Cours ReactJS suite

Les Hooks sont apparus dans React v. 16.8.0, ce sont simplement des fonctions qui permettent de bénéficier de fonctionnalités React qu'on pouvait avoir que via une class : exemple : Une état local'state', les cycles de vie, les contextes, etc..

Pas de panique, pas de rupture de comptabilité ni de dépréciation de vos notions React déjà acquises : class, props, state, ref, context, etc.. Donc on peut utiliser les hooks dans nos projets React existants sans devoir réécrire tout notre code. Les hooks sont optionnelle, cependant, si l'on souhaite utiliser les hooks dans nos applis, on doit alors mettre à jour tous nos modules, dont React Dom.

Hooks d'état – useState

C'est une fonction donc, qui nous permet d'avoir un état local (un state) dans un composant de type fonction. Avant on utiliser un composant de type class. Comment faire cela et bien on recrée une appli que vous appellerez HooksApp comme ça on sera tous pareil. Dans notre fichier App.js nous mettrons un h1

```
<div>
  <h1 className="text-center">useState Hooks</h1>
</div>
```

Créez un dossier components, révisons un peu créons un fichier ClassState.js, créez les import classique, la structure type d'une classe et faites en sorte que cela vous retourne une div

```
import React, { Component } from 'react'

class ClassState extends Component {
    render() {
        return (
            <div>

            </div>
        )
    }
}

export default ClassState
```

Et on l'importe dans son parent App.js, puis on l'affiche sous notre h1.

```
<div>
  <h1 className="text-center">useState Hooks</h1>

  <ClassState />
</div>
```

Alors on va créer un state dans notre composant de type class, on peut faire le raccourci (rc) react const qui va nous créer un constructor qui va nous permettre d'avoir un state (vous pouvez faire le state comme vous le connaissais déjà).

```
constructor(props) {
    super(props)

    this.state = {
        counter: 0
    }
}
```

On va créer un compteur en reprenant l'exemple de la doc, et l'afficher dans un 'p'.

```
<div>
    <p>Class State: {this.state.counter}</p>
</div>
```

Ensuite on va faire un bouton qui au click dessus va appeler une méthode qui elle va appeler la seule méthode qui va nous permettre d'agir avec un state local qui est setState qui nous permettra d'incrémenter le compteur.

```
addOne = () => {
    this.setState({
        counter: this.state.counter + 1
    })
}
```

Cependant personnellement, je n'aime pas trop travailler le state de cette manière-là, je préfère agir avec le state précédent et faire en sorte de le modifier si je le souhaite surtout dans le cas d'incrémentation. Je vous montre ça, alors c'est très simple on va retirer notre objet, et nous allons prendre le state précédent donc 'counter' et bien pour cela on fait une arrow function, cette function n'aura qu'un seul paramêtre (le state précédent) donc prevState qui va nous retourner un objet et cet objet là ce sera le counter. Et là on va prendre l'objet prevState que nous avons au niveau de notre state et on va accéder à son counter et l'incrémenter.

```
addOne = () => {
    this.setState(prevState => {
        return {
            counter: prevState.counter + 1
        }
```

Ok tout fonctionne parfaitement quand on click sur le bouton sa incrémente le compteur.

```
<p>Class State: {this.state.counter}</p>

<button onClick={this.addOne}>State dans Class</button>
```

Désormais on va créer un deuxième fichier et cette fois-ci on va avoir un composant de type function et on va reproduire exactement le même counter. Donc components => et on l'appel FunctionState.js puis on créer nos imports et une arrow function qui s'appele FunctionState, qui nous retourne une div avec un 'p', ou on affichera la valeur du counter on met l'exporte et on importe dans le App.js,

```
<ClassState />
<hr />
<FunctionState />
```

Dans une class on a le state dans notre constructor(ou pas), et bien dans le cadre d'une function et bien on doit l'importer de React (FunctionState.js)

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

C'est une function que l'on va déclarer à l'intérieur de notre composant, donc on va faire un useState et lui donner une valeur, contrairement au style qu'on utilise dans la classe qui est d'office un objet, celui que l'on va avoir maintenant dans un composant de type fonction qui va être généré via la fonction useState il peut contenir un string, un number, un boolean, un objet ou même un array. Donc on va partir sur un number(0), cette valeur là on va la mettre dans un array(counter), mais on va avoir un deuxième élément dans ce tableau ????

Et bien ce sera une fonction, elle va jouer le rôle de setState(qu'on utilise dans une classe), on va créer un setter, (le setter est utilisé pour la modification des valeurs des objets, et les getter pour utiliser ces valeurs),

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

const FunctionState = () => {

    const [counter, setCouner ] = useState(0)

    return (
        <div>
            <p>Function State: </p>
        </div>
    )
}

export default FunctionState
```

Vous pouvez faire un console log du counter pour vérifier (affiche 0), très bien comme on arrive à accéder à cette valeur la et bien on va créer un bouton pour incrémenter notre compteur.

```
<div>
    <p>Function State: </p>
    <button onClick={}>State dans Function</button>
</div>
```

Et on va faire notre méthode donc on va l'appeler comme la précédente addOne, et pour modifier la valeur du compteur on va utiliser le setter(setCounter), et rappeler la valeur à modifier(counter) et l'incrémenter de 1

```
const addOne = () => {
    setCounter(counter + 1)
}
```

On n'oublie pas d'appeler notre counter dans le 'p' ainsi que addOne dans le onclick.

```
<div>
    <p>Function State: {counter}</p>
    <button onClick={addOne}>State dans Function</button>
</div>
```

Vous pouvez enregistrer et aller tester, tout fonctionne parfaitement. Nous avons faite exactement la même chose. Encore une fois je préfere agir sur la valeur précédente, je vous montre ça,(vous n'êtes pas obligé c'est juste que je vous montre comment j'agis régulièrement sur les incrémentations) on va prendre la valeur précédente du counter donc prevCounter que nous allons déclarer dans une arrow Function qui va nous retourner cette valeur-là,

```
const addOne = () => {
    setCounter(prevCounter => prevCounter + 1)
}
```

La tout fonctionne très bien. Sachant que désormais on appelle une fonction inutilement(addOne), nous avons une deuxième fonction (setCounter) alors pourquoi ne pas appeler directement celle-ci au niveau du onClick, c'est que l'on va faire. Donc on pend le setCounter, et on le copie, et on peut effacer le addOne.

```
const [counter, setCounter ] = useState(0)


return (
    <div>
        <p>Function State: {counter}</p>
        <button onClick={setCounter(prevCounter => prevCounter + 1)}>State dans Functi
    </div>
)
```

Sauf que sachant que c'est une méthode qui va avoir des données comme paramètres si on la laisse comme ça elle va se lancer automatiquement au chargement de la page et ce n'est pas ce que l'on veut, et pour éviter cela on va faire une arrow function ici.

```
<div>
    <p>Function State: {counter}</p>
    <button onClick={() => setCounter(prevCounter => prevCounter + 1)}>
</div>
```

Aussi simplement que cela. Maintenant on arrive à avoir la function setCounter on va prendre le compteur précédent et l'incrémenter de 1. Vous pouvez aller tester. Récapitulons un peu, donc pour pouvoir utiliser un state dans React précédemment on devait partir sur un composant de type classe, désormais grâce au hooks, on peut avoir un state dans un composant de type fonction, pour cela on va devoir utiliser le useState qui nous permet d'avoir plusieurs éléments, le premier c'est la valeur que l'on passe au niveau du state, et dans notre cas un deuxième qui est notre fonction (setCounter) qui va nous permettre d'intéragir avec cette valeur là et de la modifier.

Pour la suite vous allez devoir installer un package UUID version 3 pour générer des id, on va donc l'importer comme ceci **import { v4 as uuidv4 } from 'uuid';** et on l'installe avec npm install uuid. Précédemment on va vu les hooks et le useState on va essayer d'aller un peu plus loin, (soit vous changer le nom de App.js précédent sois vous effacer tout ce qu'on a vu si vous changer le nom vous allez devoir recréer un App.js ce que je vous conseille si vous souhaitez garder la partir précédente). On va réaliser notre première appli plus complète un to do list (c'est la base). Au lieu de passer un number dans notre state on va passer un array, dans lequel on va avoir plusieurs objets. Nous allons également passer une deuxième state, et cette fois ci se serai un boolean. C'est parti.

Components => fichier Todo.js et on va partir sur un composant de type fonction.

On importe sur le App.js, et on l'affiche dans la div. Créons le state dans le todo, et on va lui passer ici un array, dans lequel on aura plusieurs objets le premier sera on va dire la première tâche de la todo list, on va leur attribuer un id et donc le todo, puis créer les paramètres dont le setter.

Appelons ça dans un h1,

Pour plus de simplicité j'ai installé la dépendances Bootstrap, on va styliser un petit peu.

On va maintenant faire en sorte d'afficher tous les éléments, on va faire une liste désordonnée, et les li on va les récupérer de notre tableau. On va donc utiliser la méthode map pour appeler les todos du tableau, et la méthode map dans un tableau a besoin d'accéder a l'id, on va utiliser la key pour accéder aux id. On va mettre tout ça dans une variable.

```
const myTodos = todos.map( todo => {
    return (
        <li className="list-group-item" key={todo.id}>{todo.todo}</li>
    )
})
```

De manière à l'afficher dans notre ul.

```
<ul>
    {myTodos}
</ul>
```

On enregistre et on va voir. Le bord touche donc je vais régler ça dans le container. (div) Sur App.js jemodifie le style de ma div, je me sert donc des classe bootstrap.

```
function App() {
  return (
    <div className="container">
      <Todo />
    </div>
  );
}
```

On va ajouter un bouton qui au click a la validation on va faire que quand on rajoute un todo, il s'ajoute à la liste. On va donc créer un nouveau fichier, components => AddTodoForm.js donc un formulaire qui ajoute notre todo. Ce sera un composant de type fonction,

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

const AddTodoForm = () => {
    return(
        <div>
            Form
        </div>
    )
}

export default AddTodoForm
```

Et on l'importe comme toujours, on le met sous la liste. Il s'affiche on peut le créer alors. Un form, avec un label, et deux inputs et styliser avec Bootstrap.

```
const AddTodoForm = () => {
    return(
        <form>
            <div className="card card-body">
                <div className="form-group">
                    <label>Ajouter Todo</label>
                    <input className="form-control" type="text" />
                    <input className="btn btn-success" type="submit" />
                </div>
            </div>

        </form>
    )
}
```

Réglons les détails d'affichage,

```
<input className="btn btn-success mt-4" type="submit" />
```

Pour le bouton, et pour le bloc de l'ul, et le form

```
<ul className="list-group">
    {myTodos}
</ul>
```

```
<form className="mt-4">
```

Voilà c'est plus propre déjà. Voilà maintenant on va faire en sorte que lorsque l'on ajoute un nouveau todo cela s'ajoute dans notre liste. Comment faire ????

On va créer un évènement onChange, sur notre input, son rôle sera de prendre la valeur que nous avons au niveau de cet input et de la mettre dans un state local a ce composant AddTodoForm. Essayer de créer cela avec un array. A vous de jouer.

```
const AddTodoForm = () => {

    const [ addTodo, setAddTodo ] = useState('')

    console.log(addTodo);
```

```
<input className="form-control" value={addTodo} type="text" onChange={(e) => }/>
```

```
form-control" value={addTodo} type="text" onChange={(e) => setAddTodo(e.target.value)}/>
```

En gros quand on tape qlq chose dans cet input-là, nous allons prendre son target value(la valeur qu'il y a dans ce champ-là) et nous allons la passer dans la fonction setAddTodo qui nous permet de modifier ce que nous avons initialement mis au niveau du useState, donc la chaine de caractère vide (addTodo). Maintenant l'objectif comme on a récupérer la valeur et qu'elle est au niveau de addTodo, on a mis à jour notre state via la méthode setAddTodo, et bien on va devoir prendre cette valeur là(addTodo), la passer dans le fichier Todo.js afin de mettre à jour la liste (donc l'array qui contient le state todos). On va donc créer une fonction qui va nous permettre d'appeler le setState. On va passer deux paramètres, le tableau, et un objet qui contiendra l'id et le todo.

```
const addNewTodo = (newTodo) => {
    setTodos([...todos, {
        id: ,
        todo: newTodo
      }])
```

On va gérer la dépendance qui génèrera les id automatiquement (celle installé au début uuid). Cest ici que l'on importe la dépendance, et on ajoute ceci au niveau de l'id,

```
setTodos([...todos, {
    id: newid,
    todo: newTodo
}])
```

```
const addNewTodo = (newTodo) => {
    const newid = uuidv4();
```

Dons on a une fonction ou on passe des datas. Maintenant on va prendre cette fonction, et la passer comme props au niveau du composant AddTodoForm afin qu'on puisse le récupérer. Ce composant sera dans le composant Todojs

```
<AddTodoForm addNewTodo={addNewTodo}/>
```

Maintenant qu'on a la props(addNewTodo mais on peut l'appeler diféremment) qui contient la fonction addNewTodo, on va pouvoir le récupérer.

```
const AddTodoForm = ({addNewTodo}) => {

    const [ addTodo, setAddTodo ] = useState('')

    addNewTodo(addTodo)
```

Là ça ne fonctionnera pas, il faudrait qu'on fasse en sorte d'appeler cette fonction là une fois que nous avons validé le formulaire.
Maintenant il faut faire en sorte que l'utilisateur click sur le bouton pour envoyer l'information, pour faire cela, on se rend dans le form et on lui met l'évènement onSubmit au lieu de le faire sur le bouton, on le fait sur le form, ainsi si on click sur le bouton, la data va être envoyé, et lui attribué la méthode handleTodo, et la créer.

```
const handleTodo = (e) => {
    e.preventDefault()
    addNewTodo(addTodo)
}
```

On récap un peu, on passe la valeur de l'input via le setAddTodo, qui va nous permettre de mettre à jour le state, une fois mis à jour et que l'on a validé notre formulaire à ce moment-là, on passe la valeur en question dans la fonction addTodo. Ainsi on passe la valeur dans le newTodo, et lui on va l'avoir dans notre deuxième objet(todo). Faisons un test, et tout devrais fonctionner. Actuellement si on clique sur l'input vide ça rajoute une ligne vide et nous on ne veut pas cela. C'est très très simple, on va juste appeler la fonction setAddTodo qui nous permet d'interagir avec le state, on va simplement lui attribuer une chaine vide après déclaration.

```
const handleTodo = (e) => {
    e.preventDefault()
    addNewTodo(a setAddTodo(val
                    void
    setAddTodo('')
}
```

Maintenant on faire en sorte que si l'input est vide et que l'on clique et bien on aura un petit message d'erreur. Allons sur Todo.js, et avec les hooks on peut non seulement avoir un state dans un composant de type fonction, mais on peut également en avoir plusieurs. Donc on va créer ce nouveau state, pour cela on va devoir faire appel à notre fonction useState , et cette fois ci on lui passe un booléen, on va également obtenir un tableau qui aura deux valeurs

aussi, la première c'est celle que l'on a indiqué(false) mais on va l'appeler différemment, je choisi warning, et la seconde ce sera le setter donc setWarning.

```
const [warning, setWarning] = useState(false);
```

On descend au niveau de notre fameuse fonction qui permet d'appeler le setTodo, et bien on va faire en sorte que si cela renvoie une chaine de caractère vide et bien ça ne le prenne pas en compte. On va gérer cela en créant une condition (if else) dans notre méthode addNewTodo.

```
const addNewTodo = (newTodo) => {
    const newid = uuidv4();
    if(newTodo !== '') {

        warning && setWarning(false)

        setTodos([...todos, {
            id: newid,
            todo: newTodo
        }])
        console.log(newid)
    } else {
        setWarning(true);
    }
}
```

On ne va pas le retourner directement dans le Jsx, on va faire en sorte de l'afficher seulement si ce warning retourne true.

```
    } else {
        setWarning(true);
    }
}

warning && <div class="alert alert-danger" role="alert">Veuillez indiquer un Todo</div>
```

Si c'est true tu m'affiche ça, mais on va le mettre dans une variable,

```
const warningMsg = warning && <div className="alert alert-danger" role="alert">
```

Donc prend ce warning la et on l'affiche, juste avant le h1.

```
return (
    <div>
        {warningMsg}
        <h1 className="text-center"> {todos.length} To-do</h1>
```

Allons tester cela. Tout fonctionne bien sauf, si l'on retape qlq chose et bien cela s'ajoute mais le message de warning reste actif pourquoi ??
Parceque dans le else nous l'avons défini en true et qu'il est maintenant dans l'autre objet,

```
const [warning, setWarning] = useState(false);
```

On va faire en sorte que lorsqu'on passe une chaîne de caractère différente d'une chaine vide à ce moment-là on va faire en sorte de réinitialiser notre objet et de le mettre en false.

```
const addNewTodo = (newTodo) => {

    if(newTodo !== '') {

        warning && setWarning(false)

        setTodos([...todos, {
            id: uuid(),
            todo: newTodo
        }])
    } else {
        setWarning(true);
    }
```

C’est aussi simple que ça. Allons tester cela, tout fonctionne parfaitement.

Hook d’effet le useEffect

On peut aller sur la doc et voir la partie sur les hooks, et on choisit useEffect. Cela nous permet d’avoir qlq chose que l’on pouvait faire avant qu’avec les méthodes de cycle de vie, c’est-à-dire on partait obligatoirement sur un composant de type classe afin de pouvoir accéder à ces méthodes. C’est comme une combinaison de componentDidMount, componentDidUpdate et componentWillUnmount. Rendons-nous sur components => et créons un file ClassCount.js, on va partir sur un composant de type classe on peut utiliser le raccourci **rce** pour créer toute la structure, et on l’importe dans notre App.js et on l’affiche comme toujours dans la div. Créons la méthode componentDidMount,

```
class ClassCount extends Component {


    componentDidMount() {
        console.log('Je suis dans CDM');
    }
```

Créons un compteur, que l’on va afficher a la place du title(titre dans l’onglet celui qui est dans index.html) lors du montage

```
class ClassCount extends Component {

    constructor(props) {
        super(props)

        this.state = {
            count: 0
        }
    }
```

Très bien affichons cela dans un h1,

```
<div>
    <h1>{this.state.count}</h1>
</div>
```

Il s’affiche bien, maintenant on va l’afficher en title au moment du montage,

```
componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`
}


render() {
```

Allez vérifier et vous verrez le title qui a changé déjà. On va passer a la deuxième méthode.

```
componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`
}

componentDidUpdate(prevProps, prevState) {
    document.title = `Vous avez cliqué ${this.state.count} fois`
}
```

Cette méthode s'enclenche seulement si on modifie un props ou un state. On va faire en sorte de modifier ce state, on va par exemple incrémenter le compteur,

```
div>
   <h1>{this.state.count}</h1>
   <button onClick={() => this.setState({count :this.state.count + 1})
/div>            @fa5-rotate-180  Rotates icon by 180 degrees…
```

On save et on va voir, au click cela doit modifier notre state donc notre title, vous pouvez vérifier dans l'inspecteur le title a bien changé. On a bien créer les deux méthodes, mais c'est une répétition que nous allons, grâce aux hooks, dans un composant de type fonction éviter. On va donc créer un composant de type fonction, et on va reproduire exactement ce scénario-là. Components => créons le file FunctionCount.js on peut utiliser le raccourci rfce pour créer la structure de la fonction, on exporte et on importe dans App.js et effaçant le précédent, ou le commentant.

```
function App() {
  return (
    <div className="App">
      {/*<ClassCount />*/}
      <FunctionCount />
    </div>
  );
}
```

Maintenant que l'on sait comment créer un state dans un composant de type fonction on le crée, à vous de jouer et on l'affiche.

```
function FunctionCount() {

    const [count, setCount] = useState(0)

    return (
        <div>
            <h1>{count}</h1>
```

Il s'affiche, il nous reste à gérer l'incrémentation par un click au bouton, à vous de jouer.

```
return (
    <div>
        <h1>{count}</h1>
        <button onClick={() => setCount(count + 1)}>Cliquer</button>
    </div>
)
```

Vérifions que cela fonctionne, comment je vais faire maintenant pour pouvoir modifier le titre au moment du montage du composant. Et bien en utilisant le useEffect, (toujours dans le FunctionCount.js)

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

Cette fonction s'enclenche à chaque render, que se soit au moment du montage, de la mise à jour ou du démontage.

```
function FunctionCount() {

    const [count, setCount] = useState(0)

    useEffect(() => {
        document.title = `Vous avez cliqué ${count} fois`
    })

    return (
        <div>
            <h1>{count}</h1>
            <button onClick={() => setCount(count + 1)}>Cliquer</button>
        </div>
```

Allons vérifier, et le title s'est bien mis au moment du montage, pour mieux se rendre compte de cela, on va faire un setTimeOut, avec fonction fléchée qui va se déclencher après 5sec. (ca va nous permettre de voir le title avant sa modification)

```
useEffect(() => {

    setTimeout(() => {
        document.title = `Vous avez cliqué ${count} fois`
    }, 5000)
})
```

Enregistrons et allons voir la différence, regarder le title, et attendais 5sec, il va changer. Donc désormais vous avez compris que via useEffect, on peut faire qlq chose qu'on pouvait faire avant avec la méthode componentDidMount. On doit maintenant faire la même chose que le componentDidUpdate, c'est-à-dire nous allons cliquer sur le bouton pour incrémenter le state et on va voir si on aura une modification au niveau de ce titre-là. Donc on voit bien qu'avec une seule fonction on a pu faire ce qu'on pouvait faire dans un composant de type class avec deux méthodes.

Hook useEffect avec condition

Précédemment on a appris à utiliser le useEffect et on a compris que celle-ci s'exécute a chaque render ce qui pourrait dans certains cas causer qlq soucis de performances. On va reprendre sur le composant de type class(ClassCount.js) au niveau du render on a un bouton avec un évènement onclick, qui change la valeur du state. Pour monter cela on va créer un input de type text qui nous permet d'ajouter un name dans notre state, mais avant cela on crée d'abord une nouvelle propriété name dans notre state que l'on initialise en chaine vide.

```
class ClassCount extends Component {

    constructor(props) {
        super(props)

        this.state = {
            count: 0,
            name: ''
        }
    }
```

Créons l'input sous le bouton,

```
<input type="text" value={this.state.name} onChange={e => {
    this.setState({
        name: e.target.value
    })
}}/>
```

Si vous faites un console log, et que vous vérifiez dans l'inspecteur vous remarquerez que la méthode continue alors que nous écrivons dans l'input hors on incrémente plus le count, cela n'est pas nécessaire et cause un manque d'optimisation de notre appli, pour remédier à cela, nous allons donc devoir vérifier si la valeur du count a changer au moment d'invoquer la méthode componentDidUpdate, et on peut faire cela via une simple condition. Dans le corps de cette méthode, on fait la condition,

```
componentDidUpdate(prevProps, prevState) {

    if(this.state.count !== prevState.count) {
        console.log('Mise à jour du titre');
        document.title = `Vous avez cliqué ${this.state.count} fois`
    }

}
```

Désormais si l'on incrémente cela fonctionne et si l'on écrit dans l'input elle ne s'enclenche plus. Maintenant la question qui vient c'est comment reproduire cela dans un composant de type fonction grâce à useEffect??

Rassurez vous avec les useEffect c'est encore plus simple, on met donc le composant ClassCount en commentaire, et on se concentre sur le FunctionCount. On se rend dessus, et on crée un second state avec une chaine de caractère vide,

```
const [name, setName] = useState('')
```

On crée l'input de type text, et via l'évènement onChange on invoque la fonction setName en passant évidemment par une arrow Function à laquelle on passe un event object qui nous servira à accéder à target Value qui sera par la suite injectée dans notre hookState,

```
<h1>{count}</h1>
<input type="text" value={name} onChange={e => setName(e.target.value)} />
<button onClick={() => setCount(count + 1)}>Cliquer</button>
```

Avant de tester cela on va ajouter un console log au niveau de la fonction useEffect qui affichera un message.

```
useEffect(() => {
    console.log('Mise à jour du titre via useEffect');
    document.title = `Vous avez cliqué ${count} fois`
})
```

Ainsi on pourra suivre les exécutions de celles-ci dans la console, on test, tout, cependant quand on tape qlq chose dans l'input on peut constater que la fonction useEffect est tjrs appelée alors que l'on apporte aucun changement au count. Comme dans l'exemple avec la classe, on a un problème d'optimisation que l'on va résoudre. Dans la class on a résolu cela avec une condition, et bien on va faire plus ou moins la même chose. Dans le useEffect on aura besoin d'un second paramètre et ce paramètre sera un array dans lequel on va indiquer un state ou le props qu'on va devoir vérifier avant d'exécuter la fonction.

```
useEffect(() => {
    console.log('Mise à jour du titre via useEffect');
    document.title = `Vous avez cliqué ${count} fois`
}, [count])
```

On save et on vérifie cela. Et voilà tout est réglé, comme cela vérifie que le count na pas bougé cela ne lance plus la fonction inutilement si le count n'est pas modifié.
Vous avez désormais compris comment contrôler l'exécution de la fonction useEffect dans React.

Hook useEffect comme ComponentDidMount

Précédemment on a appris à passer un 2ème paramètre à la fonction, (on avait passé l'array count). Maintenant la question qu'on peut se poser c'est comment faire pour que la fonction useEffect s'exécute une seule et unique fois ? On va reproduire qlq chose que nous avons dans les class(componentDidMount), qui s'exécute une seule et unique fois. On va se débarrasser de la dite méthode dans le file ClassCount, on va garder uniquement componentDidMount.

```
componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`
}
```

Retournons dans le App.js et on décommente le composant ClassCount, et l'on commente l'autre composant, faites un test. Re commentons le ClassCount et décommentons FunctionCount, allons sur le file FunctionCount.js, dans l'exemple précédent on lui a dit tu t'exécute seulement si on apporte un count, si on souhaite qu'elle ne s'exécute une seule et unique fois à ce moment-là, au lieu d'afficher qlq chose au niveau du array, nous allons faire un array vide. De cette manière-là, quand la fonction s'enclenche au moment du chargement de notre composant, elle nous affichera le console log.

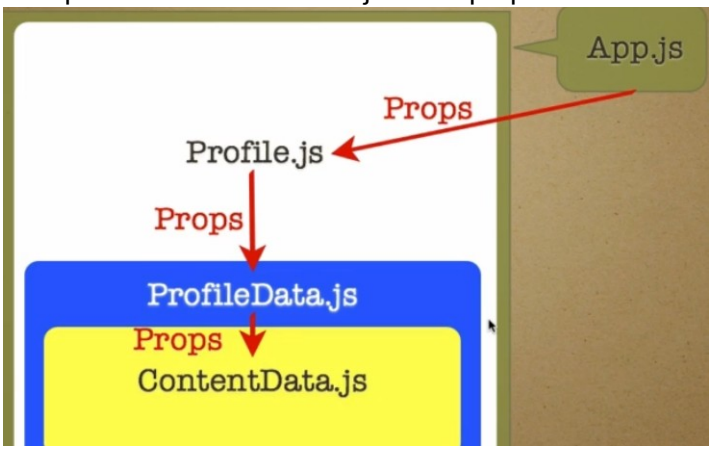
```
useEffect(() => {
    console.log('Mise à jour du titre via useEffect');
    document.title = `Vous avez cliqué ${count} fois`
}, [])
```

Il affichera bien le title vous avez cliqué zéro fois, maintenant nous allons incrémenter ce zéro là, c'est-à-dire ce count et voir si elle va s'enclencher. On va cliquer et voir si le title s'incrémente, le count s'incrémente mais pas le title. Ainsi en mettant un array vide comme deuxième paramètre à la fonction useEffect nous avons permis à celle-ci de s'exécuter une seule et unique fois c'est-à-dire au moment du montage. Dons rien qu'en mettant un array vide on à réussi à reproduire exactement le même comportement que la méthode componentDidMount de la Class.

Hook de contexte useContext

Le contexte offre un moyen de faire passer des données à travers l'arborescence du composant sans avoir à passer manuellement les props à chaque niveau. (doc react)

Le but ici serai de passer une information que nous avons au niveau de App.js, afin de l'afficher au niveau de ContentData.js. Dans le cas présent, on a pris l'information que nous avons au niveau de App.js via un props, nous l'avons récupéré au niveau de Profil.js, nous l'avons passé via le props vers ProfilData.js avant de la repasser vers ContentData.js via les props.



On a remarqué que cette technique est assez fastidieuse, quand il s'agit d'avoir juste un seul nasted Components(composant imbriqué), cela ne pose pas de problème. En revanche, dés lors que l'on a plusieurs composants qui sont imbriqués les uns dans les autres, cela va devenir impossible à gérer. La solution que React à apporté ce sont les fameux context. En effet, grâce à Context, on peut désormais récupérer la data directement dans le composant dans lequel on souhaite la consommer.

Exercice 10

Prenons cette application, et commençons, nous allons prendre une info que nous avons dans App.js, et de l'afficher directement dans ContentData.js. Voici ContentData.js,

```
import React from 'react'

const ContentData = () => {
    return(
        <div>
            <ul>
                <li> Nom: </li>
                <li> Age: </li>
            </ul>
        </div>
    )
}

export default ContentData
```

Nous avons ici un data qui nous retourne une div dans laquelle on a une ul, on a exporte et importé dans ProfileData.js, (le voici)

```
import React from 'react'
import ContentData from './ContentData'

const ProfileData = () => {

    return (
        <div>
            <ContentData/>
        </div>
    )
}

export default ProfileData
```

Qui lui-même est importé dans le Profile.js(le voici)

```
import React, { Component } from 'react'
import ProfileData from './ProfileData'

class Profile extends Component {

    render() {

        return <ProfileData />
    }
}

export default Profile;
```

Pour finir nous avons exporté ce file au niveau de App.js (le voici)

```
class App extends React.Component {

  state = {
    user: {
      name: 'Lisa',
      age: 8
    }
  }

  render() {

    return (
          <Profile />
    );
  }
}
```

Ici nous avons un composant de type class, qui contient un state, dans lequel nous avons l'objet user, si nous allons sur le navigateur nous aurons bien les données qui sont dans le file ContentData.js. Le state qui est dans le App.js nous ne l'avons pas encore utilisé, donc l'objectif est de prendre ses infos et de les afficher au niveau de ContentData.js, donc au lieu de passer par les props on va utiliser nos context. Donc MyContext.js le voici,

```
import React from 'react'

export const UserContext = React.createContext();
```

Et on importe dans le App.js,

```
import {UserContext} from './components/MyContext'
import './App.css';
```

Maintenant qu'on l'a importé, il va nous permettre d'accéder à son provider qu'on va utiliser ici pour effectuer le transfert de l'information vers les composants enfants. Pour cela je vais déclarer le UserContext provider et mettre le composant profile à l'intérieur comme nasted components. C'est-à-dire on va appliquée ce transfert de data a profil ainsi qu'à tous ces enfants. Pour cela l'information on va la définir via value et nous allons donc passer comme info l'objet 'user'.

```
  render() {

    return (
      <UserContext.Provider value={this.state.user}>
        <Profile />
      </UserContext.Provider>
    );
  }
}

export default App;
```

On save et on se rend dans ContentData, parce que c'est dans celui-ci que nous souhaitons consommer cette info. On importe a nouveau userContext, on va redéclarer dans le return le UserContext consumer, et nous allons déclarer une fonction. On lui passe comme param user, et petit console.log pour voir ce que nous avons dans ce dernier.

```
const ContentData = () => {
    return(
        <UserContext.Consumer>
        {
            user => {
                console.log(user);
            }
        }
        </UserContext.Consumer>
        <div>
            <ul>
                <li> Nom: </li>
```

Possible erreur avec la div du dessous pour le console.log

Maintenant que l'on a bien les infos on a plus qu'a l'afficher dans la fonction, (la div qui crée l'erreur) la div qui contient les ul, li. Vu que l'on a accès à l'objet on a plus qu'a afficher le name et l'âge.

```
const ContentData = () => {
    return(
        <UserContext.Consumer>
        {
            user => {
                return (
                    <div>
                        <ul>
                            <li> Nom: {user.name}</li>
                            <li> Age: {user.age}</li>
                        </ul>
                    </div>
                )
            }
```

On save et on vérifie. Tout fonctionne parfaitement, on sait également que dans un composant de type fonction on peut consommer plusieurs context. Pour illustrer cela allons donc notre file MyContext.js et créons un nouveau context,

```
import React from 'react'

export const UserContext = React.createContext();
export const ColorContext = React.createContext()
```

Comme son frère on l'importe dans App.js,

```
import {UserContext, ColorContext} from './components/MyContext'
```

Et on le met dans le return,

```
render() {

  return (
    <UserContext.Provider value={this.state.user}>
      <ColorContext.Provider value={'red'}>
        <Profile />
      </ColorContext.Provider>
    </UserContext.Provider>
  );
}
```

Comme pour le précédent on retourne dans ContentData.js on importe, maintenant comment faire pour le consommer en plus de celui que l'on a déjà ???

Et bien on se rend dans le return, et comme son précédent on va le déclarer et passer une arrow function également. Il va se situé ici.

```
const ContentData = () => {
    return(
        <UserContext.Consumer>
        {
            user => {
                return (
                    <ColorContext.Consumer>
                        {

                        }
                    </ColorContext.Consumer>
```

```
return (
    <ColorContext.Consumer>
        {
            color => {
                return (
                    <div style={{color: color}}>
                        <ul>
                            <li> Nom: {user.name}</li>
                            <li> Age: {user.age}</li>
                        </ul>
                    </div>
                )
            }
        }
    </ColorContext.Consumer>
```

Vous pouvez aller vérifier sur le navigateur vos li sont rouge donc tout fonctionne très bien. Sur ContentData.js nous avons donc deux context mais imaginez que l'on en ai plusieurs cela peut vite devenir ingérable et difficile à lire. Nous avons donc la première fonction qui nous retourne un autre consumer dans lequel nous avons une autre fonction qui nous retourne du code. La solution existe avec React Hook(pensez a avant comment il devait faire), notamment grâce a la fonction useContext. On va donc l'importer dans React, et la déclarer,

```
import React, {useContext} from 'react'
import {UserContext, ColorContext} from './MyContext'

const ContentData = () => {

    const user = useContext(UserContext)
    const color = useContext(ColorContext)
```

Une fois que j'ai obtenu ces deux valeurs, je n'ai pas besoin d'utiliser mes fameuses arrow function vous allez voir la différence.

```
const ContentData = () => {

    const user = useContext(UserContext)
    const color = useContext(ColorContext)

    return(
        <div style={{color: color}}>
            <ul>
                <li> Nom: {user.name}</li>
                <li> Age: {user.age}</li>
            </ul>
        </div>
    )
}
```

On save et on va vérifier, mais tout sera parfait.

UseReducer en React

Pour commencer on peut se rendre sur la doc => hooks et on clique sur référence de l'api des hooks, on descend légèrement on verra les hooks que l'on a vu (ceux de base), et les hooks suivants dont useReducer(). On peut cliquer dessus et regarder cela, il nous indique que useReducer() est une alternative a useState(), ils nous disent également qu'elle accepte un reducer, on a une ici une

fonction a laquelle ils ont passe deux paramètres un state et une action, et ca leur retourne un newState. Un peu comme la méthode reduce en javascript.

Rappel de la methode reduce :

Imaginons qu'on ai un array appelé myArray, on qu'on lui applique le méthode reduce avec deux paramètres le reducer et un nombre, le reducer lui c'est une fonction a qui on donnera deux paramètres, l'accumulateur, et la valeur courante que l'on a dans le tableau. Et ca va nous retourner la valeur que nous avons dans notre accumulateur et que l'on va additionner par exemple avec la valeur courante du tableau, dans cet exemple la cela retournerai quoi ??? voici le code.

```
const myArray = [1, 2, 3, 4]

const reducer = (accum, curval) => accum + curval

myArray.reduce(reducer, 5)
```

Et bien cela donnera 5 + (1+2+3+4) donc 10 = 15, cela donnera 15. Nous avons une ligne de la methode useReducer

```
const [state, dispatch] = useReducer(reducer, initialArg);
```

On a donc la methode useReducer a laquelle il passe en premier parametre un reducer, (fonction qui a un state et une action en param et qui retourne un newState.), ensuite en second on a l'etat initial. Cette methode comme dit la doc nous renvoie l'etat actuel (le state et la fonction dispatch), vous inquiétez pas ce n'est que de la théorie on va passer bientôt aux exercices.

On va faire quelque chose qui ressemble à ce qu'il y a dans la doc.

Exercice 11

Correction

App.js

```
class App extends React.Component {

  render() {

    return (
      <div className="text-center">
        test
      </div>
    );
  }
}
```

Count.js

```
import React, {useReducer} from 'react'


const initialState = 0;

const reducer = (state, action) => {

}

function Count() {

    useReducer(reducer, initialState)

    return (
        <div>
```

Avant de faire quoi que se soit, revenons sur useReducer et que dit la doc , elle nous renvoie l'état local actuel accompagné de la méthode dispatch.

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

On va ajouter cela a notre code, et c'est ces deux la justement qui vont nous permettre d'afficher un nouvel etat au niveau de la div(on le met ici dans l'exemple) dans notre cas le state c'est un count donc on met count, et on l'affiche.

```
function Count() {

    const [count, dispatch] = useReducer(reducer, initialState)

    return (
        <div>
            <h1>{count}</h1>
        </div>
```

Ensuite on met en place les boutons qui vont incrémenter ou décrémenter le compteur,

```
<h1>{count}</h1>
<button className="btn btn-success m-3">+</button>
<button className="btn btn-danger m-3">-</button>
```

Maintenant on doit les faire fonctionner **A vous de jouer**.
Et bien on ajoute un evenement on click et on declare la methode dispatch en arrow function avec parametre la chaine increment ou decrement.

```
<h1>{count}</h1>
<button className="btn btn-success m-3" onClick={() => dispatch('inrement')}>+</button>
<button className="btn btn-danger m-3" onClick={() => dispatch('decrement')}>-</button>
```

Désormais nous allons devoir accéder à la valeur que l'on met comme paramètre (increment' decrement) au niveau de l'action, qui sera dans la fonction du reducer,

```
import React, {useReducer} from 'react'


const initialState = 0;

const reducer = (state, action) => {

}

function Count() {

    const [count, dispatch] = useReducer(reducer, initialState)

    return (
        <div>
            <h1>{count}</h1>
            <button className="btn btn-success m-3" onClick={() => dispatch('inrement')}>+</
            <button className="btn btn-danger m-3" onClick={() => dispatch('decrement')}>-<
```

et pour cela on va partir sur un switch on va mettre l'action donc, c'est a dire la valeur qui passe dans les dispatch, et le premier cas on va dire increment, si on est dans ce cas la on va retourner l'état actuel qu'on incrémente de 1, le deuxième cas et bien c'est pareil juste on decremente de 1, et on va mettre un default qui lui renverra simplement l'etat actuel.

```
const initialState = 0;

const reducer = (state, action) => {
   switch(action) {
       case 'inrement':
           return state + 1
       case 'decrement':
           return state - 1

       default:
           return state
   }
```

Petit résumé : on a crée 2 boutons avec des évènement au click, on appelle la méthode dispatch qu'on a retourné de notre useReducer qui nous permet justement d'accéder a l'action pour pouvoir effectuer soi une incrémentation soi une décrémentation. On prend donc l'état actuel et on lui rajoute 1 ou on enlève 1. On peut ajouter un bouton pour réinitialiser le compteur et donc ajouter le cas.

```
<button className="btn btn-success m-3" onClick={() => dispatch('inrement')}>+</button>
<button className="btn btn-danger m-3" onClick={() => dispatch('decrement')}>-</button>
<button className="btn btn-danger m-3" onClick={() => dispatch('reinitialiser')}>-</button>
```

```
case 'inrement':
    return state + 1
case 'decrement':
    return state - 1
case 'reinitialiser':
    return initialState

default:
    return state
```

On save et on va vérifier tout ça. On va juste changer la classe du bouton réinitialiser qui est en danger, en primary. Voila pour useReducer, on a vu comment l'utiliser mais rappelez vous que c'est une alternative au useState(qui doit vous sember plus abordable), mais au moins vous la connaissez.

UseCallback en React

Ici on va faire une petite révision des concepts déjà étudiés jusqu'ici tout en réalisant notre appli sur laquelle on utilisera le hook useCallback. On peut déjà aller sur la doc pour voir comment cela s'utilise et en quoi cela consiste. On va créer une appli qui lors du click sur le bouton cela remplira notre barre de progression.

On a donc notre App.js, et notre dossier components dans lequel nous aurons deux fichiers, le Button.js, et le Count.js. Dans le Button.js nous aurons une fonction qui retourne un bouton,

```
import React from 'react'

function Button() {
    return <button>+ %</button>
}

export default Button
```

On l'importe et on l'affiche deux fois dans le App,

```
<Button >Count 1</Button>
<Button >Count 2</Button>
```

Au niveau de Count.js nous avons une fonction qui retourne un p, ainsi qu'une div qui contiendra la barre de progression grâce au classe bootstrap.

```
return (
    <>
        <p className="h1">%</p>

        <div className="progress">
            <div className="progress-bar progress-bar-striped" role="progressbar"
                style={{width: '0%'}}>
            </div>
        </div>
    </>
)
```

On l'importe et on l'affiche deux fois également.

C'est ce qui nous donne l'affichage vu précédemment, donc on crée nos deux variables d'état, que l'on va par la suite essayer d'incrémenter et de gérer la progression de ces 2 barres en fonction du bouton sur lequel on click. On va créer le state dans le App.js, et en paramètre du useState on va passer un objet dans lequel on aura la valeur initial de notre barre qui sera a 0, on va définir la couleur du bouton,

```
function App() {

    const [state, setstate] = useState({value: 0, btnColor: 'success', })
```

Il nous reste à ajouter la valeur de incrément on va dire 25,

```
function App() {

    const [state, setstate] = useState({value: 0, btnColor: 'success', increment: 25});
```

Maintenant que l'on a tout définit on va changer son nom, on va encore partir sur count, et son setter,

```
const [countOne, setCountOne] = useState({value:
```

On peut recopier tout ça pour le deuxième état, on appellera countTwo, et la valeur d'incrément sera de 20, et la classe sera danger.

```
const [countOne, setCountOne] = useState({value: 0, btnColor: 'success', increment: 25})
const [countTwo, setCountTwo] = useState({value: 0, btnColor: 'danger', increment: 20})
```

Maintenant que c'est fait on va passer ces valeurs la en tant que props à nos compteurs.

```
<Count count={countOne.value} bgColor={countOne.btnColor} />
<Count count={countTwo.value} bgColor={countTwo.btnColor} />
```

On save, et on part sur Count.js, vu que l'on a passe les props count et background color, on peut les récupérer en effectuant le destructuring au niveau de la fonction count. Et on affiche, la valeur de count au h1, au niveau de la div progress, on va créer la progression.

```
import React from 'react'

const Count = ({count, bgColor}) => {

    const progress = {width: `${count}%`};

    return (
        <>
            <p className="h1">{count} %</p>

            <div className="progress">
                <div className="progress-bar progress-bar-striped" role="progressbar"
                    style={progress}>
                </div>
            </div>
```

Allons voir la différence a l'affichage, nous avons maintenant à régler la partie de la progression au click sur les boutons. Retournons sur App.js, première chose, nous créons l'objet et la props de la couleur, ainsi que pour l'incrément, et le children(Count 1)

```
<Button btnColor={countOne.btnColor} increment={countOne.increment}>Count 1</Button>
<Button >Count 2</Button>
```

Copions sur le second bouton avec countTwo, passons au fichier Button.js, et importons les props en paramètre de la fonction, donc ensuite on va styliser les boutons et passer la valeur de increment,

```
import React from 'react'

function Button({btnColor, increment, children}) {
    return <button className={`btn btn-${btnColor}`}>+ %</button>
}

export default Button
```

Retournons au niveau de App.js et créons les méthode increment et decrement, on va partir sur une arrow function appele incrementCountOne, sachant que l'objectif c'est d'incrémenter jusqu'à 100%. On prend l'objet countOne et on accède a la value, on créera ensuite un handleCLick sur le bouton ou on passera la fonction incrementCountOne

```
const [countOne, setCountOne] = useState({value: 0, btnColor: 'success', incremen
const [countTwo, setCountTwo] = useState({value: 0, btnColor: 'danger', increment

const incrementCountOne = () => {
  countOne.value < 100 && setCountOne({...countOne, value: countOne.value + ...})
}

return (
  <div className="container">

    <Count count={countOne.value} bgColor={countOne.btnColor} />
    <Count count={countTwo.value} bgColor={countTwo.btnColor} />
```

Et le bouton,

```
return (
  <div className="container">

    <Count count={countOne.value} bgColor={countOne.btnColor} />
    <Count count={countTwo.value} bgColor={countTwo.btnColor} />

    <Button handleClick={incrementCountOne} btnColor={countOne.btnColor} increment={co
    <Button btnColor={countTwo.btnColor} increment={countTwo.increment}>Count 2</Butto

  </div>
);
```

Pensez a l'importer dans le Button.js, en tant que props (la methode handleClick),

```
function Button({handleClick, btnColor, increment, children}) {
    return <button className={`btn btn-${btnColor}`}>+ %</button>
}
```

A partir du moment où je peux accéder a cela et bien je vais m'en servir dans les boutons,

```
function Button({handleClick, btnColor, increment, children}) {
    return <button onClick={handleClick(increment)} className={`btn btn-${btnColor}`}>
}
```

Vu que l'on a passe notre fonction de cette façon la et bien on doit le faire sous fonction fléchée, sinon cela se lancera au chargement de la page et on veut pas ça,

```
import React from 'react'

function Button({handleClick, btnC var handleClick: any ren}) {
    return <button onClick={() => handleClick(increment)} className={`btn btn-${btnColor}`}>
}

export default Button
```

Sur le App.js finalisons tout ça, la value on va la définir en paramètre (val), et s'en servir ensuite comme valeur d'incrémentation,

```
const incrementCountOne = (val) => {
  countOne.value < 100 && setCountOne({...countOne, value: countOne.value + val})
}
```

On va copier cette fonction et l'appeler incrementCountTwo, et passons la méthode dans le handleClick du second bouton.

```
return (
  <div className="container">

    <Count count={countOne.value} bgColor={countOne.btnColor} />
    <Count count={countTwo.value} bgColor={countTwo.btnColor} />

    <Button handleClick={incrementCountOne} btnColor={countOne.btnColor} increment={co
    <Button handleClick={incrementCountTwo} btnColor={countTwo.btnColor} increment={co
```

Enregistrons tout ça, et testons. Nos barres seront de couleur bleu on va faire en sorte que cela soit de la couleur du bouton tant qu'à faire. Pour cela on retourne dans count là on a notre fameuse barre, on a qu'à mettre la props qui gère la couleur, mais comme on a la classe bootstrap en chaine de caractère, on va devoir combiner avec une variable, on va passer au backtick, et insérer la variable.

```
<div className="progress">
    <div className={`progress-bar progress-bar-striped bg-${bgColor}`} role="p
        style={progress}>
    </div>
</div>
```

On save et on vérifie et tout est parfait. ON peut afficher le pourcentage dans le bouton en ajoutant la variable dans le bouton.

```
dleClick, btnColor, increment, children}) {
nClick={() => handleClick(increment)} className={`btn btn-${btnColor} mt-3`}>+ {increment} %
```

Entre-le + et le % {increment} et voilà. Là on peut se dire l'exo est terminé, oui c'est le cas mais là on a que deux composants imaginez que l'on en ai 30 ou 40 ou plus, imaginez le temp de chargement de tous ces éléments, on aura des soucis d'optimisation, et c'est à ce moment-là que l'on va régler le soucie avec useCallback.
Allons ajouter une props (text) au composant Count, que l'on utilisera dans des console.log (App.js)

```
return (
  <div className="container">

    <Count text="CountOne" count={countOne.value} bgColor={countOne.btnColor} />
    <Count text="CountTwo" count={countTwo.value} bgColor={countTwo.btnColor} />
```

Allons désormais dans le Count.js, et mettons un console.log pour afficher en pourcentage.

```
const Count = ({count, bgColor}) => {

    console.log(`Pourcentage`);

    const progress = {width: `${count}%`};

    return (
```

Evidemment on importe le props text avec les props color et bgColor,

```
const Count = ({text, count, bgColor}) => {

    console.log(`Pourcentage ${text}`);
```

On va ensuite sur le Bouton.js et on va désormais utiliser le children,

```
function Button({handleClick, btnColor, increment, children}) {

    console.log(children);
```

On save et on va sur la console, on peut voir que les deux compteru se sont charge pour les deux barre de progression, ainsi que les deux count pour les deux boutons. On peut d'ailleurs modifier légèrement le console.log,

```
console.log(`Button ${children}`);
```

A click sur le bouton il se passe quoi ???
Et bien on va ajouter un console log histoire de bien comprendre, (App.js)

```
const incrementCountOne = (val) => {
  console.log('je suis dans incrementCountOne');
  countOne.value < 100 && setCountOne({...countOne, value: countOne.value + val})
}
```

Et on fait pareil dans le second, cliquons désormais sur le bouton vert et regardons ce qui se passe en console. Et bien tout les composants se recharge on ne veut pas ca nous, on va régler cela avec la methode memo de Js, comme ceci dans l'exporte du count,

```
export default React.memo(Count)
```

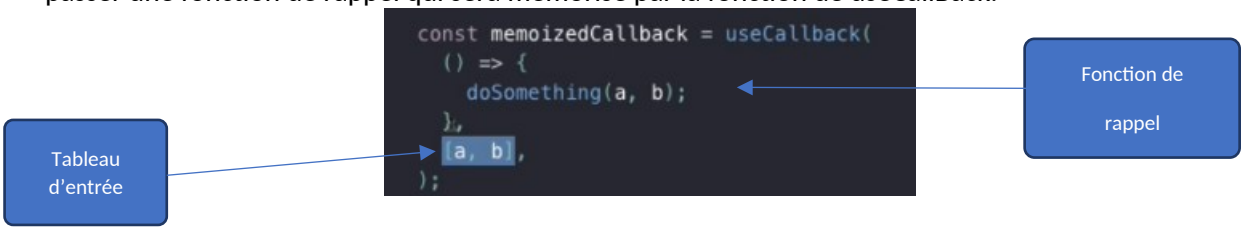
Il est possible que vous deviez changer le <> </> en div dans le return en cas d'erreur, normalement il ne les reconnait pas. Et on va appliquer exactement la même chose au composant Button.js

```
export default React.memo(Button)
```

On save et on recommence, on recharge la page et on regarde en console et bien on a pas recharger le second pourcentage la méthode memo a bien fait son job. En revanche on a recharger les deux boutons et nous on aimerai que celui sur lequel on clique non ?
Et bien la le problème vient du App.js, à chaque fois que le composant(incrementCountOne) il crée une nouvelle fonction, sachant qu'on a passé celle-ci en tant que props,

```
<Button handleClick={incrementCountOne}
<Button handleClick={incrementCountTwo}
```

Et bien le bouton lui va se recharger une nouvelle fois. En d'autre terme la méthode memo que l'on a applique au Button.js n'arrive pas à gérer ces nouvelles fonctions qui se crée car elles ne se basent pas sur les égalités référentiels(ouille ca pique les yeux)permettant d'éviter les rendus superflus. En effet ce n'est pas parce que deux fonctions qui ont le même comportement seraient pour autant identiques. Dans le cas présent au moment du montage incrementCountTwo est totalement différente de celle qui sera regénérée au moment de la mise à jour du composant. Et pour remédier à cette problématique, on va devoir mettre en place un moyen qui permet de mémoriser la fonction et de la retourner identique a son état précédent. C'est-à-dire que l'on ne va pas predre en considération la mutation de celle-ci au moment de la mise à jour. Et c'est la qu'intervient le useCallBack().
Allons sur la doc de cette méthode, un useCallBack() renvoie une fonction de rappel mémorisée, c'est-à-dire que nous allons fournir une fonction de rappel, un tableau d'entrée. Qui celui-ci va lui passer une fonction de rappel qui sera memorisé par la fonction de useCallBack.



Maintenant appliquons cela, App.js et on importe le useCallBack comme le useState, et ensuite on va l'appliquer a nos deux fonctions incrementCount One et Two, on va devoir considérer ces fonctions comme les fonction de rappel qui seront mémorisé. Pour cela on va utiliser notre callBack et on va passer la fonction en tant que premier paramètre, on a dit elle sera rendu si on apporte un changement au count(one et two), donc on va préciser cela dans le tableau d'entrée.

```
const incrementCountOne = useCallback((val) => {
  console.log('je suis dans incrementCountOne');
  countOne.value < 100 && setCountOne({...countOne, value: countOne.value + val})
}, [countOne])
```

Là on a dit si on apporte une modif au tableau(count) tu peux procéder au retour d'une nouvelle fonction. C'est-à-dire dans le cas de l'exemple ce ne sera pas nécessaire de rappeler la fonction incrementCountTwo, juste une nouvelle incrementCountOne. Evidemment on va faire de même sur incrementCountTwo. A partir du moment où la fonction n'est pas change le props sera lui aussi inchange,

```
<Button handleClick={ const incrementCo
<Button handleClick={incrementCountTwo}
```

Refaisons un test en console après rechargement de page. Vous pouvez constater la différence de chargement de composant. Ici nous avons optimiser notre code de manière significative. Si l'optimisation devient une priorité pour vous je vous conseille de regarder useMemo() et useRef(),mais qui sont totalement facultative, c'est juste a titre d'information que je les cite.
Vous pouvez également regarder le useLayoutEffect() mais on ne l'utilisera pas dans, cette partie car il a été déprécie a partir de la version 8.(gère certains souci d'affichage).

Les custom Hooks

Nous avons abordé la majorité des hooks de react, vou savez désormais compris que les hooks ne sont que des fonctions, et que l'on doit les importer de react depuis la version 16.8, ces fonctions encapsule un code codé par d'autre développeurs. Si c'est des fonctions et bien on est capable de créer nos propres fonctions donc nos propres hooks, regardons la doc et ce qu'ils disent.

> Construire vos propres Hooks vous permet d'extraire la logique d'un composant sous forme de fonctions réutilisables.
>
> **Un Hook personnalisé est une fonction JavaScript dont le nom commence par "use" et qui peut appeler d'autres Hooks.** Par exemple, `useFriendStatus` ci-dessous est notre premier Hook personnalisé :

Regardons les règles des hooks personnalisés. Nous allons prendre une de nos appli ou vous en créez une nouvelle, et dans src => component => on crée ClickSayHello.js puis on crée une fonction(rfce), on aura besoin de useState et du useEffect donc on les importe,

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

ON va enlever la div et créer un button avec evenement onClick qui renverra une arrow function, créons aussi un state, et donc la fonction du onclick.

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

function ClickSayHello() {

    const [text, setText] = useState('');

  return (
    <button onClick={() => setText("Hello World")}>Cliquez</button>
  )
}

export default ClickSayHello
```

Nous allons devoir créer également le useEffect(), n'oublions pas d'importer et d'afficher le composant dans la div du App.js,

```
return (
  <div className='App'>
    <ClickSayHello />
  </div>
);
```

Le bouton est bien apparu, gérons le useEffect(),

```
function ClickSayHello() {

    const [text, setText] = useState('');

    useEffect(() => {
        console.dir(document);
    }, [text]);

  return (
    <button onClick={() => setText("Hello World")}>Cliquez</button>
  )
}
```

Cela nuos permettra de voir tout ce que cela contient en console dont le title du 'document' sur lequel on va agir. (react app : ce qu'il y a dans l'onglet).
Modifions cela,

```
const [text, setText] = useState('');

useEffect(() => {
    document.title = text;
}, [text]);
```

On va plutôt écrire comme cela histoire d'avoir une chaine de caractère plus la variable.

```
useEffect(() => {
    document.title = `Titre: ${text}`;
}, [text]);
```

Allez voir la différence et cliquez, vous verrez le titre de l'onglet se mettre à jour. Très bien, ca on va dire que c'est notre petite logique de code, que l'on a au niveau de cette fonction. Si par exemple dans notre appli quelque part on souhaite modifier ce titre, pourrai le faire en répétant qlq chose qui ressemble à ça. Et bien au lieu de faire cela on va pouvoir encapsuler cette partie de code dans un hook, notre custom hook.
Créons un dossier hooks dans le src, (vous remarquerez un petit hameçon sur le dossier) et créons notre fichier pour le hook.
Appelons le useUpdateDocTitle.js et donc on a dit c'est quoi un hook ???
Une fonction et oui donc on part sur une fonction, on revient sur CLickSayHello.js et on recupere le morceau de code qui nous interesse,

```
const [text, setText] = useState('');

useEffect(() => {
    document.title = `Titre: ${text}`;
}, [text]);
```

Celui qui nous permet d'insérer le titre dans le title du document, on le copie et on efface de ce fichier on le colle dans notre custom hook. On mettre le useState dans le ClickSayHello.js

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

function ClickSayHello() {

    const [text, setText] = useState('');

  return (
    <button onClick={() => setText("Hello World")}>Cliquez</button>
  )
}

export default ClickSayHello
```

Du coup on modifie légèrement le fichier useUpdateDocTitle.js

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

const useUpdateDocTitle = (arg) => {

    useEffect(() => {
        document.title = `Titre: ${arg}`;
    }, [text]);
}
```

Et on ajoute l'export en bas. Et on remplace le tableau text, par le arg.

```
    useEffect(() => {
        document.title = `Titre: ${arg}`;
    }, [arg]);
```

On save et on importe ce composant dans le ClickSayHello.js, et on la rappel.

```
  const [text, setText] = useState('');
  useUpdateDocTitle(text);
return (
  <button onClick={() => setText("Hello World")}>Cliquez</button>
```

On save et on va vérifier. Très bien vous avez compris ce qu'est un custom hook.
Bons continuons , créons un fichier MyContact.js, et on part sur un composant de type fonction qui nous retourne un input, donc on créer ce dernier dans components, on l'appellera Search.js, et encore un composant de type fonction(rfce), avec l'input, le type, le placeholder, la value et la méthode,

```jsx
const Search = () => {
  return (
    <>
        <input
            type="text"
            placeholder="Chercher.."
            value={}
            onChange={}
        />
    </>
  )
}

export default Search
```

Et on l'importe dans MyContact.js,

```jsx
import Search from "./Search"

const MyContacts = () => {
    return (
        <Search />
    )
}
```

Sachant que c'est ici que l'on va utiliser la value et le onchange on va devoir les véhiculé en tant que props, donc on le défini, on va commencer par créer un state pour gérer la value(la valeur tapé par l'utilisateur),

```jsx
import { useState } from 'react';

import Search from "./Search";

const MyContacts = () => {

    const [search, setSearch] = useState('');

    return (
        <Search
            |
        />
    )
}
```

On va donc véhiculer la valeur du state donc search dans un props

```
return (
    <Search
        searchStr={search}
    />
)
```

Maintenant que l'on a le props, et bien on le récupère au niveau de Search.js via le destructuring, et on l'applique au value,

```
const Search = ({searchStr}) => {
  return (
    <>
      <input
        type="text"
        placeholder="Chercher.."
        value={searchStr}
```

Ensuite et bien il va nous rester le handler, on retourne dans le corps de notre fonction et on fait notre méthode, (MyContact.js)

```
const [search, setSearch] = useState('');

const handleChange = e => {

}

return (
    <Search
        searchStr={search}
        searchhandler={handleChange}
```

Et la valeur qu'on va passer ici c'est pour mettre à jour le search donc le setSearch,

```
const handleChange = e => {
    setSearch(e.target.value)
}

return (
    <Search
        searchStr={search}
        searchhandler={handleChange}
    />
)
```

Evidemment il faut penser a passer le props dans le search,

```
<input
    type="text"
    placeholder="Chercher.."
    value={searchStr}
    onChange={searchhandler}
/>
```

Suite à ca quand on va passer de la data au niveau du input, a chaque fois qu'on tape on va avoir le onChange qui va s'exécuter, il va invoquer notre méthode (searchhandler) et celle-ci va prendre la valeur (handleChange valeur de la props searchhandler), du input en question et va faire en sorte de mettre à jour le search. N'oubliez pas d'enregistrer vos fichiers, et si tout va bien vous avez oublié d'importer le composants MyContacts au niveau du App.js. Normalement vous aurez une erreur j'aimerais que si oui vous cherchiez comment résoudre cette erreur.

ET bien simplement car nous n'avons pas exporter le searchhandler via le destructuri,g comme pour le searchStr dans le composant Search.js on a maintenant un input, dans le quel on peut passer des valeurs donc si on fait un console.log de search

```
const MyContacts = () => {

    const [search, setSearch] = useState('');

    console.log(search)
```

Allez dans la console et regardez votre input quand vous écrivez, ok tout est bon, maintenant on aimerai que la valeur que l'on passe dans l'input on la passe au niveau du titre, on a le composant qui au click affiche hello world dans le titre de l'onglet, et bien on va avoir le input qui aura la même logique encapsule dans le custom Hook , pour cela ça va être très très simple, au niveau du file ClickSayHello, on a importé le hook useUpdateDocTitle

```
function ClickSayHello() {

   const [text, setText] = useState('');

   // Custom Hook
   useUpdateDocTitle(text);

  return (
    <button onClick={() => setText("Hello World")}>Cliquez</button>
  )
}
```

ON va donc faire exactement la même chose dans MyContacts.js,

```
const MyContacts = () => {

    const [search, setSearch] = useState('');

    useUpdateDocTitle(text);

    const handleChange = e => {
        setSearch(e.target.value)
    }
```

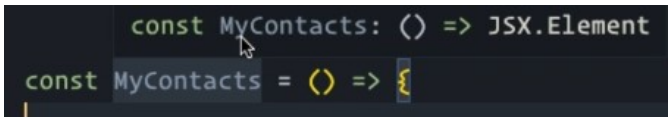
Evidemment on n'oublie pas de l'importer,

```
import { useState } from 'react';
import useUpdateDocTitle from '../hooks/useUpdateDocTitle'
```

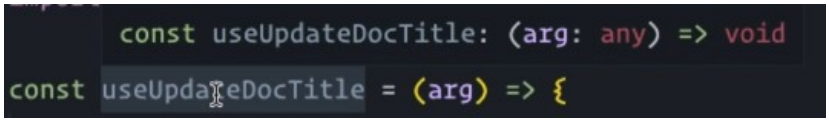
Une fois notre hook importé, on le définit et on lui passe une valeur(text), sauf que dans notre cas ce n'est pas text mais le search qu'il nous faut. On save et va voir en console. Quand vous tapez un

texte, dans l'input, il s'affichez en titre de l'onglet, et si on clique sur le bouton on remet hello world, si vous retapez du texte et recliquez et bien il ne se passera rien pas de hello world POURQUOI ?????

Revenons au niveau du composant MyContact et que l'on passe la souris dessus-il, nous donne des infos,



Il nous dit que c'est une fonction qui nous retourne du jsx, donc nous avons un composant React qui obéit au cycle de vie d'un composant React en revanche si on se rend au niveau de useUdateDocTitle.js on a dit que les hooks ne sont rien d'autres que des fonctions, regardons en core les infos,



Il nous dit que c'est une simple fonction a qui on passe un argument any, c'est-à-dire string, array etc… tout ce qu'on veut que ca nous retourne, et ça nous retourne void, en fait en gros ca nous retourne rien, quand on a le void la fonction ne retourne rien. Donc sachant qu'ici ce n'est pas un composant React on a simplement le useEffect qui se base sur la valeur que l'on passe au **arg** et a chaque fois que celle-ci change, on va avoir la callBack qui va s'enclencher pour pouvoir injecter la valeur au niveau du titre de l'objet document

Sauf que au niveau de notre bouton (dans le ClickSayHello.js), quand on clique on a au départ le state en chaine de caractères vide,

ON a donc un état et quand on clique on aura un autre état avec le hello world, (montage et mise à jour) les infos sont véhiculées au niveau de notre hook

Sauf que en recliquant sur le bouton nous allons avoir la deuxième valeur hello world que nous allons essayer de modifier en hello world, donc quand la valeur sera revenu au useEffect et bien il va constater que c'est la même chose, donc il va pas relancer le callBack donc de ce fait n'aura pas de nouvel affichage au niveau du titre après click sur le bouton. Donc on va régler cette problématique, assez facilement, on revient au niveau de notre bouton(ClickSayHello.js), donc comment faire pour véhiculer une nouvelle data ? et bien on va créer un nouveau state,

Et pour la valeur initiale on va mettre true, on prend le setter et on remplace le setText,

Ainsi on va pouvoir modifier la valeur si au départ on a true on va faire en sorte qu'il passe en false et en recliquant en true etc.. Comme ca on aura une datat qui va changer à chaque click sur le bouton et suite a ce changement de data, on va décider de ce que l'on veut avoir au niveau du text

```
const [text, setText] = useState('');
const [isTrue, setIsTrue] = useState(true);

// Custom Hook
useUpdateDocTitle(text);
```

Comment on fait cela, sachant que dans un composant on a le cycle de vie donc à chaque fois que l'on modifie le valeur (true et false) ?
On peut passer sur useEffect. On va donc le définir, et lui définir en dépendances isTrue, parce que c'est ce dernier qui change à chaque fois qu'on click sur le bouton.

```
const [text, setText] = useState('');
const [isTrue, setIsTrue] = useState(true);

useEffect(() => {

}, [isTrue]);
```

Maintenant on doit décider de ce l'on va avoir ici, on va faire un if et dire voir la valeur et istrue et si ce n'est pas true on mettra à jour, on va dire si c'est true bonjour et si c false bonsoir,

```
useEffect(() => {
  if(isTrue) {
    setText("Bonjour")
  } else {
    setText("Bonsoir")
  }
}, [isTrue]);
```

De ce fait on aura deux datas a véhiculer lors du click, et puisque nous allons avoir de nouvelles data nous allons avoir le callBack qui va s'enclencher pour injecter la data en question au niveau du titre.
Comme ce qu'on a fait au niveau de search on aura besoin de notre hook,

```
// Custom Hook
useUpdateDocTitle(text);
```

Avant de tester revenons sur qlq chose d'important, le useUpdateDocTitle il ne s'agit pas dans un composant React mais d'un simple fonction qui joue un rôle de custom Hook, on a vu qu'il y avait des règles à respecter quand on utilise les hook, c'est à dire l'endroit où on invoque le hook en question, pensez bien a ces règles, sinon vous aurez des messages d'erreur. Allons tester ce que l'on a fait.
On va se rendre sur https://jsonplaceholder.typicode.com/ pour récupérer des données comme dans une base de données en format json. Donc vous allez descendre au niveau du Try it, et récupéré le fetch de connexion à l'api, on le copie, on revient au niveau de MyContact, on importe le useEffect, et on l'incorpore aux corps de notre fonction, et donc pour l'état local on va pouvoir partir sur un autre state, on va dire users

```
const MyContacts = () => {

    const [users, setUsers] = useState([]);
    const [search, setSearch] = useState('');

    useEffect(() => {

    }, []);
```

Nous allons faire en sorte d'enregistrer les données du json, on colle donc dans le useEffect(), le fetch qu'on a récupérer,

```
const [users, setUsers] = useState([]);
const [search, setSearch] = useState('');

useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(json => console.log(json))
}, []);
```

Il nous donne en adresse http le todos et nous nous voulons les users on modifie donc cela, afin de « lier » ce que l'on veut. Evidemment ici nous allons coder uniquement pour se composant, ensuite vie notre custom hook, on va faire en sorte de passer cela(l'adresse http) en argument. (on verra ca étape par étape) Au niveau du premier then, c'est-à-dire une fois que l'on aura obtenu la promesse, nous avons la callBack function avec l'argument response,

Dans ce premier .THEN, vous pouvez par exemple vérifier si la réponse contient de la data.
En effet, .CATCH peut ne pas capturer certaines erreurs si la requête est reçue coté serveur. L'autre problème, c'est quand le serveur renvoie une data vide. Dans ce cas-là, on peut vérifier si la requête nous a retourné une data. Si, c'est "false", on gère notre message d'erreur comme ceci:

```
.then(
    response => {
        console.log(response);
        /*
        - L'objet response contient une ptopriété 'ok'
        - Si ok est true, donc vous avez récupéré la data
        */
        if(!response.ok) { // Si ok c'est FALSE
            // génère un objet Error
            throw new Error("Oups! Nous avons un problème!");
            // Cette erreur sera capturée dans le .catch
        }
    }
)
```

Ensuite vie then(le second), on prend le json et on va faire en sorte de save ce dernier au niveau de notre usage,

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(json => {
        setUsers(json);
    })
}, []);
```

On prend donc le setter, et on lui passe en paramètre json afin de mettre à jour notre variable d'état, en attendant la réponse on pourrai passer un message au user afin de lui demander de patienter. On va créer un nouveau state,

```
const [users, setUsers] = useState([]);
const [isLoading, setIsLoading] = useState(true);
const [search, setSearch] = useState('');
```

En true , c'est-à-dire on est en train d'interroger notre api, donc il va falloir patienter, mais une fois que la data est récupérer et qu'on la enregistré au niveau de users on va faire en sorte de changer le setIsLoading en false. Donc on invoque le setter et on lui passe false en argument.

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(json => {
        setUsers(json);
        setIsLoading(false);
    })
}, []);
```

Gérons le cas des erreurs, là on va le faire en en console évidemment par la suite vous afficherez ce message a vos utilisateurs,

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(json => {
        setUsers(json);
        setIsLoading(false);
    })
    .catch(error => console.log(error.message))
}, []);
```

Certains me diront peut-être (selon la veille effectué) que l'on aurai pu gérer cela avec axios mais je prend le partie de vous montrer avec le fetch car comme ça on peut l'encapsuler d'une manière beaucoup plus simplifiés afin de pouvoir le réutiliser si on le souhaite. Ok maintenant on arrive a collecter la data. Sur notre appli on a un input on va faire en sorte de ne pas l'afficher tatn que la data n'est pas chargé, et oui ca sert a rien que l'user puisse chercher si il n'y a rien a trouvé.
On va donc partir sur le search et faire une condition, on va créer une balise(fragment), ouvrir les accolades car on va écrire du Js et mettre le search dedans, et donc créer la condition. Maintenant on va pas mettre le message en dur on va plutôt créer une fonction qui va gérer cela(oui on automatise)

```
const msgDisplay = (msg, color) => {
    return (
        <p style={{textAlign: 'center', color: color}}>
            { msg }
        </p>
    )
}
```

Voici la condition,

```
return (
    <>
        {
            isLoading ? msgDisplay('Veuillez patienter!', 'red') : (
                <Search
                    searchStr={search}
                    searchhandler={handleChange}
                />
            )

        }
```

Tout fonctionne c'est parfait, au rechargement même si cela va vraiment très vite on peut apercevoir le message furtivement. Ensuite on va faire en sorte que si on rentre une chaine de caractère qui correspond à ce que l'on aura dans un de ces objets(json) nous allons faire en sorte d'uniquement afficher ce dernier. Si en revanche plusieurs membres possèdent la même chaine de caractère nous

allons également l'afficher. Avant d'avancer j'aimerai sue vous fassiez un console.log de users afin de vérifier ce que vous avez récupérer.

```
const MyContacts = () => {

    const [users, setUsers] = useState([]);
    const [isLoading, setIsLoading] = useState(true);
    const [search, setSearch] = useState('');

    console.log(users)
```

Vous verrez en console le résultat de l'object contenant les éléments.
Créons un nouveau component, qui s'appellera TableUsers.js qui sera un composant de type function, perso je pars sur une arrow function et c'est juste une question de préférence.

```
import React from 'react'

const TableUsers = () => {
  return (
    <div>TableUsers</div>
  )
}

export default TableUsers
```

Alors je préfère préciser que ce n'est pas un vrai custom table car normalement il est censé s'adapter à tout type de datas que l'on veut afficher dedans. Dans notre cas ce tableau la s'adapter uniquement à la datas que l'on a(le json des users).
Alors importons notre composant dans MyContact.js, et mettons une petite condition pour pouvoir afficher ou pas. Enlevons la div et mettons une table, ensuite allons dans le App.css,

```
/* CSS TABLE */
#table {
  width: 90%;
  margin-left: auto;
  margin-right: auto;
  margin-top: 15px
}
```

Passons au th,

```
#table th {
  text-align: center;
  background-color: lightcoral;
  color: "#fff";
  border: 1px solid white;
  padding: 12px
}
```

Revenons a notre composant TableUsers et construisons celui-ci, en commençant par le table Head,

```
return (
  <table id="table">
      <thead>
          <tr>
              <th>Nom</th>
              <th>Nom Utilisateur</th>
              <th>Email</th>
              <th>Adresse</th>
          </tr>
      </thead>
  </table>
)
```

Ensuite passons au table body, mais avant cela on doit créer notre props car on doit rappeler la datas qui est dans users(array),

```
const TableUsers = ({dataArray}) => {
  return (
```

La props est créée on doit donc la passer dans MyContact, plus précisément au niveau de notre table

```
                searchhandler={handleChange}
            />
        )

    }

    {
        <TableUsers
            dataArray
        />
    }

</>
```

Ici on définit donc la props dataArray et on lui passe nos fameux users

```
{
    <TableUsers
        dataArray={users}
    />
}
```

On passe cela pour tester un peu notre tableau, on retourne sur TableUsers.js, on va rappeler notre props et lui appliquer la méthode map, et on va passer par le destructuring et appeler directement ce que l'on veut comme infos. On a dit que lorsque l'on utilise la méthode map, le premier élément c'est à dire l'élément parent que celle-ci va retourner, doit contenir le key, dans ce cas on a l'id donc on va utiliser ce dernier,

```
</thead>
<tbody>
    {
        dataArray.map(({id, name, username, email, address}) => {
            return (
                <tr key={id}>
                    <td>{name}</td>
                    <td>{username}</td>
                    <td>{email}</td>
                    <td>{address}</td>
```

La il affichera pas l'adresse complète pourquoi ???

Car vous pouvez remarquer que c'est un objet (adresse) donc on pourra faire adresse.street, etc.. On va afficher tout cela sur la même ligne comme cela,

```
<td>{address.street} {address.suite} {address.city} {address.zipcode}</td>
```

On save et on revient sur l'appli pour vérifier tout ça. Et on a bien notre tableau certes au niveau du style c'est pas ouf, mais fonctionnel. Allez on va ajouter un peu de style ça vous manque le CSS hein !!!! Alors on retourne sur le App.css,

```
#table th {
  text-align: center;
  background-color: lightcoral;
  color: "#fff";
}

#table td, #table th {
  border: 1px solid #ddd;
  padding: 12px
}
```

Voilà c'est beaucoup mieux déjà. Bravo a vous. Pour la suite on va afficher seulement les infos users que l'on passera dans l'input. Donc on va faire en sorte de ne pas afficher les users mais seulement ce que l'on tapera dans l'input. On se rend sur MyContact.js, on va créer une nouvelle variable d'état, (state) évidemment à la suite des autres,

```
const [resultSearch, setResultSearch] = useState('');
```

Sachant que l'on risque de récupérer plusieurs utilisateurs on va préférer les mettre dans un array,

```
const [resultSearch, setResultSearch] = useState([]);
```

On va définir les conditions dans notre table évidemment si larray est vide ou si on taper une chaine de caractère mais sans résultat alors message 'Pas de resultat', et la color en red, (évidemment si les deux sont true

```
resultSearch.length === 0 && search !== '' ? msgDisplay('Pas de résultat!', 'red')
  <TableUsers
    dataArray={users}
  />
}
```

sinon si c'est false,

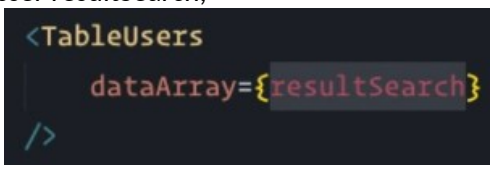
```
:
search === '' ? msgDisplay('Veuillez effectuer une recherche', 'green')
```

Et si les deux cas sont false on définit un dernier cas, et bien cela affiche tout.

```
{
  resultSearch.length === 0 && search !== '' ? msgDisplay('Pas de résultat!', 'red')
  :
  // search === '' ? msgDisplay('Veuillez effectuer une recherche', 'green')
  search === '' ? null
  :
  <TableUsers
    dataArray={users}
  />
}
```

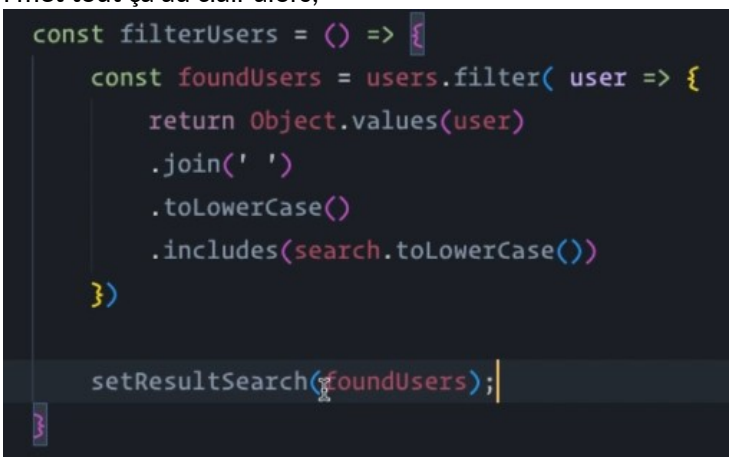
J'ai rajoute le search === '' ? null, pour tester. Mais je préfère ce qui est commenté bien sûr. Là on a pour le moment passé l'ensemble des éléments du json, mais ca serai bien a présent de faire notre fameux filtre, pour afficher seulement les personnes qui correspondent a notre recherche. Sachant

que l'on va enregistrer ca au niveau du resultSearch, du coup au lieu de passer users dans le dataArray on va désormais passer resultSearch,



Rappel : donc ce dernier va être collecté au niveau de notre TableUsers dataArray et ca va être sur ce dernier maintenant que l'on fera notre fetch, au lieu de le faire le fetch de tout le json.
Donc pour aliment notre resultSearch on va créer un useEffect, commençons par la const du filter comme on va l'utiliser de suite,

Allons voir le résultat du console log, ça correspond à ce que l'on veut ??
Oui bien sûr, alles on met tout ça au clair alors,



Et donc le useEffect,

ON peut save et aller vérifier tout est parfait ca y est. On approche du bout pour les customHooks, petit rappel le link Bootstrap est bien dans le index.js dans les imports. Je vous encourage à visiter cette page (https://nikgraf.github.io/react-hooks/), c'est une petite librairie de custom hooks déjà

crée par des devs et mis à disposition de la communauté, évidemment nous avons le site de npm (https://www.npmjs.com/) et sur celui-ci je conseille le package react-use, n'hésiter pas a aller chercher un package existant qui règlerai votre problème, il existe forcément un use qu'il vous faut.
Revenons sur notre MyContacts.js,