Pourquoi utiliser useEffect ?

Avant Hooks , les composants de fonction n'étaient utilisés que pour accepter des données sous forme de props et renvoyer du JSX à restituer. Cependant, comme nous l'avons appris dans la leçon précédente, le Hook d'état nous permet de gérer des données dynamiques, sous forme d'état de composant, au sein de nos composants de fonction.

Dans cette leçon, nous utiliserons **Effect Hook (hook d'éffet)** pour exécuter du code JavaScript après chaque rendu pour :

- récupérer des données à partir d'un service back-end.
- souscrire à un flux de données.
- gérer les minuteries et les intervalles.
- lire et apporter des modifications au DOM.

Les composants seront restitués plusieurs fois au cours de leur durée de vie. Ces moments clés constituent l'occasion idéale d'exécuter ces « effets secondaires » (effets de bord).

Il y a trois moments clés où l'Effet Hook peut être utilisé :

1. Lorsque le composant est ajouté ou *monté* pour la première fois au DOM et au rendu.
2. Lorsque l'état ou les accessoires changent, ce qui entraîne un nouveau rendu du composant.
3. Lorsque le composant est supprimé ou *démonté* du DOM.

Plus loin dans cette leçon, nous apprendrons comment affiner davantage le moment précis où le Hook Effet s'exécute.

Effets des composants de fonction

L'Effet Hook indique à notre composant de faire quelque chose à chaque fois qu'il est rendu (ou restitué). En combinaison avec les états, nous pouvons utiliser le Hook Effet pour créer des changements dynamiques intéressants dans nos pages Web !

Supposons que nous souhaitions permettre à un utilisateur de modifier le titre de l'onglet de la page Web à chaque fois qu'il tape quelque chose. Nous pouvons implémenter cela avec Effect Hook (`useEffect()`) comme ceci :

```
import React, { useState, useEffect } from 'react';

function PageTitle() {
  const [name, setName] = useState('');

  useEffect(() => {
    document.title = `Hi, ${name}`;
  });

  return (
    <div>
      <p>Use the input field below to rename this page!</p>
      <input onChange={({target}) => setName(target.value)} value={name}
type='text' />
    </div>
  );
}
```

Examinons l'exemple ci-dessus plus en détail. Tout d'abord, nous importons l'Effet Hook depuis la bibliothèque `'react'`:

```
import { useEffect } from 'react';
```

La fonction `useEffect()` n'a pas de valeur de retour car l'Effet Hook est utilisé pour appeler une autre fonction. Nous transmettons la fonction de rappel, ou *effect* , à exécuter après le rendu d'un composant comme argument de la `useEffect()`fonction. Dans notre exemple, l'effet suivant s'exécute après chaque `PageTitle`rendu du composant :

```
() => { document.title = Hi, ${name};}
```

Ici, nous attribuons `Hi, ${name}` comme valeur de `document.title`.

L' écouteur d'événements `onChange` déclenche le rendu du composant `PageTitle` à chaque fois que l'utilisateur saisit l'entrée. Par conséquent, cela déclenche `useEffect()`et modifie le titre du document.

Remarquez comment nous utilisons l' état actuel dans notre effet. Même si notre effet est appelé après le rendu du composant, nous avons toujours accès aux variables dans la portée de notre composant fonction ! Lorsque React restitue notre composant, il mettra à jour le DOM comme d'habitude, puis exécutera notre effet une fois le DOM mis à jour. Cela se produit pour chaque rendu, y compris le premier et le dernier.

Effets de nettoyage

Certains effets nécessitent **un nettoyage** . Par exemple, nous pourrions vouloir ajouter des écouteurs d'événements à certains éléments du DOM, au-delà du JSX dans notre composant. Lorsque nous ajoutons des écouteurs d'événements au DOM, il est important de supprimer ces écouteurs d'événements lorsque nous en avons terminé avec eux pour éviter les fuites de mémoire !

Considérons l'effet suivant :

```
useEffect(()=>{
  document.addEventListener('keydown', handleKeyPress);
  // Specify how to clean up after the effect:
  return () => {
    document.removeEventListener('keydown', handleKeyPress);
  };
})
```

Si notre effet ne renvoyait pas de *fonction de nettoyage* , un nouvel écouteur d'événement serait ajouté à l' objet `document` du DOM à chaque fois que notre composant effectuerait un nouveau rendu. Non seulement cela provoquerait des bugs, mais cela pourrait également entraîner une diminution des performances de notre application et peut-être même un crash !

Étant donné que les effets s'exécutent après chaque rendu et pas seulement une fois, React appelle notre fonction de nettoyage avant chaque nouveau rendu et avant le démontage pour nettoyer chaque appel d'effet.

Si notre effet renvoie une fonction, alors le Hook `useEffect()` la traite toujours comme une fonction de nettoyage. React appellera cette fonction de nettoyage avant que le

composant ne soit restitué ou démonté. Cette fonction de nettoyage étant facultative, il est de notre responsabilité de renvoyer une fonction de nettoyage de notre effet lorsque notre code d'effet pourrait créer des fuites de mémoire.

Contrôler quand les effets sont appelés

La fonction `useEffect()` appelle son premier argument (l'effet) après chaque rendu d'un composant. Nous avons appris à renvoyer une fonction de nettoyage afin de ne pas créer de problèmes de performances ni d'autres bugs, mais parfois nous souhaitons ignorer complètement l'appel de notre effet lors des nouveaux rendus.

Il est courant, lors de la définition de composants de fonction, d'exécuter un effet uniquement lorsque le composant est monté (rendu pour la première fois), mais pas lorsque le composant est restitué. Le Hook Effet nous rend cela très facile à faire ! Si nous voulons appeler notre effet uniquement après le premier rendu, nous passons un tableau vide à `useEffect()`comme deuxième argument. Ce deuxième argument est appelé **tableau de dépendances** .

Le tableau de dépendances est utilisé pour indiquer à la méthode `useEffect()` quand appeler notre effet et quand l'ignorer. Notre effet est toujours appelé après le premier rendu, mais n'est appelé à nouveau que si quelque chose dans notre tableau de dépendances a changé de valeur entre les rendus.

Nous continuerons à en apprendre davantage sur ce deuxième argument au cours des prochains exercices, mais pour l'instant, nous nous concentrerons sur l'utilisation d'un tableau de dépendances vide pour appeler un effet lors du premier montage d'un composant et si une fonction de nettoyage est renvoyée par notre effet, il l'appelle lorsque le composant se démonte.

```
useEffect(() => {
  alert("component rendered for the first time");
  return () => {
    alert("component is being removed from the DOM");
  };
}, []);
```

Sans passer un tableau vide comme deuxième argument de ce qui précède `useEffect()`, ces alertes seraient affichées avant et après chaque rendu de notre composant, ce qui n'est clairement pas le moment où ces messages sont censés être

affichés. Il suffit de passer `[]` à la fonction `useEffect()` pour configurer le moment où les fonctions d'effet et de nettoyage sont appelées !

Récupérer des données

Lors de la création de logiciels, nous commençons souvent par les comportements par défaut, puis nous les modifions pour améliorer les performances.

Nous avons appris que le comportement par défaut de Hook Effect est d'appeler la fonction d'effet après chaque rendu.

Ensuite, nous avons appris que nous pouvons passer un tableau vide comme deuxième argument de `useEffect`si nous voulons que notre effet ne soit appelé qu'après le premier rendu du composant.

Dans cet exercice, nous apprendrons à utiliser le tableau de dépendances pour configurer davantage le moment précis où nous voulons que notre effet soit appelé !

Lorsque notre effet est chargé de récupérer des données sur un serveur, nous accordons une attention particulière au moment où notre effet est appelé. Les allers-retours inutiles entre nos composants React et le serveur peuvent être coûteux en termes de :

- Traitement
- Performance
- Utilisation des données pour les utilisateurs mobiles
- Frais de service API

Lorsque les données que nos composants doivent restituer ne changent pas, nous pouvons transmettre un tableau de dépendances vide afin que les données soient récupérées après le premier rendu. Lorsque la réponse est reçue du serveur, nous pouvons utiliser un paramètre d'état du Hook State pour stocker les données de la réponse du serveur dans l'état de notre composant local pour les rendus futurs. Utiliser le Hook State et le Hook Effet ensemble de cette manière est un modèle puissant qui évite à nos composants de récupérer inutilement de nouvelles données après chaque rendu !

Un tableau de dépendances vide signale à Hook Effet que notre effet n'a jamais besoin d'être réexécuté, qu'il ne dépend de rien. Spécifier zéro dépendance signifie que le

résultat de l'exécution de cet effet ne changera pas et qu'il suffit d'appeler notre effet une seule fois.

Un tableau de dépendances qui n'est pas vide signale à Hook Effect qu'il peut ignorer l'appel de notre effet après un nouveau rendu, à moins que la valeur de l'une des variables de notre tableau de dépendances n'ait changé. Si la valeur d'une dépendance a changé, alors Hook Effet appellera à nouveau notre effet !

Voici un bel exemple tiré de la documentation officielle de React :

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if the value stored by count changes
```

Règles des crochets

Il y a deux règles principales à garder à l'esprit lors de l'utilisation des Hooks :

1. Appelez uniquement Hooks au niveau supérieur.
2. Appelez uniquement les Hooks à partir des fonctions React.

Au fur et à mesure que nous nous sommes entraînés avec le Hook State et le Hook Effet, nous avons suivi ces règles avec facilité, mais il est utile de garder ces deux règles à l'esprit lorsque vous étendez votre nouvelle compréhension des Hooks dans la nature et commencez à utiliser plus de Hooks dans vos applications React.

Lorsque React crée le DOM virtuel , la bibliothèque appelle encore et encore les fonctions qui définissent nos composants au fur et à mesure que l'utilisateur interagit avec l'interface utilisateur. React garde une trace des données et des fonctions que nous gérons avec des Hooks en fonction de leur ordre dans la définition du composant de fonction. Pour cette raison, nous appelons toujours nos Hooks au plus haut niveau ; nous n'appelons jamais de hooks à l'intérieur de boucles, de conditions ou de fonctions imbriquées.

Au lieu de confondre React avec un code comme celui-ci :

```
if (userName !== '') {
  useEffect(() => {
    localStorage.setItem('savedUserName', userName);
```

```
  });
}
```

Nous pouvons atteindre le même objectif tout en appelant systématiquement notre Hook à chaque fois :

```
useEffect(() => {
  if (userName !== '') {
    localStorage.setItem('savedUserName', userName);
  }
});
```

Deuxièmement, les Hooks ne peuvent être utilisés que dans les fonctions React. Nous avons travaillé avec `useState()` et `useEffect()` dans des composants *fonctionnels* , et c'est l'utilisation la plus courante. Le seul autre endroit où les Hooks peuvent être utilisés est celui des Hooks personnalisés. Les Hooks personnalisés sont incroyablement utiles pour organiser et réutiliser la logique avec état entre les composants de fonction.

Crochets séparés pour des effets séparés

Lorsque plusieurs valeurs sont étroitement liées et changent en même temps, il peut être judicieux de regrouper ces valeurs dans une collection comme un objet ou un tableau. Le regroupement des données peut également ajouter de la complexité au code responsable de la gestion de ces données. Par conséquent, c'est une bonne idée de séparer les préoccupations en gérant différentes données avec différents Hooks .

Comparez la complexité ici, où les données sont regroupées en un seul objet :

```
// Handle both position and menuItems with one useEffect hook.
const [data, setData] = useState({ position: { x: 0, y: 0 } });
useEffect(() => {
  get('/menu').then((response) => {
    setData((prev) => ({ ...prev, menuItems: response.data }));
  });
  const handleMove = (event) =>
    setData((prev) => ({
      ...prev,
      position: { x: event.clientX, y: event.clientY }
    }));
  window.addEventListener('mousemove', handleMove);
```

```
  return () => window.removeEventListener('mousemove', handleMove);
}, []);
```

À la simplicité ici, où nous avons séparé les préoccupations :

```
// Handle menuItems with one useEffect hook.
const [menuItems, setMenuItems] = useState(null);
useEffect(() => {
  get('/menu').then((response) => setMenuItems(response.data));
}, []);

// Handle position with a separate useEffect hook.
const [position, setPosition] = useState({ x: 0, y: 0 });
useEffect(() => {
  const handleMove = (event) =>
    setPosition({ x: event.clientX, y: event.clientY });
  window.addEventListener('mousemove', handleMove);
  return () => window.removeEventListener('mousemove', handleMove);
}, []);
```

Il n'est pas toujours évident de regrouper les données ou de les séparer, mais avec de la pratique, nous parvenons à mieux organiser notre code afin qu'il soit plus facile à comprendre, à ajouter, à réutiliser et à tester !

Resumé

Dans cette leçon, nous avons appris à écrire des effets qui gèrent les minuteries, manipulent le DOM et récupèrent les données d'un serveur. Avec l'Effet Hook, nous pouvons facilement effectuer ce type d'actions dans les composants fonctionnels !

Passons en revue les principaux concepts de cette leçon :

- Nous pouvons importer la fonction `useEffect()` depuis la bibliothèque `'react'` et l'appeler dans nos composants de fonction.
- *L'effet* fait référence à une fonction que nous passons comme premier argument de la fonction `useEffect()`. Par défaut, Hook Effect appelle cet effet après chaque rendu.
- La *fonction de nettoyage* est éventuellement renvoyée par l'effet. Si l'effet fait quelque chose qui doit être nettoyé pour éviter les fuites de mémoire, alors l'effet renvoie une fonction de nettoyage, puis Hook Effect appellera cette fonction de

nettoyage avant d'appeler à nouveau l'effet ainsi que lorsque le composant est démonté.
- Le *tableau de dépendances* est le deuxième argument facultatif `useEffect()`avec lequel la fonction peut être appelée afin d'éviter d'appeler l'effet à plusieurs reprises lorsque cela n'est pas nécessaire. Ce tableau doit être composé de toutes les variables dont dépend l'effet.

Le Hook Effet consiste à planifier le moment où le code de notre effet est exécuté. Nous pouvons utiliser le tableau de dépendances pour configurer le moment où notre effet est appelé des manières suivantes :

Tableau de dépendances	**Effet appelé après le premier rendu &…**
indéfini	à chaque nouveau rendu
Tableau vide	pas de nouveau rendu
Tableau non vide	lorsqu'une valeur du tableau de dépendances change

Les hooks nous donnent la flexibilité d'organiser notre code de différentes manières, en regroupant les données associées ainsi qu'en séparant les préoccupations pour garder le code simple, sans erreur, réutilisable et testable !