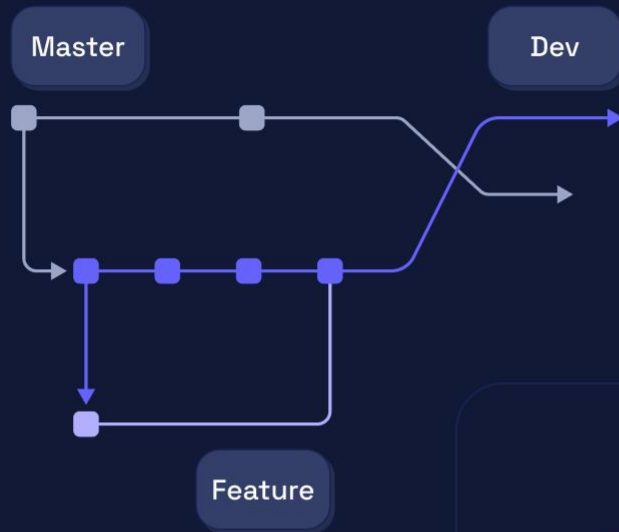


CHAPITRE 2 : ROUTER REACT



```
1 <Router>
2   <Nav />
3   <Route exact path='/blog' component={Blog} />
4   <Route exact path='/blog/:id' component={BlogPost} />
5 </Router>
```

1. Introduction au Router React

React Router est une bibliothèque puissante qui permet la navigation et le routage dans les applications React. Il facilite la création d'applications d'une page (SPA) avec plusieurs vues et permet une navigation transparente entre les différents composants. React Router améliore l'expérience utilisateur en fournissant un paradigme de navigation Web familier au sein d'une application à page unique.

Type d'application React (SPA, PWA, SSR, Hybrid)

2. Configuration du Router React

Pour configurer React Router dans un projet React :

- Installer le package **react-router-dom** en utilisant npm.
`npm install react-router-dom`
- Configurer les routes à l'aide de la fonction `createBrowserRouter` pour envelopper l'application et le tableau des routes (des objets de type `RouteObject`) afin de définir différents paths et leurs composants correspondants.

```
interface RouteObject {  
  path?: string;  
  index?: boolean;  
  children?: React.ReactNode;  
  caseSensitive?: boolean;  
  id?: string;  
  loader?: LoaderFunction;  
  action?: ActionFunction;  
  element?: React.ReactNode | null;  
  hydrateFallbackElement?: React.ReactNode | null;  
  errorElement?: React.ReactNode | null;  
  Component?: React.ComponentType | null;  
  HydrateFallback?: React.ComponentType | null;  
  ErrorBoundary?: React.ComponentType | null;  
  handle?: RouteObject["handle"];  
  shouldRevalidate?: ShouldRevalidateFunction;  
  lazy?: LazyRouteFunction<RouteObject>;  
}
```

3. Configuration de route

La configuration de route implique la définition des routes à l'aide de React Router. Cela inclut les routes de base, les routes imbriquées et les routes dynamiques qui répondent aux paramètres d'URL et query.

Exemple :

```
createBrowserRouter([
  {
    path: "/",
    element: <HomePage />,
  },
  {
    path: "about",
    element: <AboutPage />,
  },
]);
```

4. Navigation entre les routes

La navigation entre les routes peut être effectuée à l'aide de méthodes telles que :

- **Liens** : Utiliser les composants Link et NavLink pour naviguer entre les différentes vues.
- *<Link> est un composant de navigation de base, tandis que <NavLink> est une version spécialisée de <Link> avec des fonctionnalités de style supplémentaires pour mettre en évidence le lien actif.*
- **Navigation programmatique** : Utilisation de l'objet history pour naviguer programmatiquement dans les composants.
- **Redirection** : Rediriger les utilisateurs vers différentes routes en fonction de certaines conditions.

4. Navigation entre les routes

Remarque : Utilisez les composants `Link` et `NavLink` exclusivement dans un contexte de Router créé par `createBrowserRouter`.

Exemple :

```
<Link to="about">About Us</Link>
```

5. Rendu des routes

Le rendu des routes consiste à déterminer comment les composants sont rendus en fonction des routes. Cela inclut des techniques pour rendre des composants spécifiques pour différentes routes, gérer les erreurs 404 et implémenter des mises en page imbriquées pour des structures d'interface utilisateur complexes.

- Utiliser des `children` pour définir des routes imbriquées.
- Utiliser `Outlet` pour afficher l'élément de route et créer une mise en page pour le parent

Exemple de structure d'application

Voici un exemple simplifié d'une application React utilisant :

```
import { RouterProvider } from 'react-router-dom'
import router from './router.jsx'

function App() {
  return (
    <>
      <RouterProvider router={router} />
    </>
  )
}

export default App
```

```
{
  element: <Index />,
  children:[
    {
      path: '/',
      element:<HomePage />
    },{
      path: "about",
      element: <AboutPage />,
    },]
}
function Index() {
  return (
    <>
      <NavBar />
      <Outlet />
    </>
  )}
}
```

TP

Créez une application React avec les pages Connexion, Inscription et Accueil, ainsi qu'une barre de navigation et intégrer React Router

ROUTES IMBRIQUÉES (NESTED) DANS REACT ROUTER

Les routes imbriquées vous permettent de définir des routes au sein des routes, ce qui vous permet de créer des structures d'interface utilisateur complexes et imbriquées dans vos applications React. Cette documentation explique le concept de routes imbriquées et fournit des exemples de syntaxe clé à l'aide d'un exemple d'application.

1. Explication des routes imbriqués

Les routes imbriqués font référence à la pratique consistant à définir des routes au sein d'autres routes. Cela vous permet d'organiser la hiérarchie de l'interface utilisateur de votre application et de gérer le contenu imbriqué plus efficacement.

2. Exemples de syntaxe

Utilisation de Outlet pour rendre les composants de route

Le composant **Outlet** est utilisé dans une route parent pour rendre les routes imbriquées définies à l'intérieur.

```
import { Outlet } from "react-router-dom";
import NavBar from "../Layout/NavBar.jsx";

function Index() {
  return (
    <>
      <NavBar />
      <div className="container">
        <Outlet />
      </div>
    </>
  );
}
```

Définir les routes pour les children

Les routes children sont définis dans un route parent en utilisant la propriété children.


```
const parentRoute = {  
  path: "/parent",  
  element: <ParentComponent />,  
  children: [{ path: "child", element:  
<ChildComponent /> }],  
};
```

Exportation de routes imbriquées vers un autre Router

Les routes imbriquées peuvent être exportées à partir d'un fichier et importées dans une autre configuration de Router.

```
// In the parent routes file
import NestedRoutes from "../NestedRoutes";
const parentRoute = {
  path: "/parent",
  element: <ParentComponent />,
  children: NestedRoutes,
};
export default parentRoute;

// In the nested routes file
const NestedRoutes = [
  {
    path: "child",
    element: <ChildComponent />,
  },
];
export default NestedRoutes;
```


The background is a dark blue digital space with a perspective grid of lines receding into the distance. Two bright horizontal light streaks, one above and one below the text, add a sense of depth and technology.

PARAMETRES DE ROUTE

1. Accès aux paramètres de route à l'aide de `useParams()` :

- `useParams()` est un hook fourni par React Router DOM qui vous permet d'accéder aux paramètres à partir du chemin de la route actuelle.
- Pour utiliser `useParams()`, l'importer depuis **react-router-dom** :

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

- Dans votre composant fonctionnel, appelez `useParams()` pour obtenir un objet contenant tous les paramètres de route :

```
const { id } = useParams();
```

- Ici, **id** est le nom du paramètre de route défini dans votre configuration de route.
- Par exemple, si votre route est définie comme `/users/:id` et que l'URL actuelle est `/users/123`, `useParams()` retournera un objet avec `{ id : '123' }`.

2. Accès aux paramètres de query à l'aide de `useLocation()` :

- **`useLocation()`** est un autre hook fourni par React Router DOM qui vous donne accès à l'objet de localisation actuel.
- Pour utiliser **`useLocation()`**, importez-le depuis **`react-router-dom`** :

javascriptCopy code

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

- Dans votre composant fonctionnel, appelez **`useParams()`** pour obtenir l'objet location :

javascriptCopy code

```
const location = useLocation();
```

- À partir de l'objet location, vous pouvez accéder à la chaîne de query à l'aide de la propriété **`search`**.
- Pour obtenir un paramètre de query spécifique à partir de la chaîne de query, vous pouvez utiliser **`URLSearchParams`** :

- À partir de l'objet `location`, vous pouvez accéder à la chaîne de query à l'aide de la propriété **`search`**.
- Pour obtenir un paramètre de query spécifique à partir de la chaîne de query, vous pouvez utiliser **`URLSearchParams`** :

javascriptCopy code

```
const myParam = new URLSearchParams(location.search).get('myParam');
```

- Ici, **`myParam`** est le nom du paramètre de query que vous souhaitez récupérer.
- Par exemple, si l'URL actuelle est **`/path? myParam=value`**, **`myParam`** sera `'value'`.

En utilisant les **`useParams()`** et **`useLocation()`**, vous pouvez facilement accéder aux paramètres de route et de query dans vos composants React, ce qui vous permet de créer des interfaces utilisateur dynamiques et flexibles basées sur l'URL actuelle.

7. Gardiens de route et authentication :

Des gardes de route et des mécanismes d'authentification peuvent être mis en œuvre pour protéger certaines routes et restreindre l'accès en fonction de l'authentification de l'utilisateur. Cela implique de créer des composants d'ordre supérieur ou une logique personnalisée pour contrôler l'accès à des routes spécifiques.

```
<>
  <NavBar />
  <div className="container">
    {!isLoggedIn && location.pathname === "/protected" ? (
      <Navigate to="/login" />
    ) : (
      <Outlet />
    )}
  </div>
</>
```

8. Fonctions avancées :

Les fonctionnalités avancées de React Router incluent des animations de transition de route, lazy loading routes, et code splitting pour optimiser les performances et améliorer l'expérience utilisateur.

Transition des routes

npm install gsap

```
import gsap from "gsap";  
useEffect(() => {  
  gsap.fromTo(".level1", { opacity: 0, x: 50 }, {  
    opacity: 1, x: 0 });  
}, [location]);
```


Lazy Loading

```
import { lazy } from "react";  
const Protected =  
lazy(() => import("./components/pages/Protected"));
```

et pour notre mise en page nous devrions ajouter
Suspense avec Loading Component

```
<Suspense fallback={<div>Loading...</div>}>  
  <Outlet />  
</Suspense>
```

TP protected

- Ajouter une page protected.
- Protéger cette page depuis Index.jsx
- Si le user submit admin:admin en Login, ajouter une variable dans local storage.
- Vérifier local Storage s'il contient une variable non vide, pour afficher la page protected

TP transition

- Ajouter une class level1 dans le parent de Outlet, dans le fichier index.jsx
- Lancer le code suivant chaque fois location changer dans le fichier NavBar.jsx

```
gsap.fromTo(".level1", { opacity: 0, x: 50 }, { opacity: 1, x: 0 });
```

Résumé :

Dans ce chapitre, vous avez appris à utiliser React Router pour gérer la navigation et le routage dans les applications React.

Vous avez acquis une compréhension de la configuration des routes, des méthodes du rendu des routes, des gardes des routes et des fonctionnalités avancées de React Router.

En maîtrisant ces concepts, vous pouvez créer des applications web dynamiques et navigables avec une expérience utilisateur et des performances améliorées.

Ressources :

- [React Router Documentation](#)

CHAPITRE 3 : GESTION DE STATE



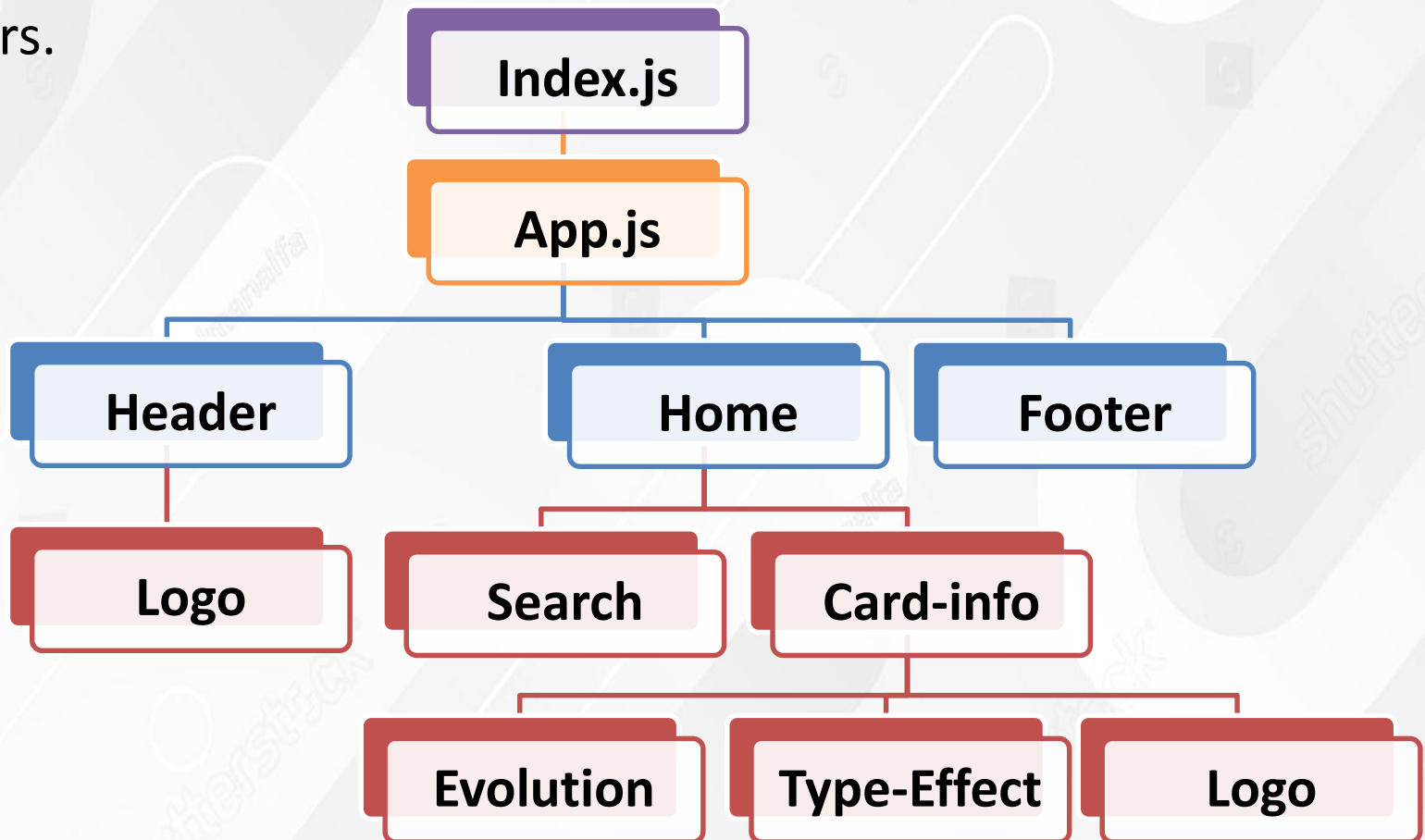


À mesure que la complexité des applications augmente, la gestion de state devient de plus en plus difficile. Voici quelques-uns des défis courants :

- **Perçage d'hélices** : Le passage de state à travers plusieurs couches de composants imbriqués peut conduire au perçage d'hélices, où les composants intermédiaires doivent passer des hélices sans les consommer directement.

```
import React, { useState } from "react";
const DeepestComponent = ({ value }) => {
  return <div>Value: {value}</div>;
};
// Intermediate component passing the prop
const IntermediateComponent = ({ value }) => {
  return <DeepestComponent value={value} />;
};
// Top-level component where the state resides
const TopComponent = () => {
  const [value, setValue] = useState("Hello, prop drilling!");
  return (
    <div>
      {/* Prop drilling: passing the prop down through multiple layers */}
      <IntermediateComponent value={value} />
    </div>
  );
};
export default TopComponent;
```

- **Hiérarchies de composants complexes** : Dans les grandes applications avec des hiérarchies de composants fortement imbriquées, le maintien de la cohérence state et sa mise à jour entre les composants deviennent complexes et sujettes aux erreurs.



- **Global** : Certains États doivent être accessibles dans différentes parties de l'application, ce qui entraîne la nécessité de solutions de gestion de l'État à l'échelle global.

Dans React, state fait référence à toutes les données qui peuvent changer au fil du temps au sein d'un composant. Cela peut inclure l'entrée de l'utilisateur, les réponses du serveur ou même les états de l'interface utilisateur, comme basculer un menu déroulant. La gestion de cet état est cruciale pour garantir que notre application reste synchronisée avec les interactions des utilisateurs et les sources de données externes.

Une gestion efficace de l'état améliore non seulement les performances et la réactivité de nos applications, mais améliore également la maintenabilité et l'évolutivité du code. En centralisant la logique state et en la découplant des composants individuels, nous pouvons créer un code modulaire et réutilisable qui est plus facile à comprendre et à maintenir.

Variable mutable et immuable :

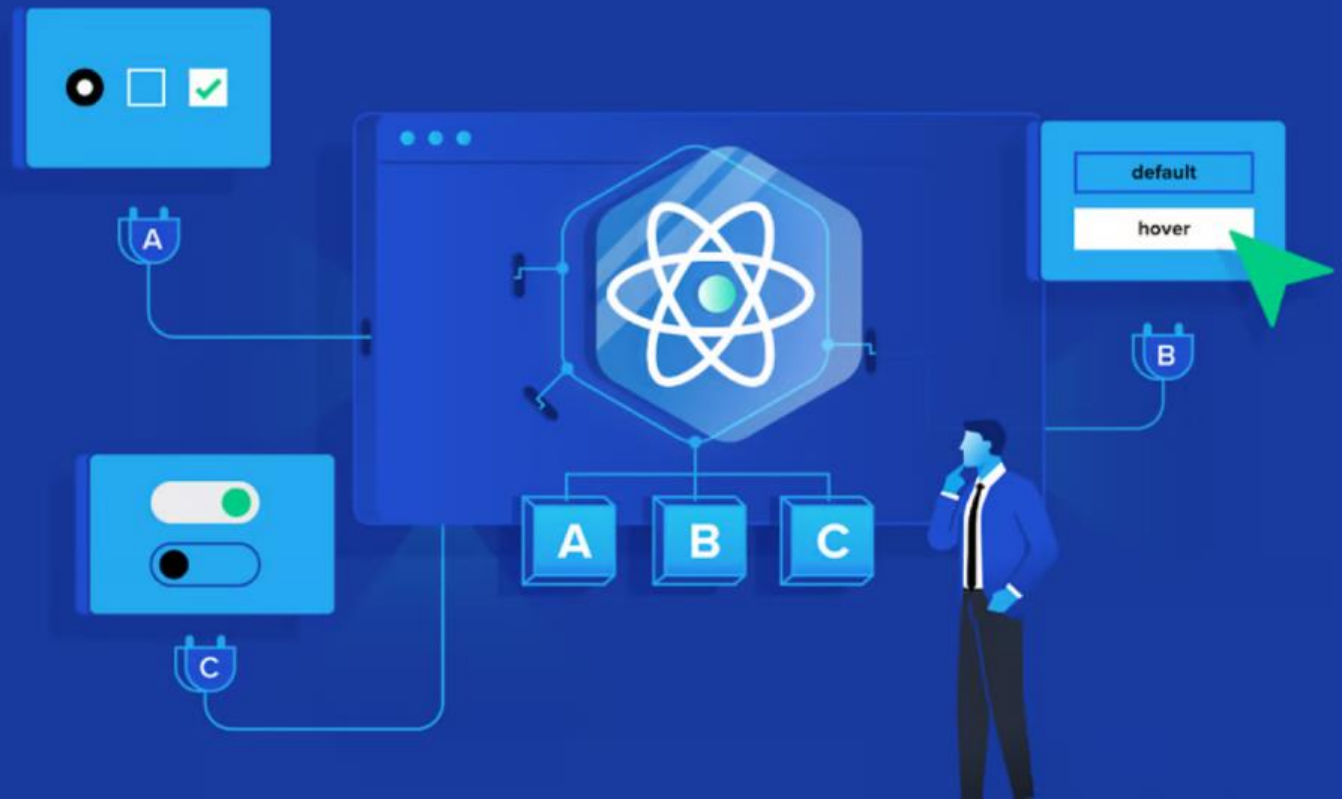
- **Variables mutables en JavaScript** : les structures de données natives de JavaScript, telles que les tableaux et les objets, sont mutables, ce qui signifie que leurs valeurs peuvent être modifiées après la création. Bien que les variables mutables offrent une flexibilité, elles peuvent entraîner un comportement inattendu, en particulier lorsqu'elles sont partagées entre les composants.
- **Variables immuables** : L'immutabilité fait référence au concept de données immuables. Dans React, l'immutabilité garantit que les changements state sont prévisibles et traçables, améliorant la stabilité et les performances de l'application.


```
let mutableArray = [1, 2, 3];  
// Direct mutation  
mutableArray.push(4);  
console.log(mutableArray); // Output: [1, 2, 3, 4]  
let immutableArray = [1, 2, 3];  
// Creating a new array with the desired changes  
let newArray = [...immutableArray, 4]; // = [1, 2, 3, 4]  
console.log(immutableArray); // Output: [1, 2, 3]
```

Dans l'approche mutable, la mutation directe du tableau peut entraîner des conséquences involontaires, en particulier dans une application React à état partagé. Inversement, l'approche immuable garantit que les données originales restent inchangées, favorisant la prévisibilité et réduisant le risque d'effets secondaires involontaires.

LA GESTION DE STATE DANS REACT

ReactJs State management Tools



2. Approches de la gestion de state dans React

React propose plusieurs solutions intégrées et tierces pour gérer l'état, chacune avec ses propres avantages et cas d'utilisation. Voici quelques-unes de ces approches :

- **État des composants locaux** : Les composants React peuvent gérer leur propre état à l'aide du Hook `useState`, qui fournit un moyen simple et léger de gérer l'état au sein des composants individuels.
- **API de contexte** : permet de partager l'état sur plusieurs composants sans forage d'hélice. Il est idéal pour gérer l'état global ou partagé de manière centralisée. L'approche combine le Hook `useReducer` pour gérer les transitions states complexes avec l'API Context pour partager l'état entre les composants.

- **useReducer Hook** : Essentiel pour gérer l'état dans React, en particulier dans les scénarios complexes comme la gestion de plusieurs sous-valeurs ou dépendances state. Il suit un modèle structuré similaire à Redux mais est généralement utilisé pour l'état des composants locaux. Il impose l'immuabilité dans la fonction de réducteur, empêchant la mutation directe state.

Exemple :

- 1- Définir un **CounterContext** en utilisant **React.createContext()** pour gérer l'état global. Créez une fonction **counterReducer** pour gérer les mises à jour state en fonction des actions distribuées.

```
import React, { useContext, useReducer } from 'react';
// Context for global state management
const CounterContext = React.createContext();
// Reducer function to manage state updates
const counterReducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};
```

2- À l'intérieur du composant App, utilisez le Hook `useReducer` pour gérer l'état global et rendre le composant Counter.

```
<CounterContext.Provider value={{ count: state.count, dispatch }}>
  <div>
    <p>Global Count: {state.count}</p>
    <Counter />
  </div>
</CounterContext.Provider>
```

3- Le composant Counter accède à l'état de count global et à la fonction de répartition à l'aide du Hook `useContext` dans le CounterContext.

```
// Counter component to display and update the local count
const Counter = () => {
  const { count, dispatch } = useContext(CounterContext);
  return (
    <div>
      <p>Local Count: {count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
};
```

Explication :

- Le Hook `useReducer` du composant `App` gère l'état de `count` global et fournit une fonction de répartition pour mettre à jour l'état en fonction des actions.
- Le composant `Counter` utilise l'état de `count` global et la fonction de répartition à l'aide du Hook `useContext` dans le `CounterContext`, ce qui lui permet d'afficher et de mettre à jour le `count` local.

bibliothèques de gestion de state

- Redux
- Zustand (conseillé)
- Hookstate
- Recoil
- Rematch

Zustand

- Une bibliothèque de gestion de state conçue pour simplifier le processus de gestion et de partage de state dans l'Application.
- Zustand apporte une nouvelle approche à la gestion d'état, offrant une solution à la fois légère et puissante.

Créer le Store

```
import { create } from 'zustand'

const useStore = create((set) => ({
  count: 1,
  inc: () => set(
    (state) => ({ count: state.count + 1 })
  ),
}))
```

Export et import store

- export default useStore;
- import useStore from 'store.jsx';

Utilisation dans un composant

```
function Counter() {  
  const { count, inc } = useStore()  
  return (  
    <div>  
      <span>{count}</span>  
      <button onClick={inc}>one up</button>  
    </div>  
  )  
}
```

TP

- Télécharger le fichier solution.html
- Créer les 4 composants dans des fichiers:
 TodoItem.jsx
 TodoList.jsx
 TodoAdd.jsx
 App.jsx

TP

- Créer un fichier todo.html
- Importer le fichier App.jsx et le booter dans l'HTML (créer main.jsx)
- Créer store.js pour gérer le state globalement