**Documentation de React :**

Lien vers la documentation : [https://legacy.reactjs.org/docs/create-a-new-react-app.html /](https://legacy.reactjs.org/docs/create-a-new-react-app.html%20/) <https://react.dev/learn>

* Lancer le serveur : **npm start**
* Regroupe l'application dans des fichiers statiques pour la production : npm run build
* Démarre le testeur : **npm test**
* La commande de **npm run eject** : Supprime cet outil et copie les dépendances de construction, les fichiers de configuration et scripts dans le répertoire de l'application. Si vous faites cela, vous ne pouvez pas revenir en arrière.

Pour créer une application en react, on fait :

npx create-react-app my-app

cd my-app

npm start

**React render HTML**

Cette ligne se trouve dans le fichier index.html

<div id="root"></div>

On rajoute ces lignes de code dans le fichier index.js :

const header = <p>Sandy</p>

const container = ReactDOM.createRoot(document.getElementById(root));

container.render(header);

**React JSX**

JSX signifie pour Javascript XML. Il permet de rajouter du code HTML dans react. Il facilite l’écriture et l’ajout du code HTML dans react. JSX nous permet d'écrire des éléments HTML en JavaScript et de les placer dans le DOM sans aucune méthode createElement() et/ou appendChild().

JSX convertit les balises HTML en éléments de réaction.

**Component (Composants)**

Les composants sont comme des fonctions qui retourne des éléments HTML.

Les composants sont des morceaux de code indépendants et réutilisables. Elles ont le même objectif que les fonctions JavaScript, mais fonctionnent de manière isolée et renvoient du HTML. Les composants sont de deux types, les composants de classe et les composants de fonction. Dans ce didacticiel, nous nous concentrerons sur les composants de fonction.

Le nom du composant doit commencer avec une lettre majuscule.

Un composant de classe doit inclure l'instruction extend React.Component. Cette instruction crée un héritage pour React.Component et donne à votre composant l'accès aux fonctions de React.Component.

Un composant de classe doit inclure l'instruction extend React.Component. Cette instruction crée un héritage pour React.Component et donne à votre composant l'accès aux fonctions de React.Component.

Le composant nécessite également une méthode render(), cette méthode renvoie HTML. Dans ce cas, il faudra créer le composant avec une classe qui hérite de la classe **React.component** du framework React.

**Exemple d’un élément créé avec un composant :**

function Car() {

return <h2>Hi, I am a Car!</h2>;

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

Les composants peuvent être passés en tant que props, ce qui signifie propriétés.

Les propriétés sont comme des arguments de fonction, et vous les envoyez dans le composant en tant qu'attributs.

function Greet(props) {

  return <h1>Hi, {props.name} </h1>

}

const greet = ReactDOM.createRoot(document.getElementById("greet"));

greet.render(<Greet name='Ousmane' />);

NB : Si vous donnez un attribut de nom *name* dansla fonction ( component) à l’attribut props, il faudra aussi donner le même nom lors de l’appel du component dans la fonction render.

**Utilisation d’un composant avec un constructeur**

S'il y a une fonction constructeur() dans votre composant, cette fonction sera appelée lorsque le composant sera lancé.

La fonction constructeur est l'endroit où vous initiez les propriétés du composant.

Dans React, les propriétés des composants doivent être conservées dans un objet appelé état.

class Player extends React.Component {

    constructor(props) {

        super(props)

        this.state = { role: 'attaquant' }

    }

    /\* méthode utilisée pour changer l'état du status \*/

    changeRole = () => {

        this.setState({role : 'Defenseur'});

    }

    render() {

        return(

            <div>

                <h1>Mon joueur sur le terrain est: {this.state.role} </h1>

                <button type='button' onClick={this.changeRole}>Changer de role</button>

            </div>

        );

    }

}

export default Player;

Cycle de vie d’un composant : Chaque composant de React a un cycle de vie que vous pouvez surveiller et manipuler au cours de ses trois phases principales.

Les 3 phases sont : Mounting, Updating et UnMounting.

**Mounting:** signifie mettre les éléments dans le DOM.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

La méthode **getDerivedStateFromProps()** est appelée juste avant de rendre le ou les éléments dans le DOM.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

La méthode **componentDidMount()** est appelée après le rendu du composant.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

**Updating :** Un composant est mis à jour chaque fois qu'il y a un changement dans l'état ou les accessoires du composant.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

Dans la méthode **shouldComponentUpdate()**, vous pouvez renvoyer une valeur booléenne qui spécifie si React doit continuer le rendu ou non. Si la valeur booléenne est à non, il sera impossible de changer le rendu.

( Continuer avec les cycles de vie des composants sur le site de w3schools )

**React Props** : les react props sont comme des arguments passées à une fonction à Javascript.

Passer un argument en utilisant des fonctions :

function Car(props) {

return <h2>I am a { props.brand }!</h2>;

}

const myElement = <Car brand="Ford" />;

**React Event:** Tout comme les événements HTML DOM, React peut effectuer des actions basées sur des événements utilisateur.

Différence des events entre React et HTML : <button onClick={shoot}>Take the Shot!</button> React

<button onclick="shoot()">Take the Shot!</button> HTML

class Header extends React.Component {

    showAlert = (a) => {

        // alert("Show alert after clicking on button");

        alert(a)

    }

    render() {

        return (

            // <button type="button" onClick={this.showAlert('Thierno Ousmane Diallo')}>Show Alert</button>

            <button type="button" onClick={() => this.showAlert('Lucky Luck')}>Show Alert</button>

        );

    }

}

export default Header;

**Listes :**

Dans React, vous rendrez des listes avec un certain type de boucle. La méthode de tableau JavaScript map() est généralement la méthode préférée.

Afficher tous les éléments de la liste sans utiliser le mot-clé key dans la méthode map :

function Car(props) {

return <li>I am a { props.brand }</li>;

}

function Garage() {

const cars = ['Ford', 'BMW', 'Audi'];

return (

<>

<h1>Who lives in my garage?</h1>

<ul>

{cars.map((car) => <Car brand={car} />)}

</ul>

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);

Afficher tous les éléments de la liste en utilisant le mot-clé key dans la méthode map :

function Jeep(props) {

  return <li> {props.index} : {props.brand} </li>

}

function Marque() {

  const cars = [

    {id: 1, brand: 'Ford'},

    {id: 2, brand: 'BMW'},

    {id: 3, brand: 'Audi'},

    {id: 4, brand: 'Nissan'}

  ];

  return (

    <>

      <h1>The list of cars in my Garage: </h1>

      <ul>

        {cars.map((car, index) => <Jeep key={car.id} index={index} brand={car.brand} />)}

      </ul>

    </>

  );

}

const myCar = ReactDOM.createRoot(document.getElementById('marque'));

myCar.render(<Marque />);

**React Forms:** React utilise des formulaires pour permettre aux utilisateurs d'interagir avec la page Web.

function Forms() {

  const [name, setName] = useState('');

  const [textarea, setTextarea] = useState('The content of a textarea goes in the value attribute');

  const handleSubmit = (event) => {

    event.preventDefault();

    alert(`The name you entered was: ${name}`);

  }

  const handleChange = (event) => {

    setTextarea(event.target.value)

  }

  return(

      <>

      <h1>Le nom entré est: {name} </h1>

      <form onSubmit={handleSubmit}>

        <label>

          entrez votre nom:

          <input type='text'

          value={name}

          onChange={(e) => setName(e.target.value)} />

          <textarea value={textarea} onChange={(e) => handleChange(e)} />

        </label>

        <input type="submit" />

      </form>

      </>

  )

}

const myForms = ReactDOM.createRoot(document.getElementById('forms'));

myForms.render(<Forms />);

**React Select:**

function MySelect() {

  const [select, setSelect] = useState('Volvo');

  const handleSelect = (event) => {

    setSelect(event.target.value);

    console.log("select volvo");

  }

  return(

    <>

      <h1>My car: {select} </h1>

      <form value={select} onChange={handleSelect} >

        <select>

          <option value="Ford">Ford</option>

          <option value="Volvo">Volvo</option>

          <option value="Fiat">Fiat</option>

        </select>

      </form>

    </>

  )

}

const Select = ReactDOM.createRoot(document.getElementById('mySelect'));

Select.render(<MySelect />);

**Utilisation des memo en React**

L'utilisation de mémo fera que React ignorera le rendu d'un composant si ses accessoires n'ont pas changé.

**Code dans le fichier Todos.js**

import { memo } from "react";

const Todos = ({ todos }) => {

    console.log("Child render");

    return (

        <>

            <h2>My Todos</h2>

            {todos.map((todo, index) => {

                return <p key={index}> {todo} </p>

            })}

        </>

    )

}

export default memo (Todos);

**Code dans le fichier index.js**

import React, {useState} from 'react';

import ReactDOM, { createRoot } from 'react-dom/client';

import Todos from './Todos';

const App = () => {

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

  const [todos, setTodos] = useState(['Todos 1', 'Todos 2']);

  const increment = () => {

    setCount((c) => c + 1);

  }

  return(

    <>

      <Todos todos={todos} />

      <hr></hr>

      <div>

        <p>Count: {count} </p>

        <button onClick={increment}>+</button>

      </div>

    </>

  )

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

**Utilisation des Hooks en React**

Les crochets (Hooks) permettent aux composants fonctionnels d'avoir accès à l'état et à d'autres fonctionnalités de React. De ce fait, les composants de classe ne sont généralement plus nécessaires.

Les hooks nous permettent de "s'accrocher" aux fonctionnalités de React telles que les méthodes d'état et de cycle de vie.

**UseState :**

const FavoriteColor = () => {

  const[color, setColor] = useState('red');

  return (

    <>

      <h1>What is your favorite color ? </h1>

      <h1>My favorite color: {color} </h1>

      <button type='button' onClick={() => setColor('blue') }>Blue</button>

      <button type='button' onClick={() => setColor('green') }>Green</button>

      <button type='button' onClick={() => setColor('pink') }>Pink</button>

      <button type='button' onClick={() => setColor('yellow') }>Yellow</button>

    </>

  )

}

const r = ReactDOM.createRoot(document.getElementById('hooks'));

r.render(< FavoriteColor/>);

**UseEffect :**

Le crochet useEffect vous permet d'effectuer des effets secondaires dans vos composants.

Quelques exemples d'effets secondaires sont : la récupération de données, la mise à jour directe du DOM et les minuteurs.

useEffect accepte deux arguments. Le deuxième argument est facultatif. useEffect(<function>, <dependency>)

const Timer = () => {

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

  useEffect(() => {

    setTimeout(() => {

      setCount((count) => count + 1)

    }, 1000)

  })

  return (

    <>

      <h1>It's {count} times</h1>

    </>

  )

}

const r = ReactDOM.createRoot(document.getElementById('hooks'));

r.render(< Timer/>);

**UserContext** : (se renseigner bien sur ce hook )

**UseRef :** Le crochet useRef vous permet de conserver des valeurs entre les rendus.

Il peut être utilisé pour stocker une valeur modifiable qui ne provoque pas de nouveau rendu lors de la mise à jour. Il peut être utilisé pour accéder directement à un élément DOM.

useRef() ne renvoie qu'un seul élément. Il renvoie un objet appelé courant. Lorsque nous initialisons useRef, nous définissons la valeur initiale : useRef(0).

const Hooks = () => {

  const [inputValue, setInputValue] = useState("");

  const previousInputValue = useRef("");

  useEffect(() => {

    previousInputValue.current = inputValue;

  }, [inputValue]);

  return (

    <>

      <input

        type="text"

        value={inputValue}

        onChange={(e) => setInputValue(e.target.value)}

      />

      <h2>Current Value: {inputValue}</h2>

      <h2>Previous Value: {previousInputValue.current}</h2>

    </>

  );

}

UseRecducer : : (se renseigner bien sur ce hook )

**UseCallback :** Le React useCallback Hook renvoie une fonction de rappel mémorisée.

Cela nous permet d'isoler les fonctions gourmandes en ressources afin qu'elles ne s'exécutent pas automatiquement à chaque rendu.

Le crochet useCallback ne s'exécute que lorsque l'une de ses dépendances est mise à jour.

Cela peut améliorer les performances.

Les crochets useCallback et useMemo sont similaires. La principale différence est que useMemo renvoie une valeur mémorisée et useCallback renvoie une fonction mémorisée

L'une des raisons d'utiliser useCallback est d'empêcher un composant de se restituer à moins que ses accessoires n'aient changé.

Dans cet exemple, vous pourriez penser que le composant Todos ne sera pas restitué à moins que les todos ne changent :

const Hooks = () => {

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

  const [todos, setTodos] = useState([]);

  const increment = () => {

    setCount( (count) => count + 1);

  }

  const addTodo = useCallback( () => {

    setTodos( (t) => [...t, 'New Todos'])

  }, [todos] )

  return (

    <>

      <Todos todos={todos} addTodo={addTodo} />

      <hr></hr>

      <div>

        <p>Count: {count} </p>

        <button onClick={increment}>+</button>

      </div>

    </>

  )

}

Dans le fichier todos.js, on a ce bout de code :

import { memo } from "react";

const Todos = ({ todos, addTodo }) => {

    console.log("Child render");

    return (

        <>

            <h2>My Todos</h2>

            {todos.map((todo, index) => {

                return <p key={index}> {todo} </p>

            })}

            <button onClick={addTodo}>Add Todo</button>

        </>

    )

}

export default memo (Todos);

**UseMemo :**