Boubacar Demba Mandiang
ReactJS Cours ISEP - Promotion 3
Exemples de code

Manipulatiton du JSX

Lorsque nous utilisons JSX, nous devons nous rappeler de fermer toutes les balises. Si nous voulons montrer plusieurs choses à la fois, nous devons les mettre à l'intérieur d'une grande chose et montrer cela à la place. Nous pouvons utiliser quelque chose appelé un fragment (<></>) pour ce faire.
Voir les exemples ci-dessous:

```
const title = <h1>Exemple de JSX</h1>;

 const helloWorld = (
   <>
     <p>Texte</p>
     <h1>Hello world</h1>
   </>
 );

const name = "Ousmane";
const message = <p>Bonjour {name}!</p>;
```

Styliser une application avec React
Il y a plusieurs manières de faire la stylisation d'une application sur React (voir code ci-dessous)
N'oubliez pas que pour que l'application fonctionne, il existe deux fichiers importants appelés "App.css" et "App.module.css" qui doivent se trouver dans un certain dossier. S'ils ne sont pas là, vous devez les créer et les alimenter vous-même.

```
import "./App.css";
import styles from "./App.module.css";

const inlineStyle = {
  color: "blue",
  fontSize: "20px",
};

const App = () => {
  return (
    <>
      <h1>Styliser une application React</h1>
```

```
    {/* Exemple avec un objet créé dans une variable */}
    <div style={inlineStyle}>
      <h1>Hello World</h1>
    </div>

    {/* Exemple avec un objet directement provisionné */}
    <div style={{ color: "cyan", fontSize: "20px" }}>
      <h1>Hello World</h1>
    </div>

    {/* Exemple avec App.css (une classe ".hello-css" doit s'y trouver) */}
    <div className="hello-css">
      <h1>Hello World</h1>
    </div>

    {/* Exemple avec App.module.css (une classe ".hello_module_css" doit s'y trouver) */}
    <div className={styles.hello_module_css}>
      <h1>Hello World</h1>
    </div>
  </>
 );
};

export default App;
```

```
// Exemple d'un tableau d'objets sans prix
const items = [
  { id: 1, name: "Pommes" },
  { id: 2, name: "Oranges" },
  { id: 3, name: "Bananes" },
 ];

// Exemple d'un tableau d'objets avec les prix
 const items = [
  { id: 1, name: "Pommes", price:400 },
  { id: 2, name: "Oranges", price:500 },
  { id: 3, name: "Bananes", price:300 },
 ];
```

Utilisation de la fonction “map”
Notons qu’ici, on utilise une fonction map qui prend en paramètre un callback (une fonction donnée en paramètre à une autre fonction).
On boucle sur le tableau `items` et pour chaque élément de ce tableau c'est-à-dire donc pour chaque objet on aura une variable `item` avec une portée locale qui nous permettra d’accéder à toutes les clés de cet objet (ex: item.id, item.name, item.price).
`key={item.id}` ici, nous permet pour le cas spécifique de la fonction map, de gérer les états uniques pour permettre) React de différencier chaque élément créé.

```
export default function App() {
 const items = [
  { id: 1, name: "Pommes", price:400 },
  { id: 2, name: "Oranges", price:500 },
  { id: 3, name: "Bananes", price:300 },
 ];

 const list = (
  <ul>
   {items.map((item) => (
    <li key={item.id}>{item.name}</li>
   ))}
  </ul>
 );
 return <>{list}</>;
}
```

Utilisation de la fonction “find”
Il est important de noter que la fonction find retourne le premier objet du tableau d’objets qui satisfait la condition posée et pour cet exemple, la condition posée est “`item.id === 1`”

```
export default function App() {
 const items = [
  { id: 1, name: "Pommes", price:400 },
  { id: 2, name: "Oranges", price:500 },
  { id: 3, name: "Bananes", price:300 },
 ];

 const find = items.find((item) => item.id === 1);
 console.log(find);
 return <></>;
```

Utilisation de la fonction “filter”
La fonction filter nous retourne tous les éléments qui passent la vérification, c’est à dire la condition posée et pour cet exemple la condition posée est “item.price >= 400”

```
export default function App() {
  const items = [
    { id: 1, name: "Pommes", price: 400 },
    { id: 2, name: "Oranges", price: 500 },
    { id: 3, name: "Bananes", price: 300 },
  ];
  const filter = items.filter((item) => item.price >= 400);

  return (
    <div>
      <h1>Mon panier</h1>
      {filter.map((item) => {
        return (
          <p key={item.id}>
            {item.name}, {item.price}
          </p>
        );
      })}
    </div>
  );
}
```

Rendu conditionnel

Pour faire le rendu conditionnel dans le JSX, on utilise le ternaire.

Cas d'un if/else
{expression **?** ***resultat à afficher si c'est vrai*** **:** ***résultat à afficher si c'est faux***}

Cas d'un if simple
{expression **&&** ***résultat à afficher si c'est vrai***}

Bien sûr, il faut noter que le résultat à afficher peut être du JSX ou un composant

```
export default function App() {
 const isLoggedIn = false;

 return (
  <>
  {isLoggedIn ? <h1>Hello</h1> : <h1>Hello world</h1>}
  </>
 )
}

export default function App() {
 const isLoggedIn = true;
 return (
  <>
  {isLoggedIn && <h1>Hello</h1>}
  </>
 )
}
```

Manipulation des composants

À noter qu'un composant est soit une fonction ou une classe qui retourne du JSX, par convention le nom de celui-ci commence par une majuscule et doit être exporté (export default) lorsqu'on veut l'utiliser dans un autre composant.
Pour cet exemple, on a un composant interne du nom de Component et la fonction App que l'on a exporté pour permettre à un autre composant de pouvoir l'appeler..
Le premier composant retourne juste du JSX affichant "Hello".

```
function Component() {
  return <h1>Hello</h1>;
}

export default function App() {
  return (
    <div>
          {/* Première manière d'appeler un composant */}
          <Component />
          {/* Deuxième manière d'appeler un composant */}
          <Component></Component>
    </div>
  );
}
```

Les évènements

Les évènements qu'on utilise le plus souvent:

onClick, onChange, onSubmit etc...

```
export default function App() {
 const click = () => {
  console.log("Le bouton a été cliqué");
 };

 const submitForm = () => {
  console.log("Le formulaire  a été soumis");
 };

 return (
  <>
   <h1>Les évènements</h1>
   <div>
    <input
     type="text"
     placeholder="Champs"
     onChange={(e) => console.log(e.target.value)}
    />
   </div>
   <br />
   <div>
    <button onClick={click}>Clique sur moi</button>
   </div>
   <br />
   <form onSubmit={submitForm}>
    <h3>Formulaire</h3>
    <input
     type="text"
     placeholder="Texte"
     onChange={(e) => console.log(e.target.value)}
    />
    <button type="submit">Soumettre</button>
   </form>
  </>
 );
```

Props (propriétés)

Pour passer des informations d'un composant parent à un composant enfant, on utilise les props. Le transfert d'informations avec les props est donc possible du parent vers l'enfant mais le l'inverse n'est pas possible.

Les informations sont envoyées en utilisant les attributs dans la balise du composant enfant (Exemple: `<Component nom_attribut={informations_a_envoyer} />`). Ces informations seront reçues dans le composant enfant en tant que paramètres de fonction.

Pour exploiter ces paramètres, on va utiliser la déstructuration qui est une expression JavaScript qui nous permet d'extraire des données de tableaux, d'objets et de cartes et de les définir dans de nouvelles variables locales distinctes. La déstructuration nous permet d'extraire plusieurs propriétés, ou éléments, d'un tableau à la fois (voir code ci-dessous)

```
function Component({ name, passion, informations, helloWorld }) {
 const { id, title } = informations;
 return (
  <>
   <h1>{helloWorld}</h1>
   <h3>
    Titre: {id} - {title}
   </h3>
   Mon nom est {name}, {passion && <span>j'aime {passion}</span>}
  </>
 );
}

export default function App() {

 const object = { id: 1, title: "Premier objet" };
 const array = { name: "Samba", passion: "le football" };

 return (
  <>
   {/* On peut passer au composant enfant un attribut du nom de notre choix */}
   {/* On peut envoyer un tableau et faire la destructuration dans une variable dans le
composant enfant */}
   {/* On peut envoyer un objet en entier et on fait la destructuration dans les paramètres  */}
   <Component helloWorld="Hello World" informations={object} {...array} />
  </>
```

```
);
```

Les Hooks

Pour plusieurs manipulations de données sur React, on va utiliser ce qu'on appelle des "**hooks**". Ce sont des fonctions fournies par React à cet effet.

Pour les **états**, le hook qu'on utilise est "**useState**" qui nous renvoie une variable et une fonction pour mettre à jour la variable.
Pour mettre à jour l'état "**count**", ne jamais faire "**count = count + 1**", pour la mettre à jour, on utilise exclusivement la fonction qui vient avec c'est à dire la fonction "**setCount**".
Le code ci-dessous nous montre comment utiliser useState.

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

export default function App() {
 const [count, setCount] = useState(0);
 const increaseCount = () => {
  setCount(count => count+1)
 }
 return (
  <>
   <h1>Les états</h1>
   <div>{count}</div>
   <button onClick={increaseCount}>Incrementer</button>
  </>
 );
}
```

On a vu plus haut qu'on ne pouvait pas faire passer des props d'un composant enfant vers un composant parent néanmoins, on peut modifier un état qui a été déclaré dans le composant parent à partir du composant enfant. Bien sûr, il faut que ce dernier ait accès à la fonction qui modifie l'état.

Dans App.jsx, on aura le code ci-dessous

```
import { useState } from 'react';
import Component from './Component';

export default function App() {

 const [state, setState] = useState('green')
```

```
 return (
   <div>
     <Component setState={setState} />
     <div style={{backgroundColor:state,width:"50px",height:"50px"}}></div>
   </div>
 );
```

Dans Component.jsx, on aura le code ci-dessous

```
export default function Component({setState}) {

   return(
     <>
       <button onClick={() => setState('red')}>Changer la couleur</button>
     </>
     )
}
```

Lorsqu'on appuie sur le bouton, la couleur de la box changera en rouge.

Dans les composants basés sur les fonctions, on peut contrôler les étapes du cycle de vie du composant avec le hook “**useEffect**”.

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

export default function App() {
 const [count, setCount] = useState(0);
 const increaseCount = () => {
   setCount(count => count+1)
 }

 // Sans dépendance (présence du deuxième paramètre mais vide), le useEffect s'exécute une seule fois au montage du composant
 useEffect(() => {
   console.log("Le composant App a été monté");
 }, []);

 // Sans second paramètre, le useEffect s'exécute à chaque fois que le composant est rendu
 useEffect(() => {
```

```
  console.log("Le composant App a été rendu");
  // Cette fonction "return" s'exécute au démontage du composant
  return () => { console.log("Le composant App a été démonté"); }
 });

 // Avec une dépendance, le useEffect s'exécute au montage et à chaque fois que la valeur de la variable count change vu que pour cet exemple, le useEffect dépend de count
 useEffect(() => {
  console.log('Ce log s\'affiche lorsque la  valeur  de la variable count change');
 },[count]);

 return (
  <div>
   {count}
   <button onClick={increaseCount }>Incrementer</button>
  </div>
 );
}
```

Pour récupérer et exploiter les données d'un champ de texte, on peut utiliser les hooks "**useState**" ou "**useRef**".
On utilise les évènements et les références pour obtenir la valeur
Le code ci-dessous permet de faire l'illustration:

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

export default function App() {
 const [inputValue, setInputValue] = useState("");
 const inputRef = useRef();

 const handleClick = () => {
  console.log(inputRef.current.value)
  console.log(inputValue)
 }

 return (
  <>
   <input type="text" onChange={(e)=>setInputValue(e.target.value)} ref={inputRef} />
   <button onClick={handleClick}>click</button>
  </>
```

```
  );
}
```

Admettons qu'on ait 3 composants à notre disposition: Parent, Child et GrandChild
Le "props drilling" en React est une technique utilisée pour transmettre des données d'un composant à un autre en passant les données via des props à travers des composants imbriqués.
Supposons que vous ayez un composant "Parent" qui contient une donnée que vous souhaitez passer à un composant "Enfant" plus profondément imbriqué. Pour ce faire, vous devez d'abord transmettre la donnée du composant Parent au composant Enfant en passant la donnée via les props de tous les composants imbriqués entre les deux.

Par exemple, si vous avez une structure de composants comme suit :

```
<Parent>
  <Child>
    <Grandchild />
  </Child>
</Parent>
```

Et si vous voulez transmettre des données de Parent à Grandchild, vous devez d'abord transmettre les données à Child via des props, puis transmettre à nouveau les données à Grandchild via des props même si Child ne les utilise pas.
Pour remédier à cela, on peut utiliser les "createContext" et "useContext".
Le code ci-dessous permet de faire l'illustration.

```
import { useState, createContext, useContext } from "react";

const MyContext = createContext(null);

export default function App() {
  const initialColor = "gray";
  const [color, setColor] = useState(initialColor);
  return (
    <MyContext.Provider
      value={{ color: color, setColor: setColor, initialColor: initialColor }}
    >
      <Form />
    </MyContext.Provider>
  );
}
```

```
function Form() {
  return (
    <>
      <form action="">
        <Input />
        <Box />
      </form>
    </>
  );
}

function Box() {
  const { color, initialColor } = useContext(MyContext);
  return (
    <div
      style={{
        backgroundColor: color ? color : initialColor,
        height: "20px",
        width: "20px",
        marginTop: "5px;",
      }}
    ></div>
  );
}

function Input() {
  const { color, setColor } = useContext(MyContext);
  return (
    <input
      type="text"
      placeholder="Nom de la couleur"
      value={color}
      onChange={(e) => setColor(e.target.value)}
    />
  );
}
```

Les requêtes HTTP (interaction avec des API)

Les requêtes HTTP sont des messages que les clients (comme les navigateurs web) envoient aux serveurs web pour demander ou envoyer des ressources, comme des documents HTML. Elles sont à la base de tout échange de données sur le Web.
Pour demander des ressources sur React, on utilise la méthode native "**fetch**" de Javascript (on peut alternativement utiliser une bibliothèques que axios).
Le code ci-dessous illustre comment demander des ressources sur React:

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

export default function App() {
 const [data, setData] = useState([]);

 const fetchData = () => {
  console.log("Récupération des données...");
  const url = "https://www.themealdb.com/api/json/v1/1/search.php?s=";
  fetch(url)
   .then((response) => response.json())
   .then((fetchedData) => {
    setData(fetchedData);
   })
   .catch((error) => {
    console.log("Une erreur est survenue");
   });
 };
 const fetchDataAsync = async () => {
  console.log("Récupération des données...");
  const url = "https://www.themealdb.com/api/json/v1/1/search.php?s=";
  const response = await fetch(url);
  const fetchedData = await response.json();
  setData(fetchedData);
 };

 useEffect(() => {
  fetchData();
 }, []);

 console.log(data);
```

```
 return <></>;
}
```

La navigation sur React

Jusqu'ici tous les exemples qu'on a eu à présenter ont été faits sur notre composant App. Cela ne nous a posé aucun problème jusque là car on a une application relativement simple. Que faire si on a une application beaucoup plus complexe? Il faudra donc utiliser un router.

React Router est une bibliothèque qui permet de créer des applications React multipages à l'aide du routage. Le routage est la capacité d'afficher différentes pages à l'utilisateur en fonction de l'URL ou du clic sur un élément. React Router vous permet de déclarer des composants qui correspondent à des chemins (paths) spécifiques et qui s'affichent de manière conditionnelle. React Router vous donne également la possibilité de passer des paramètres entre les pages, de naviguer de manière programmée, de rediriger vers une autre page ou vers une page 404, et d'utiliser des hooks pour accéder à l'historique, aux paramètres et à la localisation.

Pour utiliser le router, il faut qu'on installe le package en tapant la commande ci-dessous sur notre cdm a la racine du projet

```
npm i react-router-dom
```

Après avoir installé le package, dans notre page "**main.jsx**" ou "**index.js**" si on utilise create-react-app, on va un peu modifier le code

Tout d'abord on va importer BrowserRouter en faisant

```
import { BrowserRouter } from "react-router-dom";
```

Puis on va entourer notre application avec

```
<BrowserRouter>
  <App />
</BrowserRouter>
```

Ensuite, on va définir nos routes sur notre page **App.jsx**

```
import { Routes, Route, useParams, useNavigate, Link } from "react-router-dom";
export default function App() {
 return (
  <>
   <Routes>
    <Route path="/" element={<h1>Accueil</h1>} />
    <Route path="/boite" element={<h1>Boite à outil</h1>} />
    <Route path="/outil" element={<Outil />} />
    <Route path="/element/:id" element={<Element />} />
    {/* Si la route n'est pas définie, on affiche cette route */}
    <Route path="*" element={<h1>404 ERREUR</h1>} />
   </Routes>
  </>
 );
}

function Outil() {
 return <h1>Outil</h1>;
}

function Element() {
 const { id } = useParams();
 const navigate = useNavigate();
 return (
  <>
   Element {id} <br />
   <a onClick={() => navigate("/")} href="javascript:void()">
    Retourner sur la page d'accueil
   </a>{" "}
   <br />
   <Link to="/">Retourner à l'accueil</Link>
  </>
 );
}
```

Pour des informations plus détaillées sur React Router, vous pouvez visiter le lien ci-dessous qui explique en détails l'utilisation de React Router.

Ultimate React Router v6 Guide