

## I. Les Hook et la gestion des états

Les Hooks sont des fonctions qui permettent de « se brancher » sur la gestion d'état local et de cycle de vie de React depuis des composants.

### A. Hook d'état : useState

Le **state** d'un composant correspond à son état local. Les **props** correspondent à un état en provenance de l'extérieur, le state est lui, lié au composant en tant que tel.

useState, est donc un hook qui permet de gérer un état local de votre composant.

```
const [state, setState] = useState(initialState);
```

useState retourne un tableau avec 2 variables :

- La première est l'état du composant.

```
import { useState } from "react"; //Import du hook

const Politesse = () => {
  // Déclare une variable d'état « user » et l'initialise à Tom
  const [user] = useState("Tom");
  return <h1>Bonjour {user} </h1>; //inclusion de la variable dans le résultat
};

<Politesse />
```

Affichera : **Bonjour Tom**

*Remarquez l'utilisation d'une variable JS avec les accolades dans le code HTML  
Vous pouvez utiliser n'importe quelle expression JavaScript valide dans des accolades en JSX. Par exemple, 3x2 ou appeler une fonction mafonction(eeee)*

- La seconde variable, sera le nom de la fonction de modification de la variable-état.

Ainsi pour modifier à la volée le nom on devra déclarer :

```
const [user, setUser] = useState("Tom");
```

et utiliser **setUser** dans un évènement.

#### 1. Utilisation avec les événements

```
import { useState } from "react";
const Politesse = () => {
  const [user, setUser] = useState("Tom");
  const generer = () => setUser("Steeve"); //On affecte Steeve à la variable-état
  return (
    <div>
      <h1>Bonjour {user}</h1>
      <button onClick={generer}>Générer</button>
    </div>
  );
};
export default Politesse;
```

Affichera : **Bonjour Steeve**, au clic sur le bouton

NB : Le nom de la fonction à exécuter, rattachée à l'évènement onClick, est mis directement, sans « » ni ()

→ Si le paramètre fourni est une valeur, la variable état prendra cette valeur.

→ La fonction de modification d'état peut aussi prendre une fonction en paramètre. Dans ce cas, le paramètre de la fonction sera l'ancienne valeur de l'état

```
const [compteur, setCompteur] = useState(0)
setCompteur((oldValue) => oldValue+1) est équivalent à setCompteur(compteur+1)
```

- Dupliquez le projet react-starter et renommez-le reacte-useState
- Dans le composant Formulaire
- Importer le hook « useState »
- Dans le corps du composant, la partie supérieure (avant return), déclarez un état « **data** » qui sera initialisé par défaut à {}
- Cet état contiendra les données saisies dans le formulaire
- Créer une fonction **updateForm** qui sera déclenchée à chaque modification d'un champ du formulaire (onChange)
- La fonction updateForm enregistrera dans l'état le nom du champ et sa valeur
- Créer une fonction **submitForm** qui sera déclenchée au clic sur le bouton et affichera le contenu de data

NB : La fonction « updateForm » sera créée dans le composant formulaire et sera envoyée en paramètres au sous composant « FormBlockInput », chargé de détecter l'évènement « onChange »

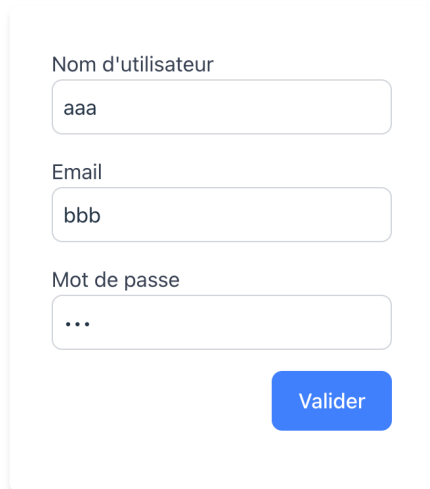
La fonction « submitForm » sera déclenchée dans le sous composant « FormBlockButton » sur l'évènement « onClick »

Les données de chaque input sont enregistrées

Pensez à annuler l'action par défaut déclenchée par le bouton valider, pour éviter l'envoi du formulaire

## Mon formulaire

```
▶ {username: 'aaa', email: 'bbb', password: 'ccc'}
```



## B. Hook d'effet : useEffect

Le Hook d'effet permet l'exécution d'effets de bord dans des composants.

Ce sont les actions qui peuvent affecter d'autres composants et qui ne peuvent pas être réalisées pendant l'affichage. On va retrouver le chargement de données depuis une source externe, les abonnements à des événements du navigateur ou encore la réaction à l'évolution d'autres composants

Il permet d'indiquer à React que notre composant doit exécuter quelque chose après chaque affichage. React enregistre la fonction passée en argument (que nous appellerons « effet »), et l'appellera plus tard, **après avoir mis à jour** le DOM.

NB : Les composants sont rendus, en cas de modification de leur propre état ou **de leur parent**. Par conséquent, **useEffect** sera déclenché à chaque modification d'un props ou d'un state.

```
useEffect(effectFunction, conditionalArray);
```

- Le premier paramètre sera une fonction à exécuter à chaque déclenchement d'un effet de bord. Pour une meilleure lisibilité et un débogage plus facile, il est préconisé d'utiliser un effet par action.

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

const Compteur = () => {
  const [counter, setcounter] = useState(0); //Etat compteur
  const [visibility, setVisibility] = useState(); // Etat visibilité du bouton -
  useEffect(() => {
    setVisibility(counter == 0);
  });

  const decrease = () => setcounter(counter - 1);
  const increase = () => setcounter(counter + 1);

  return (
    <div className="container py-4 px-3 mx-auto">
      <div>Valeur: <span className=" text-2xl text-red-500">{counter}</span></div>
      <div>
        <button onClick={decrease} disabled={visibility} >-</button>
        &nbsp;
        <button onClick={increase} >+</button>
      </div>
    </div>
  );
};

export default Compteur;
```

- Le second paramètre (optionnel), est un tableau de propriétés à surveiller.  
NB : S'il est défini et que le tableau est vide **l'effet ne se lancera qu'une fois**.

Par exemple dans l'exemple précédent on pourrait ajouter un timer

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

const Compteur = () => {
  const [counter, setcounter] = useState(0); //Etat compteur
  const [visibility, setVisibility] = useState(); // Etat visibilité du bouton -
  const [time, setTime] = useState(); // Etat pour l'horloge

  useEffect(() => {
    setVisibility(counter == 0);
    console.log("Effet lancé");
  });

  const decrease = () => setcounter(counter - 1);
  const increase = () => setcounter(counter + 1);
  function refresh() {
    var t = 1000; // rafraîchissement en millisecondes
    setTimeout(compute, t);
  }
  const compute = () => {
    var date = new Date();
    var h = date.getHours();
    var m = date.getMinutes();
    var s = date.getSeconds();
    if (h < 10) { h = "0" + h; }
    if (m < 10) { m = "0" + m; }
    if (s < 10) { s = "0" + s; }
    var time = h + ":" + m + ":" + s;
    setTime(time);
    refresh();
  };

  refresh();

  return (
    <div className="container py-4 px-3 mx-auto">
      <div><h2> Bonjour il est <strong>{time}</strong></h2></div>

      <div>Valeur: <span className="text-2xl text-red-500">{counter}</span></div>
      <div>
        <button onClick={decrease} disabled={!visibility}>-</button>
        &nbsp;
        <button onClick={increase}>+</button>
      </div>
    </div>
  );
};

export default Compteur;
```

- Essayez le script en affichant la console log, que se passe-t-il ?

## Mon compteur

29 Effet lancé

Bonjour il est 20:32:32

Valeur: 0



- Modifier la gestion de l'effet comme suit

```
useEffect(() => {
  setVisibility(counter == 0);
  console.log("Effet lancé");
}, [counter]); //On limite l'effet à la modification du compteur
```

Notre effet ne se lancera qu'au changement de la variable « counter », et non plus avec le timer.

- Dupliquez le projet react-useState et renommez-le react-useEffect
- Reprenez le composant Formulaire
- Importer le hook « useEffect »
- Déclarez un état visibility et sa fonction setVisibility pour gérer la visibilité du bouton Valider
- Le bouton ne s'affiche que si les 3 zones sont remplies

## Mon formulaire

Nom d'utilisateur

Email

Mot de passe

## Mon formulaire

Nom d'utilisateur

Email

Mot de passe

Valider

## C. Hook de référence : useRef

D'après [la documentation React](#) le useRef permet de stocker une valeur qui n'est pas nécessaire pour le rendu.

```
const ref = useRef(initialValue)
```

useRef renvoie **un objet ref modifiable** dont la propriété **current** est initialisée avec l'argument fourni (initialValue). L'objet renvoyé persistera **pendant toute la durée de vie du composant**.

Pour le modifier il faudra utiliser la méthode `xxx.current = value`

NB : Contrairement à `useState`, `UseRef` ne génère pas le rafraîchissement de l'affichage, ce qui est pratique pour stocker des variables de travail ou la référence d'un objet du DOM même.

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

const Compteur = () => {
  const [stateCounter, setStateCounter] = useState(0); //Etat compteur
  const refCounter = useRef(stateCounter);

  const incState = () => setStateCounter(stateCounter + 1);
  const incRef = () => refCounter.current++;

  return (
    <div className="container py-4 px-3 mx-auto">
      <div>
        <button onClick={incState} >
          State = {stateCounter}
        </button>
        &nbsp;
        <button onClick={incRef} >
          Ref = {refCounter.current}
        </button>
      </div>
    </div>
  );
};

export default Compteur;
```

Mon compteur

STATE = 11

REF = 20

Remarquez que la valeur sur le bouton REF, ne se met à jour que lorsque l'on clique sur l'autre bouton. Seul l'autre bouton provoque un rafraîchissement de l'affichage grâce à `useState()`.

## II. Communication avec les API

Une application frontend a bien souvent besoin d'une partie backend qui lui fournit un ensemble de services, comme l'authentification, l'accès aux données etc ...

Pour certaines applications ces services peuvent être fournis par des composants/api externes.

Dans un cas comme dans l'autre, avant d'interroger l'api, nous devons :

1. Fournir un premier état qui stockera les données.
2. Fournir un deuxième état permettra de gérer la phase de chargement.
3. Créer un dernier état pour les erreurs

```
const [data, setData] = useState(null);  
const [loading, setLoading] = useState(true);  
const [error, setError] = useState(null);
```

### A. Fetch

Pour rappel, Javascript comprend une Api en natif **Fetch**

La méthode fetch() prend en unique argument obligatoire le chemin de la ressource qu'on souhaite récupérer.

On va également pouvoir lui passer en argument facultatif une liste d'options sous forme d'objet littéral pour préciser la méthode d'envoi, les en-têtes, etc.

La méthode fetch() renvoie **une promesse** (un objet de type Promise) qui va se résoudre avec un objet Response. Notez que la promesse va être résolue **dès que le serveur renvoie les en-têtes HTTP, c'est-à-dire avant même qu'on ait le corps de la réponse.**

La promesse sera rompue si la requête HTTP n'a pas pu être effectuée. En revanche, l'envoi d'erreurs HTTP par le serveur comme un statut code 404 ou 500 vont être considérées comme normales et ne pas empêcher la promesse d'être tenue.

On va donc devoir vérifier le statut HTTP de la réponse. Pour cela, on va pouvoir utiliser les propriétés ok et status de l'objet Response renvoyé.

La propriété ok contient un booléen : true si le statut code HTTP de la réponse est compris entre 200 et 299, false sinon.

La propriété status va renvoyer le statut code HTTP de la réponse (la valeur numérique liée à ce statut comme 200, 301, 404 ou 500).

Pour récupérer le corps de la réponse, nous allons pouvoir utiliser les méthodes de l'interface Response en fonction du format qui nous intéresse :

- La méthode text() retourne la réponse sous forme de chaîne de caractères ;
- La méthode json() retourne la réponse en tant qu'objet JSON ;
- La méthode formData() retourne la réponse en tant qu'objet FormData ;
- La méthode arrayBuffer() retourne la réponse en tant qu'objet ArrayBuffer ;
- La méthode blob() retourne la réponse en tant qu'objet Blob ;

On va pouvoir faire un hook personnalisé « useFetch » qui fera le travail à notre place

- Dans un nouveau projet react, créez un fichier « useFetch.js » dans le dossier « src »

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

function useFetch(url) {
  const [data, setData] = useState(null); // gere les données
  const [loading, setLoading] = useState(true); // gère l'état du chargement
  const [error, setError] = useState(null); // les eventuelles erreurs

  //A chaque modification de l'url l'effet sera lancé
  useEffect(() => {

    //Chargement des données en mode asynchrone bien sur :- )
    async function fetchData() {
      try {
        const response = await fetch(url);
        const data = await response.json();
        setData(data);
        setLoading(false);
      } catch (error) {
        setError(error);
        setLoading(false);
      }
    }
    fetchData(); //lancement du chargement des données
  }, [url]);

  return { data, loading, error }; //on export nos 3 variables
}

export default useFetch;
```

- utilisation dans le composant « App »

```
import useFetch from "../useFetch";

function App() {
  const { data, loading, error } = useFetch("https://geo.api.gouv.fr/departements");

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <span>
      Resultat: <br />
      {JSON.stringify(data)}
    </span>
  );
}

export default App;
```



Fil Rouge #03

