Développer en Front-end

ReactJS

Réalisé par: Said MAHDAD

2024/2025

Introduction à React

1- Qu'est-ce que React ?

React est une **bibliothèque JavaScript** open-source, créée en 2013 par Facebook (désormais Meta). Elle est utilisée pour développer des **interfaces utilisateur (UI)** interactives et dynamiques pour des applications web.

React se concentre sur la **vue** (le "V" dans le modèle MVC) et permet aux développeurs de créer des composants réutilisables qui gèrent leur propre état.

2- Pourquoi React?

React est rapidement devenu l'une des bibliothèques JavaScript les plus utilisées grâce à ses nombreux avantages :

- **Performance élevée :**
  - Grâce au DOM virtuel, React minimise le temps de rendu en ne mettant à jour que les parties modifiées.
- **Modularité :**
  - La division en composants permet de créer des applications organisées, où chaque composant peut être développé et testé indépendamment.
- **Grande communauté :**
  - Avec des millions de développeurs à travers le monde, React dispose d'une énorme communauté. Cela signifie des ressources, des bibliothèques complémentaires (Material-UI, Redux, etc.) et un support constant.
- **Facilité d'apprentissage :**
  - React est basé sur JavaScript, ce qui en fait une excellente option pour les développeurs ayant déjà des connaissances en JavaScript.
- **Support des grandes entreprises :**
  - Facebook utilise React dans ses propres applications, y compris Instagram et WhatsApp. D'autres grandes entreprises, comme Netflix, Airbnb, et Uber, l'utilisent également.

3- Qui utilise React :

- **Meta :** L'interface principale de Facebook, Instagram et la version web de WhatsApp.
- **Netflix :** utilise React sur sa plateforme web suggestions de contenu.
- **Airbnb** a adopté React pour gérer ses composants dynamiques.
- **Uber** pour ses outils de gestion, notamment Uber Eats et certaines interfaces utilisateur
- **Shopify** utilise React pour améliorer ses interfaces administratives
- **Discord**, très populaire parmi les gamers, utilise React pour ses interfaces interactives
- **Dropbox** s'appuie sur React pour son interface web intuitive
- **Microsoft** utilise React pour des services comme Outlook Web App et OneDrive
- **Pinterest** utilise React pour gérer les tableaux d'épingles dynamiques
- **Twitter** a intégré React dans certaines parties de son site web
- **Reddit** a refait une grande partie de son site en utilisant React pour améliorer les performances et rendre la navigation plus fluide.
- **PayPal** a intégré React dans son interface utilisateur pour simplifier les paiements en ligne et offrir une expérience utilisateur moderne

Installation de React

1- Prérequis

Avant de commencer, assurez-vous d'avoir :

- **Node.js** installé sur votre ordinateur. Sinon télécharge le depuis ***https://nodejs.org/.***
- Node.js inclut aussi **npm** (Node Package Manager), qui est nécessaire pour installer des bibliothèques.
- Un éditeur de code, comme Visual Studio Code.

2- Vérification de l'installation

Ouvrez votre terminal (ou invite de commandes) et tapez :

```
node -v
```

Cela affichera la version de Node.js installée.

```
npm -v
```

Cela affichera la version de npm installée.
Si ces commandes ne fonctionnent pas, installez Node.js avant de continuer.

3- Création d'un projet

React propose une commande simple pour générer un projet de base avec une structure prête à l'emploi.
Dans le terminal, exécutez la commande suivante :

```
npx create-react-app mon_projet
```

- **npx :** Exécute un package sans l'installer globalement.
- **create-react-app :** Génère un projet React avec une configuration par défaut.
- **mon_projet :** Nom du dossier du projet (remplacez-le par le nom de votre choix).

⇨ Cette commande télécharge toutes les dépendances nécessaires et crée une structure de projet prête à l'emploi.

Une fois l'installation terminée, entrez dans le dossier du projet :

```
cd mon_ projet
```

Démarrez le serveur React pour voir votre application dans un navigateur :

```
npm start
```

Votre navigateur par défaut s'ouvrira automatiquement sur l'adresse : ***http://localhost:3000***.

Vous verrez une page avec le message "***Edit src/App.js and save to reload."*** Cela signifie que tout fonctionne !

Structure d'un projet React

Lorsque vous créez un projet React avec ***create-react-app***, une structure par défaut est générée. Voici la structure détaillée et son rôle pour un projet nommé ***mon_projet*** :

```
mon_projet/
├── public/            # Contient les fichiers publics
├── src/               # Contient le code source principal de l'application
├── node_modules/      # Contient les dépendances installées
├── .gitignore         # Liste des fichiers/dossiers ignorés par Git
├── package.json       # Liste des dépendances et scripts du projet
├── package-lock.json  # Verrouillage des versions des dépendances
├── README.md          # Documentation générée automatiquement
```

1- Le dossier public/

Contient des fichiers statiques accessibles directement depuis le navigateur.

```
public/
├── favicon.ico
├── index.html
├── logo192.png
├── logo512.png
├── manifest.json
├── robots.txt
```

- ***favicon.ico :*** Contient l'icône utilisée dans l'onglet du navigateur. Généralement, une petite image carrée (16x16 ou 32x32 pixels).
- ***index.html :*** Le point d'entrée HTML principal pour l'application React. Contient un élément où l'application React sera montée. Peut inclure des méta-informations spécifiques, comme le titre de la page, la description ou les scripts tiers.
- ***manifest.json :*** Décrit l'application pour les Progressive Web Apps (PWA). Il Spécifie les paramètres comme le nom, les icônes, et les couleurs.
- ***robots.txt :*** Contrôle l'accès des robots (comme Googlebot) au site. Par défaut, il empêche les moteurs de recherche de bloquer des fichiers spécifiques.

2- Le dossier src/

```
src/
├── App.css
├── App.js
├── App.test.js
├── index.css
├── index.js
├── logo.svg
├── reportWebVitals.js
└── setupTests.js
```

- ***App.js :*** Le composant principal de l'application. Il contient un exemple de structure de base React avec une simple interface utilisateur.

- ***App.css :*** Contient les styles CSS associés au composant App.js.
- ***App.test.js :*** Contient un exemple de test pour le composant App.js, en utilisant React Testing Library.
- ***index.js :*** Le point d'entrée principal de l'application. Il Monte l'application React sur l'élément DOM avec l'id root.
- ***index.css :*** Contient les styles globaux pour l'application.
- ***logo.svg :*** Une image SVG utilisée comme logo par défaut dans l'exemple fourni avec App.js.
- ***reportWebVitals.js :*** Un fichier optionnel pour mesurer les performances de l'application.
- ***setupTests.js :*** Contient la configuration initiale pour les tests. Il Ajoute automatiquement des configurations nécessaires pour React Testing Library.

3- Le dossier node_modules

Le dossier node_modules est automatiquement généré lorsqu'un projet ***Node.js*** (comme une application React) installe des dépendances via ***npm install*** ou ***yarn install***. Il contient toutes les bibliothèques et leurs dépendances nécessaires pour faire fonctionner le projet.

Bien que la structure interne de node_modules puisse varier en fonction des versions de ***Node.js*** et des gestionnaires de paquets

4- Le fichier .gitignore

Le fichier .gitignore est un élément fondamental dans tout projet versionné avec Git. Il permet de spécifier quels fichiers ou dossiers doivent être ignorés par Git, c'est-à-dire exclus du suivi dans le dépôt. Cela est particulièrement utile pour éviter d'ajouter des fichiers inutiles ou sensibles qui ne doivent pas être partagés ou versionnés.

5- Le fichier package.json

Le fichier package.json est un élément central de tout projet Node.js, y compris les applications React. Il sert de fichier de configuration pour décrire le projet, spécifier ses dépendances, et définir des scripts et des métadonnées.

⇨ Pour des projets complexes, il est préférable de structurer le dossier src/ pour mieux organiser le code. Voici une structure suggérée :

```
src/
├── components/          # Tous les composants réutilisables
│   ├── Header/
│   │   ├── Header.js
│   │   ├── Header.css
│   └── Footer/
│       ├── Footer.js
│       ├── Footer.css
├── pages/               # Pages principales de l'application
│   ├── Home.js
│   ├── About.js
│   └── Contact.js
├── services/            # Fichiers pour les appels API ou services externes
│   └── api.js
├── utils/               # Fonctions utilitaires réutilisables
│   └── helpers.js
├── styles/              # Fichiers de styles globaux
│   └── global.css
├── App.js               # Composant principal
├── index.js             # Point d'entrée de l'application
```

Première Application React

1- Le fichier App.js

Dans le dossier **src/** de votre projet, ouvre le fichier ***App.js*** , et saisir le code suivant :

```jsx
import React from 'react';
const App = () => {
    return (
        <div>
            <h1>Hello World</h1>
        </div>
    );
};
export default App;
```

Ce code affichera le texte suivant :

2- Le fichier index.js

Le fichier index.js servira simplement à monter l'application dans le DOM.

Exemple :

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

import React from 'react';

- Importe la bibliothèque React, qui est nécessaire pour écrire du code React.
- Bien que React ne soit pas directement utilisé dans ce fichier, il est souvent implicite pour interpréter le JSX (comme <App />).

import ReactDOM from 'react-dom/client';

- Importe ReactDOM, une bibliothèque permettant de monter un composant React dans le DOM.
- Le sous-module client est utilisé pour des fonctionnalités spécifiques aux applications React modernes (comme createRoot).

import './index.css';

- Importe un fichier CSS global (index.css) pour appliquer des styles à l'application.
- En React, les styles peuvent être appliqués via des fichiers CSS globaux ou via des approches plus modularisées comme les CSS Modules ou Styled Components.

import App from './App';

- Importe le composant principal de l'application depuis App.js.
- Ce composant est ensuite rendu dans le DOM via ReactDOM.

import reportWebVitals from './reportWebVitals';

- Importe une fonction reportWebVitals qui est utilisée pour mesurer les performances de l'application.
- Par défaut, create-react-app inclut ce fichier pour permettre de collecter des métriques, comme le temps de chargement initial, les interactions utilisateur, etc.
- Il peut être utilisé ou ignoré, selon les besoins du projet.

const root = ReactDOM.createRoot(document.getElementById('root'));

- Crée un point d'entrée React dans le DOM.
- document.getElementById('root') sélectionne l'élément HTML avec l'ID root (défini dans le fichier public/index.html).
- ReactDOM.createRoot prépare un conteneur pour les composants React. Cela est nécessaire pour utiliser la nouvelle API de rendu introduite avec React 18.

root.render(

- Monte l'application React dans le DOM en utilisant le conteneur défini précédemment (root).

<React.StrictMode>

- Un wrapper React pour détecter les problèmes potentiels dans le code.
- StrictMode n'affecte pas l'affichage de l'application, mais déclenche des avertissements dans la console pendant le développement pour signaler des pratiques non optimales.

<App />

- Le composant principal de l'application, qui est rendu dans le DOM.

Le SPA (Single Page Application)

Une SPA (Single Page Application) est une application web qui fonctionne à l'intérieur d'une seule page HTML, offrant une expérience utilisateur fluide et rapide.

1- Fonctionnement d'une SPA

- Chargement Initial :

Lorsqu'un utilisateur accède à l'application, une seule page HTML est chargée (souvent index.html dans React). Les scripts JavaScript gèrent tout le contenu et les interactions.

- Navigation Dynamique :

Contrairement aux applications web classiques où chaque clic sur un lien charge une nouvelle page depuis le serveur, les SPA manipulent le DOM dynamiquement pour afficher le contenu correspondant.

Par exemple, React Router permet de gérer la navigation sans recharger la page.

2- Avantages des SPA

- Performance : Les données sont récupérées via des APIs (souvent JSON), réduisant le besoin de recharger toute la page.
- Expérience Utilisateur : Une navigation rapide et fluide, similaire aux applications natives.
- Code Réutilisable : Les composants React favorisent la réutilisation et la modularité.

3- Inconvénients des SPA

- SEO : Le contenu est souvent rendu côté client, ce qui peut poser des défis pour les moteurs de recherche.
- Complexité : Nécessite des outils comme React, Redux, ou des gestionnaires de routage.

⇨ React est souvent utilisé pour créer des SPA

Le VDOM(Le Virtual DOM)

Le Virtual DOM (DOM virtuel) est un concept clé de React qui permet de rendre les interfaces utilisateur (UI) de manière performante et efficace. Il s'agit d'une abstraction du DOM réel qui réduit le coût des mises à jour du DOM en évitant les opérations inutiles.

1- Qu'est-ce que le DOM et ses Limitations ?

Le DOM(Document Object Model) est une représentation arborescente de la structure HTML d'une page. Il permet aux navigateurs de manipuler et d'afficher les éléments de la page.

Les problèmes du DOM classique :

- Les mises à jour du DOM sont lentes car elles nécessitent :
  - La modification de l'arbre DOM.
  - Le recalcul des styles CSS.
  - La re-rendu de la page par le navigateur (repaint ou reflow).
- Lorsqu'il y a plusieurs mises à jour, la performance peut en souffrir, surtout dans des applications complexes.

2- Le Concept du Virtual DOM

Le Virtual DOM est une copie légère et abstraite du DOM réel. React utilise cette copie pour déterminer exactement quelles parties du DOM doivent être modifiées, minimisant ainsi les opérations coûteuses sur le DOM réel.

Comment fonctionne le Virtual DOM ?

- **Initialisation :** Lorsque React charge pour la première fois, il crée un arbre de DOM virtuel correspondant au DOM réel.
- **Modification :** Lorsqu'un composant React change d'état ou de propriétés, React met à jour le DOM virtuel au lieu de modifier directement le DOM réel.
- **Diffing :** React compare (diff) l'ancien DOM virtuel avec le nouveau pour détecter les différences.
- **Patch (Reconciling) :** React applique uniquement les modifications nécessaires au DOM réel.

3- Avantages du Virtual DOM

- **Performance Améliorée** : Au lieu de mettre à jour tout le DOM réel, React effectue un minimum de modifications, ce qui améliore les performances.
- **Mise à Jour Optimisée** : Les mises à jour complexes sont gérées efficacement en regroupant les modifications dans des "lots".
- **Simplification du Code** : Les développeurs écrivent du code déclaratif (spécifient ce qu'ils veulent afficher, et non comment l'afficher), React gérant les mises à jour du DOM.

Les composants(compenents)

Un composant dans React est une unité autonome qui représente une partie d'interface utilisateur (UI). Il s'agit d'un bloc réutilisable de code qui gère son propre état et sa logique, et qui peut recevoir des données externes via des props. Un composant peut être une fonction ou une classe et est utilisé pour découper l'interface en éléments modulaires et réutilisables. Chaque composant retourne un JSX (un mélange de JavaScript et de HTML) qui définit ce qui doit être affiché à l'écran.

Les composants peuvent être imbriqués, ce qui permet de créer des interfaces complexes à partir de petites pièces indépendantes.

Les avantage des composants

- **Réutilisabilité** : Tu peux créer un composant une fois et l'utiliser à plusieurs endroits dans ton application.
- **Organisation** : Les composants aident à organiser ton code. Tu peux diviser une application complexe en plusieurs petits composants faciles à comprendre et à maintenir.
- **Abstraction** : Chaque composant peut fonctionner indépendamment des autres. Tu peux travailler sur un composant sans affecter le reste de l'application.

1- les composants de type classe

Les composants de classe sont des classes qui étendent React.Component et utilisent une méthode render() pour retourner du JSX. Ils peuvent avoir un état interne via this.state et utiliser des méthodes du cycle de vie de React pour gérer des opérations avant ou après le rendu.

```
import React, { Component } from 'react';
class Class_name extends Component {
    render() {
        return ;
    }
}
```

Exemple :

```
import React, { Component } from 'react';
class HelloWorld extends Component {
    render() {
        return <h1>Bonjour, le monde !</h1>;
    }
}
```

⇨ Hello World

2- Les composants fonctionnels

Les composants fonctionnels sont des fonctions JavaScript simples qui renvoient du JSX.

```
import React from 'react';
const Function_name = () => {
    return  ;
};
```

Exemple :

```
import React from 'react';
const HelloWorld = () => {
    return <h1>Bonjour, le monde !</h1>;
};
```

⇨ Hello World

3- Les composants parents vs les composants enfants

3-1- le composant parent

Un composant parent peut être considéré comme un conteneur pour d'autres composants. Il gère la structure générale de l'interface utilisateur et organise les composants enfants à l'intérieur de lui.

-2- le composant enfant

Un composant enfant est un composant qui est inclus à l'intérieur d'un composant parent. Il est souvent responsable d'une section spécifique de l'interface utilisateur.

```
function Enfant1() {
    return <p>Ceci est le composant enfant 1.</p>;
  }
function Enfant2() {
    return <p>Ceci est le composant enfant 2.</p>;
  }
function Parent() {
    return (
      <div>
        <h1>Mon Application</h1>
        <Enfant1 />
        <Enfant2 />
      </div>
    );
  }
```

Le composant Parent est défini pour contenir une structure complète de l'interface utilisateur. Il
Retourne :

- Une balise comme conteneur principal.
- Une balise <h1> affichant le titre : "Mon Application".
- Les composants Enfant1 et Enfant2 inclus comme balises <Enfant1 /> et <Enfant2 />.

4- Création des éléments HTML

4-1- fonction JavaScript react.createElement

La fonction ***React.createElement*** est au cœur de React. Elle est utilisée pour créer des éléments React, qui sont des représentations immuables d'une interface utilisateur. Ces éléments sont ensuite utilisés par le DOM virtuel de React pour effectuer des mises à jour optimales dans le DOM réel.

Syntaxe:

```
React.createElement(
  type,
  [props],
  [...children]
)
```

- **type** (obligatoire) : Type de l'élément à créer. Peut-être une **chaîne de caractères** pour les balises HTML ('div', 'span', etc.) ou une **fonction/composant React** pour les composants personnalisés.
- **props** (facultatif) : Un objet contenant les propriétés (attributs) que vous souhaitez passer à l'élément.
- **children** (facultatif) : Les enfants à inclure à l'intérieur de l'élément. Peut-être une chaîne, un nombre, un autre élément React, ou un tableau d'éléments React

Exemple :

```
1  const element = React.createElement(
2    'h1',              // Type : balise HTML
3    { className: 'greeting' },// Props : un objet avec className
4    'Bonjour, monde !' // Children : texte "Bonjour, monde !"
5  );
```

⇨ Bonjour, monde !

Avantages de la fonction createElement

- Base de React : Toutes les interfaces utilisateur en React passent par React.createElement (même via JSX).
- Souplesse : Permet de construire dynamiquement des éléments et des interfaces.
- Structure explicite : Utile pour comprendre comment JSX est transformé en appels fonctionnels.

4-2- Le JSX

JSX (JavaScript XML) est une extension de syntaxe pour JavaScript qui permet d'écrire un code similaire à HTML dans les fichiers JavaScript. Il est utilisé dans React pour rendre le code plus lisible et pour décrire l'interface utilisateur.

JSX est une abstraction syntaxique qui simplifie la création des éléments React. Sous le capot, le code JSX est transformé en appels à React.createElement.

Exemple basique:

```
1  const element = <h1>Bonjour, monde !</h1>;
```

- **<h1>** : Une balise HTML.
- **Bonjour, monde !** : Contenu texte.
- **const element** : Une variable qui contient l'élément React généré.

4-3- Babel

Babel est un translateur JavaScript qui convertit le code écrit en JSX en code JavaScript standard compréhensible par les navigateurs.

Pourquoi Babel ?

- Les navigateurs ne comprennent pas JSX.
- Babel transforme le JSX en appels React.createElement.

Exemple :

```
1  const element = <h1 className="greeting">Bonjour, monde !</h1>;
```

⬇

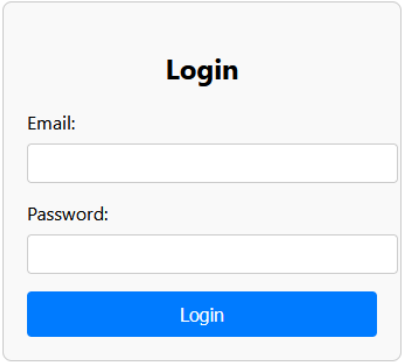
```
1  const element = React.createElement(
2    'h1',
3    { className: 'greeting' },
4    'Bonjour, monde !'
5  );
```

4-4- Exemple :

Formulaire avec code compilé

```
import React from 'react';
import './LoginForm.css';
function LoginForm() {
  return React.createElement(
    'div',
    { className: 'login-container' },
    React.createElement('h2', null, 'Login'),
    React.createElement('label', { htmlFor: 'email' }, 'Email:'),
    React.createElement('input', { type: 'email', id: 'email', name: 'email' }),
    React.createElement('label', { htmlFor: 'password' }, 'Password:'),
    React.createElement('input', { type: 'password', id: 'password', name: 'password' }),
    React.createElement('button', { type: 'submit' }, 'Login')
  );
}
export default LoginForm;
```

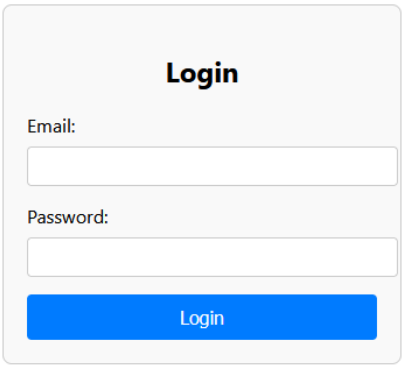
⇨



Formulaire avec JSX

```
import React from 'react';
function LoginForm() {
  return (
    <div className="login-container">
      <h2>Login</h2>
      <form>
        <label htmlFor="email">Email:</label>
        <input type="email" id="email" name="email" />
        <label htmlFor="password">Password:</label>
        <input type="password" id="password" name="password" />
        <button type="submit">Login</button>
      </form>
    </div>
  );
}
export default LoginForm;
```

⇨



5- Les fragments

En React, un Fragment est utilisé pour regrouper plusieurs éléments sans ajouter un nœud DOM supplémentaire. C'est une fonctionnalité utile pour optimiser la structure du DOM et éviter des conteneurs inutiles, tels que les balises imbriquées.

Pourquoi utiliser les Fragments ?

- Réduction des nœuds inutiles dans le DOM :
- Optimisation des performances : Les Fragments rendent le DOM plus léger en éliminant des nœuds superflus.
- Meilleure lisibilité et structure du code :Moins de balises imbriquées pour des composants complexes.

Exemple :

Avec React.Fragment

```jsx
import React from 'react';
function App() {
    return (
        <React.Fragment>
          <h1>Titre</h1>
          <p>Paragraphe</p>
        </React.Fragment>
      );
}
export default App;
```

⇨

Titre

Paragraphe

Syntaxe courte (< >) :

```jsx
import React from 'react';
function App() {
    return (
        <>
          <h1>Titre</h1>
          <p>Paragraphe</p>
        </>
      );
}
export default App;
```

⇨

Titre

Paragraphe

**Différence entre React.Fragment et les balises comme **

Aspect	Avec	Avec React.Fragment
DOM ajouté	Ajoute un conteneur au DOM.	Aucun conteneur ajouté.
Lisibilité	Peut créer une imbrication inutile.	Code plus propre et lisible.
Performances	Plus lourd (ajoute un nœud DOM).	Plus léger (aucun nœud supplémentaire).

Les propriétés(props)

Les ***props*** sont des informations que tu passes d'un composant parent à un composant enfant. Elles permettent de personnaliser un composant selon les besoins. Les ***props*** sont **immutables**, c'est-à-dire qu'un composant enfant ne peut pas les modifier.

1- Concept de Base

Les **props** sont des **paramètres** passés à un composant React. Elles sont **immutables**, ce qui signifie qu'un composant enfant ne peut pas modifier ses props directement. Elles sont accessibles dans un composant via l'objet props.

2- Pourquoi les props ?

- **Transmission de données :** Les props permettent de transmettre des données d'un composant parent à un composant enfant.
- **Immutabilité :** Les props sont immuables, ce qui signifie qu'un composant enfant ne peut pas les modifier. Cela aide à maintenir l'intégrité des données.
- **Facilite la réutilisation :** Les composants peuvent être réutilisés avec différentes données simplement en changeant les valeurs des props.

3- Passage des props

3-1- passage des chaines de caractères(String)

Composant de type classe

```jsx
import React from 'react';
function Greeting(props) {
    return <p>Bonjour, {props.name}!</p>;
  }
function App() {
    return <Greeting name="Said" />;
  }
export default App;
```

⇨ Bonjour, Said!

Composant fonctionnel

```jsx
import React from 'react';
function Greeting(props) {
  return <p>Bonjour, {props.name}!</p>;
}
function App() {
  return <Greeting name="Alice" />;
}
export default App;
```

⇨ Bonjour, Said!

3-2- passage des nombes(Number)

Composant de type classe

```jsx
import React from 'react';
class AgeDisplay extends React.Component {
    render() {
      return <p>Âge : {this.props.age}</p>;
    }
  }
  function App() {
    return <AgeDisplay age={25} />;
  }
export default App;
```

⇨ Âge : 25

Composant fonctionnel

```jsx
import React from 'react';
function AgeDisplay(props) {
    return <p>Âge : {props.age}</p>;
  }
  function App() {
    return <AgeDisplay age={25} />;
  }
export default App;
```

⇨ Âge : 25

3-3- passage d'un Booléen (boolean)

Composant de type classe

```jsx
import React from 'react';
class Status extends React.Component {
    render() {
      return <p>{this.props.isOnline ? 'En ligne' : 'Hors ligne'}</p>;
    }
  }
  function App() {
    return <Status isOnline={true} />;
  }
export default App;
```

⇨ En ligne

Composant fonctionnel

```jsx
import React from 'react';
function Status(props) {
    return <p>{props.isOnline ? 'En ligne' : 'Hors ligne'}</p>;
  }
  function App() {
    return <Status isOnline={true} />;
  }
export default App;
```

⇨ En ligne

3-4- Passage d'un Tableau (array)

Composant de type classe

```jsx
import React from 'react';
class HobbiesList extends React.Component {
    render() {
      return (
        <ul>
          {this.props.hobbies.map((hobby, index) => (
            <li key={index}>{hobby}</li>
          ))}
        </ul>
      );
    }
  }
  function App() {
    return <HobbiesList hobbies={['Lecture', 'Voyage', 'Sport']} />;
  }
export default App;
```

⇨

- Lecture
- Voyage
- Sport

Composant fonctionnel

```jsx
import React from 'react';
function HobbiesList(props) {
    return (
      <ul>
        {props.hobbies.map((hobby, index) => (
          <li key={index}>{hobby}</li>
        ))}
      </ul>
    );
  }
  function App() {
    return <HobbiesList hobbies={['Lecture', 'Voyage', 'Sport']} />;
  }
export default App;
```

⇨

- Lecture
- Voyage
- Sport

3-5- Passage d'un Objet (object)

Composant de type classe

```jsx
import React from 'react';
class UserProfile extends React.Component {
    render() {
      return <p>{this.props.user.name}: {this.props.user.job}.</p>;
    }
  }
  function App() {
    const user = { name: 'Mahdad', job: 'Développeur' };
    return <UserProfile user={user} />;
  }

export default App;
```

⇨ Mahdad: Développeur.

Composant fonctionnel

```jsx
import React from 'react';
function UserProfile(props) {
    return <p>{props.user.name}: {props.user.job}.</p>;
  }
  function App() {
    const user = { name: 'Mahdad', job: 'Développeur' };
    return <UserProfile user={user} />;
  }
export default App;
```

⇨ Mahdad: Développeur.

Les états(states)

Les états (states) permettent aux composants React de stocker des données dynamiques et de réagir aux changements dans ces données en mettant à jour l'interface utilisateur (UI). Contrairement aux props, qui sont immuables et transmises par les composants parents, les states sont internes à un composant et peuvent être modifiés localement.

Un état est généralement utilisé pour :

- Suivre les interactions utilisateur (par exemple, la saisie d'un formulaire).
- Gérer l'affichage conditionnel (par exemple, afficher ou masquer des éléments).
- Mettre à jour dynamiquement des données en fonction des actions utilisateur.

1- Déclaration et Initialisation du State:

Dans un composant React basé sur une classe, le state est déclaré et initialisé dans le constructeur. Voici un exemple simple :

```jsx
import React, { Component } from 'react';
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      compteur: 0
    };
  }
  render() {
    return (
      <div>
        <p>Le compteur est : {this.state.compteur}</p>
      </div>
    );
  }
}
export default App;
```

⇨ Le compteur est : 0

- **this.state** : L'objet state est initialisé avec une propriété compteur ayant une valeur de 0.
- **Rendu dynamique** : La valeur du compteur est affichée dynamiquement dans le DOM via this.state.compteur.

2- Modification du State avec setState

Pour modifier le state, React propose la méthode setState. Cette méthode accepte un objet ou une fonction comme argument pour déclencher une mise à jour.

```jsx
import React, { Component } from 'react';
class Compteur extends Component {
  constructor(props) {
    super(props);
    this.state = {
      compteur: 0
    };
  }
  incrementer = () => {
    this.setState({ compteur: this.state.compteur + 1 });
  };
  render() {
    return (
      <div>
        <p>Le compteur est : {this.state.compteur}</p>
        <button onClick={this.incrementer}>Incrémenter</button>
      </div>
    );
  }
}
export default Compteur;
```

⇨

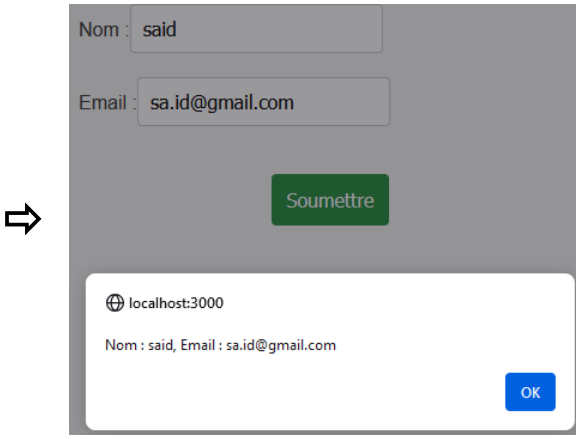
Le compteur est : 2

Incrémenter

- **Mise à jour asynchrone** : La méthode setState met à jour le state de manière asynchrone, ce qui peut occasionner des comportements imprévisibles si elle est appelée plusieurs fois consécutives.
- **Nouvelle valeur** : setState fusionne l'objet passé avec l'état actuel sans le remplacer complètement.
- **Gestion d'événement** : Le bouton appelle la fonction incrementer, qui met à jour le compteur.

3- Exemple complet

```jsx
import React, { Component } from 'react';
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      nom: '',
      email: ''
    };
  }
  handleChange = (event) => {
    const { name, value } = event.target;
    this.setState({ [name]: value });
  };
  handleSubmit = (event) => {
    event.preventDefault();
    alert(`Nom : ${this.state.nom}, Email : ${this.state.email}`);
  };
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nom :
          <input
            type="text"
            name="nom"
            value={this.state.nom}
            onChange={this.handleChange}
          />
        </label>
        <br />
        <label>
          Email :
          <input
            type="email"
            name="email"
            value={this.state.email}
            onChange={this.handleChange}
          />
        </label>
        <br />
        <button type="submit">Soumettre</button>
      </form>
    );
  }
}
export default App;
```



Remarques

- Asynchrone : La méthode setState est asynchrone. Si vous avez besoin de mettre à jour l'état en fonction de l'état précédent, utilisez une fonction au lieu d'un objet
- Fusion : React fusionne automatiquement les mises à jour de l'état. Vous pouvez mettre à jour partiellement l'état sans remplacer tout l'objet.

4- Le cycle de vie d'un composant

En React, un composant passe par plusieurs étapes au cours de son existence, généralement appelées cycle de vie du composant. L'état (state) joue un rôle central dans ce cycle, car il détermine en grande partie la manière dont un composant réagit et se met à jour. Comprendre le cycle de vie d'un état permet de mieux appréhender la gestion des changements dans une application React. Le cycle de vie d'un composant React. Il se décompose en trois grandes phases :

- Montage (Mounting),
- Mise à Jour (Updating),
- Démontage (Unmounting).

4-1- Phase de Montage

La phase de montage correspond au moment où un composant est inséré dans l'arbre DOM pour la première fois. Cette phase est essentielle pour initialiser l'état. Cette phase comprend les méthodes suivantes :

- **constructor** : Pour initialiser l'état.

```
constructor(props) {
  super(props);
  this.state = {
    count: 0,
  };
}
```

- **componentDidMount**: Pour effectuer des actions après le premier rendu.

```
componentDidMount() {
  setTimeout(() => {
    this.setState({ count: 10 });
  }, 1000);
}
```

La fonction ***componentDidMount***: Permet d'effectuer des opérations après le premier rendu, comme initialiser des données dynamiques qui pourraient modifier l'état.

4-2- Phase de Mise à Jour

Cette phase est déclenchée lorsque l'état ou les props d'un composant changent. Le changement d'état entraîne un nouveau rendu du composant. Cette phase comprend les méthodes suivantes :

- **setState** Pour mettre à jour l'état, React fournit la méthode setState. Cette méthode fusionne le nouvel état avec l'état actuel et déclenche une réévaluation de la méthode render.

```
this.setState({ count: this.state.count + 1 });
```

- **shouldComponentUpdate** (optionnelle) : Permet de contrôler si le composant doit être mis à jour après un changement d'état.

```
shouldComponentUpdate(nextProps, nextState) {
  return nextState.count !== this.state.count;
}
```

- **componentDidUpdate :** Méthode appelée juste après la mise à jour du DOM.

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    console.log('L'état a été mis à jour :', this.state.count);
  }
}
```

4-3- Phase de Démontage

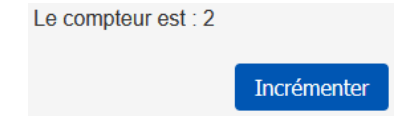
Lorsque le composant est retiré du DOM, il passe par une phase de démontage. Dans cette phase, l'état est détruit avec le composant.

- **componentWillUnmount** : Utilisée pour effectuer un nettoyage, comme l'annulation de minuteries ou de souscriptions qui pourraient dépendre de l'état.

```
componentWillUnmount() {
  console.log('Le composant est démonté, l'état est supprimé.');
}
```

Exemple complet :

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
    this.increment = this.increment.bind(this);
  }
  componentDidMount() {
    console.log('Composant monté avec un état initial :', this.state.count);
  }
  componentDidUpdate(prevProps, prevState) {
    console.log(`L'état a changé de ${prevState.count} à ${this.state.count}`);
  }
  componentWillUnmount() {
    console.log('Composant démonté, état supprimé.');
  }
  increment() {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>Compteur : {this.state.count}</p>
        <button onClick={this.increment}>Incrémenter</button>
      </div>
    );
  }
}
export default Counter;
```



⇨

Console :

```
Composant monté avec un état initial : 0
Composant démonté, état supprimé.
Composant monté avec un état initial : 0
L'état a changé de 0 à 1
L'état a changé de 1 à 2
```

Les hooks(states)

Les Hooks sont Introduits dans React 16.8, ils permettent d'utiliser les fonctionnalités de React comme l'état et le cycle de vie dans des composants fonctionnels, sans avoir besoin de créer des classes. Cela rend le code plus concis, réutilisable et facile à comprendre.
Un Hook est une fonction spéciale de React qui commence toujours par le mot use (ex. : useState, useEffect). Il permet de « brancher » (hook into) des fonctionnalités de React.

Principaux Types de Hooks

- useState : Pour gérer l'état dans un composant fonctionnel.
- useEffect : Pour gérer les effets de bord comme les abonnements, les minuteries, ou les appels API.
- useContext : Pour utiliser un contexte React sans composant de classe.
- Autres hooks comme useReducer, useMemo, useRef, etc., permettent de gérer des scénarios plus avancés.

1- Le hook useState:

useState permet d'ajouter un état à un composant fonctionnel. Contrairement à this.state dans les classes, useState retourne un tableau contenant :

- La valeur actuelle de l'état.
- Une fonction pour mettre à jour cet état.

Exemple Simple : Un Compteur

```jsx
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrémenter</button>
    </div>
  );
}
export default Counter;
```

⇨

- useState(0) initialise count à 0.
- setCount est utilisé pour mettre à jour l'état.
- Chaque fois que le bouton est cliqué, l'état est mis à jour et le composant se réaffiche.

2- Le hook useEffect

useEffect remplace les méthodes de cycle de vie comme componentDidMount, componentDidUpdate, et componentWillUnmount. Il s'utilise pour effectuer des tâches comme :

- Appeler une API.
- Ajouter/retirer des abonnements.
- Mettre en place des minuteries.

Le Hook useEffect permet de gérer des effets de bord dans les composants fonctionnels. Il accepte deux paramètres principaux :

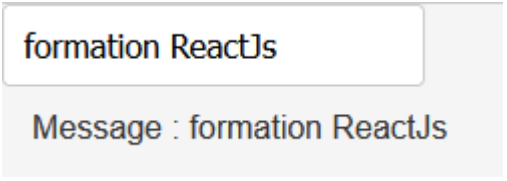
- Une fonction d'effet (obligatoire) : La fonction qui contient le code de l'effet à exécuter (contient la logique à exécuter après le rendu du composant) . Cette fonction peut également retourner une autre fonction utilisée pour nettoyer l'effet.

- Un tableau de dépendances (optionnel) : Une liste de valeurs qui déterminent quand l'effet doit être exécuté.
    - Vide ([ ]) : L'effet s'exécute une seule fois, après le premier rendu.
    - Non fourni (par défaut) : L'effet s'exécute après chaque rendu du composant.
    - Avec des dépendances : L'effet s'exécute uniquement lorsque l'une des dépendances change.

Exemple Simple : Afficher un Message à Chaque Mise à Jour

```jsx
import React, { useState, useEffect } from 'react';
function App() {
  const [message, setMessage] = useState('');
  useEffect(() => {
    console.log('Le message a changé :', message);
  }, [message]);
  return (
    <div>
      <input
        type="text"
        value={message}
        onChange={(e) => setMessage(e.target.value)}
        placeholder="Tapez un message"
      />
      <p>Message : {message}</p>
    </div>
  );
}
export default App;
```

⇨



Console

```
Le message a changé : <empty string>
Le message a changé : f
Le message a changé : fo
Le message a changé : for
Le message a changé : forl
Le message a changé : for
Le message a changé : form
Le message a changé : forma
Le message a changé : format
Le message a changé : formati
Le message a changé : formatio
Le message a changé : formation
Le message a changé : formation
Le message a changé : formation R
Le message a changé : formation Re
Le message a changé : formation Rea
Le message a changé : formation Reac
Le message a changé : formation React
Le message a changé : formation ReactJ
Le message a changé : formation ReactJs
```

useEffect s'exécute après chaque rendu si le message change (dépendance [message]).

3- Le hook useReducer

useReducer est un Hook de React utilisé pour gérer des états complexes. Il est similaire à useState, mais offre une structure plus adaptée lorsque l'état implique plusieurs sous-valeurs ou dépend d'opérations plus sophistiquées.

useReducer est basé sur le concept de la fonction reducer couramment utilisé avec des bibliothèques comme Redux. Il accepte trois paramètres principaux :

- Un réducteur (reducer) : Une fonction qui détermine comment l'état est mis à jour en fonction de l'action.
- Un état initial : La valeur initiale de l'état.
- (Optionnel) Un initialisateur : Une fonction pour initialiser l'état.

Syntaxe de base

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

- state : L'état actuel.
- dispatch : Une fonction pour envoyer des actions au réducteur.
- reducer : Une fonction qui prend l'état actuel et une action, et retourne le nouvel état.
- initialState : La valeur initiale de l'état.

Exemple Simple : Compteur avec useReducer

```jsx
import React, { useReducer } from 'react';
function counterReducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: 0 };
    default:
      throw new Error(`Action inconnue : ${action.type}`);
  }
}
function App() {
  const initialState = { count: 0 };
  const [state, dispatch] = useReducer(counterReducer, initialState);
  return (
    <div>
      <p>Compteur : {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Incrémenter</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Décrémenter</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Réinitialiser</button>
    </div>
  );
}
export default App;
```

⇨

Consommation des API

La consommation d'API est une tâche courante dans le développement React pour récupérer des données depuis un serveur. Deux outils populaires pour ce faire sont :

- **Fetch API :** Une API native de JavaScript pour effectuer des requêtes HTTP.
- **Axios :** Une bibliothèque tierce qui offre une syntaxe simplifiée et des fonctionnalités avancées.

1- Consommation des API avec Fetch

fetch est une fonction native qui retourne une promesse. Voici sa syntaxe générale :

```
fetch(url, options)
  .then(response => response.json())
  .then(data => {
    // Traiter les données ici
  })
  .catch(error => {
    // Gérer les erreurs ici
  });
```

1-1- Exemple 1 : Récupérer des Données

```
1  import React, { useEffect, useState } from 'react';
2  function FetchExample() {
3    const [data, setData] = useState([]);
4    const [error, setError] = useState(null);
5    useEffect(() => {
6      fetch('https://jsonplaceholder.typicode.com/posts')
7        .then((response) => {
8          if (!response.ok) {
9            throw new Error('Erreur de réseau');
10         }
11         return response.json();
12       })
13       .then((data) => setData(data))
14       .catch((error) => setError(error.message));
15   }, []);
16   return (
17     <div>
18       {error && <p>Erreur : {error}</p>}
19       <ul>
20         {data.map((item) => (
21           <li key={item.id}>{item.title}</li>
22         ))}
23       </ul>
24     </div>
25   );
26 }
27 export default FetchExample;
```

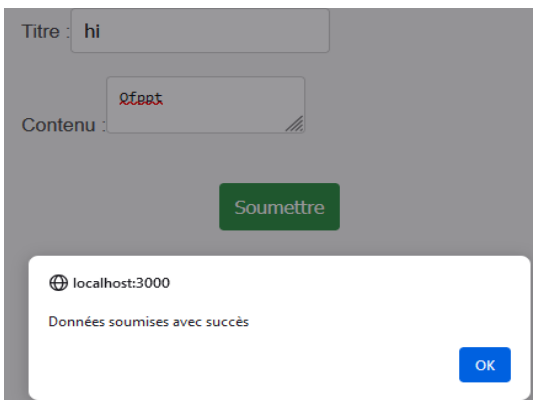
⇨

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio
- dolorem eum magni eos aperiam quia
- magnam facilis autem
- dolorem dolore est ipsam
- nesciunt iure omnis dolorem tempora et accusantium
- optio molestias id quia eum
- et ea vero quia laudantium autem
- in quibusdam tempore odit est dolorem
- dolorum ut in voluptas mollitia et saepe quo animi
- voluptatem eligendi optio
- eveniet quod temporibus
- sint suscipit perspiciatis velit dolorum rerum ipsa laboriosam odio
- fugit voluptas sed molestias voluptatem provident
- voluptate et itaque vero tempora molestiae
- adipisci placeat illum aut reiciendis qui
- doloribus ad provident suscipit at
- asperiores ea ipsam voluptatibus modi minima quia sint
- dolor sint quo a velit explicabo quia nam
- maxime id vitae nihil numquam

- Requête GET : fetch envoie une requête GET par défaut.
- Gestion des erreurs : Vérification manuelle de la réponse avec response.ok.
- État de la réponse : Les données récupérées sont stockées dans data grâce à setData.

1-2- Exemple 2 : Envoyer des Données (POST)

```jsx
import React, { useState } from 'react';
function PostDataExample() {
  const [title, setTitle] = useState('');
  const [body, setBody] = useState('');
  const handleSubmit = (event) => {
    event.preventDefault(); // Empêche le rechargement de la page
    const data = {
      title: title,
      body: body,
      userId: 1, // Exemple d'ID utilisateur statique
    };
    fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json', // Spécifie que les données sont en JSON
      },
      body: JSON.stringify(data), // Sérialise les données en JSON
    })
      .then((response) => {
        if (!response.ok) {
          throw new Error('Erreur de réseau');
        }
        return response.json();
      })
      .then((data) => {
        console.log('Données envoyées avec succès :', data);
        alert('Données soumises avec succès');
      })
      .catch((error) => {
        console.error('Erreur lors de l'envoi des données :', error);
      });
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Titre :
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
        />
      </label>
      <br />
      <label>
        Contenu :
        <textarea
          value={body}
          onChange={(e) => setBody(e.target.value)}
        ></textarea>
      </label>
      <br />
      <button type="submit">Soumettre</button>
    </form>
  );
}
export default PostDataExample;
```



■ **Gestion des Inputs** :
- Les états title et body capturent les données saisies par l'utilisateur.
- Les gestionnaires d'événements onChange mettent à jour les états respectifs.

■ **Requête POST** :
- La fonction handleSubmit prépare les données à envoyer sous forme d'objet data.
- fetch envoie ces données en tant que body après les avoir converties en JSON.

■ **Retour du Serveur** :
- Le serveur retourne une réponse qui est traitée via response.json().
- Les données retournées peuvent être affichées ou utilisées pour d'autres traitements.

1-3- Exemple 3 : Récupérer des Données depuis un Fichier JSON

```jsx
import React, { useEffect, useState } from 'react';
function FetchLocalJson() {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  useEffect(() => {
    fetch('data.json')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Erreur lors du chargement du fichier JSON');
        }
        return response.json();
      })
      .then((data) => setData(data))
      .catch((error) => setError(error.message));
  }, []);

  return (
    <div>
      {error && <p>Erreur : {error}</p>}
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}
export default FetchLocalJson;
```

⇨

- Mahdad
- kadiri
- Alami

Le fichier data.json (dans le dossier public)

```json
[
    { "id": 1, "name": "Mahdad" },
    { "id": 2, "name": "kadiri" },
    { "id": 3, "name": "Alami" }
]
```

■ Chemin d'accès : Le fichier JSON est accessible via data.json car il est placé dans le répertoire public.

■ Gestion des Erreurs : Si le fichier n'est pas trouvé ou si une autre erreur survient, elle est capturée et affichée.

■ Affichage des Données : Les données JSON sont stockées dans l'état data et affichées sous forme de liste.

2- Consommation des API avec axios

Axios est une bibliothèque basée sur des promesses qui simplifie les requêtes HTTP. Par rapport à Fetch, Axios offre :

- Une gestion automatique des erreurs HTTP.
- Un support pour la transformation des données.
- Une meilleure prise en charge des anciennes versions de navigateurs.

Avant d'utiliser Axios, installez-le avec npm :

```
npm install axios
```

2-1- Exemple 1 : Récupérer des Données

```jsx
import React, { useEffect, useState } from 'react';
import axios from 'axios';
function AxiosExample() {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  useEffect(() => {
    axios
      .get('https://jsonplaceholder.typicode.com/posts')
      .then((response) => setData(response.data))
      .catch((error) => setError(error.message));
  }, []);
  return (
    <div>
      {error && <p>Erreur : {error}</p>}
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}
export default AxiosExample;
```

⇨

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio
- dolorem eum magni eos aperiam quia
- magnam facilis autem
- dolorem dolore est ipsam
- nesciunt iure omnis dolorem tempora et accusantium
- optio molestias id quia eum
- et ea vero quia laudantium autem
- in quibusdam tempore odit est dolorem
- dolorum ut in voluptas mollitia et saepe quo animi
- voluptatem eligendi optio
- eveniet quod temporibus
- sint suscipit perspiciatis velit dolorum rerum ipsa laboriosam odio
- fugit voluptas sed molestias voluptatem provident
- voluptate et itaque vero tempora molestiae
- adipisci placeat illum aut reiciendis qui
- doloribus ad provident suscipit at
- asperiores ea ipsam voluptatibus modi minima quia sint
- dolor sint quo a velit explicabo quia nam
- maxime id vitae nihil numquam

- **Requête GET :** axios.get simplifie les requêtes GET.
- **Données dans response.data :** Axios retourne les données directement dans response.data.
- **Gestion des erreurs :** Les erreurs sont capturées dans catch.

2-2- Exemple 2 : Envoyer des Données (POST)

```jsx
import React, { useState } from 'react';
import axios from 'axios';
function PostWithAxios() {
  const [formData, setFormData] = useState({
    title: '',
    body: '',
  });
  const [responseMessage, setResponseMessage] = useState('');
  const [errorMessage, setErrorMessage] = useState('');
  const handleChange = (event) => {
    const { name, value } = event.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };
  const handleSubmit = (event) => {
    event.preventDefault();
    axios
      .post('https://jsonplaceholder.typicode.com/posts', formData)
      .then((response) => {
        console.log('Données envoyées :', response.data);
        setResponseMessage('Données soumises avec succès.');
        setErrorMessage('');
      })
      .catch((error) => {
        console.error('Erreur lors de la soumission :', error);
        setErrorMessage('Erreur lors de la soumission des données.');
        setResponseMessage('');
      });
  };
  return (
    <div>
      <h2>Soumettre un Post</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>
            Titre :
            <input
              type="text"
              name="title"
              value={formData.title}
              onChange={handleChange}
              required
            />
          </label>
        </div>
        <div>
          <label>
            Contenu :
            <textarea
              name="body"
              value={formData.body}
              onChange={handleChange}
              required
            ></textarea>
          </label>
        </div>
        <button type="submit">Envoyer</button>
      </form>
      {responseMessage && <p style={{ color: 'green' }}>{responseMessage}</p>}
      {errorMessage && <p style={{ color: 'red' }}>{errorMessage}</p>}
    </div>
  );
}
export default PostWithAxios;
```

2-3- Exemple 3 : Récupérer des Données depuis un Fichier JSON

```jsx
import React, { useEffect, useState } from 'react';
import axios from 'axios';
function FetchLocalJsonAxios() {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  useEffect(() => {
    axios
      .get('data.json')
      .then((response) => {
        setData(response.data);
      })
      .catch((error) => {
        setError(error.message);
      });
  }, []);
  return (
    <div>
      {error && <p>Erreur : {error}</p>}
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default FetchLocalJsonAxios;
```

⇨

- Mahdad
- kadiri
- Alami

Le fichier data.json (dans le dossier public)

```json
[
    { "id": 1, "name": "Mahdad" },
    { "id": 2, "name": "kadiri" },
    { "id": 3, "name": "Alami" }
]
```

Les événements et les formulaires

1- Les événements

Un événement est une action ou une occurrence détectée par le système ou le navigateur, généralement en réponse à une interaction de l'utilisateur ou à des changements dans l'environnement de l'application. Les événements permettent de capturer et de réagir à ces actions, rendant une application interactive.

Les événements dans React fonctionnent de manière similaire aux événements dans le DOM classique, mais avec quelques différences importantes. React implémente un système d'événements synthétiques, qui standardise les événements à travers différents navigateurs. Cela simplifie le développement et garantit un comportement cohérent.

1-1- Différences entre les Événements DOM et React

- **Nommage des événements :**En React, les événements utilisent la **casse camelCase** (par exemple, onClick au lieu de onclick en HTML).

En HTML

```html
<button onclick="alert('Clic sur le bouton!')">
  Cliquez-moi
</button>
```

En React

```jsx
function App() {
  const handleClick = () => {
    alert('Clic sur le bouton!');
  };
  return (
    <button onClick={handleClick}>
      Cliquez-moi
    </button>
  );
}
export default App;
```

- **Gestion des événements :**

  - En HTML, vous définissez directement un attribut ou utilisez addEventListener.
  - En React, vous passez une fonction en tant que gestionnaire d'événements.

En HTML

```html
<html>
<body>
  <button id="myButton">Cliquez-moi</button>
  <script>
    const button = document.getElementById('myButton');
    button.addEventListener('click', function () {
      alert('Clic sur le bouton!');
    });
  </script>
</body>
</html>
```

En React

```jsx
function App() {
  const handleClick = () => {
    alert('Clic sur le bouton!');
  };

  return (
    <button onClick={handleClick}>
      Cliquez-moi
    </button>
  );
}
export default App;
```

1-2- Types d'Événements Prisés en React

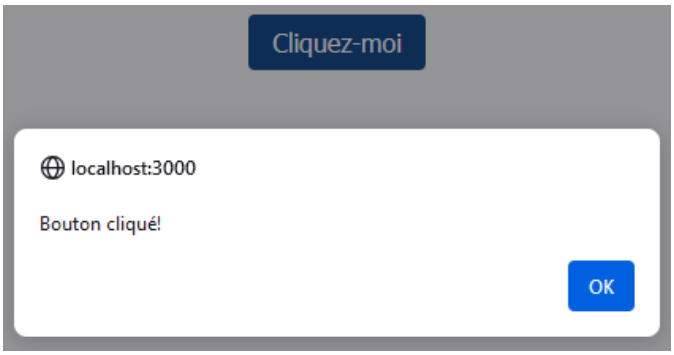
Voici une liste non exhaustive des types d'événements courants en React :

- **Événements de souris :** onClick, onMouseEnter, onMouseLeave, etc.
- **Événements clavier :** onKeyDown, onKeyPress, onKeyUp.
- **Événements formulaire :** onChange, onSubmit, onFocus, onBlur.
- **Événements de fenêtre :** onScroll, onResize.

1-3- Exemples d'Utilisation

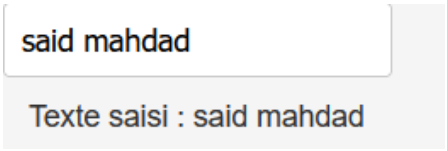
Exemple 1 : Gestionnaire d'Événement onClick

```jsx
import React from 'react';
function App() {
  const handleClick = () => {
    alert('Bouton cliqué!');
  };
  return (
    <button onClick={handleClick}>
      Cliquez-moi
    </button>
  );
}
export default App;
```



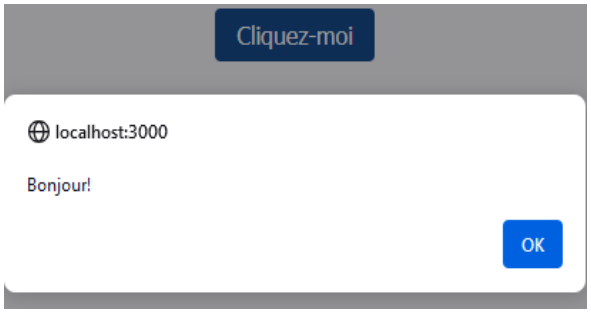
Exemple 2 : Événement onChange pour un Champ de Texte

```jsx
import React, { useState } from 'react';
function App() {
  const [value, setValue] = useState('');
  const handleChange = (event) => {
    setValue(event.target.value);
  };
  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
      <p>Texte saisi : {value}</p>
    </div>
  );
}
export default App;
```



Exemple 3 : Passer des Arguments à un Gestionnaire

```jsx
import React from 'react';
function App() {
    const handleClick = (message) => {
      alert(message);
    };
    return (
      <button onClick={() => handleClick('Bonjour!')}>
        Cliquez-moi
      </button>
    );
  }
  export default App;
```



Exemple 4 : Gestionnaire d'Événement onClick

```jsx
import React from 'react';
function App() {
    const handleSubmit = (event) => {
      event.preventDefault();
      alert('Formulaire soumis');
    };
    return (
      <form onSubmit={handleSubmit}>
        <button type="submit">Soumettre</button>
      </form>
    );
  }
  export default App;
```

2- Les formulaires

Un formulaire est une interface utilisateur conçue pour permettre la saisie, la modification ou la soumission de données par un utilisateur. Il se compose d'un ensemble de champs d'entrée tels que des cases à cocher, des boutons, des champs de texte, des sélecteurs, etc., regroupés dans une structure logique. Les formulaires sont couramment utilisés dans les applications web pour collecter des informations auprès des utilisateurs

2-1- Les formulaires contrôlés

Un **formulaire contrôlé** en React est un formulaire où chaque champ d'entrée (input) a sa valeur gérée par l'état React. Cela signifie que la valeur d'un champ est entièrement pilotée par l'état du composant, et chaque modification de l'entrée utilisateur déclenche une mise à jour de cet état.

Cette méthode permet d'avoir un contrôle total sur les données du formulaire et facilite leur validation et leur manipulation.

2-2- Caractéristiques d'un Formulaire Contrôlé

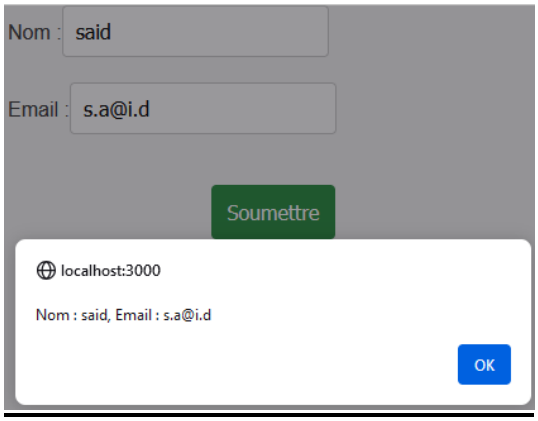
- **Lien entre l'état et les champs d'entrée :** La valeur de chaque champ est stockée dans l'état du composant. L'état est mis à jour via l'événement onChange.
- **Synchronisation constante :** Toute modification dans le champ se reflète instantanément dans l'état.
- **Gestion centralisée :** L'état sert de source unique de vérité pour les données du formulaire.

2-3- Exemples de Base :

Exemple 1 :Gestion des champs de saisie

```jsx
import React, { useState } from 'react';
function ControlledForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Nom : ${name}, Email : ${email}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nom :
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <br />
      <label>
        Email :
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
      </label>
      <br />
      <button type="submit">Soumettre</button>
    </form>
  );
}
export default ControlledForm;
```

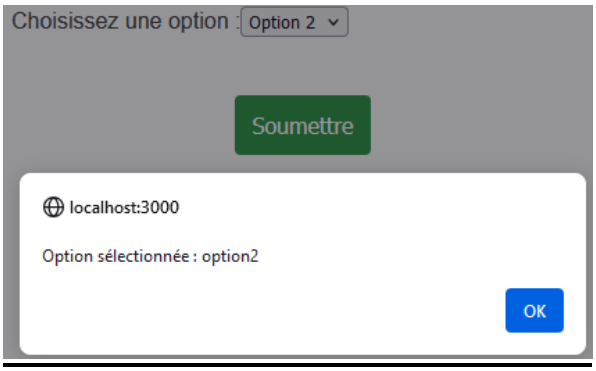
⇨



Exemple2 : Gestion des listes de séléction

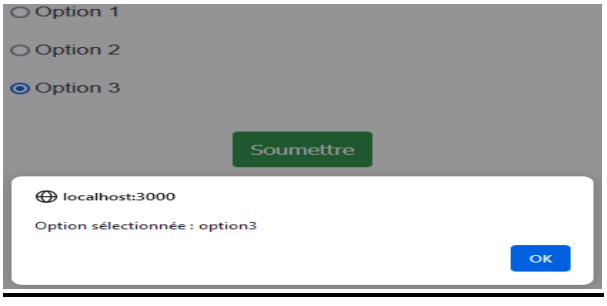
```jsx
import React, { useState } from 'react';
function ControlledSelect() {
  const [selectedOption, setSelectedOption] = useState('option1');
  const handleChange = (event) => {
    setSelectedOption(event.target.value);
  };
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Option sélectionnée : ${selectedOption}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Choisissez une option :
        <select value={selectedOption} onChange={handleChange}>
          <option value="option1">Option 1</option>
          <option value="option2">Option 2</option>
          <option value="option3">Option 3</option>
        </select>
      </label>
      <br />
      <button type="submit">Soumettre</button>
    </form>
  );
}

export default ControlledSelect;
```



Exemple 3 : Gestion des boutons radio

```jsx
import React, { useState } from 'react';
function ControlledRadioButtons() {
  const [selectedOption, setSelectedOption] = useState('option1');
  const handleChange = (event) => {
    setSelectedOption(event.target.value);
  };
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Option sélectionnée : ${selectedOption}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        <input type="radio" value="option1" checked={selectedOption === 'option1'} onChange={handleChange} />
        Option 1
      </label>
      <br />
      <label>
        <input type="radio" value="option2" checked={selectedOption === 'option2'} onChange={handleChange}/>
        Option 2
      </label>
      <br />
      <label>
        <input type="radio" value="option3" checked={selectedOption === 'option3'} onChange={handleChange}/>
        Option 3
      </label>
      <br />
      <button type="submit">Soumettre</button>
    </form>
  );
}
export default ControlledRadioButtons;
```



Stylisation des composants

La stylisation des composants en React est essentielle pour définir l'apparence et l'expérience utilisateur d'une application. React offre plusieurs approches pour styliser les composants, allant des styles en ligne aux bibliothèques spécialisées.

1- Styles En Ligne

Les styles en ligne utilisent l'attribut style, qui prend un objet en JavaScript contenant les propriétés CSS.

```jsx
import React from "react";
function App() {
    const style = {
      backgroundColor: 'blue',
      color: 'white',
      padding: '10px',
      borderRadius: '5px',
    };

    return <button style={style}>Bouton Stylisé</button>;
  }
  export default App;
```

⇨ 

2- Classes CSS

Vous pouvez utiliser des fichiers CSS pour styliser vos composants. Les classes CSS sont attribuées à l'aide de l'attribut className

```jsx
import React from "react";
import './styles.css';
    function App() {
      return <button className="button">Bouton Stylisé</button>;
    }
  export default App;
```

Le fichier styles.css

```css
.button {
    background-color: rgb(5, 5, 8);
    color: white;
    padding: 10px;
    border-radius: 5px;
    border: none;
  }
  .button:hover {
    background-color: rgb(8, 8, 13);
  }
```

⇨ 

3- Modules CSS

Les modules CSS offrent une encapsulation des styles pour éviter les conflits de noms. Chaque classe est accessible via un objet importé.

```jsx
import React from "react";
import styles from './Button.module.css';
function App() {
  return <button className={styles.button}>Bouton Stylisé</button>;
}
export default App;
```

Le fichier styles.css

⇨ 

```css
.button {
    background-color: rgb(5, 5, 8);
    color: white;
    padding: 10px;
    border-radius: 5px;
    border: none;
}
.button:hover {
    background-color: rgb(8, 8, 13);
}
```

4- Styled-Components (CSS-in-JS)

Les styled-components permettent d'écrire du CSS directement dans le fichier JavaScript en utilisant des "template literals"

```jsx
import styled from 'styled-components';
const Button = styled.button`
  background-color: blue;
  color: white;
  padding: 10px;
  border-radius: 5px;
  border: none;

  &:hover {
    background-color: darkblue;
  }
`;
function App() {
  return <Button>Bouton Stylisé</Button>;
}
export default App;
```

⇨

Mais il faut tout d’abord installer les composants stylisés en utilisant npm

```
npm install styled-components
```

Communication entre composants

La communication entre composants en React est essentielle pour construire des applications interactives et réactives. Elle consiste à transmettre des données ou des événements entre composants via plusieurs méthodes, selon leur relation hiérarchique.

Types de Communication

- De Parent à Enfant (via les props)
- De Enfant à Parent (via des fonctions de rappel)
- Entre Composants Non Liés (via un contexte ou un gestionnaire d'état global comme Redux)
- Entre Composants Frères (via un parent commun)
- Communication Globale (via des événements ou des bibliothèques externes)

1- Communication de Parent à Enfant

Le parent transmet des props (propriétés) à l'enfant pour lui fournir des données ou modifier son comportement.

```jsx
import React from "react";
function Child({ message }) {
    return <p>Message du parent : {message}</p>;
  }
  function Parent() {
    return <Child message="Bonjour, Enfant!" />;
  }
  export default Parent;
```

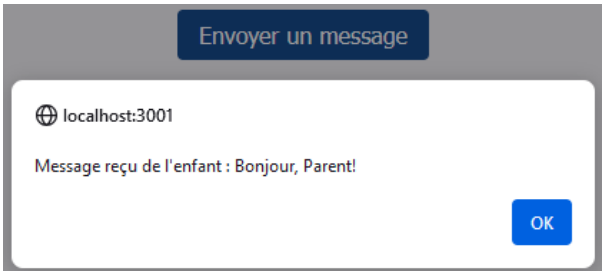
⇨ Message du parent : Bonjour, Enfant!

2- Communication de Enfant à Parent

Le parent fournit une fonction de rappel (callback) à l'enfant via les props. L'enfant peut appeler cette fonction pour transmettre des données ou des événements au parent.

```jsx
function Child({ onMessageChange }) {
    const handleClick = () => {
      onMessageChange('Bonjour, Parent!');
    };
    return <button onClick={handleClick}>Envoyer un message</button>;
  }
  function Parent() {
    const handleMessageChange = (message) => {
      alert(`Message reçu de l'enfant : ${message}`);
    };
    return <Child onMessageChange={handleMessageChange} />;
  }
  export default Parent;
```

⇨



3- Communication entre Composants Non Liés

Les composants non liés (sans relation hiérarchique) peuvent communiquer via :

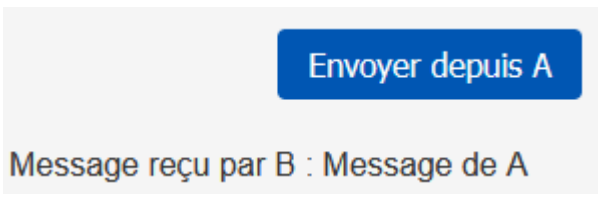
- **Le Contexte React** : Le contexte partage des données entre plusieurs composants
- **Gestionnaires d'État Global (Redux, Zustand, Recoil, etc.)** : Pour des projets plus complexes, des bibliothèques comme Redux permettent de gérer un état global

4- Communication entre Composants Frères

Les composants frères communiquent en utilisant un parent commun comme intermédiaire.

```jsx
import React from "react";
function SiblingA({ onMessageChange }) {
    const sendMessage = () => {
      onMessageChange('Message de A');
    };
    return <button onClick={sendMessage}>Envoyer depuis A</button>;
  }
  function SiblingB({ message }) {
    return <p>Message reçu par B : {message}</p>;
  }
  function Parent() {
    const [message, setMessage] = React.useState('');
    return (
      <div>
        <SiblingA onMessageChange={setMessage} />
        <SiblingB message={message} />
      </div>
    );
  }
  export default Parent;
```

⇨



le routage

Le routage en React permet de gérer la navigation entre différentes pages ou vues dans une application. Il est souvent réalisé à l'aide de bibliothèques comme React Router, qui offrent des outils puissants pour créer des routes dynamiques, gérer les URL et maintenir une expérience utilisateur fluide sans recharger la page.

Concepts de Base du Routage

- Routes : Définissent le chemin (URL) et le composant à afficher.
- Composants de Navigation : Permettent aux utilisateurs de naviguer entre les pages.
- Paramètres Dynamiques : Capturent des parties variables dans les URL.
- Redirections : Redirigent les utilisateurs vers une autre page en fonction des conditions.
- Routes Protégées : Limitent l'accès à certaines pages (authentification, autorisation).

1- Installation

Par défaut, React vient sans routage et pour l'activer, nous devons ajouter une bibliothèque nommé react-router

```
npm install react-router-dom
```

2- Création de Routes Simples

Exemple :

```jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
function Home() {
  return <h1>Accueil</h1>;
}
function About() {
  return <h1>À Propos</h1>;
}
function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Accueil</Link>
        {' | '}
        <Link to="/about">À Propos</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
}
export default App;
```

⇨

- **<Router>** : Fournit le contexte de routage à l'application.
- **<Routes> et <Route>** : Définissent les chemins et les composants correspondants.
- **<Link>** : Permet une navigation sans rechargement de la page.

3- les routes dynamiques

Les routes dynamiques permettent de capturer des segments variables dans l'URL, comme des identifiants d'utilisateur ou des catégories. Elles sont essentielles pour gérer des pages spécifiques ou personnalisées en fonction des données.
Une route dynamique contient un segment variable défini avec un : suivi d'un nom de paramètre.

Exemple :

```
<Route path="/user/:id" element={<User />} />
```

- **:id** est un paramètre dynamique.
- La partie :id dans l'URL peut correspondre à n'importe quelle valeur, comme /user/1 ou /user/ali.

4- Exemple complet

La structure de dossier src :

```
src/
├── components/
│   ├── Home.js
│   ├── User.js
│   └── NotFound.js
├── App.js
├── index.js
├── styles/
│   ├── App.css
│   └── User.css
```

Le fichier Home.js

```
import React from 'react';
function Home() {
  return <h1>Page d'accueil</h1>;
}
export default Home;
```

Le fichier NotFound.js

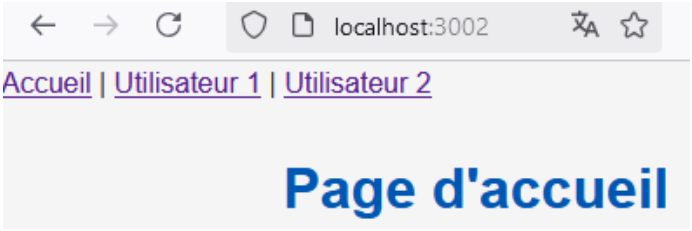
```
import React from 'react';
function NotFound() {
  return <h1>404 - Page non trouvée</h1>;
}
export default NotFound;
```

Le fichier User.js

```
import React from 'react';
import { useParams } from 'react-router-dom';
function User() {
  const { id } = useParams();
  return (
    <div>
      <h1>Profil de l'utilisateur</h1>
      <p>ID de l'utilisateur : {id}</p>
    </div>
  );
}
export default User;
```

Le fichier App.js

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './Home';
import User from './User';
import NotFound from './NotFound';
function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Accueil</Link> | <Link to="/user/1">Utilisateur 1</Link> |{' '}
        <Link to="/user/2">Utilisateur 2</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/user/:id" element={<User />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
export default App;
```



Redux

Redux est une bibliothèque de gestion d'état prévisible utilisée pour les applications JavaScript. Elle est souvent combinée avec React pour gérer l'état de manière centralisée dans des applications complexes. Redux suit trois principes fondamentaux :

- Unicité de la source de vérité
- L'état est en lecture seule
- Les changements sont effectués par des fonctions pures appelées reducers

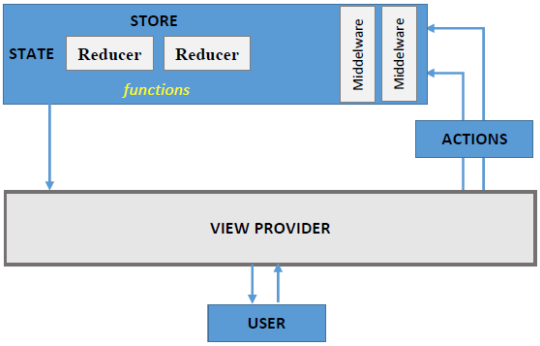
1- Pourquoi Utiliser Redux ?

- **Centralisation de l'État :** Tous les états sont gérés dans un seul endroit appelé store, facilitant l'accès et la maintenance.
- **Prédictibilité :** Les mises à jour d'état suivent un flux strict et prévisible.
- **Débogage Amélioré :** Outils comme Redux DevTools permettent de suivre les actions et l'état.

2- Concepts Clés de Redux

- **Store :** Contient l'état global de l'application.
- **Actions :** Des objets décrivant les événements qui déclenchent des changements d'état.
- **Reducers :** Des fonctions pures qui définissent comment l'état change en réponse aux actions.
- **Middleware :** Intercepte les actions avant qu'elles n'atteignent le reducer pour ajouter des fonctionnalités comme les appels API.

3- Architecture Redux



- **VIEW PROVIDER** : Représente un View Framework qui peut être React ou Angular,...
- **ACTION** : un objet pur créé pour stocker les informations relatives à l'événement d'un utilisateur (cliquez sur l'interface, ...).
- **STORE** : Gère l'état de l'application

L'objet store de Redux fournit plusieurs méthodes pour gérer et interagir avec l'état global de l'application. Voici les méthodes principales et leur rôle :

- **store.getState():** Renvoie l'état actuel du store. Elle est utilisée pour lire l'état global à tout moment.
- **store.dispatch(action) :** Envoie une action au store pour déclencher une modification de l'état via les reducers. elle est utilisée pour appliquer des modifications à l'état en déclenchant une action.
- **store.subscribe(listener) :** Enregistre une fonction de rappel (listener) qui sera exécutée chaque fois que l'état change. Elle Permet de réagir aux modifications de l'état (par exemple, mettre à jour l'interface utilisateur).

- **store.replaceReducer(nextReducer) :** Remplace le reducer actuel par un nouveau reducer. Elle est utilisée dans des scénarios avancés, comme le chargement dynamique de reducers

- **MIDDLEWARE** : fonction intermédiaire qui fournit un moyen d'interagir avec les objets Action envoyés à STORE avant leur envoi à REDUCER.
- **REDUCER** : (Modificateur) Une fonction pure pour renvoyer un nouvel état à partir de l'état initial.

4- Installation

Commencez par installer Redux et ses outils pour React :

```
npm install redux react-redux
```

5- Mise en place de redux

Exemple: Dans cet exemple, nous organisons les fichiers de manière suivante :

- Fichiers Redux (actions, reducer, store) : dans un dossier nommé Redux.
- Composants React : dans un dossier nommé Components

Étape 1 : Création des actions

Ils définissent les types d'actions et les fonctions d'action qui sont envoyées au store pour déclencher des changements d'état. Les actions sont des objets décrivant ce qui doit être fait.

```
export const INCREMENT = "INCREMENT";
export const DECREMENT = "DECREMENT";
export const increment = () => {
  return { type: INCREMENT };
};
export const decrement = () => {
  return { type: DECREMENT };
};
```

- `export const INCREMENT = "INCREMENT";` : Définit une constante pour le type d'action INCREMENT.
- `export const DECREMENT = "DECREMENT";` : Définit une constante pour le type d'action DECREMENT.
- `export const increment = () => { ... };`: Une fonction qui retourne un objet action pour incrémenter.
- `export const decrement = () => { ... };`: Une fonction qui retourne un objet action pour **décrémenter.**

Etape 2 : Création des reducers

Les reducers définissent comment l'état de l'application change en réponse aux actions. Ce sont des fonctions pures, ce qui signifie qu'elles produisent toujours le même résultat pour les mêmes entrées.

```
import { INCREMENT, DECREMENT } from './Actions';

const initialState = {
  count: 0,
};
export default function counterReducer(state = initialState, action) {
  switch (action.type) {
    case INCREMENT:
      return { ...state, count: state.count + 1 };
    case DECREMENT:
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

- `import { INCREMENT, DECREMENT } from "./actions";` : Importe les types d'actions.
- `const initialState = { count: 0 };` : Définit l'état initial avec un compteur à 0.
- `const counterReducer = (state = initialState, action) => { ... };` : Fonction du reducer qui prend l'état actuel et une action.
- `switch (action.type)` : Examine le type d'action.
- `case INCREMENT:` : Si l'action est INCREMENT, incrémente count.
- `case DECREMENT:` : Si l'action est DECREMENT, décrémente count.
- `default:` : Si aucune action n'est reconnue, retourne l'état actuel.

Étape 3 : Configuration du Store

Le store est l'endroit où l'état global de l'application est stocké. Il orchestre les reducers et les actions.

```
1   import { createStore } from 'redux';
2   import rootReducer from '../reducers';
3   const store = createStore(rootReducer);
4   export default store;
```

- `import { createStore } from "redux";` : Importe la fonction createStore de Redux.
- `import counterReducer from "./reducer";` : Importe le reducer.
- `const store = createStore(counterReducer);` : Crée le store en passant le reducer.
- `export default store;` : Exporte le store pour utilisation dans l'application.

Étape 4 : Création du composant Counter (Counter.js)

```
1   import React from "react";
2   import { useSelector, useDispatch } from "react-redux";
3   import { increment, decrement } from "../Redux/actions";
4   const Counter = () => {
5     const count = useSelector((state) => state.count);
6     const dispatch = useDispatch();
7     return (
8       <div>
9         <h1>Counter: {count}</h1>
10        <button onClick={() => dispatch(increment())}>Increment</button>
11        <button onClick={() => dispatch(decrement())}>Decrement</button>
12      </div>
13    );
14  };
15  export default Counter;
```

- `import React from "react";` : Importe React.
- `import { useSelector, useDispatch } from "react-redux";` : Importe les hooks pour interagir avec Redux.
- `import { increment, decrement } from "../Redux/actions";` : Importe les actions.
- `const count = useSelector((state) => state.count);` : Le hook useSelector est utilisé pour accéder à une partie de l'état global stocké dans Redux. Il permet de sélectionner des données spécifiques depuis le store.
- `const dispatch = useDispatch();` : Le hook useDispatch est utilisé pour envoyer (ou "dispatcher") des actions au store Redux. Il retourne une fonction dispatch que vous pouvez appeler pour exécuter une action.
- `<button onClick={() => dispatch(increment())}>Increment</button>` : Déclenche l'action increment au clic.
- `<button onClick={() => dispatch(decrement())}>Decrement</button>` : Déclenche l'action decrement au clic.

Étape 5 : Intégration dans index.js

```jsx
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./Redux/store";
import Counter from "./Components/Counter";
ReactDOM.render(
  <Provider store={store}>
    <Counter />
  </Provider>,
  document.getElementById("root")
);
```

- `import { Provider } from "react-redux";` : Importe le composant Provider pour connecter Redux à React.
- `import store from "./Redux/store";` : Importe le store configuré.
- `<Provider store={store}>` : Rend le store disponible pour tous les composants enfants.
- `ReactDOM.render(..., document.getElementById("root"));` : Rend l'application dans l'élément HTML avec l'id root.

la bibliothèque Immer

Immer est une bibliothèque JavaScript qui simplifie la gestion des états immuables, particulièrement utile dans des frameworks comme ReactJS. Avec Immer, vous pouvez manipuler les états de manière "mutative" dans une zone sûre (appelée draft), et Immer génère automatiquement un nouvel état immuable.

1- Pourquoi utiliser Immer ?

- **Facilité d'écriture :** Vous travaillez comme si vous modifiiez l'état directement.
- **Immutabilité assurée :** Immer s'assure que l'état source n'est jamais modifié.
- **Moins d'erreurs :** Réduit la complexité de la gestion manuelle d'états immuables.
- **Code plus propre et lisible.**

2- Principes de base d'Immer

Immer repose sur :

- produce : La fonction principale d'Immer qui crée une nouvelle version immuable d'un état basé sur un brouillon.
- Brouillon (draft) : Un objet temporaire que vous modifiez directement. Immer se charge de produire une copie immuable de ce brouillon.

3- Installation

```
npm install immer
```

4- Mise en place de Immer

Cas de base : Changer un compteur

Sans Immer

```jsx
import React, { useState } from "react";
const Counter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount((prevCount) => prevCount + 1);
  };
  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={increment}>Incrémenter</button>
    </div>
  );
};
export default Counter;
```

Avec immer

```jsx
import React, { useState } from "react";
import { produce } from "immer";
const Counter = () => {
  const [state, setState] = useState({ count: 0 });
  const increment = () => {
    setState((currentState) =>
      produce(currentState, (draft) => {
        draft.count += 1;
      })
    );
  };
  return (
    <div>
      <p>Compteur : {state.count}</p>
      <button onClick={increment}>Incrémenter</button>
    </div>
  );
};
export default Counter;
```

État initial : { count: 0 } est un objet au lieu d'un simple nombre.
Fonction produce : Elle crée un "brouillon" de l'état actuel, que vous modifiez directement.
Automatisation : Immer produit automatiquement une nouvelle version immuable de l'état après modification du brouillon.

5- Exemple d'un TODOLIST avec Immer

Nous allons créer une TODO List avec les fonctionnalités suivantes :

- Ajouter une tâche
- Supprimer une tâche

■ Marquer une tâche comme terminée

```jsx
import React, { useState } from "react";
import { produce } from "immer";
const TodoList = () => {
  const [todos, setTodos] = useState([]);
  const addTodo = (text) => {
    setTodos((currentTodos) =>
      produce(currentTodos, (draft) => {
        draft.push({ id: Date.now(), text, completed: false });
      })
    );
  };
  const toggleTodo = (id) => {
    setTodos((currentTodos) =>
      produce(currentTodos, (draft) => {
        const todo = draft.find((t) => t.id === id);
        if (todo) {
          todo.completed = !todo.completed;
        }
      })
    );
  };
  const deleteTodo = (id) => {
    setTodos((currentTodos) =>
      produce(currentTodos, (draft) => {
        const index = draft.findIndex((t) => t.id === id);
        if (index !== -1) {
          draft.splice(index, 1);
        }
      })
    );
  };
  return (
    <div>
      <h1>TODO List</h1>
      <AddTodoForm onAddTodo={addTodo} />
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>
            <label>
              <input
                type="checkbox"
                checked={todo.completed}
                onChange={() => toggleTodo(todo.id)}
              />
              <span style={{ textDecoration: todo.completed ? "line-through" : "none" }}>
                {todo.text}
              </span>
            </label>
            <button onClick={() => deleteTodo(todo.id)}>Supprimer</button>
          </li>
        ))}
      </ul>
    </div>
  );
};
const AddTodoForm = ({ onAddTodo }) => {
  const [text, setText] = useState("");
  const handleSubmit = (e) => {
    e.preventDefault();
    if (text.trim() === "") return;
    onAddTodo(text.trim());
    setText("");
  };
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Ajouter une tâche"
      />
      <button type="submit">Ajouter</button>
    </form>
  );
};
export default TodoList;
```

Ajout de tâche : La fonction addTodo utilise **Immer** pour ajouter une nouvelle tâche à la liste des tâches. draft.push({ ... }) modifie le brouillon directement.
Basculer l'état terminé/non terminé : La fonction toggleTodo trouve la tâche par son id et bascule sa propriété completed.

Suppression de tâche : La fonction deleteTodo trouve l'index de la tâche par son id et utilise draft.splice() pour la supprimer.

Redux Toolkit

Redux Toolkit est un outil officiel proposé par l'équipe de Redux pour simplifier le développement avec Redux. Redux, en tant que bibliothèque de gestion d'état, peut parfois sembler complexe à cause de sa configuration et des nombreux fichiers nécessaires. Redux Toolkit vise à résoudre ces problèmes en rendant Redux plus simple, rapide et intuitif.

1- Pourquoi utiliser Redux Toolkit ?

- **Configuration simplifiée** : Moins de code boilerplate (code répétitif).
- **Performance améliorée** : Des fonctions intégrées optimisées.
- **Outils modernes** : Intégration avec des outils comme Immer pour manipuler facilement l'état de manière immuable.
- **Code plus propre et lisible.**

2- Concepts clés de Redux Toolkit

- **Store** : Le magasin central qui contient l'état de votre application.
- **Slice** : Une "tranche" d'état qui regroupe état, actions, et réductions dans un seul fichier.
- **createSlice** : Fonction principale qui simplifie la création de réducteurs et d'actions.
- **createAsyncThunk** : Gestion simplifiée des requêtes asynchrones (ex. API).
- **configureStore** : Configure le magasin Redux avec des paramètres par défaut, comme le middleware.

3- Installation

```
npm install @reduxjs/toolkit react-redux
```

4- Mise en place de redux

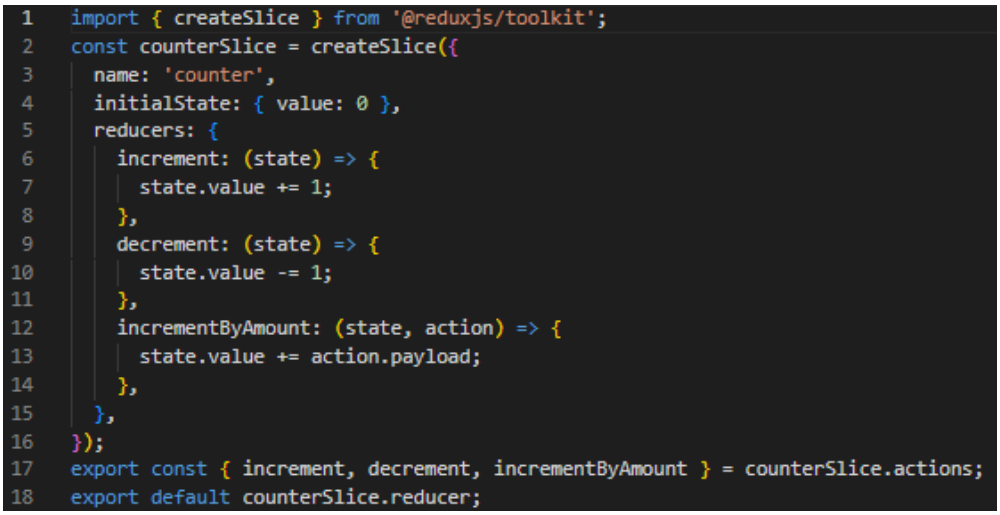
Exemple: Gestion d'un Compteur

L'organisation des fichiers sera

```
src/
├── redux/
│   ├── counterSlice.js
│   └── store.js
├── components/
│   └── Counter.js
├── App.js
├── index.js
```

- counterSlice.js : Contient la logique de l'état et des actions du compteur.
  - Définition d'un slice avec createSlice.
  - Export des actions (increment, decrement, incrementByAmount) et du réducteur.
- store.js : Configure le store Redux.
  - Configuration du store avec configureStore.
  - Intégration du réducteur counter.
- Counter.js : Composant React pour afficher et interagir avec le compteur.
  - Utilisation de useSelector pour lire l'état du compteur.
  - Utilisation de useDispatch pour déclencher des actions.

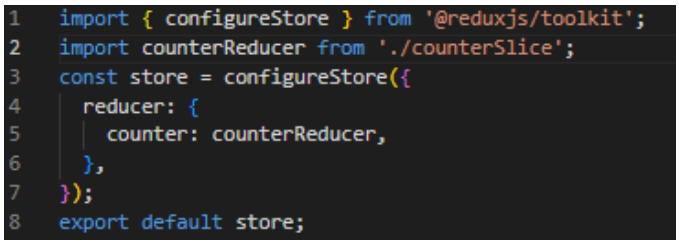
Étape 1 : Création d'une slice



```
import { createSlice } from '@reduxjs/toolkit';
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});
export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

import : On importe la fonction createSlice depuis la bibliothèque @reduxjs/toolkit.
createSlice : Une fonction qui simplifie la création de réducteurs Redux en combinant état initial, actions et réducteurs dans un seul endroit.
const counterSlice : Déclare une constante counterSlice, qui contient le slice Redux (tranche d'état).
createSlice({...}) : Configure le slice Redux en définissant son état initial, son nom, et les réducteurs associés.
name : Attribue un nom unique au slice. Ce nom est utilisé pour identifier les actions et les états générés par ce slice. Ici, le nom du slice est 'counter'.
initialState : Définit l'état initial de cette tranche. Ici, l'état initial est un objet avec une seule propriété : value, qui est initialisée à 0.
reducers : Contient les réducteurs, qui définissent comment l'état est mis à jour en réponse aux actions. Chaque clé (par exemple increment, decrement) représente une action, et sa valeur est une fonction qui modifie l'état.
increment: (state) => { state.value += 1; } : Action qui incrémente la valeur de l'état de 1.
state.value += 1 : Modifie directement la propriété value de l'état.
incrementByAmount: (state, action) => { state.value += action.payload; } : Action qui ajoute une valeur personnalisée à l'état.
action.payload : La valeur transmise à l'action (par exemple, 5 pour ajouter 5).

Étape 2 : Création du store :



```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
export default store;
```

import : On importe la fonction configureStore depuis @reduxjs/toolkit.
configureStore : Une fonction qui simplifie la création du store Redux en configurant automatiquement les middlewares et en permettant d'ajouter plusieurs slices.
import counterReducer : On importe le réducteur défini dans le fichier counterSlice.js.
Ce réducteur gère l'état lié au compteur (counter), incluant les actions increment, decrement, et incrementByAmount.
const store : Création du store Redux en utilisant configureStore.
reducer : Une propriété qui permet de définir les réducteurs associés au store.

counter: counterReducer : Le slice nommé counter est géré par le réducteur counterReducer (importé depuis counterSlice.js). Cette configuration associe l'état et les actions du slice counter au store Redux.

Étape 3 : Création de composant :

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementByAmount } from './counterSlice';
const Counter = () => {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();
  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>Add 5</button>
    </div>
  );
};
export default Counter;
```

import React : Importe la bibliothèque React pour créer le composant fonctionnel Counter

useDispatch : increment, decrement, incrementByAmount : Ces actions sont importées depuis counterSlice.js. Elles permettent d'incrémenter, décrémenter et ajouter une valeur personnalisée à l'état.

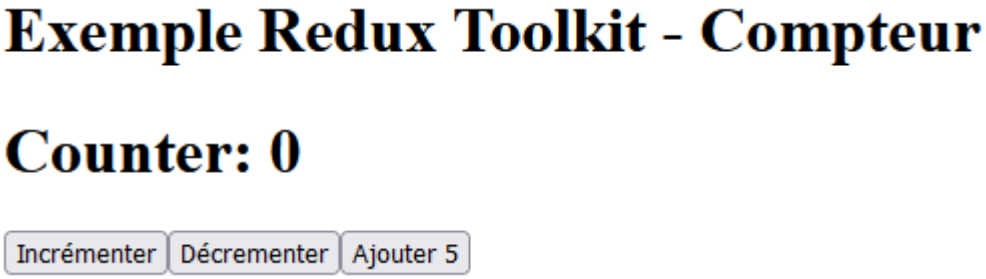
useSelector : Un hook fourni par react-redux pour accéder à l'état global du store Redux. Il Permet de lire une partie spécifique de l'état global.

state.counter.value : Accède à la propriété value de l'état du slice counter dans le store Redux.

count : Une constante qui contient la valeur actuelle de l'état.

useDispatch : Un hook fourni par react-redux pour envoyer (dispatch) des actions au store Redux. Il Permet d'obtenir la méthode dispatch pour envoyer des actions au store Redux.

dispatch : Une constante qui permet d'envoyer des actions (comme increment, decrement).



Redux DevTools

Redux DevTools est un outil puissant pour déboguer et analyser les états et actions dans une application utilisant Redux. Il permet de suivre l'historique des actions, de visualiser les modifications de l'état et même de rejouer ou modifier des actions en direct.

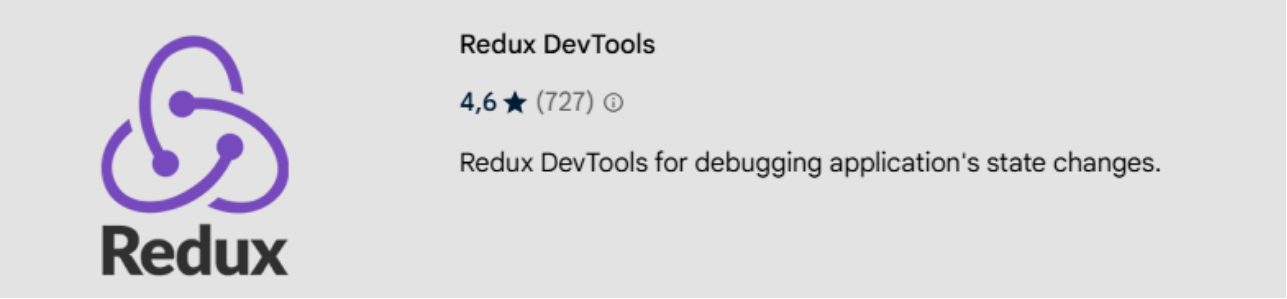
1- Avantages de Redux DevTools

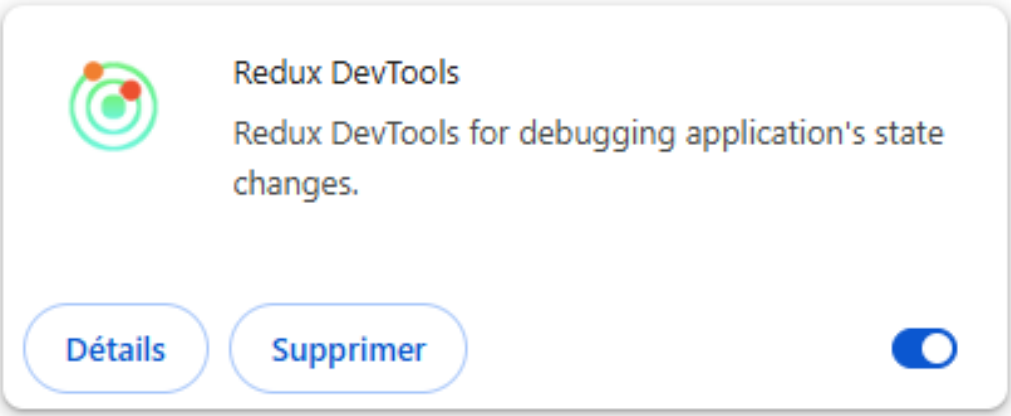
- **Suivi des actions :** Affiche chaque action déclenchée et son effet sur l'état.
- **Visualisation de l'état :** Comparez l'état précédent et actuel.
- **Time Travel Debugging :** Revenez à un état précédent pour analyser le comportement de l'application.
- **Optimisation :** Aide à identifier les actions inutiles ou les erreurs dans la gestion de l'état.

2- Installation

Installez l'extension Redux DevTools pour votre navigateur via les liens ci-dessous :

- Chrome Web Store :
  https://chromewebstore.google.com/category/extensions?utm_source=ext_sidebar&hl=fr
- Firefox Add-ons :
  about:addons





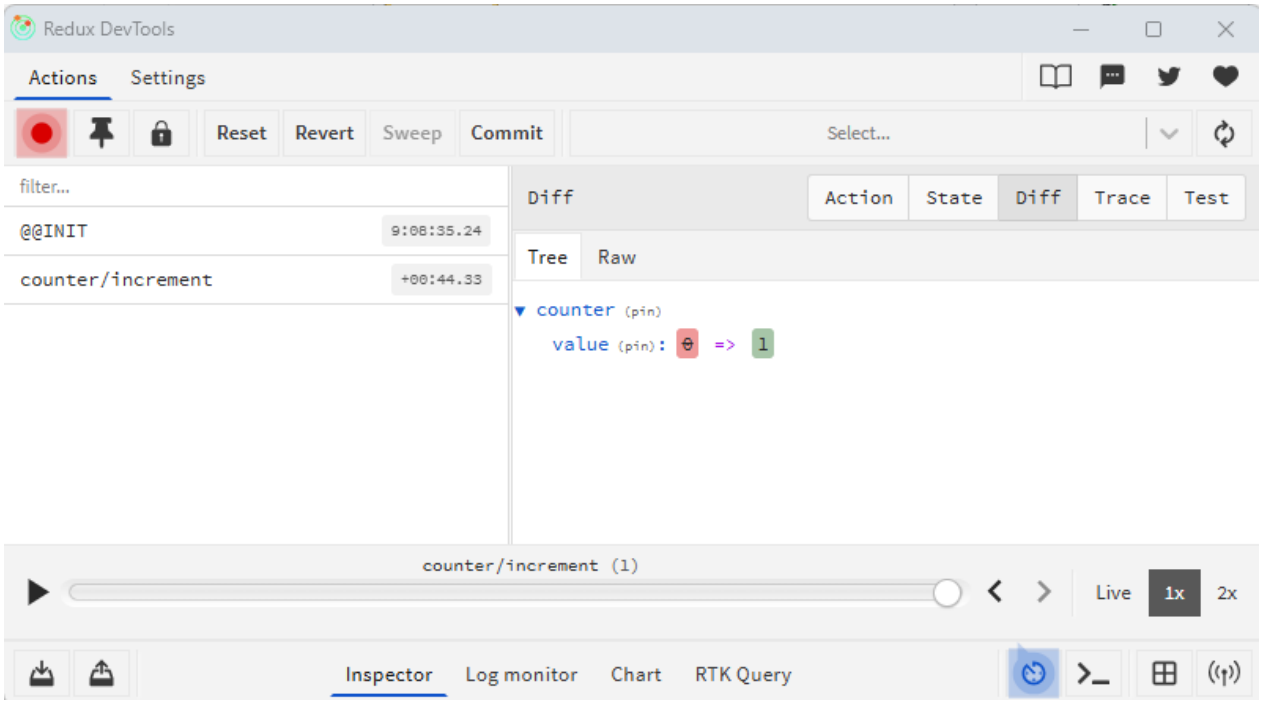
3- Exemple: Gestion d'un Compteur

Prenons l'exemple précédant (Gestion d'un compteur)

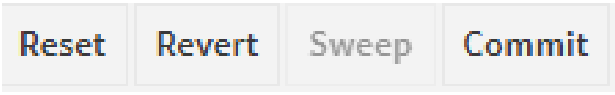
Exemple Redux Toolkit - Compteur

Counter: 1

Incrémenter | Décrementer | Ajouter 5

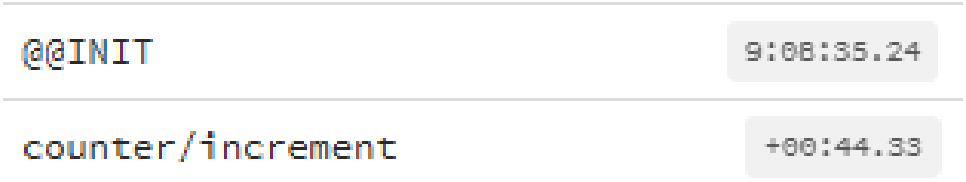


Barre d'actions principale (en haut)



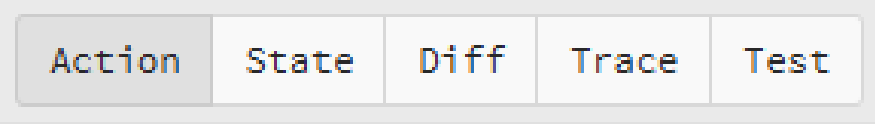
- **Reset :** Réinitialise l'état et supprime toutes les actions enregistrées. Cela revient à repartir de zéro.
- **Revert :** Annule toutes les actions effectuées et revient à l'état initial, tout en conservant les actions dans l'historique.
- **Sweep :** Supprime toutes les actions annulées pour nettoyer l'historique des actions.
- Commit : Définit l'état actuel comme nouveau point de départ et supprime l'historique précédent des actions.
- **Select :** Permet de filtrer les actions ou de visualiser des actions spécifiques à l'aide de critères définis.

Panneau de gauche : Liste des actions



- **@INIT :** Représente l'initialisation du store. Cette action est automatiquement exécutée lorsque le store est configuré.
- **Actions personnalisées (e.g., counter/increment)** : Liste les actions déclenchées dans votre application. Chaque action affiche son type (par exemple, counter/increment) et le moment où elle a été déclenchée.

Panneau principal : Détails des actions

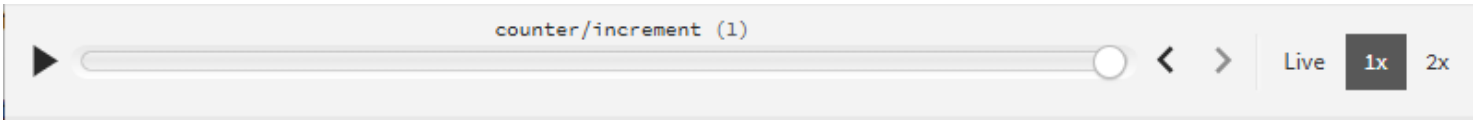


Vous pouvez explorer plusieurs onglets liés à une action ou un état :

- Action : Affiche les détails de l'action sélectionnée, comme ses données ou paramètres envoyés (payload).
- **State :** Montre l'état complet du store après que l'action sélectionnée a été appliquée.
- **Diff :** Visualise les différences entre l'état précédent et l'état actuel après l'application de l'action.
- **Trace :** Permet de déboguer l'exécution d'une action en suivant la trace des appels de fonctions dans le code.
- **Test :** Génère automatiquement du code de test pour l'action sélectionnée. Cela peut être utile pour écrire des tests unitaires.

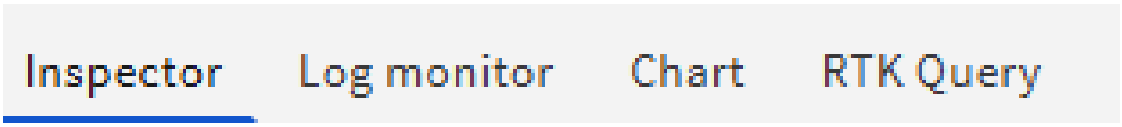
Barre inférieure : Contrôle de lecture des actions

Permet de manipuler l'historique des actions, comme si vous regardiez une vidéo :



- **Play/Pause :** Rejoue les actions enregistrées.
- **Live :** Si activé, affiche les actions en direct à mesure qu'elles sont déclenchées.
- **1x, 2x, etc. :** Contrôle la vitesse de lecture des actions enregistrées.

Onglets supplémentaires (en bas)



- **Inspector :** Vue principale où vous explorez les actions et l'état.
- **Log monitor :** Affiche les actions dans un format de journal (log).
- **Chart :** Représente l'état ou les actions sous forme graphique pour une vue plus visuelle.
- **RTK Query :** Si vous utilisez RTK Query, cet onglet affiche des informations sur les requêtes API et leur état.

Tester une application

Redux DevTools est un outil puissant pour déboguer et analyser les états et actions dans une application utilisant Redux. Il permet de suivre l'historique des actions, de visualiser les modifications de l'état et même de rejouer ou modifier des actions en direct.

1- Les types des tests

Pour tester une application React, il est important de couvrir les trois types principaux de tests : tests unitaires, tests d'intégration, et tests end-to-end (E2E).

- **Tests unitaires :**

Ces tests vérifient des composants ou fonctions isolés pour s'assurer qu'ils se comportent comme attendu.

- **Tests d'intégration:**

Ces tests vérifient que plusieurs composants fonctionnent correctement ensemble.

- **Tests end-to-end(E2E):**

Ces tests vérifient le comportement complet de l'application en simulant des interactions utilisateur.

2- Enzyme

Enzyme est une bibliothèque utilisée pour tester les composants React. Elle a été populaire avant l'émergence de React Testing Library, mais elle reste utile pour des tests basés sur la structure des composants. Voici comment l'utiliser avec des exemples concrets.

3- Installation d'Enzyme

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

Remplacez react-16 par la version de React que vous utilisez (par exemple, react-17 ou react-18).

4- Configuration d'enzyme

Applications

1- Application 1 :

■ Description :

Une application simple qui organise et recherche des notes.

■ Fonctionnalités :

- Ajouter des notes.
- Rechercher les notes par mot-clé ou catégorie.
- Stockage des données dans Redux Store avec mise à jour optimisée grâce à Immer.

■ Structure :

```
src/
├── components/
│   ├── NoteForm.js         // Composant pour ajouter des notes
│   ├── NoteList.js         // Composant pour afficher la liste des notes
│   ├── SearchBar.js        // Composant pour la recherche des notes
├── store/
│   ├── notesSlice.js       // Gestion des notes (reducers/actions)
│   ├── store.js            // Configuration du Redux Store
├── styles/
│   ├── App.css             // Fichier CSS pour le style
├── App.js                  // Composant principal
├── index.js                // Entrée principale de l'application
```

■ Les codes

- *Le fichier store.jsx*

```jsx
import { configureStore } from '@reduxjs/toolkit';
import notesReducer from './notesSlice';
const store = configureStore({
  reducer: {
    notes: notesReducer,
  },
});
export default store;
```

- *Le fichier noteSlice.jsx*

```jsx
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
  notes: [],
  searchQuery: '',
};
const notesSlice = createSlice({
  name: 'notes',
  initialState,
  reducers: {
    addNote: (state, action) => {
      state.notes.push(action.payload);
    },
    deleteNote: (state, action) => {
      state.notes = state.notes.filter((note) => note.id !== action.payload);
    },
    setSearchQuery: (state, action) => {
      state.searchQuery = action.payload;
    },
  },
});
export const { addNote, deleteNote, setSearchQuery } = notesSlice.actions;
export default notesSlice.reducer;
```

- *Le fichier NoteForm.jsx*

```jsx
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addNote } from '../store/notesSlice';
const NoteForm = () => {
  const [title, setTitle] = useState('');
  const dispatch = useDispatch();
  const handleSubmit = (e) => {
    e.preventDefault();
    if (title.trim() !== '') {
      dispatch(addNote({
        id: Date.now(),
        title,
      }));
      setTitle('');
    }
  };
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Ajouter une note"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
      <button type="submit">Ajouter</button>
    </form>
  );
};
export default NoteForm;
```

- *Le fichier NoteList.jsx*

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { deleteNote } from '../store/notesSlice';
const NoteList = () => {
  const { notes, searchQuery } = useSelector((state) => state.notes);
  const dispatch = useDispatch();
  const filteredNotes = notes.filter((note) =>
    note.title.toLowerCase().includes(searchQuery.toLowerCase())
  );
  return (
    <div>
      {filteredNotes.map((note) => (
        <div className="note" key={note.id}>
          <h3>{note.title}</h3>
          <button onClick={() => dispatch(deleteNote(note.id))}>
            Supprimer
          </button>
        </div>
      ))}
      {filteredNotes.length === 0 && <p>Aucune note trouvée.</p>}
    </div>
  );
};
export default NoteList;
```

- *Le fichier searchBar.jsx*

```jsx
import React from 'react';
import { useDispatch } from 'react-redux';
import { setSearchQuery } from '../store/notesSlice';
const SearchBar = () => {
  const dispatch = useDispatch();
  const handleSearch = (e) => {
    dispatch(setSearchQuery(e.target.value));
  };
  return (
    <input
      type="text"
      placeholder="Rechercher une note..."
      onChange={handleSearch}
      style={{ marginBottom: '20px', padding: '10px', width: '100%' }}
    />
  );
};
export default SearchBar;
```

- *Le fichier app.jsx*

```jsx
import React from 'react';
import NoteForm from './components/NoteForm';
import NoteList from './components/NoteList';
import SearchBar from './components/searchBar';
import './styles/App.css';
const App = () => {
  return (
    <div className="container">
      <h1>Gestion des Notes</h1>
      <SearchBar />
      <NoteForm />
      <NoteList />
    </div>
  );
};
export default App;
```

- *Le fichier index.jsx*

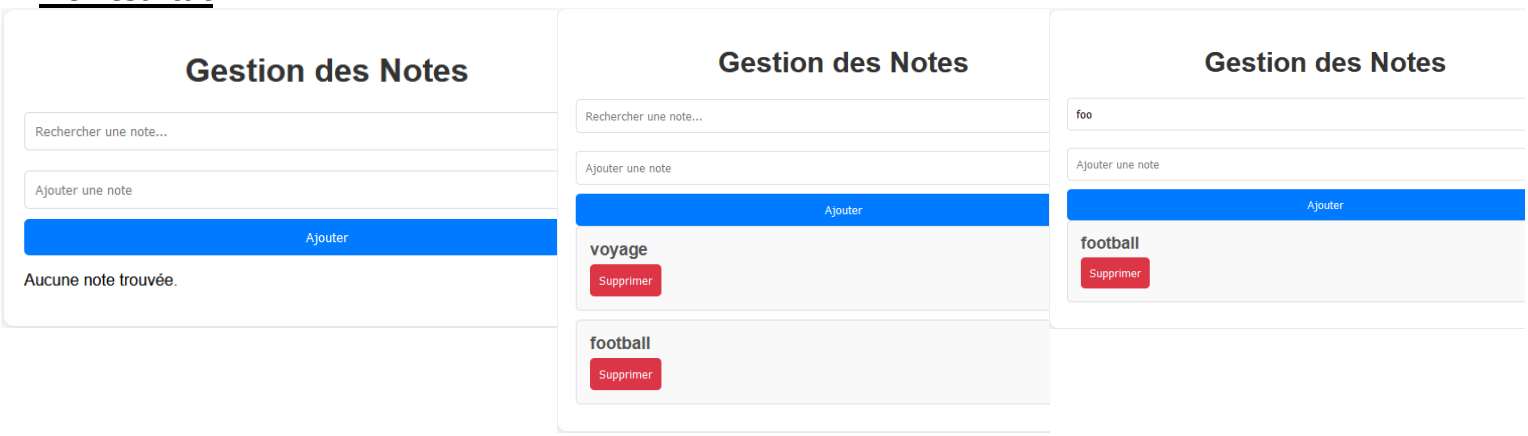
```jsx
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store/store';
import App from './App';
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

- *Le fichier app.css*

```css
body {
    font-family: Arial, sans-serif;
    background-color: #f7f7f7;
    margin: 0;
    padding: 0;
}
.container {
    max-width: 600px;
    margin: 20px auto;
    padding: 20px;
    background: white;
    border-radius: 10px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}
h1 {
    text-align: center;
    color: #333;
}
form {
    display: flex;
    flex-direction: column;
    gap: 10px;
}
input {
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
}
```

```css
button {
    padding: 10px;
    background-color: #007bff;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
button:hover {
    background-color: #0056b3;
}
.note {
    padding: 15px;
    margin-bottom: 10px;
    background-color: #f9f9f9;
    border: 1px solid #ddd;
    border-radius: 5px;
}
.note h3 {
    margin: 0;
    color: #555;
}
.note button {
    background-color: #dc3545;
    margin-top: 5px;
}
.note button:hover {
    background-color: #b02a37;
}
```

■ Le résultat



2- Application 2 :

■ **Description :** Une boutique en ligne où les utilisateurs peuvent

- Afficher et parcourir des produits,
- Filtrer les produits selon leur catégorie,
- Ajouter les produits au panier
- Supprimer un produit de panier.
- Passer la commande

■ **Fonctionnalités :**

- Gestion du panier d'achat avec **Redux Toolkit**.
- Pages produit avec informations détaillées.
- Système de filtres avancés.

- Intégration des Hooks pour gérer les interactions utilisateur.

■ **Structure :**

```
src/
├── components/          # Contient les composants réutilisables
│   ├── Filters.js       # Composant pour filtrer les produits par catégorie
│   ├── ProductCard.js   # Affiche une carte individuelle pour chaque produit
│   ├── ProductList.js   # Affiche une liste de produits sous forme de cartes
│   ├── Cart.js          # Composant pour afficher les articles ajoutés au panier
├── pages/               # Contient les pages principales de l'application
│   ├── HomePage.js      # Page d'accueil, affiche les produits avec les filtres
│   ├── ProductPage.js   # Page de détails d'un produit
├── store/               # Contient la logique Redux pour gérer l'état global
│   ├── cartSlice.js     # Slice Redux pour gérer les actions du panier
│   ├── productSlice.js  # Slice Redux pour gérer les produits et les filtres
│   ├── store.js         # Configuration du Redux Store pour connecter les slices
├── data/                # Contient les fichiers de données statiques
│   ├── products.json    # Liste des produits disponibles (format JSON)
├── styles/              # Contient les fichiers CSS pour la mise en page
│   ├── App.css          # Styles globaux pour l'application
│   ├── ProductCard.css  # Styles spécifiques aux cartes produits
│   ├── ProductPage.css  # Styles pour la page de détails d'un produit
│   ├── Cart.css         # Styles pour la page du panier
├── App.js               # Composant racine, gère les routes principales
├── index.js             # Point d'entrée principal, initialise Redux et React Router
```

■ Les codes :

- *Le fichier Cart.jsx*

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { removeFromCart } from '../store/cartSlice';
import '../styles/Cart.css';
const Cart = () => {
  const cartItems = useSelector((state) => state.cart.items);
  const total = useSelector((state) => state.cart.total);
  const dispatch = useDispatch();
  const handleRemove = (id) => {
    dispatch(removeFromCart({ id }));
  };
  return (
    <div className="cart-container">
      <h2>Votre Panier</h2>
      {cartItems.length === 0 ? (
        <p>Votre panier est vide.</p>
      ) : (
        <>
          <div className="cart-items">
            {cartItems.map((item) => (
              <div key={item.id} className="cart-item">
                <div className="cart-item-info">
                  <h3>{item.title}</h3>
                  <p>Quantité : {item.quantity}</p>
                  <p>Prix : {item.price} €</p>
                </div>
                <button
                  className="remove-button"
                  onClick={() => handleRemove(item.id)}
                >
                  Supprimer
                </button>
              </div>
            ))}
          </div>
          <div className="cart-total">
            <h3>Total : {total.toFixed(2)} €</h3>
            <button className="checkout-button">Passer la commande</button>
          </div>
        </>
      )}
    </div>
  );
};
export default Cart;
```

- *Le fichier ProductCard.jsx*

```jsx
import React from 'react';
import { useDispatch } from 'react-redux';
import { addToCart } from '../store/cartSlice';
import '../styles/ProductCard.css';
const ProductCard = ({ product }) => {
  const dispatch = useDispatch();
  const handleAddToCart = () => {
    dispatch(addToCart(product));
  };
  return (
    <div className="product-card">
      <h3>{product.title}</h3>
      <p>{product.price} €</p>
      <button onClick={handleAddToCart}>Ajouter au panier</button>
    </div>
  );
};
export default ProductCard;
```

- *Le fichier ProductList.jsx*

```jsx
import React from 'react';
import { useSelector } from 'react-redux';
import ProductCard from './ProductCard';
const ProductList = () => {
  const products = useSelector((state) => state.product.filteredProducts);
  if (!products || products.length === 0) {
    return <p>Aucun produit disponible.</p>;
  }
  return (
    <div className="product-list">
      {products.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
};
export default ProductList;
```

- *Le fichier Filters.jsx*

```jsx
import React from 'react';
import { useDispatch } from 'react-redux';
import { filterProducts } from '../store/productSlice';
const Filters = () => {
  const dispatch = useDispatch();
  const handleFilter = (e) => {
    dispatch(filterProducts({ category: e.target.value }));
  };
  return (
    <div className="filters">
      <select onChange={handleFilter}>
        <option value="">Tous</option>
        <option value="electronics">Électronique</option>
        <option value="clothing">Vêtements</option>
      </select>
    </div>
  );
};
export default Filters;
```

- *Le fichier ProductPage.jsx*

```jsx
import React from 'react';
import { useParams } from 'react-router-dom';
import { useSelector, useDispatch } from 'react-redux';
import { addToCart } from '../store/cartSlice';
import '../styles/ProductPage.css';
const ProductPage = () => {
  const { id } = useParams();
  const product = useSelector((state) =>
    state.product.products.find((prod) => prod.id === parseInt(id))
  );
  const dispatch = useDispatch();
  if (!product) {
    return <p>Produit introuvable.</p>;
  }
  const handleAddToCart = () => {
    dispatch(addToCart(product));
  };
  return (
    <div className="product-page">
      <h1>{product.title}</h1>
      <p>{product.description}</p>
      <p>{product.price} €</p>
      <button onClick={handleAddToCart}>Ajouter au panier</button>
    </div>
  );
};
export default ProductPage;
```

- *Le fichier HomePage.jsx*

```jsx
import React, { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import { setProducts } from '../store/productSlice';
import Filters from '../components/Filters';
import ProductList from '../components/ProductList';
import productsData from '../data/products.json';
const HomePage = () => {
  const dispatch = useDispatch();
  useEffect(() => {
    dispatch(setProducts(productsData));
  }, []);

  return (
    <div>
      <h1>Boutique</h1>
      <Filters />
      <ProductList />
    </div>
  );
};
export default HomePage;
```

- *Le fichier cartSlice.jsx*

```jsx
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
  items: [],
  total: 0,
};
const cartSlice = createSlice({
  name: 'cart',
  initialState,
  reducers: {
    addToCart: (state, action) => {
      const existingItem = state.items.find((item) => item.id === action.payload.id);
      if (existingItem) {
        existingItem.quantity += 1;
      } else {
        state.items.push({ ...action.payload, quantity: 1 });
      }
      state.total += action.payload.price;
    },
    removeFromCart: (state, action) => {
      const itemIndex = state.items.findIndex((item) => item.id === action.payload.id);
      if (itemIndex >= 0) {
        state.total -= state.items[itemIndex].price * state.items[itemIndex].quantity;
      }
    },
  },
});
export const { addToCart, removeFromCart } = cartSlice.actions;
export default cartSlice.reducer;
```

- *Le fichier productSlice.jsx*

```jsx
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
  products: [],
  filteredProducts: [],
};
const productSlice = createSlice({
  name: 'product',
  initialState,
  reducers: {
    setProducts: (state, action) => {
      state.products = action.payload;
      state.filteredProducts = action.payload;
    },
    filterProducts: (state, action) => {
      const { category } = action.payload;
      console.log('Catégorie reçue :', category);
      state.filteredProducts = category
        ? state.products.filter((product) => product.category === category)
        : state.products;
      console.log('Produits filtrés :', state.filteredProducts);
    },
  },
});
export const { setProducts, filterProducts } = productSlice.actions;
export default productSlice.reducer;
```

- *Le fichier store.jsx*

```jsx
import { configureStore } from '@reduxjs/toolkit';
import productReducer from './productSlice';
import cartReducer from './cartSlice';
const store = configureStore({
  reducer: {
    product: productReducer,
    cart: cartReducer,
  },
});
export default store;
```

- *Le fichier app.jsx*

```jsx
import React from 'react';
import { Routes, Route, Link } from 'react-router-dom';
import HomePage from './pages/HomePage';
import ProductPage from './pages/ProductPage';
import Cart from './components/Cart';
const App = () => {
  return (
    <>
      <nav>
        <Link to="/">Accueil</Link> | <Link to="/cart">Panier</Link>
      </nav>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/product/:id" element={<ProductPage />} />
        <Route path="/cart" element={<Cart />} />
      </Routes>
    </>
  );
};
export default App;
```

- *Le fichier index.jsx*

```jsx
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { BrowserRouter } from 'react-router-dom';
import App from './App';
import store from './store/store';
import './styles/App.css';
ReactDOM.render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

- *Le fichier app.css*

```css
body {
  font-family: Arial, sans-serif;
  background: #f7f7f7;
  margin: 0;
  padding: 0;
}
h1 {
  text-align: center;
  font-size: 2.5rem;
  margin: 20px 0;
  color: #333;
}
.filters {
  display: flex;
  justify-content: center;
  margin: 20px 0;
}
select {
  padding: 10px;
  font-size: 1rem;
  border: 1px solid #ddd;
  border-radius: 5px;
  appearance: auto;
  cursor: pointer;
}
select:focus {
  outline: none;
  border-color: #007bff;
}
```

- *Le fichier Cart.css*

```css
.cart-container {
    max-width: 800px;
    margin: 20px auto;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 10px;
    background-color: #f9f9f9;
}
.cart-container h2 {
    text-align: center;
    margin-bottom: 20px;
}
.cart-items {
    display: flex;
    flex-direction: column;
    gap: 15px;
}
.cart-item {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
    background-color: #fff;
}
.cart-item-info {
    max-width: 70%;
}
.cart-item-info h3 {
    margin: 0;
    font-size: 1.2rem;
}
.cart-item-info p {
    margin: 5px 0;
    font-size: 1rem;
}
.remove-button {
    background-color: #e74c3c;
    color: white;
    border: none;
    border-radius: 5px;
    padding: 5px 10px;
    cursor: pointer;
}
.remove-button:hover {
    background-color: #c0392b;
}
.cart-total {
    margin-top: 20px;
    text-align: right;
}
.cart-total h3 {
    font-size: 1.5rem;
    margin: 10px 0;
}
.checkout-button {
    background-color: #27ae60;
    color: white;
    border: none;
    border-radius: 5px;
    padding: 10px 15px;
    cursor: pointer;
    font-size: 1rem;
}
.checkout-button:hover {
    background-color: #229954;
}
```

- *Le fichier ProductCard.css*

```css
.product-card {
    border: 1px solid #ddd;
    border-radius: 10px;
    overflow: hidden;
    text-align: center;
    background: #fff;
    padding: 15px;
    margin: 10px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
    transition: transform 0.2s;
}
.product-card:hover {
    transform: translateY(-5px);
}
.product-card img {
    width: 100%;
    max-height: 150px;
    object-fit: cover;
    margin-bottom: 10px;
}
.product-card h3 {
    font-size: 1.2rem;
    color: #333;
}
.product-card p {
    font-size: 1rem;
    color: #555;
    margin: 5px 0;
}
.product-card button {
    padding: 10px 15px;
    border: none;
    background-color: #007bff;
    color: white;
    border-radius: 5px;
    cursor: pointer;
    transition: background-color 0.2s ease;
}
.product-card button:hover {
    background-color: #0056b3;
}
```

- *Le fichier ProductPage.css*

```css
.product-page {
    display: flex;
    flex-direction: column;
    align-items: center;
    gap: 20px;
    padding: 20px;
    background: #fff;
    border-radius: 10px;
    margin: 20px auto;
    max-width: 600px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}
.product-page h1 {
    font-size: 2rem;
    color: #333;
}
.product-page p {
    font-size: 1rem;
    color: #555;
    margin: 5px 0;
}
.product-page button {
    padding: 10px 15px;
    border: none;
    background-color: #007bff;
    color: white;
    border-radius: 5px;
    cursor: pointer;
    transition: background-color 0.2s ease;
}
.product-page button:hover {
    background-color: #0056b3;
}
```

- *Exemple du fichier products.json*

```json
[
  {
    "id": 1,
    "title": "Produit A",
    "description": "Un excellent produit.",
    "price": 15.99,
    "category": "electronics"
  },
  {
    "id": 2,
    "title": "Produit B",
    "description": "Un produit de qualité.",
    "price": 25.99,
    "category": "clothing"
  }
]
```

Le résultat :