dom';

function monComposant(){
  return (
    <BrowserRouter basename="/monComposant" >
      <Routes>
        {
           const monRouteur = createBrowserRouter([{
             element: <Team />,
             path: "/Vue1"
           }]);
        }
      </Routes>
    </BrowserRouter>
  );
```
   }). On déclare ses props comme élément d'un tableau car une route peut avoir d'autres *sous-routes*.
2. **Création d'une route en JSX :**
   (ex :
```
import {BrowserRouter, Routes, Route} from 'react-router-dom';
import {Vue1} from './Vue1.js';

function monComposant(){
  return (
    <BrowserRouter basename="/monComposant" >
      <Routes>
        <Route path="/vue1" element={<Vue1 />} />
      </Routes>
    </BrowserRouter>
  );
```
   }). Vaut mieux préférer cette notation. `<Routes>` s'appelait `<Switch>` dans les versions anciennes.

Le composant `Route` a besoin de deux props **<u>obligatoires</u>** pour être valide : `path` et `element`.

- `path` : spécifie l'URL pour laquelle une route doit correspondre.
- `element` : spécifie directement le composant qui doit être rendu lorsque la route correspondante est active.

Quand on utilise React Router, <u>la première route qui sera affichée dépend de l'ordre dans lequel on les définit et si vous avez spécifié une route par défaut</u>. Par exemple, si vous utilisez le composant `<Routes>` pour définir tes routes, la première route qui correspond à l'URL actuelle sera affichée.

Si vous voulez qu'une route soit affichée par défaut (par exemple quand l'utilisateur arrive sur cette section du site), tu peux utiliser le chemin `path="/"` ou `path="index"`, en fonction de l'organisation de tes routes (ex :
```
import {Routes, Route} from "react-router-dom";
import {VuePardéfaut, Vue1, Vue2, NavBar} from "./mesVues.js";

function MonComposant(){
  return (
    <div> {/*ce navbar sera affiché, peu importe la vue dans laquelle on se trouve*/}
     <Route path="/navbar" element={<NavBar />} />
      <Routes>
        {/* Cette route sera affichée par défaut si vous accedez à l'URL exacte */}
```

```
        <Route path="/" element={<VuePardéfaut />} />
        {/* Autres routes */}
        <Route path="/vue1" element={<Vue1 />} />
        <Route path="/vue2" element={<Vue2 />} />
      </Routes>
    </div>
  );
}
```

). Ici, la route avec `path="/"` sera affichée par défaut lorsque l'URL sera celle du composant.

Voici les autres props qu'on peut utiliser avec le composant `Route` :

- `loader` : permet de charger des données asynchrones avant que la route ne soit rendue. Elle fonctionne comme une fonction qui retourne soit une promesse, soit des données. C'est utile pour précharger les informations nécessaires au rendu d'une route (ex :
  ```
  const loaderFonction = async () => {
    const reponse = await fetch('/api/data');
    return reponse.json();
  };

  //liaison du composant à une route
  <Route path="/data" element={<DataPage />} loader={loaderFonction} />
  ```
). Avant que la route ne se rende, `loader` sera appelé pour obtenir des données. Ces données sont ensuite disponibles dans votre composant via un hook comme `useLoaderData()`.
- `action` : est utilisée pour gérer des soumissions de formulaires ou d'autres types d'actions associées à une route. Similaire à loader, elle permet une gestion asynchrone. La différence principale est que `action` est déclenchée lorsqu'un utilisateur effectue une action comme la soumission d'un formulaire (ex :
  ```
  const actionFonction = async ({requete}) => {
    const formData = await requete.formData();
    await fetch('/api/submit', {method: 'POST', body: formData});
    return redirect('/success');
  };

  //liaison du composant à une route
  <Route path="/form" element={<FormPage />} action={actionFonction} />
  ```
). `action` est appelée lorsqu'une requête POST ou PUT (souvent liée à un formulaire) est déclenchée. Vous pouvez récupérer `requete` pour traiter les données du formulaire (ici, elle est passée implicitement).
- `errorElement` : est utilisée pour définir un composant à afficher lorsqu'une erreur survient lors du chargement des données avec `loader`, ou lors de l'exécution de `action`. Cela permet de gérer gracieusement les erreurs sans casser l'application (ex :
  ```
  <Route
    path="/data"
    element={<DataPage />}
    loader={loaderFunction}
    errorElement={<ErrorPage />}
  />
  ```
). Si `loader` échoue (par exemple, une erreur de réseau ou une mauvaise réponse API), React Router rendra le composant défini dans `errorElement`.
- `shouldRevalidate` : utilisée pour contrôler si React Router doit recharger les données (par loader) lors de la navigation entre les routes ou lors de certaines actions. Par défaut, le loader est réexécuté chaque fois que l'utilisateur revisite une route, mais vous pouvez personnaliser ce comportement (ex :
  ```
  const shouldRevalidateFonction = ({actuelUrl, prochainUrl}) => {
    return actuelUrl.pathname !== prochainUrl.pathname; //slmnt si la route change
  };

  <Route
    path="/dashboard"
    element={<Dashboard />}
    loader={dashboardLoader}
    shouldRevalidate={shouldRevalidateFonction}
  ```

/>). Cette fonction reçoit un objet contenant les informations sur la route actuelle et la prochaine route. Vous pouvez utiliser ces informations pour déterminer si le rechargement des données est nécessaire.

- `hydrateFallbackElement` : est utilisée dans les environnements de rendu côté serveur (SSR) avec l'hydratation. Elle permet de spécifier un élément qui sera rendu pendant l'hydratation du composant. Une fois l'hydratation terminée, l'élément principal est rendu (ex :

```
<Route
  path="/"
  element={<Acceuil />}
  loader={homeLoader}
  hydrateFallbackElement={<LoadingIndicator />}
/>
```

). Lors du processus de SSR, cette prop permet d'afficher un indicateur ou un élément en attendant que le composant se réhydrate.

NAVIGUER ENTRE DIFFERENTES ROUTES

Pour naviguer entre les différentes routes, on peut utiliser le composant `<Link>` ou le hook `useNavigate()` de React Router (ex :

```
import {Link} from "react-router-dom";

function MonComposant(){
  return (
    <div>
      <nav> {/*ce navbar sera affiché, peu importe la vue dans laquelle on se trouve*/}
        <Link to="/">Home</Link>
        <Link to="/vue1">Vue 1</Link>
        <Link to="/vue2">Vue 2</Link>
      </nav>

      <Routes>
        <Route path="/" element={<ParDéfaut />} />
        <Route path="/vue1" element={<Vue1 />} />
        <Route path="/vue2" element={<Vue2 />} />
      </Routes>
    </div>
  );
}
```

En cliquant sur les liens, vous pourrez naviguer entre les différentes vues. `to` spécifie la destination du lien. En utilisant du hook `useNavigate()` :

```
import {useNavigate} from "react-router-dom";

function MonComposant(){
  const navigate = useNavigate();

  return (
    <div>
      <button onClick={() => navigate('/')}>Home</button>
      <button onClick={() => navigate('/vue1')}>View 1</button>
      <button onClick={() => navigate('/vue2')}>View 2</button>

      <Routes>
        <Route path="/" element={<ParDéfaut />} />
        <Route path="/vue1" element={<Vue1 />} />
        <Route path="/vue2" element={<Vue2 />} />
      </Routes>
    </div>
  );
```

}).

Le composant `<NavLink>` est une version améliorée du composant `<Link>` dans React Router. Il est utilisé pour créer des liens de navigation avec des fonctionnalités supplémentaires, notamment la possibilité de styliser un lien en fonction de la route active, c'est-à-dire la route sur laquelle l'utilisateur se trouve actuellement.

Cela est particulièrement utile pour les barres de navigation où il est important d'indiquer visuellement à l'utilisateur quelle page est active (ex :

```
import {NavLink} from 'react-router-dom';

function Navbar(){
  return (
    <nav>
      <ul>
        <li>
          <NavLink to="/" end activeClassName="active-link">Acceuil</NavLink>
        </li>
        <li>
          <NavLink to="/about" activeClassName="active-link">A Propos</NavLink>
        </li>
        <li>
          <NavLink to="/contact" activeClassName="active-link">Contacts</NavLink>
        </li>
      </ul>
    </nav>
  );
}

export default Navbar;
```

).

`to` : Spécifie la destination du lien, comme pour le composant `<Link>`.
`activeClassName="active-link"` : Ajoute automatiquement la classe CSS spécifiée (ici, `active-link`) lorsque le lien est actif, c'est-à-dire lorsque la route correspondante est affichée. Cela permet de styliser le lien différemment pour montrer qu'il est actif.
`end` : Lorsque ce prop est utilisé, le lien est considéré comme actif uniquement si la route correspond exactement à l'URL. Par exemple, `to="/"` avec `end` ne sera actif que lorsque l'utilisateur est précisément sur la page d'accueil, et non sur `/about` ou `/contact`.

En plus d'utiliser `activeClassName`, il est possible de personnaliser l'apparence des liens actifs ou inactifs en utilisant le prop `style` ou `className`, qui peut être une fonction retournant des styles ou des classes en fonction de l'état du lien (ex :

```
<NavLink
  to="/about"
  className={({isActive}) => (isActive ? 'active-link' : 'inactive-link')}
>A propos</NavLink>
```

ou

```
<NavLink
  to="/contact"
  style={({isActive}) => ({
    fontWeight: isActive ? 'bold' : 'normal',
    color: isActive ? 'green' : 'gray'})}>
Contact
</NavLink>
```

). Si la route est active, la classe `'active-link'` est appliquée, sinon la classe `'inactive-link'` est utilisée. Le lien vers `/contact` sera mis en gras et en vert lorsqu'il est actif, sinon il aura une couleur grise et un poids de police normal.

GESTION DES ROUTES NON-SPECIFIEES

Pour gérer les routes non spécifiées, on peut définir une route **fallback** avec `path="*"` qui sera rendue lorsqu'aucune des autres routes ne correspond (ex :

```
<Routes>
  <Route path="/" element={<VueParDéfaut />} />
  <Route path="/vue1" element={<Vue1 />} />
  <Route path="/vue2" element={<Vue2 />} />

  {/*Fallback pour les routes non spécifiées*/}
  <Route path="*" element={<Erreur404 />} />
</Routes>
```

). Ici, la route `path="*"` affichera le composant `Erreur404` lorsque l'utilisateur essaiera d'accéder à une route non définie dans le composant.

QUELQUES FONCTIONS UTILITAIRES DE REACT ROUTER

La fonction `json()` est utilisée pour renvoyer une réponse JSON dans le cadre des loaders ou des actions de React Router. Elle permet d'envoyer des données JSON aux composants, en particulier lors de la gestion des requêtes asynchrones pour charger ou soumettre des données via le routeur. `json(données, init)` retourne une réponse HTTP avec les données JSON fournies. On peut aussi passer des options d'initialisation (`init`), comme des en-têtes supplémentaires ou un code de statut http (ex :

```
import {json} from 'react-router-dom';

async function chargeurDonnées(){
  const données = await fetchDataFromAPI();
  return json(données);
}).
```

La fonction `redirect()` est utilisée pour rediriger un utilisateur vers une autre route. Elle est généralement appelée à partir des `loaders`, `actions` ou lorsqu'une certaine condition est remplie pour rediriger vers une autre page. `redirect(url, init)` redirige l'utilisateur vers l'URL spécifiée. L'argument `init` permet de spécifier des paramètres supplémentaires, comme le code de statut HTTP (ex :

```
import {redirect} from 'react-router-dom';

async function chargeurDonnées(){
  const isAuthenticated = await checkAuth();
  if (!isAuthenticated){
    return redirect('/login');
  }
  return null;
}).
```

La fonction `redirectDocument()` fonctionne de manière similaire à redirect, mais elle est plus spécifique aux redirections côté serveur pour les applications universelles (SSR). Elle effectue une redirection basée sur le contexte de document HTTP plutôt que sur le simple routage (ex :

```
import {redirectDocument} from 'react-router-dom/server';

async function chargeurDonnées(){
  return redirectDocument('/nouvel-endroit');
}).
```

L'attribut `replace` est un utilitaire de navigation qui remplace l'entrée actuelle dans l'historique par une nouvelle route. Cela signifie que lorsque l'utilisateur clique sur le bouton "retour", il ne pourra pas revenir à la page précédente (ex :

```
import {useNavigate} from 'react-router-dom';

function MonComposant(){
  const navigate = useNavigate();
```

```
  function handleReplace(){
    navigate('/nouvelle-page', {replace: true});
  }

  return <button onClick={handleReplace}>Aller à la nouvelle page</button>;
}).
```

`replace` dans `useNavigate()` : En spécifiant `{replace: true}`, on remplace l'entrée actuelle de l'historique avec la nouvelle URL, empêchant l'utilisateur de revenir en arrière à l'URL remplacée.

Pour le reste, vous pouvez visiter la documentation officielle de React Router.

CHAPITRE 13 : LES HOOKS

Les **hooks** en React sont des fonctions qui permettent d'ajouter des fonctionnalités à des composants fonctionnels (des composants qui sont simplement des fonctions JavaScript). Avant l'introduction des hooks dans la version 16.8 de React, les composants devaient être des **classes** pour utiliser des fonctionnalités comme l'état (state) ou le cycle de vie. Désormais, avec les hooks, ces fonctionnalités sont accessibles aux composants fonctionnels, ce qui rend le code plus simple et plus lisible.

LES HOOKS DE GESTION D'ETAT

Dans React, l'un des concepts les plus importants est la **gestion d'état**. L'**état** représente toutes les données qu'un composant gère et qui changent au fil du temps. Par exemple, dans une application où on gère un panier d'achats, l'état pourrait être la liste des articles que vous ajoutez ou retirez. React utilise ce concept pour savoir quand et comment mettre à jour l'interface utilisateur (UI) en fonction des changements d'état. Pour comprendre cela bien en profondeur, nous allons parler des hooks de gestion d'état : `useState()`, `useReducer()`, et `useSyncExternalStore()`. Ils sont chacun conçus pour des situations différentes, et on les utilise selon la complexité de l'état qu'on veut gérer.

Avant de parler des hooks, il est essentiel de bien saisir ce que signifie l'**état**. Imaginez que vous créez un site web qui affiche un compteur. Chaque fois que vous cliquez sur un bouton, le compteur doit augmenter. Ce nombre qui change à chaque clic est l'état.
Dans React, **l'état** est ce qui permet à un composant (une partie de votre page) de garder en mémoire une information qui évolue. Et à chaque fois que cette information (l'état) change, React met à jour automatiquement l'interface. Exemple simple d'état :

- Un compteur qui s'incrémente.
- Une liste d'articles dans un panier.
- La case cochée ou décochée d'un formulaire.

1. `useState()` : Ce hook est le plus simple à comprendre. C'est celui que vous utiliserez probablement le plus souvent quand vous commencez avec React. Il permet de gérer des petits morceaux d'état local, comme des nombres, des chaînes de caractères, des tableaux ou des objets (ex :

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

function Compteur(){
  const [compteur, setCompteur] = useState(0);//compteur est l'état initialisé à 0

  return (
    <div>
      <p>Vous avez cliqué {compteur} fois</p>
      <button onClick={() => setCompteur(compteur + 1)}>Cliquez ici</button>
    </div>
  );
}).
```

Dans ce code, `useState(0)` initialise l'état avec la valeur `0`. Ici, `0` est la valeur initiale du compteur. `compteur` est la variable d'état, qui représente la valeur actuelle du compteur. `setCompteur()`

est une fonction qui permet de changer la valeur de compteur. Chaque fois que vous l'appelez, React met à jour la valeur de compteur et réaffiche le composant avec la nouvelle valeur.

*Utilisez `useState()` lorsque l'état est simple. Par exemple, pour un compteur, un champ de texte, un bouton qui change de couleur... ou si l'état n'affecte qu'un seul composant et est facile à gérer.

2. `useReducer()` : Quand l'état devient plus complexe, par exemple si vous gérez plusieurs morceaux d'informations qui dépendent les uns des autres, `useState()` peut devenir compliqué à utiliser. C'est là que `useReducer()` entre en jeu. Il fonctionne un peu différemment de `useState()`. Avec ce dernier, vous avez une fonction (`setCompteur()` par exemple) qui change directement l'état. Mais avec `useReducer()`, vous définissez un ensemble de règles, appelé un **reducer**, qui décide comment l'état doit changer en fonction d'une **action** (ex :

```
import React, {useReducer} from 'react';

function CompteurAvance(){
  const [mesEtats, dispatcher] = useReducer(
    //le réducteur : Il dispatche les actions selon les états.
    (etats, action) => {
      /*cette fonction retourne tjrs un nouvel objet, identique à l'objet d'état
      mais modifiant uniquement les états concernés, ici, l'état compteur*/
      switch (action.choix){
        case 'incrementer':
          return {compteur: etats.compteur + 1};
        case 'decrementer':
          return {compteur: etats.compteur - 1};
        default:
          return etats; //on ne change rien
      }
    }, {compteur: 0}); //l'objet d'états. Ici, il n'en contient qu'un seul

  return (
    <div>
      <p>Compteur : {mesEtats.compteur}</p>
      <button onClick={() => dispatcher({choix: 'incrementer'})}>+1</button>
      <button onClick={() => dispatcher({choix: 'decrementer'})}>-1</button>
    </div>
  );
}
```

). On a commencé par déstructurer `useReducer(reducteur, objetDetats)`. Le premier argument est une fonction (qui doit toujours prendre deux paramètres : `etats` et `action`) qui sera appelée implicitement à chaque fois qu'un des états va changer et qui va se charger de dispatcher la logique adéquate. Le deuxième argument est un objet qui contient tous les états qu'on veut capturer dans le composant. `useReducer()` retourne ensuite 2 choses : un objet `mesEtats` qui contient tous les états capturés et `dispatcher()`, une fonction qui prend un argument : l'action et qui retourne un nouvel objet qui est implicitement affecté à `mesEtats`. Lorsque l'utilisateur clique sur l'un des boutons, `dispatcher()` s'exécute et en fonction de la logique, retourne un nouvel objet qui est une copie l'objet d'état, mais actualisé, et l'affecte implicitement à `mesEtats`.

*Utilisez `useReducer()` lorsque vous voulez créer de formulaires complexes (Par exemple, la gestion de plusieurs champs d'un formulaire avec des actions pour chaque modification de champ), lorsque vous gérez des états de composant (Comme un panier d'achat, où plusieurs types d'actions peuvent modifier l'état (ajouter un produit, enlever un produit, modifier la quantité)) ou avec des applications avec une logique métier complexe : Lorsque la logique de mise à jour de l'état devient trop complexe pour être gérée avec `useState()`.

3. `useSyncExternalStore()` : C'est est un hook plus avancé, introduit dans React 18. Son but est de permettre aux composants de s'abonner à une source d'état externe, c'est-à-dire une source qui existe en dehors de React (comme un store global dans Redux, ou une API qui envoie des mises à jour).

Imaginez que vous avez un store (un gestionnaire d'état centralisé) qui garde une valeur, par exemple le nombre total d'utilisateurs connectés à une application. Chaque fois que ce nombre change, plusieurs composants doivent se mettre à jour. `useSyncExternalStore()` aide à gérer cette synchronisation efficacement (ex :

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

//Simulons un store externe (un objet)
const store = {
  _valeur: 0,
  ecouteurs: new Set(),
  getValeur(){
    return this._valeur;
  },
  sAbonner(ecouteur) {
    this.ecouteurs.add(ecouteur);
    return () => this.ecouteurs.delete(ecouteur); //Fonction de désabonnement
  },
  incrementer(){
    this._valeur++;
    this.ecouteurs.forEach(ecouteur => ecouteur());
  }
};


function CompteurAvecStore(){
  const compteur = useSyncExternalStore(
    store.sAbonner,   //Fonction pour s'abonner
    store.getValeur); //Fonction pour récupérer la valeur actuelle

  return (
    <div>
      <p>Compteur (via store externe) : {compteur}</p>
      <button onClick={() => store.incrementer()}>Incrémenter</button>
    </div>
  );
}
```

). Le hook `useSyncExternalStore()` prend généralement trois paramètres :

- **subscribe** (obligatoire) : C'est une fonction qui permet au composant de s'abonner aux mises à jour du store externe. Le rôle principal de cette fonction est d'ajouter implicitement un « écouteur » (une fonction) qui sera appelé chaque fois que les données dans le store changent. Cette fonction doit retourner une fonction de « désabonnement », qui sera appelée lorsque le composant se démonte, pour que le composant arrête de recevoir des notifications de mise à jour. Ici, `store.sAbonner` est le subscribe.
- **getSnapshot** (obligatoire) : C'est une fonction qui permet de récupérer la valeur actuelle ou le « snapshot » des données dans le store. Cette fonction est appelée à chaque re-rendu pour obtenir la valeur la plus à jour. C'est ce que `useSyncExternalStore()` utilise pour déterminer la valeur actuelle du store à afficher dans le composant. Ici, `store.getValeur` est le getSnapshot.
- **getServerSnapshot** (optionnel) : Ce paramètre est utilisé principalement pour le rendu côté serveur (server-side rendering). Il permet de spécifier une fonction qui va récupérer le snapshot (ou la valeur actuelle) des données côté serveur. Lorsque le rendu se fait côté serveur, cette fonction est utilisée pour obtenir la version du store qui doit être rendue initialement. C'est un paramètre rarement utilisé dans le développement classique côté client.

Le hook `useSyncExternalStore()` retourne à la variable `compteur` la valeur actuelle du store externe (dans ce cas, la valeur du compteur) et la met automatiquement à jour chaque fois que le store change.

*Utilisez `useSyncExternalStore()` si vous utilisez une bibliothèque comme **Redux** ou une autre source externe d'état (ce hook permet de synchroniser l'état entre le store externe et vos

composants) ou si vous avez un état partagé entre plusieurs composants dans une application, et que cet état est géré de manière centralisée.

LES HOOKS DE GESTION DE CYCLE DE VIE OU D'EFFETS SECONDAIRES

Les hooks de gestion d'effets secondaires en React permettent d'exécuter du code en dehors du cycle de rendu principal de React. Ce code peut inclure des opérations comme les requêtes API, la manipulation du DOM, les abonnements à des événements globaux, etc. Ces effets sont appelés **effets secondaires** parce qu'ils ne font pas partie du processus de rendu initial, mais interviennent après que React a affiché le composant ou lorsqu'un changement se produit dans l'interface.

Un **effet secondaire** en développement correspond à une opération qui interagit avec des systèmes ou des états externes au composant. Par exemple :

- Récupérer des données d'un serveur (requête API).
- Manipuler directement le DOM pour gérer des animations.
- Mettre à jour manuellement des valeurs en dehors de React (par exemple, synchroniser une barre de défilement ou un titre de page).

En React, on ne veut pas que ce genre d'opérations interfère avec le **cycle de rendu** (c'est-à-dire la manière dont les composants s'affichent à l'écran). Les hooks comme `useEffect()`, `useLayoutEffect()` et `useInsertionEffect()` sont conçus pour gérer ces effets sans perturber le rendu de l'interface.

1. `useEffect()` **:** C'est probablement le hook le plus utilisé en React pour les effets secondaires. Il vous permet de dire à React d'exécuter un bout de code **après** que le composant a été affiché à l'écran. C'est particulièrement utile pour les opérations asynchrones, comme charger des données depuis un serveur ou s'abonner à des événements globaux (par exemple, écouter le défilement de la page ou le redimensionnement de la fenêtre). `useEffect()` prend deux arguments :
    - **Un effet** (une fonction) : C'est le code que vous voulez exécuter après que le composant a été rendu. Ce code peut inclure des actions comme faire une requête API ou définir des minuteries.
    - **Un tableau de dépendances** (facultatif) : Un tableau qui liste les valeurs ou variables que vous voulez surveiller. Si une des variables dans ce tableau change, React exécutera à nouveau l'effet.

    (ex :

```
import React, {useState, useEffect} from 'react';

function CompteArebours(){
  const [secondes, setSecondes] = useState(0);

  //Utilisation de useEffect pour démarrer une minuterie quand le cmpsnt est monté
  useEffect(() => {
    const intervalle = setInterval(() => {
      setSecondes((prevSecondes) => prevSecondes + 1);
    }, 1000);

    //= fnctn de nettoyage pour arrêter l'itrvl quand le cmpsnt est démonté
    return () => clearInterval(intervalle); //fonction de l'API Web
  }, []); //tableau vide signifie que l'effet ne s'exécute qu'1 fois, au montage.

  return <div>Temps écoulé : {secondes} secondes</div>;
}
```

). L'effet démarre lorsque le composant est monté (c'est-à-dire qu'il apparaît à l'écran). Le tableau de dépendances vide (`[]`) signifie que cet effet ne sera déclenché qu'une seule fois, juste après le premier rendu. À l'intérieur de l'effet, nous démarrons une minuterie qui incrémente le compteur chaque seconde. Si la fonction à l'intérieur de `useEffect()` retourne une fonction, celle-ci est appelée lorsque le composant est sur le point d'être retiré de l'écran. On l'appelle : **fonction de nettoyage**. Elle arrête l'intervalle pour éviter que la minuterie continue à tourner même après que le composant ne soit plus affiché.

Le tableau de dépendances que vous passez à `useEffect()` est très important. Il détermine quand l'effet est exécuté ou réexécuté :

- **Pas de tableau** : Si vous ne passez pas de tableau de dépendances, `useEffect()` sera exécuté après **chaque** rendu du composant. Cela peut entraîner des boucles infinies si l'effet modifie l'état du composant (comme appeler `setState()` à chaque fois).
- **Tableau vide (`[]`)** : L'effet ne s'exécutera qu'une seule fois, lorsque le composant est monté (similaire à `componentDidMount()` en React classe).
- **Tableau avec des variables** : Si vous passez des variables dans le tableau de dépendances (par exemple `[compte]`), l'effet sera exécuté uniquement lorsque l'une de ces variables change (similaire à `componentDidUpdate()` en React classe).

2. `useLayoutEffect()` **:** C'est très similaire à `useEffect()`, mais il a une différence cruciale : il est exécuté **juste avant que le rendu soit visible à l'écran**. Il est déclenché **synchroniquement** après que le DOM a été mis à jour, mais avant que le navigateur n'affiche le composant à l'utilisateur. Cela signifie que tout ce que vous faites dans `useLayoutEffect()` sera appliqué avant que l'utilisateur voie quoi que ce soit. Cela est utile si vous devez mesurer des éléments dans le DOM ou faire des ajustements précis avant l'affichage. Il est généralement utilisé dans des situations où la manipulation du DOM doit être faite immédiatement, comme :
   - Calculer des dimensions ou la position d'un élément.
   - Appliquer des styles de manière synchronisée pour éviter des "flickers" (effets visuels indésirables).

   (ex :
```
import React, {useState, useLayoutEffect, useRef} from 'react';

function LargeText(){
  const [taille, setTaille] = useState(0);
  const texteRef = useRef(null);

  //Utilisation de useLayoutEffect() pour mesurer
  //la taille du texte avant que le rendu ne soit visible
  useLayoutEffect(() => {
    const rect = texteRef.current.getBoundingClientRect();
    setTaille(rect.width);
  }, []);

  return (
    <div>
      <p ref={texteRef}>Ce texte est mesuré avant le rendu visible.</p>
      <p>Largeur du texte : {taille}px</p>
    </div>
  );
}
```
). Cet effet est exécuté juste après que le composant a été monté et que le DOM a été mis à jour, mais avant que l'utilisateur ne voie quoi que ce soit. Ici, `getBoundingClientRect()` est utilisé pour mesurer la largeur d'un élément de texte. Cette mesure est effectuée avant que le rendu final ne soit affiché à l'utilisateur.

   *Utilisez `useLayoutEffect()` si vous devez effectuer des calculs de positionnement ou de taille d'éléments avant que l'utilisateur ne voie la page. C'est utile pour éviter les clignotements visuels. Dans la plupart des cas, `useEffect()` suffit. `useLayoutEffect()` ne doit être utilisé que si vous avez vraiment besoin que l'effet soit exécuté **avant** que l'utilisateur ne voie le changement.

3. `useInsertionEffect()` : C'est un hook introduit dans React 18 et est destiné à des **cas très spécifiques** où vous devez insérer des styles dans le DOM **avant** que tout autre effet (ou rendu) ne soit appliqué. Il est principalement utilisé pour des bibliothèques de styles en JavaScript, comme styled-components ou émotion, qui doivent insérer des styles CSS directement dans la page à un moment très précis du cycle de rendu. Il est exécuté avant `useLayoutEffect()` et `useEffect()`, et est conçu pour gérer uniquement l'insertion de styles CSS. Il est rarement utilisé directement par les développeurs d'applications classiques, mais plus par les bibliothèques qui manipulent des styles dynamiquement.

(ex :

```
import React, {useInsertionEffect} from 'react';

function StyledComponent(){
  //Utilisation de useInsertionEffect() pour ajouter des styles avant le rendu
  useInsertionEffect(() => {
    const style = document.createElement('style');
    style.innerHTML = `
      .custom-style {
        color: red;
        font-size: 20px;
      }`;
    document.head.appendChild(style);

    //Nettoyage : retirer les styles lorsqu'ils ne sont plus nécessaires
    return () => document.head.removeChild(style);
  }, []);

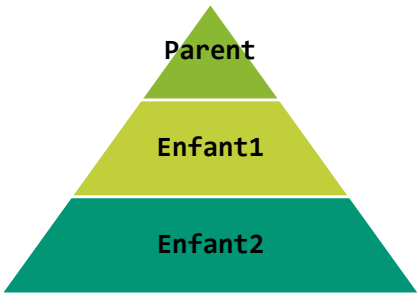
  return <div className="custom-style">Texte avec style dynamique</div>;
}).
```

Ici, `useInsertionEffect()` est utilisé pour ajouter un élément `<style>` dans le `<head>` du document, appliquant ainsi des styles avant que le composant ne soit rendu. De même qu'avec `useEffect()`, vous pouvez retourner une fonction de nettoyage pour supprimer les styles lorsque le composant est démonté.

*Utilisez `useInsertionEffect()` pour les bibliothèques de styles qui ont besoin de contrôler précisément à quel moment les styles CSS sont injectés dans la page. En général, vous n'aurez pas à l'utiliser dans des applications React classiques, sauf si vous développez une bibliothèque de styles.

LE HOOK DE GESTION DE CONTEXTE

En React, le concept de **contexte** est utilisé pour éviter ce qu'on appelle le « *prop drilling* » (passage répétitif de props à travers plusieurs couches de composants).



Par exemple, si un composant **`Parent`** veut passer un état à l'état **`Enfant2`**, il doit impérativement passer par **`Enfant1`**. Le concept de contexte permet justement à **`Parent`** de passer des états à **`Enfant2`** sans passer par **`Enfant1`**.

Il n'existe qu'un seul hook de gestion de contexte en React : **`useContext()`**. Il permet de partager des informations à travers plusieurs composants imbriqués sans avoir à passer manuellement ces données via des props à chaque niveau de cet arbre. Ce hook est particulièrement utile lorsque vous avez des données ou des états globaux à partager entre des composants distants dans une même hiérarchie.

`useContext()` est un hook qui permet à un composant de lire et utiliser la valeur d'un contexte défini au-dessus de lui dans la hiérarchie des composants. En d'autres termes, il permet de se connecter à un contexte et d'accéder aux données partagées de manière globale, sans avoir à passer ces données à chaque composant parent via des props.

Avant d'utiliser `useContext()`, vous devez d'abord créer un **objet de contexte** à l'aide du constructeur `React.createContext()`. Ensuite, vous créez un **Provider** (un composant) qui encapsule les composants qui doivent accéder aux données du contexte. Le hook `useContext()` permet alors à n'importe quel composant enfant d'accéder aux valeurs du contexte, sans avoir à les passer manuellement comme props (ex :

1. **Création du Contexte** : Vous créez un contexte en utilisant le constructeur `React.createContext()` et fournissez une valeur par défaut (facultative). Vous utilisez ensuite un champ `Provider`, défini nativement dans l'objet de contexte créé pour passer une valeur spécifique aux composants enfants (ex :

```
import React, {createContext, useState} from 'react';

//Création du contexte avec le constructeur createContext()
const ThemeContext = createContext();

function ThemeProvider({enfants}){
  //Etat du thème (on peut définir une autre logique ici)
  const [theme, setTheme] = useState('clair');

  return (
    <ThemeContext.Provider value={{theme, setTheme}}>
      {enfants}
    </ThemeContext.Provider>
  );
}).
```

2. **Utilisation de `useContext()` dans un composant enfant** : N'importe quel composant enfant peut utiliser `useContext()` pour accéder au thème sans avoir à passer par les props (ex :

```
import React, {useContext} from 'react';

function BtnChangerTheme(){
  const {theme, setTheme} = useContext(ThemeContext); //Récupérer le contexte

  return (
    <div>
      <p>Le thème actuel est : {theme}</p>
      <button onClick={() => setTheme(theme === 'clair' ? 'sombre' : 'clair')}>
        Changer le thème
      </button>
    </div>
  );
}).
```

4. **Utilisation dans l'application** : Vous pourrez encapsuler votre application ou une partie de votre application avec le Provider que vous avez créé pour permettre aux composants descendants d'accéder aux données du contexte (ex :

```
import ThemeProvider from './ThemeContext.js';
import BtnChangerTheme from './BtnChangerTheme.js';

function App(){
  return (
    <ThemeProvider>
      <BtnChangerTheme />
    </ThemeProvider>
  );
}).
```

Vous pouvez définir plusieurs contextes pour gérer différentes parties de votre état global (par exemple, un contexte pour le thème, un autre pour l'utilisateur, etc.).

LES HOOKS DE GESTION DE REFERENCES

Les hooks de gestion de référence en React permettent de manipuler directement le DOM ou des instances de composants sans passer par le cycle de rendu normal de React. Ces hooks sont essentiels lorsqu'il s'agit de conserver des informations qui ne nécessitent pas de déclencher un nouveau rendu, ou lorsque vous avez besoin de donner accès à des éléments spécifiques dans le DOM pour des interactions directes.

1. `useRef()` : c'est un hook qui permet de créer des **références** vers des éléments DOM ou pour stocker des valeurs persistantes entre différents rendus d'un composant, sans que ces valeurs ne provoquent de ré-rendu. Contrairement à l'état (via `useState()`), la modification d'une référence via `useRef()` n'entraîne pas de mise à jour du composant (ex :

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

function FocusInput(){
  //Crée une référence vers un élément DOM
  const referenceAuComposantInput = useRef(null);

  const seFocaliserSurLeComposantInput = () => {
    //Accède directement au champ de saisie
    referenceAuComposantInput.current.focus();
  };

  return (
    <div>
      <input ref={referenceAuComposantInput} type="text"
        placeholder="Cliquez sur le bouton pour me focus" />
      <button onClick={seFocaliserSurLeComposantInput}>Focus sur l'input</button>
    </div>
  );
}).
```

`useRef(null)` initialise une référence avec une valeur nulle. `useRef()` renvoie un objet qui possède une propriété current où vous pouvez stocker la référence à un élément. `referenceAuComposantInput.current.focus()` focalise `<input>` lorsque l'utilisateur clique sur le bouton, le mettant ainsi en surbrillance. Dans ce cas, `useRef()` est pratique car il permet d'accéder directement à un élément du DOM et d'interagir avec lui, sans que cela ne déclenche un nouveau rendu du composant.

En dehors des éléments DOM, `useRef()` peut également être utilisé pour stocker des informations qui doivent persister entre les rendus mais ne devraient pas déclencher un rendu quand elles sont modifiées. Par exemple, il peut être utilisé pour gérer une variable compteur interne ou pour mémoriser une valeur calculée (ex :

```
function Compteur(){
  //Stocker une valeur persistante qui ne cause pas de rendu
  const compteRef = useRef(0);

  const incrementer = () => {
    compteRef.current++; //Incrémente la valeur sans provoquer de rendu
    console.log(compteRef.current); //Log de la valeur mise à jour
  };

  return (
    <div>
      <button onClick={incrementer}>Incrémenter (sans re-rendu)</button>
    </div>
  );
}).
```

Contrairement à `useState()`, où toute modification entraîne un nouveau rendu, `useRef()` permet de modifier la valeur du champ natif `current` sans perturber le cycle de rendu.

*Utilisez `useRef()` si vous voulez accéder à des éléments du DOM directement, comme un champ de saisie ou un bouton ; Stocker des valeurs persistantes qui doivent rester entre les rendus mais ne nécessitent pas de provoquer un nouveau rendu ; ou Gérer des minuteries ou des abonnements qui doivent être conservés tout au long du cycle de vie d'un composant.

2. `useImperativeHandle()` : C'est un hook avancé qui permet à un composant de **contrôler exactement ce qui est exposé** via une référence (`ref`) lorsqu'un parent essaie d'accéder au composant enfant. Il est souvent utilisé en combinaison avec `forwardRef()` pour créer des composants dont les références sont partiellement ou entièrement contrôlées. Habituellement, lorsqu'un parent obtient une référence sur un enfant, il peut accéder directement à l'instance DOM de cet enfant. Mais `useImperativeHandle()` permet de **masquer** certains comportements ou de **contrôler** exactement ce que la référence expose. Pour utiliser `useImperativeHandle()`, le composant enfant doit être passé avec `forwardRef()`, ce qui permet de transmettre une référence du parent à l'enfant (ex :

```
import React, {useImperativeHandle, useRef, forwardRef} from 'react';

//Composant enfant qui expose une interface de commande via ref
const CustomInput = forwardRef((props, ref) => {
  const inputRef = useRef();

  //Utilisation de useImperativeHandle() pour exposer des méthodes spécifiques
  useImperativeHandle(ref, () => ({
    focusInput: () => {
      inputRef.current.focus(); //Méthode pour focus l'input
    },
    clearInput: () => {
      inputRef.current.value = ''; //Méthode pour vider l'input
    }
  }));

  return <input ref={inputRef} type="text"
    placeholder="Utilisez les méthodes exposées" />;
});

function ParentComponent(){
  const inputRef = useRef();

  return (
    <div>
      <CustomInput ref={inputRef} />
      <button onClick={() => inputRef.current.focusInput()}>
        Focus sur l'input
       </button>
      <button onClick={() => inputRef.current.clearInput()}>Vider l'input</button>
    </div>
  );
}).
```

`forwardRef()` permet à un composant d'accepter une référence passée depuis son parent. `useImperativeHandle()` permet d'exposer certaines fonctionnalités via la référence. Dans cet exemple, on expose deux méthodes : `focusInput()` et `clearInput()`, que le parent peut appeler. Contrairement à une simple référence d'élément DOM qui ne permettrait que l'accès direct au DOM, ici, le parent peut appeler des méthodes spécifiques définies par l'enfant, comme par exemple `clearInput()` ou `focusInput()`, créant ainsi une interface contrôlée.

*Utilisez `useImperativeHandle()` Si vous souhaitez limiter ou spécifier exactement ce qu'un parent peut faire avec une référence sur un composant enfant ou quand un composant enfant veut exposer des méthodes spécifiques (comme dans l'exemple avec `focusInput()` et `clearInput()`), mais pas toutes ses propriétés ou méthodes internes.

LES HOOKS DE TRANSITION

Les hooks de **gestion de transition** en React introduisent une façon de gérer les transitions de manière plus fluide et de mieux gérer les lourdes opérations de rendu dans les applications. Ils sont essentiels pour

rendre une interface utilisateur plus **réactive** et **performante** en différant certaines mises à jour ou en séparant les états critiques des états moins prioritaires.

1. `useTransition()` : C'est un hook qui permet de marquer certaines mises à jour d'état comme étant **non urgentes** (ou transitionnelles), ce qui signifie que ces mises à jour peuvent être différées pour éviter de bloquer des interactions critiques avec l'utilisateur. Cela permet à React de prioriser les opérations et de garantir que les mises à jour les plus importantes (comme les clics de bouton ou la saisie de texte) restent fluides, même si des opérations lourdes sont en cours (ex :

```
import React, {useState, useTransition} from 'react';

function rechercheLongue(recherche) {
  //Simule une recherche longue et complexe
  let resultats = [];
  for (let i = 0; i < 10000; i++){
    if (`Item ${i}`.includes(recherche)){
      resultats.push(`Item ${i}`);
    }
  }
  return resultats;
}

function Rechercher(){
  const [termesRecherché, settermesRecherché] = useState('');
  const [resultats, setResultats] = useState([]);

  const [enAttente, commencerTransition] = useTransition();

  const GestionRecherche = (e) => {
    const valeur = e.target.value;
    settermesRecherché(valeur);

    commencerTransition(() => {
      //Simulation d'une recherche lourde
      const resultatsFiltrés = rechercheLongue(valeur);
      setResultats(resultatsFiltrés);
    });
  };

  return (
    <div>
      <input
        type="text"
        value={termesRecherché}
        onChange={GestionRecherche}
        placeholder="Rechercher..."
      />
      {enAttente ? <p>Chargement des résultats...</p>
                 : <ResultsList results={resultats} />}
    </div>
  );
}
```

). Le hook `useTransition()` renvoie un tableau avec deux éléments : `enAttente` (indique si une transition est en cours) et `commencerTransition()` (une fonction pour démarrer une mise à jour non urgente). Ici, la recherche lourde est marquée comme transitionnelle en étant enveloppée dans `commencerTransition()`. Cela permet à React de reporter cette mise à jour si nécessaire, pour ne pas bloquer les interactions plus critiques (comme la saisie dans l'input). `enAttente` indique si la transition est encore en cours. Pendant cette période, un message de chargement est affiché.

*Utilisez `useTransition()` pour les opérations lourdes ou complexes (comme le filtrage, les recherches, ou les rendus massifs de listes) qui ne doivent pas bloquer les interactions utilisateur

immédiates ou si une interaction importante doit rester fluide (comme un bouton ou la saisie d'un formulaire) alors qu'une opération de fond est en cours.

2. `useDeferredValue()` **:** C'est un hook qui permet de prendre une **valeur lourde** ou une mise à jour d'état et de la différer jusqu'à ce que l'application soit plus réactive. Cela permet d'afficher d'abord les mises à jour critiques et de traiter ensuite des opérations moins prioritaires sans bloquer le rendu des autres composants. Contrairement à `useTransition()`, qui différencie les mises à jour critiques des mises à jour non urgentes au niveau de l'état, `useDeferredValue()` agit directement sur une **valeur** et peut être utilisé pour décaler l'application d'une valeur jusqu'à ce que les ressources soient disponibles (ex :

```
import React, {useState, useDeferredValue} from 'react';

function SearchComponent(){
  const [termesRecherché, settermesRecherché] = useState('');

  //Diffère la mise à jour du terme de recherche
  const rechercheDifferée = useDeferredValue(termesRecherché);

  const resultats = rechercheLongue(rechercheDifferée);

  return (
    <div>
      <input
        type="text"
        value={termesRecherché}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Rechercher..."
      />
      <ResultsList results={resultats} />
    </div>
  );
}

function rechercheLongue(recherche){
  // Simule une recherche longue et complexe
  let resultats = [];
  for (let i = 0; i < 10000; i++){
    if (`Item ${i}`.includes(recherche)){
      resultats.push(`Item ${i}`);
    }
  }
  return resultats;
}
```

). Ce hook prend la valeur `termesRecherché` et la diffère jusqu'à ce que l'application ait fini de gérer des mises à jour plus critiques. Cela permet d'éviter que le filtrage lourd ne ralentisse la mise à jour de l'interface utilisateur. La mise à jour `termesRecherché` se fait instantanément dans l'input, tandis que `rechercheDifferée` prend un peu plus de temps pour être mis à jour. Cela évite des ralentissements visibles lorsque l'utilisateur tape rapidement.

*Utilisez `useDeferredValue()` si une valeur (comme un terme de recherche ou une liste filtrée) nécessite beaucoup de ressources pour être calculée ou lorsque vous voulez garder des interactions immédiates fluides (comme taper dans un champ de recherche), tout en décalant le rendu de certains résultats non critiques.

LES HOOKS DE PERFORMANCE

Les hooks `useMemo()` et `useCallback()` sont des outils essentiels pour optimiser les performances d'une application React. Ils permettent de mémoriser des valeurs ou des fonctions, évitant ainsi des calculs

inutiles ou des recréations de fonctions à chaque rendu. Ces hooks sont particulièrement utiles dans les situations où les calculs sont coûteux ou lorsque des composants enfants reçoivent des fonctions comme props.

1. `useMemo()` : C'est un hook qui vous permet de **mémoriser** le résultat d'une fonction pour éviter de recalculer cette valeur à chaque rendu, sauf si les dépendances spécifiées changent. Il est utilisé principalement pour optimiser les performances des composants en évitant des calculs coûteux inutiles (ex :

```
import React, {useMemo, useState} from 'react';

function ListeFiltrée({items}){
  const [filtre, setFiltre] = useState('');

  //Utilisation de useMemo pour mémoriser le résultat du filtrage
  const itemsFiltrés = useMemo(() => {
    console.log('Calcul du filtrage...');
    return items.filter(item => item.includes(filtre));
  }, [items, filtre]); //Dépendances

  return (
    <div>
      <input
        type="text"
        value={filter}
        onChange={(e) => setFiltre(e.target.value)}
        placeholder="Filtrer les items..."
      />
      <ul>
        {itemsFiltrés.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
```

). Le résultat de la fonction de filtrage est mémorisé dans le tableau des dépendances. Cela signifie que si `items` et `filtre` ne changent pas, le filtrage ne sera pas recalculé lors de chaque rendu. En mémorisant le résultat, nous évitons de recalculer le filtrage si l'utilisateur tape rapidement dans le champ de saisie, rendant l'application plus réactive.

*Utilisez `useMemo()` pour mémoriser le résultat de calculs lourds qui n'ont pas besoin d'être recalculés à chaque rendu. Il est particulièrement utile pour des listes ou des tableaux qui nécessitent des transformations coûteuses.

2. `useCallback()` : C'est un hook qui vous permet de **mémoriser une fonction** afin qu'elle ne soit pas recréée à chaque rendu du composant. Cela est particulièrement utile lorsque la fonction est passée comme prop à un composant enfant, car cela peut éviter des re-rendus inutiles de ce composant enfant (ex :

```
import React, {useCallback, useState} from 'react';

function Compteur(){
  const [compte, setCompte] = useState(0);

  //Utilisation de useCallback pour mémoriser la fonction d'incrémentation
  const incrementeur = useCallback(() => {
    setCompte(prevCompte => prevCompte + 1);
  }, []); //Pas de dépendances, donc la fonction ne sera pas recréée

  return (
    <div>
      <p>Compteur : {compte}</p>
```

```
        <button onClick={incrementeur}>Incrémenter</button>
      </div>
    );
});
```

). La fonction `incrementeur()` est mémorisée, ce qui signifie qu'elle ne sera pas recréée à chaque fois que le composant Counter se rend. Si `incrementeur()` était passée à un composant enfant, ce dernier ne se re-renderait pas à chaque fois que `Compteur` se rend, ce qui améliore les performances globales.

*Utilisez `useCallback()` pour mémoriser des fonctions qui sont passées à des composants enfants afin d'éviter des re-rendus inutiles. Il est également utile pour des gestionnaires d'événements complexes ou coûteux qui doivent être passés à d'autres composants.

LES HOOKS DIVERS

En plus des hooks plus courants, React propose plusieurs hooks moins connus mais tout aussi utiles. Ces hooks divers apportent des fonctionnalités spécifiques qui peuvent faciliter le développement d'applications React. Pour en savoir plus, voir https://react.dev/reference/react/hooks#other-hooks.

CREER DES HOOKS PERSONNALISES

Les hooks personnalisés sont des fonctions JavaScript qui permettent de réutiliser la logique d'état et d'effet entre les composants React. En tirant parti des hooks intégrés de React (comme `useState()`, `useEffect()`, etc.), vous pouvez créer des hooks qui encapsulent une logique spécifique et la partagent facilement dans votre application. Cela favorise la modularité et la maintenabilité du code.

Un hook personnalisé est simplement une fonction qui commence par `use` et qui utilise au moins un des hooks intégrés de React. Par exemple, un hook qui gère la logique d'une API, la validation d'un formulaire, ou le suivi de l'état d'un composant peut être considéré comme un hook personnalisé. Ils permettent d'abstraire des logiques complexes en des fonctions plus simples à utiliser.

Voici un exemple d'un hook personnalisé appelé `useFetch()`, qui peut être utilisé pour récupérer des données à partir d'une API :

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

function useFetch(url){
  const [donnée, setDonnée] = useState(null);
  const [chargement, setChargement] = useState(true);
  const [erreur, setErreur] = useState(null);

  useEffect(() => {
    const fetchDonnées = async () => {
      try {
        const reponse = await fetch(url);
        if(!reponse.ok){
          throw new Error('Erreur lors de la récupération des données');
        }
        const resultat = await reponse.json();
        setDonnées(resultat);
      } catch(erreur){
        setErreur(erreur.message);
      } finally {
        setChargement(false);
      }
    };

    fetchDonnées();
  }, [url]); //Récupérer les données à chaque changement de l'URL
```

```
  return {donnée, chargement, erreur};

}
```

*Utilisation du Hook `useFetch()` :

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

function AfficheurDonnées(){
  const {donnée, chargement, erreur} = useFetch('https://api.exemple.com/data');

  if(chargement) return <p>Chargement...</p>;
  if(erreur) return <p>Erreur : {erreur}</p>;

  return (
    <div>
      <h1>Données récupérées :</h1>
      <pre>{JSON.stringify(donnée, null, 2)}</pre>
    </div>
  );
}
```

`useFetch()` prend une URL en entrée, gère l'état de chargement, les données récupérées, et les erreurs. `useEffect()` effectue la récupération des données chaque fois que l'URL change. Les composants peuvent facilement accéder aux données, à l'état de chargement, et aux erreurs grâce aux valeurs retournées.

En bref,

- Nommez vos Hooks : Commencez toujours le nom de votre fonction par `use` pour respecter la convention de React.
- Restez Simple : Un hook doit avoir une responsabilité claire et être aussi simple que possible. Évitez d'ajouter trop de logique dans un seul hook.
- Utilisez des Hooks Intégrés : N'hésitez pas à combiner plusieurs hooks intégrés dans votre hook personnalisé.
- Gestion des Erreurs : Ajoutez une gestion des erreurs pour que les composants qui utilisent votre hook puissent réagir en conséquence.
- Tests : Écrivez des tests pour vos hooks personnalisés afin de vous assurer qu'ils fonctionnent comme prévu.

CHAPITRE 14 : PENSER EN REACT

Lorsque vous travaillez sur la création d'une page web avec React, la première étape est de collaborer avec deux équipes clés : l'équipe de design (UI/UX) et l'équipe backend. Ces deux équipes vous fournissent les éléments nécessaires suivants pour transformer une idée en page fonctionnelle :

1. Le **Prototype** : L'équipe de web design crée le prototype de la page que vous devez développer. Ce prototype définit l'apparence de la page, son organisation, et l'expérience utilisateur. Il est souvent réalisé à l'aide d'outils comme Figma ou Adobe XD, qui permettent de concevoir les interfaces graphiques et de montrer à quoi doit ressembler l'application finale. Par exemple, dans Figma, vous pouvez avoir une page web composée d'une barre de navigation, d'une section d'affichage de produits, et d'un formulaire de contact. Chaque élément visuel du prototype correspondra à un composant que vous créerez dans React.
2. Les **API Backend** : Ensuite, vous avez besoin de la spécification des API que vous utiliserez pour interagir avec les données stockées dans le backend. L'équipe backend développe ces API et les documente souvent à l'aide de standards comme **Swagger** ou **OpenAPI**. Ce document vous donne tous les détails dont vous avez besoin :
   - Les endpoints des API (comme `/api.exemple.com/produits` pour récupérer les produits à afficher).
   - Les méthodes HTTP à utiliser (`GET`, `POST`, `PUT`, `DELETE`).

- Le format des données (par exemple, JSON avec des propriétés comme `id`, `nom`, `prix` pour un produit).

Cette spécification peut être partagée sous forme de fichier PDF généré à partir de Swagger, ou directement accessible via un lien web (ex :

```
[
  {
    "id": 1,
    "nom": "Produit A",
    "prix": 29.99
  },
  {
    "id": 2,
    "nom": "Produit B",
    "prix": 19.99
  }
]
```

). Une fois ces éléments en main, vous pouvez vous concentrer sur le développement en React.

ÉTAPE 1 : DECOUPER L'INTERFACE EN COMPOSANTS

La première étape pour penser en React est de diviser l'interface en composants réutilisables et indépendants. Chaque section de la page (boutons, formulaires, cartes de produit, etc.) doit être vue comme un composant (ex : Prenons une page de recherche de produits. On peut la décomposer en 3 composants :

- Un composant pour le champ de recherche.
- Un composant pour la liste des résultats.
- Un composant pour chaque produit dans la liste.

). Le découpage en composants vous permet de mieux gérer votre application et de réutiliser les mêmes éléments ailleurs.

ÉTAPE 2 : CONSTRUIRE UNE VERSION STATIQUE DE L'INTERFACE

Une fois que vous avez identifié vos composants, construisez une version statique de l'interface. À ce stade, vous ne gérez pas encore les interactions ou les changements d'état. Vous utilisez simplement les props pour faire passer les données d'un composant parent à ses enfants. Cette étape vous permet de vous concentrer uniquement sur la structure visuelle de votre application (ex : Dans la page de recherche de produits, vous passez une liste fictive de produits en props au composant qui affiche les résultats. Cela vous permet de visualiser le rendu sans vous soucier des fonctionnalités dynamiques).

ÉTAPE 3 : IDENTIFIER L'ETAT MINIMAL NECESSAIRE

Dans React, l'état (state) représente les données qui peuvent changer au cours du temps. Vous devez maintenant réfléchir aux informations qui devront changer dans votre application, et les gérer avec le state. Voici quelques questions à vous poser pour identifier l'état nécessaire :

- Quelles données vont changer ?
- Qui en est responsable ?
- Quelles interactions d'utilisateur affecteront ces données ?

(ex : Dans votre page de recherche de produits, l'état inclut : Le texte saisi dans la barre de recherche ; La liste des produits retournée par l'API après la recherche).

ÉTAPE 4 : DETERMINER OU L'ETAT DOIT ETRE STOCKE

Maintenant que vous avez identifié les états, vous devez déterminer quel composant est responsable de chaque état. Il est essentiel de positionner les états de manière hiérarchique, c'est-à-dire dans le composant le plus élevé qui en a besoin (ex : Le texte de recherche peut être géré dans le composant de la barre de recherche, mais la liste des produits doit être gérée dans un composant parent, car elle sera affichée dans plusieurs composants enfants (par exemple, la liste de résultats et les filtres)).

ÉTAPE 5 : AJOUTER LE FLUX DE DONNEES INVERSE

Les données en React circulent principalement du parent vers l'enfant via les props. Cependant, lorsque l'enfant doit envoyer des données au parent (comme dans un formulaire), vous devez utiliser un callback (fonction de rappel). Le parent transmet une fonction à l'enfant via les props, et l'enfant appelle cette fonction lorsque nécessaire pour informer le parent des changements.

(ex : Lorsque l'utilisateur saisit un texte dans la barre de recherche, la barre de recherche doit informer le parent (composant qui gère l'état de la liste de produits) que la requête a changé, afin qu'il puisse envoyer une nouvelle requête à l'API et mettre à jour les résultats affichés).

ÉTAPE 6 : INTEGRER L'API BACKEND

Maintenant que l'interface fonctionne bien et que l'état est géré efficacement, il est temps d'intégrer les données réelles à partir de l'API backend. Utilisez des appels HTTP (fetch ou des bibliothèques comme Axios) pour interagir avec les endpoints fournis par l'équipe backend.

(ex : Lorsqu'un utilisateur soumet une recherche, vous envoyez une requête GET à l'API pour récupérer les produits correspondants, puis vous mettez à jour l'état du composant parent pour réafficher les résultats avec les données réelles).

ÉTAPE 7 : EFFECTUER DES TESTS UNITAIRES AVEC **JEST**

Les tests unitaires sont une méthode de vérification du bon fonctionnement d'une unité de code, comme un composant React, de manière isolée. Cela permet de s'assurer que chaque partie de votre application fonctionne comme prévu, même après des modifications ou des ajouts de code. **Jest** est un framework de test populaire conçu spécifiquement pour les applications JavaScript, y compris celles utilisant React.

Jest est un outil de test JavaScript développé par Facebook, qui offre une série de fonctionnalités pratiques pour tester des applications React.

1. Installation : Pour utiliser Jest avec React, vous devez d'abord l'installer ainsi que la bibliothèque **React Testing Library**, qui facilite les tests des composants React. Dans votre terminal, exécutez la commande suivante :
   `npm install --save-dev jest @testing-library/react @testing-library/jest-dom`
   Cette commande installe Jest et les bibliothèques nécessaires pour les tests.
2. Configuration de Jest : Ensuite, vous devez configurer Jest pour votre projet. Ouvrez `package.json` et ajoutez un script pour exécuter les tests :
   ```
   "scripts": {
     "test": "jest"
   }
   ```
   Cela vous permettra de lancer vos tests en utilisant la commande `npm test`.

Pour illustrer l'écriture d'un test unitaire, considérons un exemple d'un composant simple, un bouton :

```
//Bouton.js
import React from 'react';

export const Bouton = ({label}) => {
  return <button>{label}</button>;
};
```

Pour tester ce composant, créez un fichier nommé Bouton.test.js dans le répertoire `__test__` :

```
//Bouton.test.js
import React from 'react';
import {render, screen} from '@testing-library/react';
import {Bouton} from '../Bouton';

test('affiche le label correct', () => {
```

```
  render(<Bouton label="Cliquez ici" />);
  expect(screen.getByText('Cliquez ici')).toBeInTheDocument();
});
```

Vous importez React, les fonctions `render()` et screen de `@testing-library/react`, et le composant que vous souhaitez tester. La fonction test prend deux paramètres : une description du test et une fonction qui contient le code du test. La fonction `render()` monte le composant dans un environnement de test virtuel. L'utilisation de `expect()` permet de vérifier si le texte `'Cliquez ici'` est présent dans le document, grâce à la méthode `toBeInTheDocument()` fournie par `@testing-library/jest-dom`.

Il est important de tester comment les utilisateurs interagissent avec vos composants. (ex : si vous souhaitez tester un bouton qui déclenche une fonction lors d'un clic :

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

//Bouton.test.js
test('déclenche la fonction onClick', () => {
  const fonctionSimulee = jest.fn(); //Crée une fonction simulée
  render(<Bouton label="Cliquez" onClick={fonctionSimulee} />);
  screen.getByText('Cliquez').click(); //Simule le clic sur le bouton
  //Vérifie que la fonction a été appelée une fois
  expect(fonctionSimulee).toHaveBeenCalledTimes(1);
});
```

). `jest.fn()` crée une fonction simulée qui peut être utilisée pour suivre si elle a été appelée. `click()` simule un clic sur le bouton, ce qui déclenche la fonction `onClick`. On vérifie que la fonction simulée a été appelée une fois lors de l'événement de clic.

Si votre composant effectue des opérations asynchrones, comme récupérer des données depuis une API, vous devez tester ces cas également (ex :

```
//RecupererDonnees.js
import React, {useEffect, useState} from 'react';

export const RecupererDonnees = () => {
  const [donnees, setDonnees] = useState(null);

  useEffect(() => {
    const recupererDonnees = async () => {
      const reponse = await fetch('/api/donnees');
      const resultat = await reponse.json();
      setDonnees(resultat);
    };
    recupererDonnees();
  }, []);

  return <div>{donnees ? donnees.message : 'Chargement...'}</div>;
};

//RecupererDonnees.test.js
test('affiche les données après le chargement', async () => {
  global.fetch = jest.fn(() =>
    Promise.resolve({
      json: () => Promise.resolve({message: 'Données chargées'}),
    })
  );

  render(<RecupererDonnees />);
  expect(screen.getByText('Chargement...')).toBeInTheDocument();

  //Attendez que le message chargé apparaisse
```

```
  const messageCharge = await screen.findByText('Données chargées');
  expect(messageCharge).toBeInTheDocument(); //Vérifiez que le message est présent
});
```

). Vous créez une version simulée de la fonction `fetch()` pour éviter des requêtes réelles lors des tests. `await screen.findByText()` est utilisé pour attendre que le message `'Données chargées'` apparaisse dans le DOM, essentiel pour les opérations asynchrones.
La couverture de code est une mesure utile pour savoir quelle portion de votre code est testée. Pour générer un rapport de couverture, exécutez la commande suivante : **`npm test -- --coverage`**

Cela affichera un rapport indiquant le pourcentage de lignes de code, de fonctions et de branches couvertes par vos tests. Cela vous aide à identifier les zones à améliorer.

En bref,

- Utilisez des noms de tests explicites qui décrivent ce que le test est censé faire.
- Évitez les dépendances entre les tests pour garantir leur indépendance.
- Lors de tests d'interactions ou d'appels API, utilisez des fonctions simulées pour contrôler le comportement des dépendances.
- Visez une couverture de code élevée, mais ne sacrifiez pas la qualité des tests pour obtenir des chiffres.

Pour approfondir vos connaissances, consultez la documentation officielle de Jest et la documentation de React Testing Library : https://jestjs.io/fr/docs/getting-started.

Voici quelques bibliothèques React que vous pouvez utiliser pour améliorer vos UI :

- **React Skeleton**
  - Utilisation : Afficher des placeholders animés en attendant le chargement du contenu.
  - Commande : **`npm install react-loading-skeleton`**
- **React Spinners**
  - Utilisation : Ajouter des animations de chargement élégantes (spinners).
  - Commande : **`npm install react-spinners`**
- **React Icons**
  - Utilisation : Large collection d'icônes populaires pour les applications React.
  - Commande : **`npm install react-icons`**
- **Framer Motion**
  - Utilisation : Animer des composants React avec des transitions fluides.
  - Commande : **`npm install framer-motion`**
- **React Toastify**
  - Utilisation : Notifications toast (petites popups) faciles à configurer.
  - Commande : **`npm install react-toastify`**
- **React Lottie**
  - Utilisation : Utiliser des animations Lottie (JSON) dans vos applications React.
  - Commande : **`npm install lottie-react`**
- **React Tooltip**
  - Utilisation : Ajouter des tooltips élégants et personnalisables.
  - Commande : **`npm install react-tooltip`**
- **React Spring**
  - Utilisation : Créer des animations fluides pour vos composants React.
  - Commande : **`npm install react-spring`**
- **React Select**
  - Utilisation : Composant de sélection avancé et personnalisable avec recherche.
  - Commande : **`npm install react-select`**
- **React Slick**
  - Utilisation : Carrousels et sliders responsive pour vos interfaces.
  - Commande : **`npm install react-slick slick-carousel`**
- **React Modal**
  - Utilisation : Créer des modales ou dialogues modaux facilement.
  - Commande : **`npm install react-modal`**
- **React Image Gallery**
  - Utilisation : Composant de galerie d'images responsive et interactif.

    - Commande : `npm install react-image-gallery`
- **React DnD (Drag and Drop)**
    - Utilisation : Ajouter des fonctionnalités de glisser-déposer (drag and drop).
    - Commande : `npm install react-dnd`
- **React Beautiful DnD**
    - Utilisation : Drag and drop avec animations et gestion d'éléments ordonnables.
    - Commande : `npm install react-beautiful-dnd`
- **React Virtualized**
    - Utilisation : Optimiser le rendu de listes ou tableaux volumineux avec virtualisation.
    - Commande : `npm install react-virtualized`
- **React Infinite Scroll Component**
    - Utilisation : Scrolling infini pour charger plus de contenu dynamiquement.
    - Commande : `npm install react-infinite-scroll-component`
- **React Table**
    - Utilisation : Composant de tableau flexible et performant.
    - Commande : `npm install @tanstack/react-table`
- **React Query**
    - Utilisation : Gestion efficace des données serveur (fetch, cache, synchronisation).
    - Commande : `npm install @tanstack/react-query`
- **React Hook Form**
    - Utilisation : Gérer les formulaires avec validation et gestion des données.
    - Commande : `npm install react-hook-form`
- **Formik**
    - Utilisation : Gérer et valider les formulaires React de manière intuitive.
    - Commande : `npm install formik`
- **Yup**
    - Utilisation : Valider des objets JavaScript, souvent utilisé avec Formik.
    - Commande : `npm install yup`
- **React Datepicker**
    - Utilisation : Sélection de dates avec un datepicker simple et élégant.
    - Commande : `npm install react-datepicker`
- **React Time Picker**
    - Utilisation : Sélectionneurs d'heures personnalisables pour vos formulaires.
    - Commande : `npm install react-time-picker`
- **React Window**
    - Utilisation : Rendu optimisé de listes volumineuses en utilisant le "windowing".
    - Commande : `npm install react-window`
- **React Flip Move**
    - Utilisation : Animer des éléments qui changent d'ordre dans une liste.
    - Commande : `npm install react-flip-move`
- **React Paginate**
    - Utilisation : Pagination simple et élégante pour les listes.
    - Commande : `npm install react-paginate`
- **React Hotkeys**
    - Utilisation : Ajouter des raccourcis clavier personnalisés à votre interface.
    - Commande : `npm install react-hotkeys-hook`
- **React Loader Spinner**
    - Utilisation : Afficher des spinners et des loaders d'attente.
    - Commande : `npm install react-loader-spinner`
- **React Content Loader**
    - Utilisation : Créer des animations de chargement personnalisées (placeholders skeleton).
    - Commande : `npm install react-content-loader`
- **React Responsive**
    - Utilisation : Composants réactifs pour gérer les points de rupture (breakpoints).
    - Commande : `npm install react-responsive`

Bon ! moi, je m'arrête là. Libre à vous de découvrir d'autres bibliothèques ou carrément apprendre des frameworks React comme Remix ou **Next.Js**.