**Babel** est un outil très populaire utilisé dans l'écosystème React (et plus généralement dans le développement JavaScript) pour la transpilation du code source JavaScript moderne vers une version compatible avec les navigateurs plus anciens ou qui ne prennent pas en charge certaines fonctionnalités du langage.

La méthode **render()** est la seule méthode obligatoire dansun composant de type class dans React.

JSX: JavaScipt XML

npm: Node Package Manager

dans React il y a deux composant

* **State Component**(class, state) (composant de classe, déjà disponible dans Js et ES6)
* **UI Component** (Function, state(React Hooks°) ou **stateless Component**

le **"state"** (état) est un objet spécial qui permet à un composant de React de gérer et de stocker des données qui peuvent changer au fil du temps. Il représente l'état interne d'un composant et peut être utilisé pour influencer le rendu et le comportement du composant.

props

import React, {Component} from 'react';

import Car from './Cars';

class Mycars extends Component{

render(){

return (

<div>

<h1>Catalogue voiture</h1>

<Car color="red"></Car>

<Car color="black">Mercedes</Car>

<Car color="green">Peugeot</Car>

</div>

);

}

}

export default Mycars;

ici, les color, le Mercedes et le peugeot sont de **props.**

import React from "react";

const Car = (props) =>{

const couleurInfo = props.color ? (<p>Couleur: {props.color} </p>) : (<p>Couleur: Néant </p>);

if(props.children){

return(

<div style={{backgroundColor: 'pink', width : '400px', padding: '10px', margin: '5px auto'}}>

<p>Marque: {props.children}</p>

{ couleurInfo }

</div>

);

}

if(!props.children){

return(

<div style={{backgroundColor: 'pink', width : '400px', padding: '10px', margin: '5px auto'}}>

<p>No data!</p>

</div>

)

}

}

export default Car;

state c’est dans une classe

le "state" (état) est un concept fondamental qui fait référence à l'état interne d'un composant. Le state représente les données qui peuvent changer au fil du temps et qui affectent le rendu et le comportement d'un composant.

La classe était considérée comme un “State Component” avant l’arrivé des Hooks dans React.

State est un mot réservé, on ne peut pas l’utiliser.

dans JSX on utilise “{}” pour écrire du javascript.

**Fragment** c’est pour éviter de surcharger le DOM en react, il faut l’importer comme le suivant:

import React, {Component, Fragment} from 'react';

c’est un composant spécial qui permet d'englober un groupe d'éléments sans créer de nœud DOM supplémentaire. Il est utilisé lorsque vous avez besoin de retourner plusieurs éléments adjacents à partir d'un composant, mais vous ne voulez pas les encapsuler dans un élément parent supplémentaire.

return (

<Fragment key={index}>

<Car nom={voiture.name} color={voiture.color} year={year - voiture.year + ' ans'}/>

</Fragment>

)

Destructuring de l’objet props en name et age dans le paramètre

import React from "react";

const Users = ({name,age}) =>{ //destructuring de l'objet props dans le paramètre du fonction flèché

console.log(name,age);

return (

<div>

<p>Chanteur: </p>

</div>

)

}

export default Users;

dans la classe le props est disponible dans this.props

class Users extends React.Component {

render(){

console.log(this.props);

return (

<div>

<p>Chanteur: </p>

</div>

)

}

}

destructuring de l’objet props dans la class

class Users extends React.Component {

render(){

const {name, age} = this.props;

return (

<div>

<p>Chanteur:{name} {age} ans</p>

</div>

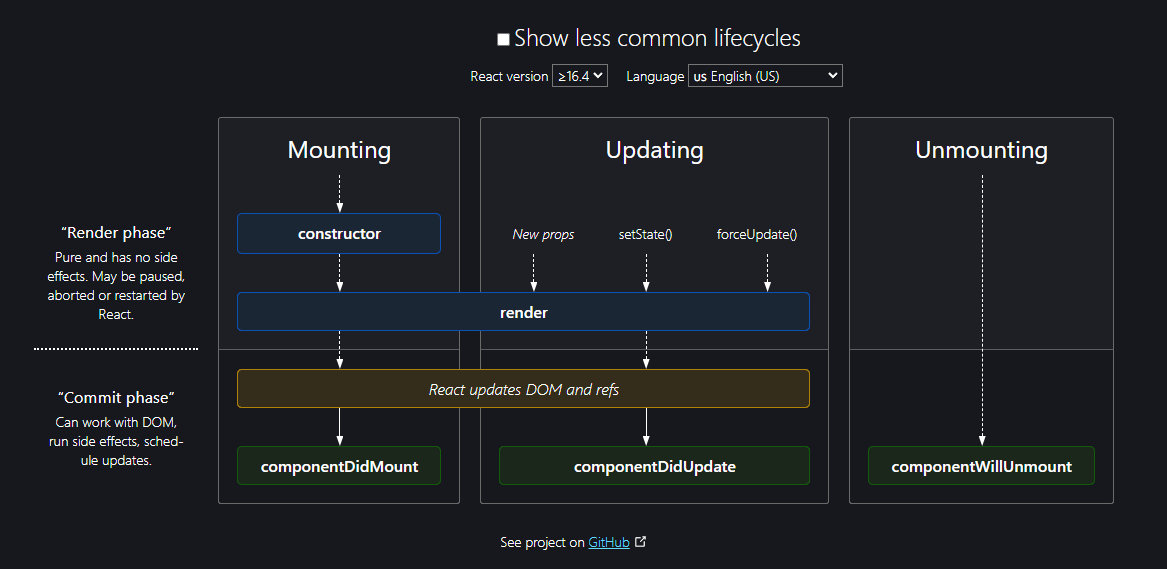
)

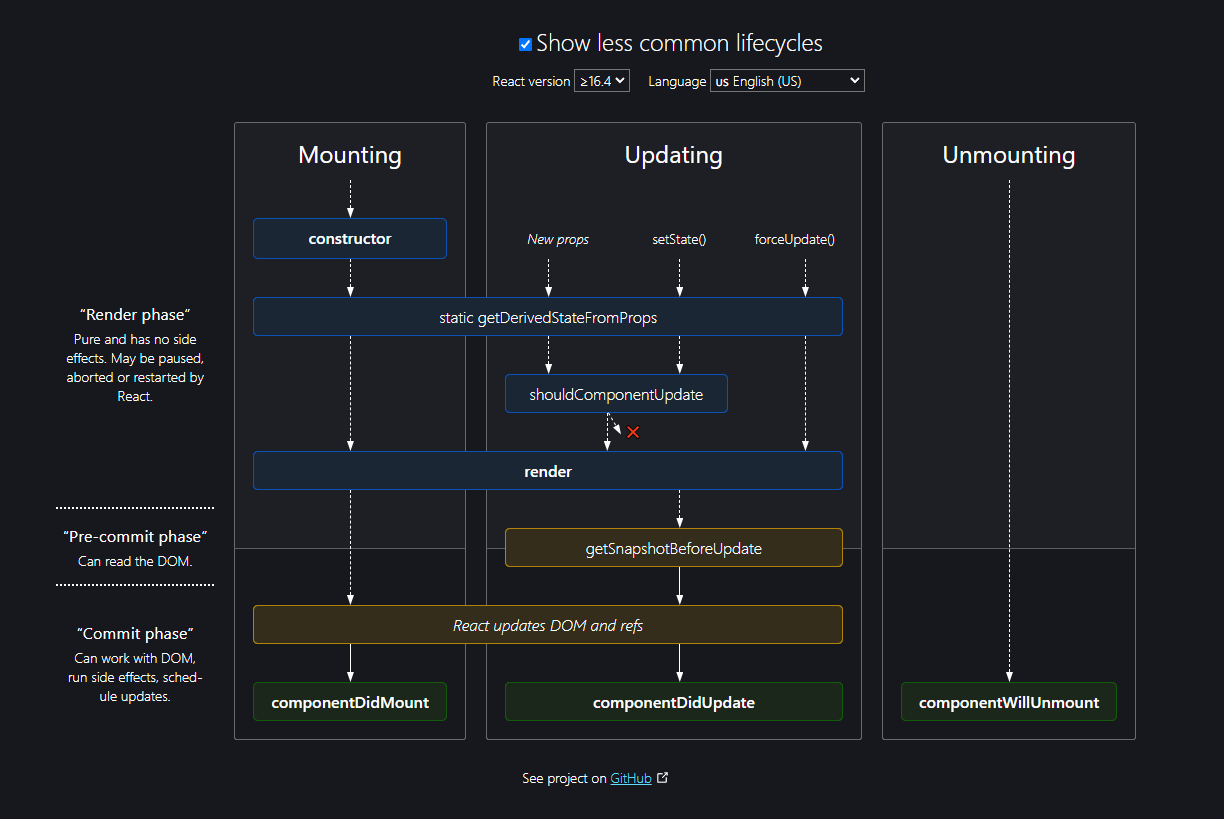
}

}

### **Life cycle components (Le cycle de vie du composant)**

Chaque composant a plusieurs « méthodes de cycle de vie » que vous pouvez surcharger pour exécuter du code à des moments précis du processus.





En JavaScript**, super()** est utilisé dans le contexte des classes et de l'héritage pour appeler le constructeur de la classe parente.

Lorsque vous créez une classe enfant qui étend une classe parente, vous pouvez utiliser super() pour appeler le constructeur de la classe parente à l'intérieur du constructeur de la classe enfant. Cela permet d'exécuter le code d'initialisation de la classe parente avant d'ajouter des fonctionnalités supplémentaires spécifiques à la classe enfant.

En JavaScript, la fonction bind() est utilisée pour créer une nouvelle fonction qui, lorsqu'elle est appelée, a un contexte lié à une valeur spécifique. Le contexte, ou "this", fait référence à l'objet auquel une fonction fait référence lorsqu'elle est exécutée.

Lorsque vous appelez la méthode bind() sur une fonction, vous pouvez spécifier la valeur à laquelle le contexte "this" doit être lié. Cela crée une nouvelle fonction liée avec le contexte fixé à la valeur fournie. Voici un exemple pour illustrer son utilisation :

const person = {

name: 'John',

greet: function() {

console.log('Hello, ' + this.name);

}

};

const sayHello = person.greet.bind(person);

const sayHelloWithoutBind = person.greet;

sayHello(); // Affiche : "Hello, John"

sayHelloWithoutBind(); // Affiche : "Hello, undefined"

**Mounting phase**

La méthode ComponentDidMount() sera invoqué seulement une seule fois après que le composant ainsi que ses enfants aurai été monté, cad chargé dans le DOM.

Dernier lifeCycle dans Mounting phase c’est la fonction **componentDidMount()**

**Updating phase**

Dernier lifeCycle dans updating phase c’est la fonction **componentDidUpdate(prevProps,prevState)**

**componentDidUpdate(prevProps, prevState){**

**console.log(`Etape ${this.state.step} :je suis dans le componentDidUpdate`);**

**console.log(prevState);//state avant la modification de state**

**console.log(this.state);//state courant**

**}**

Il ne faut pas faire un setState dans componentDidUpdate car on va tomber dans une boucle infini.

**Unmount Phase**

la fonction **componentWillUnmount()** est appelé quand un composant est retiré du DOM.  
Il est appelé immédiatement avant qu’un composant soit démonté ou détruit. Mettez-y tout nettoyage nécessaire, tel que l’invalidation de minuteurs, l’annulation de requêtes réseau ou la résilitation d’abonnements effectués dans componentDidMount()

on ne devra pas appeler **setState() dans componentWillUnmount()** car le composant ne sera de toutes façons jamais ré-affiché. Une fois l’instance du composant démontée, elle ne sera plus jamais re-montée.

**getDeriveStateFromProps()**

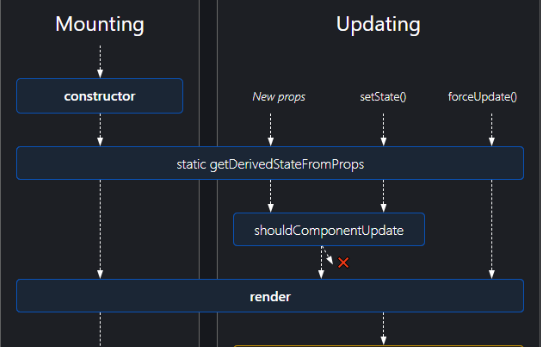
it should be static function.

il doit retourner un valeur d’objet ou null

**forceUpdate**

this.forceUpdate(callback);

**ShouldComponentUpdate()**

****

est appelé avant le rendu quand de nouvelles props ou un nouvel état local sont reçues. Par défaut, elle renvoie true.Cette méthode n’est pas appelée avant le rendu initial ou lorsque forceUpdate() est utilisé.

cette méthode n’existe qu’en tant qu’optimisation de performance.Ne vous en servez pas pour “empêcher” un refraîchissement, car ça finirait par causer des Bugs.

Utiliser alors plutôt la classe de base prédéfinie **PureComponent.**

**PureComponent**

dans le composant simple il a besoin de vérifier que le state a changer ou pas avant de retourner true or false dans shouldComponentUpdate pour effectuer le render comme la code suivante:

shouldComponentUpdate(nextProps, nextState) {

console.log('shouldComponentUpdate() décide TRUE par défaut');

if(this.state.name !== nextState.name){

return true;

}

return true;

}

mais dans PureComponent il n’est pas besoin de vérifier ça c’est automatique dans PureComponent.

shouldComponentUpdate à ne pas déclarer dans le composant PureComponent se charge d’effectuer une comparaison de surface des props et de l’état local

import React, { PureComponent } from 'react'

class PureComp extends PureComponent {

render() {

console.log('%c render() du PureComponent enfant simple','color:pink');

return (

<div>

Pure Component

</div>

)

}

}

export default PureComp;

pour que **stateless component** ou **UI component** a le même comportement comme PureComponent il faut y ajouter l’objet **React.memo()** au côté de l’éxportation du composant comme la code suivante

import React from "react";

function FunctionComp(props){

console.log('%c render() du Function Component enfant simple','color:purple');

return(

<div>

<p>

<span className="purple">FunctionComponent</span>

{props.name}

</p>

</div>

)

}

export default React.memo(FunctionComp) ;

11/06/2023

**Les portails**

Les portails fournissent une excellente solution pour afficher des composants enfants dans un noeud en dehors de la hiérachie DOM du composant

ReactDOM.createPortal(child, container)

en pratique:

function MyComponent (){

return ReactDOM.createPortal (

<div>

<p>

Je suis dans Route

</p>

</div>,

document.getElementById('second-root')

)

}

**Ref**

**<h1 ref={this.myTitle} >Valeur: {this.state.value} </h1>**

**ref** est un mot réservé comme **ClassName.**

passer un ref du composant parent au composant enfant on utilise le **forwardRef**

composant parent

class App extends Component {

constructor(props) {

super(props)

this.refComponent = React.createRef();

}

handleClick = () =>{

this.refComponent.current.focus();

}

render(){

return (

<div className="App">

<MyRefs ref={this.refComponent} name='Toto'/>

<button onClick={this.handleClick}>Valider</button>

</div>

)

}

}

Composant enfant

const MyRefs = React.forwardRef((props, ref) => {

console.log(props);

console.log(ref);

return (

<div>

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

</div>

)

})

on peut mettre un focus sur le composant enfant via le composant parents

on peut passer en paramètre de **forwardRef()** l’élément dans lequel on vas appliquer le ref.

**higherOrderComponent (Composant d’ordre supéieur)**

un composant d’ordre supérieur est une fonction qui accepte un composant et renvoie un nouveau composant

**Error Boundaries**

Les périmètres d’erreurs sont des composants React qui interceptent les erreurs JavaScript n’importe ou au sein de leur arbre de composants enfants, enregistrert ces erreurs, et affichent une UI de repli à la place de l’arbre de composants qui a planté.

import React from "react";

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = {

hasError: false

};

}

static getDerivedStateFromError(error) {

// Mettez à jour l'état, de façon à montrer l'UI de repli au prochain rendu.

return {

hasError: true

};

}

// componentDidCatch(error, errorInfo) {

// // Vous pouvez aussi enregistrer l'erreur au sein d'un service de rapport.

// logErrorToMyService(error, errorInfo);

// }

render() {

if (this.state.hasError) {

// Vous pouvez afficher n'importe quelle UI de repli.

return (

<div className="col bg-danger">

<p style={{color:'#ffffff'}}>Oups, nous avons un problème!</p>

</div>

)

}

return this.props.children;

}

}

export default ErrorBoundary;

import React, { Component } from 'react';

import Naruto from './components/Naruto';

import Pain from './components/Pain';

import Sasuke from './components/Sasuke';

import 'bootstrap/dist/css/bootstrap.min.css';

import ErrorBoundary from './components/ErrorBoundary';

import './App.css';

class App extends Component {

render() {

return (

<div className='container text-center'>

<h1>Cliquez sur les gentils</h1>

<div className='row'>

<ErrorBoundary >

<Naruto />

</ErrorBoundary>

<ErrorBoundary >

<Pain />

</ErrorBoundary>

<ErrorBoundary >

<Sasuke />

</ErrorBoundary>

</div>

</div>

)

}

}

export default App

**Props de rendu**

Le term “prop de rendu” fait référence à une technique qui consiste à partager du code entre des composants React en utilisant une prop dont la valeur est une fonction.

<div className='row'>

<AddHits

render={

(hits,addOne, uzumaki) =>(

uzumaki.Naruto && <Naruto hits={hits} addOne={addOne} name="Naruto" />

)

}

/>

<AddHits

render={

(hits,addOne, uzumaki) =>(

uzumaki.Pain && <Pain hits={hits} addOne={addOne} name="Pain"/>

)

}

/>

</div>

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

class AddHits extends Component{

state={

fighters:{

Naruto: true,

Pain: true

},

hits: 0

}

addOne = () =>{

this.setState( prevState =>{

return {

hits: prevState.hits + 1

}

}

)

}

render(){

return(

<Fragment>

{this.props.render(this.state.hits, this.addOne,this.state.fighters)}

</Fragment>

)

}

}

export default AddHits;

13/06/2023

**Single-Page Application**

Un Single-page application (SPA) est une application qui charge une unique page HTML et toutes les ressources nécessaires (telles que du javascript et des CSS) requise pour le fonctionnement de l’application. Aucune interaction sur la page ou les pages ultérieures ne nécessitera un nouvel aller-retour avec le serveur, ce qui signifie que la page n’a pas besoin d’être rafraîchie.

npm install react-router-dom

pour installer la dépendance de react-router-dom

import { Link, NavLink } from 'react-router-dom';

ceci contient le prevent default qui enlève la comportement par défaut de a href sur la chargement de la page  
au lieu de

<li className="nav-item">

<a className="nav-link active" aria-current="page" href="/community">Community</a>

</li>

c’est

<li className="nav-item">

<Link className="nav-link active" aria-current="page" to="/community">Community</Link >

</li>

dans la react-router-dom ^6   
a changé en Link

et href>to

Route & BrowserRouter

import {BrowserRouter, Route} from 'react-router-dom'

function App() {

return (

<BrowserRouter >

<Menu/>

<Route exact path="/" component={Docs} />

<Route path="/tutorial" component={Tutorials} />

<Route strict path="/community/" component={Community} />

</BrowserRouter>

);

}

20/06/2023

**Contexte**

En React, le concept de "provider" fait référence à un composant spécial qui permet de propager des données à travers l'arborescence des composants sans avoir à les passer explicitement à chaque niveau.(props à chaque niveau)

un "consumer"(c’est un concept ce n'est pas de signification spécifique liée au framework lui-même) est un composant qui utilise la fonction useContext ou le composant Consumer fourni par le contexte pour accéder aux valeurs fournies par le composant parent.  
  
21/06/2023  
MyClasse.contextType au dehors de ma classe si j’ai besoin de context et il faut l’importer, voir le code suivant:  
import React,{Component} from "react";

import naruto from './naruto.png';

import ProfileData from "./components/ProfileData";

import { MyContext } from "./components/MyContext";

const styleImage = {

width: '30%'

}

class Profile extends Component {

render(){

let value = this.context;

console.log(value)

return(

<div className="container">

<h1>{this.props.info.name}</h1>

<p>Age: {value.age} ans</p>

<img src={naruto} alt="Naruto" className="img-trumbail mb-3" style={styleImage}/>

<ProfileData welcome={this.props.info}/>

</div>

)

}

}

Profile.contextType = MyContext;

console.log(Profile.contextType)

export default Profile

MyContext.js

import React from "react";

export const UserContext = React.createContext() ;

export const ColorContext = React.createContext();

App.js

import './App.css';

import React from 'react';

import Profile from './components/Profile';

import 'bootstrap/dist/css/bootstrap.min.css';

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

class App extends React.Component {

state = {

user:{

name: 'Naruto',

age : 30

}

}

render(){

return (

<UserContext.Provider value={this.state.user}>

<ColorContext.Provider value={'red'}>

<Profile/>

</ColorContext.Provider>

</UserContext.Provider>

);

}

}

export default App;

ContentData.js

import React from "react";

import {UserContext,ColorContext} from './MyContext';

const ContentData = () =>{

return(

<UserContext.Consumer>

{

user =>{

return(

<ColorContext.Consumer>

{

color=>{

return(

<div style={{color: color}}>

<ul>

<li>Nom:{user.name}</li>

<li>Age:{user.age}</li>

</ul>

</div>

)

}

}

</ColorContext.Consumer>

)

}

}

</UserContext.Consumer>

)

}

export default ContentData

**HOOKS**

useState

import React, {useState} from "react";

const FunctionState = () =>{

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

return(

<div>

<p>Function State : {counter}</p>

<button onClick={()=>setCounter(prevCounter => prevCounter + 1)}>State dans fonction</button>

</div>

)

}

02/07/2023

**UseState (Hook d’état)**

spread oprérateur

notion destructuring

notre state dans hooks qui contient un tableau:

const[todos, setTodos] = useState([

{id: 1, todo: 'acheter du lait' },

{id: 2, todo:'acheter du pain'},

{id: 3, todo:'acheter du bière'}

])

un autre exemple de state:

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

ici le todos et addTodo c’est le nom de notre state et le 2ème dans le tableau (setTodos et setAddTodo) c’est comme le this.setState() dans UI Component ou composant de class.

avant de faire ça il faut importer   
use state

import React, {useState} from 'react';

**Use effect**

Si vous avez l'habitude des méthodes de cycle de vie des classes React, pensez au Hook useEffect comme à une combinaison de **componentDidMount**, **componentDidUpdate** et **componentWillUnMount**.  
  
voici l’exemple de code des deux component **State Component** et **StateLessComponent** de la même fonctionnalité mais l’autre a utilisé useEffect de Hook.  
le premier exemple c’est le StateComponent

import React, { Component } from 'react'

class ClassCount extends Component {

constructor(props){

super(props)

this.state = {

count: 0

}

}

componentDidMount() {

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

}

componentDidUpdate(prevProps, prevState) {

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

}

render() {

return (

<div>

<h1>

{this.state.count}

</h1>

<button onClick={()=>this.setState({count: this.state.count + 1})}>Cliquer</button>

</div>

)

}

}

export default ClassCount

l’exemple suivant c’est **StateLessComponent** et on a mis le timeOut pour voir le changement entre la fonction **componentDidMount** et **componentDidUpdate** dans useEffect car **useEffec**t rassemble les deux fonctions en général.  
Il execute à chaque render.

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

function FunctionCount(){

const[count, setCount] = useState(0)

useEffect(()=>{

setTimeout(()=>{

document.title = `vous avez cliqué ${count} fois`

},2500)

})

return(

<div>

<h1>{count}</h1>

<button onClick={()=>setCount(count + 1)}>Cliquez</button>

</div>

)

}

export default FunctionCount

**le condition dans use effect**

l’exemple suivant présente le condition dans componentDidUpdate si on veut que l’instruction s’execute si le propriété de state count est changé.

componentDidUpdate(prevProps, prevState) {

if(this.state.count !== prevState.count){

console.log('mise à jour du titre');

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

}

}

celà est identique comme dans hook useEffect suivant

useEffect(()=>{

console.log('mise à jour du titre via useEffect');

document.title = `vous avez cliqué ${count} fois`;

},[count])

dans ce useEffect on a mis un deuxième paramètre count car on veut que si count est changer? On doit l'exécuter l’instruction dans la fonction du premier paramètre.

**UseEffect Comme ComponentDidMount**voici le componentDidMount de StateComponent

componentDidMount() {

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

}

et dans useEffect on va mettre le tableau vide au deuxième paramètre

useEffect(()=>{

console.log('mise à jour du titre via useEffect');

document.title = `vous avez cliqué ${count} fois`;

},[])

**componentWillUnmount Hook**

StateComponent

componentDidMount(){

document.addEventListener("keyup", this.handleKeyCode)

}

componentWillUnmount() {

console.log('add event listener a été arrêté');

document.removeEventListener('keyup',this.handleKeyCode)

}

StateLessComponent

useEffect(()=>{

document.addEventListener('keyup', handleKeyCode);

return ()=>{

console.log('addEventListener retiré');

document.removeEventListener('keyup',handleKeyCode);

}

},[])

**Hook UseContext**

Contexte sans Hook

import React from "react";

import {UserContext,ColorContext} from './MyContext';

const ContentData = () =>{

return(

<UserContext.Consumer>

{

user =>{

return(

<ColorContext.Consumer>

{

color=>{

return(

<div style={{color: color}}>

<ul>

<li>Nom:{user.name}</li>

<li>Age:{user.age}</li>

</ul>

</div>

)

}

}

</ColorContext.Consumer>

)

}

}

</UserContext.Consumer>

)

}

export default ContentData

**contexte dans Hook**

import React,{useContext} from "react";

import {UserContext,ColorContext} from './MyContext';

const ContentData = () =>{

const user = useContext(UserContext);

const color = useContext(ColorContext);

return(

<div style={{color: color}}>

<ul>

<li>Nom:{user.name}</li>

<li>Age:{user.age}</li>

</ul>

</div>

)

}

export default ContentData

**Hook UseReducer**

En JavaScript, la méthode reduce() est une fonction intégrée aux tableaux (Array) qui permet de réduire les éléments d'un tableau à une seule valeur en appliquant une fonction de rappel sur chaque élément du tableau. Elle itère sur chaque élément du tableau, accumule le résultat intermédiaire et le renvoie finalement.

La syntaxe générale de la méthode reduce() est la suivante :

array.reduce(callback, initialValue);

Voici un exemple simple pour illustrer l'utilisation de reduce() :

const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, current) => {

return accumulator + current;

}, 0);

console.log(sum); // Affiche: 15

03/07/2023

import React,{useReducer} from 'react'

const initialState = 0;

const reducer = (state, action) =>{

switch(action){

case 'increment':

return state + 1

case 'decrement':

return state -1

case 'reinitialiser':

return initialState

default:

return state

}

}

function Count() {

//count ici c’est notre state

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

return (

<div>

<h1>{count}</h1>

<button className='btn btn-success m-3' onClick={()=>dispatch('increment')}>+</button>

<button className='btn btn-danger m-3' onClick={()=>dispatch('decrement')}>-</button>

<button className='btn btn-primary m-3' onClick={()=>dispatch('reinitialiser')}>0</button>

</div>

)

}

export default Count

04/07/2023  
**useCallBack**

**PureComponent** est pour UI component ou state component  
**React.memo** est l’équivalence de PureComponent mais il a une petite soucis au niveau de la performance.

un peu de notion de destructuring

const[countOne, setCountOne] = useState({value: 0, btnColor:'success',increment: 25});

countOne.value < 100 && setCountOne({...countOne, value: countOne.value + val})  
  
équivalent à  
  
const incrementCountOne = val => { if (countOne.value < 100) { const updatedCountOne = Object.assign({}, countOne); // Copie de l'objet countOne

updatedCountOne.value = countOne.value + val; // Mise à jour de la valeur

setCountOne(updatedCountOne); // Mise à jour de l'état avec la nouvelle version de l'objet } };

useCallBack fournissez une fonction de rapper et un tableau d’entrées. useCallBack renvera une version mémorisé de la fonction de rappel qui changera uniquement si une des entrées a changé.

const[countOne, setCountOne] = useState({value: 0, btnColor:'success',increment: 25});

const incrementCountOne = useCallback(val=>{

console.log("je suis dans incrementCountOne");

countOne.value < 100 && setCountOne({...countOne, value: countOne.value + val})

}, [countOne])

s’il y a un changement sur le state de CountOne, la fonction s’applique.

05/07/2023

**useMemo**

Fournissez une fonction de création et un tableau d’entrées. useMemo recalculer la valeur mémorisée seulement si une des entrées a changé. Cette optimisation permet d’éviter des calculs coûteux à chaque rendu.

const[count, setCount] = useState(1);

const memorizedValue = useMemo(() =>{

console.log("jesuis dans isOverThen")

return count > 10

},[count])

08/07/2023 à bira

**UseRef**

UseRef renvoie l’objet ref modifiable dont l propriété currenct est initialisée avec l’argument fourni (initialValue). L’objet renvoyé persistera pendant toute la durée de vie composant.

voici l’exemple:

import React,{useRef} from "react";

function Input(){

const usernameRef = useRef(null);

console.log(usernameRef);

const handleSubmit = () =>{

usernameRef.current.focus();

}

return(

<div>

<input ref={usernameRef} type="text"/>

<button onClick={handleSubmit}>Focus</button>

</div>

)

}

//si on clique sur ce bouton, le onFocues de l’input est activé.

export default Input;

Vous avez peut -être l’habitude d’utiliser des refs principalement pour accéder au DOM. si vous passez un objet ref à React avec <div ref={myRef}>, React calera sa propriété .current sur le noeud DOM correspondant chaque fois que ce dernier change.

ceci dit, useRef() est utile au-delà du seulattribut ref. C’est pratique pour garder des valeurs modifiables sous la main, comme lorsque vous utilisez des champs d’instance dans les classes.

Ici on n’a pas d’accès a au increment or qu’on doivent arrêter l’incrementation

useEffect(()=>{

const increment = setInterval(()=>{

setCount(prevCount =>{

return prevCount + 1;

})

},1000)

},[]);

const stopIncrement = ()=>{

clearInterval(increment)

}

useRef est notre hero ;D.

c’est current de notre useRef.

const setIntervalRef = useRef();

useEffect(()=>{

setIntervalRef.current = setInterval(()=>{

setCount(prevCount =>{

return prevCount + 1;

})

},1000)

},[]);

const stopIncrement = ()=>{

clearInterval(setIntervalRef.current)

}

Ça marche.

13/07/2023

**useImperativeHandle**

La fonction `useImperativeHandle` est un hook de React qui permet aux composants enfants d'accéder à certaines fonctions ou propriétés d'un composant parent. Elle est généralement utilisée lorsque vous avez besoin de fournir une API impérative spécifique à un composant enfant. Voici quelques cas d'utilisation courants de `useImperativeHandle` : 1. Encapsulation d'une bibliothèque tierce : Si vous utilisez une bibliothèque tierce qui nécessite une référence à un composant (par exemple, un lecteur vidéo), vous pouvez utiliser `useImperativeHandle` pour exposer des méthodes spécifiques de ce composant à partir de son parent. Cela permet à votre composant parent de contrôler le comportement du composant encapsulé. 2. Référence d

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

const FancyInput = React.forwardRef((props, ref) => {

const [count, setCount] = useState(0);

const inputRef = useRef();

React.useImperativeHandle(ref, ()=>(

{

focus: (val)=>{

if(count === 4){

const colors = ['red','blue','orange','green','pink','purple','yellow'];

const style = `4px solid ${colors[val]}`;

inputRef.current.style.border = style;

}

},

countMax: () =>{

if(count < 5){

setCount(count + 1)

}

}

}

))

return (

<div>

<h1>{count}</h1>

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

</div>

);

});

export default FancyInput;

import React, { useRef } from "react";

import FancyInput from "./components/FancyInput";

import "./App.css";

function App() {

const date = new Date();

const fancyInputRef = useRef();

const focusInput = (val) => {

fancyInputRef.current.focus(val);

fancyInputRef.current.countMax();

};

return (

<div className="App">

<FancyInput ref={fancyInputRef} />

<button onClick={() => focusInput(date.getDay())}>Cliquer</button>

</div>

);

}

export default App;

13/07/2023

**UseLayoutEffect**

La signature est identique à celle de useEffect, mais useLayoutEffect s’exécute de manière synchrone après que toutes les mutations du DOM ont eu lieu.Les mise à jour planifiées dans useLyaoutEffect seront traitées de manières synchrone avant que le navigateur ait pu procéder l’affichage.

en 90% on utilise useEffect mais au moment qu’on veut mesurer les élément du DOM par exemple.