**ReactHooks**

1. **Initialisation**
2. **C’est quoi les hooks**

Les "Hooks" sont une nouvelle fonctionnalité ajoutée dans la version 16.8 de React, qui nous permet d'utiliser les fonctionnalités de React sans avoir à écrire une classe. Exemple : L'état d'un composant Les "Hooks" ne fonctionnent pas à l'intérieur des classes.

1. **Pourquoi les hooks**

Les Hooks en React offrent plusieurs avantages et raisons convaincantes pour lesquelles on devrait les utiliser :

1. **Gestion de l'état local** : Les Hooks permettent de gérer l'état local dans les composants fonctionnels sans avoir besoin de créer des classes. Cela rend le code plus concis et plus facile à lire.
2. **Réutilisation de la logique** : Les Hooks permettent d'extraire la logique de composants en dehors des composants eux-mêmes, ce qui facilite la réutilisation de cette logique dans différents composants.
3. **Simplicité du code** : Les Hooks éliminent la nécessité d'utiliser des classes et de comprendre les concepts liés au mot-clé "this", rendant le code plus simple et plus intuitif.
4. **Fonctionnalités de cycle de vie** : Les Hooks permettent d'utiliser des fonctionnalités de cycle de vie telles que **useEffect**, qui facilite la gestion des effets secondaires dans les composants fonctionnels.
5. **Séparation des préoccupations** : Avec les Hooks, la logique liée à l'état et aux effets secondaires est séparée de la logique de rendu du composant, ce qui améliore la lisibilité du code et facilite la maintenance.
6. **Meilleure performance** : Les Hooks peuvent aider à améliorer les performances du composant en évitant la création de nouvelles instances de fonctions liées aux événements lors des rendus, ce qui peut être coûteux en termes de performances.
7. **Facilité de test** : Les composants basés sur les Hooks sont plus faciles à tester car la logique peut être testée de manière isolée sans dépendre de l'infrastructure de classe.
8. **Compatibilité avec les futures mises à jour de React** : React encourage l'utilisation des Hooks et les améliorera probablement dans les futures versions. Ils représentent la direction future de développement dans React.

En résumé, l'utilisation des Hooks en React améliore la lisibilité, la réutilisabilité, la séparation des préoccupations et les performances du code. Ils facilitent également le développement et sont alignés sur les évolutions futures de React. En tant que développeur React, il est donc judicieux de considérer les Hooks comme une approche préférable pour gérer la logique dans les composants fonctionnels.

1. **Pourquoi les composant fonctionnels en React**

Les composants fonctionnels en React sont utilisés pour plusieurs raisons importantes :

1. **Simplicité** : Les composants fonctionnels sont plus simples et concis par rapport aux composants de classe. Ils sont définis sous forme de fonctions JavaScript, ce qui les rend plus faciles à comprendre, à écrire et à maintenir.
2. **Performance** : Les composants fonctionnels sont généralement plus performants que les composants de classe. Ils évitent la surcharge liée à la création d'instances de classes et aux mécanismes internes associés aux classes.
3. **Pas de gestion du "this"** : Dans les composants fonctionnels, il n'y a pas de nécessité de se soucier du mot-clé "this". Cela évite les erreurs courantes liées à la liaison des méthodes dans les classes.
4. **Hooks** : Avec l'introduction des Hooks dans React, les composants fonctionnels peuvent désormais gérer leur propre état local, ainsi que d'autres fonctionnalités telles que les effets secondaires, auparavant réservées aux classes. Les Hooks permettent de réutiliser la logique entre différents composants fonctionnels sans avoir à utiliser des classes.
5. **Meilleure compatibilité avec les futures mises à jour de React** : React encourage l'utilisation des composants fonctionnels et des Hooks. Ils représentent la direction future du développement dans React, ce qui signifie qu'il y aura probablement plus d'améliorations et de fonctionnalités associées aux composants fonctionnels dans les futures versions.
6. **Facilité de test** : Les composants fonctionnels sont plus faciles à tester car ils sont basés sur des fonctions pures. Cela facilite la mise en œuvre de tests unitaires sans dépendre de l'infrastructure de classe.
7. **Facilité de réutilisation** : Étant donné que les composants fonctionnels sont plus simples et ne dépendent pas de l'héritage de classe, ils peuvent être réutilisés plus facilement dans différentes parties de l'application.

En résumé, les composants fonctionnels en React offrent une meilleure simplicité, une meilleure performance, une meilleure réutilisation, une meilleure compatibilité avec les futures mises à jour de React, et ils sont plus faciles à tester. Avec l'introduction des Hooks, ils sont désormais encore plus puissants et flexibles pour gérer l'état et les effets secondaires dans les composants.

1. **Point important a savoire**

1- "N'utilisez les Hooks qu'au niveau supérieur" N'utilisez pas les Hooks à l'intérieur des boucles, des conditions ou des fonctions imbriquées.

2- "N'utilisez les Hooks que depuis des fonctions React" Appelez-les uniquement à l'intérieur des fonctions React.

1. **useState**

**useState** est un Hook en React qui permet aux composants fonctionnels de déclarer et de gérer un état local en utilisant la déstructuration d'un tableau en une seule ligne.

* 1. **UseState Hook**

import React ,{useState} from 'react'

function App() {

  const [count,setCount]=useState(0)

  return (

    <div className="App">

      Count:

      <button onClick={()=>setCount(count+1)}>{count}</button>

    </div>

);

}

export default App;

* 1. **useState avec l’ancien état**

import React ,{useState} from 'react'

function App() {

const [count,setCount]=useState(0);

const AddFive=()=>{

for(let i=0;i<5;i++){

//setCount(count +1) c'est pas la bonne manniere

// il ajouter 1 seulement une seule fois pas 5 il faut

        //faire comme e bas pour que ca marche

        setCount(prev=>prev+1)

    }

  }

  return (

    <div className="App">

      Count:

      <button onClick={()=>AddFive()}>{count}</button>

    </div>

  );

}

export default App;

* 1. **useState avec un obj**

import React, { useState } from "react";

function App() {

  const [name, setName] = useState({ firstName: "", lastName: "" });

  return (

    <form>

      <input

        type="text"

        value={name.firstName}

        onChange={(e) => setName({ ...name, firstName: e.target.value })}

      />

      <input

        type="text"

        value={name.lastName}

        onChange={(e) => setName({ ...name, lastName: e.target.value })}

      />

      <h2>Your first name is - {name.firstName}</h2>

      <h2>Your last name is - {name.lastName}</h2>

      <h2>{JSON.stringify(name)}</h2>

    </form>

  );

}

* 1. **useState avec un array**

import React, { useState } from "react";

function App() {

  const [items, setItems] = useState([]);

  const addItem = () => {

    setItems([

      ...items,

      {

        id: items.length,

        value: Math.floor(Math.random() \* 10) + 1,

      },

    ]);

  };

  return (

    <div>

      <button onClick={addItem}>Add a number</button>

      <ul>

        {items.map((item) => (

          <li key={item.id}>{item.value}</li>

        ))}

      </ul>

  </div>

);

}

* 1. **Résume**

Le Hook **useState** vous permet d'ajouter un état à des composants fonctionnels.

Dans les classes, l'état est toujours un objet. Avec le Hook **useState**, l'état n'a pas besoin d'être un objet.

Le Hook **useState** retourne un tableau avec 2 éléments. Le premier élément est la valeur actuelle de l'état, et le deuxième élément est une fonction pour définir l'état (state setter).

Si la nouvelle valeur de l'état dépend de la valeur précédente, vous pouvez passer une fonction à la fonction de définition de l'état.

Lorsque vous travaillez avec des objets ou des tableaux, assurez-vous toujours de propager votre variable d'état (state variable) et ensuite appelez la fonction de définition de l'état.

1. **useEffect**
2. **Introduction**

Le Hook Effect vous permet d'effectuer des effets secondaires dans des composants fonctionnels. Il remplace étroitement componentDidMount, componentDidUpdate et componentWillUnmount. Si vous souhaitez appeler une fonction dans useEffect, il est préférable de créer cette fonction à l'intérieur du useEffect. Nous pouvons utiliser plusieurs useEffect dans une seule fonction Hook React.

1. **useEffect après chaque render**

const [count, setCount] = useState(0);

  useEffect(() => {

    document.title = `You clicked ${count} times`;

  });

  //UseEffect Run After every Runder

1. **useEffect après chaque render mais avec une condition**

Appliquer l'effet après chaque rendu crée parfois de nombreux problèmes, donc nous avons besoin d'utiliser l'effet conditionnellement (Conditionally Effect).

1. **useEffect runj une seule fois**

Pour faire en sorte que **useEffect** s'exécute uniquement une fois (au moment du montage initial) dans un composant fonctionnel, vous pouvez passer un tableau vide **[]** en tant que deuxième argument du hook **useEffect**. Ce tableau vide sert de liste de dépendances, et étant donné qu'il est vide, l'effet ne sera exécuté qu'une seule fois après le rendu initial.

  useEffect(() => {

    console.log("UseEffect alled")

    window.addEventListener('mousemove',logMousePosition);

  },[]);

1. **useEffect avec les cleanup**

Lorsque vous utilisez **useEffect**, vous pouvez également inclure une fonction de nettoyage (cleanup) qui sera exécutée lorsque le composant est démonté ou lorsqu'une dépendance change de valeur. Cette fonction de nettoyage est utile pour effectuer des tâches de nettoyage, annuler des abonnements, arrêter des timers ou toute autre opération de nettoyage nécessaire lorsque le composant n'est plus utilisé.

Voici un exemple d'utilisation de **useEffect** avec une fonction de nettoyage :

const [count, setCount] = useState(0);

useEffect(() => {

  console.log('Effect runs');

  // Exemple de nettoyage : nous créons un timer pour afficher un message toutes les secondes.

  const timerId = setInterval(() => {

    setCount(prev=>prev +1)

    console.log('Tick');

  }, 1000);

  // La fonction de nettoyage sera exécutée lorsque le composant est démonté ou lorsque la dépendance 'count' change.

  return () => {

    console.log('Cleanup: Stopping timer');

    clearInterval(timerId);

  };

}, [count]); // En incluant 'count' dans le tableau de dépendances, l'effet et le cleanup seront réexécutés à chaque changement de la valeur de 'count'

1. **useContext**

**useContext** est un hook en React qui permet d'accéder à la valeur actuelle d'un contexte donné dans un composant fonctionnel. Il est utilisé pour éviter de propager les données à travers de nombreux niveaux d'arborescence de composants, en permettant aux composants enfants d'accéder directement aux données du contexte dont ils ont besoin.

Le contexte fournit un moyen de transmettre des données à travers l'arborescence des composants sans avoir à transmettre manuellement des props à chaque niveau.

App.js

import "./App.css";

import React from "react";

// import HookMouse from "./Component/HookMouse";

import ComponentC from "./Component/ComponentC";

export const UserContext=React.createContext()

export const ChannelContext=React.createContext()

function App() {

  return (

    <div>

      <UserContext.Provider value={'Vishwas'}>

      <ChannelContext.Provider value={'codeEvolution'}>

      <ComponentC/>

      </ChannelContext.Provider>

      </UserContext.Provider>

    </div>

  );

}

export default App;

ComponentF.js

import React,{ useContext} from 'react'

import { UserContext ,ChannelContext } from '../App'

export default function ComponentF() {

    const user=useContext(UserContext)

    const channel=useContext(ChannelContext)

  return (

    <div>{user} {channel}</div>

  )

}

1. **useReducer**
2. **Definition**

Le hook **useReducer** est un hook intégré à React qui permet de gérer l'état local d'un composant fonctionnel en utilisant une approche basée sur le concept de "réducteur" (reducer). Il est utilisé pour gérer des états complexes ou des logiques de mise à jour plus avancées.

**useState** est idéal pour les états locaux simples sans logique de mise à jour complexe, tandis que **useReducer** est recommandé pour les états plus complexes avec des logiques de mise à jour avancées. Si vous n'avez pas besoin d'une gestion d'état complexe, **useState** est généralement suffisant et plus simple à mettre en œuvre. Cependant, si votre état devient plus compliqué, **useReducer** peut fournir une meilleure structure pour gérer l'état et les actions associées.

L'utilisation de **useReducer** est similaire à celle de Redux, mais il est généralement préféré pour les états locaux à l'intérieur d'un composant, tandis que Redux est utilisé pour gérer l'état global de l'application.

Le hook **useReducer** prend deux arguments : une fonction de réduction (reducer) et un état initial. La fonction de réduction prend deux arguments : l'état actuel et une action, puis retourne un nouvel état en fonction de l'action spécifiée.

Voici un exemple simple d'utilisation de **useReducer** :

import React, { useReducer } from 'react';

// Fonction de réduction (reducer)

const reducer = (state, action) => {

  switch (action.type) {

    case 'INCREMENT':

      return { count: state.count + 1 };

    case 'DECREMENT':

      return { count: state.count - 1 };

    default:

      return state;

  }

};

const ComponentWithReducer = () => {

  // Utilisation de useReducer avec le réducteur et l'état initial

  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (

    <div>

      <p>Count: {state.count}</p>

      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>

      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>

    </div>

  );

};

export default ComponentWithReducer;

Dans cet exemple, nous créons une fonction de réduction (**reducer**) qui gère l'état **count**. En utilisant **useReducer** avec cette fonction et l'état initial **{ count: 0 }**, nous obtenons un état local **state** et une fonction **dispatch** pour déclencher des actions pour mettre à jour l'état. Lorsque les boutons sont cliqués, nous envoyons des actions avec un type spécifique pour augmenter ou diminuer la valeur de **count**, et le réducteur gère la mise à jour de l'état en fonction de ces actions.

En résumé, **useReducer** est un hook puissant pour gérer l'état local complexe dans les composants fonctionnels, en utilisant une approche basée sur les réducteurs similaires à Redux.

1. **Multiple Reducer**

Nous pouvons utiliser plusieurs **useReducer** comme plusieurs états en utilisant la même fonction de réduction (reducer) ou différentes fonctions de réduction (reducers).

1. **useReducer avec useContext**

**useReducer🡺 Local state Management**

**useReducer + useContext 🡺 Global state Management**

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

App.js

import "./App.css";

import React, { useReducer } from "react";

// import HookMouse from "./Component/HookMouse";

import ComponentA from "./Component/ComponentA";

import ComponentD from "./Component/ComponentD";

import ComponentF from "./Component/ComponentF";

export const CountContext = React.createContext();

function App() {

  const initialState = 0;

  const reducer = (state, action) => {

    switch (action) {

      case "increment":

        return state + 1;

      case "decrement":

        return state - 1;

      case "reset":

        return initialState;

      default:

        return state;

    }

  };

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

  return (

    <CountContext.Provider

      value={{ countState: count, countDispatch: dispatch }}

    >

      <div>

        {count}

        <ComponentA/>

        <ComponentD />

        <ComponentF />

      </div>

    </CountContext.Provider>

  );

}

export default App;

ComponentD

import React, { useContext } from 'react'

import { CountContext } from '../App';

export default function ComponentA() {

    const countContext = useContext(CountContext);

  return (

    <>

      <div>ComponentA -{countContext.countState}</div>

     <button onClick={()=>countContext.countDispatch("increment")}>Zide</button>

     <button onClick={()=>countContext.countDispatch("decrement")}>n9sse</button>

     <button onClick={()=>countContext.countDispatch("reset")}>Rde l9lawi ki kane</button>

    </>

  )

}

ComponentD

import React, { useContext } from "react";

import {  CountContext } from "../App";

export default function ComponentD() {

    const countContext = useContext(CountContext);

    return (

      <>

        <div>ComponentD -{countContext.countState}</div>

       <button onClick={()=>countContext.countDispatch("increment")}>Zide</button>

       <button onClick={()=>countContext.countDispatch("decrement")}>n9sse</button>

       <button onClick={()=>countContext.countDispatch("reset")}>Rde l9lawi ki kane</button>

      </>

    )

}

ComponentF

import React, { useContext } from "react";

import { CountContext } from "../App";

export default function Component() {

  const countContext = useContext(CountContext);

  return (

    <>

      <div>ComponentF -{countContext.countState}</div>

     <button onClick={()=>countContext.countDispatch("increment")}>Zide</button>

     <button onClick={()=>countContext.countDispatch("decrement")}>n9sse</button>

     <button onClick={()=>countContext.countDispatch("reset")}>Rde l9lawi ki kane</button>

    </>

  );

}

1. **useReducer vs Redux**

Utiliser **useReducer** en combinaison avec **useContext** pour gérer l'état global est une alternative légère et simple à Redux. Cela vous permet de vous passer de la bibliothèque externe Redux tout en profitant des fonctionnalités de gestion d'état centralisé.

Cependant, il est important de noter que Redux offre des fonctionnalités avancées telles que les middlewares, les outils de développement Redux DevTools, et une architecture plus robuste pour la gestion d'état dans les grandes applications. Si votre application devient complexe avec de nombreux états, des interactions complexes et nécessite des fonctionnalités avancées, Redux peut être plus approprié.

1. **Memo & useMemo**
   1. **Memo**

En React, "memo" fait référence à une technique d'optimisation qui peut être appliquée à un composant fonctionnel pour éviter les rendus inutiles lorsque les props ou l'état d'un composant n'ont pas changé.

Le "React.memo" est un composant de rendu à haute performance qui s'assure que le composant n'est pas rendu à moins que ses props ne changent. Cela peut être très utile lorsque le rendu d'un composant est coûteux et que vous voulez éviter des mises à jour inutiles.

Voici comment utiliser "React.memo" pour optimiser un composant fonctionnel :

import React from 'react';

// Composant fonctionnel normal

const MyComponent = ({ propA, propB }) => {

  // Comportement et rendu du composant

  return (

    <div>

      <p>Prop A: {propA}</p>

      <p>Prop B: {propB}</p>

    </div>

  );

};

// Composant fonctionnel optimisé avec React.memo

const OptimizedComponent = React.memo(MyComponent);

export default OptimizedComponent;

Dans cet exemple, le composant "OptimizedComponent" est le résultat de l'application de "React.memo" au composant "MyComponent". Grâce à cette optimisation, le composant "OptimizedComponent" ne sera rendu que si les props "propA" ou "propB" changent. S'il n'y a pas de changement dans les props, le composant évitera un rendu inutile, ce qui peut améliorer les performances de votre application.

Il est important de noter que l'optimisation avec "React.memo" ne fonctionnera que si les props du composant sont des valeurs primitives ou des objets / tableaux immuables. Si vos props contiennent des objets ou des tableaux modifiables, "React.memo" ne sera pas en mesure de détecter les changements, et le composant sera toujours rendu. Dans ce cas, il peut être nécessaire d'utiliser des techniques supplémentaires comme "useMemo" pour optimiser davantage le rendu du composant.

* 1. **useMemo**

**useMemo** est un hook React qui permet de mémoriser (cacher) le résultat d'un calcul coûteux et de le réutiliser ultérieurement, tant que ses dépendances restent inchangées. Cela permet d'éviter de recalculer le résultat à chaque nouveau rendu, ce qui peut améliorer les performances de l'application en réduisant les calculs inutiles.

import React, { useMemo, useState } from "react";

const Counter = () => {

  const [counterOne, setCounterOne] = useState(0);

  const [counterTwo, setCounterTwo] = useState(0);

  const incrementone = () => {

    setCounterOne((counterOne) => counterOne + 1);

  };

  const incrementTwo = () => {

    setCounterTwo((counterTwo) => counterTwo + 1);

  };

  const isEven = useMemo(() => {

    console.log("Render");

    let i = 0;

    while (i < 200000000) i++;

    return counterOne % 2 === 0;

  },[counterOne]);

  return (

    <div>

      <div>

        <button onClick={incrementone}>Count One - {counterOne}</button>

        {isEven ? "Even" : "odd"}

      </div>

      <div>

        <button onClick={incrementTwo}>Count Two - {counterTwo}</button>

      </div>

    </div>

  );

};

export default Counter;

1. **useCallBack**

**useCallback** est un hook React qui vous permet de mémoriser une fonction entre les rendus du composant. Il est utilisé pour optimiser les performances en évitant la recréation inutile de fonctions, en particulier lorsque ces fonctions sont transmises en tant que props à des composants enfants.

Lorsque vous créez une fonction à l'intérieur d'un composant, cette fonction est recréée à chaque nouveau rendu du composant. Si cette fonction est passée en tant que prop à un composant enfant, cela peut entraîner des rendus inutiles de cet enfant, car il considérera que la fonction a changé à chaque nouveau rendu du parent.

En utilisant **useCallback**, vous pouvez mémoriser cette fonction, ce qui signifie qu'elle ne sera créée qu'une seule fois pendant le cycle de vie du composant, à moins que l'une de ses dépendances change. Ainsi, lorsque vous transmettez cette fonction à un composant enfant, il ne se rendra à nouveau que si la fonction a été recalculée.

const memoizedCallback = useCallback(() => {

    // Logique de la fonction à mémoriser

  }, [dependencies]);

* **memoizedCallback** est la fonction mémorisée que vous pouvez transmettre à d'autres composants en tant que prop.
* La fonction à mémoriser est définie à l'intérieur de **useCallback**.
* **dependencies** est un tableau contenant les dépendances de la fonction. Si l'une de ces dépendances change, la fonction sera recalculée, sinon, elle sera mémorisée et réutilisée lors des futurs rendus.

**useCallback** est particulièrement utile lorsque vous avez des fonctions de rappel ou des fonctions passées en tant que props à des composants enfants, car cela permet d'éviter des rendus inutiles et d'optimiser les performances de l'application.