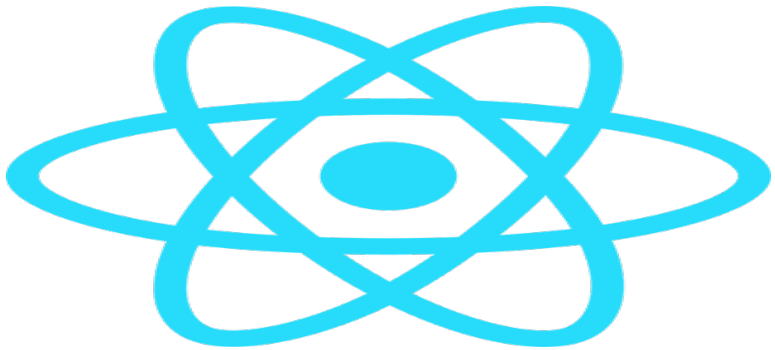
React : hooks

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

React

Hook

Une fonction qui permet de se griffer (to hook up) sur des fonctionnalités **React** comme l'état ou le cycle de vie dans des composants fonctionnels.

React

Hooks [**React** 16.8 (Février 2019)]

- **Fonctions JavaScript réutilisables** : permettent d'ajouter des fonctionnalités comme l'état et le cycle de vie à des composants fonctionnels.
- **Gestion de l'état local** : `useState` permet de gérer l'état dans les composants sans utiliser de classes.
- **Effets de bord** : `useEffect` permet de gérer des effets comme les requêtes réseau ou les abonnements.
- **Cycle de vie des composants** : Avec `useEffect`, vous pouvez contrôler les phases du cycle de vie comme le montage, la mise à jour et le démontage d'un composant.
- **Extensibilité** : Les hooks personnalisés permettent de factoriser la logique réutilisable entre les composants.

React

Question

Comment on faisait avant **React** 16.8 ?

React

Pour la gestion d'état

- Les composants de classe utilisaient `this.state` (comme `this.props`).
- Les composants fonctionnels ne pouvaient pas gérer l'état.

React

Pour le cycle de vie

- Les composants de classe utilisaient des méthodes spécifiques pour contrôler le cycle de vie, telles que
  - `componentDidMount` : appelée après le montage du composant.
  - `componentDidUpdate` : appelée après chaque mise à jour du composant.
  - `componentWillUnmount` : appelée juste avant le démontage du composant.
- Les composants fonctionnels, étaient des composants simples, qui ne pouvaient pas gérer d'état ou répondre aux cycles de vie du composant. Ils se contentaient de recevoir des **props** et de renvoyer du **JSX** pour le rendu.

React

Exercice

- Créer un composant `Compteur.js`
- Ajouter une variable `counter` initialisée à zéro
- Ajouter un bouton `+` permettant d'incrémenter `counter` et un bouton `-` permettant de le décrémenter.

React

Une première proposition

```
export default function Compteur() {
    let counter = 0

    const incrementer = () => {
        counter++
    }
    const decrementer = () => {
        counter--
    }
    return (
        <div>
            <h1>Compteur </h1>
            <button onClick={decrementer}>-</button>
            {counter}
            <button onClick={incrementer}>+</button>
        </div>
    )
}
```

React

Remarque

En cliquant sur les boutons, le rendu ne se met pas à jour.

Bien que la valeur change dans le composant

```
export default function Compteur() {
    let counter = 0

    const incrementer = () => {
        console.log(counter);
        counter++
        console.log(counter);
    }
    const decrementer = () => {
        console.log(counter);
        counter--
        console.log(counter);
    }
    return (
        <div>
            <h1>Compteur </h1>
            <button onClick={decrementer}>-</button>
            {counter}
            <button onClick={incrementer}>+</button>
        </div>
    )
}
```

React

Explication

- L'utilisation d'une variable locale ne permet pas, à **React**, de gérer l'état du composant.
- En **React**, pour que l'interface utilisateur se mette à jour automatiquement lorsqu'une variable change, cette variable doit être un état **React** (`state`).

React

Solution avec les hooks

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

export default function Compteur() {
    let [counter, setCounter] = useState(0);

    const incrementer = () => {
        setCounter(counter + 1)
    }
    const decrementer = () => {
        setCounter(counter - 1)
    }
    return (
        <div>
            <h1>Compteur </h1>
            <button onClick={decrementer}>-</button>
            {counter}
            <button onClick={incrementer}>+</button>
        </div>
    )
}
```

React

Explication

- Le hook `useState` permet de déclarer une variable d'état dans le composant fonctionnel.
- `useState(0)` initialise l'état `counter` à 0.
- `setCounter` est la fonction utilisée pour mettre à jour cette valeur.
- `incrementer` et `decrementer` utilisent `setCounter` pour mettre à jour l'état du compteur.
- La mise à jour de la valeur de `counter` déclenche un nouveau rendu du composant avec la nouvelle valeur.

React

La fonction qui modifie le state, fournie par `useState`, peut accepter deux types de paramètres

- **Une nouvelle valeur directe** : Cette valeur sera utilisée comme le nouvel état.
- **Une fonction de mise à jour** : Cette fonction reçoit l'état actuel et doit retourner le nouvel état. Cela pourrait être utile lorsque la mise à jour dépend de l'état précédent.

React

Re-render

- Un changement d'état dans le composant $\Rightarrow$ un re-render : l'interface utilisateur est mise à jour en réponse aux changements d'état.
- Un re-render $\equiv$ re-exécution du code de la fonction.
- Un re-render $\not\Rightarrow$ recréation du composant : les valeurs d'état et les références sont conservées à travers les re-renders.

React

Re-render

- Un changement d'état dans le composant $\Rightarrow$ un re-render : l'interface utilisateur est mise à jour en réponse aux changements d'état.
- Un re-render $\equiv$ re-exécution du code de la fonction.
- Un re-render $\not\Rightarrow$ recréation du composant : les valeurs d'état et les références sont conservées à travers les re-renders.

Attention

- En **React**, le déclenchement d'un re-render est étroitement lié au concept d'état (`state`) et de propriétés (`props`).
- Les variables locales au sein d'un composant ne sont pas directement observées par **React** et elles ne déclenchent pas un re-render.

React

Remarques

- Si l'état d'un composant enfant change, cela ne provoque pas de re-render du composant parent.
- Si l'état d'un composant parent change, cela déclenche un re-render du parent, et donc potentiellement des composants enfants aussi. Les enfants seront re-rendus si leurs `props` ont changé.

React

Remarques

- Si l'état d'un composant enfant change, cela ne provoque pas de re-render du composant parent.
- Si l'état d'un composant parent change, cela déclenche un re-render du parent, et donc potentiellement des composants enfants aussi. Les enfants seront re-rendus si leurs `props` ont changé.

En plus simple, le re-render est déclenché

- **Modification de l'état** : En utilisant `setState` pour modifier une valeur d'état.
- **Modification des props** : Si un composant enfant reçoit de nouvelles `props` de son parent.

React

Exercice

- Créer un composant `OnOff`.
- Ajouter deux boutons **HTML** dont l'un est initialement désactivé (`disabled`).
- En cliquant chaque fois sur le bouton activé, ce dernier se désactive et l'autre s'active.

React

Solution

```
import { useState } from "react"

export default function OnOff() {
    let [disabled, setDisabled] = useState(true)
    function changerEtat() {
        setDisabled(!disabled)
    }
    return (
        <div>
            <h2>OnOff</h2>
            <button
                onClick={changerEtat}
                disabled={disabled}>
                {!disabled ? 'On' : 'Off'}
            </button>
            <button
                onClick={changerEtat}
                disabled={!disabled}>
                {disabled ? 'On' : 'Off'}
            </button>
        </div>
    )
}
```

React

Considérons un deuxième état `signe` dans `Compteur.js`

```
const [counter, setCounter] = useState(0);
const [signe, setSigne] = useState('nul');
```

React

Considérons un deuxième état `signe` dans `Compteur.js`

```
const [counter, setCounter] = useState(0);
const [signe, setSigne] = useState('nul');
```


Exercice

Afficher la valeur du compteur ainsi que le signe (`nul`, `positif` ou `négatif`) en fonction de la valeur de `counter`.

React

La première partie de la solution

```
const mettreAJourSigne = () => {
    if (counter > 0) {
        setSigne('positif');
    } else if (counter < 0) {
        setSigne('négatif');
    } else {
        setSigne('nul');
    }
};

const incrementer = () => {
    setCounter(counter + 1);
    mettreAJourSigne()
};

const decrementer = () => {
    setCounter(counter - 1);
    mettreAJourSigne()
};
```

React

Et le rendu

```
return (
    <div>
        <h1>Compteur</h1>
        <div>
            <button onClick={decrementer}>-</button>
                {counter} ({signe})
            <button onClick={incrementer}>+</button>
        </div>
    </div>
);
```

React

Deux inconvénients de la solution précédente

- Le signe se met à ajour en retard.
- Les fonctions `incrementer` et `decrementer` appellent `mettreAJourSigne` en plus de la mise à jour de `counter`.

React

Deux inconvénients de la solution précédente

- Le signe se met à ajour en retard.
- Les fonctions `incrementer` et `decrementer` appellent `mettreAJourSigne` en plus de la mise à jour de `counter`.

Rappel

`setCounter` effectue la mise à jour en asynchrone.

React

Remarques

- En utilisant `++counter`, le signe est correctement mis à jour.
- Cependant, cela fonctionne en modifiant directement la variable `counter`, ce qui va à l'encontre des pratiques recommandées dans **React**.
- En effet, l'état doit être mis à jour via les fonctions de mise à jour de l'état (`setCounter`) et non pas directement, car la gestion de l'état dans **React** est asynchrone pour des raisons de performance et d'optimisation.

React

Pour la mise à jour immédiate du signe

```
const [counter, setCounter] = useState(0);
const [signe, setSigne] = useState('nul');

const mettreAJourSigne = (valeur) => {
    if (valeur > 0) {
        setSigne('positif');
    } else if (valeur < 0) {
        setSigne('négatif');
    } else {
        setSigne('nul');
    }
};

const incrementer = () => {
    const nouvelleValeur = counter + 1;
    setCounter(nouvelleValeur);
    mettreAJourSigne(nouvelleValeur)
};

const decrementer = () => {
    const nouvelleValeur = counter - 1;
    setCounter(nouvelleValeur);
    mettreAJourSigne(nouvelleValeur)
};
```

React

Et le rendu reste inchangé

```
return (
    <div>
        <h1>Compteur</h1>
        <div>
            <button onClick={decrementer}>-</button>
                {counter} ({signe})
            <button onClick={incrementer}>+</button>
        </div>
    </div>
);
```

React

Remarques

- Le signe se met à ajour immédiatement.
- En revanche, les fonctions `incrementer` et `decrementer` appellent toujours `mettreAJourSigne`.

React

Deuxième solution avec `useEffect`

- Se déclenche après le rendu : à chaque modification dans une `prop` ou dans le state, le composant concerné et ses enfants sont re-rendus.
- Effectue une action à un moment donné du cycle de vie du composant.

La deuxième solution avec `useEffect`

```
const [counter, setCounter] = useState(0);
const [signe, setSigne] = useState('nul');

useEffect(() => {
    if (counter > 0) {
        setSigne('positif');
    } else if (counter < 0) {
        setSigne('négatif');
    } else {
        setSigne('nul');
    }
}, [counter])

const incrementer = () => {
    const nouvelleValeur = counter + 1;
    setCounter(nouvelleValeur);
};

const decrementer = () => {
    const nouvelleValeur = counter - 1;
    setCounter(nouvelleValeur);
};
```

React

Le rendu ne change pas

```
return (
    <div>
        <h1>Compteur</h1>
        <div>
            <button onClick={decrementer}>-</button>
                {counter} ({signe})
            <button onClick={incrementer}>+</button>
        </div>
    </div>
);
```

React

Explication

- `useEffect` prend deux paramètres :
  1. Le premier étant une fonction qui vérifie la valeur de `counter` et met à jour l'état de `signe` en conséquence.
  2. Le deuxième étant un tableau de dépendances. Il indique que l'effet doit être exécuté chaque fois que `counter` change.
- Chaque fois que `counter` est mis à jour, la fonction passée à `useEffect` est exécutée pour mettre à jour `signe`.

React

Explication

- `useEffect` prend deux paramètres :
  1. Le premier étant une fonction qui vérifie la valeur de `counter` et met à jour l'état de `signe` en conséquence.
  2. Le deuxième étant un tableau de dépendances. Il indique que l'effet doit être exécuté chaque fois que `counter` change.
- Chaque fois que `counter` est mis à jour, la fonction passée à `useEffect` est exécutée pour mettre à jour `signe`.

Remarque

- Le tableau de dépendance contient souvent des variables réactives : variables d'état, des **props** ou des fonctions stables (définies dans certaines librairies).
- Si le tableau de dépendances est omis, alors `useEffect` se déclenche après chaque rendu du composant.
- Si le tableau de dépendances est vide, alors `useEffect` se déclenche uniquement après le rendu initial.

React

Considérons le composant suivant

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

export default function Effect() {
  useEffect(() => {
    console.log('effect');
  }, []);

  return <div>Effect Component</div>;
}
```

React

Considérons le composant suivant

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

export default function Effect() {
  useEffect(() => {
    console.log('effect');
  }, []);

  return <div>Effect Component</div>;
}
```


Remarque

Lancez l'application et vérifiez que `effect` s'affiche deux fois.

Explication

- **React** 18 a introduit un comportement en mode strict où il exécute certains **hooks** (comme `useEffect`, `useState`...) deux fois lors du rendu initial dans le but de détecter des effets secondaires non sécurisés (qui doivent être exécutés une seule fois).
- Cela permet aux développeurs de repérer les erreurs potentielles avant de passer en production.
- Le double appel de l'effet ne se produit qu'en mode de développement. En production, l'effet sera exécuté une seule fois.

Explication

- **React** 18 a introduit un comportement en mode strict où il exécute certains **hooks** (comme `useEffect`, `useState`...) deux fois lors du rendu initial dans le but de détecter des effets secondaires non sécurisés (qui doivent être exécutés une seule fois).
- Cela permet aux développeurs de repérer les erreurs potentielles avant de passer en production.
- Le double appel de l'effet ne se produit qu'en mode de développement. En production, l'effet sera exécuté une seule fois.

Supprimez les balises ouvrante et fermante (`<React.StrictMode>` et `</React.StrictMode>`) dans `index.js` et vérifiez que `effect` s'affiche une seule fois

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Cliquez plusieurs fois sur le bouton et vérifiez que le `useEffect` ne se déclenche pas après le changement de valeur d'une variable simple (ni état, ni `props`)

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

export default function Effect() {
    let x = 0
    const test = () => {
        console.log(x)
        if (x < 2) {
            x++
        }
    }
    useEffect(() => {
        console.log('useEffect')
    }, [x])
    return (
        <div>
            <h2>Effect</h2>
            <button onClick={test}>
                test
            </button>
        </div>
    )
}
```

React

Définissez `x` comme état et vérifiez que `useEffect` se déclenche après chaque changement de valeur (pas après chaque affectation))

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

export default function Effect() {
    let [x, setX] = useState(0)
    const test = () => {
        console.log(x)
        if (x < 2) {
            setX(x => x + 1)
        } else {
            setX(x)
        }
    }
    useEffect(() => {
        console.log('useEffect')
    }, [x])
    return (
        <div>
            <h2>Effect</h2>
            <button onClick={test}>
                test
            </button>
        </div>
    )
}
```

React

Avec un tableau de dépendance vide, vérifiez que `useEffect` se déclenche uniquement après le rendu initial

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

export default function Effect() {
    let [x, setX] = useState(0)
    const test = () => {
        console.log(x)
        if (x < 2) {
            setX(x => x + 1)
        } else {
            setX(x)
        }
    }
    useEffect(() => {
        console.log('useEffect')
    }, [])
    return (
        <div>
            <h2>Effect</h2>
            <button onClick={test}>
                test
            </button>
        </div>
    )
}
```

React

Sans tableau de dépendance, vérifiez que `useEffect` se déclenche après chaque changement de valeur (pas après chaque affectation))

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

export default function Effect() {
    let [x, setX] = useState(0)
    const test = () => {
        console.log(x)
        if (x < 2) {
            setX(x => x + 1)
        } else {
            setX(x)
        }
    }
    useEffect(() => {
        console.log('useEffect')
    })
    return (
        <div>
            <h2>Effect</h2>
            <button onClick={test}>
                test
            </button>
        </div>
    )
}
```

React

Exercice

Afficher la valeur du compteur :

- en rouge lorsqu'elle est négative,
- en jaune lorsqu'elle est nulle,
- en noir lorsqu'elle est positive.

React

Solution

```
const [counter, setCounter] = useState(0);
const [signe, setSigne] = useState('nul');
const [couleur, setCouleur] = useState('black');

useEffect(() => {
    if (counter > 0) {
        setSigne('positif')
        setCouleur('black')
    } else if (counter < 0) {
        setSigne('négatif')
        setCouleur('tomato')
    } else {
        setSigne('nul')
        setCouleur('gold')
    }
}, [counter])

const incrementer = () => {
    const nouvelleValeur = counter + 1;
    setCounter(nouvelleValeur);
};

const decrementer = () => {
    const nouvelleValeur = counter - 1;
    setCounter(nouvelleValeur);
};
```

React

Et le rendu

```
return (
    <div>
        <h1>Compteur</h1>
        <div style={{color: couleur}}>
            <button onClick={decrementer}>-</button>
                {counter} ({signe})
            <button onClick={incrementer}>+</button>
        </div>
    </div>
);
```

React

Exercice

Dans un composant quelconque (`Hello` ou autre)

- Ajouter un `input:text` et un bouton
- En cliquant sur le bouton, afficher la valeur de l'`input` dans un `alert`

La solution la plus intuitive (mais qui ne fonctionne pas)

```
const [valeur, setValeur] = useState('')

const showValue = () => {
    alert(valeur)
}
```

La solution la plus intuitive (mais qui ne fonctionne pas)

```
const [valeur, setValeur] = useState('')

const showValue = () => {
    alert(valeur)
}
```

La valeur initiale d'un état associé à un `input` ne doit pas être `null`.

La solution la plus intuitive (mais qui ne fonctionne pas)

```
const [valeur, setValeur] = useState('')

const showValue = () => {
    alert(valeur)
}
```

La valeur initiale d'un état associé à un `input` ne doit pas être `null`.

Le rendu

```
return (
    <div>
        <input type="text" value={valeur} />
        <div>
            <button onClick={showValue}>
                alert
            </button>
        </div>
    </div>
)
```

React

En testant, on a le message suivant qui s'affiche

```
Warning: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.
    at input
    at div
    at UpperConvert (http://localhost:3000/static/js/bundle.js:1150:78)
    at div
    at App (http://localhost:3000/static/js/bundle.js:90:78)
```

React

En testant, on a le message suivant qui s'affiche

```
Warning: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.
    at input
    at div
    at UpperConvert (http://localhost:3000/static/js/bundle.js:1150:78)
    at div
    at App (http://localhost:3000/static/js/bundle.js:90:78)
```


Explication

Il faut définir l'évènement `change` ou `input` pour synchroniser la valeur de l'input avec l'état du composant.

React

Rappel : onChange vs onInput vs onBlur

- onChange est déclenché lorsque l'utilisateur quitte le champ de saisie (par exemple, en cliquant en dehors) et que la valeur du champ a changé. Autrement dit, le changement est détecté uniquement lorsque le champ perd le focus.
- onInput est déclenché chaque fois que la valeur du champ change, c'est-à-dire à chaque saisie de caractère, suppression, ou autre modification du contenu.
- onBlur est déclenché à chaque fois que l'élément perd le focus, que la valeur ait changé ou non.

Nouvelle solution

```
const [valeur, setValeur] = useState('')

const showValue = () => {
  alert(valeur)
}
const sendValue = (e) => {
  setValeur(e.target.value)
}
```

Nouvelle solution

```
const [valeur, setValeur] = useState('')

const showValue = () => {
  alert(valeur)
}
const sendValue = (e) => {
  setValeur(e.target.value)
}
```

Le rendu

```
return (
    <div>
        <input type="text" value={valeur} onInput={sendValue} />
        <div>
            <button onClick={showValue}>
                alert
            </button>
        </div>
    </div>
)
```

React

Remarque

Une deuxième solution consiste à utiliser `useRef`.

React

Remarque

Une deuxième solution consiste à utiliser `useRef`.

`useRef`

- Utilisé pour
  - accéder directement à un élément du **DOM**
  - stocker une valeur mutable qui ne nécessite pas de re-rendu lorsqu'elle change
- Retournant un objet **JavaScript** contenant une propriété `current` faisant référence à un élément **HTML** spécifique, typiquement un objet de type `HTMLElement` (ou une de ses sous-classes, comme `HTMLInputElement`, `HTMLDivElement`...).

React

Commençons par déclarer une référence

```
const valeur = useRef('')
```

React

Commençons par déclarer une référence

```
const valeur = useRef('')
```

Associons la référence à l'`input`

```
<div>
    <input type="text" ref={valeur}  />
</div>
```

React

Commençons par déclarer une référence

```
const valeur = useRef('')
```

Associons la référence à l'`input`

```
<div>
    <input type="text" ref={valeur}  />
</div>
```


Récupérons la valeur dans `alert`

```
const showValue = () => {
    alert(valeur.current.value)
}
```

React

Remarques

- `valeur.current` : retourne l'élément input, avec toutes ses propriétés.
- `valeur.current.value` : retourne la valeur actuelle de cet input.

React

Exercice

- Dans le composant `Compteur`, ajouter un `input`
- La valeur de `input` sera utilisé comme un pas pour le compteur

React

Commençons par déclarer une référence

```
const pas = useRef();
```

React

Commençons par déclarer une référence

```
const pas = useRef();
```

Associons la référence à un champ nombre pour

```
<div>
    Pas : <input type='number' ref={pas} />
</div>
```

React

Utilisons le pas dans l'incrémentation et la decrémentation du compteur

```
const incrementer = () => {
    const nouvelleValeur = counter + Number(pas.current.value);
    setCounter(nouvelleValeur);
};

const decrementer = () => {
    const nouvelleValeur = counter - Number(pas.current.value);
    setCounter(nouvelleValeur);
};
```

React

Une référence peut être utilisée pour placer le curseur par exemple

```
useEffect(() => {
    if (pas.current) {
        pas.current.focus();
    }
    if (counter > 0) {
        setSigne('positif')
        setCouleur('black')
    } else if (counter < 0) {
        setSigne('négatif')
        setCouleur('tomato')
    } else {
        setSigne('nul')
        setCouleur('gold')
    }
}, [counter])
```

React

Ou pour modifier toute autre propriété JavaScript (modifier la couleur de la bordure autour de la zone de saisie)

```
useEffect(() => {
    if (pas.current) {
        pas.current.focus();
        pas.current.style.outlineColor= 'teal';
    }
    if (counter > 0) {
        setSigne('positif')
        setCouleur('black')
    } else if (counter < 0) {
        setSigne('négatif')
        setCouleur('tomato')
    } else {
        setSigne('nul')
        setCouleur('gold')
    }
}, [counter])
```

React

Exercice (À faire avec `useRef`)

- Créer un nouveau composant `CompteurClic`,
- Ajouter un bouton dans le composant,
- À chaque clic sur le bouton, afficher une `alert` avec un message indiquant le nombre total de clics.

React

Solution

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

export default function CompteurClic() {

    const count = useRef(0);

    const compterClic = () => {
        count.current++
        alert(`Vous avez cliqué ${count.current} fois`)
    }

    return (
        <div>
            <h2>Compteur de clics</h2>
            <button onClick={compterClic}>
                cliquer
            </button>
        </div>
    );
}
```

React

Affichons la valeur du compteur et vérifions qu'elle ne change pas

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

export default function CompteurClic() {

    const count = useRef(0);

    const compterClic = () => {
        count.current++
        alert(`Vous avez cliqué ${count.current} fois`)
    }

    return (
        <div>
            <h2>Compteur de clics</h2>
            <button onClick={compterClic}>
                cliquer
            </button>
            <p>Nombre de clics : {count.current}</p>
        </div>
    );
}
```

React

Rappel

Contrairement à `useState`, le changement de valeur d'une variable créée avec `useRef` ne nécessite pas de re-rendu.

React

Exercice : **primeur-produit** (Deuxième partie)

- Dans `Primeur`, ajoutez trois zones de saisie : une pour le nom, une pour le prix et une pour la quantité ainsi qu'un bouton `ajouter`.
- En cliquant sur le bouton `ajouter`, un nouveau composant `produit` s'ajoute (s'affiche dans la page) avec les trois valeurs saisies par l'utilisateur.

React

Correction : composant `Primeur` (Partie script)

```
let [produits, setProduits] = useState([
    { nom: "banane", prix: 3, quantite: 10 },
    { nom: "fraise", prix: 10, quantite: 20 },
    { nom: "poivron", prix: 5, quantite: 10 }
])

const nomRef = useRef('');
const quantiteRef = useRef(0);
const prixRef = useRef(0);

const ajouter = () => {
    const nouveauProduit = {
        nom: nomRef.current.value,
        quantite: parseInt(quantiteRef.current.value, 10),
        prix: parseFloat(prixRef.current.value)
    };
    setProduits(p => [...p, nouveauProduit]);
    viderChamps();

};

const viderChamps = () => {
    nomRef.current.value = '';
    quantiteRef.current.value = '';
    prixRef.current.value = '';
};
```

React

Remarque

- `setProduits(p => [...p, nouveauProduit])` :
  - respecte le principe de l'immuabilité,
  - consiste à ne pas modifier directement l'état, mais plutôt à créer une nouvelle version de l'état.
- `setProduits(p => p.push(nouveauProduit))` :
  - ne respecte pas l'immuabilité, car elle modifie directement l'état actuel (p).
  - peut poser problème dans **React**, car le changement de l'état peut ne pas être détecté correctement, et le composant pourrait ne pas se re-rendre comme prévu.

React

Correction : composant `Primeur` (Partie render)

```
return (
    <div>
        <h2>Primeur</h2>
        <div>
            <input type="text" ref={nomRef} placeholder='Nom' />
            <input type="number" ref={quantiteRef} placeholder='Quantité' />
            <input type="number" ref={prixRef} placeholder='Prix' />
            <button onClick={ajouter}>Ajouter</button>

        </div>
        <ul>
            {
                produits.map((elt, ind) =>
                    <Produit key={ind} elt={elt} />
                )
            }
        </ul>
    </div>
)
```

React

Correction : composant `Produit` (rien ne change)

```
export default function Produit({ elt }) {

    return (
        <li>{elt.nom} {elt.quantite} {elt.prix}</li>
    )
}
```

React

Question

Peut-on avoir une référence à un objet ?

React

Question

Peut-on avoir une référence à un objet ?

Réponse

- Avec `useRef`, il est tout à fait possible de stocker un objet et de manipuler ses propriétés.
- Mais pour les `<input>`, il n'est pas possible de référencer directement les propriétés de l'objet avec `ref={objet.champ}`
- À la place, il faut utiliser des références séparées : une pour chaque `<input>`.

React

useReducer

- utilisé pour gérer l'état complexe d'un composant
- offrant une alternative à l'utilisation de `useState` lorsque l'état comporte des structures complexes ou que les mises à jour d'état nécessitent une logique avancée.
- acceptant deux paramètres `reducer` et `initialState`
  - `initialState` : la valeur de départ de l'état géré par le `reducer`,
  - `reducer` : fonction prenant comme paramètres l'état actuel et une action et retournant un nouvel état en fonction de l'action.
- retournant deux valeurs :
  - `state` : objet contenant l'état du composant,
  - `dispatch` : utilisée pour envoyer des actions au `reducer` permettant de déclencher des mises à jour de l'état.

React

Commençons par créer l'état initial dans `Compteur.js`

```
const initialState = {
    counter: 0,
    signe: 'nul',
    couleur: 'black'
};
```

React

Commençons par créer l'état initial dans `Compteur.js`

```
const initialState = {
    counter: 0,
    signe: 'nul',
    couleur: 'black'
};
```

La fonction `reducer`

```
const reducer = (state, action) => {
    switch (action.type) {
        case 'increment':
            return { ...state, counter: state.counter + action.payload };
        case 'decrement':
            return { ...state, counter: state.counter - action.payload };
        case 'modifier':
            return { ...state, signe: action.payload.signe, couleur: action.payload.couleur };
        default:
            return state;
    }
};
```

React

Explication

- `action.type`
  - chaîne de caractères (`string`) qui décrivant la nature de l'action à effectuer.
  - utilisé dans `reducer` pour déterminer quelle logique appliquer pour mettre à jour l'état.
- `action.payload`
  - propriété optionnelle de l'action contenant des données supplémentaires nécessaires pour la mise à jour de l'état.
  - pouvant être de type nombre, chaîne, objet, tableau...

React

Question

En `React`, Pourquoi est-il essentiel de ne pas modifier l'état directement mais de toujours retourner un nouvel état ?

React

Question

En `React`, Pourquoi est-il essentiel de ne pas modifier l'état directement mais de toujours retourner un nouvel état ?

Réponse

Si l'état est modifié directement, **React** peut ne pas détecter les changements correctement.

React

Remplaçons les `useState` **par** `useReducer`

```
export default function Compteur() {
    const [state, dispatch] = useReducer(reducer, initialState);
    const pas = useRef();
};
```

React

Remplaçons les `useState` par `useReducer`

```
export default function Compteur() {
    const [state, dispatch] = useReducer(reducer, initialState);
    const pas = useRef();
};
```


Les imports

```
import { useEffect, useReducer, useRef } from 'react';
```

React

Modifions `useEffect`

```
useEffect(() => {
    if (pas.current) {
        pas.current.focus();
        pas.current.style.outlineColor = 'teal';
    }
    if (state.counter > 0) {
        dispatch({
            type: 'modifier', payload: {
                signe: 'positif',
                couleur: 'black'
            }
        })
    } else if (state.counter < 0) {
        dispatch({
            type: 'modifier', payload: {
                signe: 'négatif',
                couleur: 'tomato'
            }
        })
    } else {
        dispatch({
            type: 'modifier', payload: {
                signe: 'nul',
                couleur: 'gold'
            }
        })
    }
}, [state.counter])
```

React

Et `incrementer` **et** `decrementer` **en utilisant** `dispatch`

```
const incrementer = () => {
    dispatch({ type: 'increment', payload: Number(pas.current.value) })
};

const decrementer = () => {
    dispatch({ type: 'decrement', payload: Number(pas.current.value) })
};
```

React

Et `incrementer` et `decrementer` en utilisant `dispatch`

```
const incrementer = () => {
    dispatch({ type: 'increment', payload: Number(pas.current.value) })
};

const decrementer = () => {
    dispatch({ type: 'decrement', payload: Number(pas.current.value) })
};
```

Et le rendu

```
return (
    <div>
        <h1>Compteur </h1>
        <div>
            <input type='number' ref={pas} />
        </div>
        <div style={{ color: state.couleur }}>
            <button onClick={decrementer}>-</button>
            {state.counter} ({state.signe})
            <button onClick={incrementer}>+</button>
        </div>
    </div>
);
```

React

Objectif

Transmettre de données depuis un composant enfant vers un composant parent.

React

Dans `Hello`, ajoutons le contenu suivant

```
<div>
    <label htmlFor="pays">Pays</label>
    <input type="text" id="pays" ref={pays} />
    <button onClick={envoyer}>
        Envoyer
    </button>
</div>
```

React

Dans `Hello`, ajoutons le contenu suivant

```
<div>
    <label htmlFor="pays">Pays</label>
    <input type="text" id="pays" ref={pays} />
    <button onClick={envoyer}>
        Envoyer
    </button>
</div>
```


Sans oublier de déclarer `pays` comme référence

```
const pays = useRef('')
```

React

Et la fonction `envoyer`

```
function envoyer() {
    onValeurChange(pays.current.value)
}
```

React

Et la fonction `envoyer`

```
function envoyer() {
    onValeurChange(pays.current.value)
}
```

La fonction `onValeurChange` **est à déclarer comme** `props` **du composant** `Hello`

```
export default function Hello({ children, nom, onValeurChange }) {

}
```

React

Explication

- En cliquant sur le bouton, le composant enfant (`Hello`) émet un évènement à son parent (`App`).
- Par conséquent, l'évènement `onValeurChange` (à définir dans la balise `Hello` de `App` déclenchera l'exécution d'une fonction à définir dans `App`.
- Le `pays` sera envoyé comme paramètre.

React

Dans `App`, ajoutons un évènement de type `onValeurChange` (comme attribut du composant `Hello`) qui déclenchera la fonction `updateValue()`

```
function App() {
  let [country, setCountry] = useState(null)
  function updateValue(value) {
    setCountry(value)
  }
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React in {country}
        </a>
      </header>
      <Hi nom="Wick">Marseille</Hi>
      <Hello nom="Wick" onValeurChange={updateValue}>Paris</Hello>
    </div>
  );
}
```

React

Exercice 1 : **primeur-produit**

- Dans `Primeur`, déclarez une variable `total`.
- Dans `Produit`, ajoutez une zone de saisie et un bouton.
- En choisissant une quantité et appuyant sur le bouton, le total sera mis à jour et le bouton sera désactivé.

React

Solution (`Produit.js`)

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

export default function Produit({ elt, onSendQuantite }) {
    const quantite = useRef(0)
    const [disabled, setDisabled] = useState(false)

    const ajouter = () => {
        onSendQuantite(quantite.current.value)
        setDisabled(true)
    }
    return (
        <li>
            {elt.nom} {elt.quantite} {elt.prix}
            <input type="number" ref={quantite}/>
            <button onClick={ajouter} disabled={disabled}>
                Ajouter
            </button>
        </li>
    )
}
```

React

Solution (`Primeur.js` : ce qu'il faut ajouter dans le script)

```
let [total, setTotal] = useState(0)

const calculerTotal = (qte, ind) => {
    setTotal(tot => tot + qte * produits[ind].prix)
}
```

React

Solution (`Primeur.js` : nouveau render)

```
return (
    <div>
        <h2>Primeur : {total}</h2>
        <div>
            <input type="text" ref={nomRef} placeholder='Nom' />
            <input type="number" ref={quantiteRef} placeholder='Quantité' />
            <input type="number" ref={prixRef} placeholder='Prix' />
            <button onClick={ajouter}>Ajouter</button>
        </div>
        <ul>
            {
                produits.map((elt, ind) =>
                    <Produit
                    key={ind} elt={elt}
                    onSendQuantite={(qte) => calculerTotal(qte, ind)} />
                )
            }
        </ul>
    </div>
)
```

React

Exercice 2 : **compteur-controls**

- Créer un composant `Controls`
- Déplacer les deux boutons `+` et `−` depuis `Compteur` vers `Controls`
- Faire le nécessaire pour que les boutons continuent à modifier la valeur du compteur en fonction du pas.

React

Solution (`Controls.js`)

```
const Controls = ({  incrementer, decrementer }) => {
    return (
        <div>
            <button onClick={decrementer}>-</button>
            <button onClick={incrementer}>+</button>
        </div>
    );
};

export default Controls;
```

React

Nouveau render de `Compteur.js` (le reste ne change pas)

```
return (
    <div>
        <h1>Compteur </h1>
        <div>
            <input type='number' ref={pas} />
        </div>
        <div style={{ color: state.couleur }}>
            {state.counter} ({state.signe})
            <Controls
            pas={pas}
            incrementer={incrementer}
            decrementer={decrementer} />
        </div>
    </div>
);
```

React

Exercice 3 : **clavier-touche** (Simulation d'un clavier virtuel)

- Créez deux composants `Clavier` et `Touche`.
- Le composant `Clavier` a un attribut `lettres` (voir ci-dessous)
- le composant `Touche` a un attribut `value` recevant une valeur du tableau `lettres` (un composant fils `Touche` pour chaque valeur du tableau `lettres`).
- Chaque `Touche` affiche la lettre qu'il a reçue sur un bouton.
- En cliquant sur ce bouton, la lettre s'affiche (à la suite des autres) dans une balise `textarea` définie dans le composant `Clavier`.

Contenu du tableau `lettres`

```
let lettres = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', '
  l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
   'z']
```

React

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

React

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

Première solution

Chaque parent transmet les données à ses enfants, les enfants utiliseront `props` pour récupérer les données (qu'ils devront, à leur tour, les transmettre à leurs enfants...)

React

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

Première solution

Chaque parent transmet les données à ses enfants, les enfants utiliseront `props` pour récupérer les données (qu'ils devront, à leur tour, les transmettre à leurs enfants...)

Deuxième solution

Utiliser le hook `useContext`.

React

Exercice

- Créer un composant `Prix`
- Le composant `Prix` est l'enfant de `Produit` qui est lui même l'enfant de `Primeur`.
- Le composant `Primeur` fournit la valeur de la `TVA` aux composants `Produit`.
- Le composant `Produit` fournit la valeur reçue de la `TVA` aux composants `Prix`.
- Le composant `Prix` reçoit la `TVA` et affiche le prix HT et TTC de chaque produit.

React

Solution (`Primeur.js`)

```
import { useState } from 'react'
import Produit from './Produit'

export default function Primeur() {
    let produits = [
        { nom: "banane", prix: 3, quantite: 10 },
        { nom: "fraise", prix: 10, quantite: 20 },
        { nom: "poivron", prix: 5, quantite: 10 }
    ]
    let [total, setTotal] = useState(0)
    const tva = 20
    const calculerTotal = (qte, ind) => {
        setTotal(tot => tot + qte * produits[ind].prix)
    }
    return (
        <div>
            <h2>Primeur : {total}</h2>
            <ul>
                {
                    produits.map((elt, ind) =>
                        <Produit key={ind} tva={tva} elt={elt} onSendQuantite={(qte) =>
                          calculerTotal(qte, ind)} />
                    )
                }
            </ul>
        </div>
    )
}
```

React

Solution (`Produit.js`)

```
import { useRef, useState } from "react"
import Prix from './Prix'

export default function Produit({ elt, onSendQuantite, tva }) {

    const quantite = useRef(0)
    const [disabled, setDisabled] = useState(false)

    const ajouter = () => {
        onSendQuantite(quantite.current.value)
        setDisabled(true)
    }

    return (
        <li>
            {elt.nom} {elt.quantite} <Prix prixHt={elt.prix} tva={tva} />
            <input type="number" ref={quantite} />
            <button onClick={ajouter} disabled={disabled} >
                Ajouter
            </button>
        </li>
    )
}
```

React

Solution (`Prix.jsx`)

```
export default function Prix({tva, prixHt}) {

    const prixTtc = prixHt + (prixHt * tva)/100

    return (
        <span>
            {prixTtc} TTC (prix HT : {prixHt}, TVA : {tva}%)
        </span>
    )
}
```

React

Deuxième solution : utilisation du contexte

Trois étapes principales

1. **Créer un contexte** : cela fournit à la fois un `Provider` et un `Consumer` (utilisé avec le hook `useContext`).
2. **Fournir le contexte** : permet de définir la partie de composant où la valeur du contexte est accessible. La valeur est partagée via la prop `value`.
3. **Utiliser le contexte** : avec le hook `useContext` pour accéder à la valeur du contexte dans un composant enfant.

React

Dans `Primeur.js`, on commence par créer le contexte

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

export const TvaContext = createContext();
```

React

Dans `Primeur.js`, on commence par créer le contexte

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

export const TvaContext = createContext();
```

Dans le render du composant `Primeur`, on fournit la valeur de `tva` à tous les composants enfants

```
<TvaContext.Provider value={tva}>
<ul>
    {produits.map((elt, ind) =>
        <Produit
        key={ind}
        elt={elt}
        onSendQuantite={(qte) => calculerTotal(qte, ind)} />
    )}
</ul>
</TvaContext.Provider>
```

React

Remarque

La valeur de `tva` n'est plus envoyée comme `props` au composant enfant `Produit`.

React

Plus de `tva` **dans** `Produit.js`

```
import { useRef, useState } from "react"
import Prix from './Prix'

export default function Produit({ elt, onSendQuantite }) {
    const quantite = useRef(0)
    const [disabled, setDisabled] = useState(false)

    const ajouter = () => {
        onSendQuantite(quantite.current.value)
        setDisabled(true)
    }
    return (
        <li>
            {elt.nom} {elt.quantite} <Prix prixHt={elt.prix}  />
            <input type="number" ref={quantite}/>
            <button onClick={ajouter} disabled={disabled}>
                Ajouter
            </button>
        </li>
    )
}
```

React

Dans `Prix.js`, on utilise `useContext` pour récupérer la `tva`

```
import { useContext } from 'react';
import { TvaContext } from './Primeur';

export default function Prix({ prixHt }) {

    const tva = useContext(TvaContext);
    const prixTtc = prixHt + (prixHt * tva) / 100;

    return (
        <span>
            {prixTtc} TTC (prix HT : {prixHt}, TVA : {tva}%)
        </span>
    );
}
```

React

Une valeur par défaut peut être spécifiée à la création du contexte

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

export const TvaContext = createContext(20);
```

React

Une valeur par défaut peut être spécifiée à la création du contexte

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

export const TvaContext = createContext(20);
```

Le `Provider` n'est plus nécessaire dans le render

```
<ul>
    {produits.map((elt, ind) =>
        <Produit
        key={ind}
        elt={elt}
        onSendQuantite={(qte) => calculerTotal(qte, ind)} />
    )}
</ul>
```

React

Remarques

- La constante `tva` n'est plus nécessaire dans `Primeur` après avoir spécifiée la valeur par défaut.
- La valeur est désormais accessible à tous les enfants (et pas seulement ceux qui sont définis dans le `Provider`).

React

Rien à changer dans `Prix.js`

```
import { useContext } from 'react';
import { TvaContext } from './Primeur';

export default function Prix({ prixHt }) {

    const tva = useContext(TvaContext);
    const prixTtc = prixHt + (prixHt * tva) / 100;

    return (
        <span>
            {prixTtc} TTC (prix HT : {prixHt}, TVA : {tva}%)
        </span>
    );
}
```

React

Exercice : **ContextGlobal**

- Dans `src/context/GlobalContext.js`, créez un contexte et un `Provider` globaux.
- L'objectif est que tous les composants puissent interagir via ce contexte global.
- Déclarez une variable `msg`, partagez-la via le contexte global et vérifiez que les composants aient accès à cette variable.

React

Contenu de `src/context/GlobalContext.js`

```
import { createContext } from "react";

export const GlobalContext = createContext()

const msg = 'Hello world'

export const Provider = ({ children }) => {
    return (
        <GlobalContext.Provider value={{ msg }}>
            {children}
        </GlobalContext.Provider>
    )
}
```

React

Déclarons `GlobalContext` dans `main.jsx` pour l'utiliser globalement

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { Provider } from './contexts/GlobalContext';


const root = createRoot(document.getElementById('root'))
root.render(
  <StrictMode>
    <Provider>
      <App />
    </Provider>
  </StrictMode>,
)
```

React

Dans n'importe quel composant, commençons par importer le `Provider`

```
import { useContext } from 'react';
import { GlobalContext } from '../contexts/GlobalContext';
```

React

Dans n'importe quel composant, commençons par importer le `Provider`

```
import { useContext } from 'react';
import { GlobalContext } from '../contexts/GlobalContext';
```

Utilisons ensuite le contexte

```
const { msg } = useContext(GlobalContext)
```

React

Dans n'importe quel composant, commençons par importer le `Provider`

```
import { useContext } from 'react';
import { GlobalContext } from '../contexts/GlobalContext';
```

Utilisons ensuite le contexte

```
const { msg } = useContext(GlobalContext)
```

Le `msg` peut être rendu

```
<p>
    { msg }
</p>
```

React

Exercice : partie 2

- Créer trois composants `Container`, `First` et `Second`.
- `First` et `Second` sont les enfants de `Container`.
- Utilisez `GlobalContext` pour faire communiquer `First` et `Second` de la manière suivante :
  - Dans `First`, ajoutez une zone de saisie `input:text` et un bouton.
  - Dans `second`, ajoutez une liste **HTML** dans laquelle vous affichez les messages envoyez par `First`.
  - Utilisez `useReducer` et `useState` dans `GlobalContext` pour faire communiquer `First` et `Second`.
  - Pensez à partager `state` et `dispatch` dans `Provider`.

React

Contenu de `GlobalContext` (Solution avec useReducer)

```
import { createContext, useReducer } from 'react';

export const GlobalContext = createContext();

const initialState = {
    messages: []
};

const reducer = (state, action) => {
    switch (action.type) {
        case 'ADD_MESSAGE':
            return { ...state, messages: [...state.messages, action.payload] };
        default:
            return state;
    }
};

export const Provider = ({ children }) => {
    const [state, dispatch] = useReducer(reducer, initialState);

    return (
        <GlobalContext.Provider value={{ state, dispatch }}>
            {children}
        </GlobalContext.Provider>
    );
};
```

React

Le composant `First`

```
import { useState, useContext } from 'react';
import { GlobalContext } from '../contexts/GlobalContext'

const First = () => {

    const [inputValue, setInputValue] = useState('');
    const { dispatch } = useContext(GlobalContext);

    const handleSubmit = () => {
        dispatch({ type: 'ADD_MESSAGE', payload: inputValue });
        setInputValue('');
    };

    return (
        <div>
            <input
                type="text"
                value={inputValue}
                onInput={(e) => setInputValue(e.target.value)}
            />
            <button onClick={handleSubmit}>Envoyer</button>
        </div>
    );
};

export default First;
```

React

Le composant `Second`

```
import { useContext } from 'react';
import { GlobalContext } from '../contexts/GlobalContext'

const Second = () => {

    const { state } = useContext(GlobalContext);

    return (
        <ul>
            {state.messages.map((message, index) => (
                <li key={index}>{message}</li>
            ))}
        </ul>
    );
};

export default Second;
```

React

Le composant `Container`

```
import First from './First';
import Second from './Second';

const Container = () => {
    return (
        <div>
            <h2>Conteneur</h2>
            <First />
            <Second />
        </div>
    );
};

export default Container;
```

React

Le composant `Container`

```
import First from './First';
import Second from './Second';

const Container = () => {
    return (
        <div>
            <h2>Conteneur</h2>
            <First />
            <Second />
        </div>
    );
};

export default Container;
```

N'oubliez pas d'ajouter `Container` dans le `render` du composant `App`.

React

Contenu de `GlobalContext` (Solution sans useReducer)

```
import { createContext, useReducer } from 'react';

export const GlobalContext = createContext()


export const Provider = ({ children }) => {

    const [textes, setTextes] = useState([])

    function ajouterTexte(texte) {
        setTextes([...textes, texte])
    }

    return (
        <GlobalContext.Provider value={{ textes, ajouterTexte }}>
            {children}
        </GlobalContext.Provider>
    );
};
```

React

Le composant `First`

```
import { useContext, useRef } from "react"
import { GlobalContext } from "../contexts/GlobalContext"

export default function First() {
    const { ajouterTexte } = useContext(GlobalContext)
    const texte = useRef()
    function envoyer() {
        ajouterTexte(texte.current.value)
        texte.current.value = ''
    }
    return (
        <div className="col-6">
            <h3>First</h3>
            <div>
                <input type="text" ref={texte} placeholder="Ton texte par ici" />
                <button onClick={envoyer}>
                    Envoyer
                </button>
            </div>
        </div>
    )
}
```

React

Le composant `Second`

```
import { useContext } from "react"
import { GlobalContext } from "../contexts/GlobalContext"

export default function Second() {
    const { textes } = useContext(GlobalContext)
    return (
        <div className="col-6">
            <h3>Second</h3>
            <ul>
                {
                    textes.map((t, ind) =>
                        <li key={ind}>{t}</li>
                    )
                }
            </ul>
        </div>
    )
}
```

React

Le composant `Container`

```
import First from './First';
import Second from './Second';

const Container = () => {
    return (
        <div>
            <h2>Conteneur</h2>
            <First />
            <Second />
        </div>
    );
};

export default Container;
```

React

Exercice

- On veut que l'échange entre deux composants soit bidirectionnel.
- Quand le premier envoie, le deuxième affiche et inversement.

React

Exercice

- Créez deux composant `tchat` et `participant`
- Définissez une route `/tchat` pour le composant `tchat`
- Le composant `tchat` contient un champ texte `nom` et un bouton `ajouter` qui permet d'ajouter un nouveau composant `participant` dans `tchat`
- Chaque composant `participant` peut envoyer des messages à tous les autres participants
- Les participants reçoivent immédiatement les messages envoyés et les affichent
- Un participant n'affiche pas ses propres messages

Considérons le composant `List` suivant

```
import { useState } from "react";
import Items from "./Items";


export default function List() {
    const [nombre, setNombre] = useState(1)
    const [couleur, setCouleur] = useState(true)
    const theme = {
        color: couleur ? 'black' : 'white',
        backgroundColor: couleur ? 'white' : 'black'
    }

    const getItems = () => {
        return [nombre - 1, nombre, nombre + 1]
    }

    return (
        <div style={theme}>
            <h1>Liste-items</h1>
            <input
                type="number"
                value={nombre}
                onChange={e => setNombre(parseInt(e.target.value))}
            />
            <button onClick={() => setCouleur(couleur => !couleur)}>
                Changer thème
            </button>
            <Items getItems={getItems} />
        </div>
    )
}
```

React

Exercice

- Créer le composant `Items`
- Utiliser `useState` et `useEffect` pour afficher et mettre à jour les trois valeurs transmises par le composant `List`

Une première solution

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

export default function Items({ getItems }) {
    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems())
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Une première solution

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

export default function Items({ getItems }) {
    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems())
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Cliquez sur le bouton `Changer thème` et vérifiez que le message `Items` s'affiche.

React

useCallback(callback, [dépendances])

- renvoie une fonction de rappel mémorisée
- ne s'exécute que si une de ses dépendances est mise à jour
- permet d'améliorer les performances en empêchant l'exécution de certaines fonctions à chaque rendu

Utilisons `useCallback` **dans** `List`

```
import { useCallback, useState } from "react";
import Items from "./Items";


export default function List() {

    const [nombre, setNombre] = useState(1)
    const [couleur, setCouleur] = useState(true)

    const theme = {
        color: couleur ? 'black' : 'white',
        backgroundColor: couleur ? 'white' : 'black'
    }

    const getItems = useCallback(() => {
        return [nombre - 1, nombre, nombre + 1]
    }, [nombre])
    return (
        <div style={theme}>
            <h1>Liste-items</h1>
            <input
                type="number"
                value={nombre}
                onChange={e => setNombre(parseInt(e.target.value))}
            />
            <button onClick={() => setCouleur(couleur => !couleur)}>
                Changer thème
            </button>
            <Items getItems={getItems} />
        </div>
    )
}
```

Rien ne change dans `Items`

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

export default function Items({getItems}) {

    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems())
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Rien ne change dans `Items`

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

export default function Items({getItems}) {

    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems())
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Cliquez sur le bouton `Changer thème` et vérifiez que le message `Items` ne s'affiche plus.

React

useMemo(callback, [dépendances])

- évalue la fonction passée et mémorise son résultat.
- fonctionne de manière similaire à `useCallback`, mais avec des différences importantes.
  - `useMemo` retourne une valeur mémorisée au lieu d'une fonction (callback).
  - Si les dépendances n'ont pas changé entre les rendus, `useMemo` renverra la valeur mémorisée précédente, évitant ainsi des calculs inutiles.
- permet d'optimiser les performances en mémorisant le résultat d'une fonction coûteuse tant que les dépendances n'ont pas changé.

Remplaçons `useCallback` **par** `useMemo` **dans** `List`

```
import { useMemo, useState } from "react";
import Items from "./Items";


export default function List() {

    const [nombre, setNombre] = useState(1)
    const [couleur, setCouleur] = useState(true)

    const theme = {
        color: couleur ? 'black' : 'white',
        backgroundColor: couleur ? 'white' : 'black'
    }

    const getItems = useMemo(() => {
        return [nombre - 1, nombre, nombre + 1]
    }, [nombre])
    return (
        <div style={theme}>
            <h1>Liste-items</h1>
            <input
                type="number"
                value={nombre}
                onChange={e => setNombre(parseInt(e.target.value))}
            />
            <button onClick={() => setCouleur(couleur => !couleur)}>
                Changer thème
            </button>
            <Items getItems={getItems} />
        </div>
    )
}
```

Remplaçons `getItems()` **par** `getItems` **dans** `Items.js`

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

export default function Items({ getItems }) {

    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems)
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Remplaçons `getItems()` **par** `getItems` **dans** `Items.js`

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

export default function Items({ getItems }) {

    const [items, setItems] = useState([])

    useEffect(() => {
        console.log('Items');
        setItems(getItems)
    }, [getItems])

    return (
        <ul>
            {
                items.map(item => <li key={item}>{item}</li>)
            }
        </ul>
    )
}
```

Cliquez sur le bouton `Changer thème` et vérifiez que le message `Items` ne s'affiche pas.

React

Remarques

- Utiliser `useMemo` pour mémoriser le résultat d'une fonction.
- Utiliser `useCallback` pour mémoriser une fonction, surtout si vous la passez en `props` à des composants enfants.

React

Exercice

- Créer un composant `Inscription.js`
- Ajouter deux `input` : un pour le nom d'utilisateur et un pour le mot de passe.
- Définir une fonction `motDePasseSecurite` qui retourne `Vide` si le mot de passe est vide, `Court` s'il contient moins de 4 caractère, `Correct sinon`.
- Appeler la fonction à l'initialisation du composant et stocker sa valeur de retour dans une variable appelée `securite`.
- Afficher la variable `securite` à côté du champ de saisie du mot de passe.
- Vérifier que la fonction `motDePasseSecurite` est exécutée même lors de la modification du nom d'utilisateur.
- Utiliser `useMemo` pour éviter le comportement précédent.

Remplaçons `getItems()` **par** `getItems` **dans** `Items.js`

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

export default function Inscription() {
    let [nom, setNom] = useState('')
    let [moteDePasse, setMotDepasse] = useState('')
    let securite = moteDepasseSecurite(moteDePasse)
    return (
        <div>
            <h2>Inscription</h2>
            <div>
                Nom d'utilisateur<input value={nom} onChange={e => setNom(e.target.value)} />
            </div>
            <div>
                Mot de passe<input value={moteDePasse} onChange={e => setMotDepasse(e.target.
                   value)} />
                <span>{securite}</span>
            </div>

        </div>
    )
}

function moteDepasseSecurite(param) {
    console.log(param);
    if (!param) {
        return "Vide"
    }
    if (param.length< 5) {
        return "Court"
    }
    return "Correct"
}
```

React

Règle générale

Tout ce qui commence par `use` en **React** est un **hook**.

React

Hook personnalisé

- Une fonction **JavaScript** dont le nom commence par `use`
- Pouvant contenir des **hooks** prédéfinis (comme `useState`, `useEffect`...)
- Ne devant pas être utilisés dans boucle, structure conditionnelle...
- Permettant de
  - réutiliser de la logique commune entre plusieurs composants
  - alléger le code des composants

React

Commençons par créer le hook `useCount` dans `src/hooks`

```
export default function useCount(initialValue = 0) {

}
```

React

On peut utiliser des hooks prédéfinis dans un hook personnalisé

```
import { useState } from "react"

export default function useCount(initialValue = 0) {

    const [count, setCount] = useState(initialValue)

}
```

React

Un hook personnalisé doit retourner les éléments à utiliser (sous forme d'un tableau, objet ou valeur primitive)

```
import { useState } from "react"

export default function useCount(initialValue = 0) {

    const [count, setCount] = useState(initialValue)

    return {
        count,
        increment: () => setCount(val => val + 1),
        decrement: () => setCount(val => val - 1)
    }
}
```

React

Pour l'utiliser, il faut commencer par l'importer

```
import useCount from './hooks/useCount';
```

React

Pour l'utiliser, il faut commencer par l'importer

```
import useCount from './hooks/useCount';
```

Ensuite déstructurer ce qu'on a besoin d'utiliser

```
const { count, increment, decrement } = useCount()
```

React

Pour l'utiliser, il faut commencer par l'importer

```
import useCount from './hooks/useCount';
```

Ensuite déstructurer ce qu'on a besoin d'utiliser

```
const { count, increment, decrement } = useCount()
```


Et enfin l'utiliser

```
<h1>Compteur : {count}</h1>
<button onClick={increment}>+</button>
<button onClick={decrement}>-</button>
```

React

Les fonctions définies dans un hook personnalisé peuvent prendre des paramètres

```
import { useState } from "react"

export default function useCount(initialValue = 0) {

    const [count, setCount] = useState(initialValue)

    return {
        count,
        increment: (step) => setCount(val => val + Number(step)),
        decrement: (step) => setCount(val => val - Number(step))
    }
}
```

React

Pour l'utiliser, il faut définir une référence

```
const step = useRef(0)
```

React

Pour l'utiliser, il faut définir une référence

```
const step = useRef(0)
```

Et enfin l'utiliser

```
<h1>Compteur : {count}</h1>
<input type='number' ref={step} defaultValue={step.current} />
<button onClick={() => increment(step.current.value)}>+</button>
<button onClick={() => decrement(step.current.value)}>-</button>
```

React.js

Question 1

`defaultValue` ou `value` ?

React.js

Question 1

`defaultValue` ou `value` ?

`defaultValue`

- Utilisé pour initialiser la valeur d'un élément de formulaire non contrôlé.
- Une fois le composant rendu, `defaultValue` n'affecte plus l'élément. Si l'utilisateur modifie la valeur, le composant ne se réinitialise pas.
- Ne nécessite pas de gestion d'événements comme `onChange` ou `onInput`.

`value`

- Utilisé dans les composants contrôlés, où la valeur de l'input est entièrement contrôlée par l'état **React**.
- `value` est associé à une variable d'état, et tout changement doit être géré via un événement `onChange` ou `onInput`.

React

Question 2

Pourquoi avons-nous utilisé `step.current` pour l'initialisation et `step.current.value` lors de l'événement du bouton ?

React

Question 2

Pourquoi avons-nous utilisé `step.current` pour l'initialisation et `step.current.value` lors de l'événement du bouton ?

`step.current`

- Avant le premier rendu, elle vaut la `initialValue` (spécifiée dans `useRef(initialValue)`).
- Après le premier rendu, si la référence `step` est attachée à un élément **DOM** (comme un champ de formulaire), **React** mettra à jour `step.current` pour qu'il pointe vers l'élément **DOM** correspondant (par exemple, un objet **HTMLInputElement**).
- Pour accéder à la valeur de la référence `step` après le premier rendu, il faut utiliser `step.current.value`.

React

On peut aussi utiliser `useRef` sans valeur initiale

```
const step = useRef()
```

React

On peut aussi utiliser `useRef` sans valeur initiale

```
const step = useRef()
```

Et la spécifier statiquement dans `defaultValue`

```
<h1>Compteur : {count}</h1>
<input type='number' ref={step} defaultValue={0} />
<button onClick={() => increment(step.current.value)}>+</button>
<button onClick={() => decrement(step.current.value)}>-</button>
```

React

Le hook personnalisé peut accepter plusieurs paramètres

```
import { useState } from "react"

export default function useCount({ initialValue = 0, maxValue = Infinity, minValue = -Infinity
   }) {

    const [count, setCount] = useState(initialValue)

    return {
        count,
        increment: (step) => setCount(val => val < maxValue ? val + Number(step) : val),
        decrement: (step) => setCount(val => val > minValue ? val - Number(step) : val)
    }
}
```

React

Ainsi, nous pourrons

```
const { count, increment, decrement } = useCount({
  maxValue: 3,
  minValue: 0
})
```

React

Quelques sites proposant des hooks

- `https://usehooks.com/`
- `https://github.com/streamich/react-use`
- `https://usehooks-ts.com/`

React

HOC : **H**igher-**O**rder **C**omponents

- Une fonction qui peut prendre en paramètre un composant et retourne un nouveau composant enrichi.
- Initialement conçus pour être utilisés avec les composants de classe, mais ils fonctionnent également très bien avec les composants fonctionnels.
- Possibilité d'utiliser des **hooks** comme `useState` ou `useEffect` dans les **HOC**.

React

Considérons le composant `Frontend` suivant

```
export function Frontend({ framework }) {

    const frameworks = ['Angular', 'Vue.js', 'React', 'Svelte']

    if (!frameworks.includes(framework)) {
        return (
            <div>
                <h2>Erreur</h2>
                <p>{`${framework} n'est pas utilisé comme framework frontend`}</p>
            </div>
        );
    }
    return (
        <div>
            <h2>Frontend</h2>
            <p>{`Le framework ${framework} est utilisé comme framework frontend`}</p>
        </div>
    )
}
```

React

Et le composant `Backend`

```
export function Backend({ framework }) {

    const frameworks = ['Spring', 'Symfony', 'Laravel', 'Struts', 'Django']

    if (!frameworks.includes(framework)) {
        return (
            <div>
                <h2>Erreur</h2>
                <p>{`${framework} n'est pas utilisé comme framework backend`}</p>
            </div>
        );
    }
    return (
        <div>
            <h2>Backend</h2>
            <p>{`${framework} est utilisé comme framework backend`}</p>
        </div>
    )
}
```

React

Pour tester

```
<Backend framework='Spring' />
<Frontend framework='Spring' />
<Backend framework='React' />
<Frontend framework='React' />
```

React

Constant

Les deux composants `Frontend` et `Backend` sont quasi-identiques.

React

Constant

Les deux composants `Frontend` et `Backend` sont quasi-identiques.

Question

Comment factoriser le code de ces deux composants ?

React

Constant

Les deux composants `Frontend` et `Backend` sont quasi-identiques.

Question

Comment factoriser le code de ces deux composants ?

Réponse

Utiliser les **HOC**.

Commençons par créer le composant HOC `withFrameworkValidation`

```
export function withFrameworkValidation(frameworks, componentType) {
    return ({ framework }) => {
        if (!frameworks.includes(framework)) {
            return (
                <div>
                    <h2>Erreur</h2>
                    <p>{`${framework} n'est pas utilisé comme framework ${componentType}`}</p>
                </div>
            );
        }
        return (
            <div>
                <h2>{componentType.charAt(0).toUpperCase() + componentType.slice(1)}</h2>
                <p>{`${framework} est utilisé comme framework ${componentType}`}</p>
            </div>
        );
    };
}
```

Commençons par créer le composant HOC `withFrameworkValidation`

```
export function withFrameworkValidation(frameworks, componentType) {
    return ({ framework }) => {
        if (!frameworks.includes(framework)) {
            return (
                <div>
                    <h2>Erreur</h2>
                    <p>{`${framework} n'est pas utilisé comme framework ${componentType}`}</p>
                </div>
            );
        }
        return (
            <div>
                <h2>{componentType.charAt(0).toUpperCase() + componentType.slice(1)}</h2>
                <p>{`${framework} est utilisé comme framework ${componentType}`}</p>
            </div>
        );
    };
}
```


Explication

- Par convention, le nom d'un **HOC** commence par `with`
- Le **Hoc** retourne un composant contenant la logique métier de `Backend` et `Frontend`

React

Nouveau contenu du composant `Backend`

```
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Backend = withFrameworkValidation(
    ['Spring', 'Symfony', 'Laravel', 'Struts', 'Django'],
    'backend'
);
```

React

Nouveau contenu du composant `Backend`

```
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Backend = withFrameworkValidation(
    ['Spring', 'Symfony', 'Laravel', 'Struts', 'Django'],
    'backend'
);
```

Et le nouveau contenu du composant `Frontend`

```
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Frontend = withFrameworkValidation(
    ['Angular', 'Vue.js', 'React', 'Svelte'],
    'frontend'
);
```

React

Rien à changer pour tester

```
<Backend framework='Spring' />
<Frontend framework='Spring' />
<Backend framework='React' />
<Frontend framework='React' />
```

React

Dans `withFrameworkValidation`, nous pourrons également utiliser un composant intermédiaire (`FrameworkInfo`)

```
export function withFrameworkValidation(Component, frameworks, componentType) {
    return ({ framework }) => {
        if (!frameworks.includes(framework)) {
            return (
                <div>
                    <h2>Erreur</h2>
                    <p>{`${framework} n'est pas utilisé comme framework ${componentType}`}</p>
                </div>
            );
        }
        return <Component framework={framework} componentType={componentType} />
    };
}
```

React

Dans `withFrameworkValidation`, nous pourrons également utiliser un composant intermédiaire (`FrameworkInfo`)

```
export function withFrameworkValidation(Component, frameworks, componentType) {
    return ({ framework }) => {
        if (!frameworks.includes(framework)) {
            return (
                <div>
                    <h2>Erreur</h2>
                    <p>{`${framework} n'est pas utilisé comme framework ${componentType}`}</p>
                </div>
            );
        }
        return <Component framework={framework} componentType={componentType} />
    };
}
```


Contenu du composant `FrameworkInfo`

```
export function FrameworkInfo({ framework, componentType }) {
    return (
        <div>
            <h2>{componentType.charAt(0).toUpperCase() + componentType.slice(1)}</h2>
            <p>{`${framework} est utilisé comme framework ${componentType}`}</p>
        </div>
    );
}
```

React

Utilisons `FrameworkInfo` **dans** `Backend`

```
import { FrameworkInfo } from "./FrameworkInfo";
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Backend = withFrameworkValidation(
    (props) => <FrameworkInfo {...props}  />,
    ['Spring', 'Symfony', 'Laravel', 'Struts', 'Django'],
    'backend'
);
```

React

Utilisons `FrameworkInfo` **dans** `Backend`

```
import { FrameworkInfo } from "./FrameworkInfo";
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Backend = withFrameworkValidation(
    (props) => <FrameworkInfo {...props}  />,
    ['Spring', 'Symfony', 'Laravel', 'Struts', 'Django'],
    'backend'
);
```

Et dans `Frontend`

```
import { FrameworkInfo } from "./FrameworkInfo";
import { withFrameworkValidation } from "./withFrameworkValidation";

export const Frontend = withFrameworkValidation(
    (props) => <FrameworkInfo {...props} />,
    ['Angular', 'Vue.js', 'React', 'Svelte'],
    'frontend'
);
```

React

Rien à changer pour tester

```
<Backend framework='Spring' />
<Frontend framework='Spring' />
<Backend framework='React' />
<Frontend framework='React' />
```

React.js

HOC : avantages

- Permet une meilleure composition et réutilisation de code.
- Facilite la séparation des préoccupations.
- Évite les problèmes d'héritage multiples ou complexes.

HOC : inconvénients

- Peut entraîner des problèmes de performance si mal utilisé.
- Peut rendre le code plus difficile à lire, surtout s'il y a de nombreux **HOC**s imbriqués.

React.js

Bonnes pratiques

- Évitez d'utiliser la syntaxe spread (`{...props}`) sauf si vous êtes certain des **props** qui sont passées. Cela peut introduire des **props** non désirées ou incorrectes.
- Nommez les **props** de manière explicite pour qu'elles décrivent clairement leur usage.
- N'appelez jamais les **hooks** dans des boucles, des conditions ou des fonctions imbriquées.
- Si vous créez un **hook** personnalisé, utilisez le préfixe use pour indiquer clairement que c'est un **hook**.
- Assurez-vous que toutes les variables ou fonctions utilisées dans `useEffect` sont présentes dans le tableau de dépendances.
- Par convention, préfixez les noms de **HOC**s par `with`. Par exemple : `withAuth`, `withLogging`...
- Ne créez pas de **HOC** dynamiquement à l'intérieur de la méthode `render` : Cela entraîne la création d'un nouveau composant à chaque rendu.