**Why Use useEffect?**

Avant Hooks, les composants de fonction n'étaient utilisés que pour accepter des données sous forme des props et renvoyer du JSX à rendre. Cependant, comme nous l'avons appris dans la dernière leçon, le State Hook 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 pour exécuter du code JavaScript après chaque rendu, par exemple :

* récupérer des données à partir d'un service backend.
* souscrire à un flux de données.
* gestion des minuteries et des intervalles.
* lire et apporter des modifications au DOM.

**Why after each render?**

Les composants les plus intéressants seront restitués plusieurs fois au cours de leur durée de vie et ces moments clés offrent l'occasion idéale d'exécuter ces «effets secondaires».

Il y a trois moments clés où le crochet d'effet peut être utilisé :

* Lorsque le composant est ajouté ou monté pour la première fois au DOM et rend.
* Lorsque le state ou les props changent, provoquant un nouveau rendu du composant.
* Lorsque le composant est supprimé ou démonté du DOM.

**Instructions :**

Dans l'éditeur, nous avons défini un composant à la fois comme une classe et comme une fonction, chacun avec les mêmes "effets secondaires".

Même si vous n'êtes pas familier avec [class component lifecycle methods](https://reactjs.org/docs/react-component.html) , commencez par jeter un coup d'œil aux deux implémentations, juste pour avoir une idée des deux options.

Remarque : La compréhension des méthodes de cycle de vie dans les composants de classe n'est pas un prérequis pour cette leçon.

Dans le composant de classe de **PageTitleClass.js**, la logique de définition du titre du document est répartie entre deux fonctions -componentDidMount() et componentDidUpdate(). Comparez cela au composant de fonction dans **PageTitleFunction.js**, où la logique est écrite à un seul endroit – useEffect().

componentDidMount() + componentDidUpdate() => useEffect() dans le cycle de vie d'un composant fonctionnel.

Les deux implémentations de composants ont le même comportement, mais la lecture et la maintenance du composant de fonction seront plus faciles. Prêt à commencer à utiliser le crochet d'effet ? Génial! Nous allons plonger dans les détails de l'utilisation de ce crochet tout au long de cette leçon !

**Function Component Effects**

Voyons comment notre composant de fonction PageTitle() utilise le hook d'effet pour exécuter du code après chaque rendu !

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

function PageTitle() {  
  const [name, setName] = useState('');  
  useEffect(() => {  
    document.title = `Hi, ${name}`;

console.log("useEffect is called");  
  },[name]);

// notre effet soit appelé seulement que après le premier rendu du composant.  
  return (  
    <div>  
      <p>Use the input field below to rename this page!</p>  
      <input onChange={(e) => setName(e.target.value)} value={name} type='text' />  
    </div>  
  );  
}

Tout d'abord, nous importons l'Effect Hook de la bibliothèque React, comme ceci:

import { useEffect } from 'react';

Le crochet d'effet est utilisé pour appeler une autre fonction qui fait quelque chose pour nous, donc rien n'est renvoyé lorsque nous appelons la fonction useEffect().

Le premier argument passé à la fonction useEffect() est la fonction de rappel que nous voulons que React appelle après chaque rendu de ce composant. Nous appellerons cette fonction de rappel notre effet.

Dans notre exemple, l'effet est :

() => { document.title = name; }

Dans notre effet, nous attribuons la valeur de la variable name au document.title. Pour en savoir plus sur cette syntaxe, consultez cette explication de la propriété title du document.

Remarquez comment nous utilisons le state actuel à l'intérieur de 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 de fonction ! Lorsque React rend notre composant, il mettra à jour le DOM comme d'habitude, puis exécutera notre effet après la mise à jour du DOM. Cela se produit pour chaque rendu, y compris le premier et le dernier.

**Instructions :**

1.

Importez Effect Hook, ainsi que State Hook et React depuis la bibliothèque 'react'.

2.

Appelez useEffect() avec une fonction de rappel qui crée une alerte avec la valeur actuelle de count. Commencez à cliquer sur le bouton pour voir quand notre fonction alert() est appelée et assurez-vous qu'elle enregistre les valeurs auxquelles nous nous attendons !

3.

Utilisez un modèle littéral pour que le message dans notre alertdialog, indique : " Compte : 0 ", puis " Compte : 1 ", puis " Compte : 2 ", etc.

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

export default function Counter() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    alert(`Count: ${count}`);

console.log("useEffect is called");

  },[count])

//notre effect est appelé seulement aprés le premier rendu du composant.

  const handleClick = () => {

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

  };

  return (

    <div>

      <p>You clicked {count} times</p>

      <button onClick={handleClick}>

        Click me

      </button>

    </div>

  );

}

**Clean Up Effects**

Certains effets nécessitent un nettoyage. Par exemple, nous pourrions vouloir ajouter des event listeners à certains éléments du DOM, au-delà du JSX dans notre composant. quand nous  add event listeners to the DOM , il est important de supprimer ces event listeners lorsque nous en avons fini avec eux pour éviter [memory leaks](https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/)!

Considérons l'effet suivant :

useEffect(()=>{  
  document.addEventListener('keydown', handleKeyPress);  
  return () => {  
    document.removeEventListener('keydown', handleKeyPress);  
  };  
})

Si notre effet ne renvoyait pas de fonction de nettoyage, un nouvel event listener serait alors ajouté à l'objet document du DOM chaque fois que notre composant restituerait. Non seulement cela provoquerait des bogues, mais cela pourrait également entraîner une diminution des performances de notre application et peut-être même un plantage !

Parce 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, le hook useEffect() la traite toujours comme une fonction de nettoyage. React appellera cette fonction de nettoyage avant que le composant ne restitue ou ne se démonte. Étant donné que cette fonction de nettoyage est 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.

**Instructions :**

1.

Écrivez un event handler nommé increment(). Définissez cette fonction pour qu'elle appelle setClickCount() avec une fonction de rappel d'état, en ajoutant 1 à la valeur précédente de , cliquez sur Count.

2.

Importez le hook useEffect() et appelez-le avec un effet qui ajoute un écouteur d'événement pour les événements 'mousedown' sur l'objet document. Lorsqu'un événement "mousedown" se produit n'importe où sur le document, nous voulons que notre gestionnaire d'événements increment() soit appelé.

3.

Si vous ne l'avez pas déjà fait, exécutez notre code et cliquez autour de la fenêtre du navigateur. Qu'est-ce qui se passe? Pourquoi cela arrive-t-il?

Chaque fois que notre composant est rendu, notre effet est appelé, ajoutant un autre event listener. En quelques clics et rendus, nous avons attaché de nombreux auditeurs d'événements au DOM ! Nous devons nettoyer après nous!

Mettez à jour notre effet afin qu'il renvoie une fonction de nettoyage (cleanup) qui supprimera notre dernier event listener du DOM.

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

export default function Counter() {

  const [clickCount, setClickCount] = useState(0);

  const increment = () => setClickCount((prev) => prev + 1);

const decrement = () => setClickCount((prev) => prev - 1);

  useEffect(() => {

document.title = `count: ${clickCount}`

    document.addEventListener('mouseup', increment);

console.log("component rendered for the first time and added from the DOM");

    return () => {

      document.removeEventListener('mousedown', decrement);

console.log("component is being removed from the DOM");

    };

  }, [clickCount]);

  return (

      <h1>Document Clicks: {clickCount}</h1>

  );

}

**Control When Effects Are Called**

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 et d'autres bogues, mais parfois nous voulons ignorer complètement l'appel de notre effet sur les nouveaux rendus.

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

Le tableau de dépendances est utilisé pour dire à la méthode useEffect() quand appeler notre effet et quand l'ignorer. Notre effet est toujours appelé après le premier rendu, mais seulement appelé à nouveau 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 lorsqu'un composant est monté pour la première fois, et si une fonction de nettoyage est renvoyée par notre effet, appelant cela 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 à useEffect() ci-dessus, 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. Simplement, passer [] à la fonction useEffect() suffit pour configurer quand les fonctions d'effet et de nettoyage sont appelées !

**Instructions :**

1.

Commençons par utiliser quatre fonctions pour faire avancer le nombre stocké par temps chaque seconde :

useEffect() - le crochet d'effet, importé de la bibliothèque 'react'

setInterval() - Documentation W3Schools

setTime() - notre fonction de définition d'état

fonction de rappel du paramètre d'état - utilisée par setTime() car nous voulons calculer la prochaine valeur de temps en fonction de la valeur de temps précédente. Ajoutez un effet qui utilise la fonction setInterval() pour appeler setTime() toutes les secondes (ou 1000 ms).

2.

Eh bien, cela ne semble pas tout à fait en ce moment, n'est-ce pas ? Notre valeur temps se met à jour trop rapidement ! Une idée pourquoi cela peut se produire?

Par défaut, le crochet d'effet appelle notre effet après chaque rendu. Notre effet est de créer un nouvel intervalle qui met à jour la valeur du temps à chaque seconde. Nous continuons à ajouter de plus en plus d'intervalles qui mettent à jour la même variable de temps. Nous devons nettoyer nos anciens intervalles avant d'en ajouter de nouveaux !

Dans votre effet, utilisez le mot-clé return pour renvoyer une fonction de nettoyage. Notre fonction de nettoyage utilisera la fonction clearInterval(). Pour en savoir plus sur l'utilisation de cette fonction de minuterie JavaScript, jetez un coup d'œil à l'explication de W3Schools.

3.

Eh bien, cela semble avoir résolu notre bug de trop d'intervalles de mise à jour de la même variable !

Créons un nouvel élément de saisie de texte afin que l'utilisateur puisse taper un message pendant que la minuterie compte.

Ajoutez une balise d'entrée à notre JSX dont la valeur est gérée par une variable d'état actuelle appelée name avec un paramètre d'état appelé setName().

Définissez un gestionnaire d'événements nommé handleChange() et appelez ce gestionnaire d'événements avec l'attribut d'écouteur d'événement onChange de l'entrée.

4.

Oh oh. Plus de bogues. L'avez-vous déjà remarqué ? Tapez votre nom complet dans le champ de saisie de texte. Voyez comment la minuterie semble arrêter de compter pendant que vous tapez ? Ce n'est pas ce que nous voulons !

Qu'est-ce qui se passe ici? Nous créons un nouvel intervalle après chaque rendu, cet intervalle appellera notre state setter pour mettre à jour le temps exactement une seconde après chaque rendu. Lorsque nous tapons dans le champ de saisie, notre composant continue de restituer, de nettoyer les anciens intervalles et d'en démarrer de nouveaux… mais notre setter d'état n'est jamais appelé avant une seconde après que nous ayons fini de taper !

Réglons cela une fois pour toutes ! Nous voulons vraiment utiliser un seul intervalle. Nous voulons que cet intervalle commence à s'écouler après notre premier rendu et nous voulons qu'il soit nettoyé après le rendu final. Pour ce faire, utilisez un tableau de dépendances vide !

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

export default function Timer() {

  const [time, setTime] = useState(0);

  const [name, setName] = useState("");

  //code here:

  useEffect(() => {

  //new Date() creates a new date object with the current date and time:

   const intervalId = setInterval(() => {

      setTime((prev) => prev + 1);

    }, 1000)

   // Return a clean up function from our effect

   return () => {

     clearInterval(intervalId);

   }

  },[])

  // Define an event handler named handleChange()

    const handleChange = ({ target }) => {

        setName(target.value)

    }

  return (

    <>

      <h1>Time: {time}</h1>

      <input type='text' value={name} onChange={handleChange} />

    </>

  );

}

**Fetch Data**

Lors de la création de logiciels, nous commençons souvent avec des comportements par défaut, puis nous les modifions pour améliorer les performances. Nous avons appris que le comportement par défaut du crochet d'effet est d'appeler la fonction d'effet après chaque rendu. Ensuite, nous avons appris que nous pouvions passer un tableau vide comme deuxième argument pour useEffect() si nous voulons seulement que notre effet soit appelé après le premier rendu du composant. Dans cet exercice, nous apprendrons à utiliser le tableau de dépendances pour configurer plus précisément quand nous voulons que notre effet soit appelé !

Lorsque notre effet est responsable de la récupération des données d'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 :

* Processing
* Performance
* Data usage for mobile users
* API service fees

Lorsque les données que nos composants doivent rendre ne changent pas, nous pouvons transmettre un tableau de dépendances vide, de sorte que les données soient récupérées après le premier rendu. Lorsque la réponse est reçue du serveur, nous pouvons utiliser un setter d'état du State Hook pour stocker les données de la réponse du serveur dans notre état de composant local pour les rendus futurs. Utiliser le crochet d'état et le crochet d'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 Effect Hook 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 appeler notre effet une fois suffit.

Un tableau de dépendances qui n'est pas vide signale au crochet d'effet qu'il peut ignorer l'appel de notre effet après un nouveau rendu, sauf si la valeur de l'une des variables de notre tableau de dépendances a changé. Si la valeur d'une dépendance a changé, alors le crochet d'effet appellera à nouveau notre effet !

Voici un bel exemple de la [official React docs](https://reactjs.org/docs/hooks-effect.html):

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

**Instructions :**

1.

Nous avons commencé à créer une application de planification météo qui récupère des données sur la météo et permet à nos utilisateurs d'écrire des notes à côté des prévisions. Beaucoup de bon code a déjà été écrit, mais il n'y a actuellement rien de rendu à l'écran.

Lisons le code et commençons à comprendre ce qui se passe ici. Selon nous, quelle partie de notre code empêche le composant de s'afficher ?

Dans notre JSX, nous essayons de mapper sur un tableau stocké par data, mais notre effet qui récupère ces données n'est appelé qu'après le premier rendu. Ainsi, lors du premier rendu, les données sont indéfinies et la tentative d'appel de map() sur undefined provoque notre erreur !

Évitons cette erreur en vérifiant si les données ont déjà été chargées. Si ce n'est pas le cas, nous voulons que notre composant de fonction renvoie simplement une balise de paragraphe avec le texte "Chargement...". Si les données ne sont plus indéfinies, alors les données ont été chargées, et nous pouvons aller de l'avant et rendre le JSX complet !

2.

Notre récupération de données se fait dans notre effet. Remarquez comment nous utilisons actuellement les messages alert() pour garder une trace de la demande et de la réception de données de notre serveur. Au lieu de simplement chaîner les données de réponse et de les afficher dans un message d'alerte, stockons ces données dans notre state.

Lorsque les données ont été récupérées, utilisez notre fonction de définition de state pour stocker ces données dans le state de notre composant !

PS N'oubliez pas que nous voulons stocker un tableau dans notre variable de state de données, et non l'intégralité de l'objet de réponse.

3.

Tapez dans chacun des champs de saisie des notes dans notre tableau. Que remarquez-vous ? Pourquoi pensez-vous que cela se passe?

Chaque fois que nous tapons dans un champ de saisie, le composant s'affiche à nouveau pour afficher la nouvelle valeur de ce champ. Même si nous n'avons pas besoin de nouvelles données du backend, notre composant récupère de nouvelles données après chaque rendu !

Utilisez un tableau de dépendances vide pour vous assurer que les données ne sont récupérées qu'après le premier rendu de notre composant.

4.

Wow, ce petit changement de code a fait une énorme différence dans les performances de notre application de planification météo !

Apportons une autre amélioration. Avez-vous remarqué les boutons en haut de notre application ? Nous voulons que nos utilisateurs puissent choisir entre planifier en fonction des prévisions météorologiques quotidiennes et des prévisions météorologiques hebdomadaires. Cliquer sur ces boutons ne change actuellement rien. Réparons ça !

Le serveur a deux points de terminaison différents appelés : /daily et /hourly. Utilisons la valeur de la variable d'état ForecastType pour déterminer à partir de quel point de terminaison notre effet doit demander des données.

Après avoir effectué cette modification, notre effet se comporte différemment en fonction de la valeur de forecastType. On pourrait dire que la façon dont nous utilisons notre effet en dépend ! Ajoutez cette variable à notre tableau de dépendances afin que l'effet soit appelé à nouveau, en mettant à jour les données de manière appropriée, après un nouveau rendu où l'utilisateur a sélectionné un type de prévision différent.

Data.js

// You don't need to look at this, but you can if you want!

//

// In this modele, we use some functions like Math.random(),

// Math.floor(), and Array.from() to generate some fake data.

// When building user interfaces, it can be helpful to

// mock out simplified versions of the data that our code

// will eventually fetch from backend servers

const getRandomId = () => `${Math.random()}-${Math.random()}`;

const getRandomNumber = (min, range) =>

  Math.floor((Math.random() \* 100 \* range) / 100) + min;

const summaries = [

  "Rainy",

  "Cloudy",

  "Partly Cloudy",

  "Partly Sunny",

  "Mostly Sunny",

  "Sunny"

];

const randomFromList = (list) => list[getRandomNumber(0, list.length)];

const getTemp = () => {

  const avg = getRandomNumber(60, 30);

  return {

    avg,

    min: avg - 10,

    max: avg + 10

  };

};

const getWeatherItem = () => ({

  id: getRandomId(),

  summary: randomFromList(summaries),

  temp: getTemp(),

  precip: getRandomNumber(0, 100)

});

export default {

  "/daily": Array.from({ length: 5 }, getWeatherItem),

  "/hourly": Array.from({ length: 24 }, getWeatherItem)

};

fetch.js

// You don't need to look at this, but you can if you want!

// This module exports fake data fetching functionality.

// In a real app, this would grab data from the internet, but

// this module just waits a little bit before responding.

import DATA from "./data";

export function get(endpoint) {

  const delay = Math.floor(Math.random() \* 1000);

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (!DATA.hasOwnProperty(endpoint)) {

        const validEndpoints = Object.keys(DATA)

          .map((endpoint) => ` - "${endpoint}"`)

          .join("\n ");

        reject(

          `"${endpoint}" is an invalid endpoint. Try getting data from: \n ${validEndpoints}`

        );

      }

      const response = { status: 200, data: DATA[endpoint] };

      resolve(response);

    }, delay);

  });

}

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

import { get } from './mockBackend/fetch';

export default function Forecast() {

  const [data, setData] = useState(null);

  const [notes, setNotes] = useState({});

  const [forecastType, setForecastType] = useState('/daily');

  useEffect(() => {

    alert('Requested data from server...');

    get(forecastType).then((response) => {

      alert('Response: ' + JSON.stringify(response,'',2));

      setData(response.data);

    });

  }, [forecastType]);

  const handleChange = (itemId) => ({ target }) =>

    setNotes((prev) => ({

      ...prev,

      [itemId]: target.value

    }));

  if (!data) {

    return <p>Loading...</p>;

  }

  return (

    <div className='App'>

      <h1>My Weather Planner</h1>

      <div>

        <button onClick={() => setForecastType('/daily')}>5-day</button>

        <button onClick={() => setForecastType('/hourly')}>Today</button>

      </div>

      <table>

        <thead>

          <tr>

            <th>Summary</th>

            <th>Avg Temp</th>

            <th>Precip</th>

            <th>Notes</th>

          </tr>

        </thead>

        <tbody>

          {data.map((item) => {

            return (

              <tr key={item.id}>

                <td>{item.summary}</td>

                <td> {item.temp.avg}°F</td>

                <td>{item.precip}%</td>

                <td>

                  <input

                    value={notes[item.id] || ''}

                    onChange={handleChange(item.id)}

                  />

                </td>

              </tr>

            );

          })}

        </tbody>

      </table>

    </div>

  );

}

**Rules of Hooks**

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

* n'appelez les hooks qu'au niveau supérieur
* appeler uniquement les hooks à partir des fonctions React

Comme nous nous sommes entraînés avec le hook de state et le hook d'effet, nous avons suivi ces règles avec facilité, mais il est utile de garder ces deux règles à l'esprit lorsque vous sortez votre nouvelle compréhension des hooks dans la nature et que vous commencez à utiliser plus de Hooks dans vos applications React.

Lorsque React construit le DOM virtuel, la bibliothèque appelle les fonctions qui définissent nos composants encore et encore lorsque l'utilisateur interagit avec l'interface utilisateur. React garde une trace des données et des fonctions que nous gérons avec Hooks en fonction de leur ordre dans la définition du composant de fonction. Pour cette raison, nous appelons toujours nos hooks au niveau supérieur ; nous n'appelons jamais de crochets 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 constamment Hook à chaque fois :

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

Deuxièmement, les hooks ne peuvent être utilisés que dans les fonctions React. Nous ne pouvons pas utiliser les hooks dans les composants de classe et nous ne pouvons pas utiliser les hooks dans les fonctions JavaScript régulières. Nous avons travaillé avec useState() et useEffect() dans les composants de fonction, et c'est l'utilisation la plus courante. Le seul autre endroit où les hooks peuvent être utilisés est dans les hooks personnalisés. Les hooks personnalisés sont incroyablement utiles pour organiser et réutiliser la logique avec state entre les composants de la fonction. Pour en savoir plus sur ce sujet, rendez-vous sur [React Docs](https://reactjs.org/docs/hooks-custom.html).

**Instructions :**

1.

Le code avec lequel nous commençons a beaucoup de bonnes idées, mais il y a quelques bogues que nous devons aider à résoudre. Commençons par refactoriser le code afin que le State Hook soit toujours appelé au niveau supérieur.

Il semble que les développeurs qui ont écrit ce code aient voulu retarder l'utilisation des variables de state selectedCategory et items jusqu'à ce que les catégories aient été récupérées. Conceptuellement, cela a du sens, mais React nécessite que tous les crochets soient appelés à chaque rendu, donc l'imbrication de ces appels useState() n'est pas une option valide.

Amenons tous nos appels State Hook au plus haut niveau.

Pour être clair sur les valeurs initiales, définissons explicitement la valeur de state initial pour les catégories et selectedCategory sur null.

2.

Cela ressemble à l'idée derrière l'utilisation de cette expression : if (!categories) était de ne récupérer qu'une seule fois les données des catégories du serveur.

L'imbrication d'un appel au crochet d'effet dans une condition comme celle-ci entraînera l'appel de différents hooks sur différents rendus, ce qui entraînera des erreurs. Heureusement, nous connaissons un meilleur moyen!

Refactorisez ce code afin que l'effet chargé de récupérer les données de categories à partir du backend et de les enregistrer dans le state local suive les règles des hooks et ne récupère les données de catégories qu'une seule fois.

3.

Ouf, on avance bien ! C'est tellement agréable de transformer les écrans d'erreur en code fonctionnel, n'est-ce pas ?

Maintenant que nous récupérons la liste des catégories depuis le backend et que nous affichons avec succès les boutons de chacune d'entre elles à l'écran, nous sommes prêts à utiliser un autre effet pour récupérer les éléments de chacune de ces catégories, lorsque l'utilisateur clique sur chacune d'entre elles !

Décommentez le bloc de code qui tentait de le faire et refactorisez-le afin que nous suivions les règles de Hooks. Pour optimiser les performances, n'appelez le backend pour les données que lorsque nous ne les avons pas encore stockées dans l'état du composant comme ce code essayait de le faire.

HINT/

Le comportement de cet effet dépend des valeurs de deux variables : items et selectedCategory. Bien que nous ne puissions pas imbriquer tout l'appel de fonction useEffect() dans le bloc if, ce code…

if (selectedCategory && !items[selectedCategory]) { }

… nous est toujours très utile. Si selectedCategory et !items[selectedCategory] sont tous les deux véridiques, alors nous savons que l'utilisateur a cliqué sur un bouton pour voir les éléments d'une catégorie pour laquelle nous n'avons pas encore les données, nous voulons donc les récupérer depuis le backend et stockez-les dans le state local, sinon, nous n'avons pas besoin de récupérer quoi que ce soit du backend.

Nous avons déjà la plupart du code correct, nous avons juste besoin de le réorganiser un peu :

useEffect(() => {  
    if (selectedCategory && !items[selectedCategory]) {  
     /\* fetch data and store it to local state \*/  
    }  
  }, [ /\* list the two variables that this effect depends on here \*/]);

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

import { get } from './mockBackend/fetch';

export default function Shop() {

  const [categories, setCategories] = useState(null);

  const [selectedCategory, setSelectedCategory] = useState(null);

  const [items, setItems] = useState({});

  useEffect(() => {

      get('/categories').then((response) => {

        setCategories(response.data);

      });

  });

useEffect(() => {

      if (selectedCategory && !items[selectedCategory]) {

      get(`/items?category=${selectedCategory}`).then((response) =>  {

         setItems((prev) => ({ ...prev, [selectedCategory]: response.data }));

       });

       // Use a dependency array to control when the Effect Hook calls the get() function

     },[items, selectedCategory]);

}

   // Pass two variables in our dependency array so that the Effect Hook only re-calls our effect when the value of items or selectedCategory changes.

  if (!categories) {

    return <p>Loading..</p>;

  }

  return (

    <div className='App'>

      <h1>Clothes 'n Things</h1>

      <nav>

        {categories.map((category) => (

          <button key={category} onClick={() => setSelectedCategory(category)}>

            {category}

          </button>

        ))}

      </nav>

      <h2>{selectedCategory}</h2>

      <ul>

        {!items[selectedCategory]

          ? null

          : items[selectedCategory].map((item) => <li key={item}>{item}</li>)}

      </ul>

    </div>

  );

}

Data.js:

// You don't need to look at this, but you can if you want!

/\*

When building user interfaces, it can be helpful to

mock out simplified versions of the data that our code

will eventually fetch from backend servers

\*/

export default {

  "/categories": ["Shirts", "Pants", "Shoes", "Accessories"],

  "/items?category=Shirts": ["T-Shirts", "Casual", "Formal"],

  "/items?category=Pants": ["Long Pants", "Sweat Pants", "Shorts", "Swimwear"],

  "/items?category=Shoes": ["Athletic", "Professional", "Casual", "Walking"],

  "/items?category=Accessories": ["Hats", "Wallets", "Belts"]

};

Fetch.js:

// You don't need to look at this, but you can if you want!

// This module exports fake data fetching functionality.

// In a real app, this would grab data from the internet, but

// this module just waits a little bit before responding.

import DATA from "./data";

export function get(endpoint) {

  const delay = Math.floor(Math.random() \* 1000) + 500;

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (!DATA.hasOwnProperty(endpoint)) {

        const validEndpoints = Object.keys(DATA)

          .map((endpoint) => ` - "${endpoint}"`)

          .join("\n ");

        reject(

          `"${endpoint}" is an invalid endpoint. Try getting data from: \n ${validEndpoints}`

        );

      }

      const response = { status: 200, data: DATA[endpoint] };

      resolve(response);

    }, delay);

  });

}

**Separate Hooks for Separate Effects**

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 dans 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);  
}, []);

Pour la simplicité ici, où nous avons séparé les soucis :

// 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 la pratique, nous devenons meilleurs pour organiser notre code afin qu'il soit plus facile à comprendre, à ajouter, à réutiliser et à tester !

**Instructions :**

1.

Pour le moment, ce code semble fonctionner correctement. Trois requêtes réseau différentes sont effectuées en un seul effet et de nombreuses données différentes et non liées sont gérées dans une seule variable de state. Mettons-nous au travail en décomposant ces appels uniques useState() et useEffect() en crochets séparés et plus simples. Cela rendra ce code plus facile à comprendre, à développer et à réutiliser à mesure que nous continuons à améliorer notre application !

Commencez à refactoriser ce composant :

Utilisez un hook de state séparé pour le menu, newsFeed et friends. Utilisez ces nouveaux setters d'état au lieu de setData() dans l'effet.

Simplifiez notre JSX pour utiliser ces nouvelles variables de state au lieu data.