# **Why Use Hooks?**

En tant que développeurs React, nous aimons décomposer les problèmes complexes en éléments simples.

Les classes, cependant, ne sont pas simples. Ils:

* sont difficiles à réutiliser entre les composants.
* sont difficiles et chronophages à tester.
* ont dérouté de nombreux développeurs et causé de nombreux bugs.

L'équipe principale de React a entendu tous ces commentaires de développeurs comme nous, et ils ont conçu une solution incroyable pour nous ! React 16.8+ prend en charge les crochets. Avec Hooks, nous pouvons utiliser des composants de fonction simples pour faire beaucoup de choses fantaisistes que nous ne pouvions faire qu'avec des composants de classe dans le passé.

Les React Hooks, en clair, sont des fonctions qui nous permettent de gérer l'état interne des composants et de gérer les effets secondaires post-rendu directement à partir de nos composants fonctionnels. Les crochets ne fonctionnent pas à l'intérieur des classes - ils nous permettent d'utiliser des fonctionnalités sophistiquées de React sans classes. Gardez à l'esprit que les composants de fonction et React Hooks ne remplacent pas les composants de classe. Ils sont complètement facultatifs ; juste un nouvel outil dont nous pouvons profiter.

Remarque : Si vous êtes familier avec les méthodes de cycle de vie des composants de classe, vous pouvez dire que les crochets nous permettent de nous "accrocher" aux fonctionnalités de state et de cycle de vie directement à partir de nos composants de fonction.

**React** propose un certain nombre de crochets intégrés. Certains d'entre eux incluent **useState()**, **useEffect()**, **useContext()**, **useReducer()** et **useRef()**. Voir la liste complète dans les docs. Dans cette leçon, nous allons apprendre différentes façons de gérer l'état dans un composant de fonction.

**Instructions:**

1.

Passez en revue le composant de classe défini dans le fichier AppClass.js.

Remarquez comment la logique de rendu a été déléguée pour séparer les composants de la fonction de présentation. Ce composant AppClass utilise un constructeur, ses propres méthodes de classe, ainsi que les méthodes this.setState() et this.render() de la classe Component de React.

Faites des prédictions sur le comportement de ce code, puis exécutez-le pour vérifier vos prédictions !

2.

Bouclez votre ceinture de sécurité. Nous sommes sur le point de nous aventurer dans un nouveau territoire.

Ouvrez le fichier AppFunction.js. Nous apprendrons comment ce code fonctionne dans les prochains exercices. Ne vous souciez pas encore des détails de ce qui se passe ici, mais prenez quelques instants pour lire la définition de ce composant de fonction et développez quelques théories sur ce que ce code peut faire.

Ouvrez le fichier index.js et changez d'où ce module importe le composant App, afin que nous puissions restituer l'AppFunction au lieu du composant AppClass. Appuyez sur Exécuter pour voir comment le code se comporte !

import React from "react";

import ReactDOM from "react-dom";

//import App from "./Container/AppClass";

import App from "./Container/AppFunction";

ReactDOM.render(

  <App />,

  document.getElementById("app")

);

import React, { useState } from "react";

import NewTask from "../Presentational/NewTask";

import TasksList from "../Presentational/TasksList";

export default function AppFunction() {

  const [newTask, setNewTask] = useState({});

  const handleChange = ({ target }) => {

    const { name, value } = target;

    setNewTask((prev) => ({ ...prev, id: Date.now(), [name]: value }));

  };

  const [allTasks, setAllTasks] = useState([]);

  const handleSubmit = (event) => {

    event.preventDefault();

    if (!newTask.title) return;

    setAllTasks((prev) => [newTask, ...prev]);

    setNewTask({});

  };

  const handleDelete = (taskIdToRemove) => {

    setAllTasks((prev) => prev.filter((task) => task.id !== taskIdToRemove));

  };

  return (

    <main>

      <h1>Tasks</h1>

      <NewTask

        newTask={newTask}

        handleChange={handleChange}

        handleSubmit={handleSubmit}

      />

      <TasksList allTasks={allTasks} handleDelete={handleDelete} />

    </main>

  );

}

import React from "react";

export default function NewTask({ newTask, handleChange, handleSubmit }) {

  return (

    <form onSubmit={handleSubmit}>

      <input

        name="title"

        placeholder="New task"

        value={newTask.title || ""}

        onChange={handleChange}

      />

      {!newTask.title ? null : (

        <>

          <textarea

            name="description"

            placeholder="Details..."

            value={newTask.description || ""}

            onChange={handleChange}

          />

          <button type="submit">Add Task</button>

        </>

      )}

    </form>

  );

}

import React from "react";

export default function TasksList({ allTasks, handleDelete }) {

  return (

    <ul>

      {allTasks.map(({ title, description, id }) => (

        <li key={id}>

          <div>

            <h2>{title}</h2>

            <button onClick={() => handleDelete(id)}>X</button>

          </div>

          {!description ? null : <p>{description}</p>}

        </li>

      ))}

    </ul>

  );

}

# **Update Function Component State**

Commençons par le State Hook, le Hook le plus couramment utilisé pour construire des composants React. Le State Hook est une exportation nommée de la bibliothèque React, nous l'importons donc comme ceci :

import React, { useState } from 'react';

useState() est une fonction JavaScript définie dans la bibliothèque React. Lorsque nous appelons cette fonction, elle renvoie un tableau avec deux valeurs :

* current state- la valeur actuelle de cet état
* state setter- une fonction que nous pouvons utiliser pour mettre à jour la valeur de cet état

Parce que React renvoie ces deux valeurs dans un tableau, nous pouvons les affecter à des variables locales, en les nommant comme bon nous semble. Par example: const [toggle, setToggle] = useState();

Examinons un exemple de composant de fonction utilisant 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 de state, setToggle(), est appelée par nos event Listeners 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.

Inutile de vous soucier de lier des fonctions à des instances de classe, de travailler avec des constructeurs ou de gérer le mot-clé this. Avec le crochet de state, la mise à jour de state est aussi simple que d'appeler state setter function.

L'appel du setter de state signale à React que le composant doit être rendu à nouveau, de sorte que toute la fonction définissant le composant est appelée à nouveau. La magie de useState() est qu'elle permet à React de garder une trace de la valeur actuelle de state d'un rendu à l'autre !

# **Initialize State**

Excellent travail de construction de votre premier composant de fonction avec state dans le dernier exercice. Tout comme vous avez utilisé le crochet de state pour gérer une variable avec des valeurs de chaîne, nous pouvons utiliser le crochet de state pour gérer la valeur de n'importe quel type de données primitif et même des collections de données comme 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 de 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 setter de state, setIsLoading(). Ce code fonctionne très bien tel quel, mais que se passe-t-il si nous voulons que notre composant démarre avec isLoading défini sur true ?

Pour initialiser notre state avec la valeur que nous voulons, nous passons simplement la valeur initiale comme argument à l'appel de la fonction useState().

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

Ce code affecte notre composant de trois manières :

* Lors du premier rendu, l'argument de state initial est utilisé.
* Lorsque le setter de state est appelé, React ignore l'argument de state initial et utilise la nouvelle valeur.
* Lorsque le composant est restitué 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 à useState(), la valeur actuelle de state lors du premier rendu sera undefined. Souvent, cela convient parfaitement aux machines, mais cela peut ne pas être clair pour les humains qui lisent notre code. Nous préférons donc initialiser explicitement notre state. Si nous n'avons pas la valeur nécessaire lors du premier rendu, nous pouvons passer explicitement null au lieu de simplement laisser passivement la valeur comme undefined.

# **Use State Setter Outside of JSX**

Voyons comment gérer 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} />  
  );  
}

Décomposons comment ce code fonctionne !

* Les crochets sur le côté gauche de la déstructuration du tableau de signaux de l'opérateur d'affectation.
* La variable locale nommée email reçoit la valeur de l'état actuel à l'index 0 du tableau renvoyé par useState().
* La variable locale nommée setEmail() est affectée d'une référence à la fonction de définition d'état à l'index 1 à partir du tableau renvoyé par useState().
* Il est conventionnel de nommer cette variable en utilisant la variable de state actuelle (email) avec "set" en préfixe.

La balise d'entrée JSX a un écouteur d'événement appelé onChange. Cet Event Listener appelle un Event Handler chaque fois que l'utilisateur tape quelque chose dans cet élément. Dans l'exemple ci-dessus, notre Event Listener est défini à l'intérieur de la définition de notre composant de fonction, mais en dehors de notre JSX. Plus tôt dans cette leçon, nous avons écrit nos Event Listeners directement dans notre JSX. Ces Event Listeners en ligne fonctionnent parfaitement bien, mais lorsque nous voulons faire quelque chose de plus intéressant que d'appeler simplement le setter de state avec une valeur statique, c'est une bonne idée de séparer cette logique de tout ce qui se passe dans notre JSX. Cette séparation des préoccupations rend notre code plus facile à lire, à tester et à modifier.

C'est si courant dans le code React, que nous pouvons confortablement simplifier ceci :

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

pour ça:

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

Ou même, utilisez la déstructuration d'objet pour simplement écrire ceci :

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

Ces trois extraits de code se comportent de la même manière, il n'y a donc pas vraiment de bien et de mal entre ces différentes façons de procéder. Nous utiliserons la dernière version, la plus concise, à l'avenir.

**Instructions**

1.

Déclarez et affectez des valeurs pour notre état actuel et notre setter d'état avec useState(). Utilisez phone comme nom de notre variable d'état actuelle.

2.

Ajoutez l'attribut value et l'écouteur d'événement onChange à notre balise d'entrée JSX. Donnez à ces attributs les valeurs de notre variable d'état actuelle et de notre fonction de gestionnaire d'événements.

Nous voulons donner à notre balise d'entrée les attributs suivants :

<input value={currentState} onChange={eventHandler} id='phone-input' />

Notre state actuel est phone.

Notre eventHandler est handleChange (pas notre state setter, setPhone)

3.

Utilisez notre paramètre de state pour mettre à jour le state uniquement lorsque la valeur de l'événement de modification de l'utilisateur réussit notre test d'expression régulière pour les chaînes de numéros de téléphone valides.

Curieux de connaître le /^\d{1,10}$/ dans notre code ? Découvrez cette excellente leçon pour en savoir plus sur les expressions régulières !

import React, { useState } from 'react';

// regex to match numbers between 1 and 10 digits long

const validPhoneNumber = /^\d{1,10}$/;

export default function PhoneNumber() {

  // declare current state and state setter

  const [phone, setPhone] = useState('');

const handleChange = ({ target })=> {

    const newPhone = target.value;

    const isValid = validPhoneNumber.test(newPhone);

    if (isValid) {

        // update state

        setPhone(newPhone);

    }

    // just ignore the event, when new value is invalid

  };

  return (

    <div className='phone'>

      <label for='phone-input'>Phone: </label>

      <input value={phone} onChange={handleChange} id='phone-input'/>

    </div>

  );

}

# **Set From Previous State**

Souvent, la prochaine valeur de notre état est calculée en utilisant l'état actuel. Dans ce cas, il est recommandé de mettre à jour l'état avec une fonction de rappel. Si nous ne le faisons pas, nous risquons de capturer des valeurs d'état obsolètes ou "périmées".

Jetons un œil au code suivant :

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, l’Event Handler increment() est appelé. Dans cette fonction, nous utilisons notre setter d'état setCount() d'une nouvelle manière ! Étant donné que la valeur suivante de count dépend de la valeur précédente de count, nous passons une fonction de rappel comme argument pour setCount() au lieu d'une valeur (comme nous l'avons fait dans les exercices précédents).

setCount(prevCount => prevCount + 1);

Lorsque notre setter de state appelle la fonction de rappel, cette fonction de rappel de setter de state prend notre décompte précédent comme argument. La valeur renvoyée par cette fonction de rappel de paramètre d'état est utilisée comme valeur suivante de count (dans ce cas, prevCount + 1). Remarque : nous pouvons 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 callback. Si vous souhaitez en savoir plus sur les raisons pour lesquelles la méthode de rappel est plus sûre, cette section de la documentation est un excellent point de départ.

**Instructions:**

1.

Définissez un event Handler goBack(). Étant donné que notre prochaine valeur de state dépend de la valeur de state précédente, cette fonction doit appeler le paramètre de state avec une fonction de rappel. Notre fonction de rappel de définition d'état doit calculer la valeur suivante de questionIndex à l'aide d'un argument nommé prevQuestionIndex. Ajoutez un event Listener au bouton "Retour" qui appellera notre event Handler nouvellement défini.

2.

Définissez un event Handler goToNext(). Étant donné que notre prochaine valeur de state dépend de la valeur de state précédente, cette fonction doit appeler le paramètre de state avec une fonction de rappel. Notre fonction de rappel de définition de state doit calculer la valeur suivante de questionIndex à l'aide d'un argument nommé prevQuestionIndex. Ajoutez un event Listener au bouton "Question suivante" qui appellera notre event Handler nouvellement défini.

import React, { useState } from 'react';

export default function QuizNavBar({ questions }) {

  const [questionIndex, setQuestionIndex] = useState(0);

  // define event handlers

const goBack =()=> setQuestionIndex(prevQuestionIndex => prevQuestionIndex - 1);

  // determine if on the first question or not

  const onLastQuestion = questionIndex === questions.length - 1;

return (

    <nav>

      <span>Question #{questionIndex + 1}</span>

      <div>

        <button onClick={goBack}>

          Go Back

        </button>

        <button disabled={onLastQuestion}>

          Next Question

        </button>

      </div>

    </nav>

  );

}

# **Arrays in State**

pensez à la dernière fois que vous avez commandé une pizza en ligne. Mmmm… Une partie du site Web magique qui vous a apporté de la nourriture savoureuse a été construite avec un code comme celui-ci :

import React, { useState } from "react";  
  
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) ? "Rénove " : "Ad "}  
          {option}  
        </buttons>  
      ))}  
      <p>Ordre a {selected.join(", ")} pizza</p>  
    </div>  
  );  
Les tableaux JavaScript sont le meilleur modèle de données pour la gestion et le rendu des listes JSX. Dans cet exemple, nous utilisons deux tableaux :

options est un tableau qui contient les noms de toutes les garnitures de pizza disponibles.

selected est un tableau représentant les garnitures sélectionnées pour notre pizza personnelle.

Le tableau d'options contient des données statiques, ce qui signifie qu'il ne change pas. Nous aimons définir des modèles de données statiques en dehors de nos composants de fonction car ils n'ont pas besoin d'être recréés à chaque fois que notre composant est restitué. Dans notre JSX, nous utilisons la méthode map pour afficher un bouton pour chacune des garnitures de notre tableau d'options.

Le tableau sélectionné contient des données dynamiques, ce qui signifie qu'il change, généralement en fonction des actions d'un utilisateur. Nous initialisons la sélection en tant que 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 quelle garniture a été cliquée.

Lors de la mise à jour d'un tableau dans 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 à partir du tableau précédent doivent être explicitement copiées dans notre nouveau tableau. C'est ce que cette syntaxe de propagation fait pour nous : ...prev.

Remarquez comment nous utilisons les méthodes includes(), filter() et map() de nos tableaux. S'ils sont nouveaux pour vous ou si vous souhaitez simplement un rappel, prenez une minute pour passer en revue ces méthodes de tableau. Nous n'avons pas besoin d'être des gourous JavaScript à part entière pour créer des interfaces utilisateur React, mais sachez qu'investir du temps pour renforcer nos compétences JavaScript nous aidera toujours à faire plus vite (et à nous amuser beaucoup plus) en tant que développeurs React.

### **Instructions**

1. Déclarez et initialisez une variable de state appelée cart qui gardera une trace d'une liste de valeurs de chaîne.

**2.** Ajoutez des paramètres à nos deux gestionnaires d'événements. addItem() doit accepter un argument appelé item et removeItem() doit accepter un élément appelé targetIndex.

**3.** Utilisez la syntaxe de propagation de tableau pour ajouter un nouvel article à notre état de panier lorsque la fonction addItem() est appelée.

**4.** Lorsque la fonction removeItem() est appelée, utilisez la méthode array filter() pour supprimer l'élément de notre état qui se trouve à l'index de l'élément sur lequel vous avez cliqué dans notre liste.

import React, { useState } from "react";

import ItemList from "./ItemList";

import { produce, pantryItems } from "./storeItems";

export default function GroceryCart() {

  // declare and initialize state

  const [cart, setCart] = useState([])

  const addItem = (item) => {

    setCart([...cart, item])

   }

  const removeItem = (targetIndex) => {

    setCart((prev) => {

      return prev.filter((myCart, index) => index !== targetIndex);

    })

  }

  return (

    <div>

      <h1>Grocery Cart</h1>

      <ul>

        {cart.map((item, index) => (

          <li onClick={() => removeItem(index)} key={index}>

            {item}

          </li>

        ))}

      </ul>

      <h2>Produce</h2>

      <ItemList items={produce} onItemClick={addItem} />

      <h2>Pantry Items</h2>

      <ItemList items={pantryItems} onItemClick={addItem} />

    </div>

  )

}

Import React from "react";

export default function ItemList({ items, onItemClick }) {

  const handleClick = ({ target }) => {

    const item = target.value;

    onItemClick(item);

  };

  return (

    <div>

      {items.map((item, index) => (

        <button value={item} onClick={handleClick} key={index}>

          {item}

        </button>

      ))}

    </div>

  )

}

# **Objects in State**

Lorsque nous travaillons avec un ensemble de variables liées, il peut être très utile de les regrouper dans un objet. Prenons un exemple !

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 éléments à noter :

* Nous utilisons une fonction callback de setter de state pour mettre à jour le state en fonction de la valeur précédente
* La syntaxe de propagation est la même pour les objets que pour les tableaux :{ ...oldObject, new Key: newValue }
* Nous réutilisons notre gestionnaire d'événements sur plusieurs entrées en utilisant l'attribut de nom de la balise d'entrée pour identifier l'entrée d'où provient l'événement de modification.

Encore une fois, lors de la mise à jour de l'état avec setFormState() à l'intérieur d'un composant de fonction, nous ne modifions pas le même objet. Nous devons copier les valeurs de l'objet précédent lors de la définition de la valeur suivante de l'état. Heureusement, la syntaxe de propagation rend cela super facile à faire !

Chaque fois que l'une des valeurs d'entrée est mise à jour, la fonction handleChange() sera appelée. À l'intérieur de event handlers, nous utilisons la décompression d'objet pour décompresser la propriété cible de notre objet événement, puis nous utilisons à nouveau la décompression d'objet pour décompresser les propriétés de nom et de valeur de l'objet cible.

À l'intérieur de notre fonction callback de paramètre de state, 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 de propagation, pour remplir les champs correspondants de notre état précédent. Enfin, nous écrasons la clé appropriée avec sa valeur mise à jour. Avez-vous remarqué les crochets autour du nom ? Ce nom de propriété calculé nous permet d'utiliser la valeur de chaîne stockée par la variable de nom comme clé de propriété !

### **Instructions**

**1.**

Tout au long de notre JSX, nous recherchons les propriétés stockées sur l'objet de profil. Sur le premier rendu, c'est un problème car tenter d'obtenir la valeur d'une propriété à partir d'un objet qui n'a pas été défini provoque l'envoi d'une erreur par JavaScript. Pour se défendre contre ces erreurs, initialisons le profil en tant qu'objet vide !

**2.**

Ajoutez event listeners à nos balises JSX pour appeler handleChange() chaque fois qu'un utilisateur tape dans un champ de saisie.

**3.**

Rendons notre fonction handleChange() un peu plus facile à lire. Utilisez **object destructuring** pour initialiser le nom et la valeur de manière plus concise.

**4.**

Il y a un bogue dans notre code ! L'avez-vous remarqué ? Essayez de saisir une entrée, puis saisissez une autre entrée. Ce qui se produit? Pourquoi?

Chaque fois que nous appelons setProfile() dans notre event Handlers, nous donnons au profil la valeur d'un nouvel objet avec le nom et la valeur de l'entrée la plus récemment modifiée, mais nous perdons les valeurs qui étaient stockées pour les entrées avec tout autre nom.

Utilisons l'opérateur de propagation pour corriger ce bogue. Nous voulons copier toutes les valeurs de notre objet de profil précédent chaque fois que nous appelons notre fonction de définition de state. Utilisez prevProfile comme argument pour notre fonction de rappel de paramètre de state.

**5.**

Ajoutez un écouteur d'événement pour appeler notre fonction handleSubmit() lorsque l'utilisateur soumet le formulaire.

import React, { useState } from "react";

export default function EditProfile() {

  const [profile, setProfile] = useState({})

  const handleChange = ({ target }) => {

    //we use a object destructuring

    const { name, value } = target

    setProfile((prevProfile) => ({

      ...prevProfile,

      [name] : value

    }))

  }

  const handleSubmit = (event) => {

    event.preventDefault();

    alert(JSON.stringify(profile, '', 2))

  }

  return (

    <form action="/action.php" onSubmit={handleSubmit}>

      <input

        value={profile.firstName || ''}

        name="firstName"

        type="text"

        placeholder="First Name"

        onChange={handleChange}

      />

      <input

        value={profile.lastName || ''}

        type="text"

        name="lastName"

        placeholder="Last Name"

        onChange={handleChange}

      />

      <input

        value={profile.bday || ''}

        type="date"

        name="bday"

        onChange={handleChange}

      />

      <input

        value={profile.password || ''}

        type="password"

        name="password"

        placeholder="Password"

        onChange={handleChange}

      />

      <button type="submit">Submit</button>

    </form>

  );

}

# **Separate Hooks for Separate States**

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 séparer les données qui changent séparément en variables d'état complètement différentes. 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 contenant un état pour une matière que vous étudiez à l'école, cela 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 il pourrait être désordonné de copier toutes les autres valeurs lorsque nous devons mettre à jour quelque chose dans ce grand objet de state. Par exemple, pour mettre à jour la note d'un examen, nous aurions besoin d'un event listeners qui ferait 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;  
    }  
  }),

Aïe ! Un code complexe comme celui-ci est susceptible de provoquer des bogues ! Heureusement, il existe une autre option… Nous pouvons faire plus d'un appel au State Hook. En fait, nous pouvons faire autant d'appels à useState() que nous le voulons ! Il est préférable de diviser l'état en plusieurs variables d'état en fonction des valeurs qui ont tendance à changer ensemble. Nous pouvons 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 de state distinctes présente de nombreux avantages, comme rendre notre code plus simple à écrire, lire, tester et réutiliser à travers les composants.

Souvent, nous nous retrouvons à emballer et à organiser des données dans des collections à transmettre entre les composants, puis à séparer ces mêmes 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 !

Si vous le souhaitez, jetez un œil à un autre exemple d'utilisation multiple State Hooks for managing separate data.

### **Instructions**

**1.**

À l'intérieur du composant MusicalRefactored, réécrivez le state de Musical afin que son objet de state soit divisé en trois variables distinctes : title, actors et locations, chacun avec sa propre fonction de définition de state.

State Hook de React nous permet de nommer nos setters de state comme nous le voulons, mais vous avez peut-être remarqué un modèle :

* toggle & setToggle()
* isLoading & setIsLoading()
* email & setEmail()

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"

      }

    }

  })

 }

function MusicalRefactored() {

// write your code here: Declare title as its own state variable

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"

      }})

}

# **Lesson Review**

Travail génial, nous pouvons maintenant créer des composants de fonction « avec état » à l'aide du crochet useState React !

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

* Avec React, nous fournissons des modèles de données statiques et dynamiques à JSX pour afficher une vue à l'écran
* Utilisez des crochets pour "s'accrocher" à l'état des composants internes pour gérer les données dynamiques dans les composants de fonction
* Nous utilisons le State Hook en utilisant le code ci-dessous :
* currentState to reference the current value of state
* stateSetter to reference a function used to update the value of this state
* the initialState argument to initialize the value of state for the component’s first render

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

* Appeler les setters DE STATE dans les event handlers
* Définissez des event handlers simples en ligne avec nos écouteurs d'événements JSX et définissez des gestionnaires d'événements complexes en dehors de notre JSX
* Utiliser une fonction de rappel de state setter lorsque notre valeur suivante dépend de notre valeur précédente
* Utilisez des tableaux et des objets pour organiser et gérer les données associées qui ont tendance à changer ensemble
* Utilisez la syntaxe de propagation sur les collections de données dynamiques pour copier l'état précédent dans l'état suivant comme suit : setArrayState((prev) => [ ...prev ]) and setObjectState((prev) => ({ ...prev } ))
* Diviser l'état en plusieurs variables plus simples au lieu de tout jeter dans un seul objet d'état