Les Fragments

Annexes

Les fragments

Le rendu d'un composant React doit toujours être composé d'un seul noeud racine. Pour respecter cela, il est possible d'ajouter une balise « div » pour regrouper nos différents éléments sous un même noeud.

Cela peut provoquer des problèmes :

- Un excès de « div » dû à de multiples composants imbriqués.
- Casser la sémantique du HTML
  Exemple : Une balise « div » dans la structure de table

Les fragments

Les Fragments permettent de regrouper plusieurs éléments sans ajouter de noeud supplémentaire au sein du DOM.

```
import React, {Fragment} from "react";

export default function DemoFragment(props) {
    return (
        <Fragment>
            <MyComponent />
            <MyComponent />
        </Fragment>
    );
}
```

Les fragments

Il est possible d'utiliser les fragments avec la syntaxe raccourci « <> ... </> »

```
import React from "react";

export default function DemoFragment(props) {
    return (
        <>
            <MyComponent />
            <MyComponent />
        </>
    );
}
```

Attention, cette syntaxe n'est pas supportée par tous les IDE

Délégation de contenu

Annexes

Délégation de contenu

Lors de la définition de composants réutilisable, il est possible de les construire de manière générique sans connaître leurs contenues à l'avance.

C'est par exemple le cas pour un composant de type « DialogBox », « Sidebar », ...

Dans ce cas, le composant est utiliser sous la forme d'une balise ouvrante et fermante. Afin de pouvoir définir le contenu de celui-ci (en JSX) entre ses balises.

Le contenu est récupérable en utilisant l'attribut réservé « children » sur le paramètre d'entrée (props) du composant.

Délégation de contenu - Exemple

```
import React from 'react';
import PropTypes from 'prop-types';
import style from './fancy-border.module.css';

const FancyBorder = (props) => {
    return (
        <div className={`${props.className} ${style.fancyBorder}`}>
            {props.children}
        </div>
    );
}
FancyBorder.defaultProps = {
    className: ''
}
FancyBorder.propTypes = {
    children: PropTypes.node.isRequired,
    className: PropTypes.string
}
export default FancyBorder;
```

Utilisation de l'objet « ref »

Annexes

Utilité de l'objet « ref »

L'objet « ref » peut être utilisé pour deux types de scénario :

- Accéder de manière impérative à un nœud du DOM ou à un composant React.

  Quelques cas d'usages possible :
  - Gérer le focus, la sélection du texte, ou la lecture de média.
  - Lancer des animations impératives.
  - S'interfacer avec des bibliothèques DOM tierces.

  ⚠️ Evitez d'utiliser les refs pour tout ce qui peut être fait déclarativement.

- Générer un objet persistant au sein d'un composant de type « fonction ».

Mise en place d'un acces imperatif sur un élément

Pour commencer, il est faut de créer un objet de type « ref » dans notre composant :

- Utiliser le hook de référence à l'aide de la méthode « useRef ».

Ensuite, il faut associer la "ref" créé et l'élément React à l'aide de l'attribut « ref={...} ».

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

Mise en place d'un acces imperatif sur un élément

La référence de l'élément devient accessible via l'attribut « current » de l'objet "ref".

Le type de référence récupéré change en fonction du type d'élément ciblé :

- Balise HTML : On obtient la référence du DOM de l'élément.
- Composant de classe : On a accès à l'instance du composant.

⚠️ Il ne faut pas utiliser l'attribut "ref" sur un composant de type « fonction », car ceux-ci ne possède pas d'instance !

Acces imperatif - Exemple via le hook de référence

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

const DemoRefHook = () => {

    const inputRef = useRef();

    const handleFocus = () => {
        inputRef.current.focus();
    }

    return (
        <div>
            <input ref={inputRef} type="text" />
            <button onClick={handleFocus}>Get Focus</button>
        </div>
    );
}

export default DemoRefHook;
```

Objet persistant à l'aide du hook de référence

Contrairement au composant classe qui possède des champs d'instance, les composants de type « fonction » n'ont pas de persistance de donnée pour stocker des valeurs business, car ceux-ci ne possède pas d'instance.

L'utilisation du Hook « useRef » permet de solutionner cette problématique.

L'objet renvoyé par le hook possède un attribut « current » qui est modifiable. Cette objet persistera pendant toute la durée de vie composant, cela permet donc de stocker une valeur business dans le composant.

⚠️ La modification du contenu de l'objet "ref" n'entraîne pas de rafraîchissement.

Objet persistant - Exemple d'utilisation

Dans cette exemple, une horloge qui s'actualise à l'aide d'un « setInterval ».

La valeur de l'id généré est placé dans l'objet "ref" obtenu grâce au « useRef ».

Cela permet d'arrêter l'horloge dans un événement *(en dehors du useEffect)*, où l'id n'aurait pas été disponible.

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

const DemoPersistance = () => {
    const [time, setTime] = useState(new Date());
    const timerId = useRef(null);

    useEffect(() => {
        timerId.current = setInterval(() => setTime(new Date), 500);
        return () => {
            clearInterval(timerId.current)
        }
    }, [])

    const handleStop = () => {
        clearInterval(timerId.current)
    }
    return (
        <div>
            <p>{time.toLocaleTimeString('fr-FR')}</p>
            <button onClick={handleStop}>Stop !</button>
        </div>
    )
}

export default DemoPersistance
```