**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'