Mettre à jour l'état du composant de fonction

Commençons par le **State Hook** , le Hook le plus couramment utilisé pour créer des composants React. Le State est une exportation nommée de la bibliothèque React, nous importons donc avec une desctructuration d'objet comme suit :

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

Lorsque nous appelons la fonction `useState()`, elle renvoie un tableau avec deux valeurs :

- L' *état actuel* : La valeur actuelle de cet état.
- Le *setter d'état* : Une fonction que l'on peut utiliser pour mettre à jour la valeur de cet état.

Nous pouvons utiliser ces deux valeurs pour suivre l'état actuel d'une valeur de données ou d'une propriété et la modifier lorsque nous en avons besoin. Pour extraire les deux valeurs du tableau, nous pouvons les affecter à des variables locales en utilisant la déstructuration du tableau. Par exemple:

```
const [currentState, setCurrentState] = useState();
```

Jetons un coup d'œil à un autre exemple de composant de fonction qui utilise le State Hook :

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

function Toggle() {
  const [toggle, setToggle] = useState();

  return (
    <div>
      <p>The toggle is {toggle}</p>
      <button onClick={() => setToggle("On")}>On</button>
      <button onClick={() => setToggle("Off")}>Off</button>
    </div>
  );
}
```

Remarquez comment la fonction de définition d'état, `setToggle()`, est appelée par nos *écouteurs d'événements* `onClick`. Pour mettre à jour la valeur de `toggle` et restituer ce composant avec la nouvelle valeur, tout ce que nous avons à faire est d'appeler la fonction `setToggle()` avec la valeur d'état suivante comme argument.

Avec le State Hook, mettre à jour l'état est aussi simple que d'appeler une fonction de définition d'état. L'appel du setter d'état signale à React que le composant doit être restitué, de sorte que toute la fonction définissant le composant est à nouveau appelée. La magie de `useState()` c'est que cela permet à React de garder une trace de la valeur actuelle de l'état d'un rendu à l'autre !

Initialiser l'état

De la même manière que vous avez utilisé le State Hook pour gérer une variable avec des valeurs de chaîne, nous pouvons utiliser le State Hook pour gérer la valeur de n'importe quel type de données primitif et même des collections de données telles que des tableaux et des objets !

Jetez un œil au composant de fonction suivant. Quel type de données contient cette variable d'état ?

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

function ToggleLoading() {
  const [isLoading, setIsLoading] = useState();

  return (
    <div>
      <p>The data is {isLoading ? 'Loading' : 'Not Loading'}</p>
```

```
        <button onClick={() => setIsLoading(true)}>
          Turn Loading On
        </button>
        <button onClick={() => setIsLoading(false)}>
          Turn Loading Off
        </button>
      </div>
    );
  }
```

Le composant fonction `ToggleLoading()` ci-dessus utilise le plus simple de tous les types de données, un booléen. Les booléens sont fréquemment utilisés dans les applications React pour indiquer si les données ont été chargées ou non. Dans l'exemple ci-dessus, nous voyons que les valeurs `true` et `false` sont transmises au paramètre d'état, `setIsLoading()`.

Ce code fonctionne très bien tel quel, mais que se passe-t-il si nous voulons que notre composant démarre avec set `isLoading` sur `true`?

Pour initialiser notre état avec n'importe quelle valeur souhaitée, nous transmettons simplement la valeur initiale comme argument à l'appel de fonction `useState()`.

```
const [isLoading, setIsLoading] = useState(true);
```

Ce code affecte notre composant de trois manières :

1. Lors du premier rendu, l' *argument d'état initial* est utilisé.
2. Lorsque le setter d'état est appelé, React ignore l'argument d'état initial et utilise la nouvelle valeur.
3. Lorsque le composant effectue un nouveau rendu pour une autre raison, React continue d'utiliser la même valeur du rendu précédent.

Si nous ne transmettons pas de valeur initiale lors de l'appel `useState()`, la valeur actuelle de l'état lors du premier rendu sera `undefined`. Souvent, cela convient parfaitement aux ordinateurs qui exécutent le code, mais cela peut ne pas être clair pour les humains qui lisent notre code. Nous préférons donc initialiser explicitement notre état. Si nous n'avons pas la valeur nécessaire lors du premier rendu, nous pouvons transmettre explicitement `null` au lieu de laisser passivement la valeur comme `undefined`

Utiliser State Setter en dehors de JSX

Voyons un exemple de gestion de la valeur changeante d'une chaîne lorsqu'un utilisateur tape dans un champ de saisie de texte :

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

export default function EmailTextInput() {
  const [email, setEmail] = useState('');
  const handleChange = (event) => {
    const updatedEmail = event.target.value;
    setEmail(updatedEmail);
  }

  return (
    <input value={email} onChange={handleChange} />
  );
}
```

Voici un aperçu du fonctionnement du code ci-dessus :

- Nous utilisons le destructuring du tableau pour créer notre variable d'état locale `email` et notre fonction de définition locale `setEmail()`.
- La variable locale `email` se voit attribuer la valeur d'état actuelle à l'index `0` du tableau renvoyé par `useState()`.
- La variable locale `setEmail()` se voit attribuer une référence à la fonction de définition d'état à l'index `1` du tableau renvoyé par `useState()`.
- C'est une convention de nommer la variable setter en utilisant la variable d'état actuelle (dans cet exemple, `email`) avec « set » préfixé.

  La balise d'entrée JSX possède un écouteur d'événement appelé `onChange`. Cet écouteur d'événements appelle un *gestionnaire d'événements* chaque fois que l'utilisateur tape quelque chose dans cet élément. Dans l'exemple ci-dessus, notre gestionnaire d'événements est défini à l'intérieur de la définition de notre composant fonction, mais en dehors de notre JSX. Plus tôt dans cette leçon, nous avons écrit nos gestionnaires d'événements directement dans notre JSX. Ces gestionnaires d'événements en ligne fonctionnent parfaitement, mais lorsque nous voulons faire quelque chose de plus intéressant que simplement appeler le setter d'état avec une valeur statique, c'est une bonne pratique de séparer cette

logique de notre JSX. Cette séparation des préoccupations rend notre code plus facile à lire, tester et modifier.

I est courant dans le code React de simplifier cela :

```
const handleChange = (event) => {
  const newEmail = event.target.value;
  setEmail(newEmail);
}
```

Par ceci :

```
const handleChange = (event) =>
setEmail(event.target.value);
```

ou, en utilisant la déstructuration d'objets, ceci :

```
const handleChange = ({target}) =>
setEmail(target.value);
```

Les trois extraits de code ci-dessus se comportent de la même manière, il n'y a donc pas vraiment de bien ou de mal entre ces différents extraits de code. Nous utiliserons désormais la dernière version, la plus concise.

Définir à partir de l'état précédent

Dans l'exercice précédent (phone_number_app), nous avons appris à mettre à jour l' état en lui passant une nouvelle valeur comme celle-ci :

```
setState(newStateValue);
```

Cependant, les mises à jour de l'état de React sont *asynchrones* . Cela signifie qu'il existe certains scénarios dans lesquels des parties de votre code seront exécutées avant la fin de la mise à jour de l'état.

C'est une bonne et une mauvaise chose ! Le regroupement des mises à jour d'état peut améliorer les performances de votre application, mais cela peut entraîner des

valeurs d'état obsolètes. Par conséquent, il est recommandé de mettre à jour un état avec une fonction de rappel, évitant ainsi les valeurs obsolètes accidentelles.

Jetons un coup d'œil au code suivant pour voir comment cela fonctionne :

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

export default function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => setCount(prevCount => prevCount + 1);

  return (
    <div>
      <p>Wow, you've clicked that button: {count} times</p>
      <button onClick={increment}>Click here!</button>
    </div>
  );
}
```

Lorsque le bouton est enfoncé, le gestionnaire d'événements `increment()` est appelé. Dans cette fonction, nous utilisons notre setter d'état `setCount()` avec une fonction de rappel.

Étant donné que la valeur suivante de `count` dépend de la valeur précédente de `count`, nous passons une fonction de rappel (callBack) comme argument à `setCount()` au lieu d'une valeur (comme nous l'avons fait dans les exercices précédents).

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

Lorsque notre setter d'état appelle la fonction de rappel, cette *fonction de rappel du setter d'état* prend notre précédent `count` comme argument. La valeur renvoyée par ce callBack de définition d'état est utilisée comme valeur suivante de `count`(dans ce cas, `prevCount + 1`).

Nous pouvons aussi simplement appeler `setCount(count +1)`et cela fonctionnerait de la même manière dans cet exemple, mais pour des raisons qui sortent du cadre de cette leçon, il est plus sûr d'utiliser la méthode de rappel.

Tableaux dans l'état

Les tableaux JavaScript constituent le meilleur modèle de données pour gérer et afficher des listes JSX . Jetons un coup d'œil au code d'un site Web pour une pizzeria.

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

//Static array of pizza options offered.
const options = ['Bell Pepper', 'Sausage', 'Pepperoni', 'Pineapple'];

export default function PersonalPizza() {
  const [selected, setSelected] = useState([]);

  const toggleTopping = ({target}) => {
    const clickedTopping = target.value;
    setSelected((prev) => {
     // check if clicked topping is already selected
      if (prev.includes(clickedTopping)) {
        // filter the clicked topping out of state
        return prev.filter(t => t !== clickedTopping);
      } else {
        // add the clicked topping to our state
        return [clickedTopping, ...prev];
      }
    });
  };

  return (
    <div>
      {options.map(option => (
        <button value={option} onClick={toggleTopping} key={option}>
          {selected.includes(option) ? 'Remove ' : 'Add '}
          {option}
        </button>
      ))}
      <p>Order a {selected.join(', ')} pizza</p>
    </div>
  );
}
```

Dans l'exemple ci-dessus, nous utilisons deux tableaux :

- Le tableau `options` contient les noms de toutes les garnitures de pizza disponibles.
- Le tableau `selected` représente les garnitures sélectionnées pour notre pizza personnelle.

Le tableau `options` contient *des données statiques* , ce qui signifie qu'elles ne changent pas. Il est recommandé de définir des modèles de données statiques en dehors des composants de fonction, car ils n'ont pas besoin d'être recréés à chaque nouveau rendu de notre composant. Dans notre JSX, nous utilisons la méthode

`.map()`J de avaScript pour afficher un bouton pour chacune des garnitures de notre tableau `options`.

Le tableau `selected` contient *des données dynamiques* , ce qui signifie qu'elles changent, généralement en fonction des actions d'un utilisateur. Nous initialisons `selected` comme un tableau vide. Lorsqu'un bouton est cliqué, le gestionnaire d'événements `toggleTopping()` est appelé. Remarquez comment ce gestionnaire d'événements utilise les informations de l'objet événement pour déterminer sur quelle garniture a été cliqué.

Lors de la mise à jour d'un tableau dans un state , nous n'ajoutons pas simplement de nouvelles données au tableau précédent. Nous remplaçons le tableau précédent par un tout nouveau tableau. Cela signifie que toutes les informations que nous souhaitons enregistrer du tableau précédent doivent être explicitement copiées vers notre nouveau tableau. C'est ce que cette syntaxe de propagation (spread syntaxe) fait pour nous : `...prev`

Remarquez comment nous utilisons les méthodes `.includes()`, `.filter()` et `.map()`de nos tableaux. Si ces méthodes sont nouvelles pour vous ou si vous souhaitez simplement un rappel, prenez une minute pour examiner ces méthodes de tableau. Nous n'avons pas besoin d'être des gourous de JavaScript à part entière pour créer des applications React, mais sachez qu'investir du temps pour renforcer nos compétences JavaScript nous aidera toujours à faire plus plus rapidement (et à avoir beaucoup plus de plaisir à le faire) en tant que développeurs React.

Objets dans l'état

Nous pouvons également utiliser l'état avec des objets. Lorsque nous travaillons avec un ensemble de variables liées, il peut être très utile de les regrouper dans un objet. Regardons un exemple de ceci en action.

```
export default function Login() {
  const [formState, setFormState] = useState({});
  const handleChange = ({ target }) => {
    const { name, value } = target;
    setFormState((prev) => ({
      ...prev,
      [name]: value
    }));
  };

  return (
```

```
      <form>
        <input
          value={formState.firstName}
          onChange={handleChange}
          name="firstName"
          type="text"
        />
        <input
          value={formState.password}
          onChange={handleChange}
          type="password"
          name="password"
        />
      </form>
    );
  }
```

Quelques points à remarquer :

- Nous utilisons une fonction de rappel (callBack) de définition d'état pour mettre à jour un état en fonction de la valeur précédente.
- La syntaxe de propagation (spread operator) est la même pour les objets que pour les tableaux : `{ ...oldObject, newKey: newValue }`.
- Nous réutilisons notre gestionnaire d'événements sur plusieurs entrées en utilisant l' attribut `name` de la balise d'entrée pour identifier de quelle entrée provient l'événement de changement.

Encore une fois, lors de la mise à jour `setFormState()` de l'état à l'intérieur d'un composant fonction, on ne modifie pas le même objet (important). Nous devons copier les valeurs de l'objet précédent lors de la définition de la valeur suivante d'un état. Heureusement, la syntaxe spread rend cela très facile à faire !

Chaque fois qu'une des valeurs d'entrée est mise à jour, la fonction `handleChange()` sera appelée. Dans ce gestionnaire d'événements, nous utilisons la déstructuration d'objet pour décompresser la propriété `target` de notre objet `event`, puis nous utilisons à nouveau la déstructuration d'objet pour décompresser les propriétés `name` et `value`de l' objet `target`.

Dans notre fonction de rappel de définition d'état, nous enveloppons nos accolades entre parenthèses comme ceci :

```
setFormState((prev) => ({ ...prev }))
```

Cela indique à JavaScript que nos accolades font référence à un nouvel objet à renvoyer. Nous utilisons `...`, l'opérateur spread, pour remplir les champs correspondants de notre état précédent. Enfin, nous écrasons la clé appropriée par sa valeur mise à jour.

Avez-vous remarqué les crochets autour du `name`? Ce nom de propriété calculé nous permet d'utiliser la valeur de chaîne stockée par la variable `name` comme clé de propriété.

Hooks séparés pour des États séparés

Bien qu'il puisse parfois être utile de stocker des données associées dans une collection de données, comme un tableau ou un objet, il peut également être utile de créer différentes variables d'état pour les données qui changent séparément. La gestion des données dynamiques est beaucoup plus facile lorsque nous gardons nos modèles de données aussi simples que possible.

Par exemple, si nous avions un seul objet détenant un état pour une matière que vous étudiez à l'école, il pourrait ressembler à ceci :

```
function Subject() { const [state, setState] = useState({ currentGrade: 'B', classmates: ['Hasan', 'Sam', 'Emma'], classDetails: {topic: 'Math', teacher: 'Ms. Barry', room: 201}, exams: [{unit: 1, score: 91}, {unit: 2, score: 88}] })}
```

Cela fonctionnerait, mais pensez à quel point cela pourrait être compliqué de copier toutes les autres valeurs lorsque nous devons mettre à jour quelque chose dans ce gros objet d'état. Par exemple, pour mettre à jour la note d'un examen, nous aurions besoin d'un gestionnaire d'événements qui fasse quelque chose comme ceci :

```
setState((prev) => ({
 ...prev,
  exams: prev.exams.map((exam) => {
    if( exam.unit === updatedExam.unit ){
      return {
        ...exam,
        score: updatedExam.score
      };
    } else {
      return exam;
    }
```

```
  }),
}));
```

Un code complexe comme celui-ci est susceptible de provoquer des bugs. Il est préférable de créer plusieurs variables d'état en fonction des valeurs qui ont tendance à changer ensemble.

On peut réécrire l'exemple précédent comme suit :

```
function Subject() {
  const [currentGrade, setGrade] = useState('B');
  const [classmates, setClassmates] = useState(['Hasan', 'Sam', 'Emma']);
  const [classDetails, setClassDetails] = useState({topic: 'Math', teacher: 'Ms.
Barry', room: 201});
  const [exams, setExams] = useState([{unit: 1, score: 91}, {unit: 2, score: 88}]);
  // ...
}
```

La gestion des données dynamiques avec des variables d'état distinctes présente de nombreux avantages, comme rendre notre code plus simple à écrire, lire, tester et réutiliser entre les composants.

Souvent, nous nous retrouvons à regrouper et à organiser les données dans des collections pour les transmettre entre les composants, puis à séparer ces données au sein de composants où différentes parties des données changent séparément. Ce qui est merveilleux dans le fait de travailler avec Hooks, c'est que nous avons la liberté d'organiser nos données de la manière qui nous semble la plus logique !

Exemple

1 .

Jetez un œil au composant fonctionnel `Musical`. Il a un grand objet d'état.

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

function Musical() {
   const [state, setState] = useState({
    title: "Best Musical Ever",
    actors: ["George Wilson", "Tim Hughes", "Larry Clements"],
    locations: {
      Chicago: {
        dates: ["1/1", "2/2"],
```

```
        address: "chicago theater"},
      SanFrancisco: {
        dates: ["5/2"],
        address: "sf theater"
      }
    }
  })
 }
```

Refactorisons-le pour le rendre plus lisible et réutilisable. Nous allons travailler sur sa division en trois variables distinctes : `title`, `actors`, et `locations`.

```
function MusicalRefactored() {
  const [title, setTitle] = useState("Best Musical Ever");
    const [actors, setActors] = useState(["George Wilson", "Tim Hughes", "Larry
Clements"]);
  const [locations, setLocations] = useState({
    Chicago: {
      dates: ["1/1", "2/2"],
      address: "chicago theater"
    },
    SanFrancisco: {
      dates: ["5/2"],
      address: "sf theater"
    }
  });
}
```

Resumé

Nous pouvons désormais créer des composants de fonctions avec état à l'aide du Hook `useState`!

Passons en revue ce que nous avons appris et mis en pratique dans cette leçon :

Avec React, nous transmettons des modèles de données statiques et dynamiques à JSX pour afficher une vue à l'écran.

Les Hooks sont utilisés pour « se connecter » à l'état du composant interne afin de gérer les données dynamiques dans les composants de fonction.

Nous utilisons le State Hook en utilisant le code ci-dessous. Le `currentState` fait référence à la valeur actuelle de l'état et `initialState`initialise la valeur de l'état pour le premier rendu du composant.

```
const [currentState, stateSetter] = useState( initialState );
```

- Les setters d'état peuvent être appelés dans les gestionnaires d'événements.
- Nous pouvons définir des gestionnaires d'événements simples en ligne dans notre JSX et des gestionnaires d'événements complexes en dehors de notre JSX.
- Nous utilisons une fonction de rappel de définition d'état lorsque notre prochaine valeur dépend de notre valeur précédente.
- Nous utilisons des tableaux et des objets pour organiser et gérer les données associées qui ont tendance à changer ensemble.
- Utilisez la syntaxe répartie sur les collections de données dynamiques pour copier l'état précédent dans l'état suivant comme ceci : `setArrayState((prev) => [ ...prev ])` et `setObjectState((prev) => ({ ...prev }))`.

Il est préférable d'avoir plusieurs états plus simples au lieu d'avoir un seul objet d'état complexe.