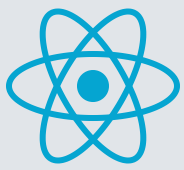

React

5. LES HOOKS

- C'est quoi ?
- useState
- useEffect

HOOKS

Les hooks sont des fonctions qui permettent de :

- rendre les **function components "intelligents"**
- offrir les mêmes capacités que les class components

Notes :

Les hooks ont été introduits avec React 16.8 sorti en février 2019 (pour React Native c'est dans la version 0.59). Depuis, leur adoption a été massive et même si l'équipe de React a tout de suite recommandé de ne pas réécrire ses composants juste pour les convertir aux hooks, beaucoup ont succombé !

En effet, les hooks apportent beaucoup d'avantages et même si leur syntaxe peut sembler un peu étrange au premier abord la courbe d'apprentissage est assez rapide.

Au delà de avantages indiqués ci-dessus, la team de React indique que la programmation fonctionnelle est plus facile à comprendre que la POO, à la fois pour les humains ET pour les machines. A mon sens, tout dépend évidemment de l'expérience qu'un développeur a avec la POO, mais ceci dit même si l'on est plus à l'aise avec la programmation objet que fonctionnelle, les hooks apportent des avantages indéniables.

Vous trouverez la documentation des hooks sur https://reactjs.org/docs/hooks-intro.html. Je vous invite à regarder la vidéo de la React conf 2018 qui est mise en avant sur cette page, la présentation de Dan Abramov est très intéressante pour comprendre en quoi les hooks permettent de rendre nos composants plus "beaux". Si vous êtes pressés (la vidéo fait quand même 1h35 !) vous pouvez passer directement à 11m30s et arrêter à la fin de la présentation de Dan (aux alentours de 1h00m00s) https://youtu.be/dpw9EHDh2bM?t=687.

HOOKS : USESTATE

ajoute un state local dans un function component

```
import {useState} from 'react';
function Gale() {
    const [isDead, setIsDead] = useState(false);
    return (
        <>
            <h1>I am a {isDead ? 'dead' : ''} chemist</h1>
            <button onClick={() => setIsDead(true)}>
                shoot Gale in the head
            </button>
        </>
    )
}
```

Notes :

Le hook useState permet de créer des fonctions stateful. A chaque modification du state généré avec useState, React relancera la fonction du composant et donc le render.

USESTATE *vs* THIS.STATE


```
class Slider extends React.Component {
    state = {
        slides: [{id: 12},{id: 42},{id: 1337}],
        currentIndex: 0
    }
    constructor(...args) {
        super(...args);
        this.handleNextClick = this.handleNextClick.bind(this);
    }
    handleNextClick() {
        const {currentIndex, slides} = this.state;
        this.setState({currentIndex:(currentIndex + 1)%slides.length});
    }
    render() {
        const {currentIndex, slides} = this.state;
        return ( <button onClick={this.handleNextClick}>
            { slides[currentIndex].id }
        </button> );
    }
}
```

```
function Slider() {
    const slides = [{id: 12},{id: 42},{id: 1337}];
    const [currentIndex, setCurrentIndex] = useState(0);

    function handleNextClick() {
        setCurrentIndex((currentIndex + 1)%slides.length);
    }

    return ( <button onClick={handleNextClick}>
        { slides[currentIndex].id }
    </button> );
}
```

HOOKS : USEEFFECT

Permet de lancer un traitement après le render

```
import {useState, useEffect} from 'react';
function Gale() {
    const [isDead, setIsDead] = useState(false);

    useEffect(() => {
        console.log('Je me lance à chaque render');
    });

    /* return ... */
}
```

```
import {useState, useEffect} from 'react';
function Gale() {
    const [isDead, setIsDead] = useState(false);

    useEffect(() => {
        console.log('Je ne me lance qu\'après le 1er render');
    }, []);

    /* return ... */
}
```

```
import {useState, useEffect} from 'react';
function Gale() {
    const [isDead, setIsDead] = useState(false);

    useEffect(() => {
        console.log('Je me lance si isDead a changé : ' + isDead);
    }, [isDead]);

    /* return ... */
}
```

Notes :

useEffect permet de déclencher des traitements (side effects) à chaque nouveau render. C'est l'équivalent du componentDidMount ET du componentDidUpdate réunis.

Le deuxième paramètre de useEffect, permet d'indiquer à React quand l'effet doit se lancer. Par défaut c'est à chaque re-render, mais si l'on passe un tableau on peut indiquer à React les dépendances du side effect. Si

l'on passe dans ce tableau une référence à une variable, la fonction ne se relancera que lorsque la valeur de cette variable changera.

NB: si l'on ne passe aucun paramètre la fonction se relancera à chaque render

NB2: si l'on passe un tableau vide, la fonction ne se lancera qu'une fois, puis plus jamais (équivalent du componentDidMount)

NB3: la fonction que l'on passe en paramètre de useEffect peut retourner une fonction de cleanup qui sera exécutée avant chaque nouveau passage dans l'effect ET au unmount du composant. Cette feature est utile lorsqu'on utilise des fonctions comme setTimeout/setInterval ou des mécanismes de connexion à une source de données temps réel.

USEEFFECT *vs* COMPONENTDID...

1.7

```
class Slider extends React.Component {
    state = {
        slides: [{id: 12},{id: 42},{id: 1337}],
        currentIndex: 0
    }
    componentDidMount() {
        fetchPhoto();
    }
    componentDidUpdate(prevProps, prevState) {
        if (this.state.currentIndex != prevState.currentIndex) {
            fetchPhoto();
        }
    }
    fetchPhoto() {
        const {currentIndex, slides} = this.state;
        api.getPhotoById(slides[currentIndex].id)
            .then( data => {
                const clone = [...slides];
                clone[currentIndex] = data;
                this.setState({slides: clone});
            });
    }
    // ... constructor, handleNextClick et render
}
```

```
function Slider() {
    const [slides, setSlides] = useState([{id: 12},{id: 42},{id: 1337}]);
    const [currentIndex, setCurrentIndex] = useState(0);

    function fetchPhoto() {
        api.getPhotoById(slides[currentIndex].id)
            .then( data => {
                const clone = [...slides];
                clone[currentIndex] = data;
                setSlides(clone);
            });
    }
    useEffect(fetchPhoto, [currentIndex] ); // ne se relancera que si
                                            //currentIndex change !
    function handleNextClick() {
        setCurrentIndex(index => (index + 1)%slides.length);
    }
    return ( <button onClick={handleNextClick}>
        { slides[currentIndex].id }
```

```
        </button> );
    }
```

Notes :

Exemple en ligne du code précédent : https://codepen.io/uidlt/pen/GRqodoM?editors=0100

HOOKS : LES AUTRES HOOKS

- useRef() : référence vers un élément DOM
- useReducer() : version avancée du useState()
- les autres hooks "officiels" : useMemo, useCallback, etc.
- hooks "maison"
- hooks fournis par des lib tierces (ex. react router)

Notes :

React fournit de base toute une série de hooks qui sont décrits dans la documentation : https://reactjs.org/docs/hooks-reference.html

Il est aussi possible de développer ses propres hooks pour simplifier ses développements : https://reactjs.org/docs/hooks-custom.html \ L'idée c'est qu'on va pouvoir externaliser certains comportements pour les réutiliser dans plusieurs composants.

Par ailleurs, beaucoup de lib externes offrent maintenant leurs propres hooks, c'est notamment le cas de React Router que l'on va étudier par la suite.