Cours Inf3522 - Développement d'Applications N tiers

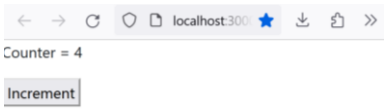
Lab 8 : Hooks

Ce lab présente l'utilisation des hooks React permettant de créer des composants fonctionnels à états sans avoir recours aux classes ES6.

Il existe certaines règles importantes pour l'utilisation des hooks dans React. Vous devez toujours appeler les hooks au niveau supérieur de votre composant fonction React. Vous ne devez pas appeler les hooks à l'intérieur de boucles, d'instructions conditionnelles ou de fonctions imbriquées.

useState

Nous sommes déjà familiers avec la fonction hook useState qui est utilisée pour déclarer des états. Créons un exemple supplémentaire d'utilisation du hook useState. Nous allons créer un exemple de compteur qui contient un bouton, et lorsque celui-ci est pressé, le compteur est augmenté de 1, comme illustré dans la capture d'écran suivante :



Tout d'abord, nous créons un composant **Counter** et déclarons un état appelé count avec la valeur initiale 0. La valeur de l'état du compteur peut être mise à jour en utilisant la fonction **setCount**. Le code est illustré dans l'extrait suivant :

```
import React, { useState } from 'react';
function Counter() {
 // État count avec une valeur initiale de 0
 const [count, setCount] = useState(0);
 return <div></div>;
}
export default Counter;
```

Ensuite, nous rendons un élément de bouton qui incrémente l'état de 1. Nous utilisons l'attribut d'événement **onClick** pour appeler la fonction **setCount**, et la nouvelle valeur est la valeur actuelle plus 1. Nous affichons également la valeur de l'état du compteur. Le code est illustré dans l'extrait suivant :

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

Maintenant, notre composant **Counter** est prêt, et le compteur est incrémenté de 1 à chaque fois que le bouton est pressé. Lorsque l'état est mis à jour, React réaffiche le composant, et nous pouvons voir la nouvelle valeur de count.

NB: En React, les événements sont nommés en utilisant la casse camelCase, par exemple onClick.

Les mises à jour de l'état sont asynchrones, vous devez donc faire attention lorsque la nouvelle valeur de l'état dépend de la valeur actuelle de l'état. Pour vous assurer que la dernière valeur est utilisée, vous pouvez passer une fonction à la fonction de mise à jour. Par exemple :

```
setCount(prevCount => prevCount + 1)
```

Maintenant, la valeur précédente est transmise à la fonction, et la valeur mise à jour est renvoyée et enregistrée dans l'état **count**. Il existe également une fonction hook appelée **useReducer** qui est recommandée lorsque vous avez un état complexe.

React utilise également le regroupement (batching) dans les mises à jour de l'état pour réduire les réaffichages. Avant la version 18 de React, le regroupement fonctionnait uniquement pour les mises à jour de l'état pendant les événements du navigateur, par exemple lors d'un clic sur un bouton. L'exemple suivant illustre l'idée de mises à jour groupées :

```
import React, { useState } from 'react';
function App() {
 const [count, setCount] = useState(0);
 const [count2, setCount2] = useState(0);
 const increment = () => {
  setCount(count + 1); // Pas encore de réaffichage
  setCount2(count2 + 1);
  // Le composant est réaffiché après toutes les mises à jour de l'état
 };
 return (
  <div>
   <p>Compteurs : {count} {count2}</p>
   <button onClick={increment}>Incrémenter</button>
  </div>
 );
}
export default App;
```

Si vous ne souhaitez pas utiliser les mises à jour groupées dans certains cas, vous pouvez utiliser l'API **flushSync** de la bibliothèque **react-dom** pour éviter le regroupement. Par exemple, vous pouvez avoir un cas où vous souhaitez mettre à jour un état avant de mettre à jour le suivant. Voici un exemple :

```
import { flushSync } from "react-dom";
// ...
const increment = () => {
 flushSync(() => {
  setCount(count + 1); // Pas de mise à jour groupée
 });
};
```

useEffect

La fonction hook **useEffect** peut être utilisée pour effectuer des effets de bord dans le composant fonctionnel React. L'effet de bord peut être, par exemple, une requête fetch. Le hook **useEffect** prend deux arguments, comme indiqué ici :

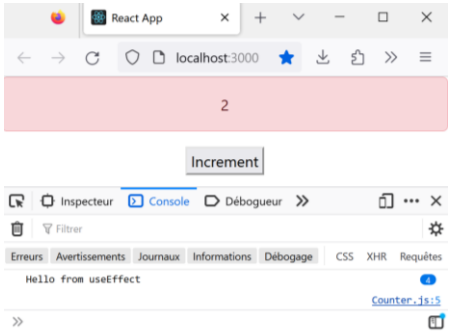
useEffect(callback, [dependencies])

La fonction de rappel (callback) contient la logique de l'effet de bord, et les dépendances sont un tableau facultatif de dépendances.

Le code suivant montre l'exemple précédent du compteur, mais nous avons ajouté le hook **useEffect**. Maintenant, lorsque le bouton est pressé, la valeur de l'état count augmente et le composant est réaffiché. Après chaque rendu, la fonction de rappel du **useEffect** est invoquée, et nous pouvons voir "*Hello from useEffect*" dans la console, comme illustré dans le code suivant :

```
import React, { useState, useEffect } from 'react';
function Counter() {
 const [count, setCount] = useState(0);
 useEffect(() => {
  console.log('Hello from useEffect');
 });
 return (
  <div>
   <p>{count}</p>
   <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}
export default Counter;
```

Dans la capture d'écran suivante, nous pouvons voir à quoi ressemble maintenant la console, et nous pouvons constater que la fonction de rappel du **useEffect** est invoquée après chaque rendu. La première ligne du log est affichée après le rendu initial, et les suivantes sont affichées après avoir cliqué sur le bouton deux fois et réaffiché le composant en raison des mises à jour de l'état :



Le hook **useEffect** a un deuxième argument facultatif que vous pouvez utiliser pour l'empêcher de s'exécuter à chaque rendu. Dans le code suivant, nous définissons que si la valeur de l'état count est modifiée (ce qui signifie que les valeurs précédente et actuelle diffèrent), la fonction de rappel du useEffect sera invoquée. Nous pouvons également définir plusieurs états dans le deuxième argument. Si l'une de ces valeurs d'état est modifiée, le hook useEffect sera invoqué :

```
useEffect(() => {
 console.log('Counter value is now ' + count);
}, [count]);
```

Si vous passez un tableau vide en tant que deuxième argument, la fonction de rappel du **useEffect** ne s'exécute qu'après le premier rendu, comme illustré dans le code suivant :

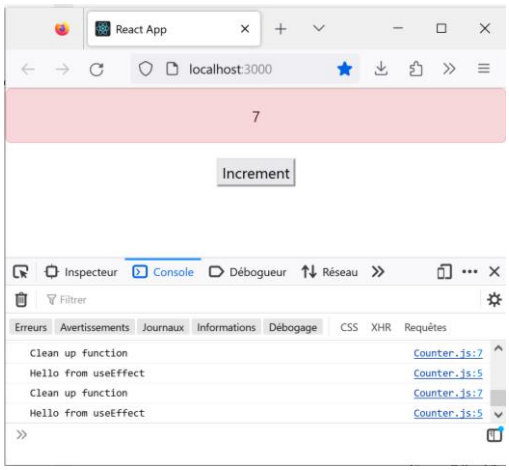
```
useEffect(() => {
 console.log('Hello from useEffect');
}, []);
```

Maintenant, vous pouvez voir que "*Hello from useEffect*" est imprimé une seule fois après le rendu initial, et si vous appuyez sur le bouton, le texte n'est pas imprimé.

La fonction useEffect peut également renvoyer une fonction qui sera exécutée avant chaque effet, comme illustré dans le code suivant. Avec ce mécanisme, vous pouvez nettoyer chaque effet du rendu précédent avant de l'exécuter la prochaine fois :

```
useEffect(() => {
 console.log('Hello from useEffect');
 return () => {
  console.log('Clean up function');
 };
}, [count]);
```

Maintenant, si vous exécutez une application de compteur avec ces modifications, vous pouvez voir ce qui se passe dans la console, comme illustré dans la capture d'écran :



useRef

Le hook useRef renvoie un objet de référence mutable et peut être utilisé, par exemple, pour accéder aux nœuds DOM. Vous pouvez voir un exemple ci-dessous :

```
const ref = useRef(initialValue);
```

L'objet de référence retourné a une propriété **current** qui est initialisée avec l'argument passé (**initialValue**). Dans l'exemple suivant, nous créons un objet de référence appelé inputRef et nous l'initialisons avec null. Ensuite, nous utilisons la propriété ref de l'élément JSX et nous lui passons notre objet de référence. Maintenant, il contient notre élément input, et nous pouvons utiliser la propriété current pour exécuter la fonction focus de l'élément input :

```
import React, { useRef } from 'react';
function TestRef() {
 const inputRef = useRef(null);
 return (
  <div>
   <input ref={inputRef} />
   <button onClick={() => inputRef.current.focus()}>Focus input</button>
  </div>
 );
}
export default TestRef;
```

Il existe d'autres fonctions hook utiles, et nous en aborderons certaines plus tard. Dans cette section, nous avons appris les bases sur les hooks de React et comment vous pouvez créer vos propres hooks personnalisés.

Hooks personnalisés

Vous pouvez également créer vos propres hooks dans React. Les noms des hooks doivent commencer par le mot "use" et ce sont des fonctions JavaScript. Les hooks personnalisés peuvent également appeler d'autres hooks. Avec des hooks personnalisés, vous pouvez réduire la complexité du code de vos composants.

Prenons l'exemple simple de la création d'un hook personnalisé. Nous allons créer un hook appelé "**useTitle**" qui peut être utilisé pour mettre à jour le titre d'un document. Nous le définirons dans son propre fichier appelé "**useTitle.js**". Tout d'abord, nous définissons une fonction qui prend un argument nommé "title".

Ensuite, nous utiliserons un hook useEffect pour mettre à jour le titre du document à chaque fois que l'argument "**title**" est modifié, comme suit :

```
import { useEffect } from 'react';
function useTitle(title) {
 useEffect(() => {
  document.title = title;
 }, [title]);
}
export default useTitle;
```

Maintenant, nous pouvons commencer à utiliser notre hook personnalisé. Utilisons-le dans notre exemple de compteur et affichons la valeur actuelle du compteur dans le titre du document. Tout d'abord, nous devons importer le hook "**useTitle**" dans notre composant **Counter**, comme ceci :

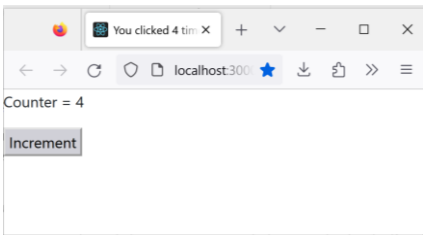
```
import useTitle from './useTitle';
function Counter() {
 return (
  <div>
  </div>
 );
};
export default Counter;
```

Ensuite, nous utiliserons le hook "**useTitle**" pour afficher la valeur de l'état "**count**" dans le titre du document. Nous pouvons appeler notre fonction de hook au niveau supérieur de la fonction du

composant **Counter**, et à chaque fois que le composant est rendu, la fonction du hook "**useTitle**" est appelée et nous pouvons voir la valeur actuelle du compteur dans le titre du document. Le code est illustré dans l'extrait suivant :

```
import React, { useState } from 'react';
import useTitle from './useTitle';
function App() {
 const [count, setCount] = useState(0);
 useTitle(`You clicked ${count} times`);
 return (
  <div>
   <p>Counter = {count}</p>
   <button onClick={() => setCount(count + 1)}>
    Increment
   </button>
  </div>
 );
};
export default App;
```

Maintenant, si vous cliquez sur le bouton, la valeur de l'état "count" est également affichée dans le titre du document à l'aide de notre hook personnalisé, comme illustré dans la ci dessous :



Vous avez maintenant des connaissances de base sur les hooks React et sur la façon de créer vos propres hooks personnalisés.